Add IPv6 support for IPFW2 and DUMMYNET for FreeBSD

Raffaele De Lorenzo, Luigi Rizzo, Mariano Tortoriello

***Abstract*—We illustrate how to build a firewall and a traffic shaper for the FreeBSD system (including Releng 4.x, 5.x, 6.x) with enhanced IPv6 filter capabilities compared to standard IPv4 capabilities. IPv6 will become soon the new standard Internet Protocol, and it differs radically from IPv4 Internet Protocol. New security policies are needed for all systems that currently use (or will use) the IPv6 protocol. As far as compatibility is concerned, the new protocol can coexist with the old one, since they can work independently. Therefore, it will be possible to move gradually from IPv4 to IPv6. The goal of this paper and related codes is the implementation of the IPv6 protocol inside existing firewall/traffic shaping programs (IPFW2/DUMMYNET) supporting only IPv4. In this way, compatibility is preserved. In the first section we describe in detail the IPFW2 Firewall and DUMMYNET Traffic Shaper, including functionality and rule structure. We also describe the technical implementation and the hook with the IPv4 FreeBSD Kernel stack. In the second section we describe the Internet Protocol Version 6 (IPv6), the main differences with respect to IPv4, and how IPv6 is included in the FreeBSD kernel (IPv6 stack). In the third section we describe our implementation aimed at making IPFW2 and DUMMYNET working with IPv6 rules. We describe in detail the hooking with the FreeBSD IPv6 stack, crucial for a correct implementation. Tests are described in the last section. On April 18th 2005 this code was committed in FreeBSD CURRENT by Brooks Davis (via Luigi Rizzo). See *http://www.freebsd.org/news/status/report-jan-2005-mar-2005.html* for more info.**

I. IPFW2 AND DUMMYNET

FreeBSD have more reliable Firewall from version 4.x:ipfw2.Ipfw2 is advanced stateful firewall, very powerful for all use.You can use ipfw with dummynet traffic shaper for completed security of your systems or more just,nearly complete, because there is no IPv6 support for ipfw and dummynet.

In FreeBSD 4.x the IPv6 entrust the IPv6 traffic to ip6fw, a firewall based on old 'ipfw', low poured them and that it doesn't have advanced features like ipfw2, for example you can think stateful rule or dynamic rule rather traffic shaping.Ipfw2 and Dummynet sources are a very nice high poured them 'c' codes and it is more flexible for changes and add on, for this reasons they are powerfoul and clean. In order to cache up our prefixed scopes, all work wassplit in for jobs:

- Adapting IPFW2 Firewall module
- Adapting DUMMYNET Traffic shaping module
- Adapting FreeBSD's IPv6 stack

Most relevants add on in Ipfw2 was introduction of new pool of rules for IPv6, based of those present ones for IPv4, with another pool of rules for IPv6 only, to take advantages of innovation for new protocol. Most relevants add on in Dummynet was introduction of hook to IPv6 stacks and some

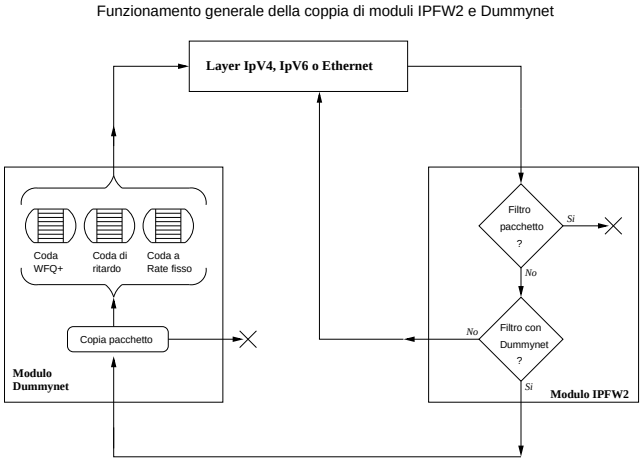

Figure 1: Package Management with Dummynet and Ipfw2

adapting structures for this scopes. Most relevants add on in user module was introduction of new parsing method for the new IPv6's rules. This rules are syntax based like IPv4 existent syntax, to preserve compatibility with all existens rulese. Most relevants add on in FreeBSD's IPv6 stack was introduction of Ipfw2's and Dummynet's hooks in the same way used for FreeBSD's IPv4 stack. Now we will describe operative functionally of Ipfw2, Dummynet and the FreeBSD's IPv6 stack (developed by Kame project). Later we will describe in detail all add on, including test phases.

II. IPFW2, THE FIREWALL

In the last years Internet was expanded in exponential mode, users that utilize it are very mutch, and the world's networking traffic is continous increase. The people need some methods for checkyng internet traffic (Firewall) and to control comunications flows (Traffic Shaper). Firewall is a packets analizer set againts between packets operative management (like fragmentation, CRC check..) and true packets management. Flows control is practically all dependent of Firewall's policies.

Ipfw2 can be splitted into two parts: the first half is operative and the second half is for control. The control part dial up with user (root) through user interface. User can also interact with Ipfw2 trough sysctl variables (see manual pages) that it control entire firewall behavior. The operative part is the core of Ipfw2 and it's the routine invocated for check-in the rules. This operation is a sequential scan of all rules in descent mode like reference their number of order from first before until

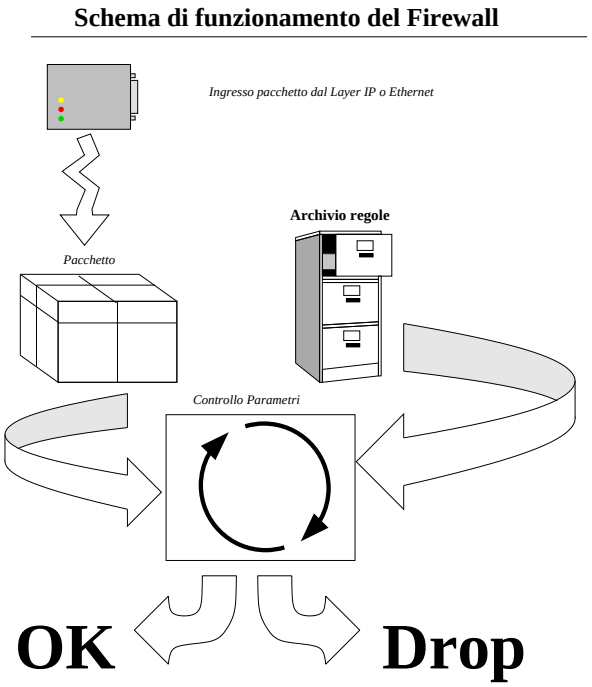

Figure 2: How Ipfw2 working

to the last rule said ̈default rule.̈ The first rule that matches with the packet arrived are applied. The sequence number of rule is automatic increased or you can insert the preferred sequence number when you define the rule. The ̈default rulëhas a sequence number 65535 and it is special, because you can't erase/modified it! this is policy compiled and it can be IPFIREWALL_DEFAULT_TO_ACCEPT understanding like to accept all packets from all or IPFIREWALL_DEFAULT_TO_DENY instead. How is Ipfw2 invocated and who call it? see the picture 3, firewall is called in some points of IP/ETHERNET stack, and the conclusion is that Ipfw2 can be called more then one time.You can understand from this picture that Ipfw2 works fine in input and in output and can protect from extern attacks but it can limit traffic requests to extern, useless to flow control inside system.

A. Structure of rules

Rules are organized in list said *struct ip_fw*. The relevant part of this structure is the body of rule, the body is structured as a set of microinstruction blocks everywone regarding some IP/TCP packet parts and besides the action policy for it. Microinstructions core is the structure *ipfw_insn* that contains the microcode which identify the istruction type, and besides it have the microcode's dimension value. This structure is very small and not adapt for complex parameters but it's very powerful and scalable. If you need some complex parameters, you must include all in the base structure and the dimension of it will be added to the dimension parameter value. Some rules were created in this sense, to have some special type for fast and comprensive programmer code. Structure of rule is:

1) action

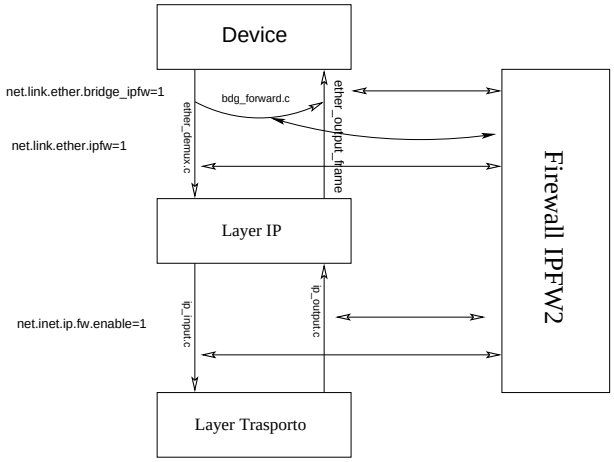

Figure 3: How ipfw2 and Dummynet are invocated

2) policy filter

Both the parts have one or more structures *ipfw_insn* followed by other parameters regarding the operation that the microinstruction encodes. The supported opcodes are listed in *enum ipfw_opcodes* and have built in the same classes reported up. Filter rule has one or more microistrucions that will be analized by firewall in some blocks for matching with the packet. Microistruction can discriminate all parts of packet, like IP header (for example source address and source destination), like IP payload and high protocol header (TCP ports,UDP ports...). Some frequently working commands have a dedicated type in order to simplify the programmer code, for example *ipfw_insn_sa* is a structure dedicated to IPv4 socket while the structure *ipfw_insn_ip* is used for IPv4 address and relative netmask. You can see how these structures follow the mentioned S̈tructure of rule b̈ecause they are build with a structure *ipfw_insn* and the type *struct sockaddr_in* (for the first example) and *struct in_addr* (for the second one). The äctionö f rule describes the real firewall action when the analized packet match with filtered commands. Tipycal actions are "DENY","ACCEPT","DIVERT","FORWARDING". The actions that divert the packet to DUMMYNET are (*O_PIPE* and *O_QUEUE*). These microinstructions have the same structure of filtered microinstructions but differ from them because the action is easier and need only structure *ipfw_insn*. The structure *ipfw_insn_cmd* in the rule is the first filter command and eventually other commands are next allocated contiguous and their dimension is defined from parameter created by build module. The same method is used for äctions,̈ but the first action's offset is a free parameter inside basic structure for fast use. You can understand the strenght-point of Ipfw2 like easier expansion of rules and easier creation of new rules (for last operation you must insert new *opcode* and new dedicated structure for it).

1) Ipfw2 Control Part: Control part is an interface that throught a comunication socket with the kernel module (pointer *ip_fw_ctl_ptr* that reference the function *ipfw_ctl*) organizes the rules. With this strument you can obtain this

information.

1) List of free rules
2) Add/Remove rules
3) Groupping rule set
4) Reset or view rule's counter

When you insert new rule, the control part makes a simple check for it that consists in measurement of the microsinstruction dimension.

2) Ipfw2 Operative part: Operative part is invocated by *ipfw_chk* function throught *ip_fw_chk_ptr* and it takes filtered operations. The hooks that call Ipfw2 are posted in some points:

1) Ethernet stack (*if_ethersubr.c* and *bridge.c*)
2) IPv4 stack (*ip_input.c* and *ip_output.c*)

The Operative part is invocated by some parameters that are stored in a structure called *args*. This structure contains the packet that will be processed and some parameters like a pointer to last rules if Ipfw2 was called before. This is important if the *sysctl* variable *net.inet.ipfw.one_pass* is setted, because make faster the checking rule process. The operative part,when it is invocated, collect some information from the packet that will be used for filter operations. Next it checks and matches all microsinstructions that build the rules and if these operations are true, actions will be runned.

III. DUMMYNET: THE TRAFFIC SHAPER

Dummynet is the module that allows to mould the IP traffic that runs thought the net interfaces Throught ipfw control part you can configure how to model the traffic throught available policies. How it works is very simple: every traffic rule can be seen as a tap that can be opened or closed by the same rule and the water flow is like the matched bandwith. Dummynet allows different kinds of date flow control. For every queue and for every parameter Dummynet can decides the modality for the traffic. the queues can be of 3 different kinds:

1) Fixed rate Queue
2) Delay Queue
3) WF2Q+ Queue

Fixed rate queue is used for setting up the bandwith permanent to a single rate. Delay queue is used for slowing down the speed of packets. These two kinds are also called *pipe*. A queue WFQ2+ (Worst-case Fair Weighted Fair Queueing) policy, which is an efficient variant of the WFQ policy. The queue associates a weight and a reference pipe to each flow, and then all backlogged (i.e., with packets queued) flow, and then all backlogged (i.e., with packets queued) portionally to their weights. Note that weights are not priorities; a flow with a lower weight is still guaranteed to orities; a flow with a lower weight is still guaranteed to higher weight is permanently backlogged. In practice, pipes can be used to set hard limits to the bandwidth that a flow can use, whereas queues can be used to determine how different flow share the available bandwidth.

If you insert some Dummnynet rules, the packet flow is relative of figure 4:

Dummynet: schema di funzionamento

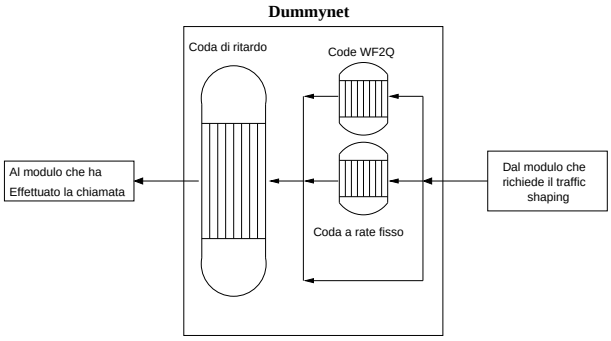

Figure 4: How Dummynet work

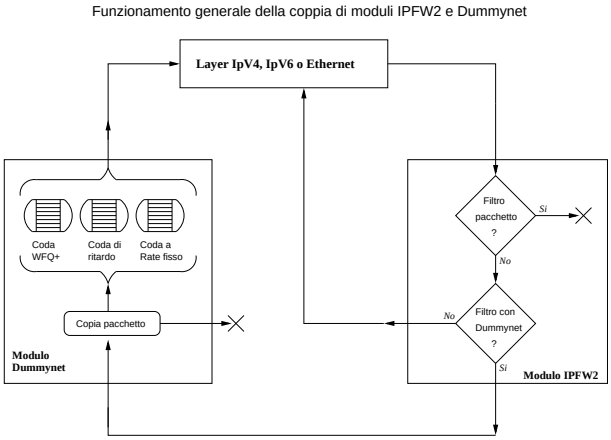

Figure 5: Packet flow with Dummynet e Ipfw2

In the same way as Ipfw2 you can split Dummynet's structure in three parts:

- Control part
- Operative part
- Interface

3) Dummynet Control Part: In the same way as Ipfw2 the control part is used to create and configure *pipe/queue* throught the function *ip_dn_ctl*. More common configuration for Dummynet are:

1) **mask**: Used for data flow control throught identification mask that is composed to parameters from TCP/IP/UDP/ICMP protocols for example.
2) **plr**: It's the packet loss ratio, and it is a probability parameter for packet loss.
3) **red/gred**: They are alghorythm for traffic flow.

For the first flow type (pipe) it corresponds somes parameters:

1) **bw**: This is the *bandwidth* assigned for the flow
2) **delay**: This is the delay for slowing the packets flow.

If you you want to configure a *queue*, you can use this parameters:

1) **pipe**: A pipe to redirect the flow for somes filters
2) **weight**: A weight used for the *fair queuing* alghoritm.

All configurations parameters were stored in the structure *dn_pipe*.

4) Dummynet operative part: The committed work by Dummynet operative part is a regular run slices determinated by kernel variable *HZ*. This variable determines therefore Dummynet's queues wake-up. The operative part runs actions to packets in queues; in fact inside all queues's structures there are some temporal policies for it. The routine *dummynet* sends get-ready packets (stored in *dn_pkt*) to interface (stored in flag *dn_dir*) that are back grabbed by *dummynet_io*, according to an alghorythm for traffic flow (**red/gred**). Next the packet (tagged by DUMMYNET first) is passed to IP/Ethernet layer and is re-inserted to Ipfw2 and according to *sysctl* variable *one_pass* is now re-filtered.

IV. Internet Protocol Version 6

IPv6 (see rfc2460) will replace the actual IPv4 protocol for the new network need (like multimedia flow traffic, and simply the end of IPv4-addresses). IPv6 use 128 bit for addresses space and moreover then *unicast* and *multicast* classes, introduct the *anycast* class. This addresses class is used for servicies that are erogated by some hosts. IPSEC have native support in IPv6. The IPv6 header is illustrated in fig. 7 (see rfc2460) then IPv4 header is illustrated in fig. 6. You can see that IPv6 header is easier then IPv4 in fact it is faster.

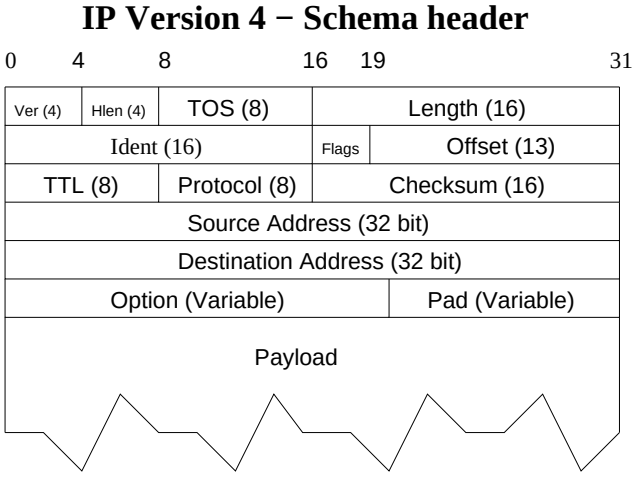

Figure 6: IPv4 Header

There are the main differences from IPv6 to IPv4.

Traffic Class: Used by router for identifying the same traffic class (priority packets). *Flow Label:* Data flow (like the same for ATM Protocol), used for real-time.

You can see that in IPv6 header was removed the *checksum* field for service reasons. The *header lenght* field was removed beacause the dimensions of header are fixed by 40 byte. The payload lenght is 16 bit fixed but this is small for higher capacity LAN; for these aims is used the *jumbogram* field. The *options* field was removed because IPv6 uses a list of extension-headers for options. The extension-headers are six

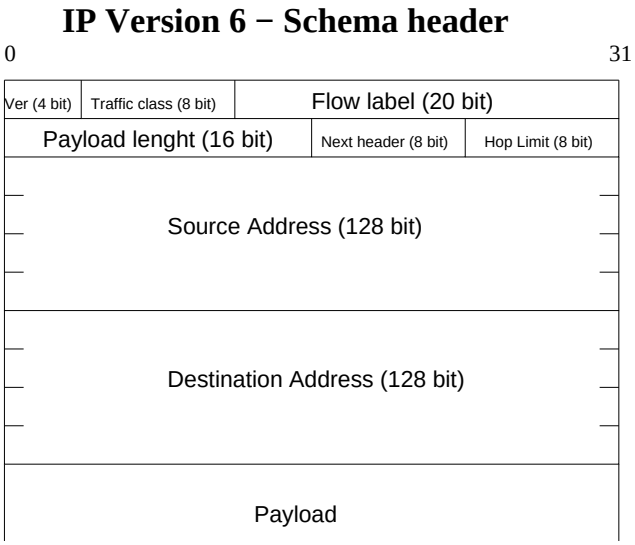

Figure 7: IPv6 Header

and are inserted next IPv6 header and the routers' forwarding are very fast because they don't analyze it (payload) see *rfc2460*. The order by IPv6 extension-header is:

- header IPv6;
- hop by hop option header
- destination option header
- routing header
- fragmentation header
- authentication header
- encrypted security payload header
- destination option header
- upper layer header (es. TCP o UDP).

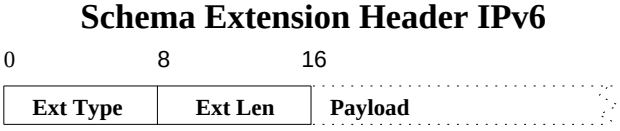

Figure 8: IPv6 extension header

The fields for fragmentation are removed, because this is made by extensions-header. In IPv6 fragmentation is not made in routers any more, and this is very fast. The true MTU for trasmission is calcolated by *MTU Discovery* procedure. In IPv6 MTU have a minimun value by 1280 byte. For compatibility IPv6 used encapsulated *tunneling* procedure for IPv4 packets.

A. IPv6 stack in FreeBSD

FreeBSD IPv6 are made by Kame group. The I/O part are controlled by *ip6_input* and *ip6_output* functions.

V. Userspace interface

A. General Description

There is a unique interface for both Dummynet and IPFW2, the main command is *ipfw*. You can see more details in the man page [2]. The main goals of the ipfw command line are:

- *rule management*: modify all rules in the ipfw2 ruleset
- *rule statistics*: perform visualization of rule status and statistics for the administrator
- *dummynet management*: add dummynet specific rule in the ruleset

The command interface is designed to create directly the structure needed by ipfw2 and send it the ipfw daemon, the syntax used is backward compatible with the ipfw1 ruleset, at least for the basic rules, bacause of the increase of capability some command were added to the interface.

The command line interpreter takes each option written by the user and create the related micro-op, it's simple to understand that the rule is created simply by acquiring the commands in writing order. This very simple way to perform packet filtering permits to develop very complex ruleset that can be interpreted ad the start time of the deamon, the ruleset can be considered as a network language that ensures the sicurity in the system. Once the interface has completed the creation of the rule it opens a socket with the control part of ipfw2 ad performs per operation sending an appropriate structure of command and rule.

The structure of a command cound be summarized as follows

ipfw ***main_command*** *[rule_body]*

the main command encodes the behaviour of the interface, some common main commands could be

- *add*/*delete*: ipfw2 rule management
- *list* or *show*: performs visualization of the firewall statistics and ruleset
- *pipe*/*queue*: dummynet management

the rule body depends on the main command but reflects the ipfw2 rule structure described above, and can be divided in:

action protocol ***from*** *source_address* ***to*** *destination_address [option]*

in the following example we could identify a first part in which we encode the action and a second part in which we store the match criteria for the IPv4 packet:

ipfw add deny ip from 192.168.0.1 to 192.168.0.20

some other action could be:

- allow/permit/accept/pass
- deny/drop

The introduction of the IPv6 caused some changes in the address interpreter, in the protocol interpreter and in the statistic output, but the global structure of the command line is mantained. The structure of the rules is mantained in both IPv4 and IPv6 environment so the flexibility and power of ipfw2 ruleset is avaible for both the tecnologies.

In addition with this simple syntax there are a lot of commands and option by which it's possible to modify the behaviour of the firewall during the ruleset analysis. For example it's possible to modify the sequence of the scanning,

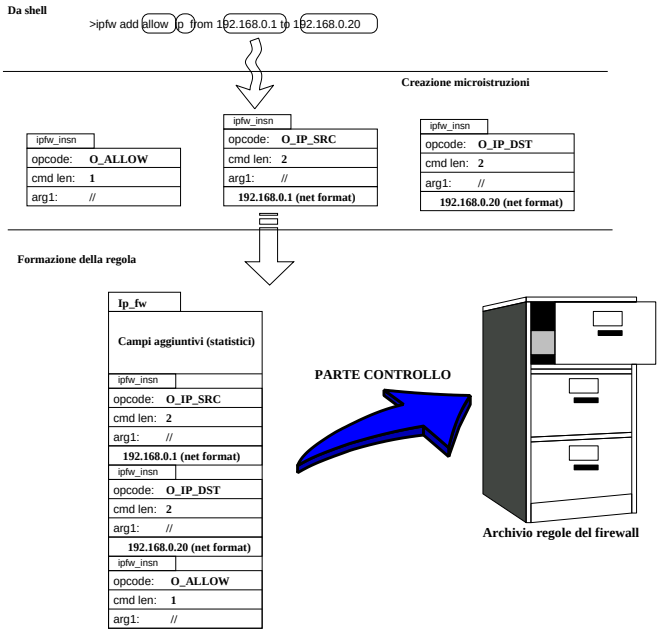

Figure 9: IPv4 Translation Rule

deactivate some rules, support additional statistics, create sono rule on the upper layer protocol i.e. TCP/UDP port. Some of those commands could be the following:

- **ipfw enable debug**
- **ipfw show**
- **ipfw pipe show**
- **ipfw flush**

The traffic shaper Dummynet uses the same basic structures and commands but has some particular feature to configure its own capabilities. Dummynet can be used only in addition with ipfw2 beacuse they share the scanning engine. We can say that dummynet is a particular action that a matching packet must follow, so even if the importance of dummynet is so rilevant that some dedicated command were developed in the ipfw2 interface the main program remains the ipfw2 core. The following example illustrate how to limit all the IPv4 bandwidth:

- **ipfw add pipe 1 ip from any to any**
- **ipfw pipe 1 config bw 30Kbit/s**

For details please see [2] and [3].

VI. Adding IPv6 Support for all

1) Add IPv6 support to Ipfw2: For adding IPv6 support to Ipfw2 and Dummynet we are interventioned so:

1) *IPFW2*: Add interception of IPv6 packets and creation of new filtered rules for it.
2) *ipfw*: Adduce user interface parser to new IPv6's opcodes.
3) *Dummynet*: Add support for IPv6 pipe/queue and moreover, support to calling the IPv6 stack.

4) *ip6_input,ip6_output*: Adding Ipfw2 hooks and Dummynet's packets management.

2) IPv6 support for IPFW2: The first intervention was the interception of packets from IPv6 layer inside the function *ipfw_chk*. In the same way of IPv4, FreeBSD store network packets in a kernel fixed array of struct **mbuf**: we has grabbed the packet from it:

```
/* Identify ipv6 packets and fill up variables. */
if (pktlen >= sizeof(struct ip6_hdr) &&
       (!args->eh ||
       ntohs(args->eh->ether_type)==ETHERTYPE_IPV6) &&
         mtod(m, struct ip *)->ip_v == 6)
```

the variable *pktlen* have the IP header's dimension grabbed from *mbuf* struct, but the *mtod(m,struct ip*)* grab the header from mbuf. You must see that management of Ethernet packets differ to management IPv4/IPv6 packets, because in the first case there is also the ethernet header. The flag *is_ipv6* was setted if IPv6 protocol was found, and next it will used for separate IPv4 code-flow to IPv6 code-flow. Next you must grab higher protocol header(ICMPv6,TCP,UDP), for collect informations from it that will used for matching rules.

```
/* Search extension headers to
   find upper layer protocols
*/
while (ulp == NULL) {
 switch (proto) {
  case IPPROTO_ICMPV6:
    PULLUP6(hlen, ulp, struct icmp6_hdr);
    args->f_id.flags = ((struct icmp6_hdr *)
   break;
.............................
```

If Ipfw2 found the fragmentation header it don't grab the higher protocol header while it don't will have all packet informations (defragmented packet). One of new rules inserted for IPv6 is the filtered from extension header. Therefore you can see the flag *ext_hd* (bit vector) used for this intention. The pointer *ulp* is used for point the header of higher protocol. Next Ipfw2 store some informations, they will used by filtered rules and Dummynet:

```
args->f_id.src_ip6 =
mtod(m, struct ip6_hdr *)->ip6_src;
args->f_id.dst_ip6 =
mtod(m, struct ip6_hdr *)->ip6_dst;
args->f_id.src_ip = 0;
args->f_id.dst_ip = 0;
rgs->f_id.flow_id6 =
ntohs(mtod(m, struct ip6_hdr *)->ip6_flow);
```

3) Add on for static rules: Now Ipfw2 check the static rules throught some *opcodes* for matching the informations grabbed from packet; if it have good match then setting the flag *match*.

```
switch (cmd->opcode) {
.........................

case O_ICMP6TYPE:
    match = is_ipv6 && offset == 0 &&
    proto==IPPROTO_ICMPV6 &&
    icmp6type_match(
((struct icmp6_hdr *)ulp)->icmp6_type,
(ipfw_insn_u32 *)cmd);
    break;
.............................
```

ICMPv6 packets was matched by this function:

```
static __inline int
icmp6type_match (int type, ipfw_insn_u32 *cmd)
{
      return (type <= ICMP6_MAXTYPE &&
      (cmd->d[type/32] & (1<<(type%32)) ) );
}


case O_IP6_SRC:
    match = is_ipv6 &&
    IN6_ARE_ADDR_EQUAL(&args->f_id.src_ip6,
    &((ipfw_insn_ip6 *)cmd)->addr6);
    break;
.........................................
```

The last two opcodes are used for filtered packet sended from/to localhost (called "me6" for distinguos to IPv4 "me"). This operation is made by the function *search_ip6_addr_net*, that it gain the local IPv6 address and match it with packet address.

```
static int
search_ip6_addr_net (struct in6_addr * ip6_addr)
{
 struct ifnet *mdc;
 struct ifaddr *mdc2;
 struct in6_ifaddr *fdm;
 struct in6_addr copia;

 TAILQ_FOREACH(mdc, &ifnet, if_link)
 for (mdc2 = mdc->if_addrlist.tqh_first; mdc2;
    mdc2 = mdc2->ifa_list.tqe_next) {
    if (!mdc2->ifa_addr)
continue;
    if (mdc2->ifa_addr->sa_family == AF_INET6) {
fdm = (struct in6_ifaddr *)mdc2;
copia = fdm->ia_addr.sin6_addr;
/* need for leaving scope_id in the sock_addr */
in6_clearscope(&copia);
if (IN6_ARE_ADDR_EQUAL(ip6_addr, &copia))
return 1;
    }
 }
return 0;
}
```

Other OPCODES....

```
case O_FLOW6ID:
    match = is_ipv6 &&
    flow6id_match(args->f_id.flow_id6,
              (ipfw_insn_u32 *) cmd);
    break;
.............................
```

The opcode O_FLOW6ID is used to filter from IPv6 packet flow_id field. This operations is made by *flow6id_match* function.

```
static int
flow6id_match( int curr_flow, ipfw_insn_u32 *cmd )
{
 int i;
 for (i=0; i <= cmd->o.arg1; ++i )
  if (curr_flow == cmd->d[i] )
   return 1;
  return 0;
}
```

4) Add on for dynamic rules: Dynamic rules are supported by IPv6 in accordyng to this changes/add on:

- Adapting *lookup_dyn_rule* function used for search the rule and eventually the expiretion time.

```
if (IS_IP6_FLOW_ID(pkt)) {
```

```
    if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
    &(q->id.src_ip6)) &&
        IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
    &(q->id.dst_ip6)) &&
        pkt->src_port == q->id.src_port &&
        pkt->dst_port == q->id.dst_port ) {
        dir = MATCH_FORWARD;
        break;
    }
    if (IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
    &(q->id.dst_ip6)) &&
        IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
    &(q->id.src_ip6)) &&
        pkt->src_port == q->id.dst_port &&
        pkt->dst_port == q->id.src_port ) {
        dir = MATCH_REVERSE;
        break;
    }
```

- Adapting *lookup_dyn_parent* function used for adding new dynamic rule.

```
    static ipfw_dyn_rule *
    lookup_dyn_parent(struct ipfw_flow_id *pkt,
        struct ip_fw *rule)
    {
     ipfw_dyn_rule *q;
     int i;
    ..............
        (is_v6 &&
         IN6_ARE_ADDR_EQUAL(&(pkt->src_ip6),
         &(q->id.src_ip6)) &&
         IN6_ARE_ADDR_EQUAL(&(pkt->dst_ip6),
         &(q->id.dst_ip6))) ||
         (!is_v6 &&
         pkt->src_ip == q->id.src_ip &&
         pkt->dst_ip == q->id.dst_ip)
        )
    ..................
```

- Adapting hash function *hash_packet*, used to store dynamic rules.

```
    static __inline int
    hash_packet(struct ipfw_flow_id *id)
    {
     u_int32_t i;
     i = IS_IP6_FLOW_ID(id) ? hash_packet6(id):
    (id->dst_ip) ^ (id->src_ip) ^
    (id->dst_port) ^ (id->src_port);
     i &= (curr_dyn_buckets - 1);
     return i;
    }


static __inline int
hash_packet6(struct ipfw_flow_id *id)
{
 u_int32_t i;
 i= (id->dst_ip6.__u6_addr.__u6_addr32[0]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[1]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[2]) ^
 (id->dst_ip6.__u6_addr.__u6_addr32[3]) ^
 (id->dst_port) ^ (id->src_port) ^ (id->flow_id6);
 i &= (curr_dyn_buckets - 1);
 return i;
}
```

5) Other changes: The *check_ipfw_struct* function is used for check the validance of opcode. We has added some new opcodes for IPv6 and they need check inside this function.

```
case O_IP6_SRC:
case O_IP6_DST:
    if (cmdlen != F_INSN_SIZE(struct in6_addr)
       + F_INSN_SIZE(ipfw_insn))
    goto bad_size;
    break;
.................................
```

Antispoof algoorithm implemented for IPv4 was replicated for IPv6 in the same way.

```
static int
verify_rev_path6(struct in6_addr *src, struct ifnet *ifp)
{
 static struct route_in6 ro;
 struct sockaddr_in6 *dst;
 dst = (struct sockaddr_in6 * )&(ro.ro_dst);
 if ( !(IN6_ARE_ADDR_EQUAL (src, &dst->sin6_addr) )) {
   bzero(dst, sizeof(*dst));
   dst->sin6_family = AF_INET6;
   dst->sin6_len = sizeof(*dst);
   dst->sin6_addr = *src;
   rtalloc_ign((struct route *)&ro,
        RTF_CLONING | RTF_PRCLONING);
 }
 if ((ro.ro_rt == NULL) || (ifp == NULL) ||
    (ro.ro_rt->rt_ifp->if_index != ifp->if_index))
  return 0;
return 1;
}
```

6) Changes for Ipfw2 header: The first change to Ipfw header (ip_fw.h) was adding the new IPv6 opcodes in *ip_fw_opcode* structure

```
O_IP6_SRC,          /* address without mask */
O_IP6_SRC_ME,       /* my addresses */
O_IP6_SRC_MASK,     /* address with the mask */
O_IP6_DST,
O_IP6_DST_ME,
O_IP6_DST_MASK,
O_FLOW6ID,   /* for flow id tag in the ipv6 pkt */
O_ICMP6TYPE, /* icmp6 packet type filtering */
O_EXT_HDR, /* filtering for ipv6 extension header */
O_IP6,
```

We has defined some codes for Extension Header, used for matching filtered rules for it.

```
/*
 * The extension header are filtered only for
 * presence using a bit vector
 * with a flag for each header.
 */
#define EXT_FRAGMENT 0x1
#define EXT_HOPOPTS 0x2
#define EXT_ROUTING 0x4
#define EXT_AH 0x8
#define EXT_ESP 0x10
```

The structure *ipfw_isn_ip6* is used for matching rules for source/destination address.

```
/* Structure for ipv6 */
typedef struct _ipfw_insn_ip6 {
 ipfw_insn o;
 struct in6_addr addr6;
 struct in6_addr mask6;
} ipfw_insn_ip6;
```

For the new ICMPv6 protocol we need new structure called *ipfw_insn_icmp6*, you can see new implementation of ICMP in *rfc2542*. The "types" of ICMPv6 codes are many more and are defined in *netinet/icmp6.h*, and now they are 203 types. A bit vector structure permit to filterer multi ICMPv6 types in the same rule.

```
#define IPFW2_ICMP6_MAXV 7
typedef struct _ipfw_insn_icmp6 {
ipfw_insn o;
uint32_t d[IPFW2_ICMP6_MAXV];
} ipfw_insn_icmp6;
```

The structure *ip_fw_flow_id* is used for distinguos IPv4 addresses to IPv6 addresses, therefore the structure *ip6_dn_args* is used for store some parameters used by Dummynet (see later).

7) IPv6 support for Dummynet: The relevant changes for Dummynet they has regarded the I/O sections, also they build new pipe/queue for IPv6 packet and they build new hash table function for IPv6 packets. The I/O section is composed by two functions:

- *trasmit_event*: It is invocated when Dummynet need to insert the packet in a queue. You can understand that this function is called also by the scheduler periodycally in according to policies. We has inserted it the calling to IPv6 stack through *ip6_input* and *ip6_output* functions.

```
static void
transmit_event(struct dn_pipe *pipe)
{
.....................................
    case DN_TO_IP6_IN:
     ip6_input((struct mbuf *)pkt) ;
     break ;
    case DN_TO_IP6_OUT:
     (void)ip6_output((struct mbuf *)pkt, NULL,
                   NULL,0 , NULL, NULL, NULL);
     rt_unref (pkt->ip6opt.ro_or.ro_rt) ;
     break ;
.....................................
```

- *dummynet_io*: This function is used to insert or create the pipes for IPv6 packets.

```
static int
dummynet_io(struct mbuf *m, int pipe_nr,
.....................................
 } else if (dir == DN_TO_IP6_OUT) {
  memcpy( &(pkt->ip6opt.ro_or),
&(fwa->dummypar.ro_or),
  sizeof(fwa->dummypar.ro_or));
  if (fwa->dummypar.ro_or.ro_rt)
  fwa->dummypar.ro_or.ro_rt->rt_refcnt++;
  if (fwa->dummypar.dst_or ==
    (struct sockaddr_in6 *) &
    (fwa->dummypar.ro_or.ro_dst));
    fwa->dummypar.dst_or =
(struct sockaddr_in6 *) &
(pkt->ip6opt.ro_or.ro_dst);
    pkt->ip6opt.dst_or =
fwa->dummypar.dst_or;
    pkt->ip6opt.flags_or =
fwa->dummypar.flags_or;
 }
 if (q->head == NULL)
.....................................
```

You can see that if you has an **output pipe** then you must store routing parameters, like entire structure *ro*, the source address, the testination address and then the interface (*ifp*) where the packet from. This is necessary because Dummynet grab the packet from stack and store it in internal queue. In according to policies Dummynet re-inserted the packet later but in this delay times the routing parameters stored in the routing table can be lost. If we don't save this parameters we will are in trouble because when Dummynet will re-insertthe the packet stack this cause kernel panic!!!

8) Dummynet - other changes: A new hash packet function was inserted for IPv6 packets, in the same way for the IPv4 function; this changes are interessed the *find_queue* function.

```
.....................................
if (is_v6) {
 APPLY_MASK(&id->dst_ip6, &fs->flow_mask.dst_ip6);
 APPLY_MASK(&id->src_ip6, &fs->flow_mask.src_ip6);
 id->flow_id6 &= fs->flow_mask.flow_id6;
 i = ((id->dst_ip6.__u6_addr.__u6_addr32[0]) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[1]) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[2]) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[3]) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[0] >> 15) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[1] >> 15) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[2] >> 15) & 0xffff)^
  ((id->dst_ip6.__u6_addr.__u6_addr32[3] >> 15) & 0xffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[0] << 1) & 0xfffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[1] << 1) & 0xfffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[2] << 1) & 0xfffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[3] << 1) & 0xfffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[0] << 16) & 0xffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[1] << 16) & 0xffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[2] << 16) & 0xffff)^
  ((id->src_ip6.__u6_addr.__u6_addr32[3] << 16) & 0xffff)^
  (id->dst_port << 1) ^ (id->src_port) ^
  (id->proto ) ^
  (id->flow_id6);
.....................................................
```

9) Dummynet - Header changes: The changes to Dummynet header was some added parameters in *dn_pkt* structure for IPv6 management.

```
struct dn_pkt {
.....................................
#define DN_TO_IP6_IN 6
#define DN_TO_IP6_OUT 7
 dn_key output_time; /* wh\en the pkt is due for delivery
*/
.....................................
 struct _ip6dn_args ip6opt; /* XXX ipv6 options */
};
```

We added the "direction" to IPv6 stack (*DN_TO_IP6_IN e DN_TO_IP6_OUT*) and moreover the *ip6opt* structure. This structure contains the routing informations saved before insert the packet in Dummynet's queue.

```
struct _ip6dn_args {
 struct route_in6 ro_or;
 int flags_or;
 struct ifnet* ifp_or,origifp_or;
 struct sockaddr_in6* dst_or;
};
```

A. Hooks to IPv6 stack

We has added in *ip6_input* and *ip6_output* some informations that permit to working fine Ipfw2 and Dummynet. The intention of this work is:

1) Intercept the IPv6 packets in input and output and next, calling Ipfw2 to make they destiny.
2) Send the packets to Dummynet's queue and re-insert they next.

1) Changes to ip6_input:

```
/* now check with the firewall ipfw2 */
if (fw_enable && IPFW_LOADED) {
.....................
  goto pass6;
  if (DUMMYNET_LOADED &&
     (i & IP_FW_PORT_DYNT_FLAG) != 0) {
/* Send packet to the appropriate pipe */
      ip_dn_io_ptr(m, i & 0xffff,
DN_TO_IP6_IN, &args);
      return;
  }
```

```
.....................
}
.............
```

When you extract mbuf structure (we remember that the mbuf structure in FreeBSD contain packet informations) you copy it in *args* structure used by Ipfw2 and calling it. The Ipfw call returned code decide if the packet will be accepted or dropped. If Dummynet was loaded and there is a corrispondent rule for the packet then *i* variable has store the number of pipe/queue for it. When Dummynet will re-insert the packet from the queue it TAG them before.

```
.................
 case PACKET_TAG_DUMMYNET:
args.rule = ((struct dn_pkt *)m)->rule;
break;
.............................
*/
 }
}
 KASSERT(m != NULL && (m->m_flags & M_PKTHDR) != 0,
    ("ip6_input: no HDR"));
 if (args.rule) { /* dummynet already filtered us */
 ip6 = mtod(m, struct ip6_hdr *);
 hlen = sizeof (struct ip6_hdr);
 goto send_after_dummynet ;
}
```

When ip6_input recieve a packet it search existent "TAG" generated by Dummynet or Ipfw2 and if this is true, forward it. You can see that the packet never don't cycle it this schema.

2) Changes to ip6_output:

```
if (fw_enable && IPFW_LOADED && !args.next_hop) {
struct sockaddr_in6 *old = dst;
args.m = m;
args.next_hop = (struct sockaddr_in *) dst;
args.oif = ifp;
off = ip_fw_chk_ptr(&args);
m = args.m;
dst = (struct sockaddr_in6 *) args.next_hop;
..............
if (DUMMYNET_LOADED &&
   (off & IP_FW_PORT_DYNT_FLAG) != 0) {
.................
 args.dummypar.ro_or = *ro;
 args.dummypar.flags_or = flags;
 args.dummypar.ifp_or = ifp;
 args.dummypar.origifp_or = origifp;
 args.dummypar.dst_or = *dst;
 args.flags = flags;
 error = ip_dn_io_ptr(m, off & 0xffff,
     DN_TO_IP6_OUT, &args);
 goto done;
}
}
pass6:
```

ip6_output save mbuf structure in *args* structure and next call Ipfw2. The Ipfw2 returned code decide the destiny of packet and moreover if Dummynet was loaded the packet flow will be the same of ip_input. Ip6_ouptut before calling Dummynet save the routing parameters of packet (ro), the network interface parameters(ifp,orig_ifp), the destination socket (st). This parameters are very important!! Whem Dummynet will re-insert the packet, it will restore the parameters, otherwise ip6_output don't can forward the packet and cause kernel panic.

```
.............................
  case PACKET_TAG_DUMMYNET:
/*
 * the packet was already tagged, so part of the
 * processing was already done, and we need to go down.
 * Get parameters from the header.
 */
   opt = NULL;
   ro = &((struct dn_pkt *)m0)->ip6opt.ro_or;
   flags = ((struct dn_pkt *)m0)->ip6opt.flags_or;
   im6o = NULL;
   origifp = ((struct dn_pkt *)m0)->ip6opt.origifp_or;
   ifp = ((struct dn_pkt *)m0)->ip6opt.ifp_or;
   dst =  &((struct dn_pkt *)m0)->ip6opt.dst_or;
   args.rule=((struct dn_pkt *)m0)->rule;
   if (args.rule != NULL)
     printf("Collecting parameters\n");
     break;
.........................
if (args.rule ) {        /* dummynet already saw us */
 ip6 = mtod(m, struct ip6_hdr *);
 hlen = sizeof (struct ip6_hdr) ;
 if (ro->ro_rt)
  ia = ifatoia6(ro->ro_rt->rt_ifa);
  bzero(&exthdrs, sizeof(exthdrs));
  ro_pmtu = ro;
  goto send_after_dummynet;
}
```

When ip6_output recieve a packet it search existent "TAG" generated by Dummynet or Ipfw2 and if this is true, first restore the routing parameters, the interface flag, the socket flag and moreover the MTU value, saved before, and then forward it. You can see that the packet never don't cycle it this schema.

B. Test phase

The tests for verify the new added features was build some rules in a network that support IPv6, and next check the response from Dummynet and Ipfw2. The tests for user interface (pratically the parser), was simplly the insertion of rules from shell done.

1) Test for IPFW2 - static and dynamic rules: The various tests consist to adding new rule for correspondenting case and next we verify it building network traffic flow in the same way. The good insertion of rule granted the successfull test for parser. Now if the counter of rule increment said that *ip6_input/output*working fine, and this this is a succesfull test for the hooks to IPv6 stack. The tests for static rule was:

1) **Test for filtering single address**: We has inserted a rule like this:
   ipfw add deny ipv6 from fe80::250:baff:fe78:5941 to me
   The shell output is:
   00100 deny ipv6 from fe80::250:baff:fe78:5941 to me6
   We has tested the rule taked some access from the machine fe80::250:baff:fe78:5941 to localhost, like telnet for TCP,ping for ICMPv6, ssh (TCP with SSL), and traceroute for UDP, and next we has verifyng that the traffic passed from other address.
2) **Test for filtering multi address**: We has inserted a rule like this:
   ipfw add deny ipv6 from fe80::250:baff:fe78:5941, fe78::250::fe78:3342 to me
   You can insert an arbitrary number of addrresses for it. The test was to be the same of the previously rule.
3) **Test for filtering address with some subnet-mask:** We has inserted a rule like this:

ipfw add deny ipv6 from fe80::250:baff:fe78:5941/80 to me

The shell output is:

00100 deny ipv6 from fe80::250:baff:fe78:5941/80 to me6

The test was to be the same of the previously rule but now we has also distinct the hosts from the subnet mask.

4) **Test for filtering me6 option:** See the previously rules.
5) **Test for filtering higher protocol (TCP/UDP/ICMPv6):** For testing TCP packet filtering we has insert a rule like this:

   ipfw add deny tcp from fe80::250:baff:fe78:5941/80 to me

   The shell output is:

   00100 deny tcp from fe80::250:baff:fe78:5941/80 to me6

   The tests was to be the same of the previously rules but in this case the ping6 (ICMPv6) and traceroute (UDP) packets sended from fe80::250:baff:fe78:5941/80 was passed, but telnet and ssh (TCP) they don't.

   For testing UDP packet filtering we has insert a rule like this:

   ipfw add deny udp from fe80::250:baff:fe78:5941/80 to me

   The tests was the same, but in this case only Traceroute (UDP) packets from the host was dropped, then the others pass.

   For testing ICMPv6 packet filtering we has insert some rules like this:

   ipfw add deny icmp6 from fe80::350:baff:fe78:5941/80 to me

   ipfw add deny icmp6 from fe80::250:baff:fe78:5941/80 to me icmp6types 16,127

   The tests was the same, but in the first case all ping6 packets from the host was dropped, then the others pass. In the last case was dropped only ping6 packet typed 16 and 127.
6) **Test for filtering IPv6 flow-id:** For testing flow-id filter we has inserted a rule like this:

   ipfw add deny ipv6 from any to any flow-id 20,30,50

   This rule block all datagram that have flow id specificated. We has developed a little program for test this functionally. The program open a UDP socket with the host and send it a datagram with the desidered flow id.
7) **Test for filtering IPv6 Extensions-header:** For testing extension-header filters we has inserted a rule like this:

   ipfw add deny ipv6 from any to any ext6hdr frag

   This rule drop all datagrams that was fragmented, you can insert some options like flow id.

The other test are for *dynamic* rules and they are from limit the number of connections TCP/UDP from/to host. So, for test this we has insert rules like this:

ipfw add allow tcp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4

ipfw add allow udp from fe80::250:baff:fe78:5941 to me setup limit src-addr 4

This rules limit the connections TCP and UDP to only 4 from the host. The tests was to be taked some TCP/UDP access from the machine fe80::250:baff:fe78:5941 to localhost, and then verify that if the number of connections was above to 4 they will be lost.

2) TEST for DUMMYNET: The tests for Dummynet verify the nice functionally for this rules:

- *Testing Pipe rules:* We has inserted a rule like this:

  ipfw add pipe 1 ipv6 from me to any

  ipfw pipe 1 config bw 30Kbit/s

  This rule create a IPv6 pipe and set flow speed to 30Kbit/s. If you ping6 the host and next you write *ipfw pipe show* you see:

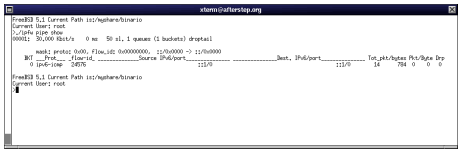

Figure 10: Pipe Status

  From the delay value of ping6 you must verify the correct flow speed.
- *Testing Queue rules:* We has inserted a rule like this:

  ipfw add pipe 1 ipv6 from me to any

  ipfw pipe 1 config bw 64Kbit/s queue 10Kbytes

  This rule create a IPv6 pipe and set flow speed to 64Kbit/s, then configure a pipe that limit traffic flow to 10KBytes. If you ping6 the host and next you write *ipfw pipe show* you see:

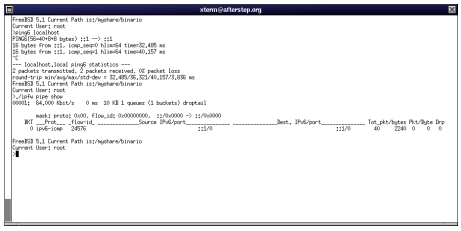

Figure 11: Queue Status

REFERENCES

[1] Paolo Valente - *Sviluppo di un sistema di Fair Queuing in ambiente Unix*, Tesi di laurea, October 2000

[2] ipfw2 manual: *man ipfw* *http://www.FreeBSD.org/cgi/man.cgi?query=ipfw*

[3] dummynet manual:*man dummynethttp://www.FreeBSD.org/cgi/man.cgi?query=dummynet*.

[4] C. Patridge. *Request for Comment 1809: Using the flow label field in IPv6*, June 1995.

[5] IAB, IESG. *Request for Comment 1881: IPv6 Address Allocation Management*, December 1995.

[6] Y. Rekhter, T. Li. *Request for Comment 1887: An Architecture for IPv6 Unicast Address Allocation*, December 1995.

[7] R. Elz. *Request for Comment 1887: A Compact Representation of IPv6 Addresses*, April 1996.

[8] R. Callon, D. Haskin. *Request for Comment 2185: Routing Aspects Of IPv6 Transition*, September 1997.

[9] S. Deering, R. Hinden. *Request for Comment 2460: Internet Protocol, Version 6 (IPv6) Specification*, December 1998.

[10] A. Conta, S. Deering. *Request for Comment 2463: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, December 1998.

[11] M. Crawford. *Request for Comment 2464: Transmission of IPv6 Packets over Ethernet Networks*, December 1998.

[12] D. Haskin, S. Onishi. *Request for Comment 2465: Management Information Base for IP Version 6: Textual Conventions and General Group*, December 1998.
[13] D. Haskin, S. Onishi. *Request for Comment 2466: Management Information Base for IP Version 6: ICMPv6 Group*, December 1998.
[14] D. Johnson, S. Deering. *Request for Comment 2526: Reserved IPv6 Subnet Anycast Addresses*, March 1999.
[15] D. Borman, S. Deering, R. Hinden. *Request for Comment 2675: IPv6 Jumbograms*, August 1999.
[16] R. Hinden, B. Carpenter, L. Masinter. *Request for Comment 2732: Format for Literal IPv6 Addresses in URL's*, December 1999.
[17] R. Gilligan, E. Nordmark. *Request for Comment 2893: Transition Mechanisms for IPv6 Hosts and Routers*, August 2000.
[18] B. Haberman, D. Thaler. *Request for Comment 3306: Unicast-Prefix-based IPv6 Multicast Addresses*, August 2002.
[19] R. Draves. *Request for Comment 3484: Default Address Selection for Internet Protocol version 6 (IPv6)*, Febrary 2003.
[20] R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens. *Request for Comment 3493: Basic Socket Interface Extensions for IPv6*, Febrary 2003.
[21] R. Hinden, S. Deering. *Request for Comment 3513: IPv6 Addressing Architecture*, April 2003.
[22] W. Stevens, M. Thomas, E. Nordmark, T. Jinmei. *Request for Comment 3542: Advanced Sockets Application Program Interface (API) for IPv6*, May 2003.
[23] R. Hinden, S. Deering, E. Nordmark. *Request for Comment 3587: IPv6 Global Unicast Address Format*, August 2003.
[24] B. Wijnen. *Request for Comment 3595: Textual Conventions for IPv6 Flow Label*, September 2003.
[25] J. Rajahalme, A. Conta, B. Carpenter, S. Deering. *Request for Comment 3697: IPv6 Flow Label Specification*, March 2004.
[26] Larry L. Peterson & Bruce S. Davie. *Computer Networks - A System Approach* Second Edition. Morgan Kaufmann Publishers (S. Francisco, California).
[27] The Kame Project - *http://www.kame.org*