UNIVERSITY OF LIEGE
Faculty of Applied Sciences
Montéfiore Electricity Institute

Extension of a network scanning tool with IPv6 features (Nmap)

End of studies dissertation by
Sebastien Peterson
in order to achieve the title of
Civil Engineer in Computer Science
Academic year 2001-2002

UNIVERSITY OF LIEGE
Faculty of Applied Sciences
Montéfiore Electricity Institute

Extension of a network scanning tool with IPv6 features (Nmap)

End of studies dissertation by
Sebastien Peterson
Avenue du Centenaire, 27
4053 Embourg Belgium
e-mail : seb.peterson@easynet.be
gsm : 32 (0)477 653748

Acknowledgments

I would like to thank Mister Guy Leduc for giving me the opportunity to achieve my end of studies dissertation in the computer networks domain. During my years of study in the Montéfiore Institute, his teachings and passion for networking have intensified my interests in this domain.

I am also thankful to Mister Emmanuel Tychon for his support, and for providing this really interesting subject, which led me to discover a lot about network technologies and programming.

Thanks to François Blanchy, who's judicious explanation helped me in deepening my knowledge of the Linux Environment.

Index

Chapter 1

Introduction

With the appearance of computer networking technologies came the possibility for distant hosts to share information. Until then, the only way to retrieve information from one computer and use it in another was to save the information on tape or disk and bring it back to be used on the other computer. With this way of sharing information there wasn't much risk of someone intruding the system, since only the user had access to the computer. Today, networks give us the opportunity to share that information much more efficiently. Local networks have given the possibility to develop the way computers could be used in the industry, helping communication and information sharing. And with the global development of the computers and telecommunication technologies, networks aren't restricted any more to the industrial world, but have conquered all the layers of society. The Internet is the best example of global network integration.

Being able to share information so easily is of course a very large technological improvement, but it's also a new opportunity for intruders to get information you don't necessarily want to share. It is obvious, for economical or just for confidential reasons, that security has become an inevitable issue.

Administrators can provide security to their network with good knowledge of network technologies and security gaps (this is why hackers make very good security consultants...). But today, not just professionals have to bother securing their networks. With the apparition of permanent internet connections (cable or ADSL lines), individuals should also be aware that their private networks could be the targets (or source) of diverse attacks by hackers.

With the help of diverse security applications, we can get information on the security of our networks. Nmap ("Network Mapper") is a well known open source tool, designed for network exploring and security auditing. It was developed by Fyodor (Fyodor@insecure.org) and new versions come out regularly, thanks to the help of the Internet community which reports bugs and proposes improvements. Nmap is recognized to be one of the best security products (for press references see "http://www.insecure.org/nmap/nmap_inthenews.html"). Nmap is a very powerful tool, but for the moment, it only supports the current Internet Protocol, namely IPv4.

The next generation IP protocol, IPv6, is now waiting to take over from IPv4. Not only IPv6 will supplant the lack of addressing space of IPv4, but it will also bring numerous technological improvements.

The purpose of my work is to bring IPv6 capabilities to Nmap. Knowing that a lot of requests have been done on the porting of nmap to IPv6, and knowing that the work has not been undertaken yet, it's a thrilling experience to work on this project that will make me discover the new IPv6 protocol, and that will hopefully be useful to the open source community.

Here's how my work will be organized:

- My work will start with discovering the Nmap source code and with understanding the program and its scan techniques.
- Besides this, I will acquire knowledge on the IPv6 protocol as well on its implementation in the Linux environment.
- After that, the first goal is to port a simple scan to IPv6. This part will require to change the parsing of the new IPv6 addresses by Nmap. It will also lead to discover the IPv6 API.
- Then I will go on porting the more advanced features of nmap, such as "stealth" scanning, with the use of raw packets.
- After porting nmap to IPv6, the changes will be submitted to Fyodor and his team.

Chapter 2

Nmap presentation

2.1 Nmap description

Nmap is an open source security tool designed to scan hosts on large networks. The primary function of Nmap is to determine if the hosts are reachable and which ones of their ports are open. Knowing that, we can tell the services available on the hosts. So Nmap can be very useful to network administrators since they can easily monitor all the hosts on their network. Nmap quickly finds which hosts are "up" or "down". The administrator will also be able to get information on the security of the hosts on his network, by knowing the open ports and services. Of course Nmap is also very useful to hackers; it is a very strong tool to find security gaps left by open ports and can inform the hacker on the presence of a firewall and which ports are filtered. Nmap can be very fast, and scan very large networks.

Nmap supports different types of scanning techniques: TCP/UDP connect(), TCP SYN, ICMP ping sweep, FIN, ACK sweep, Xmas Tree, FTP proxy bounce attack, reverse-ident. All those types of scans implement different techniques to suite the user's needs; these techniques go from simple scan's to more advanced techniques to get through different firewall configurations.

Nmap provides advanced features as Operating System detection, stealth scanning, parallel scanning, decoy scanning, port-filtering detection, ...

Nmap has a powerful target notation. Besides specifying one target host, you can specify several hosts or even, several subnets. The subnets are specified by using mask notation (for example: 212.123.45.78/24) or asterisks notation (192.168.*.* , this will scan from 192.168.0.0 up to 192.168.255.255).

2.2 Different types of scans [Fyod]

2.2.1 Tcp connect() scan

This is the most simple type of scan. It tries a TCP "connect()" system call on all the desired ports on the selected host. If a connection succeeds, it means that the port was open and listening. In the case of a connection refusal, the port is closed. The scanning speed can go up by opening connections in parallel (useful when scanning large networks). An advantage of this technique is that it doesn't require "root" privileges.

The problem with this type of scan is that when you quickly scan a large number of ports on a host, the targeted system will easily detect that it is being scanned (by logging all the connection attempts) and will stop any further connection attempt.

The purpose of most following scans is to avoid this detection problem by using different techniques.

2.2.2 Syn scan

The trick here is that you should not use the "connect()" call, which deals with all the TCP connection process. But instead, the SYN scan sends a raw packet with the SYN flag set. This raw packet is in fact a simple TCP/IP packet built "bit by bit" with the SYN flag enabled. This packet is the same as the first packet sent by the "connect()" call.

In fact we act as if we were going to open a TCP connection. If the port is closed, the target sends back a RST packet. If the port is listening, it sends a SYN/ACK packet back, then nmap sends a RST packet to stop the connection process. At this time the connection has not been "open" (we can talk about "half-open connection") and on most systems not been logged, and thus the scan isn't detected.

Unfortunately, this feature requires "root" privileges.

2.2.3 Ack scan

This scan is used to detect if ports on a host are filtered by a firewall. The technique is to send an ACK packet to the target host. If no firewall filters the port, a RST packet should be sent back. Otherwise when filtered, the firewall usually ignores the packet and doesn't send a RST packet back.

2.2.4 Window scan

This scan is similar to the ACK scan, except that it can sometimes detect open ports as well as filtered/non-filtered due to an anomaly in the TCP window size reporting by some operating systems, such as some versions of AIX, Amiga, BeOS, BSDI, Cray, Tru64 UNIX, DG/UX, OpenVMS, Digital UNIX, FreeBSD, HP-UX, OS/2, IRIX, MacOS, NetBSD, OpenBSD, OpenStep, QNX, Rhapsody, SunOS 4.X, Ultrix, VAX, and VxWorks.

2.2.5 Fin, Xmas, Null scans

A difficulty that Nmap has to go by is the presence of firewalls. Some packet filters and firewalls can detect and log the arrival of SYN packets and can for example detect and block a SYN scan. Rather than sending a SYN packet, the FIN scan sends a probe packet with the FIN flag set. According to RFC 793 (page 64), closed ports are required to reply to this probe with a RST packet. On the other side, open ports should ignore the probe and discard it.

These FIN packets have better chances to pass trough firewalls and packet-filters. The bad side is that the FIN scan won't work with all OS's; Microsoft Windows for instance, doesn't follow RFC 793's recommendations in this figure. Nevertheless, this can be a good way of detecting a Windows host.

The Xmas Tree scan uses the same principle as the FIN scan and enables the FIN, URG and PUSH flags.

The Null scan turns off all flags.

2.2.6 Fragmented packets scan

This option can be used with the SYN and FIN, Xmas and Null scans. Rather than sending the whole raw packets, the packets are fragmented. The purpose of the fragmentation is to separate the TCP header. In most cases, when a firewall reads the first fragmented packet, and doesn't have enough information on the TCP header, it will let the packet through.

2.2.7 UDP scan

The purpose of this scan is to determine which UDP ports are open on a host. The technique for this scan is to send raw UDP packets to all the ports we wish to scan. If the target sends back an ICMP "port unreachable" message, it means that the port is closed. In the other case, the port is open.

This scan can sometimes be very slow. Some systems limit the number of ICMP "destination unreachable" messages that can be sent. For example, the Linux Kernel limits the number of ICMP messages to 80 per 40 seconds. Nmap detects the speed of incoming ICMP messages and adapts the scan speed in order not to flood the network with useless packets. Windows systems don't follow this procedure and consequently, can be scanned very fast.

2.2.7 IP protocols scan

This scan's purpose is to determine which protocols are available on a host. The technique is to send raw packets to all the different protocols. If an ICMP error message is received, we assume the protocol isn't supported.

Chapter 3

IPv6

3.1 Introduction [Huit]

Days of the Ipv4 protocol are now counted. With the continuous growth of the Internet and the imminent convergence of networks, audio-visual and entertaining industries, Ipv4 will soon be out of addresses. Predictions based on the growth of the number of hosts in the Internet predict that in the next 10 years, there won't be enough IPv4 addresses left. Unfortunately, with the 32 bits address fields, the IPv4 protocol won't be able to stay around for ever, but who would have predicted such a huge growth of the Internet?

After a few years of negotiation, the specifications of the basic IPv6 protocol were published in January 1996.

The first objective of the Ipv6 protocol was to resolve the addresses problem. But other goals were to be met with the new protocol:

- Simplicity of the protocol in order to fasten the routing by the routers.
- Reduction of routing tables.
- Provide better security.
- Give attention to different types of services, better flow distinction. As an example, for "Real Time" traffic.
- Better multicast support.
- Auto-configuration support.
- The new protocol must permit future evolution.
- IPv4 and IPv6 must be able to work together.

The IPv6 protocol meets all these objectives. It keeps the best features of IPv4, drops the bad ones and adds new ones which didn't exist in the older protocol.

3.2 Address architecture [RFC 2373]

The first feature that we can notice with IPv6 is the enlargement of the number of addresses, compared to IPv4. With IPv6, we have 128 bits addresses, which should largely provide enough space for the future development of the Internet. This large number will also be useful to provide a bigger variety of layers of subnets than the four main layers of IPv4.

Each address on the internet identifies an interface. Unlike IPv4, with IPv6, an interface can have more than one address. This will make administration and routing easier.

There are three types of addresses:

- Unicast: identifies one interface on a host.
- Multicast: identifies a set of interfaces. A packet sent to this address is sent to all the interfaces identified with it.
- Anycast: identifies a set of interfaces. A packet sent to this address is sent to the "nearest" interface identified with this address.

There are no broadcast addresses, because multicast is used instead of it.

3.2.1 Notation

An address is written as eight 16-bits integers separated by colons. Each one of these integers is represented in the hexadecimal format.
Here's an example: `FE80:0000:0000:AD25:0210:0000:FE04:00D4`

This representation is quite compact (compared to the 8-bit integers representation), but it has the disadvantage that manipulating the hexadecimal form isn't very friendly.
But some abbreviations are possible:

- It is permitted to change one and only one string of zero's (0:0:0:0 as an example) into `::` .
- You can also take out the high level zero's (00F2 → F2).

The last example would become: `FE80::AD25:210:0:FE04:D4`

The subnet mask simple form is represented like in IPv4. The following example describes a 64 bits prefix:

```
FE80::AD25:210:CFF:FE04:D4/64
```

3.2.2 Unicast addresses

3.2.2.1 Standard unicast address

Simple hosts should be aware of the following IPv6 address structure. This form includes a subnet prefix.

`0                                              128-n          127`

n bits	128–n bits
subnet prefix	interface ID

3.2.2.2 Aggregatable global unicast Address

More sophisticated hosts can be aware of other hierarchical boundaries in the unicast address. In this type of address, hierarchical information is given by it's different fields. This type of address is assigned following a plan based on the providers (opposed to assignment based on geographical considerations) .

`0                                  64                            127`

3	13 bits	32 bits	16 bits	64 bits
001	TLA	NLA	SLA	Interface ID

TLA: Top Level Aggregator.

It's the provider or exchange point ID.

NLA: Next Level Aggregator.

Structured by the provider. It can be subdivised for more levels of hierarchy inside the provider's site.

SLA: Site Local Aggregator.

Identifier assigned to a link within a site.

3.2.2.3 Special addresses

The unspecified address: `0:0:0:0:0:0:0:0` or `::`

This indicates the absence of address. It can be used only as a source address, for example when a host doesn't have its address assigned yet.

The loopback address: `0:0:0:0:0:0:0:1` or `::1`

Can be used as a destination address to send a packet to itself.

IPv4-based address: `- 0:0:0:0:0:0:x.x.x.x` or `::x.x.x.x`

where x.x.x.x is the ipv4 address. This type of address is for nodes with both IPv4 and IPv6 capabilities.

`- 0:0:0:0:0:FFFF:x.x.x.x` or `::FFFF:x.x.x.x`

For node that only have IPv4 capabilities.

Site-local address:

This is for addressing on a site without requiring a global prefix. Routers must not forward packets out of the site.

```
0                32               64               96              127
```

10 bits	38 bits	16 bits	64 bits
1111111011	0	subnet ID	interface ID

Link-local address: This is for addressing on a single link. Routers must not forward any packets with a link-local source address.

0 32 64 96 127

10 bits	54 bits	64 bits
1111111010	0	interface ID

3.2.3 Anycast addresses

An IPv6 anycast address is an address that is assigned to several interfaces (belonging to different nodes), with the property that a packet sent to an anycast address is routed to the "closest" interface having that address, according to the routing protocols' measure of distance. Once the packet has reached a node with the anycast address, it is not forwarded to the other anycast nodes.

Anycast addresses are syntactically indistinguishable from unicast addresses, because they're allocated from the unicast address space. All the nodes with the same address must be aware they are anycast and should be configured explicitly.

Some restrictions are imposed on IPv6 anycast addresses:

- An anycast address must not be used as the source address of an IPv6 packet.
- An anycast address must not be assigned to an IPv6 host, that is, it may be assigned to an IPv6 router only.

3.2.4 Multicast addresses

An IPv6 multicast address is an identifier for a group of nodes. A node may belong to any number of multicast groups. Multicast addresses have the following format:

0 64 127

8 bits	4 bits	4 bits	112 bits
11111111	flags	scope	group ID

11111111 at the start of the address identifies the address as being a multicast address.

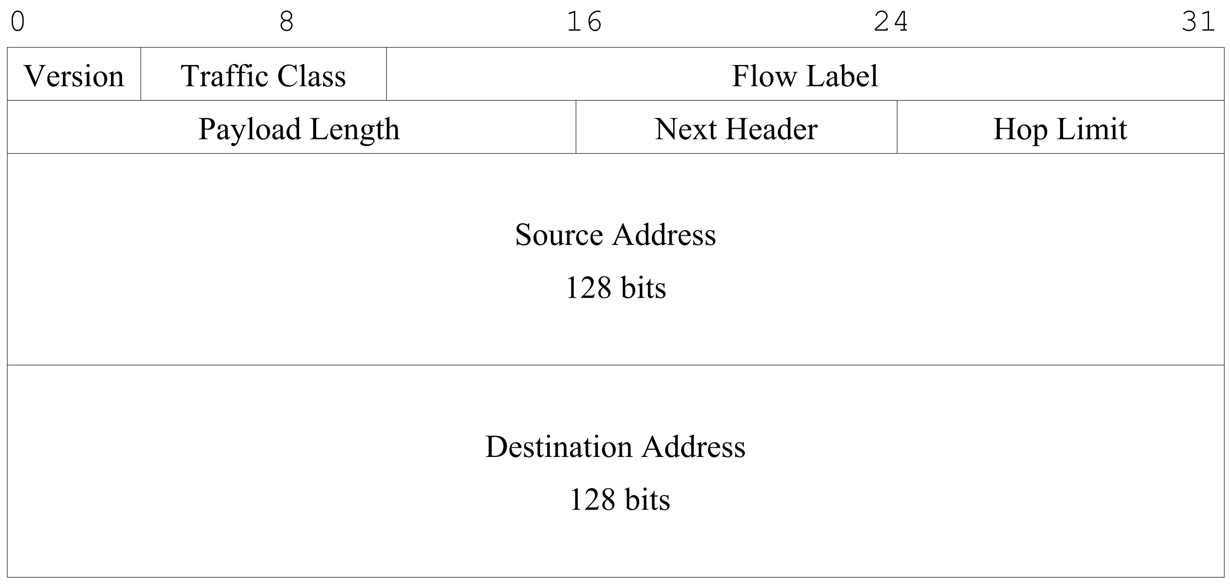
The flags field, among other functions, can specify the type of multicast address: permanently-assigned ("well-known") multicast or non-permanently-assigned multicast address.

The scope field is used to limit the scope of the multicast address: global scope, node-local scope, link-local scope, site-local scope,...

3.3 The header format

3.3.1 IPv6 header

Here's the specification for the new IPv6 header [RFC 2460]:



Header fields:

Version:	4-bit Internet Protocol version number = 6.
Traffic Class:	8-bit traffic class field. It is used to distinguish traffic that requires flow control from the others. Class from 0 to 7 are assigned to sources that can regulate their

	flow in case of congestion. Class 8 to 15 are assigned to real-time source with constant bit-rates.
Flow Label:	20-bit flow label. Contains a unique number chosen by the source and that identifies the flow. This will facilitate routers work in putting up Quality of Service (QoS) functionalities like routing or real-time traffic treatment.
Payload Length:	16-bit unsigned integer. Length of the IPv6 payload. Since the size of the IPv6 header is fixed, the Length field no longer includes the header size. But it include the size of all the optional extension headers.
Next Header:	8-bit selector. Identifies the type of header immediately following the IPv6 header. This one can be an Extension Ipv6 header, or any other type of protocol like TCP, UDP, ICMP, ...
Hop Limit:	8-bit unsigned integer. Has the same function than the IPv4 *Time to Live* field. It is decremented by 1 by each node that forwards the packet. The packet is discarded if Hop Limit is decremented to zero.
Source Address:	128-bit address of the originator of the packet.
Destination Address:	128-bit address of the intended recipient of the packet.

3.3.2 Comparison with the IPv4 header

IPv4 Header specification [RFC 791]:

0		8	16	24	31
Version	IHL	Type of Service	Total Length		
Identification			Flags	Fragment Offset	
Time to Live		Protocol	Header Checksum		
Source Address					
Destination Address					
Options					Padding

First thing to observe is that the *Version* field is the same. This allows the two Internet Protocol versions to work together. The routers just have to look in the first field to determine the IP version.

We can see that the IPv6 header has been simplified compared to IPv4. It was designed that way to simplify the processing of the header by the routers.

Main simplifications:

- The header has a fixed format. IPv4 had a variable format.
- No more header checksum. This will also simplify header processing. Checksums will be done by the lower layers.
- No more hop-by-hop segmentation. Hosts must learn the maximum acceptable segment size. This is done with the *path MTU discovery*. If a packet is too large it will simply be rejected. No automatic segmentation is provided by IPv6. However, an end-to-end segmentation procedure exists with extension headers.

3.3.3 Extension headers

With IPv6, optional internet layer information isn't included in the IP header any more. Each different option information is placed in an extension header. Each one of these headers has a next-header field, like the basic IPv6 header. All the extension headers are placed between the IPv6 header and the payload.

Here are the different extension headers:

- Hop-by-Hop Options header. This header contains information for all the routers on the path.
- Routing header. This header contains a list of routers the packet has to go through.
- Fragment header (see chapter 8.4.2)
- Destination options header: This header is only handled by the destination host. It will be used for future options that could be brought to IPv6.
- Authentication header: Header used to insure the identity of the source.
- Encapsulating security payload header. For encryption of the payload. This way sniffers won't be able to read the data. This header is provided with a sequence number which will protect against a "replay" type attack.

Here's an example of what would be an IPv6 packet with two extension headers:

IPv6 header Next header = Routing	Routing header Next header = Authentication	Authentication header Next header = TCP	TCP header & data …

3.3.4 Extension headers order

If several extension headers are used in a packet, it is recommended to put these headers in a certain order. The different layers can't always be processed in an arbitrary order. For instance, reassembling a fragmented packet can only be done after reception of the last packet. The header following the fragment header can only be processed after this reassembly. For the routing header, the next header won't even be processed if the node isn't the final destination. This is why the routing header should be the first extension header to be treated.

Here's the recommended order for the extension headers:

1. IPv6 header.
2. Hop-by-Hop Options header.
3. Destination Options header (for options to be processed by the first destination that appears in the IPv6 Destination Address field plus subsequent destinations listed in the Routing header).
4. Routing header.
5. Fragment header.
6. Authentication header .
7. Encapsulating Security Payload header.
8. Destination Options header (for options to be processed only by the final destination of the packet).
9. Upper-layer header.

3.4 Socket Interface API for IPv6 [RFC 2553]

This sections describes aspects of the Application Program Interface for the IPv6 sockets that are relevant for porting Nmap to IPv6.

3.4.1 Design considerations

The API changes should provide source and binary compatibility with existing code and application. This means that the existing ipv4 binaries will continue to run.

The changes in the API should be minimized in order to facilitate the porting process.

The applications must not be aware of the type of hosts they are dealing with. The API will interoperate in an invisible way with IPv4 or IPv6.

3.4.2 Socket interface

A new address family, AF_INET6, and a new protocol family, PF_INET6, are defined to suit the new IPv6 specifications. They are defined in <sys/socket.h>.

A new *in6_addr* structure is defined in <netinet/in.h>. This structure holds the address of a single ipv6 host.

```
struct in6_addr
  {
    union
      {
        uint8_t  u6_addr8[16];
        uint16_t u6_addr16[8];
        uint32_t u6_addr32[4];
      } in6_u;
#define s6_addr         in6_u.u6_addr8
#define s6_addr16       in6_u.u6_addr16
#define s6_addr32       in6_u.u6_addr32
  };
```

The *sockaddr_in6* structure is defined in <netinet/in.h>. This structure is the protocol-specific address data.

```
struct sockaddr_in6
  {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* Transport layer port */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* IPv6 scope-id */
  };
```

The *sin6_family* field identifies that this is a *sockaddr_in6* structure. This field may overlay the *sa_family* field of the *sockaddr* structure.

The *sin6_port* field contains the 16-bit UDP or TCP port number.

The *sin6_flowinfo* field contains the traffic class and flow label.

The *sin6_addr* field is a *in6_addr* structure which contains the 128 bits ipv6 address. The address is stored in network byte order.

The *sin6_scope_id* field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the sin6_addr field. If *sin6_addr* is a link scope, then *sin6_scope_id* is an interface index. And if *sin6_addr* is a site scope, then *sin6_scope_id* is a site identifier.

To create an IPv6 UDP/TCP socket descriptor, applications have to call the socket() function. Its use is the same as with IPv4, but the IPv6 protocol family needs to be specified.

Ex: s = socket(PF_INET6, SOCK_STREAM, 0);   for a TCP socket.

Special addresses:

When binding the source address of a connection, applications may want to let the system choose the address. This is done with the global variable:

`extern const struct in6_addr in6addr_any;`   defined in <netinet/in.h>

To call the loopback address, we can use the global variable:

`extern const struct in6_addr in6addr_loopback;` defined in <netinet/in.h>

3.4.3 Interface identification

Each interface on the system is identified by an interface index. This index is a small positive integer assigned by the kernel. There is a mapping done between those interfaces indexes and the interface names.

Functions are provided by the API to convert between name and index identification.

This function returns the index of a given interface name.

```
#include <net/if.h>
unsigned int  if_nametoindex(const char *ifname);
```

The next function returns the name of a given interface index.

```
#include <net/if.h>
char  *if_indextoname(unsigned int ifindex, char *ifname);
```

3.4.4 Node-name to address translation

The *gethostbyname()* function is left aside. Instead of that, we are going to use the *getaddrinfo()* function which can handle both versions of the Internet Protocol.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);

void freeaddrinfo(struct addrinfo *res);

char *gai_strerror(int errcode);
```

Considering the different arguments, this function returns a pointer to the *res addrinfo* structure filled up with among others, the address information.

```
struct addrinfo {
           int     ai_flags;
           int     ai_family;
           int     ai_socktype;
           int     ai_protocol;
           size_t  ai_addrlen;
           struct sockaddr *ai_addr;
           char   *ai_canonname;
           struct addrinfo *ai_next;
};
```

When resolving the address from a name, different types of addresses could be assigned to that name (IPv4, IPv6). This is why, if the resolved node has different types of addresses, not only one structure *addrinfo* will be returned, but a chained list of several *addrinfo* structures. There is one structure *addrinfo* returned for each type of address.

If you desire to resolve only one type of address, you must build an *addrinfo* structure containing the specifications of the type you want to resolve and point it with *hints*. The *hints* variable is an argument of the *getaddrinfo()* function. To resolve an IPv6 address for example, I built a *struct addrinfo *hints* structure where the *ai_family* field is set to AF_INET6.

Note that the *getaddrinfo()* function combines the possibilities of the *getipbynode(), getipnodebyaddr(), getservbyname()* and *getservbyport()* functions.

3.4.5 Address to node-name translation

The *gethostbyaddr()* function is left aside. Instead of that, we are going to use the *getnameinfo()* function which can handle both versions of the Internet Protocol.

```
#include <sys/socket.h>
#include <netdb.h>
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags);
```

This function returns the names of the address given by the *sockaddr* structure pointed by *sa*, in the location pointed by *host* and *serv*.

Note that *getnameinfo()* combines the functionality of *gethostbyaddr()* and *getservbyport()*.

3.4.6 Address conversion functions

We already knew the *inet_aton()* and *inet_ntoa()* functions which convert ipv4 addresses between text and binary form. IPv6 requires new functions to handle the *in6_addr* structure. These following functions handle both IPv4 and IPv6 families.

```
#include <sys/socket.h>
#include <arpa/inet.h>

   int inet_pton(int af, const char *src, void *dst);

   const char *inet_ntop(int af, const void *src,
                         char *dst, size_t size);
```

Inet_pton() converts an address from text representation (characters string) to an address structure whose family is specified by the *af* argument.

Inet_ntop() converts an address from the numeric representation (placed in a *sockaddr* structure) to characters string form.

3.5 Connecting to the 6bone

In the future, the entire Internet should become IPv6 capable. This will only be possible when all the routers will have been upgraded with IPv6 capabilities. Since it is obvious the transition can't be done all at once, a solution exists to get IPv6 connectivity before that day. The solution is tunneling with IPv6 in IPv4 encapsulation.

3.5.1 The 6bone (www.6bone.net)

The 6bone is an IPv6 testbed that is an outgrowth of the IETF IPng project that created the IPv6 protocols intended to eventually replace the current Ipv4. It's currently a world wide informal collaborative project.

The 6bone started as a virtual network (using IPv6 over IPv4 tunneling/encapsulation) operating over the IPv4-based Internet to support IPv6 transport, and is slowly migrating to native links for IPv6 transport.

The initial 6bone focus was on testing of standards and implementations, while the current focus is more on testing of transition and operational procedures. It operates under the IPv6 testing address allocation.

3.5.2 Tunneling

Since the 6bone is actually geographically limited, it isn't always possible to create a physical link directly to the IPv6 Network, by routing packets trough IPv6 routers. The solution for an IPv6 host to connect the 6bone is to send IPv6 packets encapsulated in IPv4 packets, in order for these packets to get routed on the IPv4 network. To achieve this you need an end point on the 6bone that extracts the IPv6 packets at the end of the tunnel and sends them on the IPv6 native network (6bone).

You can also connect large IPv6 networks to the 6bone with this tunneling method. If one host on the network has a configured tunnel to the 6bone, it can act as a gateway and forward packet between the networks.

Some ISP's provide this service but it is also possible to get free tunnels to the 6bone. For my experiments with the IPv6 scans, I will connect to the 6bone with a tunnel provided by "www.freenet6.net". Freenet6 provides a program that establishes a tunnel to their server, and assigns a global IPv6 address automatically to my interface. The program also modifies the routing tables of the system. After that, everything works as if we were connected directly on the 6bone.

Chapter 4

Porting Nmap to IPv6

The objective of this work is to extend Nmap with IPv6 capabilities. Considering the large number of Nmap's functionalities, the first ones to be modified will be the most used and interesting ones. The changes of the Nmap code are made from the source code of nmap-2.54BETA29, available at "www.insecure.org/nmap".

The source code is written in C. The porting of nmap is developed under Linux and is aimed to be used in the Linux environment.

4.1 What needs to be changed first

4.1.1 Adding an Ipv6 option

To start with, I added an *af* field to the global options structure of nmap. This field will contain the address family used for the scan (AF_INET or AF_INET6). Then I added the option "-6" in the list of arguments. If the user specifies this option in the nmap command line, during the parsing of the arguments, the program sets the *o.af* option to the value AF_INET6. If no "-6" option is specified, *o.af* is set by default to the value AF_INET. I will use this variable all along the program to check if we are running in IPv4 or IPv6 mode.

Note that if the user specifies an IPv6 address in the numeric notation (this type of notation: fe80::210:dcff:fe04:78d4), the program will automatically detect the IPv6 mode without the "-6" option. If several targets are specified in the command line, they must be from the same AF family. If a target host is name specified, the user should add the "-6" option if he wants the host to be scanned as an IPv6 host. This is because a lot of named hosts have both types of address family.

4.1.2 Global structures changes

The structures containing the specifications of the targeted hosts have to be modified to fit with the new IPv6 address format. To do this, I had to change or add to the *struct in_addr* fields, the new *struct in6_addr*. Each time an IPv4 address structure was needed, the equivalent IPv6 address structure had to be added.

The *in_addr* structures are used in many stages: in target specification, spoofed source specification, interfaces addresses resolution. This means that all parts of code which deal with those addresses should be modified.

4.1.3 Address parsing

When a target host is specified in the nmap command line, it can be given in different forms.

- In the named form, only one address will correspond to the given name. In this form, the user should specify the -6 option if he wants nmap to resolve the address from the AF_INET6 family.
- In the numeric form, the specified address can correspond to one or several addresses. The first way to specify a subnet to scan is the use of the netmask. Here's an example of the use of the netmask on an IPv6 address: fe80::210:dcff:fe04:78d4/10. The IPv6 mask can go from 0 to 128, 128 specifying one host, and 0 specifying the whole Internet. I decided to limit the use of this type of mask between 96 and 128. Using a mask value under 96 would mean that we are intending to scan more than 2^{32} hosts. Doing this wouldn't make much sense.

The parsing of the values of the address had to be modified to suit the new address representation.

4.1.4 Route discovering

4.1.4.1 Why do we need to know the route?

After creating an IPv6 socket, and before sending a packet on that socket, we need to fill the *sockaddr_in6* structure fields. These fields describe all the address information of the destination host.

```
struct sockaddr_in6
```

```
{
  sa_family_t     sin6_family;   /* AF_INET6 */
  in_port_t       sin6_port;     /* Transport layer port */
  uint32_t        sin6_flowinfo; /* IPv6 flow information */
  struct in6_addr sin6_addr;     /* IPv6 address */
  uint32_t        sin6_scope_id; /* IPv6 scope-id */
};
```

sin6_family must be "AF_INET6".

sin6_port is the tcp port.

sin6_flow_info is the flow id.

sin6_scope_id is the index of the interface on which the socket will be binded.

So, for each host to be scanned, we need to know through which interface the packets will be routed. Nmap already has a function (*routethrough()*) that does this for IPv4 addresses. There's no *scope_id* field with Ipv4, but it is necessary for a lot of types of scans to know on which interface of the system to route, as well as the associated ip address. All the raw type of scans need to know explicitly the source addresses to write in the packets.

4.1.4.2 The *routethrough()* function

The first thing this function does is to fetch all the interfaces of the system. This is done (in the *getinterfaces()* function) with an *ioctl(sd, SIOCGIFCONF, &ifconf)* system call. This call returns a list with all the interfaces on the system and their ip addresses.

After that, knowing the target address, this function has to determine through which interface the packets will be sent. This is done by several techniques:

The "procroutetechnique" for Linux type of environments:

This technique uses the file "/proc/net/route". This file contains all the routing information. The interesting fields are destination, mask and interface. Here's what the file looks like:

```
Iface Destination   Gateway   Flags   RefCnt  Use  Metric  Mask      MTU  Window IRTT
eth0  0000A8C0      00000000  0001    0       0       0    00FFFFFF   40    0       0
lo    0000007F      00000000  0001    0       0       0    000000FF   40    0       0
eth0  00000000      0100A8C0  0003    0       0       0    00000000   40    0       0
```

In formated form with "/sbin/route":

```
../$sbin/route
```

```
Table de routage IP du noyau
Destination     Passerelle      Genmask         Indic Metric Ref    Use Iface
192.168.0.0     *               255.255.255.0   U     0      0        0 eth0
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
default         192.168.0.1     0.0.0.0         UG    0      0        0 eth0
```

The file is read and all the different routes are saved. The mask of each one of those routes is applied to the destination address of the scan. Once we have the "masked" destination address, we compare it to the destination address of the routing table. If it matches, it's the proper route to use. We then copy the name of the corresponding interface.

Example:

We use the routing table given previously. If the target is 127.0.0.1:

We apply the first mask 255.255.255.0 to 127.0.0.1

→ 127.0.0.0 is different than 192.168.0.0

We apply the second mask 255.0.0.0

→ 127.0.0.0 is equal to 127.0.0.0

→ lo is the interface the packet should be routed to.

The "connectsockettechnique":

If the first technique doesn't work, the second option is to create a socket with the destination address, then to make a connect() call. After, we copy the source address that the kernel has binded the socket to. The function then verifies that this source address belongs to an existing interface. This technique may seem easier but it leaves less control in which route is chosen since the kernel makes the choice.

4.1.4.3 Porting *routethrough()* to *routethrough6()*

The first thing to do is to get all the interfaces of the system and the corresponding ipv6 addresses. First, I would have used the *ioctl()* call like with IPv4, but actually, this system call isn't compatible with IPv6 since the returned structure is too small to receive *in6_addr* type of address.

I had to find another way to get the interfaces. The file "/proc/net/if_inet6" (used by *ifconfig*) contains the information I'm looking for: the names of the interfaces and the corresponding addresses. I wrote the *getinterfaces6()* which reads this file and save all the info's.

To write *routethrough6()* I followed the ipv4 technique, except I had to look in the "/proc/net/ipv6_route" file instead. This file is quite like "proc/net/route". Here's an example:

```
00000000000000000000000000000001 80 00000000000000000000000000000000 00
00000000000000000000000000000000 60 00000000000000000000000000000000 00
00000000000000000000000000000000 60 00000000000000000000000000000000 00     ....
00000000000000000000ffff00000000 60 00000000000000000000000000000000 00
20020a00000000000000000000000000 18 00000000000000000000000000000000 00
20027f00000000000000000000000000 18 00000000000000000000000000000000 00
2002a9fe000000000000000000000000 20 00000000000000000000000000000000 00     ....
2002ac10000000000000000000000000 1c 00000000000000000000000000000000 00
2002c0a8000000000000000000000000 20 00000000000000000000000000000000 00
2002e000000000000000000000000000 13 00000000000000000000000000000000 00
fe800000000000000210dcfffe0478d4 80 00000000000000000000000000000000 00     ....
fe800000000000000000000000000000 0a 00000000000000000000000000000000 00
ff000000000000000000000000000000 08 00000000000000000000000000000000 00
00000000000000000000000000000000 00 00000000000000000000000000000000 00
```

```
00000000000000000000000000000000 00000000 00000000 00000000 00200001       lo
00000000000000000000000000000000 00000100 00000000 00000000 00200001     sit0
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000400 00000000 00000000 00200200       lo
00000000000000000000000000000000 00000000 00000000 00000000 00200001       lo
00000000000000000000000000000000 00000100 00000000 00000000 00040001     eth0
00000000000000000000000000000000 00000100 00000000 00000000 00040001     eth0
00000000000000000000000000000000 ffffffff 00000001 00000001 00200200       lo
```

In formatted style with */sbin/route*:

```
..$ /sbin/route -A inet6
Table de routage IPv6 du noyau
Destination                     Prochain Hop   Indic Metric Ref    Utilis. Iface
::1/128                         ::             U     0      0        0      lo
::/96                           ::             U     256    0        0      sit0
fe80::210:dcff:fe04:78d4/128    ::             U     0      0        0      lo
fe80::/10                       ::             UA    256    0        0      eth0
ff00::/8                        ::             UA    256    0        0      eth0
```

After retrieving the routes, masks, and interfaces, the principle is the same than in *routethrough()*. An exception is that I couldn't apply directly the mask bit to bit with the destination address because the mask and address aren't in the same form in "/proc/net/ipv6_route". A new little routine calculates the masked address.

4.2 TCP connect scan

4.2.1 IPv4 procedure of the scan

The principle of this scan is quite simple. A *connect()* call is made to the target host on a specific port. If the "connect()" works, the port is considered "open". If the call doesn't work, the error is retrieved and then nmap tries to figure out the state of the port: closed or firewalled. If the connection is refused, it means it is closed. But if no response is received, it is probably due to a firewall ignoring the connection attempt.

The TCP scan is a "positive" type of scan. It is called this way because it waits for a positive response from open ports. We will see farther that the SYN, ACK and Window scans are also "positive" scans. This is why these four scans are treated by the same function, *pos_scan()*.

4.2.2 Porting to IPv6

At that point, everything seemed ready for some actual scanning!! The main thing to do was to change the sockets from IPv4 to IPv6.

I started by adding the IPv6 *sockaddr_in6* structure. Then, at different places in the code, I assigned the values of the different fields of the *sockaddr_in6* structure, corresponding to the target's address. I regrouped next the *sockaddr_in6* field assignments. You will notice the way the ipv6 and ipv4 codes live together and are selected with the *"if"* statement on the global option *o.af.* This is the way choice is made between IPv4 or IPv6 all along the "TCP connect scan" process.

```
struct sockaddr_in sock;
struct sockaddr_in6 sock6;
....
if(o.af == AF_INET6){
     bzero((char *)&sock6,sizeof(struct sockaddr_in6));
     sock6.sin6_family=AF_INET6;
     sock6.sin6_port = htons(current->portno);
     for(k=0;k<4;k++)
       sock6.sin6_addr.s6_addr32[k]= target->host6.s6_addr32[k];
     sock6.sin6_flowinfo=0;
     sock6.sin6_scope_id=if_nametoindex(target->device);
    }
else{
     bzero((char *)&sock,sizeof(struct sockaddr_in));
```

```
      sock.sin_addr.s_addr = target->host.s_addr;
     sock.sin_port = htons(current->portno);
      sock.sin_family=AF_INET;
    }
```

In the case of the IPv6 sockets, we have to assign a value to the *scope_id*. I used the *if_nametoindex()* function to get the index of the interface through which the packet has to be routed. Note that it's the *routethrough6()* function that assigned the device to be used.

A bit farther, it's time to create the socket and to make the *connect()* call.

```
res = socket(o.af, SOCK_STREAM, IPPROTO_TCP);
if(o.af == AF_INET6)
  res = connect(res,(struct sockaddr*)&sock6,
                 sizeof(struct sockaddr_in6));
else
  res = connect(res,(struct sockaddr *)&sock,
                sizeof(struct sockaddr_in));
```

After sending the packets, we wait for the responses with the *get_connect_results()* function. A few modifications are also made to this function to make it IPv6 "capable".

4.2.3 Connect scan tests

4.2.3.1 Localhost test:

Here's the kind of output that an IPv4 scan gives on my localhost.

```
[seb@celeron seb]$ nmap localhost

Starting nmap V. 2.54BETA29 ( www.insecure.org/nmap/ )
Interesting ports on Celeron (127.0.0.1):
(The 1543 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
25/tcp     open        smtp
111/tcp    open        sunrpc
1024/tcp   open        kdm
6000/tcp   open        X11
Nmap run completed -- 1 IP address (1 host up) scanned in 0
seconds
```

Now let's try the new IPv6 scan on the same localhost and see what ipv6 services are available. To be sure all ipv6 services are recognized, I run two small server applications (source code in appendix B) that just listen for one connect call and receive one message on a specified TCP port. Of course, these application use IPv6 sockets. You should also know that Nmap doesn't scan all the ports available on a system, but only scans ports on which well known services could run. Nmap maintains a list of all those ports and related services. I chose to get my two applications to listen on ports 1030 and 6001, which are part of nmap's port list.

Let's have a try ...

```
[seb@celeron nmap-2.54BETA29_IPv6]$./nmap -sT -6 ip6-localhost

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on ip6-localhost (::1):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1030/tcp   open        iad1
6001/tcp   open        X11:1

Nmap run completed -- 1 IP address (1 host up) scanned in 0
seconds
```

You can see that that the two ports 1030 and 6002 which were listening, are properly detected by the scan. We also see that an ipv6 ssh service is running. I confirm that my two applications were scanned, because they both received a message from nmap.

4.2.3.2 Distant host with link-local address test

This test is done on a local network (link-local address) between two linux hosts. I runned my small application on the distant host on TCP ports 1032 and 6002.

```
[seb@celeron nmap-2.54BETA29_IPv6]$ ./nmap -sT
                                    fe80::250:baff:fee9:c7be

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1032/tcp   open        iad3
6002/tcp   open        X11:2

Nmap run completed -- 1 IP address (1 host up) scanned in 1
```

```
second
```

4.2.3.3 Distant host with global address

To do this test, I connect to the 6bone threw an IPv6 in IPv4 tunnel (see chapter 3.5 on connecting to the 6bone). I will scan an http server on the 6bone.

```
[seb@celeron nmap-2.54BETA29_IPv6]$ ./nmap -sT -6 www.kame.net

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on 2001:200:0:4819:210:f3ff:fe03:4d0
(2001:200:0:4819:210:f3ff:fe03:4d0):
(The 1542 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
53/tcp     open        domain
80/tcp     open        http
111/tcp    open        sunrpc
2401/tcp   open        cvspserver

Nmap run completed -- 1 IP address (1 host up) scanned in 51
seconds
```

Which we can compare to the IPv4 scan:

```
[seb@celeron nmap-2.54BETA29_IPv6]$ ./nmap www.kame.net

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on kame220.kame.net (203.178.141.220):
(The 1532 ports scanned but not shown below are in state: closed)
Port       State       Service
19/tcp     filtered    chargen
21/tcp     open        ftp
22/tcp     open        ssh
53/tcp     open        domain
80/tcp     open        http
111/tcp    filtered    sunrpc
137/tcp    filtered    netbios-ns
138/tcp    filtered    netbios-dgm
139/tcp    filtered    netbios-ssn
513/tcp    filtered    login
```

```
514/tcp    filtered    shell
2049/tcp   filtered    nfs
2401/tcp   open        cvspserver
5999/tcp   open        ncd-conf
7597/tcp   filtered    qaz
31337/tcp  filtered    Elite

Nmap run completed -- 1 IP address (1 host up) scanned in 23
seconds
```

An other target:

```
[seb@celeron nmap-2.54BETA29_IPv6]$ ./nmap -6 www.6bone.net

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on www.6bone.net (3ffe:b00:c18:1::10):
(The 1547 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http

Nmap run completed -- 1 IP address (1 host up) scanned in 19
seconds
```

4.3 Raw sockets considerations

4.3.1 Construction of raw packets

Nmap needs for some of its scans to send probe packets without previously establishing any connection. In fact most scans except the "TCP connect scan" use this raw packet probe technique. This permits Nmap to control precisely each value of the fields of the packets it sends. These raw packets must be built from a to z. To do this, we need to use raw sockets. Note that this functionality is only available for the root user. The following call will initialize a raw socket descriptor:

```
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

This is the socket on which the raw packet will be sent. To build the packet we need to build a structure containing all the structures of the desired TCP/IP packet and fill up all it's fields with the proper values.

Once the packet is constructed, you can send it on the socket with the call:

```
int sendto(int sd, const void *msg, size_t len, int  flags,
                    const struct sockaddr *to, socklen_t tolen);
```

4.3.2 Sniffing packets with libpcap

After sending raw packets to the target host, Nmap needs to sniff the incoming packets in order to determine the responses of the scanned host. To do this, Nmap has to access the content of the incoming raw packets.

In order to read those raw packets, Nmap uses the libpcap C library. Libpcap (Packet Capture library") provides a high level interface to packet capturing.

The use of "libpcap" allows us to access the entire content of the packets. The source address, source TCP port, and TCP flags must be retrieved from each one of the incoming packets. The payload data can also be read, but it isn't relevant to Nmap's purpose.

Here's an example of a simple sniffing of a packet using the "libpcap" library [Carst].

```
#include <pcap.h>
#include <stdio.h>
int main()
{
  pcap_t *handle;               /* Session handle */
  char *dev;                    /* The device to sniff on */
  char errbuf[PCAP_ERRBUF_SIZE]; /* Error string */
  struct bpf_program filter;    /* The compiled filter */
  char filter_app[] = "port 23"; /* The filter expression */
  bpf_u_int32 mask;             /* Our netmask */
  bpf_u_int32 net;              /* Our IP */
  struct pcap_pkthdr header;  /* The header that pcap gives us */
  const u_char *packet;         /* The actual packet */
  dev = pcap_lookupdev(errbuf);  /* Define the device */

  /* Find the properties for the device */
  pcap_lookupnet(dev, &net, &mask, errbuf);

  /* Open the session in promiscuous mode */
  handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);

  /* Compile and apply the filter */
  pcap_compile(handle, &filter, filter_app, 0, net);
  pcap_setfilter(handle, &filter);
  /* Capture a packet */
  packet = pcap_next(handle, &header);  /* Grab a packet */
  printf("Jacked a packet with length of [%d]\n", header.len);
  pcap_close(handle);/* And close the session */
  return(0);
}
```

"pcap_open_live()" opens a pcap session. The session can be in promiscuous or non promiscuous mode.

- *Promiscuous mode*: the host sniffs all traffic on the wire. In a non-switched environment, this could be all network traffic.
- *Non-promiscuous mode* (standard mode): the host sniffs only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer.

A filter can be applied to the traffic that is sniffed. This is very practical since you don't always want to sniff all the packets coming into the device. As an example, you could set a filter on port 23. This way you could sniff all the telnet traffic. The filter is compiled and set with the *pcap_compile()* and *pcap_setfilter()* calls.

You can after that, handle one packet with *pcap_next()* or enter a loop to capture *n* packets with *pcap_loop()* or *pcap_dispatch()* calls.

Nmap uses all these functions to capture the incoming packets.

4.3.3 Construction of a raw TCP/IPv4 packet

In the first case I will study the SYN scan. For this scan I will need to build a raw IP packet containing a TCP header with the SYN flag and a random payload. This is why this chapter explains how to build the TCP/IP packet. But the same principles will apply for the other types of raw IP packets (UDP, ICMP, TCP fragmented).

4.3.3.1 Structure of the packet

Here are the ip and tcp header structures from the GNU C Library:

```
struct ip
  {
#if WORDS_BIGENDIAN
    u_int8_t ip_v:4;                /* version */
    u_int8_t ip_hl:4;               /* header length */
#else
    u_int8_t ip_hl:4;               /* header length */
    u_int8_t ip_v:4;                /* version */
#endif
    u_int8_t ip_tos;                /* type of service */
    u_short ip_len;                 /* total length */
    u_short ip_id;                  /* identification */
    u_short ip_off;                 /* fragment offset field */
#define IP_RF 0x8000                /* reserved fragment flag */
#define IP_DF 0x4000                /* dont fragment flag */
#define IP_MF 0x2000                /* more fragments flag */
#define IP_OFFMASK 0x1fff           /* mask for fragmenting bits */
    u_int8_t ip_ttl;                /* time to live */
    u_int8_t ip_p;                  /* protocol */
    u_short ip_sum;                 /* checksum */
    struct in_addr ip_src, ip_dst;  /* source and dest address */
  };

struct tcphdr
  {
    u_int16_t th_sport;               /* source port */
    u_int16_t th_dport;               /* destination port */
    tcp_seq th_seq;                  /* sequence number */
    tcp_seq th_ack;                  /* acknowledgement number */
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;            /* (unused) */
    u_int8_t th_off:4;                /* data offset */
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;                /* data offset */
    u_int8_t th_x2:4;            /* (unused) */
#  endif
    u_int8_t th_flags;
#  define TH_FIN0x01
#  define TH_SYN0x02           /* SYN flag */
#  define TH_RST0x04
#  define TH_PUSH    0x08
#  define TH_ACK0x10
#  define TH_URG0x20
    u_int16_t th_win;            /* window */
    u_int16_t th_sum;            /* checksum */
    u_int16_t th_urp;            /* urgent pointer */
};
```

To build the TCP/IP packet we have to allocate enough memory to contain these two structures and the optional data payload. Then we fill up all the fields with the information corresponding to the specifications of the packet we want to send.

Since we know the characteristics of our raw packet, finding the values to fill the fields isn't too hard, except for the computation of the TCP checksum that is a little bit tricky. Let's see that next.

4.3.3.2 Computation of the TCP checksum

The checksum field, "`th_sum`", is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a pseudo header conceptually prefixed to the TCP header. This pseudo-header contains information on the upper IP layer. So this header is different according to the IP protocol version.

4.3.3.3 The pseudo-headers

Ipv4 pseudo-header[RFC 793]*:*

0	8	16 24 31
Source Address		
Destination Address		
zero	PTCL	TCP Length

This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

4.3.4 Construction of the new TCP/IPv6 raw packet.

The principle is the same as with IPv4, except that the headers and the pseudo-headers are different. Some research had to be done to provide the correct values to all the fields of the packet.

4.3.4.1 Structure of the IPv6 packet

We first have the IPv6 header. Here's it's structure from GNU C Library:

```
struct ip6_hdr
  {
    union
      {
      struct ip6_hdrctl
        {
          uint32_t ip6_un1_flow;   /* 4 bits version, 8 bits TC,
                                    * 20 bits flow-ID */
          uint16_t ip6_un1_plen;   /* payload length */
          uint8_t  ip6_un1_nxt;    /* next header */
          uint8_t  ip6_un1_hlim;   /* hop limit */
        } ip6_un1;
      uint8_t ip6_un2_vfc;         /* 4 bits version,
                                    * top 4 bits tclass */
      } ip6_ctlun;
    struct in6_addr ip6_src;      /* source address */
    struct in6_addr ip6_dst;      /* destination address */
  };

#define ip6_vfc   ip6_ctlun.ip6_un2_vfc
#define ip6_flow  ip6_ctlun.ip6_un1.ip6_un1_flow
#define ip6_plen  ip6_ctlun.ip6_un1.ip6_un1_plen
#define ip6_nxt   ip6_ctlun.ip6_un1.ip6_un1_nxt
#define ip6_hlim  ip6_ctlun.ip6_un1.ip6_un1_hlim
#define ip6_hops  ip6_ctlun.ip6_un1.ip6_un1_hlim
```

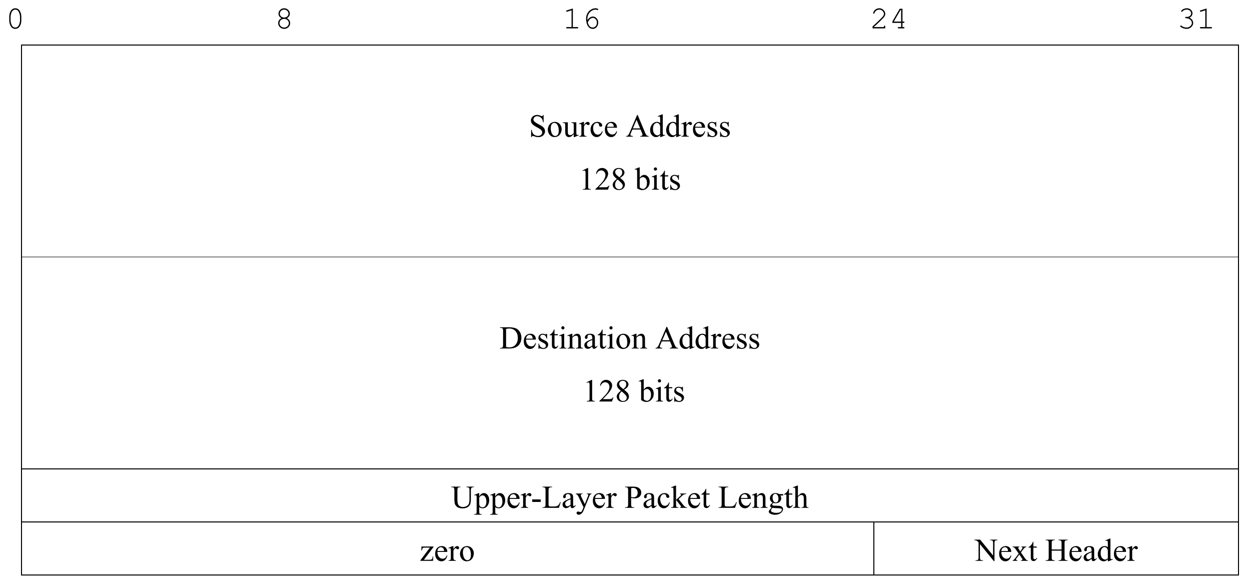
The *ip6_nxt* Next Header field is filled up in this case to indicate the TCP protocol with the value "IPPROTO_TCP".

We consider for the moment, only the basic header. We don't need extension headers for the moment. Of course, the TCP header is the same as with IPv4. But it will be necessary to change the value of the checksum since the pseudo-header is different.

4.3.4.2 Computation of the TCP checksum

The pseudo-header for IPv6 is different from IPv4. So the TCP checksum will be calculated following the new pseudo-header specifications.

Ipv6 pseudo-header[RFC 2460]*:*



- This pseudo-header includes the 128-bit source and destination IPv6 addresses instead of 32-bit IPv4 addresses.
- If the IPv6 packet contains a routing header, the destination address used in the pseudo-header is the final destination address.

- The *next header* value in the pseudo-header identifies the upper-layer protocol (6 for TCP, or 17 for UDP). It will differ from the *next header* value in the IPv6 header if there are extension headers between the IPv6 header and the upper-layer header.
- The *upper-layer packet Length* in the pseudo-header is the length of the upper-layer header and data (ex: TCP header plus TCP data). Some upper-layer protocols carry their own length information (ex: the Length field in the UDP header); for such protocols, that is the length used in the pseudo-header. Other protocols (such as TCP) do not carry their own length information, in which case the length used in the pseudo-header is the payload length from the IPv6 header, minus the length of any extension headers present between the IPv6 header and the upper-layer header.

4.4 SYN scan

The technique of the SYN scan is to avoid using the *connect()* call procedure and to build a raw packet with the SYN flag enabled. This will simulate the first packet sent by the *connect().* The targeted host will respond with a SYN/ACK packet if the port is open, or with a RST packet if it is closed. The trick is that hopefully, the connection attempt isn't logged because any connection procedure was completely accomplished. And if the attempt is not logged, the scan in most cases isn't detected.

4.4.1 The raw TCP/IPv6 SYN packet

I now have a function that builds an IPv6 TCP packet. Data can be added to the packet and different flags can be set. This function also takes care of sending the raw packet.

For the SYN scan, we have to call this function and specify all the information to put in the packet:

- source and destination IPv6 addresses.
- source and destination TCP ports.
- TCP flags: in our case, the SYN flag is specified.
- optional sequence and ack numbers.
- optional payload.

A different packet will be sent for each port and host to scan.

4.4.2 Waiting for the responses

After sending a SYN packet, we have to wait for the response packet. This is done by opening a pcap session which sniffs the incoming packets. The source ip and TCP port is retrieved from each incoming packet and compared to the destination ip address and TCP port of the SYN packet we've just sent. If all the information match, we can consider that the incoming packet is the response of the targeted host and port.

We now check the flags of the inbound packet the see whether the port is open or not. If the SYN and ACK flags are enabled, it means that the target port is ready to open a connection and we conclude that the port is open. If the RST flag is set, it means that the target refuses the connection and the port is closed.

If no response comes back from the target, after a time-out, another SYN packet is sent. If no response comes back after 20 retries, we consider the port firewalled. This means that there seems to be a firewall on the host ignoring the SYN packets for that port.

Since with this technique the open ports have to give a positive response, SYN scan is qualified as "positive scan" type. Notice that the connect, Ack and Window scan are also "positive scans" and are all processed by the same function in the Nmap program.

4.4.3 SYN scan tests

4.4.3.1 Distant host with link-local address test

Once again, I run my server application on known ports (6002 and 1032) to verify the scan.

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sS
                                     fe80::250:baff:fee9:c7be

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1032/tcp   open        iad3
6002/tcp   open        X11:2
```

```
Nmap run completed -- 1 IP address (1 host up) scanned in 1
second
```

Writing the code for creating raw IPv6 packets, I also wrote a function that read the interesting fields of the packet for debugging. I then knew that the IPv6 raw packets were correctly written. For the scan tests, I had to verify that those packets were correctly sent. To do this, I used the "tcpdump" application. "tcpdump", which also uses the pcap library, captures al the packets on a given interface. You can also provide traffic-filters to tcpdump to select the traffic to observe.

Here's an extract of the *tcpdump* results on the interesting ports for the previous SYN scan:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i eth0 port ssh or 1032 or
6002
tcpdump: listening on eth0
fe80::210:dcff:fe04:78d4.35793 > athlon6.1032: S  win 1024
athlon6.1032 > fe80::210:dcff:fe04:78d4.35793: S  ack 891976331 win
5760
fe80::210:dcff:fe04:78d4.35793 > athlon6.1032: R  win 0

fe80::210:dcff:fe04:78d4.35793 > athlon6.ssh: S 8 win 1024
athlon6.ssh > fe80::210:dcff:fe04:78d4.35793: S 4 ack 891976331 win
5760
fe80::210:dcff:fe04:78d4.35793 > athlon6.ssh: R  win 0

fe80::210:dcff:fe04:78d4.35793 > athlon6.6002: S  win 1024
athlon6.6002 > fe80::210:dcff:fe04:78d4.35793: S  ack 891976331 win
5760
fe80::210:dcff:fe04:78d4.35793 > athlon6.6002: R  win 0
```

The previous results correspond to the open ports. You can see that the target responds with a SYN/ACK packet to the SYN probe, and that nmap sends a RST packet after to stop the connection process.

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i eth0 port http or telnet
or 6003
tcpdump: listening on eth0

fe80::210:dcff:fe04:78d4.38208 > athlon6.6003: S  win 4096
athlon6.6003>fe80::210:dcff:fe04:78d4.38208: R 0:0(0) ack 3511209276
win 0

fe80::210:dcff:fe04:78d4.38208 > athlon6.http: S  win 4096
athlon6.http>fe80::210:dcff:fe04:78d4.38208: R 0:0(0) ack 3511209276
win 0

fe80::210:dcff:fe04:78d4.38208 > athlon6.telnet: S win 4096
```

```
athlon6.telnet > fe80::210:dcff:fe04:78d4.38208: R 0:0(0) ack
3511209276 win 0
```

Here I dumped packet on closed ports. You can see that the target sends a RST packet back meaning the port is closed.

My applications running on the target host on ports 1032 and 6002 waiting for a "connect()" call didn't receive any message. This shows the "stealth "capacity of the SYN scan technique.

4.4.3.2 Distant host with global address test

Again I connect to the 6bone with an IPv6 encapsulated in IPv4 tunnel. It's the provider of this tunnel (in my case www.freenet6.net) who assigns me an IPv6 global address. I will scan the same host as with the TCP connect scan, in order to compare.

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -6 -sS www.kame.net

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on apple.kame.net
(3ffe:501:4819:2000:210:f3ff:fe03:4d0):
(The 1542 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
53/tcp     open        domain
80/tcp     open        http
111/tcp    open        sunrpc
2401/tcp   open        cvspserver

Nmap run completed -- 1 IP address (1 host up) scanned in 82
seconds
```

Here's an extract of the tcpdump results on the interesting ports for the previous SYN scan:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i sit1 port 20 or 21 or
22
tcpdump: WARNING: sit1: no IPv4 address assigned
tcpdump: listening on sit1

sebpeterson.tsps1.freenet6.net.39114 > apple.kame.net.ftp-data: S
4073128729:4073128729(0) win 1024

apple.kame.net.ftp-data > sebpeterson.tsps1.freenet6.net.39114: R
0:0(0) ack 4073128730 win 0 [flowlabel 0x6800e]
```

```
sebpeterson.tsps1.freenet6.net.39113 > apple.kame.net.ssh: S
2699399320:2699399320(0) win 1024

apple.kame.net.ssh > sebpeterson.tsps1.freenet6.net.39113: S
221552378:221552378(0) ack 2699399321 win 16384 <mss 1440>

sebpeterson.tsps1.freenet6.net.39113 > apple.kame.net.ssh: R
2699399321:2699399321(0) win 0

sebpeterson.tsps1.freenet6.net.39113 > apple.kame.net.ftp: S
2699399320:2699399320(0) win 1024

apple.kame.net.ftp > sebpeterson.tsps1.freenet6.net.39113: S
323065027:323065027(0) ack 2699399321 win 16384 <mss 1440>

sebpeterson.tsps1.freenet6.net.39113 > apple.kame.net.ftp: R
2699399321:2699399321(0) win 0
```

All these packets transmission show well the Nmap SYN scan process. The port 20 (ftp-data) is closed because a RST packet is received after the SYN probe. And ports 21 (ftp) and 22 (ssh) are open because they sent back a SYN/ACK packet.

You can also see that Nmap works properly over a "IPv6 in IPv4" tunnel. This type of tunnel should be used by many hosts during the transition period to a full IPv6 capable Internet.

4.5 Ack and Window scans

The purpose of the ACK scan is to determine if ports on a host are filtered by a firewall. To do that, we send a raw packet with the ACK flag enabled, and with random looking acknowledgment and sequence numbers. If the port isn't filtered, the rule should be to send an RST packet back. If the port is actually filtered, the firewall usually ignores the ACK packet.

The purpose of the Window scan is to determine the open ports on certain systems which have an anomaly in reporting window sizes. This scan sends the same raw packet (with ACK flag) as the Ack scan, but results are interpreted differently. Here the rule is that open ports should send back a RST packet and the window size is positive.

4.5.1 ACK and Window scans procedure

The principle is the same as with the SYN scan. We have to build a TCP/IPv6 packet with the

ACK flag set and with the address and port information of the target host. We also need to provide fake ack and sequence numbers to the raw packet.

After sending the packet we wait for the response or for a time-out before concluding on the filtered or not-filtered state of the port.

4.5.2 ACK scan tests

In order to test firewall rules, I will configure a packet filter on a Linux host on a local network. To achieve this I use the "Netfilter-Iptables" tools (www.netfilter.org). Netfilter and iptables are the framework inside the Linux 2.4.x kernel which enables packet filtering and network address translation (NAT). It is the successor of the ipchains and ipfwadm systems.

- Netfilter is a set of hooks inside the linux 2.4.x kernel's network stack which allows kernel modules to register callback functions called every time a network packet traverses one of those hooks.
- Iptables is a generic table structure for the definition of rulesets. Each rule within an IP table consists of a number of classifiers (matches) and one connected action (target).

I configure the target host's packet filter by blocking ports 22(ssh) 80(http) and 1025. It is to notice that the IPv4 and IPv6 rule tables are separated and that I need to use "ip6tables" instead of "iptables".

```
[root@Athlon seb]# ip6tables -A INPUT -p tcp --destination-port
22 -j DROP
[root@Athlon seb]# ip6tables -A INPUT -p tcp --destination-port
80 -j DROP
[root@Athlon seb]# ip6tables -A INPUT -p tcp --destination-port
1025 -j DROP
[root@Athlon seb]# ip6tables --list

Chain INPUT (policy ACCEPT)
target     prot opt source         destination
           all      anywhere       anywhere
DROP       tcp      anywhere       anywhere       tcp dpt:ssh
DROP       tcp      anywhere       anywhere       tcp dpt:http
DROP       tcp      anywhere       anywhere       tcp dpt:blackjack
```

This last part shows the packet filter rules table for the incoming packets. We see that the three selected ports are blocked.

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sA -6 athlon6
```

```
Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state:
UNfiltered)
Port       State       Service
22/tcp     filtered    ssh
80/tcp     filtered    http
1025/tcp   filtered    listen

Nmap run completed -- 1 IP address (1 host up) scanned in 3
seconds
```

Here I dump packets from one filtered port and from one unfiltered port to see if everything works like we expected.

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i eth0 port 20 or 22
tcpdump: listening on eth0
celeron6.43523 > athlon6.ftp-data: . ack 2863392747 win 1024
athlon6.ftp-data > celeron6.43523: R 2863392747:2863392747(0)
celeron6.43523 > athlon6.ssh: . ack 2863392747 win 1024
celeron6.43524 > athlon6.ssh: . ack 2863392747 win 1024
celeron6.43525 > athlon6.ssh: . ack 2863392747 win 1024
celeron6.43526 > athlon6.ssh: . ack 2863392747 win 1024
celeron6.43527 > athlon6.ssh: . ack 2863392747 win 1024
celeron6.43528 > athlon6.ssh: . ack 2863392747 win 1024
```

We can verify that the filtered ssh port ignores the probe packet and doesn't send anything back. The open ftp port sends a RST packet back like it should.

4.6 FIN Scan

Since some firewalls and packet-filters can detect SYN scanning by logging the arrival of SYN packets, another solution is required to get past those filters. The stealth FIN scan sends raw packet with the FIN flag (end of connection) set. Hopefully these packets will get through packet-filters.

Open ports should ignore FIN packets and closed ports should send back an RST packet.

4.6.1 FIN scan procedure

We use the same function that builds a raw TCP/IPv6 packet, like in the SYN scan, except that we specify that the FIN flag must be set. Then we send the packets and wait for responses.

Since the open ports send no positive response, this scan isn't a "positive scan" type of scan. It is handled in a different function than Connect, Syn, Ack and Window scans. The FIN scan is a "super scan" type of scan, just like the Xmas, Null and UDP scans. These scans are treated in a different function than the "positive "scans (SYN, ACK, Windows). They are handled by the *super_scan()* function.

Receiving the responses is also different from the SYN scan since we don't interpret the incoming packets in the same way.

4.6.2 FIN scan tests

4.6.2.1 Distant host with link-local address test

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sF
                                     fe80::250:baff:fee9:c7be

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1032/tcp   open        iad3
6002/tcp   open        X11:2

Nmap run completed -- 1 IP address (1 host up) scanned in 2
seconds
```

Here are the dumps for the open ports. Since the FIN scan technique is to consider that open ports ignore the FIN packets, we can see that the following ports are open because no RST packets are sent back. Note that the FIN packets are sent 4 times, this to avoid errors due to packet loss or time-out.

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i eth0 port ssh or 1032
or 6002
tcpdump: listening on eth0

fe80::210:dcff:fe04:78d4.43318 > athlon6.1032: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43319 > athlon6.1032: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43318 > athlon6.ssh: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43319 > athlon6.ssh: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43318 > athlon6.6002: F 0:0(0) win 4096
```

```
fe80::210:dcff:fe04:78d4.43319 > athlon6.6002: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43318 > athlon6.6002: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43318 > athlon6.ssh: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43318 > athlon6.1032: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43319 > athlon6.6002: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43319 > athlon6.ssh: F 0:0(0) win 4096
fe80::210:dcff:fe04:78d4.43319 > athlon6.1032: F 0:0(0) win 4096
```

And now for a few closed ports, you see that a RST packet is always sent back:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i eth0 port http or
telnet or 6003
tcpdump: listening on eth0

fe80::210:dcff:fe04:78d4.34124 > athlon6.telnet: F 0:0(0)
athlon6.telnet > fe80::210:dcff:fe04:78d4.34124: R 0:0(0) ack 1

fe80::210:dcff:fe04:78d4.34124 > athlon6.6003: F 0:0(0)
athlon6.6003 > fe80::210:dcff:fe04:78d4.34124: R 0:0(0) ack 1

fe80::210:dcff:fe04:78d4.34124 > athlon6.http: F 0:0(0)
athlon6.http > fe80::210:dcff:fe04:78d4.34124: R 0:0(0) ack 1
```

4.6.2.2 Distant host with global address test

```
[root@celeron nmap-2.54BETA29_IPv6]#./nmap -6 -sF
                                    www.ipv6.euronet.be

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on 3ffe:8100:200:2::2 (3ffe:8100:200:2::2):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http
2601/tcp   open        zebra
2603/tcp   open        ripngd

Nmap run completed -- 1 IP address (1 host up) scanned in 60
seconds
```

```
[root@celeron seb]# /usr/sbin/tcpdump -t -i sit1 port 22 or 80
tcpdump: WARNING: sit1: no IPv4 address assigned
tcpdump: listening on sit1
sebpeterson.tsps1.freenet6.net.37477 > 3ffe:8100:200:2::2.http: F
sebpeterson.tsps1.freenet6.net.37478 > 3ffe:8100:200:2::2.http: F

sebpeterson.tsps1.freenet6.net.37477 > 3ffe:8100:200:2::2.ssh: F
```

```
3ffe:8100:200:2::2.ssh > sebpeterson.tsps1.freenet6.net.37477: R

sebpeterson.tsps1.freenet6.net.37477 > 3ffe:8100:200:2::2.http: F
sebpeterson.tsps1.freenet6.net.37478 > 3ffe:8100:200:2::2.http: F
sebpeterson.tsps1.freenet6.net.37477 > 3ffe:8100:200:2::2.http: F
sebpeterson.tsps1.freenet6.net.37478 > 3ffe:8100:200:2::2.http: F
```

We can see that the open port http (80) never sends an ACK packet back.

4.7 Xmas and Null scans

4.7.1 Scan procedure

The procedure is the same as with the FIN scan, except for the flags.

- Xmas scan : FIN, URG, PUSH flags enabled.
- Null scan : no flags enabled.

4.7.2 Xmas and Null scans tests

4.7.2.1 Distant host with link-local address test, for the Xmas scan

For this test, I run the TCP server applications on ports 1032 and 6002.

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sX -6 athlon6
Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1032/tcp   open        iad3
6002/tcp   open        X11:2

Nmap run completed -- 1 IP address (1 host up) scanned in 2
seconds
```

The open IPv6 ports are correctly recognized.

Let's dump the interesting packets in order to verify the procedure of the Xmas scan.

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port ssh
or 1032 or 6002
tcpdump: listening on eth0
celeron6.36826 > athlon6.ssh: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.ssh: FP 0:0(0) win 1024 urg 0
celeron6.36826 > athlon6.1032: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.1032: FP 0:0(0) win 1024 urg 0
celeron6.36826 > athlon6.6002: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.6002: FP 0:0(0) win 1024 urg 0
celeron6.36826 > athlon6.6002: FP 0:0(0) win 1024 urg 0
celeron6.36826 > athlon6.1032: FP 0:0(0) win 1024 urg 0
celeron6.36826 > athlon6.ssh: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.6002: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.1032: FP 0:0(0) win 1024 urg 0
celeron6.36827 > athlon6.ssh: FP 0:0(0) win 1024 urg 0
```

You can easily see here that this scan is a "non-positive" scan since the open ports give no response. You also see that the FIN, PUSH and URG flags are set (FP urg).

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port http
or telnet or 6003
tcpdump: listening on eth0
celeron6.63644 > athlon6.6003: FP 0:0(0) win 3072 urg 0
athlon6.6003 > celeron6.63644: R 0:0(0) ack 1 win 0

celeron6.63644 > athlon6.telnet: FP 0:0(0) win 3072 urg 0
athlon6.telnet > celeron6.63644: R 0:0(0) ack 1 win 0

celeron6.63644 > athlon6.http: FP 0:0(0) win 3072 urg 0
athlon6.http > celeron6.63644: R 0:0(0) ack 1 win 0
```

And here we see that closed ports send back a RST packet.

4.7.2.2 Distant host with link-local address test, for the Null scan

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sN -6 athlon6

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
```

```
Port       State       Service
22/tcp     open        ssh
1032/tcp   open        iad3
6002/tcp   open        X11:2

Nmap run completed -- 1 IP address (1 host up) scanned in 2
seconds
```

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port ssh
or 1032 or 6002
tcpdump: listening on eth0
celeron6.33639 > athlon6.1032: . win 2048
celeron6.33640 > athlon6.1032: . win 2048
celeron6.33639 > athlon6.6002: . win 2048
celeron6.33640 > athlon6.6002: . win 2048
celeron6.33639 > athlon6.ssh: . win 2048
celeron6.33640 > athlon6.ssh: . win 2048
```

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port http
or telnet or 6003
tcpdump: listening on eth0
celeron6.57069 > athlon6.6003: . win 2048
athlon6.6003 > celeron6.57069: R 0:0(0) ack 0 win 0

celeron6.57069 > athlon6.telnet: . win 2048
athlon6.telnet > celeron6.57069: R 0:0(0) ack 0 win 0

celeron6.57069 > athlon6.http: . win 2048
athlon6.http > celeron6.57069: R 0:0(0) ack 0 win 0
```

Here, we verify that no flags are set in the probe packets.

4.7.2.3 Distant host with global address tests: Null & Xmas

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sN -6
                                     www.ipv6.euronet.be

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on 3ffe:8100:200:2::2 (3ffe:8100:200:2::2):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http
2601/tcp   open        zebra
2603/tcp   open        ripngd

Nmap run completed -- 1 IP address (1 host up) scanned in 57
```

```
seconds

[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sX -6
                                     carmen.ipv6.cselt.it/ipv6/

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on
carmen.ipv6.tilab.com.0.0.0.0.0.0.0.0.8.b.6.0.1.0.0.2.ip6.int
(2001:6b8::198):
(The 1540 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
25/tcp     open        smtp
80/tcp     open        http
111/tcp    open        sunrpc
443/tcp    open        https
4045/tcp   open        lockd
8080/tcp   open        http-proxy

Nmap run completed -- 1 IP address (1 host up) scanned in 112
seconds
```

4.8 Fragment scan

In fact this isn't a particular type of scan. Actually, it's an option you can use in SYN, FIN, XMAS, or NULL scans. It causes these scans to use tiny fragmented IP packets. The idea is to split up the TCP header over several packets to make it harder for packet filters and intrusion detection systems to detect what you are doing.

4.8.1 Porting to IPv6.

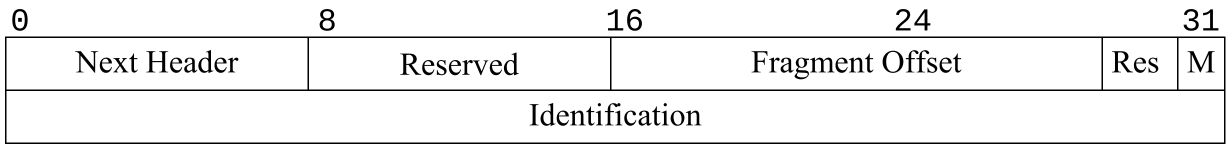
In order to carry out this scan, Nmap builds raw fragmented ip packets. Actually Nmap builds two ip packets where the first fragments stops in the middle of the TCP header. This, of course, has the effect that firewalls will have a "harder time" reading the TCP information. Most of them won't bother with reassembling the whole packet before forwarding it towards the host and port we are trying to scan.

To port this option to IPv6, since we have a new IPv6 header, I need to build new IP fragmented packets.

Luckily, the targeted host will reassemble the packet and send a response back. The responses shouldn't be fragmented (even if it was, the kernel would reassemble it for us), and the results analysis work as usual for all the scans.

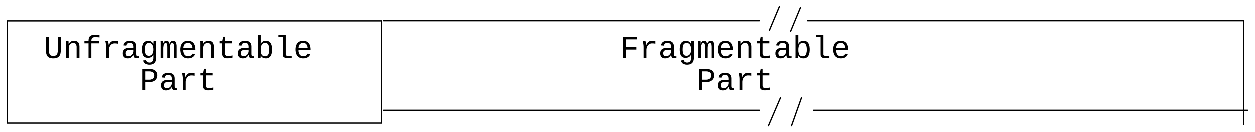
4.8.2 The IPv6 fragmentation header [RFC 2460]

The Fragmention header is used by an IPv6 source when it needs to send a packet larger than the path MTU to its destination. Unlike IPv4, fragmentation in IPv6 is performed only by source nodes, not by routers along a packet's delivery path. The Fragment header is identified by a Next Header value of 44 in the immediately preceding header, and has the following format:



Next Header:	8-bit selector. Identifies the initial header type of the fragmentable part of the original packet (defined below). Uses the same values as the IPv4 Protocol field.
Reserved	8-bit reserved field. Initialized to zero for transmission; ignored on reception.
Fragment Offset:	13-bit unsigned integer. The offset, in 8-octet units, of the data following this header, relative to the start of the fragmentable part of the original packet.
Res:	2-bit reserved field. Initialized to zero for transmission; ignored on reception.
M flag:	1 = more fragments; 0 = last fragment.
Identification:	32 bits. The source has to generate an identification value for each fragment. This value is generated by a counter that is incremented between each fragment. The identification must be different than that of any other fragmented packet sent recently with the same source address and destination address.

Here's the representation of an IPv6 packet we would like to fragment.:



First we have to divide the packet in two main parts:

- The unfragmentable part, which consists of the IP header and all the extension headers that have to be processed by the destination node.

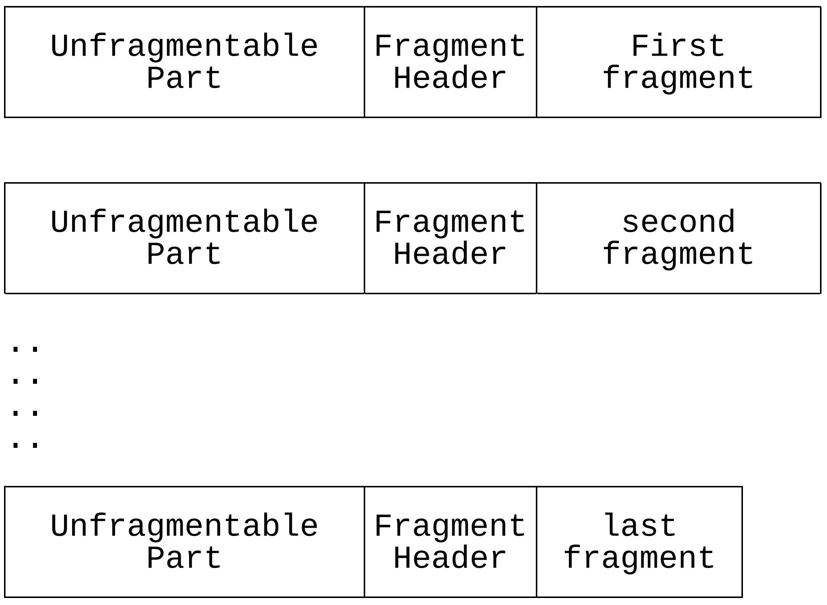
- The fragmentable part is the rest of the packet.

To send the fragments, the unfragmentable part has to be reproduced for each fragment. Then a fragmentation extension header has to be added.

This fragmentation header will contain the following values:

- The Next Header value that identifies the first header of the Fragmentable Part of the original packet.
- A Fragment Offset containing the offset of the fragment, in 8-octet units, relative to the start of the fragmentable part of the original packet. The Fragment Offset of the first ("leftmost") fragment is 0. An M flag value of 0 if the fragment is the last ("rightmost") one, else an M flag value of 1.
- The Identification value generated for the original packet.

Here's the result of the fragmentation:



4.8.3 Building the IPv6 fragmented packets.

The base IP packet we want to send is the same type as the one used with the SYN or FIN scans. We just need a tcp header with the proper content and flag, and no data payload. In these terms, two fragments will be enough to separate the tcp header.

Description of the first packet:

IPv6 header:

- The payload length field must be the length of the fragmentation extension header added to the length of the data.
- The length of the TCP header fragment. The first fragment of the TCP header will be 16 bytes long. All the fragments except the last one have to be a multiple of 8 bytes. Notice that the TCP header is in this case, with no options, 20 bytes long.
- The next header field has to mention that a fragmentation extension header follows the IPv6 header.

Fragmentation header:

- Next header: TCP header.
- Fragment offset: O.
- M (More fragments) : 1.
- Identification: random number, but must be saved to be written in the second fragment.

Description of the second packet:

IPv6 header:

- The payload length field must be the length of fragmentation extension header added to the length of the tcp header fragment. The second fragment of the TCP header is 4 bytes long, since the total TCP header was 20 bytes long.
- The next header field has to mention that a fragmentation extension header follows the IPv6 header.

Fragmentation header:

- Next header: TCP header.
- Fragment offset: 2 (2 * 8 bytes).
- M (More fragments) : 0.
- Identification: same as in the first fragment.

4.8.4 Fragmented scan tests

4.8.4.1 Distant host with link-local address test

I changed the ports listening on the target host to 1032, 1080 and ssh (22).

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -f -sF -6 athlon6

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
1080/tcp   open        socks
1374/tcp   open        molly
Nmap run completed -- 1 IP address (1 host up) scanned in 3
seconds
```

I can't dump just the packet destined to a specific target port since tcpdump can't read the port numbers of the tcp header because it is fragmented. This being exactly the purpose of the fragment option. On the other hand I can dump the good incoming packets, since they aren't fragmented.

So let's have a look at a few samples of probe packets and responses on closed ports:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0
....
celeron6 > athlon6: frag (0|16) 36904 > xfs: [|tcp]
celeron6 > athlon6: frag (16|4)
athlon6.xfs > celeron6.36904: R 0:0(0) ack 4126255222 win 0
celeron6 > athlon6: frag (0|16) 36904 > 643: [|tcp]
celeron6 > athlon6: frag (16|4)
athlon6.643 > celeron6.36904: R 0:0(0) ack 2814760156 win 0
celeron6 > athlon6: frag (0|16) 36904 > 16: [|tcp]
celeron6 > athlon6: frag (16|4)
....
```

You can see the two fragmented packets that are sent. The first has the tag (0 | 16) meaning the sequence number is 0 and the length of the packet is 16 bytes. The second one has the (16 | 4) tag: sequence number 16 and length of 4 bytes.

Again, the closed ports receive a RST packet.

Let's verify the open ports don't get these ACK's:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port ssh
or 1080 or 1374
tcpdump: listening on eth0
```

Nothing....ok.

4.8.4.2 Distant host with global address test

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sF -f -6
                                     www.ipv6.euronet.be
Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )

Interesting ports on 3ffe:8100:200:2::2 (3ffe:8100:200:2::2):
(The 1545 ports scanned but not shown below are in state: closed)
Port       State       Service
80/tcp     open        http
2601/tcp   open        zebra
2603/tcp   open        ripngd

Nmap run completed -- 1 IP address (1 host up) scanned in 137
seconds


[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sX -f -6
                                     www.v6.linux.or.jp

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )

Interesting ports on 2002:d29d:9e19::ffff:2
(2002:d29d:9e19::ffff:2):
(The 1540 ports scanned but not shown below are in state: closed)
Port       State       Service
21/tcp     open        ftp
22/tcp     open        ssh
53/tcp     open        domain
110/tcp    open        pop-3
873/tcp    open        rsync
1178/tcp   open        skkserv
8080/tcp   open        http-proxy
8081/tcp   open        blackice-icecap

Nmap run completed -- 1 IP address (1 host up) scanned in 135
seconds
```

4.9 UDP Scan

This scan requires to send a raw UDP packet to the target host. If the target sends back an ICMP "port unreachable" message, it means that the port is closed. In the other case, the port is open. This is why it requires to sniff incoming ICMPv6 packets to determine the responses of the target hosts.

UDP scan is a "non-positive" type of scan. It means that no positive response is received from open ports. Here it's the closed ports that send icmp messages back.

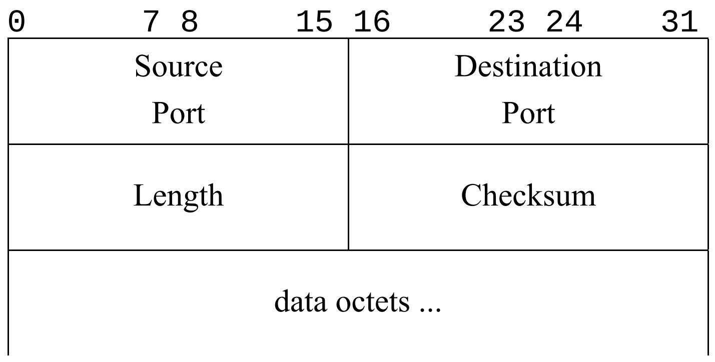
4.9.1 The raw UDP/IPv6 packet

Again we will have to build a whole raw IPv6 packet. I start building the packet like the TCP/SYN packet, except that an UDP header is used instead of the TCP one.

Here's the specification of the UDP header from GlibC:

```
struct udphdr {
        u_int16_t uh_sport;           /* source port */
        u_int16_t uh_dport;           /* destination port */
        u_int16_t uh_ulen;            /* udp length */
        u_int16_t uh_sum;             /* udp checksum */
};
```

User Datagram Header Format:



- The IPv6 header is the same as with the TCP SYN scan except the "next header" field, which will be in this case "IPPROTO_UDP".
- The UDP checksum ("`uh_sum`" field) is computed using the IPv6 pseudo-header in the same way as with TCP(described in chapter 4.3.4.2).

4.9.2 The ICMPv6 protocol (ICMP6)

4.9.2.1 General description

ICMPv6 is used by IPv6 nodes to report errors encountered in processing packets It is also used to perform other internet-layer functions, such as diagnostics (ICMPv6 "ping"). ICMPv6 is an integral part of IPv6 and MUST be fully implemented by every IPv6 node.

There are two main types of ICMPv6 messages:

- Error message. Type from 0 to 127.

1	Destination Unreachable
2	Packet Too Big
3	Time Exceeded
4	Parameter Problem

- Information message. Type from 128 to 255.

128	Echo Request
129	Echo Reply

The ICMPv6 message are encapsulated in an IPv6 packet. The next header field of the IPv6 header will contain the value "IPPROTO_ICMPV6".

The ICMPv6 messages have the following general format:

```
0              8              16               24              31
```

Type	Code	Checksum
Message Body		

Next is the ICMPv6 structure we will need to capture the packets, from GlibC:

```
struct icmp6_hdr
  {
    uint8_t     icmp6_type;   /* type field */
    uint8_t     icmp6_code;   /* code field */
    uint16_t    icmp6_cksum;  /* checksum field */
    union
      {
     uint32_t  icmp6_un_data32[1]; /* type-specific field */
     uint16_t  icmp6_un_data16[2]; /* type-specific field */
     uint8_t   icmp6_un_data8[4];  /* type-specific field */
      } icmp6_dataun;
  };
```

The type field is the ICMP message type: error or information.
The code field identifies the different categories of errors or information.

4.9.2.2 Destination Unreachable Message

This is the type 1 ICMP message. It's icmp6 message the target host will send us for the closed UDP ports. We then have to know how it is composed so we can retrieve from it the information we need.

```
0              8               16              24              31
```

Type	Code	Checksum
Unused		
As much of invoking packet as will fit without the ICMPv6 packet exceeding the minimum IPv6 MTU [IPv6]		

The possible codes for this "Destination Unreachable" message are:

- 0 - no route to destination
- 1 - communication with destination administratively prohibited
- 2 - (not assigned)
- 3 - address unreachable
- 4 - port unreachable

The original IPv6 packet which the ICMPv6 message originated from, is copied in the last field. If the packet exceeds the minimum MTU (Maximum Transmission Unit), it is truncated to that MTU.

4.9.3 Sniffing for the incoming icmpv6 packets

Using libpcap, we sniff the incoming packets. If one of the incoming packets is an ICMPv6 packet with the "Destination Unreachable Message" type (type 1) and "port unreachable" code (code 4), then we verify if the address of the original IP packet matches the target host. If it does, we conclude that the port is closed.

4.9.4 UDP scan tests

4.9.4.1 Distant host with link-local address test

The IPv4 UDP scan:

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sU  athlon

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on Athlon (192.168.0.2):
(The 1445 ports scanned but not shown below are in state: closed)
Port       State       Service
111/udp    open        sunrpc
137/udp    open        netbios-ns
138/udp    open        netbios-dgm
```

```
669/udp    open        unknown
852/udp    open        unknown
2049/udp   open        nfs
3130/udp   open        squid-ipc
32770/udp  open        sometimes-rpc4
Nmap run completed -- 1 IP address (1 host up) scanned in 1514
seconds
```

You can see that the scan has taken 1514 seconds. This is quite annoying, but it's the only way of scanning a Linux box since the kernel limits the number of icmp "destination unreachable" messages.

Since I have no IPv6 UDP services running on my Linux host, I wrote a small UDP/IPv6 server (see appendix C) that just waits for packets on a given port. I run this application on several ports (that are in the nmap well know UDP port list) , 1025, 1400, 27960.

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sU -6 athlon6

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on athlon6 (fe80::250:baff:fee9:c7be):
(The 1450 ports scanned but not shown below are in state: closed)
Port       State       Service
1025/udp   open        blackjack
1400/udp   open        cadkey-tablet
27960/udp  open        Quake3Server

Nmap run completed -- 1 IP address (1 host up) scanned in 1492
seconds
```

All open ports detected !

In the meantime, I checked the outgoing packets...:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 port ftp
or telnet or 1025 or 27960 or 1400
tcpdump: listening on eth0
celeron6.58101 > athlon6.ftp:  udp 0
celeron6.58101 > athlon6.telnet:  udp 0
celeron6.58101 > athlon6.1025:  udp 0
celeron6.58101 > athlon6.1400:  udp 0
celeron6.58101 > athlon6.27960:  udp 0
```

At the same time , I dumped all the ICMPv6 packets (protocol 58). I kept only a few because every closed port sends back an icmp6 message:

```
[root@celeron seb]# /usr/sbin/tcpdump -t -s 110 -i eth0 ip6 proto
58
tcpdump: listening on eth0
athlon6 > celeron6: icmp6: athlon6 udp port 927 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 1542 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 781 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 1363 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 1448 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 328 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 1490 unreachable
athlon6 > celeron6: icmp6: athlon6 udp port 32786 unreachable
```

4.9.4.2 Distant host with global address test

```
[root@celeron nmap-2.54BETA29_IPv6]# ./nmap -sU -6
                                     carmen.ipv6.cselt.it/ipv6/

Starting nmap V. 2.54BETA29_IPv6 ( www.insecure.org/nmap/ )
Interesting ports on
carmen.ipv6.tilab.com.0.0.0.0.0.0.0.0.8.b.6.0.1.0.0.2.ip6.int
(2001:6b8::198):
(The 1449 ports scanned but not shown below are in state: closed)
Port       State       Service
111/udp    open        sunrpc
514/udp    open        syslog
4045/udp   open        lockd
32771/udp  open        sometimes-rpc6

Nmap run completed -- 1 IP address (1 host up) scanned in 197
seconds
```

Chapter 5

Conclusion

Throughout this work I've been able to discover the basics of the new IPv6 protocol. I had to go through the handling of the new address structures. I also had to learn to use the new application program interface of IPv6. All this was done in the context of network security program, Nmap.

It was very interesting to understand the scan techniques of Nmap. I could see how it is possible to play around with protocol specifications to obtain information in unexpected ways. Programming at a low level, with the use of raw packet, is a very practical way to precisely control the content of all packets sent on the network. It allows the implementation of all the different advanced techniques.

Most Nmap scan's were ported to IPv6. At least the more important and used ones. Some other features could also be ported, such as the RPC scan, FTP bounce attack , IP protocol scan or host operating system identification.

Besides having been an enriching experience for myself, porting most features of Nmap to IPv6 will hopefully be useful to the open source community. Nmap is used by numerous people around the world. In the years to come, with the continuing integration of the new IPv6 protocol, the interest for an IPv6 capable Nmap can only get bigger.

Since nmap is a security auditing program, it would be interesting to lean on the security features included in IPv6. The new authentication and encryption headers provide security directly in the IP protocol. A future work could be to get Nmap to analyze whether open services on a host require any of these security information.

Appendix A

Notes on Red Hat Linux 7.2 IPv6 activation

By default with Red Hat 7.2, IPv6 features are compiled as modules. In order to load those modules you just need to edit the file "/etc/sysconfig/network" and add these lines:

```
NETWORKING_IPV6=yes
IPV6FORWARDING=yes
IPV6_AUTOTUNNEL=yes
IPV6_AUTOCONF=yes
```

IPv6 should be activated as well as interfaces configuration, auto tunnel configuration (sit0 interface) and ip forwarding (system acts like a router).

You should also edit the "/etc/hosts" file adding the following lines:

```
::1      ip6-localhost ip6-loopback localhost
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters
ff02::3  ip6-allhosts
```

Appendix B

Small IPv6 servers source code

1. Small TCP server

The purpose of this program is to get a TCP port listening on a host. This way the port is open and can be scanned.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#define  MSGSIZE 1024
#define PORT 6002

main(int argc, char *argv[])
{
  int port=PORT;
  char buffer [MSGSIZE];
  int server_socket, client_socket, client_adr_lenght, p;
  struct sockaddr_in6 server_adr, client_adr;

  if (argc != 2)
    printf("usage: ./s6 <port>
           \n...Using default port (%d).\n", port);
  else
    port = atoi(argv[1]);
  printf("\n*** IPv6 Server ***\n\n");

  /* Definition of the server address: */

  server_adr.sin6_family=AF_INET6;
  server_adr.sin6_port=htons(port);
  server_adr.sin6_flowinfo=0;         /* IPv6 flow information */
  server_adr.sin6_addr=in6addr_any;   /* IPv6 address */
  server_adr.sin6_scope_id=0;         /* IPv6 scope-id */
  /* Creation of the server socket.
     ------------------------------*/

  if(( server_socket= socket(PF_INET6, SOCK_STREAM, 0)) == -1)
    {
      fprintf(stderr,"\n Error %d in creating the socket: %s\n",
              errno,sys_errlist[errno]);
      exit(errno);
    }

  if(bind(server_socket,(struct sockaddr *)&server_adr,
        sizeof(server_adr)) == -1)
    {
      fprintf(stderr,"\n Error %d in the bind: %s\n" ,
              errno,sys_errlist[errno]);
      exit(errno);
    }

  /* Getting the socket to listen.
     ------------------------------*/

  if(listen(server_socket, 5) == -1)
    {
      fprintf(stderr,"\n Error %d in the listen: %s\n",
              errno,sys_errlist[errno]);
      exit(errno);
    }


  printf(" The server is waiting for a client on port %d.\n",port);

  if((client_socket=accept(server_socket, (struct sockaddr *)
                          &client_adr, &client_adr_lenght)) == -1)
    {
      fprintf(stderr,"\n Error %d in the accept: %s\n",
              errno,sys_errlist[errno]);
      exit(errno);
    }
  else if(client_socket != -1)
    printf("\nconnexion\n");

  (p=recv(client_socket, buffer, MSGSIZE,0));
  printf(" Lenght of the message: %d\n\nMessage received: %s\n\n",
         p, buffer);

  close(server_socket);
}
```

2. Small UDP server

The purpose of this program is to simulate a open UDP port on a host.

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define DEFAULT_PORT   1025
```

```
int main(int argc, char *argv[])
{
  struct sockaddr_in6 addr;
  int sd, bytes, addr_len=sizeof(addr);
  int port=DEFAULT_PORT;
  char buffer[1024];
  char adr_buffer[INET6_ADDRSTRLEN];

  if (argc != 2)
    printf("usage: ./s6_udp <port>\n Default port (%d) used.\n",
           port);
  else
    port = atoi(argv[1]);

  sd = socket(PF_INET6, SOCK_DGRAM, 0);

  bzero(&addr, sizeof(addr));
  addr.sin6_family = AF_INET6;
  addr.sin6_port = htons(port);
  addr.sin6_addr = in6addr_any;
  addr.sin6_scope_id=0;

  if ( bind(sd, (struct sockaddr*)&addr, sizeof(addr)) != 0 )
    {
      fprintf(stderr,"\n Error %d in the bind: %s\n",
              errno,sys_errlist[errno]);
      exit(errno);
    }
  while (1)
    {
      bytes = recvfrom(sd, buffer, sizeof(buffer), 0,
                       (struct sockaddr*)&addr, &addr_len);
      printf("msg from %s:%d (%d bytes)\n", inet_ntop(AF_INET6,
             &addr, adr_buffer, sizeof(adr_buffer)),
              ntohs(addr.sin6_port), bytes);
    }
  close(sd);
}
```

Appendix C

Nmap IPv6 source code

In this appendix are listed some of the main modifications I applied to Nmap. Since I have applied small changes in many places inside the code, I join only the source code of the new functions. Not all the new functions are included but only the most interesting one:

- Interface and route identification.
- Raw packet construction for all the scans.

The whole source code is available on the web at http://nmap6.webhop.net
For future locations of the program, just mail me at seb.peterson@easynet.be

1. Interface and route identification

The next function searches for all the devices on the system and finds their ipv6 address.

```
struct interface_info *getinterfaces6(int *howmany) {
  static struct interface_info mydevs[128];
  int numinterfaces = 0;
  char *p, *endptr;
  char buf[10240], addr[9];
  int i,j;

  FILE *devz;

  devz = fopen("/proc/net/if_inet6", "r");
  if(devz){
    while(fgets(buf,sizeof(buf), devz)) {
      p = strtok(buf, " ");

      for(j=0;j<4;j++)
       {
         for(i=0;i<8;i++)
           addr[i]=buf[i+j*8];
         addr[8]='\0';
         p = addr;
         endptr = NULL;
```

```
          mydevs[numinterfaces].addr6.s6_addr32[j] = htonl(strtoul(p, &endptr, 16));
        }

       for(i=0;i<4;i++)
        p = strtok(NULL, " \t\n");
       p = strtok(NULL, " \t\n");
       endptr = NULL;

       strcpy(mydevs[numinterfaces].name, p);

       numinterfaces++;
       if (numinterfaces == 127)  {
        error("My God!  You seem to have WAY too many interfaces!  Things may not work
              right\n");
        break;
       }

       mydevs[numinterfaces].name[0] = '\0';
     }
  }
  else
    fatal("Failed to open file /proc/net/if_inet6.  Unable to determine your
           interfaces(ipv6).");
  if (howmany) *howmany = numinterfaces;
  return mydevs;
}
```

The next function returns the name of the interface to use to route the outgoing packets, knowing the source and destination addresses.

```
char *routethrough6(struct in6_addr *dest, struct in6_addr *source) {
  static int initialized = 0;
  int i,j;
  static enum { procroutetechnique, connectsockettechnique, guesstechnique }
         technique = procroutetechnique;
  char buf[10240], addr[9], buf_adr[INET6_ADDRSTRLEN];
  struct interface_info *mydevs;
  static struct myroute {
    struct interface_info *dev;
    u32 mask;
    u32 dest[4];
  } myroutes[ROUTETHROUGH_MAXROUTES];
  u32 mask128[4];
  u32 temp_mask;
  int numinterfaces = 0;
  char *p, *endptr;
  char iface[64];
  static int numroutes = 0;
  FILE *routez;

  if (!dest) fatal("ipaddr2devname passed a NULL dest address");

  if (!initialized) {
    initialized = 1;
    mydevs = getinterfaces6(&numinterfaces);

    /* Now we must go through several techniques to determine info */

    routez = fopen("/proc/net/ipv6_route", "r");

    if (routez) {
      /* OK, linux style /proc/net/ipv6_route ... we can handle this ...
       * Now that we've got the interfaces, we g0 after the r0ut3Z
       */

      while(fgets(buf,sizeof(buf), routez)) {
       p = strtok(buf, " \t\n");

       for(j=0;j<4;j++)
         {
           for(i=0;i<8;i++)
             addr[i]=buf[i+j*8];
           addr[8]='\0';
           p = addr;
           endptr = NULL;
           myroutes[numroutes].dest[j] = strtoul(p, &endptr, 16);
         }

       p = strtok(NULL, " \t\n");

       endptr = NULL;
       myroutes[numroutes].mask = strtoul(p, &endptr, 16);
       if (!endptr || *endptr) {
         error("Failed to determine mask from /proc/net/route");
         continue;
       }

       for(i=0; i < 8; i++)
         p = strtok(NULL, " \t\n");

       strncpy(iface, p, sizeof(iface));
       if ((p = strchr(iface, ':'))) {
         *p = '\0'; /* To support IP aliasing */
       }

       for(i=0; i < numinterfaces; i++)
         if (!strcmp(iface, mydevs[i].name)) {
           myroutes[numroutes].dev = &mydevs[i];
           break;
         }
       if (i == numinterfaces)
         printf("Failed to find interface %s mentioned in /proc/net/route\n", iface);
       numroutes++;
         if (numroutes == ROUTETHROUGH_MAXROUTES)
           fatal("My God!  You seem to have WAY too many routes!\n");
      }
      fclose(routez);
    } else {
      technique = connectsockettechnique;
      fatal("procroute technique failed, no other techique...for now.");
       }
  } else {
    mydevs = getinterfaces6(&numinterfaces);
  }


  /* WHEW, that takes care of initializing, now we have the easy job of
   * finding which route matches
   */

  if (islocalhost6(dest)) {
    if (source){
      *source =  in6addr_loopback;
    }

    /* Now we find the localhost interface name, assuming ::1 is
     * localhost (it damn well better be!)...
     */

    for(i=0; i < numinterfaces; i++) {
      if (mydevs[i].addr6.s6_addr32[0] == htonl(0) &&
        mydevs[i].addr6.s6_addr32[1] == htonl(0) &&
        mydevs[i].addr6.s6_addr32[2] == htonl(0) &&
        mydevs[i].addr6.s6_addr32[3] == htonl(1))
      {
        return mydevs[i].name;
      }
    }
    return NULL;
  }

  if (technique == procroutetechnique) {
    for(i=0; i < numroutes; i++) {

      /* We need to transform the mask from integer form
       * to the 128 bits form.
       */

      bzero(mask128, 4 * sizeof(mask128));
      temp_mask = myroutes[i].mask;
      j = 0;
      while(temp_mask > 32)
       {
        temp_mask = temp_mask - 32;
        mask128[j] = 0xFFFFFFFF;
        j++;
       }

      mask128[j] = 0xFFFFFFFF;
      temp_mask =  32 - temp_mask;
      while(temp_mask > 0)
       {
        mask128[j] =  mask128[j] << 1;
        temp_mask--;
       }

      j++;
      while(j < 4)
       {
        mask128[j] = 0;
        j++;
       }

      if ((ntohl(dest->s6_addr32[0]) & mask128[0]) == myroutes[i].dest[0] &&
        (ntohl(dest->s6_addr32[1]) & mask128[1]) == myroutes[i].dest[1] &&
        (ntohl(dest->s6_addr32[2]) & mask128[2]) == myroutes[i].dest[2] &&
        (ntohl(dest->s6_addr32[3]) & mask128[3]) == myroutes[i].dest[3]) {
       if (source) {
        for(j=0;j<4;j++)
          source->s6_addr32[j] = myroutes[i].dev->addr6.s6_addr32[j];
       }
       printf("\n Found route through interface: %s\n\n",myroutes[i].dev->name);
       return myroutes[i].dev->name;
      }
    }
  }
  else
    fatal("I know sendmail technique ... I know rdist technique ... but I don't know
          what the hell kindof technique you are attempting!!!");
  return NULL;
}
```

2. Raw TCP/IPv6 packet construction

The next function constructs a raw TCP/IPv6 packet. It fills up the fields with the help of information given in the arguments: source and destination addresses and ports, sequence number, flags, window, TCP options, data. After constructing the raw packet it sends it on the *sd* raw socket.

```
int send_tcp_raw6( int sd, struct in6_addr *source, struct hoststruct *victim,
                   u16 sport, u16 dport, u32 seq, u32 ack, u8 flags,
                   u16 window, u8 *options, int optlen, char *data,
                   u16 datalen)
{
  struct pseudo_header6 {  /* for computing the checksum. */
    struct in6_addr s_addy;
    struct in6_addr d_addr;
    u32 up_layer_lenght;
    u16 zer0;
    u8 zer02;
    u8 next_header;
  };

  /* We build here a IPv6 packet with no Extension Header. */

  u8 *packet = (u8 *) safe_malloc(sizeof(struct ip6_hdr) + sizeof(struct tcphdr) +
                               optlen + datalen);  /* Total lenght of the packet. */
  struct ip6_hdr *ip6 = (struct ip6_hdr *) packet;
  struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct ip6_hdr));
  struct pseudo_header6 *pseudo = (struct pseudo_header6 *) (packet + sizeof(struct
                                   ip6_hdr) - sizeof(struct pseudo_header6));
  static int hop_lim = 0;

  /* With these placement we get data and some field alignment so we aren't
   * wasting too much in computing the checksum.
   */

  int k, res = -1;
  struct sockaddr_in6 sock6;
  char myname[MAXHOSTNAMELEN + 1];
  struct addrinfo hints, *result;
  int source_malloced = 0;

  /* check that required fields are there and not too silly
   * We used to check that sport and dport were nonzer0, but scr3w that!
   */

  if ( !victim || sd < 0) {
    fprintf(stderr, "send_tcp_raw: One or more of your parameters suck!\n");
    free(packet);
    return -1;
  }

  if (optlen % 4) {
    fatal("send_tcp_raw called with an option length argument of %d which is illegal
           because it is not divisible by 4", optlen);
  }

  if (!hop_lim) hop_lim = (get_random_uint() % 23) + 37;


  sethdrinclude(sd);

  /* if they didn't give a source address, fill in our first address */
  if (!source) {
    source_malloced = 1;
    source = (struct in6_addr *) safe_malloc(sizeof(struct in6_addr));

    hints.ai_family = AF_INET6;
    if (gethostname(myname, MAXHOSTNAMELEN) ||
        (getaddrinfo(myname, NULL, &hints, &result)))
           fatal("Cannot get hostname!  Try using -S <my_IP_address> or -e <interface
           to scan through>\n");

    memcpy(source, &((struct sockaddr_in6 *)result->ai_addr)->sin6_addr,
           sizeof(struct in6_addr));
  }


  sock6.sin6_family = AF_INET6;
  sock6.sin6_port = htons(0);
  for(k=0;k<4;k++)
    sock6.sin6_addr.s6_addr32[k]= victim->host6.s6_addr32[k];
  sock6.sin6_flowinfo=0;
  sock6.sin6_scope_id=if_nametoindex(victim->device);

  bzero((char *) packet, sizeof(struct ip6_hdr) + sizeof(struct tcphdr));

  for(k=0;k<4;k++)
    {
      pseudo->s_addy.s6_addr32[k] = source->s6_addr32[k];
      pseudo->d_addr.s6_addr32[k] = victim->host6.s6_addr32[k];
    }
  pseudo->next_header = IPPROTO_TCP;
  pseudo->up_layer_lenght = htonl(sizeof(struct tcphdr) + datalen);

  tcp->th_sport = htons(sport);
  tcp->th_dport = htons(dport);
  if (seq) {
    tcp->th_seq = htonl(seq);
  }
  else if (flags & TH_SYN) {
    get_random_bytes(&(tcp->th_seq), 4);
  }

  if (ack)
    tcp->th_ack = htonl(ack);

  tcp->th_off = 5 + (optlen /4) /*data offset*/;
  tcp->th_flags = flags;

  if (window)
    tcp->th_win = htons(window);
  else tcp->th_win = htons(1024 * (hop_lim % 4 + 1)); /* Who cares */

  /* We should probably copy the data over too */
  if (data && datalen)
    memcpy(packet + sizeof(struct ip6_hdr) + sizeof(struct tcphdr) + optlen,
           data, datalen);

#if STUPID_SOLARIS_CHECKSUM_BUG
  tcp->th_sum = sizeof(struct tcphdr) + datalen;
#else
  tcp->th_sum = in_cksum((unsigned short *)pseudo, sizeof(struct tcphdr) +
                          optlen + sizeof(struct pseudo_header6) + datalen);
#endif


  /* Now for the ip6 header */

  bzero(packet, sizeof(struct ip6_hdr));
  ip6->ip6_flow = htonl(0x60000000);
  ip6->ip6_plen = htons(sizeof(struct tcphdr) + datalen);
  ip6->ip6_nxt = IPPROTO_TCP;
  ip6->ip6_hlim =  hop_lim;
  for(k=0;k<4;k++)
    {
      ip6->ip6_src.s6_addr32[k] = source->s6_addr32[k];
      ip6->ip6_dst.s6_addr32[k] = victim->host6.s6_addr32[k];
    }

  if(TCPIP_DEBUGGING > 1)
    {
      log_write(LOG_STDOUT, "Raw TCP packet creation completed!  Here it is:\n");
      readtcppacket6(packet, 1);
    }


  res = Sendto("send_tcp_raw6", sd, packet,
               ntohs(ip6->ip6_plen) + sizeof(struct ip6_hdr), 0,
               (struct sockaddr *)&sock6,  sizeof(struct sockaddr_in6));

  if (source_malloced) free(source);
  free(packet);
  return res;
}
```

The next function is for debugging the building of raw TCP packets.

```
int readtcppacket6(unsigned char *packet, int readdata)
{
  struct ip6_hdr *ip6 = (struct ip6_hdr *) packet;
  struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct ip6_hdr));
  unsigned char *data = packet +  sizeof(struct ip6_hdr) + sizeof(struct tcphdr);
  int payload_len;
  struct in6_addr bullshit, bullshit2;

  char sourcehost[46], desthost[46];
  int k;

  printf("\nRaw TCP packet creation completed!  Here it is:\n");

  if (!packet) {
    fprintf(stderr, "readtcppacket: packet is NULL!\n");
    return -1;
  }

  for(k=0;k<4;k++){
    bullshit.s6_addr32[k] = ip6->ip6_src.s6_addr32[k];
    bullshit2.s6_addr32[k] = ip6->ip6_dst.s6_addr32[k];
  }


  payload_len = (ip6->ip6_plen);
  inet_ntop(AF_INET6, &bullshit, sourcehost, sizeof(sourcehost));
  inet_ntop(AF_INET6, &bullshit2, desthost, sizeof(desthost));



  if (ip6->ip6_nxt == IPPROTO_TCP) {
    printf("Version&Class&Flow label: %ux\n", ip6->ip6_flow);
    printf("Payload lenght: %i\n", ntohs(ip6->ip6_plen));
    printf("Next Header: %i\n", ip6->ip6_nxt);
    printf("Hop limit: %i\n", ip6->ip6_hlim);
    printf("TCP-IPv6 packet: %s[%d] -> %s[%d] \n", sourcehost,
          ntohs(tcp->th_sport), desthost,
          ntohs(tcp->th_dport));

    printf("Flags: ");
    if (!tcp->th_flags) printf("(none)");
    if (tcp->th_flags & TH_RST) printf("RST ");
    if (tcp->th_flags & TH_SYN) printf("SYN ");
    if (tcp->th_flags & TH_ACK) printf("ACK ");
    if (tcp->th_flags & TH_PUSH) printf("PSH ");
    if (tcp->th_flags & TH_FIN) printf("FIN ");
    if (tcp->th_flags & TH_URG) printf("URG ");
    printf("\n");

    if (tcp->th_flags & (TH_SYN | TH_ACK))
      printf("Seq: %u\tAck: %u\n",
             (unsigned int) ntohl(tcp->th_seq), (unsigned int) ntohl(tcp->th_ack));
    else if(tcp->th_flags & TH_SYN)
      printf("Seq: %u\n", (unsigned int) ntohl(tcp->th_seq));
    else if (tcp->th_flags & TH_ACK)
      printf("Ack: %u\n", (unsigned int) ntohl(tcp->th_ack));
  }

  return 0;
}
```

3. Fragmented raw TCP/IPv6 packet construction

The next function builds two fragmented raw TCP packets and sends them on the *sd* raw socket.

```
int send_small_fragz6(int sd, struct in6_addr *source, struct hoststruct *victim, u32
seq, u16 sport, u16 dport, int flags)
{
  struct pseudo_header6 {    /* for computing the checksum. */
    struct in6_addr s_addy;
    struct in6_addr d_addr;
    u32 up_layer_lenght;
    u16 zer0;
    u8 zer02;
    u8 next_header;
  };

  unsigned char packet[sizeof(struct ip6_hdr) + sizeof(struct ip6_frag) +
                       sizeof(struct tcphdr) + 100];
  struct ip6_hdr *ip6 = (struct ip6_hdr *) packet;
  struct ip6_frag *ip6frag=  (struct ip6_frag *) (packet + sizeof(struct ip6_hdr));
  struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct ip6_hdr) +
                                          sizeof(struct ip6_frag));
  struct pseudo_header6 *pseudo = (struct pseudo_header6 *) (packet + sizeof(struct
                                   ip6_hdr) + sizeof(struct ip6_frag)
                                   - sizeof(struct pseudo_header6));
  unsigned char *frag2 = packet + sizeof(struct ip6_hdr)
                         + sizeof(struct ip6_frag) + 16;
  struct ip6_hdr *ip6_2 = (struct ip6_hdr *) (frag2 - sizeof(struct ip6_hdr)
                                              - sizeof(struct ip6_frag));
  struct ip6_frag *ip6frag_2=  (struct ip6_frag *) (frag2 - sizeof(struct ip6_frag));
  static int hop_lim = 0;
  int res,k;
  struct sockaddr_in6 sock6;
  long id;

  if (!hop_lim)   hop_lim = (time(NULL) % 14) + 51;

  sethdrinclude(sd);

  sock6.sin6_family = AF_INET6;
  sock6.sin6_port = htons(0);
  for(k=0;k<4;k++)
    sock6.sin6_addr.s6_addr32[k]= victim->host6.s6_addr32[k];
  sock6.sin6_flowinfo=0;
  sock6.sin6_scope_id=if_nametoindex(victim->device);

  bzero((char *) packet, sizeof(struct ip6_hdr) + sizeof(struct ip6_frag) +
          sizeof(struct tcphdr));

  for(k=0;k<4;k++)
    {
      pseudo->s_addy.s6_addr32[k] = source->s6_addr32[k];
      pseudo->d_addr.s6_addr32[k] = victim->host6.s6_addr32[k];
    }
  pseudo->next_header = IPPROTO_TCP;
  pseudo->up_layer_lenght = htonl(sizeof(struct tcphdr));

  tcp->th_sport = htons(sport);
  tcp->th_dport = htons(dport);
  tcp->th_seq = (seq)? htonl(seq) : get_random_uint();
  tcp->th_off = 5 /*words*/;
  tcp->th_flags = flags;
  tcp->th_win = htons(2048); /* Who cares */
  tcp->th_sum = in_cksum((unsigned short *)pseudo,
                    sizeof(struct tcphdr) + sizeof(struct pseudo_header6));


  /* Now for the ip header of frag1
   * ------------------------------
   */

  bzero(packet, sizeof(struct ip6_hdr) + sizeof(struct ip6_frag));
  ip6->ip6_flow = htonl(0x60000000);
  ip6->ip6_plen = htons(sizeof(struct ip6_frag) + 16);
  ip6->ip6_nxt = IPPROTO_FRAGMENT;
  ip6->ip6_hlim =  hop_lim;
  for(k=0;k<4;k++)
    {
      ip6->ip6_src.s6_addr32[k] = source->s6_addr32[k];
      ip6->ip6_dst.s6_addr32[k] = victim->host6.s6_addr32[k];
    }


 /* Now for the fragmentation extension header of frag1
  */

  ip6frag->ip6f_nxt = IPPROTO_TCP;                          /* next header */
  ip6frag->ip6f_offlg = htons(1);                           /* more fragments flag */
  id = ip6frag->ip6f_ident =  get_random_u32();             /* identification */

  if (o.debugging > 1) {
    printf("Frag 1 written: \n");
    hdump(ip6,sizeof(struct ip6_hdr) + sizeof(struct ip6_frag) + 16 );
  }

  if ((res = sendto(sd, (const char *) packet,sizeof(struct ip6_hdr) +
       sizeof(struct ip6_frag) + 16 , 0,  (struct sockaddr *)&sock6,
       sizeof(struct sockaddr_in6))) == -1)
    {
      perror("sendto in send_syn_fragz6");
      return -1;
    }


  /* Now for the ip header of the second fragment
   * ------------------------------------------
   */

  bzero(ip6_2, sizeof(struct ip6_hdr) + sizeof(struct ip6_frag));
  ip6_2->ip6_flow = htonl(0x60000000);
  ip6_2->ip6_plen = htons(sizeof(struct ip6_frag) + 4);
  ip6_2->ip6_nxt = IPPROTO_FRAGMENT;
  ip6_2->ip6_hlim =  hop_lim;
  for(k=0;k<4;k++)
    {
      ip6_2->ip6_src.s6_addr32[k] = source->s6_addr32[k];
      ip6_2->ip6_dst.s6_addr32[k] = victim->host6.s6_addr32[k];
    }

 /* Now for the fragmentation extension header of frag2
  */

  ip6frag_2->ip6f_nxt = IPPROTO_TCP;                    /* next header */
  ip6frag_2->ip6f_offlg = htons(16);                    /* last fragment, offset 2 */
  ip6frag_2->ip6f_ident = id;                           /* identification */

  if (o.debugging > 1) {
    printf("Frag 2 written: \n");
    hdump(ip6_2,sizeof(struct ip6_hdr) + sizeof(struct ip6_frag) + 4);
  }

  if ((res = sendto(sd, (const char *)ip6_2, sizeof(struct ip6_hdr) + sizeof(struct
ip6_frag) + 4 , 0, (struct sockaddr *)&sock6, (int) sizeof(struct sockaddr_in6))) == -
1)
  {
    perror("sendto in send_tcp_raw6 frag #2");
    return -1;
  }

return 1;
}
```

4. Raw UDP/IPv6 packet construction

```
int send_udp_raw6( int sd, struct in6_addr *source, struct hoststruct *victim,
               u16 sport, u16 dport, u8 *data, u16 datalen)
{
  struct pseudo_header6 {  /* for computing the checksum. */
    struct in6_addr s_addy;
    struct in6_addr d_addr;
    u32 up_layer_lenght;
    u16 zer0;
    u8 zer02;
    u8 next_header;
  };
  unsigned char *packet = (unsigned char *) safe_malloc(sizeof(struct ip6_hdr) +
                           sizeof(struct udphdr_bsd) + datalen);
  struct ip6_hdr *ip6 = (struct ip6_hdr *) packet;
  struct udphdr_bsd *udp = (struct udphdr_bsd *) (packet + sizeof(struct ip6_hdr));
  struct pseudo_header6 *pseudo = (struct pseudo_header6 *) (packet + sizeof
                                   (struct ip6_hdr) - sizeof(struct pseudo_header6));
  static int hop_lim = 0;

  int res,k;
  struct sockaddr_in6 sock6;
  char myname[MAXHOSTNAMELEN + 1];
  struct addrinfo hints, *result;
  int source_malloced = 0;


  /* check that required fields are there and not too silly */
  if ( !victim || !sport || !dport || sd < 0) {
    fprintf(stderr, "send_udp_raw: One or more of your parameters suck!\n");
    free(packet);
    return -1;
  }


  if (!hop_lim) hop_lim = (get_random_uint() % 23) + 37;

  sethdrinclude(sd);

  /* if they didn't give a source address, fill in our first address */
  if (!source) {
    source_malloced = 1;
    source = (struct in6_addr *) safe_malloc(sizeof(struct in6_addr));

    if (gethostname(myname, MAXHOSTNAMELEN) ||
        (getaddrinfo(myname, NULL, &hints, &result)))
      fatal("Cannot get hostname!  Try using -S <my_IP_address> or -e <interface to
            scan through>\n");

    memcpy(source, &((struct sockaddr_in6 *)result->ai_addr)->sin6_addr,
           sizeof(struct in6_addr));
  }


  sock6.sin6_family = AF_INET6;
  sock6.sin6_port = htons(0); /* Must leave 0(< +- 500), or doesn't sendto ????!!!! */
  for(k=0;k<4;k++)
    sock6.sin6_addr.s6_addr32[k]= victim->host6.s6_addr32[k];
  sock6.sin6_flowinfo=0;
  sock6.sin6_scope_id=if_nametoindex(victim->device);

  bzero((char *) packet, sizeof(struct ip6_hdr) + sizeof(struct udphdr_bsd));


  /* Now the pseudo header for checksuming */

  for(k=0;k<4;k++)
    {
      pseudo->s_addy.s6_addr32[k] = source->s6_addr32[k];
      pseudo->d_addr.s6_addr32[k] = victim->host6.s6_addr32[k];
    }
  pseudo->next_header = IPPROTO_UDP;
  pseudo->up_layer_lenght = htonl(sizeof(struct udphdr_bsd) + datalen);

  udp->uh_sport = htons(sport);
  udp->uh_dport = htons(dport);
  udp->uh_ulen = htons(8 + datalen);

  /* We should probably copy the data over too */
  if (data)
    memcpy(packet + sizeof(struct ip6_hdr) + sizeof(struct udphdr_bsd),
           data, datalen);


  /* OK, now we should be able to compute a valid checksum */
#if STUPID_SOLARIS_CHECKSUM_BUG
  udp->uh_sum = sizeof(struct udphdr) + datalen;
#else
  udp->uh_sum = in_cksum((unsigned short *)pseudo, sizeof(struct udphdr_bsd) +
                     sizeof(struct pseudo_header6) + datalen );
#endif

  /* Goodbye, pseudo header! */
  bzero(pseudo, 12);


  /* Now for the ip header */

  bzero(packet, sizeof(struct ip6_hdr));
  ip6->ip6_flow = htonl(0x60000000);
  ip6->ip6_plen = htons(sizeof(struct udphdr_bsd) + datalen);
  ip6->ip6_nxt = IPPROTO_UDP;
  ip6->ip6_hlim =  hop_lim;
  for(k=0;k<4;k++)
    {
      ip6->ip6_src.s6_addr32[k] = source->s6_addr32[k];
      ip6->ip6_dst.s6_addr32[k] = victim->host6.s6_addr32[k];
    }

  if(TCPIP_DEBUGGING > 1)
    {
      log_write(LOG_STDOUT, "Raw UDP6 packet creation completed!  Here it is:\n");
      readudppacket6(packet, 1);
    }

  res = Sendto("send_udp_raw6", sd, packet,
             ntohs(ip6->ip6_plen) + sizeof(struct ip6_hdr), 0,
             (struct sockaddr *)&sock6,  (int)sizeof(struct sockaddr_in6));

  if (source_malloced) free(source);
  free(packet);
  return res;
}
```

The next function is for debugging the building of raw UDP packets.

```
int readudppacket6(unsigned char *packet, int readdata)
{
  struct ip6_hdr *ip6 = (struct ip6_hdr *) packet;
  struct udphdr_bsd *udp = (struct udphdr_bsd *) (packet + sizeof(struct ip6_hdr));
  unsigned char *data = packet +  sizeof(struct ip6_hdr) + sizeof(struct udphdr_bsd);
  int payload_len;
  struct in6_addr bullshit, bullshit2;

  char sourcehost[46], desthost[46];
  int k;

  printf("\nRaw UDP packet creation completed!  Here it is:\n");

  if (!packet) {
    fprintf(stderr, "readudppacket: packet is NULL!\n");
    return -1;
  }

  for(k=0;k<4;k++){
    bullshit.s6_addr32[k] = ip6->ip6_src.s6_addr32[k];
    bullshit2.s6_addr32[k] = ip6->ip6_dst.s6_addr32[k];
  }


  payload_len = (ip6->ip6_plen);
  inet_ntop(AF_INET6, &bullshit, sourcehost, sizeof(sourcehost));
  inet_ntop(AF_INET6, &bullshit2, desthost, sizeof(desthost));



  if (ip6->ip6_nxt == IPPROTO_UDP) {
    printf("Version&Class&Flow label: %ux\n", ip6->ip6_flow);
    printf("Payload lenght: %i\n", ntohs(ip6->ip6_plen));
    printf("Next Header: %i\n", ip6->ip6_nxt);
    printf("Hop limit: %i\n", ip6->ip6_hlim);
    printf("UDP-IPv6 packet: %s[%d] -> %s[%d] \n", sourcehost,
          ntohs(udp->uh_sport), desthost,
           ntohs(udp->uh_dport));
  }

  return 0;
}
```

Bibliography

[Fyod] Fyodor <fyodor@insecure.org>, *Nmap network security scanner man page.*

[Huit] Christian Huitema, *Ipv6 the new internet protocol. Second Edition*. Prentice Hall.

[Carst] Tim Carstens, *Programming with pcap*. timcarst at yahoo dot com The latest version of this document can be found at http://broker.dhs.org/pcap.htm

[RFC 793] *Transmission Control Protocol Functional Specification.* Information Sciences Institute University of Southern California, September 1981.

[RFC 791] *Internet Protocol Specification*. Information Sciences Institute University of Southern California, September 1981.

[RFC 2373] *IP Version 6 Addressing Architecture.* R. Hinden, Nokia, S. Deering, Cisco Systems. July 1998

[RFC 2460] *Internet Protocol, Version 6 (IPv6) Specification.* S. Deering, Cisco, R. Hinden, Nokia. December 1998

[RFC 2463] *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6.* A. Conta, Lucent, S. Deering, Cisco Systems. December 1998

[RFC 2553] *Basic Socket Interface Extensions for IPv6.* A. Conta, Lucent, S. Deering,Cisco Systems, R. Gilligan, FreeGate, S. Thomson, Bellcore, J. Bound, Compaq, W. Stevens. March 1999