
Ndmon Technical Report

for Madynes research group

September 2006

Author: **Thibault CHOLEZ**

Summary

I	General presentation	3
I.1	What is Ndmmon?	3
I.2	How does Ndmmon work?	3
I.3	Using Ndmmon	4
II	Ndmmon Implementation	6
II.1	libpcap	6
II.2	libxml2	6
II.3	Preparing packets	8
II.4	Monitoring functions	8
II.5	Sending alarms:	9
	Conclusion	10
	Related Links	10

I General presentation

I.1 What is Ndmon?

Ndmon means *Neighbor Discovery Monitor*, it's a monitoring tool designed to work with IPv6 protocol¹[1]. Its purpose is to analyse packets received from the network in order to detect an attack or a wrong configuration. Such a tool can be used by a network administrator to have better knowledge of what happens on the network. This is very usefull on a network using IPv6 because it's very easy to introduce errors in ND²[2] autoconfiguration. When it detects a suspicious ND message, it notifies the administrator by writing in the syslog and in some cases by sending an email report.

I.2 How does Ndmon work?

The first thing Ndmon has to know is what is considered to be the right network configuration. To know that, Ndmon reads a XML file named *config_ndmon.xml* by default. Here is an example of the file:

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE config_ndmon SYSTEM "config_ndmon.dtd">
3 <config_ndmon>
4   <admin_mail>cholezth@loria.fr</admin_mail>
5   <authorised_routers>
6     <mac>0:30:b6:51:d4:1c</mac>
7     <ip>fe80:0:0:0:230:b6ff:fe51:d41c</ip>
8   </authorised_routers>
9   <authorised_prefixes>
10    <prefix>2001:660:4501:1:0:0:0:0</prefix>
11  </authorised_prefixes>
12 </config_ndmon>
```

This file is small and the data have to be fill by hand, for the moment. First we can see the field *<admin_mail>* which will be used to send email to the administrator if a problem is detected. Next we have the part *<authorised_routers>* which is a list of IP and Mac addresses of the network official routers. The third information is *<authorised_prefixes>* which is simply the list of prefixes that may be announced on the link. Please just don't use the abbreviation "::" when typing an IP address because it will result in a bug for the time being.

The second file needed is built by the tool. That's a database containing for each node seen on the link the MAC and IP addresses and the time of the last ICMP[3] packet received. Ndmon can be used with an option disabling the monitoring advertisement to create this database without inappropriate warnings. When this learning phase is over, Ndmon can use the list to monitor the network.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE neighbor_list
3 SYSTEM "neighbor_list.dtd">
4 <neighbor_list>
5   <neighbor>
```

¹Internet Protocol version 6

²Neighbor Discover

```
6 <mac>0:2:a5:63:1a:66</mac>
7 <ip>fe80:0:0:0:202:a5ff:fe63:1a66</ip>
8 <time>1156781131</time>
9 </neighbor>
10 <neighbor>
11 <mac>0:4:76:ef:3:88</mac>
12 <ip>fe80:0:0:0:204:76ff:feef:388</ip>
13 <time>1156781204</time>
14 </neighbor>
15 [...]
16 </neighbor_list>
```

The general idea is to compare with specific algorithms the information get from this two XML files with the information get from the ND received packets. Ndmon structure is shown in the diagram 1.

Figure 1: Implantation de Ndmon

I.3 Using Ndmon

Ndmon doesn't need any network configuration to work. In fact, it does its work with the packets it can capture from the broadcast (link multicast) traffic, it just has to be run as root.

Concerning Ndmon deployment on a switched network, Ndmon must be run on one computer of each local network. Then, the notifications can be sent to the same email address and syslog can be redirected toward the administrator computer. That's shown in the diagram 2.

Figure 2: Déploiement de Ndmon

Ndmon should be run in the background as a demon. It doesn't use much cpu or memory, thanks to the great tools Valgrind and KCacheGrind.

II Ndmmon Implementation

Ndmmon is all written in C language. Firstly, it uses libpcap to get and filter neighbor discovery packets.

II.1 libpcap

So the first lines of the main function of Ndmmon initialize libpcap to be ready to receive packets. The pcap³ library provides many services to capture packets and is used like this:

- choice of the interface to listen: the name of the interface can be specified (ex: eth0), or let libpcap choose with the function *pcap_lookupdev()*.
- initialize the interface with the function *pcap_open_live()*
- filter the traffic to only keep a specific type of packets: to do that, a filter must be created and set with *pcap_compile()* and *pcap_setfilter()*. The filter is identified with a string expression (ex: "icmp6") to select only ICMPv6 packets for Ndmmon.
- capture packets: the functions *pcap_loop()* or *pcap_dispatch()* start capturing the packets until a specified count or an interruption.
- then libpcap loops and each time a packet passes through the filter and is captured, the *callback()* function is called in which are done all the tests. This function gives a pointer on the packet data in memory.

There is a point to take care to make the lib pcap work for Ndmmon. In fact, when using the *./configure* before installing libpcap, it has to be run with the option *-enable-ipv6*. Otherwise all filters like "ip6" or "icmp6" will not work because they are not active by default in the library. This problem may be time consuming because there is a lack of documentation about that property.

II.2 libxml2

As mentioned above, Ndmmon works with two xml files using libxml2. Most of the things using libxml are done in the file *config.c*.

The first call concerning libxml is done in the main function, before looking for packets. The function *parse_files(config_path, cache_path)* parses the two XML files and creates structures *xmlXPathContextPtr* used to execute requests. When libxml parses the files, it is careful of the file structure. They must be well written according their structure from the DTD files:

config_ndmon.dtd:

```
1 <!ELEMENT config_ndmon (admin_mail, authorised_routers, authorised_prefixes)>
2 <!ELEMENT admin_mail (#PCDATA)>
3 <!ELEMENT authorised_routers (mac*, ip*)>
4 <!ELEMENT authorised_prefixes (prefix*)>
5 <!ELEMENT ip (#PCDATA)>
```

³Packet Capture Library

```

6 <!ELEMENT mac (#PCDATA)>
7 <!ELEMENT prefix (#PCDATA)>

```

neighbor_list.dtd:

```

1 <!ELEMENT config_neighbor_list (neighbor_list)>
2 <!ELEMENT neighbor_list (neighbor*)>
3 <!ELEMENT neighbor (mac, ip, time)>
4 <!ELEMENT ip (#PCDATA)>
5 <!ELEMENT mac (#PCDATA)>
6 <!ELEMENT time (#PCDATA)>

```

When both files are parsed, the function *init_neighbor_cache()* is called from the main. It uses libcap to build a DOM tree containing the neighbor list and scan it to create a *neighbor_cache* structure in memory.

from neighborhood.h:

```

1 /*Define a neighbor on the link with its
2  *link-layer addr and IPv6 addr
3  *time is the last notification date
4  */
5 typedef struct neighbor{
6
7     struct ether_addr mac;
8     struct in6_addr ip;
9     time_t time;
10
11 }neighbor;
12
13 /*Define the neighbor cache
14  */
15 typedef struct neighbor_cache{
16
17     struct neighbor** list;
18     int nb_neighbor;
19
20 }neighbor_cache;

```

The neighbor cache is periodically saved on disk and when the application stops thanks to a handler.

When a monitoring function has to know some data from the configuration file, it can call the functions *authorised_prefixes()*, *authorised_routers_mac()* or *authorised_routers_ip()* from *config.c*. When one of these functions is called for the first time, a request is run over the parsed XML file and the result is stocked in a static structure so that it doesn't use XPath each time. Then, the next calls of the function will simply return the wanted information which type is *ip6_addr_list** or *mac_addr_list**.

```

1 /*A list of IPv6 addresses*/
2 typedef struct ip6_addr_list{
3
4     struct in6_addr** list;
5     int nb;
6
7 }ip6_addr_list;

```

Maybe the code using the DOM API of libxml2 is a bit strange regarding the tree scan. In fact DOM API has a strange behavior. When it builds the DOM tree after having done a request on the XML file, it creates some useless nodes. These nodes have a <text> type and are empty, they are in reality all the "return" pressed to layout the XML file. So, the wanted information is often the children of this empty node. A bit strange for a semantic oriented language.

II.3 Preparing packets

When a packet is captured, all the work is done in the callback function called by the lib pcap. Inside the callback function there is a pointer on the content of the packet. Then other pointers are added on the packet data to organize it inside C structures, so that the different fields and information in the packet can be easily accessed for further analysis.

The first pointer to add is on the ethernet header (*struct ether_header**) and a second one is placed on the IP header (*struct ip6_hdr**). Then there are some cases where we can't gain directly an access to the ICMP header because IPv6 protocol allows optional IP headers before the next layer. So, there is an algorithm to jump optional headers if there are. A next header value of 58 announces an ICMP packet and a *struct icmp6_hdr** points on.

Finally, the ICMP data found, a switch-case is looking for Neighbor Discovery messages and when the type of a ND message is found, it can run specific analyses. The structures used depend on the type of the message (like *struct nd_router_solicit**, *struct nd_neighbor_advert**...). All these structures are defined in different include files available in recent linux kernels. The end of the callback function is a kind of timer to periodically save the neighbor cache.

II.4 Monitoring functions

This part explains which analyses are made to detect an attack or a bad configuration on the link when a packet is received .

In the callback function and after having found the ICMP header, Ndmmon has all the needed elements to do make tests.

General analysis: A first series of analyses from *monitoring.c* is made before looking for a specific kind of ND message. The function *watch_eth_broadcast()* looks at the ethernet header to check if the source MAC address isn't a broadcast address. The same kind of test is done by the functions *watch_ip_broadcast()* and *watch_bogon()* which looks for a specific kind of source IP address. The last general test: *watch_eth_mismatch()*, checks that the address which may be specified in option of the packet is the same that the source IP address. Comparing two addresses (MAC or IP) is simply done by comparing bytes in memory.

Neighbor announced: The function *neighbor_announced* from *neighborhood.c* is called when receiving RS, RA, NS, or NA messages because these 4 messages can be used, if we follow the "should" of IPv6 protocol, to constitute the neighbor cache. First, the IP and MAC addresses are sought in the neighbor cache built by Ndmmon. If both are found, the function

watch_last_time() checks if the source has been recently seen, if true, for this address the field *time* in the neighbor database will be updated with the current time.

If neither the MAC or IP addresses are found in the neighbor cache, Ndmon adds the source of the packet as a new neighbor.

If the IP address is known but MAC address isn't, the *changed_ethernet* warning is send. The 10 last ethernet address changes are memorized by Ndmon.

```
1 typedef struct eth_change{
2
3     struct ether_addr list[2][ETH_CHANGE_SIZE];
4     int next;
5     int nb_changes;
6
7 }eth_change;
```

Then other functions are called to detect address changes like flip flop or reused old adress.

Router Solicitation/Advertisement messages: When a RA message is captured, the functions *watch_ra_mac()* and *watch_ra_ip()* check that the source MAC and IP addresses are in the configuration file as the addresses of an official router. Then *watch_ra_prefix()* checks that the prefix announced with the RA is right according the configuration file.

Neighbor Solicitation/Adverisement messages: When a node asks for neighbors in the Duplicate Address Detection context (the sender of the NS has an unspcied IP address), the wanted address is memorized by ndmon in *watch_dad()*. Then, when the next NA is received, the function *watch_dad_dos()* checks if the message answers to the pervious NS to avoid the node takibg the address. If the NA comes from a new MAC address or if the MAC and IP addresses don't match with the neighbohr cache, Ndmon raises a warning. Another function checks if the NA doesn't set the flag router if it isn't.

Redirection message: The function *watch_R_flag()* checks that the sender is an official router.

II.5 Sending alarms:

Reports sending is done in the file *alarm.c*. It can be disable with the function *set_alarm()*. The function *notify()* allows two levels of warning. To avoid multiple warnings for the same problem (cause several wrong packets are sent) the function *already_send()* makes a little history of the previous warnings. Finally, the email is sent by another program which must be previously configured (linux mail command).

Conclusion

Ndmon is a first step to have a better control over networks using ipv6 protocol. Regarding the different tests done, it works fine. As future works, it can be improved by adding new detection rules. The source code and organisation of Ndmon should make it quite easy to do. Ndmon should also be portable to other operating systems because the C libraries libpcap and libxml2 are, there is just to include the header files used from 2.6 kernel.

References

- [1] IPv6 : <http://www.ietf.org/rfc/rfc2460.txt>
- [2] Neighbor Discovery : <http://www.ietf.org/rfc/rfc2461.txt>
- [3] ICMPv6 : <http://www.ietf.org/rfc/rfc2463.txt>
- [4] Jari Arkko, Tuomas Aura, James Kempf, Vesa-Matti Mantyla, Pekka Nikander, Michael Roe, *Securing IPv6 Neighbor and Router Discovery*
- [5] P. Nikander, J. Kempf, E. Nordmark, *IPv6 Neighbor Discovery Trust Models and Threats*, May 2004
- [6] Tim Carstens, *Programming with pcap* <http://www.tcpdump.org/pcap.htm>
- [7] Packet Capture With libpcap and other Low Level Network Tricks <http://www.cet.nau.edu/~mc8/Socket/Tutorials/section1.html>
- [8] Libxml : The XML C parser and toolkit of Gnome <http://www.xmlsoft.org>
- [9] Yves Mettier, *Briques en C: libxml2 et petits fichiers XML* http://ymettier.free.fr/articles_lmag/lmag51_briques_en_C17/lmag51_briques_en_C17.html
- [10] Peter Bieringer, HOWTO IPv6 Linux (fr) <http://mirrors.bieringer.de/Linux+IPv6-HOWTO/>
- [11] Gisèle Cizault, *IPv6 théorie et pratique*, O'Reilly, 2005 <http://livre.point6.net/index.php/Accueil>