12-2008

Steganography in IPV6

Barret Miller
University of Arkansas, Fayetteville

Steganography in IPv6

Steganography in IPv6

A thesis submitted in partial
fulfillment of the requirements for the degree of
Bachelors of Science in Computer Science

By

Barret Miller

December 2008
University of Arkansas

ABSTRACT

Steganography is the process of hiding a secret message within another message such that it is difficult to detect the presence of the secret message. In other words, the existence of the secret message is hidden. A covert channel refers to the actual medium that is used to communicate the information such as a message, image, or file. This honors thesis uses steganography within the source address fields of Internet Protocol Version 6 (IPv6) packets to create a covert channel through which clandestine messages are passed from one party to another. A fully functional computer program was designed and written that transparently embeds messages into the source address fields of packets and decodes embedded messages from these packets across IPv6 networks. This demonstrates the possibility of a covert channel within a protocol that will eventually be the default Internet protocol. This channel could be used for a malicious purpose such as stealing encryption keys, passwords, or other secrets from remote hosts in a manner not easily detectable, but it could also be used for a noble cause such as passing messages secretly under the watchful eyes of an oppressive regime. The demonstration of the covert channel in itself increases the overall information security of society by bringing awareness to the existence of such a steganographic medium.

This thesis is approved for recommendation to the Honors College

Thesis Director:

Dale R. Thompson

Dale R. Thompson, Ph.D., P.E.

Thesis Committee:

[illegible]

Craig W. Thompson, Ph.D.

[illegible]

Russell Deaton, Ph.D.

THESIS DUPLICATION RELEASE

I hereby authorize the University of Arkansas Libraries to duplicate this thesis when needed for research and/or scholarship.

Agreed [illegible] 11/20/08

Refused ________________________________

ACKNOWLEDGEMENTS

The research and work for this thesis was supported by a grant from the Honors College at the University of Arkansas in the form of the Honors College Research Grant. I thank Dr. Dale R. Thompson for all of his help and knowledgeable encouragement and for the use of equipment and space in the Security, Network, Analysis, and Privacy (SNAP) Lab throughout my work on this thesis. I also would like to thank Dr. Craig W. Thompson and Dr. Russell Deaton for serving on my committee and all of my professors at the University of Arkansas without whom I would have never have had the foundation on which to stand in order to reach this far. Additionally, I would like to thank Janne Lindqvist for his presentation at the DefCon 2007 conference that gave me the idea for this thesis.

Finally, I want to thank my family for encouraging me to work hard in order to achieve my dreams.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem

In an increasingly connected world, more and more devices are being networked together than ever before. This trend will continue with not only more laptop and desktop computers needing to be able to talk to one another around the world, but handheld PDAs, cell phones, cars, and, eventually, even toasters, refrigerators, and other household items. Most networked devices today are connected through Internet Protocol version 4 (IPv4), a layer three protocol of the International Standard Organization's Open System Interconnect (ISO/OSI) model. IPv4 provides for a theoretical possibility of two raised to the thirty-second power or a little over 4 billion possible addresses, but for practical reasons of off-limits address ranges, the actual number of IPv4 addresses available for use to the world is about 3.7 billion, and these are rapidly running out.

Internet Protocol Version 6 (IPv6) is the "next generation" Internet protocol that is set to slowly merge with and ultimately replace IPv4. According to a recent article [1] from Ars Technica, if the world continues at its current rate of adding 170 million IP addresses per year for new hosts that are connected to the Internet, people will exhaust the current address space allowed for by IPv4 in 7.5 years. This is the main driving force behind the push to switch to IPv6. IPv6 allows for astronomically[1] more addresses than people could possibly ever use, which shows that the Internet Engineering Task Force, the group that guides the development of protocols that run the Internet, does not want to run into the "limited address space" problem again in the future. Switching to IPv6 is

[1] $2^{128} = 3.4 \times 10^{38}$ possible addresses.

necessary and inevitable. However, society must also be aware of the risks and otherwise unintended possibilities that accompany the adoption of any new protocol or technology, and IPv6 carries at least one significant unintended possibility of allowing for a covert channel to be created and exploited using the built-in mechanism of the source address of the header.

1.2 Objective

The objective of this thesis is to demonstrate the existence of a covert channel inherent in the IPv6 protocol that provides the ability to steganographically transmit secret messages between parties.

Contributions of this research include:

- Exposure of a vulnerability in the Internet Protocol version 6 (IPv6). The protocol IPv6 will become the standard protocol for the Internet in the future.
- Demonstration that the vulnerability can be exploited.
- Implementation of both the transmitting and receiving programs to create a covert channel over IPv6.
- Discussion of possible ways to mitigate this threat.

1.3 Approach

As pointed out in [2], the IPv6 specification [3] along with the privacy extensions for the stateless address autoconfiguration feature of IPv6 [4] introduce the possibility of embedding a significant amount of secret data into the source address field of an IPv6 packet header that will likely be undetectable to an uninformed observer. The source address is a 128-bit field, which is intended to contain the universally unique Internet address of the originator of the packet. The privacy extensions proposed for IPv6 rely on the random generation of a 64-bit portion of the 128-bit source address, and it is the expectation that the built-in randomness will create a shield of entropy, which should effectively hide any enclosed message.

The approach taken in this thesis was to demonstrate this vulnerability by writing a program that used the interface id portion of the source address fields of IPv6 packets as a covert channel in which to transmit secret data. This program can both transmit and receive secret data in the source addresses of IPv6 packets in two different and important ways.

The first way the program can be configured to work is to embed the message by changing the MAC address of the originating host to the secret message. With this configuration, the secret message will simply be embedded into all packets that would leave the originating host during normal network use as long as the MAC address of the network interface that contains the secret message is used to connect to an IPv6 network. This works because of the default way IPv6 source addresses are derived from the MAC address of the network interface device used by the host.

The second way the program can embed messages is by explicitly creating IPv6 packets containing the message in the source address field, which are then injected into the network. The packets transmitted in this manner would not be transmitted as part of the normal networking activities of the host and is, therefore, slightly less stealthy, but can, in some cases, be more practical.

The way the software decodes the received messages depends upon the way in which the message was transmitted. All parties involved in sending and receiving the secret message ahead of time must know the method of transporting the message.

1.4 Potential Impact

The immediately apparent potential impact of this thesis is that the application developed in conjunction with the thesis could be used for noble or malicious purposes. One malicious way in which it could be used is to steal information such as passwords, encryption keys, or other sensitive data from a remote host in a way not easily detectable to the victim. In nobler applications, it could also be used by a covert agent behind enemy lines that needs to get sensitive information out to the headquarters secretly, or by political dissidents of an oppressive regime that need to operate secretly.

The far-reaching impact is that the awareness of the possibility of this covert channel increases the overall information security of society as well as the general knowledge of the workings of the protocol that will become the default for the Internet.

1.5 Organization of this Thesis

Chapter 2 covers the essential background knowledge required for a full understanding of the work presented in this thesis including IPv6 and steganography. Chapter 3 describes in detail the approach and methods used to complete this thesis. Chapter 4 details the specific implementation of the encoding and decoding program. Chapter 5 presents tangible results of the system in the form of screenshots of the program sending and receiving data, and packet captures from sending and receiving hosts on an IPv6 network. Chapter 6 summarizes the conclusions of this work and describes possible future work and extensions of this project. The Appendix contains a full list of options and details of how to use the program provided with this thesis.

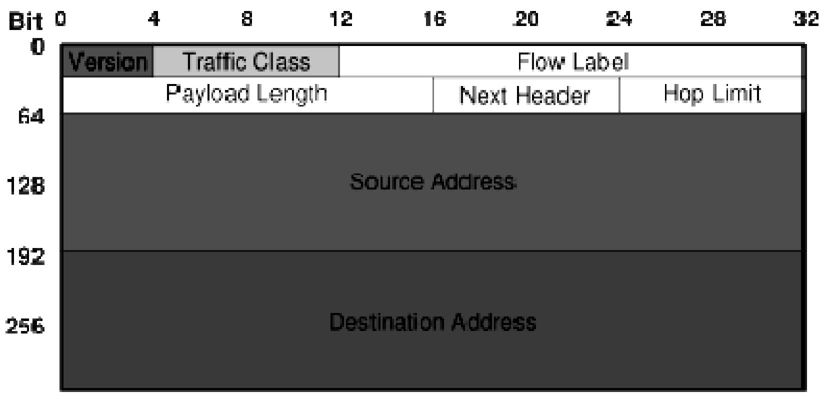
2. BACKGROUND

2.1 Key Concepts

In order to get a full understanding of the scope of this thesis and how the implementation functions, one must first understand some of the background concepts. The first concept to understand is the architecture of the IPv6 packet as a whole and the format of the address fields of the IPv6 packet. It is also important to understand the ways in which this address can be obtained or calculated by hosts on an IPv6 network. The second concept is the area of sending secret messages, called steganography, and covert channels.

2.1.1 The Architecture of an IPv6 Packet

An IPv6 Packet Header consists of the fields shown below in Figure 1: IPv6 packet. The exact use of all the different fields is unimportant for a full understanding of this thesis, but it is useful for the reader to be able to visualize the packet header. Important fields are the destination and source address fields, which allows routers to direct the packet to its destination and provide a return address to that destination. In addition, there is a version field indicating IPv4 or IPv6 and the next-header field, which specifies the layer above the current IP layer, such as TCP or UDP.



From http://upload.wikimedia.org/wikipedia/commons/thumb/3/32/IPv6_header_rv1.svg/410px-IPv6_header_rv1.svg.png

Figure 1: IPv6 packet

2.1.2 IPv6 Addresses

IPv6 Addresses are 128-bit long fields in the packet header. It is useful for humans to represent binary IPv6 addresses with text that is easier to comprehend than the underlying ones and zeros, and this method is detailed in Section 2.1.2.1. These addresses are broken down into different classes each with its own purpose defined in Section 2.1.2.2. The specification for the IPv6 addressing architecture [5] makes a distinction between a node and an interface in that a node may have multiple interfaces each with a different address, and this specification states that any interface can be used as an identifier for the node. The addressing architecture specification [5] also mandates that each interface have at least a single Link-Local unicast address, which is defined below, but it also allows single interfaces to have multiple addresses of all types defined below.

2.1.2.1 Text Representation

IPv6 addresses are represented in a meaningful way to humans by eight quadruples of hexadecimal (hex) digits separated by colons for a total of 32 hex digits representing 128 bits. For example, an IPv6 address can be written as fe80:0000:0000:0000:021a:70ff:fe14:8ac0. For addresses such as this that contain a string of consecutive zeros, they can be written in a shorthand notation as fe80::21a:70ff:fe14:8ac0. In other words, all zeros in the chain can be omitted from the address with two colons in their place. This can be done for one and only one chain of consecutive zeros within a single address. It is therefore incorrect to write an address such as the following (in longhand notation) fe80:0000:0000:0000:021a:7000:0000:00c0 as the shorthand fe80::21a:7::c0. Only one of the two strings of consecutive zeros may be chosen to leave out in the shorthand representation.

2.1.2.2 Types of Addresses and Their Scope

In IPv6 there are three main classes of addresses. The first class is called Unicast, which means that this type of address represents the unique (within a subnet prefix) address of a single interface on a network. There are two important subtypes of unicast addresses defined in [5] and they are: Global Unicast defined by all prefixes not designated for the other unicast address subtypes and Link-Local unicast defined by the prefix fe80::/10 [5].

The second class of addresses is multicast addresses, and these are designed to replace the notion of a broadcast address familiar from IPv4. Packets sent to a multicast address are delivered to all interfaces that are part of the multicast group, and a router will refuse to forward any packets addressed to a multicast group to which the router has no route. This mitigates denial of service type attacks and networks being flooded with broadcast packets [6]. A multicast address of particular importance is the Link Local Multicast Address, FF02::1, which must be queried in order to discover other hosts on a link-local connection before they can be addressed specifically.

The third class of addresses is anycast. A packet sent to an anycast address is forwarded to any of a defined group of addresses that the router deems to be closest according to the routing protocol in use. In other words, a packet sent to an anycast address is guaranteed to be delivered to one in a group.

Other important addresses are the localhost or loopback address (denoted by ::1), which just a self-reference address, and the unspecified address of all zeros (denoted by two colons ::), which is used by interfaces that do not already have an address to indicate the absence of an address. Addresses representing the all-routers multicast groups are FF01:0:0:0:0:0:0:2, FF02:0:0:0:0:0:0:2, and FF05:0:0:0:0:0:0:2, where 1 indicates interface-local scope, 2 is link-local scope, and 5 is site-local scope.

2.2 Related Work

This section covers the topics of steganography and IPv6 as they relate to this thesis. Specifically, steganography in TCP/IP will be discussed along with steganography in images and sound files. Two pieces of the IPv6 protocol, Stateless Address Autoconfiguration (SAA) and the privacy extensions to SAA, are crucial to the work done in this thesis, and they are discussed here as well.

2.2.1 Stegonography

Steganography literally means “covered writing” in Greek, and there are many ways of performing steganography, which may or may not involve the use of a computer. For example, a commonly known form of steganography is performed with lemon juice and paper. A pen dipped in lemon juice is capable of writing an invisible message that can only be made visible again with the application of heat to the paper. The existence of a secret message written in such a fashion would have to be known to the intended recipient ahead of time, and the purpose of this secrecy is to protect the participating trusted parties against any third party that may intercept the message in transit. With the message secretly embedded into the cover medium, the participating parties can be reasonably secure in the knowledge that any third party is unlikely to discover the hidden message. In the case of the lemon juice on paper, the secret message might be written in between lines of a message written in normal ink that is meant to be seen. The ink message could be innocent or deliberately wrong information designed to lead an enemy down a false path of disinformation.

Another analog steganographic scheme would be the one in [7] that discusses Greek historian Herodotus’s account of King Darius of Susa writing a message on the shaved head of a courier and allowing the hair to grow back. Once enough hair grows back to hide the message, the courier could be sent on a cover mission with an outwardly visible purpose concealing the true secret one.

Just as computers have come a long way in helping humans create more secure cryptographic methods, so too have computers enhanced the ability of humans to create increasingly harder to detect steganographic schemes. Often, modern digital steganography is completely undetectable to humans without the aid of a computer even for the intended recipient of a message. Steganography can also be combined with cryptography for an extra layer of security in the case that the existence of the secret becomes known to a third party. Sometimes, steganography is desirable in place of or on top of cryptography for the simple fact that cryptography itself arouses suspicion. The assumption is made that if data is encrypted, it must be something worth hiding and therefore, valuable. This could draw unwanted and unjustified attention. In addition, in [7] Dunbar asserts that “many governments have created laws that either limit the strength of cryptosystems or prohibit them completely.” So, steganography could be used in place of cryptography or to conceal cryptography in such situations.

The following methods are modern steganographic techniques that deal with digital data in various ways. At the most basic level, these techniques rely on “redundant data or noise” [7] contained within the data that is being manipulated.

2.2.1.1 Steganography in Images and Sound Files

Many current steganographic methods rely on the weakness of primary human information sensors such as ears or eyes. Data representing an image or sound file can be modified in subtle ways that cause the resulting file, which is actually different from the original, to appear or sound identical to the original as far as the unaided senses can tell.

Images have plenty of redundant data to work with. Depending on image format, the image can have more of less “bandwidth” for embedding the secret message. Images use numerical values to represent types of light that appear at each pixel, and different image formats allow different ranges of values for each pixel. The more values that are allowed, the more bandwidth that image has, and the easier it is to change a value without the change being detectable to the human eye. For example, if there are 15 shades of green that can be used, then the contrast or difference between two shades will be much more easily detectable than if the same progression of change in colors were spread across 150 shades. This is what provides the redundant data needed for the covert channel.

The same concept exists for digital sound files, which are also lists of data values that represent samples taken from an analog sound wave over time. The more digital samples that are taken from the analog wave, the more accurately the digital map represents the actual sound, and the more bandwidth that exists for the secret message. As [7] notes, the human auditor system is very good at detecting subtle differences in power and frequency of sound, but the weakness of the human ear comes in its inability to accurately distinguish different sounds that occur simultaneously.

A common method for embedding secret data into both of these mediums is least significant bit (LSB) encoding, which embeds a bit of secret data into the LSBs of “blocks” of data. How many bits a “block” consists of is arbitrary (it depends on the medium), but the more bits or information per block, the least likely that modifying the least significant bit of that block will change it’s meaning. In other words, that LSB in a longer block has less weight. Sometimes several LSBs are used [7]. The number of blocks that a file contains multiplied by the bits used per block is the capacity in bits of the covert channel.

Methods such as these of hiding secret data in images and sound files are not ideal for two reasons. One reason is that these types of files have a somewhat predictable pattern of the “noise and redundant data”. Through the aid of computers, statistical analysis can be performed to detect the presence of steganography in the medium, thus defeating the purpose. The second, and probably more practical, reason is that images and sound files are often run through compression algorithms as they are transferred around different locations and users, and these compression algorithms are designed to alter or eliminate the redundant and noisy data that is being used as a covert channel. If the channel is destroyed, the message is not transmitted.

2.2.1.2 Steganography in TCP/IP

Various schemes exist for embedding secret messages into the TCP/IP stack. A few discussed here involve IPv4 and TCP and not IPv6, although some exist [8]. In [8], the authors identify all of the practical places within TCP/IP headers wherein secret data can be hidden. In the IP header, the identified fields are the Type of Service (ToS), IP Identification (IP ID), IP Flags of Do Not Fragment (DF) and More Fragments (MF), IP Fragment Offset, and IP Options fields. In the TCP header [8] there is the TCP Sequence Number (primarily the Initial Sequence Number or ISN) and the TCP Timestamp fields. A program called Covert-TCP identified in [8] and an extension of it called Nushu are capable of embedding secret data into TCP ISNs and IP IDs.

The previous work suffers from one or more basic flaws as far as the use for steganography is concerned. First, the normal values of the fields are well-known, meaning that anything other than this default value is likely to arouse suspicion. Secondly, the fields exhibit predictable characteristics on the network which a steganographic scheme would destroy. Finally, some of the above fields are extremely limited in bandwidth. For example, IP packets rarely contain anything in the IP Options field, so any data in there would be suspect. Similarly, the Type of Service field is not even used except in modified versions of IPv4, and therefore, anything other than a zero in this field would be an anomaly.

The work in [8] discusses TCP ISNs, which are used in the Covert-TCP program as a secret channel that the program simply fills with the secret data. However, as [8] also shows, this field exhibits predictable characteristics because plaintext patterns are easily predictable and would cause the sequence numbers to appear out of the ordinary. Nushu builds upon Covert-TCP by encrypting the data that it embeds into the ISN field, but as [8] notes, even the encrypted data looks out of the ordinary, because ISNs, although meant to be unpredictable, adhere closely enough to a pattern that anything anomalous can be noticed. The main reason for this is that the ISNs are not random. They also claim that there is a flaw in the DES encryption performed by Nushu that adds to the problem [8].

2.2.1.3 Contrast of Current Schemes to This Scheme

In [7] the author quotes Fabien A.P. Petitcolas as saying “in a perfect system, a normal cover should not be distinguishable from a stego-object, neither by a human nor by a computer looking for statistical patterns.” Even though the steganography discussed in Section 2.2.1.1 and Section 2.2.1.2 cannot be detected by the unaided human, often it can be detected by statistical analysis and pattern searching with the aid of a computer. One thing Petitcolas failed to mention, more than likely as an implicitly understood fact, is that in a perfect steganographic system, the covert medium should also be such that it is not easily destructible. The TCP/IP steganography discussed above, for the most part, possesses this trait insofar as it uses fields that remain unmodified in transit, but the same confidence cannot be given to the steganographic techniques involving image and sound files.

In contrast to the methods above, it is conjectured that the method of steganography implemented in this thesis, when used with encryption, will create stego-objects that are indistinguishable from unmodified cover objects, because the unmodified objects, by meticulous design, should not conform to any predictable pattern according to the IPv6 standard. In addition to this benefit, IPv6 source addresses will never be intentionally modified in transit from sender to receiver by any node that is behaving as designed to according to the IPv6 protocol, thus preserving the covert channel.

2.2.2 IPv6

IPv6 has been gaining ground recently as IPv4 addresses move increasingly closer to exhaustion. For example, China, driven by the increasing industrialization of their enormous population, has been moving most quickly in their adoption of IPv6 as the primary internet infrastructure. The protocol has been designed, with the problem of address shortage in mind, to incorporate addresses of 128-bits, which allows for 340,282,366,920,938,463,463,374,607,431,768,211,456 possible addresses. Only the relevant pieces of information pertaining to IPv6 are detailed in this thesis. For a complete description, see the IPv6 specification [3].

2.2.2.1 Stateless Address Autoconfiguration

Stateless Address Autoconfiguration, defined in [4], is one of the processes by which interfaces on an IPv6 network automatically generate their addresses without the need for manual configuration. This process of address acquisition by nodes on a network is performed when complete control over addresses on a network is not desired. The only constraint that this mechanism imposes on addresses is that they be unique on the subnet and are not any of the reserved addresses or in any of the reserved ranges of addresses. This process involves calculation done by the node on which the interface resides as well as information received from a router on the network. The two pieces of information combine to form a complete address for the interface.

The portion of the source address generated by the node that houses the interface is the interface identifier portion of the address, and this will usually be the lower 64 bits derived from a link layer address such as the media access control (MAC) address of an Ethernet card using Modified EUI-64 format. The higher leftmost bits of the address (usually 64) comprise the subnet prefix, also referred to as the network identifier, subnet identifier, network portion, or subnet portion of the address. The router that the interface is connected to supplies the network identifier portion of the address through Router Advertisements, which the router broadcasts on a regular interval. All nodes connected to a link on a network are required to generate a link local address that provides point-to-point connectivity whether or not any router is present. This address is generated automatically as soon as a node is connected to a live link with another node by generating the interface identifier portion of the address and combining it with the constant link-local prefix FE80::/10.

One way the interface identifier portion of an address can be constructed for Stateless Address Autoconfiguration is by creating an IEEE EUI-64 identifier from an IEEE 48-bit MAC address detailed in [5]. In this method, the MAC address is split in half with the 24 bits of the company id on the left and the 24 bits of the vendor supplied id on the right. The hexadecimal value of 0xFFFE is then inserted between the two halves to form a 64-bit identifier. For example, a MAC address of 11:22:30:AA:BB:CC would be split up into the company id of 11:22:30 and the vendor id of AA:BB:CC, and the value FFFE would be inserted in between these to form an interface id that looks like the following in hex: 1122:30FF:FEAA:BBCC. In addition to the insertion of 0xFFFE, the second least significant bit is inverted, so the above would become 1322:30FF:FEAA:BBCC (the second hex digit changed from 1 to 3). To make this clearer, the figures below show the process taken directly from [5]. Figure 2: Standard MAC Address shows a standard MAC address with the company bits defined by the letter ‘c’ and the vendor specific bits defined by the letter ‘m’. The ‘0’ character indicates global scope and is known as the universal/local bit, while the ‘g’ indicates the individual/group bit [5]. Each letter represents a single bit.

```
|0              1|1              3|3              4|
|0              5|6              1|2              7|
+----------------+----------------+----------------+
|cccccc0gcccccccc|ccccccccmmmmmmmm|mmmmmmmmmmmmmmmm|
+----------------+----------------+----------------+
```

Figure 2: Standard MAC Address

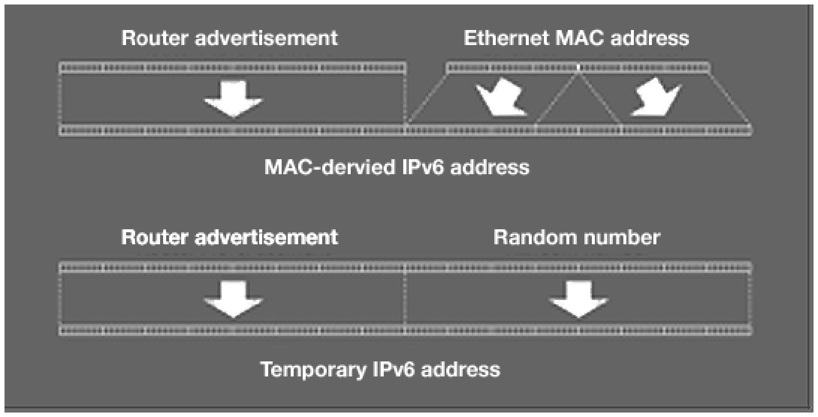
Figure 3: EUI-64 Identifier below shows what the interface identifier derived from the above MAC looks like when the FF FE (or binary 11111111 11111110) value is inserted in the middle transforming the identifier in Modified EUI-64 format. Note that the universal/local bit is inverted as part of the process to indicate universal scope, but could also be set to a ‘0’ to indicate local scope. This bit is to allow development of future protocols that can take advantage of the universal scoped interface identifiers, according

to [5], but nodes are not required to verify that an identifier with this bit set is actually unique.

```
|0              1|1              3|3              4|4              6|
|0              5|6              1|2              7|8              3|
+----------------+----------------+----------------+----------------+
|cccccc1gcccccccc|cccccccc11111111|11111110mmmmmmmm|mmmmmmmmmmmmmmmm|
+----------------+----------------+----------------+----------------+
```

Figure 3: EUI-64 Identifier

The top of Figure 4: Stateless Address Autoconfiguration Process below shows the process graphically. The bottom half of the figure shows the random number method used for the privacy extensions discussed in Section 2.2.2.2.



From http://media.arstechnica.com/articles/paedia/IPv6.media/autoconfig.png

Figure 4: Stateless Address Autoconfiguration Process

The steps of the Autoconfiguration process are as follows:

1) A host generates its link-local address.
2) The host then attempts to verify if any other host on the link is using the generated address in a process called Duplicate Address Detection by sending Neighbor

Solicitation packets to the generated address. If another host is using the address, it will respond to the query of the asking host with a Neighbor Advertisement message. The time that a node waits for a Neighbor Advertisement response and the number of retransmissions of Neighbor Solicitations is configurable by network administrators.

3) If the address is not unique, then Autoconfiguration cannot continue.
4) If the address is unique, then it is assigned to the generating host.

After the above steps are performed, the next part of the process involved with generating addresses with scope other than link-local are specific to end hosts and do not apply to routers. Hosts will look for Router Advertisements that contain the network prefix to use when generating a global address along with the lifetime values of the address. Lifetime values specify how long an address is valid or preferred before becoming invalid or deprecated, respectively. The different lifetime values are defined with greater detail in [4]. Alternatively, hosts do not have to wait around for a Router Advertisement to be broadcast. They have the option to send out Router Solicitation packets to the all-routers multicast group, as mentioned in Section 2.1.2.2 above, indicating that they want a Router Advertisement to be broadcast by the router. The host subsequently performs Duplicate Address Detection in the same manner as described above after it generates the global address using the same interface identifier as in the above process with the subnet prefix supplied in the Router Advertisement.

2.2.2.2 Privacy Extensions to Stateless Address Autoconfiguration

The Privacy Extensions to Stateless Address Autoconfiguration came out of a legitimate concern for privacy of the individuals using hosts connected to IPv6 networks. Specifically, the prospect of tracking user activity across the network over time is a very real possibility when a universally unique identifier such as a MAC address is used by Stateless Address Autoconfiguration to derive the interface identifier portion of a node's address. If a portion of the address of an interface is always the same and is universally unique, then activities of the user using the interface can be tracked and correlated no matter where the user is geographically, whether the user connects to different subnets, or the time span between consecutive uses of the network. In [9], T. Narten et al. were concerned with this problem pertaining to user's privacy on the network and proposed a mechanism to get around it. This new mechanism is to allow nodes generating addresses for their interfaces to use a random number instead of converting a MAC address to an EUI-64 formatted identifier. The process is illustrated in contrast to the MAC derived process in Figure 4: Stateless Address Autoconfiguration Process above. All other aspects of Stateless Address Autoconfiguration remain the same, except for the process by which nodes generate the interface identifier portion of their address. In addition, for nodes using this random identifier portion, a temporary preferred lifetime (or temp-prefered-lft as found in the system file) specific to these temporary addresses is specified by the administrator of the node using the privacy extensions. The temporary preferred lifetime specifies when the address for the node is to be changed to the next random number generated by the node. This value defaults to 86400 seconds or 24 hours, but can be changed by the system administrator. The exact process by which this occurs is unimportant for this thesis, but can be found in [9]. It is, however, relevant to the time interval that the program for this Thesis uses to send out messages on as described in Section 4.2.1.

The privacy extensions solve the privacy and activity correlation problem, but they also introduce the possibility of the covert channel within source addresses of IPv6 packets by allowing the interface identifier to be generated with a random number. Because the interface identifier can be expected to be random, it cannot be expected to conform to any predictable value or statistical sequence. In other words, if a third party inspects the source address field of a packet, it cannot easily be said to appear out of the ordinary, because any value is fair game. This is key to the work in this thesis.

3. APPROACH

3.1 High Level Design

The approach taken by this thesis to embed messages into IPv6 sources address uses two methods, each of which has its advantages and its disadvantages depending on the application. The first way that messages are embedded into the source address fields of packets is through the media access control (MAC) address, which is detailed in Section 3.2. The MAC address is like a serial number for a network interface. The second method is to directly create packets using a native Perl framework built for that purpose, and this method is described in Section 3.3. The program written for this thesis is able to decode messages from packets that have been encoded using either of the described methods, and, in fact, the decoding process depends upon the method of encoding used.

For both methods of embedding the message, the bits of the ASCII characters that make up the message are used as the input to the steganographic medium, henceforth referred to as "entity," which means either a MAC address or the interface identifier of an IPv6 address. For example, the bits for the message 'invade normandy tomorrow. dawn' are represented in hexadecimal notation as:

```
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e
```

In this representation, each two hex characters represent a byte of the message for a total of 30 bytes. This format of the message represents the raw bytes that underlie the human readable ASCII characters, and these bytes are used to create either entity specified as the steganographic medium, because both of these entities can be represented by hex digits[2]. One can see this correspondence quite easily by looking up 0x69 and 0x6e on an ASCII chart and noticing that these two numbers represent the letters ‘i’ and ‘n’, respectively. For the purpose of decoding the messages encoded using one of two different schemes (see Section 3.3 below), the length of the original message in bytes is calculated and appended to the beginning of the message between ‘<’ and ‘>’ characters before it is actually encoded to the interface identifiers. So with the byte length information appended, the above 30-byte message actually becomes:

```
<30>invade normandy tomorrow. dawn
```

For MAC encoding, the process is slightly different. The string ‘<eNd>’ is appended to the end of the message creating the string below:

```
invade normandy tomorrow. dawn<eNd>
```

This message is then either encrypted with the Blowfish block cipher algorithm, or the original text is used directly to create the series of entities that will hold the message. In either case the hex representation of the text or cipher text is used to create the list of the entities specified, whether they are MAC addresses or interface identifiers.

If the number of bytes available in the message cannot be divided evenly into the desired medium as six bytes each for a normal MAC address (five bytes for the short version) or eight bytes each for IPv6 interface identifiers, random bytes of data are created and concatenated to the remaining message bytes in order to create another

[2] Of course, this hex representation is just a way of making things more readable on the human level. What is really going on is that the binary digits that represent the characters in the message are being directly used to create either MAC addresses or IPv6 source addresses.

complete entity. At most, five random bytes are created for normal MAC encoding (four bytes for short encoding) and seven bytes are created for interface identifiers in the worst cases, because in all these cases, there is one leftover byte for the message that will not fit into the previous entity. In the case of the above message, there are 34 total bytes when the ‘<30>’ byte-length information string is appended to the original message. When divided by eight for IPv6 interface identifier encoding, there are two bytes left over, so six random bytes must be created to build an additional eight-byte interface identifier. However, for MAC addresses, the message ends up containing 35 total bytes with the ‘<eNd>’ string included. For long MAC encoding, 6 bytes per MAC are used, and this divides into five full MACs with five bytes leftover. One extra random byte must be generated to complete the last MAC. However, when divided by five for the alternative short MAC encoding, there are no bytes left over, so no random bytes must be created to tack on to the end of the message.

3.2 Encoding Messages through MAC Addresses (Passive Injection)

MAC addresses are composed of 48 bits (six 8-bit bytes) and can be represented by six octets (two hex digits each) separated by colons such as in the following:

```
AA:BB:CC:11:22:33
```

Using the MAC address method, a list of MAC addresses is created from the bytes in the original message. However, MAC addresses have the constraint that the least significant bit of the first octet in the address be zero unless it is a multicast MAC address. In other words, it must be even for normal MAC addresses belonging to a single interface. This produces difficulty if one wishes to be able to encode an arbitrary message into the medium, because the message may have to be changed to create a legal MAC address.

The solution taken in this thesis is to allow the encoding into MAC addresses in two different ways. The first way is described in Section 3.2.1, while the second is described in Section 3.2.2. In both cases, the method of sending out the message is call passive injection because of the passive manner in which it works. As described above, when a network host has not elected to use the privacy extensions of IPv6, the interface identifier portion of IPv6 addresses is created using the MAC address of the interface on the IPv6 network. For this reason, when the MAC address of an interface is changed, the interface identifier portion of the interface's IPv6 address is automatically changed. As long as the MAC is set to one of the entities representing a piece of the secret message, this piece of the message will be embedded implicitly into the interface identifier portion of the source address of all packets leaving the interface of the host on the network. This happens regardless of the details, such as source and destination ports and destination addresses, that the application or higher level protocol uses as long as the interface containing the piece of the message as its MAC address is used.

3.2.1 Long MAC Encoding

The first way is to encode the message into all eight octets of the MAC and change the least significant bit of the first octet to a zero, if it is a one. The problem with this approach is that the receiver of the message has no way of knowing whether this bit was changed and that the message was altered as a result. This is different from the EUI-64 conversion process described in Section 2.2.2.1 that simply inverts the seventh bit of the first octet regardless of whether it is a one or a zero, and decoding based on the EUI-64 bit process is just a matter of always flipping that seventh bit back. If this MAC encoding method is used, the receiver must decode two messages for every one IPv6 address they receive that is part of the message stream. It is then solely up to the discretion of the user to decide which decoding is correct. In some cases, such as simple text messages it is easy to tell which is the correct decoding, but in some cases, it is not so easy. An example of legal MAC addresses encoded from the message ‘invade normandy tomorrow. dawn<eNd>’ using this method is below. Note that the actual bytes of this message in hex (divided into 6-byte pieces for clarity) are:

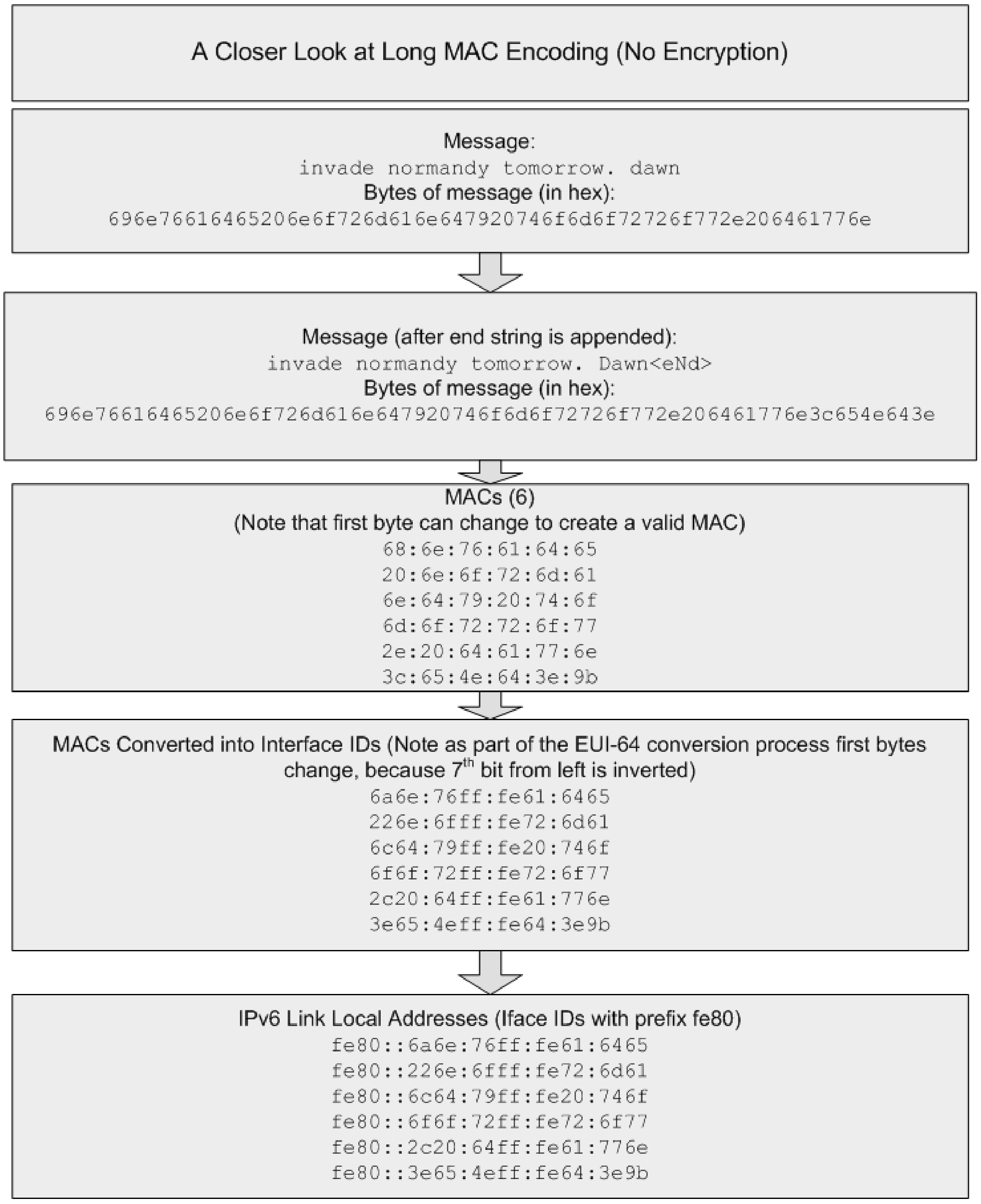
```
696e76616465
206e6f726d61
6e647920746f
6d6f72726f77
2e206461776e
3c654e643e
```

These bytes turn into the MAC addresses:

```
68:6e:76:61:64:65
20:6e:6f:72:6d:61
6e:64:79:20:74:6f
6d:6f:72:72:6f:77
2e:20:64:61:77:6e
3c:65:4e:64:3e:9b
```

As can be seen, it takes six MAC addresses to fully encapsulate the 35-byte message, with the appended <eNd> string. It is important to note that because no encryption was used, each octet of each MAC address represents an ASCII character of the message, not including the ending ‘9b’ byte on the end of the last MAC, which is just random data created to make a full six-byte MAC address. In addition, it can be seen that the first byte of the message 0x69 was changed to 0x68 to be compatible with standard MAC address form. If it had stayed 0x69 it would have resulted in a multicast MAC address, which would be invalid. The first octets of the other MACs were already even, and they remain unchanged.

After this list of MAC address is created from the message, the interface specified by the user will have its MAC changed to each of these new MACs on a time interval also specified by the user. When the MAC address of an interface connected to an IPv6 network is changed, and the node that the interface is connected to has not elected to use the privacy extensions of IPv6, the interface identifier portion of that interface's IPv6 address is then changed automatically by the operating system. This is done in the Modified EUI-64 format as described in Section 2.2.2.1. Because of this process, the message embedded into the MAC address is embedded into the interface identifier of the IPv6 source address. The figure below depicts this process.


A Closer Look at Long MAC Encoding (No Encryption)

Message:

```
invade normandy tomorrow. dawn
```

Bytes of message (in hex):

```
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e
```

Message (after end string is appended):

```
invade normandy tomorrow. Dawn<eNd>
```

Bytes of message (in hex):

```
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e3c654e643e
```

MACs (6)
(Note that first byte can change to create a valid MAC)

```
68:6e:76:61:64:65
20:6e:6f:72:6d:61
6e:64:79:20:74:6f
6d:6f:72:72:6f:77
2e:20:64:61:77:6e
3c:65:4e:64:3e:9b
```

MACs Converted into Interface IDs (Note as part of the EUI-64 conversion process first bytes change, because 7th bit from left is inverted)

```
6a6e:76ff:fe61:6465
226e:6fff:fe72:6d61
6c64:79ff:fe20:746f
6f6f:72ff:fe72:6f77
2c20:64ff:fe61:776e
3e65:4eff:fe64:3e9b
```

IPv6 Link Local Addresses (Iface IDs with prefix fe80)

```
fe80::6a6e:76ff:fe61:6465
fe80::226e:6fff:fe72:6d61
fe80::6c64:79ff:fe20:746f
fe80::6f6f:72ff:fe72:6f77
fe80::2c20:64ff:fe61:776e
fe80::3e65:4eff:fe64:3e9b
```

Figure 5: Long MAC Encoding Process

3.2.2 Short MAC Encoding

An alternative method of encoding the secret message to MAC addresses is to do so using only the lower five bytes of the MAC, thus avoiding the problem of having to force the first octet to be even. Obviously, this method more often than not requires more MAC addresses to be created to contain the entire message. Accordingly, this method has less bandwidth than the previous one, but it is more straightforward and easier decode.

For this method, a clean octet, which is defined as an octet whose least significant bit is zero, is created for the first octet of every MAC that contains a piece of the message. For the same message "invade normandy tomorrow, dawn<eNd>" the bytes in hex remain the same as above, but this time the bytes are presented in 5-byte blocks for clarity:

```
696e766164
65206e6f72
6d616e6479
20746f6d6f
72726f772e
206461776e
3c654e643e
```

The list of MAC addresses encoded using this method is below:

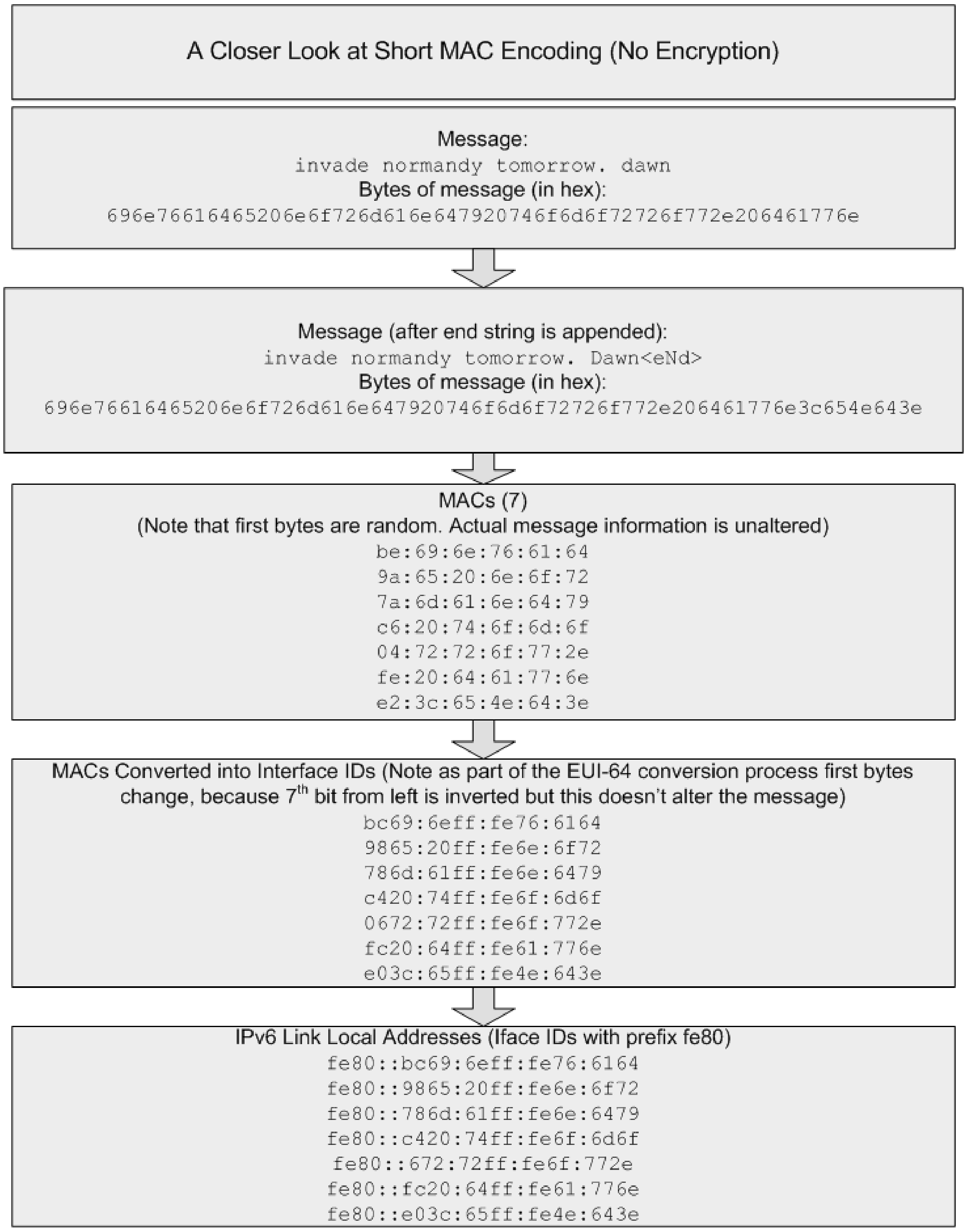
```
be:69:6e:76:61:64
9a:65:20:6e:6f:72
7a:6d:61:6e:64:79
c6:20:74:6f:6d:6f
04:72:72:6f:77:2e
```

```
fe:20:64:61:77:6e
e2:3c:65:4e:64:3e
```

The first octet of each MAC is just a randomly generated number. It is notable that using this scheme requires one more MAC address to be created than the normal six-octet encoding in order to encapsulate the entire message. This can be significant depending on the time interval between MAC changes that one uses to send out messages. The choice of MAC encodings is up to the user of the supplementary program and depends on their specific circumstances, namely whether it is more important to have an easier decoding process, or whether it is more important to get a message out quickly.

After the list of MACs has been generated, the target network interface has its MAC changed to be each of these on the predefined time interval set up by the user. In the same manner as in the Long MAC Encoding, the IPv6 source address is derived automatically by the operating system as the Modified EUI-64 format of the MAC.

Figure 6 below provides a more detailed look.



A Closer Look at Short MAC Encoding (No Encryption)

Message:

```
invade normandy tomorrow. dawn
```

Bytes of message (in hex):

```
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e
```

Message (after end string is appended):

```
invade normandy tomorrow. Dawn<eNd>
```

Bytes of message (in hex):

```
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e3c654e643e
```

MACs (7)

(Note that first bytes are random. Actual message information is unaltered)

```
be:69:6e:76:61:64
9a:65:20:6e:6f:72
7a:6d:61:6e:64:79
c6:20:74:6f:6d:6f
04:72:72:6f:77:2e
fe:20:64:61:77:6e
e2:3c:65:4e:64:3e
```

MACs Converted into Interface IDs (Note as part of the EUI-64 conversion process first bytes change, because 7th bit from left is inverted but this doesn't alter the message)

```
bc69:6eff:fe76:6164
9865:20ff:fe6e:6f72
786d:61ff:fe6e:6479
c420:74ff:fe6f:6d6f
0672:72ff:fe6f:772e
fc20:64ff:fe61:776e
e03c:65ff:fe4e:643e
```

IPv6 Link Local Addresses (Iface IDs with prefix fe80)

```
fe80::bc69:6eff:fe76:6164
fe80::9865:20ff:fe6e:6f72
fe80::786d:61ff:fe6e:6479
fe80::c420:74ff:fe6f:6d6f
fe80::672:72ff:fe6f:772e
fe80::fc20:64ff:fe61:776e
fe80::e03c:65ff:fe4e:643e
```

Figure 6: Short MAC Encoding Process

3.3 Encoding Messages through Packet Creation (Active Injection)

Another method used to send messages in the source-address covert channel of IPv6 is called Active Injection. This method is referred to as active because packets with spoofed source addresses containing the secret message are created by the program and injected into the network solely for the purpose of sending the secret message, whereas in the passive mode, no packets are actually created by the program at all. However, in the passive mode, packets are created by other applications that simply use the IPv6 source addresses with the embedded message set by the stego program for an interface. In active mode, the actual address of the interface connected to the network never changes at all.

IPv6 packets are constructed by the program in pieces from the link-layer to the application layer, with most properties of these packets configurable by the user, and injected into the network on a time interval set by the user. An example of a property not directly configurable by the user in these packets is the IP source address, which will obviously be set to a piece of the secret message specified by the user. Other properties, however, such as source and destination port, destination address, data for the layer-seven data portion, and much more can be specified by the user in many different configurations that can help to make the packet look more legitimate on the network the user is connected to.

Using this method, instead of creating a series of MAC addresses to embed the messages into and then setting the target interface's MAC address to these new MACs in series, the supplemental program will encode the message (or cipher-text if encryption is used) directly into a series of IPv6 interface identifiers. These interface identifiers, as discussed in Section 2.2.2.1, are 64 bits long as opposed to the 48 bits of MAC addresses, so it will on average take less of the entity, into which pieces of the message are embedded, to send out the full message. The network portion of the address (the first 64-bits), also discussed in Section 2.2.2.1, can be derived from pre-existing prefixes of interfaces on the node that are connected to the IPv6 network.

It is important to note, though obvious, that if a reply such as a TCP acknowledgement is sent out by the receiver of these packets with spoofed source addresses, routers on the network will attempt to route the reply from the receiver to the spoofed source address, or they will drop the reply if they have no route to the spoofed address. In any case, it will be impossible for the sender of the secret message to receive any reply from the receiver unless the receiver ahead of time knows the sender's true address.

As an example, the message '<30>invade normandy tomorrow. dawn' with length information appended to the front is represented by the following hex divided into 8-byte chunks for clarity:

```
3c33303e696e7661

6465206e6f726d61

6e647920746f6d6f

72726f772e206461

776e
```

This information becomes the following interface IDs:

```
3c33:303e:696e:7661

6465:206e:6f72:6d61

6e64:7920:746f:6d6f

7272:6f77:2e20:6461

776e:1627:c77d:834c
```

A diagram of this process is presented below in Figure 7: Direct Encoding Process.

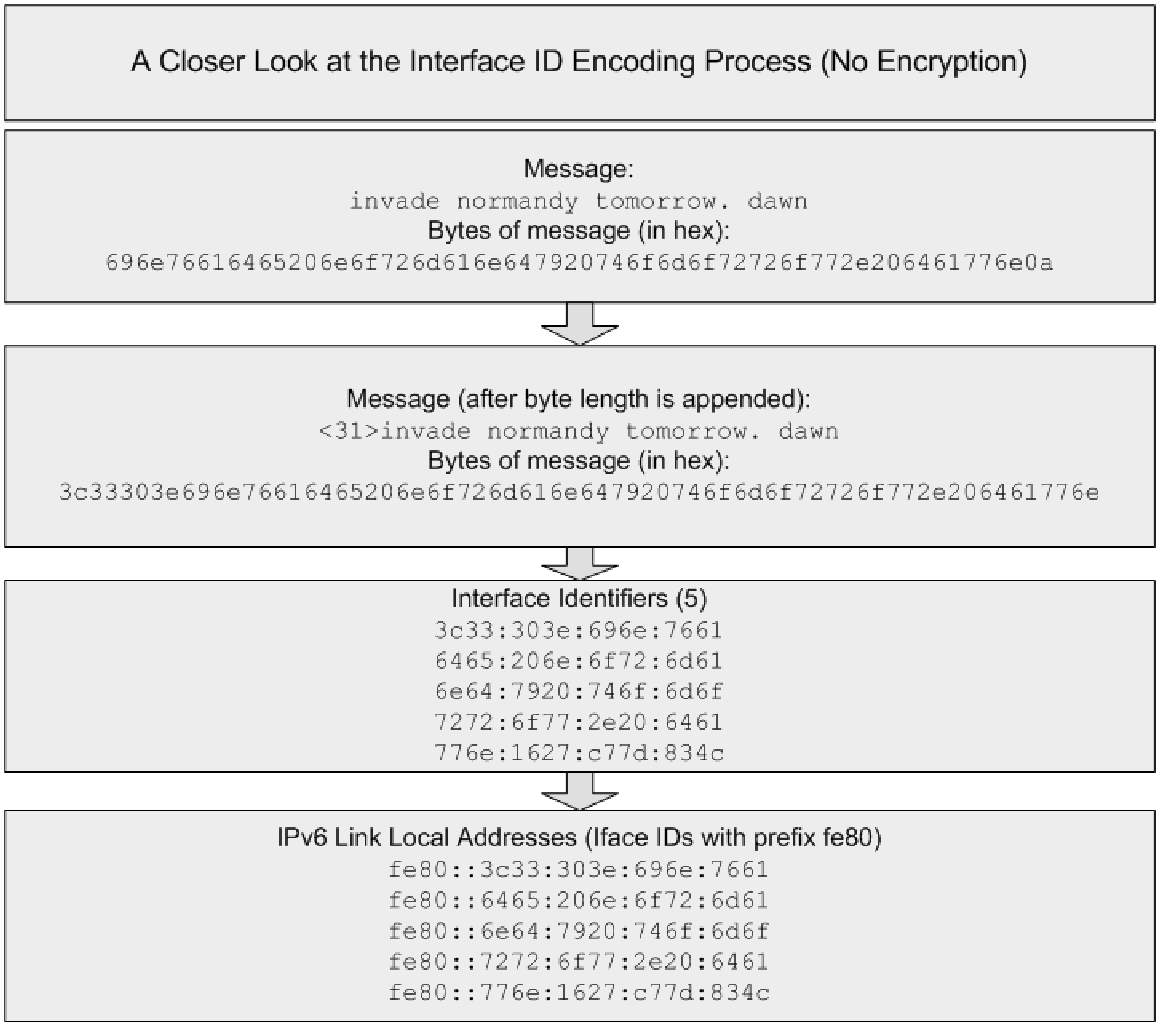

A Closer Look at the Interface ID Encoding Process (No Encryption)
Message:
invade normandy tomorrow. dawn
Bytes of message (in hex):
696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e0a
Message (after byte length is appended):
<31>invade normandy tomorrow. dawn
Bytes of message (in hex):
3c33303e696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e
Interface Identifiers (5)
3c33:303e:696e:7661
6465:206e:6f72:6d61
6e64:7920:746f:6d6f
7272:6f77:2e20:6461
776e:1627:c77d:834c
IPv6 Link Local Addresses (Iface IDs with prefix fe80)
fe80::3c33:303e:696e:7661
fe80::6465:206e:6f72:6d61
fe80::6e64:7920:746f:6d6f
fe80::7272:6f77:2e20:6461
fe80::776e:1627:c77d:834c

Figure 7: Direct Encoding Process

3.4 The Decoding Process

A receiver can be configured to sniff a local interface and look for packets with certain properties. These properties can be set up ahead of time by the two parties to make it easier to sniff only relevant packets that contain the secret message. For example, one might look only for packets with a predetermined source port or source MAC address. The receiver can also choose to sniff indefinitely, or to sniff only until a complete single message has been received. When relevant packets are received, the program will attempt to decode them based on how it is told that the packets are encoded.

The decoding process depends heavily on the type of encoding used. If the encoding process used is Long MAC Encoding, described in Section 3.2.1, then the receiving program will extract the interface ID portion of the address and eliminate the middle octets with hexadecimal values 0xFF and 0xFE. These values would have been inserted during the process of the originating node constructing the IPv6 interface ID in Modified EUI-64 Format, which is described in Section 2.2.2.1. The seventh bit would also need to be flipped. Then, two possibilities for the decoded message must be created by decoding the bits resulting from removing the 0xFF and 0xFE directly and flipping the seventh bit, and again by flipping the least significant (or eighth) bit of the first octet to its opposite and decoding that combination. For example, if the first octet after reversing the EUI-64 process is 0x72, represented in binary as 0111 0010, this would decode to an ‘r’ character. However, because the binary 0 in the eighth place could have been set that way to create a valid MAC, the original message could have actually been 0111 0011 in binary or 0x73 in hex, which would subsequently decode to an ‘s’ character in ASCII. It is impossible for the program to tell which of these two decoded results is correct, because it will not know if the original message had a value of one in the eighth position, or if it was flipped to a one to create a valid MAC address. As pieces of the message are generated, an increasingly complex tree of possible paths for the actual message is generated, and it is left up to the user to decide which is correct. If encryption is used, then the program will attempt to decrypt the two possibilities leaving an obvious incorrect decryption that would probably appear as gibberish. Therefore, encryption is recommended for this encoding and decoding method.

The next possible encoding is Short MAC Encoding. This process is more straightforward than decoding the Long MAC Encoded messages, because it does not bother with the first octet of the extracted message strings. By ignoring this first octet, it is unnecessary to create a tree of possible message paths, and the message can be decoded directly after removing the 0xFF and 0xFE bits from the middle of the string, if no encryption was used. If the message is encrypted, there will be the extra step of decrypting the cipher-text into plaintext between removing the 0xFF and 0xFE bits and subsequently decoding the hexadecimal digits into ASCII.

The final way that a message is decoded is the simplest. If active injection mode is used, then the messages will be embedded straight into the lower 64 bits of the source addresses of the relevant sniffed packets. There is no need to extract the 0xFF and 0xFE strings as in the other two decoding methods, because messages sent using active injection are not based on the MAC addresses, and, therefore, these messages never undergo Modified EUI-64 format encoding. Rather, these messages were embedded directly into the interface ID portion of the source addresses. If the message is encrypted, then there is one extra step of decrypting the cipher-text after pulling it from the source address. After recovering the plaintext, the program decodes the hexadecimal representation of the ASCII into actual characters.

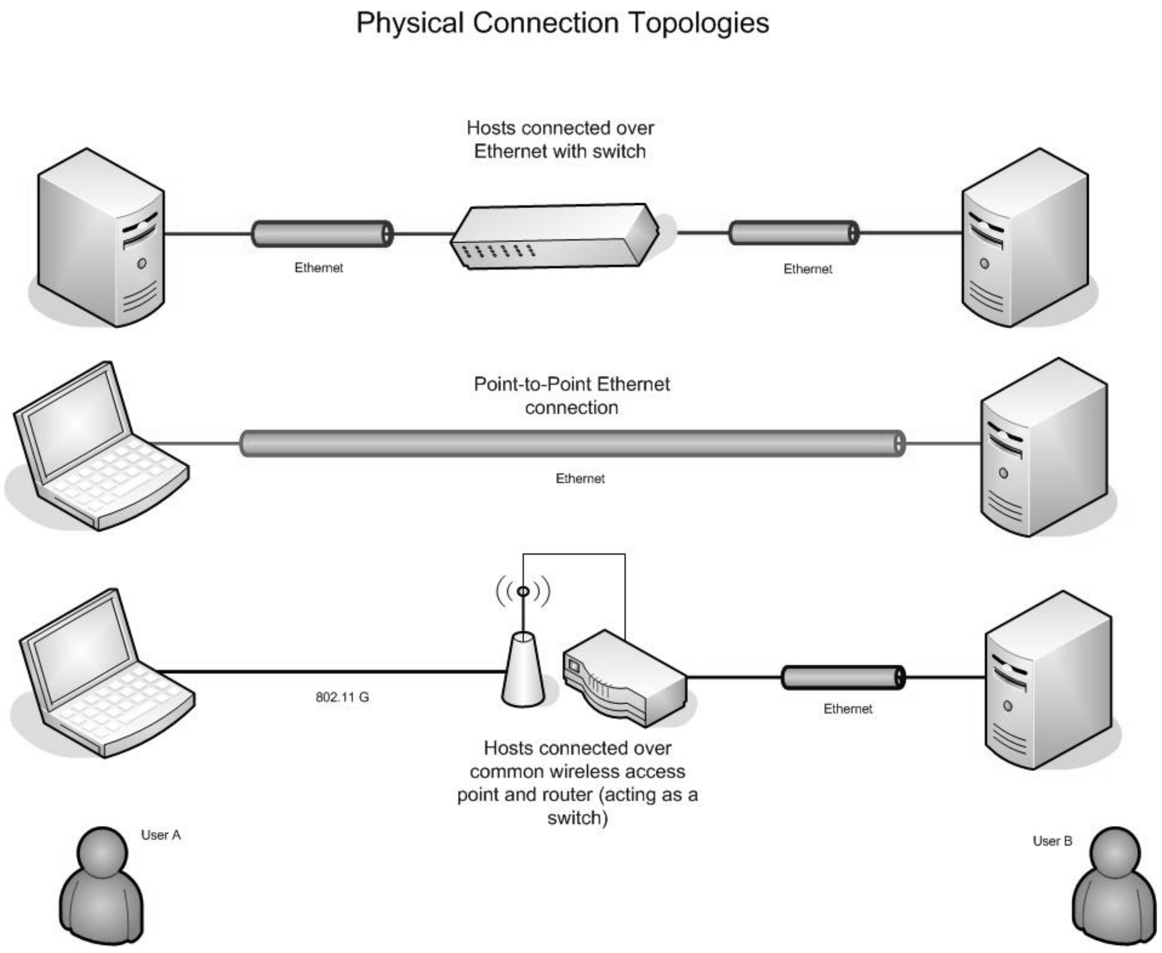
4. IMPLEMENTATION

4.1 Platform and Environment

The program written for this thesis was implemented in the Perl programming language version 5.8.8 and tested on the Slackware 12, Backtrack 2 (a derivation of Slackware), and Ubuntu 8.04 distributions of the Linux operating system. Some of the system calls that the program uses are Linux specific. No testing was performed on the Microsoft Windows operating system. The Net::Packet, Crypt::CBC (Chain Block Cipher), and Crypt::Blowfish Perl modules, as well as the dependencies for these modules must be installed for the program to run correctly with full functionality. The Net::Packet module is a library that allows for the arbitrary creation and manipulation of network packets from the Ethernet layer through the network layer (including IPv4 and IPv6) and transport layer to the application layer, according to the OSI Model. The Crypt::CBC and Crypt::Blowfish modules work in conjunction to allow for encryption of the message using the Blowfish encryption algorithm.

The Ubuntu and Backtrack 2 distributions were run from a Dell Laptop with an Intel Centrino dual-core processor at 1.66GHz per core and 1GB of RAM in a dual-boot configuration. The Slackware 12 operating system was run on two lab desktop servers with 512 MB of RAM each, one with a 2.8 GHz Pentium 4 and the other running a 1.5 GHz Pentium 4. Slackware 12 was also run on one home desktop server with 512 MB of RAM and dual 800 MHz Pentium 3 processors. Logical point-to-point tests using link-local IPv6 addresses (prefix fe80) for connectivity were used in all testing scenarios. On the link-layer, hosts were connected directly with either ad-hoc wireless connections or direct Ethernet cord connections, or they were connected via Ethernet over a Nortel

460/470 series Ethernet switch that passively moved information between connected interfaces. No intermediate nodes functioning as routers were set up for testing. Figure 8: Physical Connection Topologies shows some of the physical network topologies that were used during testing.



Physical Connection Topologies
Hosts connected over Ethernet with switch
Ethernet
Ethernet
Point-to-Point Ethernet connection
Ethernet
802.11 G
Ethernet
Hosts connected over common wireless access point and router (acting as a switch)
User A
User B

Figure 8: Physical Connection Topologies

4.2 Running the Program

The program, named 'm2a.pl' (for Message To Address), is run from the command line of a Linux console, and all user configurable options are specified either in a configuration file or via command line arguments. A complete list of all configurable options with a description of each is included in the appendix for quick reference. The default path for the configuration file is the '/etc/m2a.conf'. The simplest way of running the program is to specify command line arguments, but when a given configuration is expected to be used many times, the configuration file can save considerable time. The following sections discuss different ways that the program functions and give an overview of a few setups that are anticipated to be the most common configurations.

The two primary operating modes for the program are Message Mode and Decode Mode. Message mode is used by the sender of the message, while decode mode is used by the receiver. Depending on which primary operating mode is selected, the user can specify a multitude of sub-options that control the operation of the m2a.pl program, and these are useful in different circumstances depending on exactly what the user is trying to accomplish and the specific constraints of different scenarios.

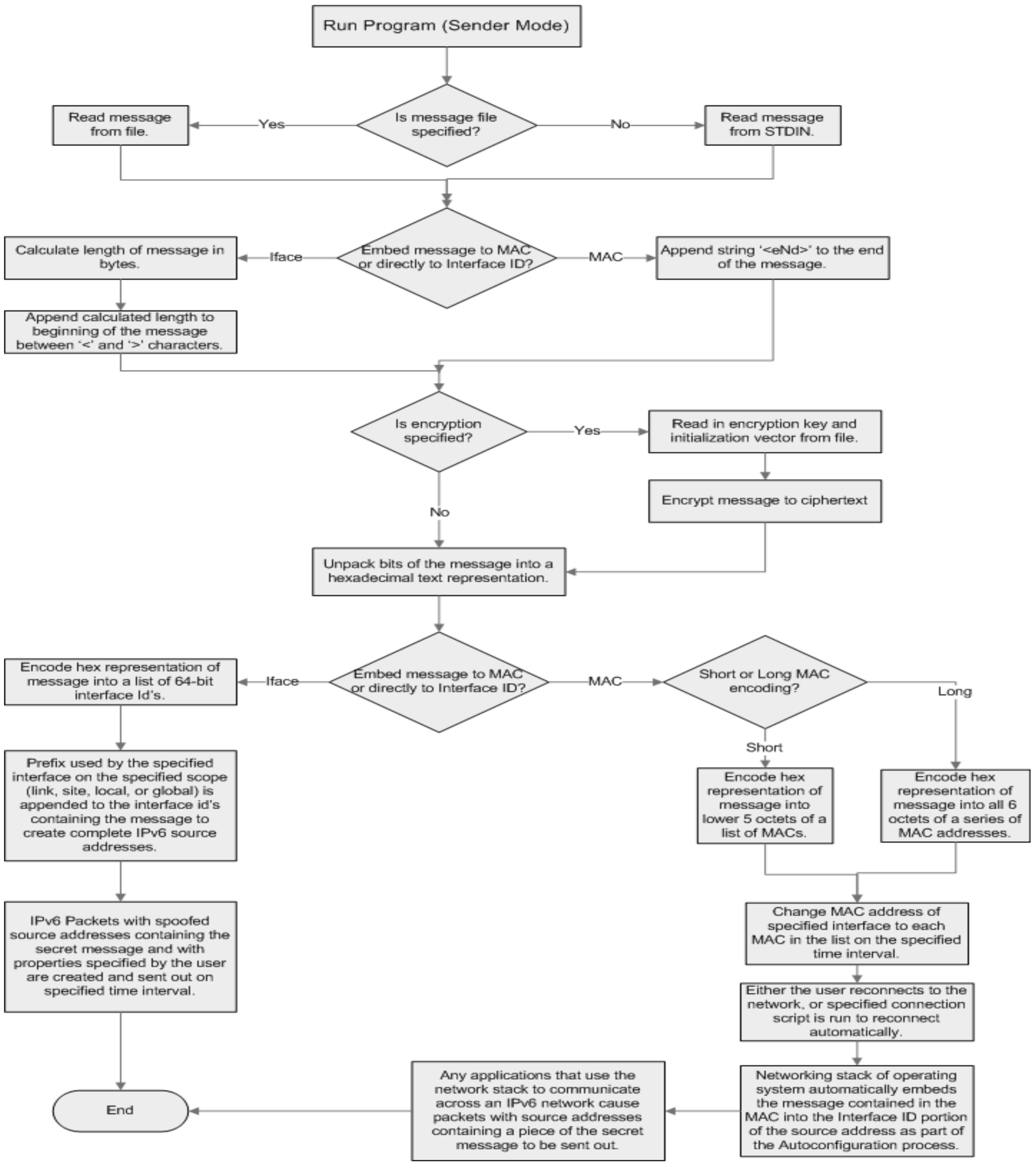
Both primary operation modes require the user to specify a network interface either to send message out on, or to sniff packets from. If no interface is specified, then m2a.pl will attempt to pull an interface out of the /proc/net/if_inet6 file, but this is not recommended. Because encryption of the message is an option, both of the primary modes require the user to specify whether encryption is to be used, and, if so, to specify a file containing the 56-byte (448-bit) encryption key and the 8-byte (64-bit) initialization vector to be used. A supplementary program to generate encryption keys and initialization vectors is included named 'genKey.pl'.

The encryption module used by the program, Crypt::CBC, requires that blocks of 8 bytes (the size of the initialization vector) be encrypted at a time, and this has some serious implications for the implementation of the MAC Encode methods. For the

Interface ID encoding mode, each unique piece of the message is 8 bytes (64 bits) long, because it occupies the full 64-bit interface identifier portion of the IP address. These fragments of the message can be decrypted, piece-by-piece, as it they are received with no extra work involved. However, for MAC Encoding mode, each individual piece of the message that the receiver obtains is only either 5 bytes (40 bits) or 6 bytes (48 bits) long depending on whether Short or Long encoding is used, respectively. A buffer, therefore, is established that only attempts to decrypt a piece of the message after 8 bytes of the message are received. In the case of either type of MAC encoding, this requires at least two changes of MAC address on the part of the sender before enough information is gathered in the buffer for a piece to be decrypted.

4.2.1 Message Mode

Message mode is the primary mode for the sender of the message. It is further divided into Short and Long MAC Encode modes (both forms of Passive Injection) and Interface ID Encode Mode (Active Injection). The basics of these different modes are described in Section 3. Both passive and active modes will send out unique pieces of the message on an interval specified by the user. If no interval is specified, then the temporary preferred lifetime (tmp_prefered_lifetime) value for the interface found in the file /proc/sys/net/ipv6/conf/[interface]/tmp_prefered_lifetime is used for the time interval to send a new message piece out on. If encryption is specified, then the file containing the key and initialization vector must be supplied, and the message will be encrypted after being concatenated with a certain type of metadata to facilitate decoding. The type of metadata added to the original message depends on the type of encoding used as is explained below in Sections 4.2.1.1 and 4.2.1.2. Figure 9: Message Mode Process Flow at the end of this section shows a high-level visual diagram of the encoding process performed by the sender of the message.



Run Program (Sender Mode)
Is message file specified?
Yes
No
Read message from file.
Read message from STDIN.
Embed message to MAC or directly to Interface ID?
Iface
MAC
Calculate length of message in bytes.
Append calculated length to beginning of the message between '<' and '>' characters.
Append string '<eNd>' to the end of the message.
Is encryption specified?
Yes
No
Read in encryption key and initialization vector from file.
Encrypt message to ciphertext
Unpack bits of the message into a hexadecimal text representation.
Embed message to MAC or directly to Interface ID?
Iface
MAC
Encode hex representation of message into a list of 64-bit interface Id's.
Short or Long MAC encoding?
Short
Long
Prefix used by the specified interface on the specified scope (link, site, local, or global) is appended to the interface id's containing the message to create complete IPv6 source addresses.
Encode hex representation of message into lower 5 octets of a list of MACs.
Encode hex representation of message into all 6 octets of a series of MAC addresses.
IPv6 Packets with spoofed source addresses containing the secret message and with properties specified by the user are created and sent out on specified time interval.
Change MAC address of specified interface to each MAC in the list on the specified time interval.
Either the user reconnects to the network, or specified connection script is run to reconnect automatically.
Networking stack of operating system automatically embeds the message contained in the MAC into the Interface ID portion of the source address as part of the Autoconfiguration process.
Any applications that use the network stack to communicate across an IPv6 network cause packets with source addresses containing a piece of the secret message to be sent out.
End

Figure 9: Message Mode Process Flow

4.2.1.1 MAC Encoding

If Long or Short MAC Encode Mode is used, the program will not explicitly send out any packets. It is assumed that the user will be using applications that will use the IPv6 layer to communicate across the network, and if this occurs, the message will be sent out as part of the packet stream that the application uses for its purposes, whatever those may be. Individual pieces of the message in this mode will likely be sent out multiple times, so the Decode mode must compensate for this by recognizing repeats.

Before the message is encrypted (if specified) and encoded into the MAC addresses, the string ‘<eNd>’ is appended to the end of the original message to facilitate decoding by signaling when the last piece of the message has been received. It is thought to be highly unlikely for the exact string ‘<eNd>’ to occur in a message naturally, but the user should be aware that if this string exists within the actual message, it would cause a premature end to the decoding of the message.

For Long MAC encoding, the MAC addresses created from the message are run through a cleanup routine that makes sure the first octet of each MAC is even (i.e. the least significant bit is set to zero). This has the potential to alter the message and the decoding process must account for this as explained below.

For Short MAC encoding, the message is embedded into the lower five bytes of a MAC address, and a random even byte is generated for the first octet of each prospective MAC address. This method, as noted in Section 3.2.2 will potentially require more MAC addresses to be created, and more intervals will be required to get the entire message out, but it avoids the problem of having to alter the actual message being transmitted.

Different network setups for each end user across multiple physical locations will likely vary greatly. For example, default gateways will be different, and nodes wishing to connect to wireless networks will have different access point ESSIDs along with their associated wireless security protocols and keys for WEP, WPA, or lack thereof. When MAC encoding is used, the network interface must be brought down and disconnected from the network in order to change the MAC address. Because of this process, the user will have to reconnect to the network every time the MAC address of the specified interface is changed by the program. To help automate this manual process, there is an option to provide a connect script for the program to use to reconnect to the network each time a MAC address for an interface is changed. This can be any script that is capable of being run on the host system, as the m2a.pl program makes an external call to the specified connect-script. For example, the external connect-script supplied with the documentation is a bash script.

4.2.1.2 Direct Encoding

If Interface ID Encode Mode, or Direct Encoding, is used, then the program will explicitly create packets and send out one and only one packet for each piece of the message on the interval as set by the user. The user also has complete control over attributes of the packets. For example, through the configuration file or command line arguments, the user can specify source and destination MAC addresses, as well as source and destination ports. Additionally, the user can prescribe a transport protocol to use (currently TCP or UDP) and data to stuff into the application layer portion of the packet. If the user elects to use TCP as the transport layer protocol, then TCP header options can also be supplied as a hexadecimal string. TCP is used by default, if no transport layer protocol is signaled. Similarly, if no source MAC address is specified, then the MAC of the interface used to send the messages is used for the source MAC field of the packets. The user must specify a destination IPv6 address as well as a destination port, since this is an active process. No outside applications will actually use any of the properties specified to the program.

In a manner similar to the MAC encode method, before the message is encrypted (if the user has so determined) and embedded into the series of interface identifiers, metadata is appended to the message for decoding purposes. In this case, the length of the original message in bytes is calculated and appended to the beginning of the message as described in Section 3.1.

4.2.2 Decode Mode

The decoding process can be operated in either Live Mode or Offline Mode. The Live Mode will monitor network traffic on a specified interface in real-time as traffic flows along. In Offline Mode, the user must specify a file containing previously captured packets in the ‘tcpdump’ tool’s format, otherwise known as a ‘pcap’ file.

It will greatly benefit the sending and receiving parties of the message to set up a filter ahead of time to sniff packets by. Only packets matching the given filter will be sniffed from the live traffic or pcap file and decoded, which will greatly reduce the number of packets that will have to be looked at. For instance, users could decide to only sniff packets destined for MAC address 10:10:10:10:10:10 with destination port 4444, or many other combinations. This is particularly useful for the Direct Encoding method where these values can be specified directly. For MAC Encoding, a little more thought must be given to the type of upper layers such as transport and application layers that will be used by the sender in order for the receiver to know what to filter on. The filter string is also specified by the user in the ‘tcpdump’ tool’s syntax. For example to filter on destination MAC address 10:10:10:10:10:10 and destination port 4444, the filter string would b e ‘ether dst 10:10:10:10:10:10 and dst port 4444’. More information on constructing filter strings can be found in the tcpdump man page [10].

If Live Mode is selected, then the receiver may also choose to sniff the network only until a single complete message is received, or to sniff indefinitely for an arbitrary number of separate messages in Continuous Mode. Separate messages, after decoding, are output to separate files in the directory that the program is run from in the format msg-dd_mmm_yyyy-hh.mm.ss. For example, a message for which the last piece was received on October 21, 2008 at 5:55 PM on the dot, would be written to a file named msg-21_Oct_21008-17.55.0, and would exist in the directory the program was run from. Figure 10 at the end of this section gives a high-level overview of the decoding process performed by the receiver.

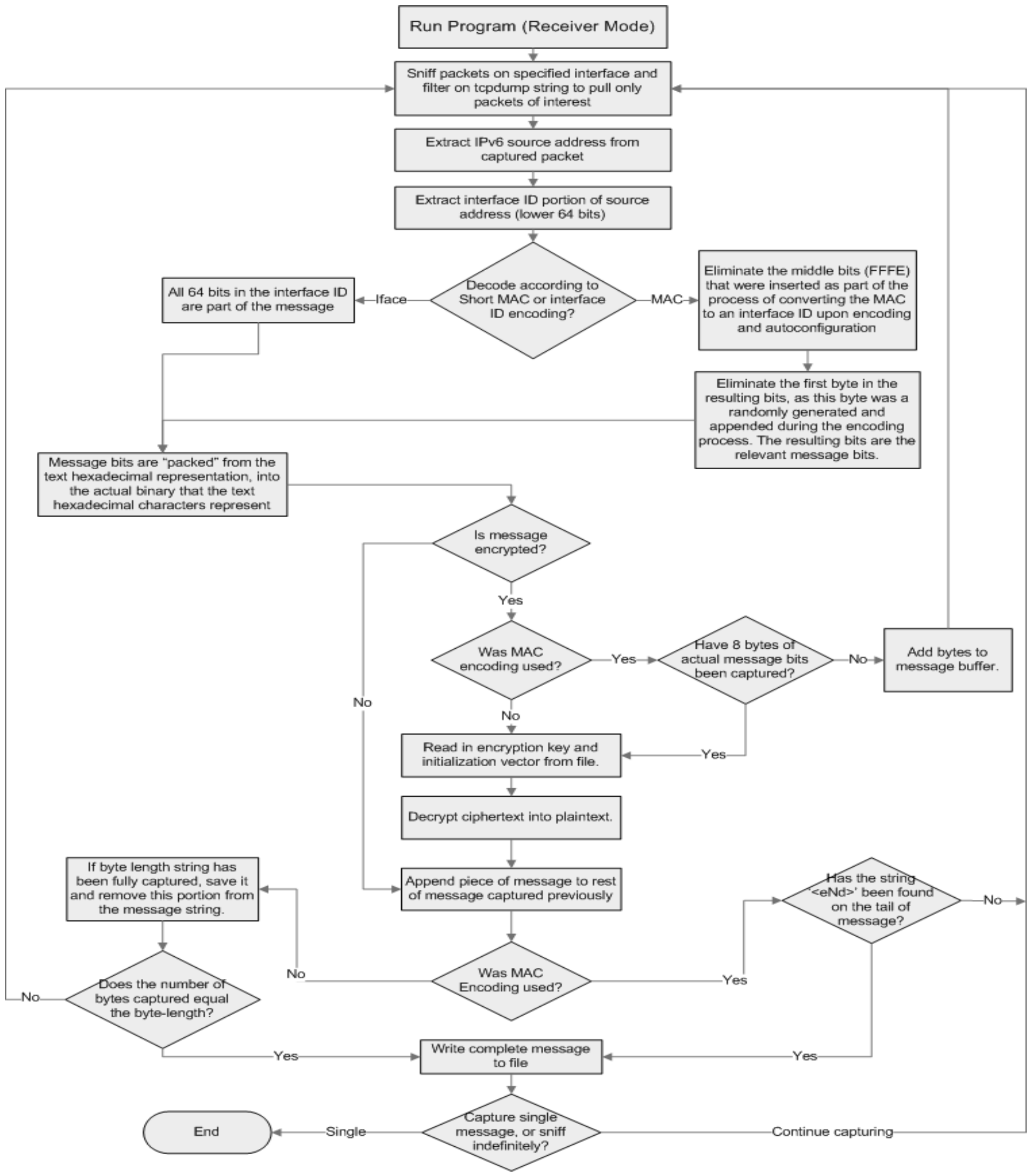

Run Program (Receiver Mode)
Sniff packets on specified interface and filter on tcpdump string to pull only packets of interest
Extract IPv6 source address from captured packet
Extract interface ID portion of source address (lower 64 bits)
Decode according to Short MAC or interface ID encoding?
Iface
MAC
All 64 bits in the interface ID are part of the message
Eliminate the middle bits (FFFE) that were inserted as part of the process of converting the MAC to an interface ID upon encoding and autoconfiguration
Eliminate the first byte in the resulting bits, as this byte was a randomly generated and appended during the encoding process. The resulting bits are the relevant message bits.
Message bits are "packed" from the text hexadecimal representation, into the actual binary that the text hexadecimal characters represent
Is message encrypted?
Yes
No
Was MAC encoding used?
Yes
No
Have 8 bytes of actual message bits been captured?
No
Yes
Add bytes to message buffer.
Read in encryption key and initialization vector from file.
Decrypt ciphertext into plaintext.
Append piece of message to rest of message captured previously
If byte length string has been fully captured, save it and remove this portion from the message string.
Was MAC Encoding used?
No
Yes
Has the string '<eNd>' been found on the tail of message?
No
Yes
Does the number of bytes captured equal the byte-length?
No
Yes
Write complete message to file
Capture single message, or sniff indefinitely?
Single
Continue capturing
End

Figure 10: Decode Mode Process Flow

4.2.2.1 MAC Decoding

MAC decoding involves more overhead that Direct Decoding. The basic process of decoding the address to ASCII characters is explained in Section 3.4, and a full step-by-step example is provided in Section 5.1. The process is finished when the program reads the string ‘<eNd>’ anywhere in a message. The reason that the program cannot only look at the tail end of the message is that, if the original message could not be divided evenly into 5-byte chunks, then the leftover pieces would have been padded out with random bytes to create a complete MAC address. Additionally, if encryption was used in the encoding process, there could be additional garbage at the end of the message. The encryption module requires complete blocks of eight bytes to be encrypted at a time, and if the original message was not evenly divisible by eight, then the leftover pieces would have been padded with spaces out to 8 bytes. Any messages received after the first one that contains the string ‘<eNd>’ are ignored.

Another important implementation detail, which concerns how the program compares two messages that have been received to decide whether they are the same, only applies in MAC encode mode, because only in this mode are the same pieces of the message sent more than once. The program does not compare the messages after they have been extracted and decoded. It compares the two interface IDs of the current and previously received packets. This is important in the case of messages containing an intentional repeat. For example, if a user where to send a message such as ‘ABCDEABCDE’ it would broken into 5-byte pieces to be embedded into the lower five bytes of a MAC address. If the messages themselves were compared, then the program would treat them as the same and not two distinct pieces. However, because the interface IDs are compared before being decoded instead of the messages, the random byte created by the encoding process to be the first octet of the MAC address before it is encoded into an interface ID is likely to distinguish the two distinct pieces.

Though Long MAC encoding has been implemented, the decoding process has not been implemented as of this release of the m2a.pl program. The reason for this is largely presentational. Within the program, a tree structure whose branch paths represent possible messages is fairly straightforward to build, however, displaying this in a useful way in a command-line based program is more difficult.

4.2.2.2 Direct Decoding

Direct decoding is a simpler task than MAC decoding. The process for decoding from an interface ID is explained in section 3.4, and an example is provided in Section 5.2. In this process, the beginning of the message contains the length of the actual message intended by the user in bytes. For example, a 31-byte message that the user wishes to send will have the string '<31>' appended to the beginning. After this length value is read, the program buffers the message until all of the bytes have been captured, and subsequently writes the message to a file in the format described in Section 4.2.2.

In some cases of extremely long messages, the metadata string that specifies the length of the message will not arrive in a single packet, but will take several. For example, if a message is greater than 999999 bytes, then the byte-length metadata is nine bytes (with the '<' and '>' characters) and therefore, requires more than one interface ID to fit it all in. The program handles this by simply buffering the pieces of the metadata until the whole string specifying the length is received. The use of the m2a.pl program for messages of such large size is not recommended or expected, but it should still function correctly.

5. RESULTS

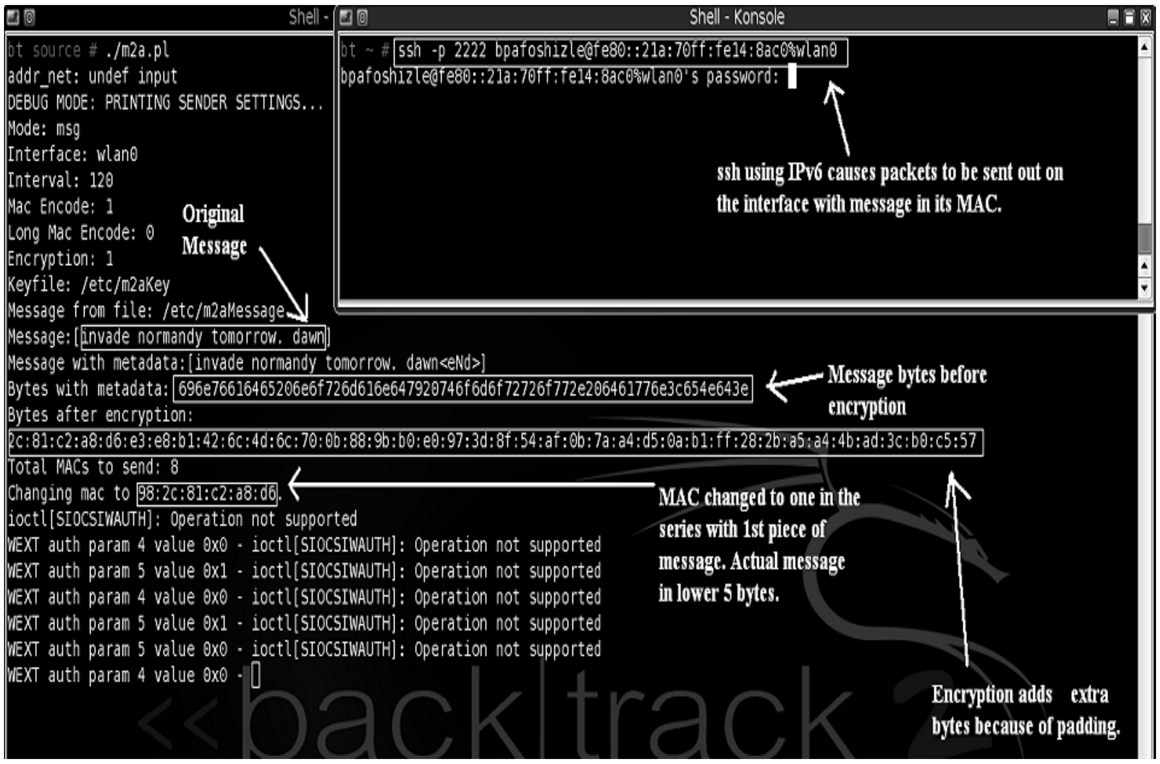
5.1 Scenario A: The MAC Encoding and Decoding Process in Action

On the sending side, User A wants to send the message ‘invade normandy tomorrow. dawn’ that is contained in the file /etc/m2aMessage to User B. User A would like the message to be encrypted and would also like to use the Short MAC encoding method of sending the message. Ahead of time, User A and User B have traded the encryption key and initialization vector that will be used to encrypt and decrypt the message. User A has also informed User B that he will be using the ‘ssh’ program, which uses port 2222 in this case, over IPv6 to connect to a host C. This could be a host that User B has control over, or User B himself. The only necessity is that User B be able to see all traffic in transit from User A to host C. With this information, User B has something to filter on in order to pull only the relevant packets when sniffing the network, namely the specific destination port and address to look for in the packets.

User A wants to get the message out quickly without generating too much suspicion on the network, so he specifies 120 seconds as his time interval that the program will wait before sending each new piece of the message. User B has set up the program on his end to sniff the network waiting for packets that match his filter, and to terminate when a complete message has been received.

Figure 11: MAC Encoding: User A (Sender) - Step 1a below shows what happens when User A, the sender, initially runs the program. The picture points out the message that is being sent and what the underlying bytes of the message looks like before it is encrypted, as well as the setting of the first MAC address in the list for the interface specified. The figure also shows the bytes after encryption, and it is important to note

that the encryption process added five more bytes, which causes eight MACs to be created to contain the entire message. The reason for this is that the original message consisted of 35-bytes with the ‘<eNd>’ string appended, and the encryption block cipher can only work with blocks of eight bytes. Five bytes are appended to make the message 40 bytes, so it can be evenly divided into five blocks of eight bytes each. Figure 11: MAC Encoding: User A (Sender) - Step 1a also points out the fact that because the User A has elected to use MAC encoding as the method to send the message by, a third party application that uses the IP layer must be used in order to cause packets containing the secret message to be sent out. In this case, the application used is secure shell (ssh) over IPv6.

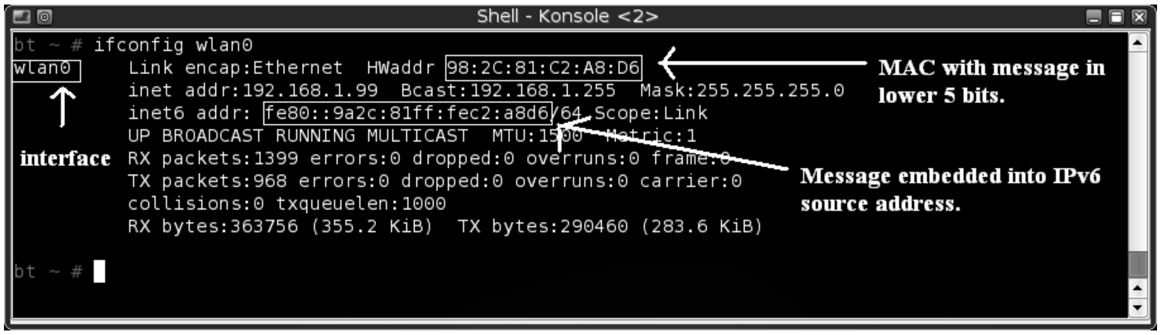

Shell -
Shell - Konsole
ssh using IPv6 causes packets to be sent out on the interface with message in its MAC.
Original Message
Message bytes before encryption
MAC changed to one in the series with 1st piece of message. Actual message in lower 5 bytes.
Encryption adds extra bytes because of padding.

```
bt source # ./m2a.pl
addr_net: undef input
DEBUG MODE: PRINTING SENDER SETTINGS...
Mode: msg
Interface: wlan0
Interval: 120
Mac Encode: 1
Long Mac Encode: 0
Encryption: 1
Keyfile: /etc/m2aKey
Message from file: /etc/m2aMessage
Message:[invade normandy tomorrow. dawn]
Message with metadata:[invade normandy tomorrow. dawn<eNd>]
Bytes with metadata: 696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e3c654e643e
Bytes after encryption:
2c:81:c2:a8:d6:e3:e8:b1:42:6c:4d:6c:70:0b:88:9b:b0:e0:97:3d:8f:54:af:0b:7a:a4:d5:0a:b1:ff:28:2b:a5:a4:4b:ad:3c:b0:c5:57
Total MACs to send: 8
Changing mac to 98:2c:81:c2:a8:d6.
ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x1 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x1 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 -
```

```
bt ~ # ssh -p 2222 bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0
bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0's password:
```

Figure 11: MAC Encoding: User A (Sender) - Step 1a

The next figure below simply shows the output of the ‘ifconfig’ command that lists details for network interfaces. Figure 12: MAC Encoding: User A - Step 1b points out that the MAC address for the ‘wlan0’ interface has been set to the MAC containing the first part of the message, and that the link-local IPv6 address for that interface has been changed as a result.


Shell - Konsole <2>
MAC with message in lower 5 bits.
interface
Message embedded into IPv6 source address.

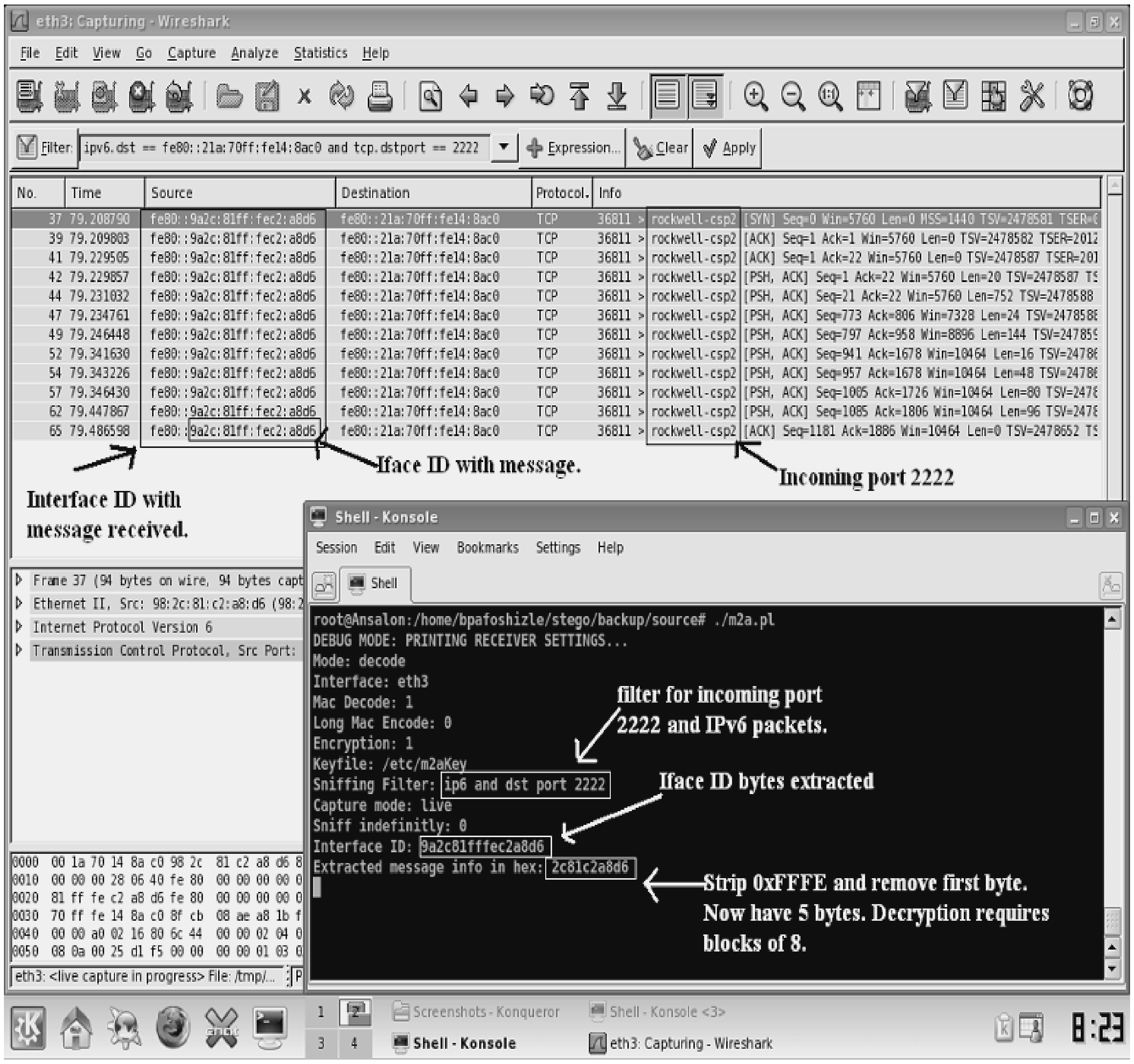
```
bt ~ # ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 98:2C:81:C2:A8:D6
          inet addr:192.168.1.99  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::9a2c:81ff:fec2:a8d6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:968 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:363756 (355.2 KiB)  TX bytes:290460 (283.6 KiB)

bt ~ #
```

Figure 12: MAC Encoding: User A - Step 1b

After the program has changed the MAC address of the wlan0 interface to the first part of the message, it then brings the interface back up and connects it to the network with the connect script specified by the user. Upon connecting to the network, Stateless Address Autoconfiguration is performed by the interface and an IPv6 source address containing the message is set for the interface. Then, User A calls the ssh program to connect over IPv6 to User B causing packets to be sent to User B that contain the secret message.

Figure 13: MAC Encoding: User B (Receiver) - Step 1 below shows what happens on the receiver side for User B. The image depicts a Wireshark packet capture to make clear what the program is doing internally. The figure also shows some output from the program in decode or receiver mode. User B knows that User A will be sending packets to destination port 2222 and that the packets will be IPv6 (Wireshark and the

program’s internal sniffer can sniff IPv4 packets as well), so the user has specified these properties in the filter. Figure 13: MAC Encoding: User B (Receiver) - Step 1 shows the link-local source address containing the message, and points out the interface ID portion of the address. It also shows the bytes of the message after the EUI-64 process has been reversed by removing the 0xFFFE bits and removing the first byte of the message. The resulting hex digits shown are the encrypted first five bytes of the message, but the program cannot perform decryption until it has at least eight bytes, so it must wait. It is important to notice here that the program is receiving many packets that contain the same piece of the message. It recognizes this and ignores the repeats.

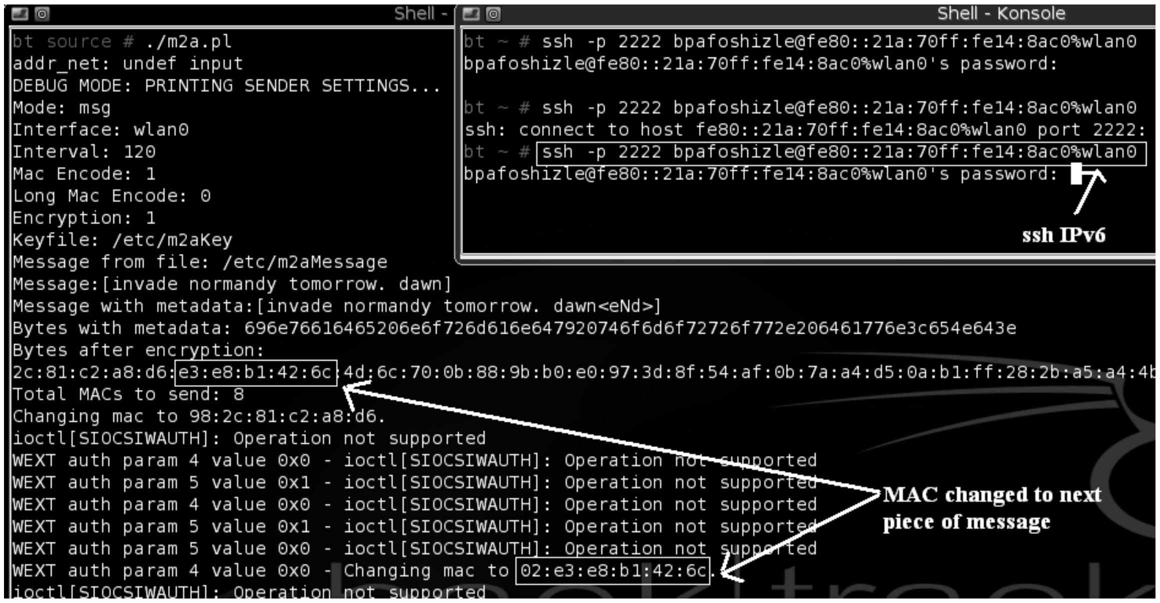

eth3: Capturing - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
Interface ID with message received.
Iface ID with message.
Incoming port 2222
Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
filter for incoming port 2222 and IPv6 packets.
Iface ID bytes extracted
Strip 0xFFFE and remove first byte. Now have 5 bytes. Decryption requires blocks of 8.
Screenshots - Konqueror
Shell - Konsole <3>
Shell - Konsole
eth3: Capturing - Wireshark
8:23

```
Filter: ipv6.dst == fe80::21a:70ff:fe14:8ac0 and tcp.dstport == 2222

No.  Time       Source                     Destination              Protocol. Info
 37  79.208790  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [SYN] Seq=0 Win=5760 Len=0 MSS=1440 TSV=2478581 TSER=
 39  79.209803  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [ACK] Seq=1 Ack=1 Win=5760 Len=0 TSV=2478582 TSER=2012
 41  79.229505  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [ACK] Seq=1 Ack=22 Win=5760 Len=0 TSV=2478587 TSER=201
 42  79.229857  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=1 Ack=22 Win=5760 Len=20 TSV=2478587 T
 44  79.231032  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=21 Ack=22 Win=5760 Len=752 TSV=2478588
 47  79.234761  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=773 Ack=806 Win=7328 Len=24 TSV=247858
 49  79.246448  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=797 Ack=958 Win=8896 Len=144 TSV=24785
 52  79.341630  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=941 Ack=1678 Win=10464 Len=16 TSV=2478
 54  79.343226  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=957 Ack=1678 Win=10464 Len=48 TSV=2478
 57  79.346430  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=1005 Ack=1726 Win=10464 Len=80 TSV=247
 62  79.447867  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [PSH, ACK] Seq=1085 Ack=1806 Win=10464 Len=96 TSV=247
 65  79.486598  fe80::9a2c:81ff:fec2:a8d6  fe80::21a:70ff:fe14:8ac0  TCP   36811 > rockwell-csp2 [ACK] Seq=1181 Ack=1886 Win=10464 Len=0 TSV=2478652 T

▷ Frame 37 (94 bytes on wire, 94 bytes capt
▷ Ethernet II, Src: 98:2c:81:c2:a8:d6 (98:2
▷ Internet Protocol Version 6
▷ Transmission Control Protocol, Src Port:

0000  00 1a 70 14 8a c0 98 2c  81 c2 a8 d6 8
0010  00 00 00 28 06 40 fe 80  00 00 00 00 0
0020  81 ff fe c2 a8 d6 fe 80  00 00 00 00 0
0030  70 ff fe 14 8a c0 8f cb  08 ae a8 1b f
0040  00 00 a0 02 16 80 6c 44  00 00 02 04 0
0050  08 0a 00 25 d1 f5 00 00  00 00 01 03 0

eth3: <live capture in progress> File: /tmp/...
```

```
root@Ansalon:/home/bpafoshizle/stego/backup/source# ./m2a.pl
DEBUG MODE: PRINTING RECEIVER SETTINGS...
Mode: decode
Interface: eth3
Mac Decode: 1
Long Mac Encode: 0
Encryption: 1
Keyfile: /etc/m2aKey
Sniffing Filter: ip6 and dst port 2222
Capture mode: live
Sniff indefinitly: 0
Interface ID: 9a2c81fffec2a8d6
Extracted message info in hex: 2c81c2a8d6
```

Figure 13: MAC Encoding: User B (Receiver) - Step 1

The next image, Figure 14: MAC Encoding: User A - Step 2a shows User A's side sending out the second piece of the message. This looks similar to Figure 11: MAC Encoding: User A (Sender) - Step 1a, and also shows that the MAC is changed and that the user must again use the third party ssh program that will in turn use the IPv6 network layer to communicate.

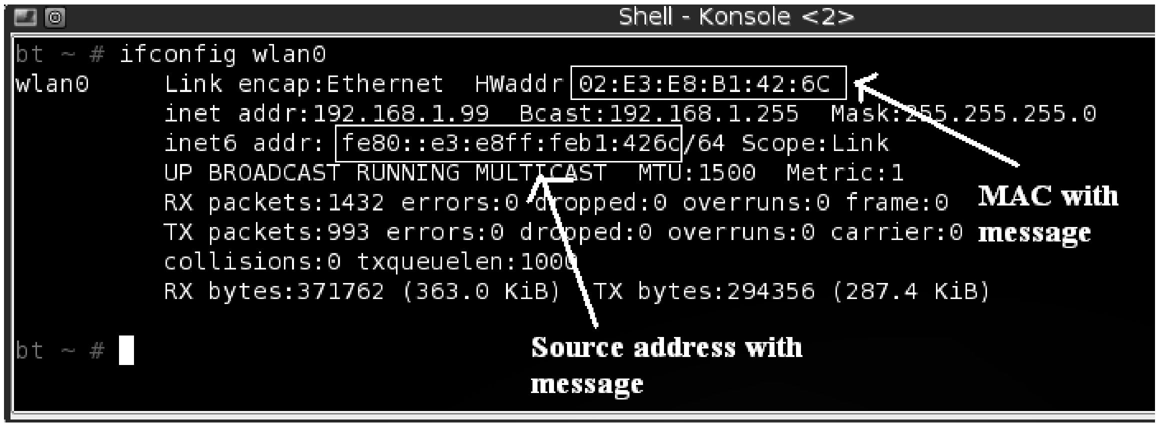


```
bt source # ./m2a.pl
addr_net: undef input
DEBUG MODE: PRINTING SENDER SETTINGS...
Mode: msg
Interface: wlan0
Interval: 120
Mac Encode: 1
Long Mac Encode: 0
Encryption: 1
Keyfile: /etc/m2aKey
Message from file: /etc/m2aMessage
Message:[invade normandy tomorrow. dawn]
Message with metadata:[invade normandy tomorrow. dawn<eNd>]
Bytes with metadata: 696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e3c654e643e
Bytes after encryption:
2c:81:c2:a8:d6:e3:e8:b1:42:6c:4d:6c:70:0b:88:9b:b0:e0:97:3d:8f:54:af:0b:7a:a4:d5:0a:b1:ff:28:2b:a5:a4:4b
Total MACs to send: 8
Changing mac to 98:2c:81:c2:a8:d6.
ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x1 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x1 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 5 value 0x0 - ioctl[SIOCSIWAUTH]: Operation not supported
WEXT auth param 4 value 0x0 - Changing mac to 02:e3:e8:b1:42:6c.
ioctl[SIOCSIWAUTH]: Operation not supported
```

```
bt ~ # ssh -p 2222 bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0
bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0's password:

bt ~ # ssh -p 2222 bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0
ssh: connect to host fe80::21a:70ff:fe14:8ac0%wlan0 port 2222:
bt ~ # ssh -p 2222 bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0
bpafoshizle@fe80::21a:70ff:fe14:8ac0%wlan0's password:
```

Figure 14: MAC Encoding: User A - Step 2a

Figure 15: MAC Encoding: User A - Step 2b is similar to Figure 12: MAC Encoding: User A - Step 1b and shows that the MAC address for wlan0 has been changed to the second piece of the message as claimed by Figure 14: MAC Encoding: User A - Step 2a above. Again, the message is embedded into the Source address as well.

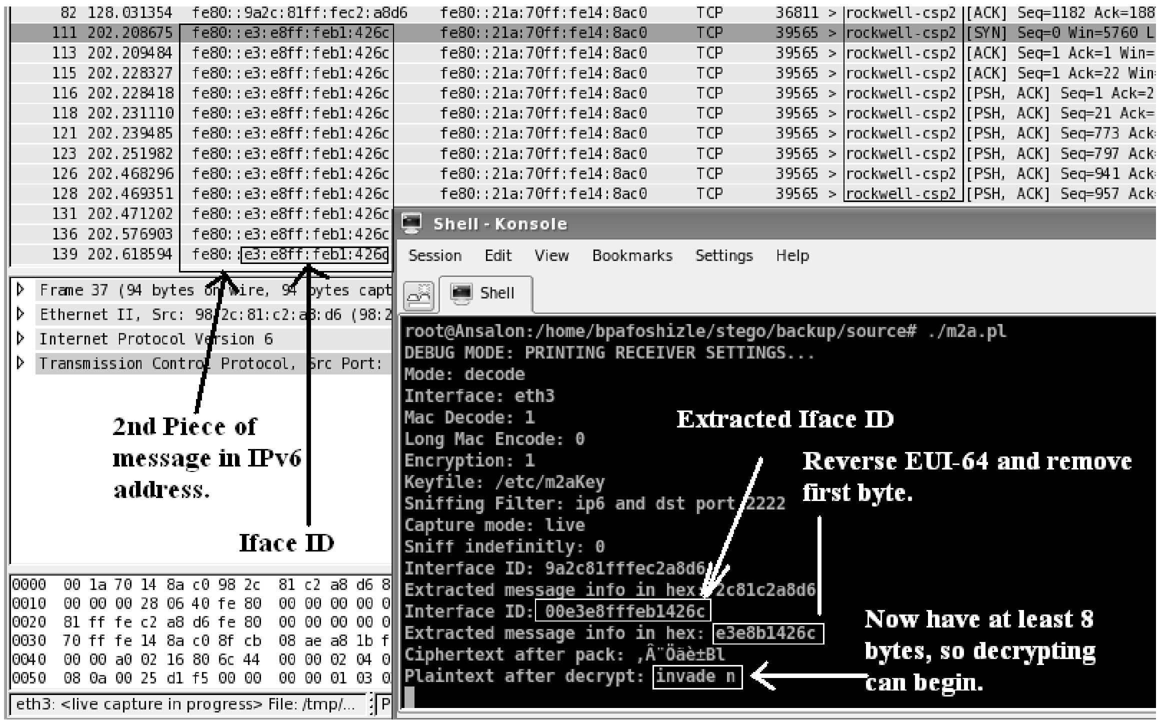


```
bt ~ # ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 02:E3:E8:B1:42:6C
          inet addr:192.168.1.99  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::e3:e8ff:feb1:426c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1432 errors:0 dropped:0 overruns:0 frame:0
          TX packets:993 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:371762 (363.0 KiB)  TX bytes:294356 (287.4 KiB)

bt ~ #
```

Figure 15: MAC Encoding: User A - Step 2b

Figure 16: MAC Encoding: User B - Step 2 shows step two for the receiver, User B. This figure is similar to Step 1 for User B depicted in Figure 13: MAC Encoding:

User B (Receiver) - Step 1 above, in that it shows the link-local source address containing the message, the interface ID, and the reversal of the EUI-64 process to retrieve the bytes of the message. However, this Figure is important because now that the receiver has received at least eight bytes of actual message (they have received 10 bytes), the first piece of the message can be decrypted. This is shown below, as the first piece of the message ‘invade n’ appears in plaintext.



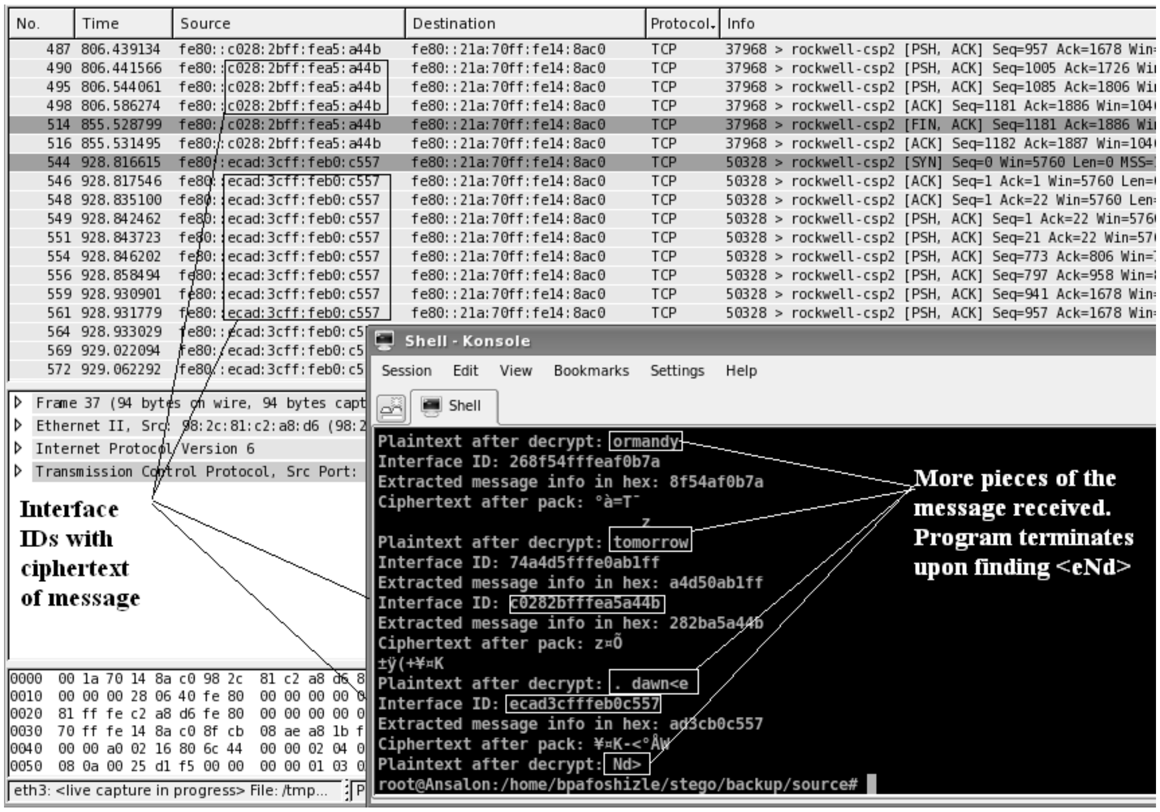
```
 82 128.031354  fe80::9a2c:81ff:fec2:a8d6   fe80::21a:70ff:fe14:8ac0   TCP   36811 > rockwell-csp2 [ACK] Seq=1182 Ack=188
111 202.208675  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [SYN] Seq=0 Win=5760 L
113 202.209484  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [ACK] Seq=1 Ack=1 Win=
115 202.228327  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [ACK] Seq=1 Ack=22 Win
116 202.228418  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=1 Ack=2
118 202.231110  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=21 Ack=
121 202.239485  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=773 Ack
123 202.251982  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=797 Ack
126 202.468296  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=941 Ack
128 202.469351  fe80::e3:e8ff:feb1:426c     fe80::21a:70ff:fe14:8ac0   TCP   39565 > rockwell-csp2 [PSH, ACK] Seq=957 Ack
131 202.471202  fe80::e3:e8ff:feb1:426c
136 202.576903  fe80::e3:e8ff:feb1:426c
139 202.618594  fe80::e3:e8ff:feb1:426c

▷ Frame 37 (94 bytes on wire, 94 bytes capt
▷ Ethernet II, Src: 98:2c:81:c2:a8:d6 (98:2
▷ Internet Protocol Version 6
▷ Transmission Control Protocol, Src Port:

0000  00 1a 70 14 8a c0 98 2c  81 c2 a8 d6 8
0010  00 00 00 28 06 40 fe 80  00 00 00 00 0
0020  81 ff fe c2 a8 d6 fe 80  00 00 00 00 0
0030  70 ff fe 14 8a c0 8f cb  08 ae a8 1b f
0040  00 00 a0 02 16 80 6c 44  00 00 02 04 0
0050  08 0a 00 25 d1 f5 00 00  00 00 01 03 0

eth3: <live capture in progress> File: /tmp/...
```

```
root@Ansalon:/home/bpafoshizle/stego/backup/source# ./m2a.pl
DEBUG MODE: PRINTING RECEIVER SETTINGS...
Mode: decode
Interface: eth3
Mac Decode: 1
Long Mac Encode: 0
Encryption: 1
Keyfile: /etc/m2aKey
Sniffing Filter: ip6 and dst port 2222
Capture mode: live
Sniff indefinitly: 0
Interface ID: 9a2c81fffec2a8d6
Extracted message info in hex: 2c81c2a8d6
Interface ID: 00e3e8fffeb1426c
Extracted message info in hex: e3e8b1426c
Ciphertext after pack: ,Â¨Öàè±Bl
Plaintext after decrypt: invade n
```

Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
2nd Piece of message in IPv6 address.
Iface ID
Extracted Iface ID
Reverse EUI-64 and remove first byte.
Now have at least 8 bytes, so decrypting can begin.

Figure 16: MAC Encoding: User B - Step 2

The above process is repeated for however many steps it takes to send the entire message out. In this case, there are eight steps because there are eight pieces of the message to be sent and received. With an interval of 120 seconds or two minutes between each successive message, the entire process takes about 14 minutes. Figure 17: MAC Encoding: User B - Final Steps below shows the end of the process on the receiver side after each piece of the message has been received and the string ‘<eNd>’ has been found. Because User B has chosen to terminate the program after a complete message has been received instead of sniffing indefinitely, the program quits after a single message.



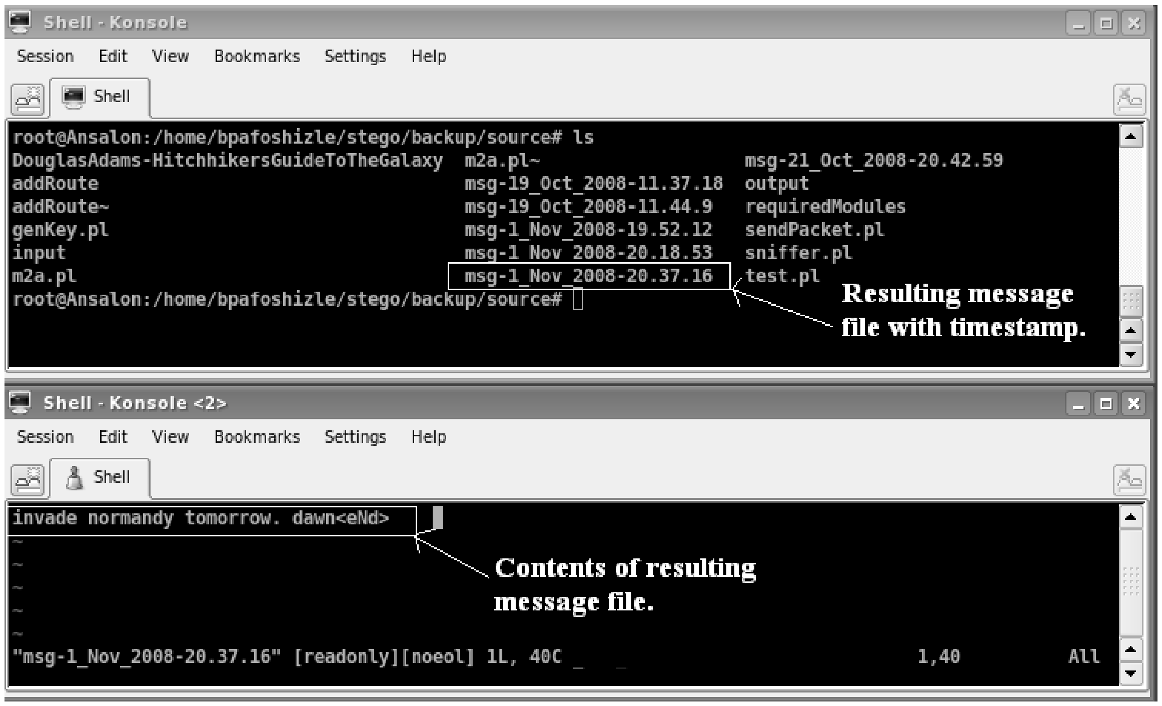
```
No.  Time        Source                          Destination                Protocol  Info
 487 806.439134  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [PSH, ACK] Seq=957 Ack=1678 Win
 490 806.441566  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [PSH, ACK] Seq=1005 Ack=1726 Wi
 495 806.544061  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [PSH, ACK] Seq=1085 Ack=1806 Wi
 498 806.586274  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [ACK] Seq=1181 Ack=1886 Win=104
 514 855.528799  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [FIN, ACK] Seq=1181 Ack=1886 Wi
 516 855.531495  fe80::c028:2bff:fea5:a44b       fe80::21a:70ff:fe14:8ac0   TCP       37968 > rockwell-csp2 [ACK] Seq=1182 Ack=1887 Win=104
 544 928.816615  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [SYN] Seq=0 Win=5760 Len=0 MSS=
 546 928.817546  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [ACK] Seq=1 Ack=1 Win=5760 Len=
 548 928.835100  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [ACK] Seq=1 Ack=22 Win=5760 Len
 549 928.842462  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=1 Ack=22 Win=576
 551 928.843723  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=21 Ack=22 Win=57
 554 928.846202  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=773 Ack=806 Win=
 556 928.858494  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=797 Ack=958 Win=
 559 928.930901  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=941 Ack=1678 Win
 561 928.931779  fe80::ecad:3cff:feb0:c557       fe80::21a:70ff:fe14:8ac0   TCP       50328 > rockwell-csp2 [PSH, ACK] Seq=957 Ack=1678 Win
 564 928.933029  fe80::ecad:3cff:feb0:c5
 569 929.022094  fe80::ecad:3cff:feb0:c5
 572 929.062292  fe80::ecad:3cff:feb0:c5

▷ Frame 37 (94 bytes on wire, 94 bytes capt
▷ Ethernet II, Src: 98:2c:81:c2:a8:d6 (98:2
▷ Internet Protocol Version 6
▷ Transmission Control Protocol, Src Port:

0000  00 1a 70 14 8a c0 98 2c  81 c2 a8 d6 8
0010  00 00 00 28 06 40 fe 80  00 00 00 00 0
0020  81 ff fe c2 a8 d6 fe 80  00 00 00 00 0
0030  70 ff fe 14 8a c0 8f cb  08 ae a8 1b f
0040  00 00 a0 02 16 80 6c 44  00 00 02 04 0
0050  08 0a 00 25 d1 f5 00 00  00 00 01 03 0
eth3: <live capture in progress> File: /tmp...
```

```
Plaintext after decrypt: ormandy
Interface ID: 268f54fffeaf0b7a
Extracted message info in hex: 8f54af0b7a
Ciphertext after pack: °à=T¯
                          z
Plaintext after decrypt: tomorrow
Interface ID: 74a4d5fffe0ab1ff
Extracted message info in hex: a4d50ab1ff
Interface ID: c0282bfffea5a44b
Extracted message info in hex: 282ba5a44b
Ciphertext after pack: z¤Õ
±ÿ(+¥¤K
Plaintext after decrypt: . dawn<e
Interface ID: ecad3cfffeb0c557
Extracted message info in hex: ad3cb0c557
Ciphertext after pack: ¥¤K-<°ÅW
Plaintext after decrypt: Nd>
root@Ansalon:/home/bpafoshizle/stego/backup/source# 
```

Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Interface IDs with ciphertext of message
More pieces of the message received. Program terminates upon finding <eNd>

Figure 17: MAC Encoding: User B - Final Steps

After an entire message has been received by the program in decode mode, a file is created containing the entire message with a timestamp as part of the name as explained in Section 4.2.2. Figure 18: MAC Encoding: Resulting Message File and Contents shows the resulting file and it's contents created on the decoding side.


```
root@Ansalon:/home/bpafoshizle/stego/backup/source# ls
DouglasAdams-HitchhikersGuideToTheGalaxy  m2a.pl~                    msg-21_Oct_2008-20.42.59
addRoute                                  msg-19_Oct_2008-11.37.18  output
addRoute~                                 msg-19_Oct_2008-11.44.9   requiredModules
genKey.pl                                 msg-1_Nov_2008-19.52.12   sendPacket.pl
input                                     msg-1_Nov_2008-20.18.53   sniffer.pl
m2a.pl                                    msg-1_Nov_2008-20.37.16   test.pl
root@Ansalon:/home/bpafoshizle/stego/backup/source#
```

```
invade normandy tomorrow. dawn<eNd>
~
~
~
~
~
"msg-1_Nov_2008-20.37.16" [readonly][noeol] 1L, 40C                    1,40          All
```
Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Resulting message file with timestamp.
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Contents of resulting message file.

Figure 18: MAC Encoding: Resulting Message File and Contents

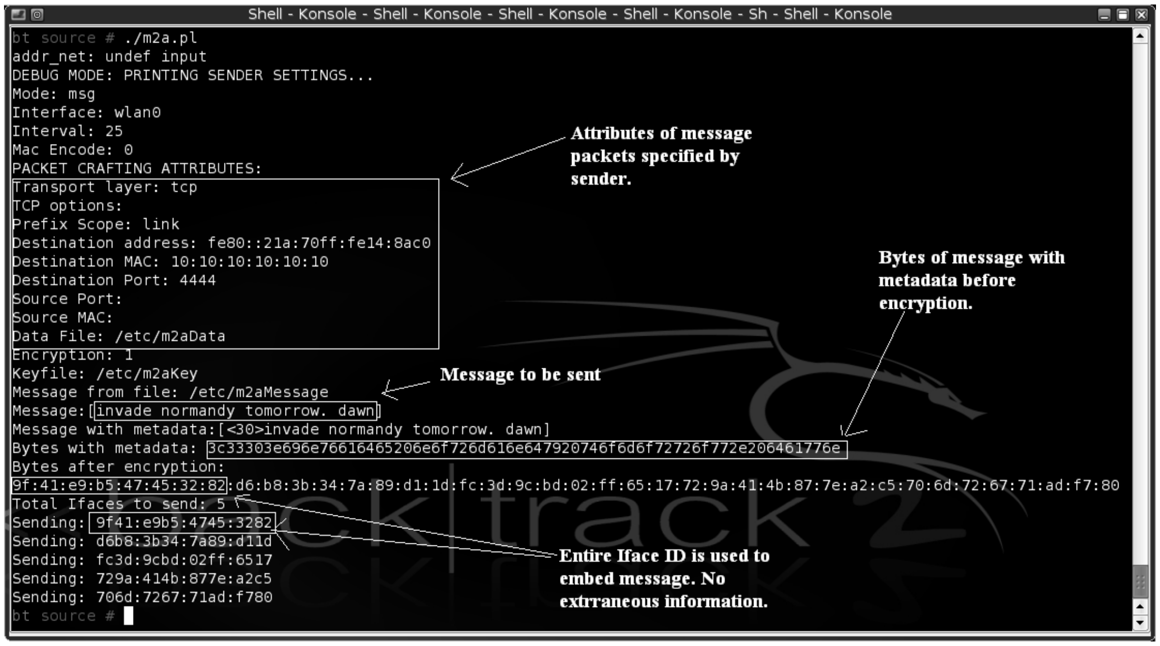
5.2 Scenario B: The Direct Encoding and Decoding Process in Action

For Scenario B, it is supposed that User A wants to send the same message ‘invade normandy tomorrow. dawn’ to User B, but does not want to have to actually change the MAC address in order to do it. There could be many reasons for wanting to use this method over the MAC method. One reason to use this method is that there is more bandwidth per message container (or cover message) resulting in less total cover messages having to be sent out. It also avoids the overhead of having to reconnect to the network every time the MAC address is changed. User A accepts the fact that only one of piece of the message will be sent out, and that there is no way for any reply to be sent unless User B knows the true address of User A ahead of time. Furthermore, User A wants to send the message out quickly, so he specifies a delay interval of 25 seconds.

User A wants also to send the message to the destination link-local IPv6 address fe80::21a:70ff:fe14:8ac0, the same as in Scenario A. User A has a lot more freedom over the properties of packets being sent out in this mode. He chooses to specify the destination MAC address as 10:10:10:10:10:10, which does not have to be, and in this case is not, the real destination MAC of the receiver. The destination MAC address above could be useful as a filter value for the receiver to pull relevant packets, if the two parties are operating on a LAN where link-layer information is not stripped from packets. User A and B have agreed ahead of time that the message will be sent in TCP packets with a destination port of 4444 to facilitate filtering, and User A specifies this accordingly. The users also want to use encryption as was the case for Scenario A, so a file containing an encryption key and initialization vector are specified. In addition, because this scenario is operating on a LAN, User A specifies ‘link’ as the scope to use, so the program knows to use whatever prefix is currently associated with link-local for the interface specified. The scope could also be global, site, or local, in which case the program will pull whatever prefix is currently used by the interface for that scope to append to the interface identifiers containing the message. User A has also elected to specify a file containing text that will be used for the layer seven data portion of all packets sent out. This is the cover message, and in this case, the information contained in the text is intentionally designed to be disinformation that would be false or misleading to any third party that might happen to intercept the message.

Finally, User A has not bothered to specify values for the Source Port, the Source MAC, or the TCP-Options string, so default values for all of these will be used. The Source MAC for the packets sent out will be the true MAC address for the interface specified, while the source port will be whatever is randomly generated by the TCP layer of the network stack.

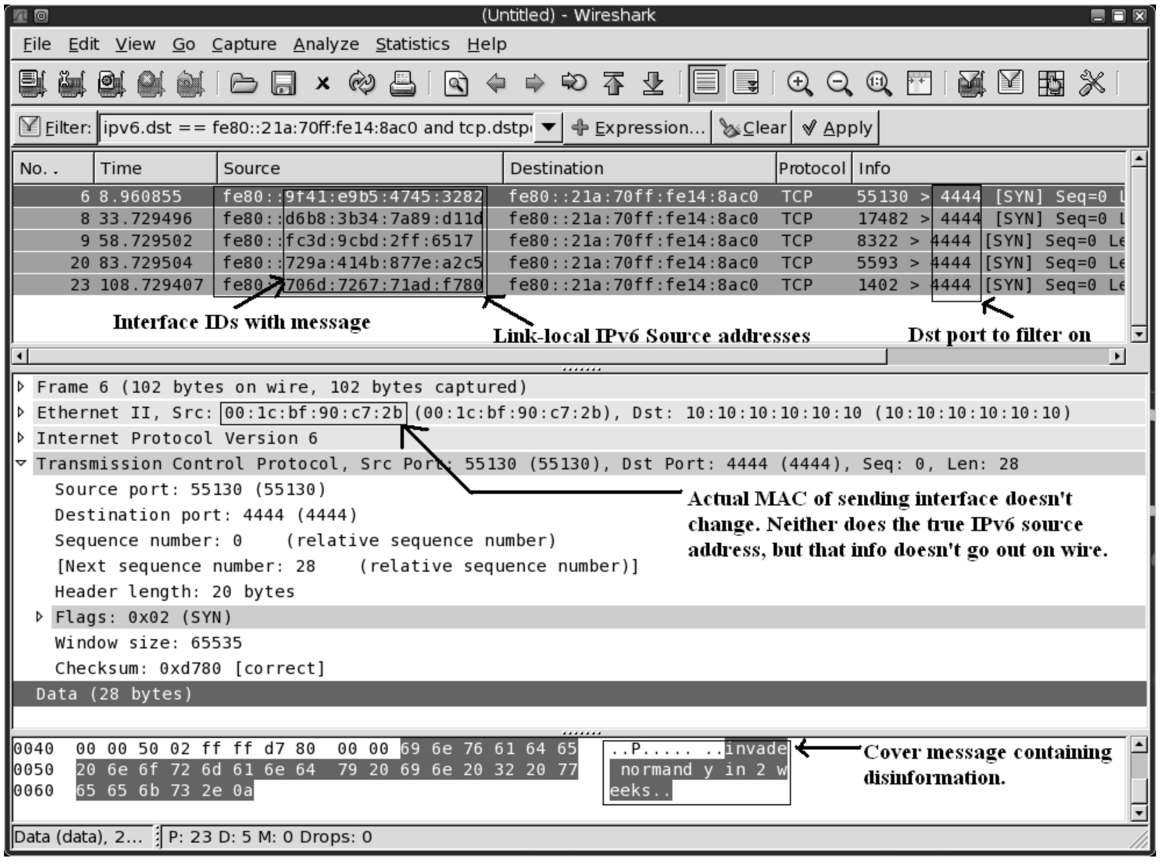
Figure 19: Direct Encoding: User A (Sender) – Program Output below shows the output of the program for User A, the sender. This figure shows the time after the program has sent out all packets on the 25-second interval. In this case, as in Scenario A, the encryption block cipher only works with 8-byte blocks, so some information is added to the end of the message in order to pad the last piece out to eight bytes. In this case, the message with metadata is 34 bytes long, which divides into four complete blocks of eight bytes with two leftover bytes. The encryption automatically tacks on six bytes to make another complete block, but this does not cause any more interface ID's to be created in order to send out the message than would be created if encryption were not used. It is a useful coincidence for this reason and for the decryption process that the block cipher works with eight bytes at a time, and the interface IDs are exactly eight bytes long. If encryption is not used, and the bytes of the message and metadata are not evenly divisible by eight, then the leftover bytes must be padded out to eight with random data anyway to create another whole interface ID. This is not so convenient for short and long MAC encoding, which use five and six bytes, respectively for the medium, while the block cipher requires eight bytes. Because of this, using encryption will often cause more MAC addresses be created than would be if encryption were not used. This is not an issue with direct encoding into the interface ID.


Shell - Konsole - Shell - Konsole - Shell - Konsole - Shell - Konsole - Sh - Shell - Konsole
Attributes of message packets specified by sender.
Bytes of message with metadata before encryption.
Message to be sent
Entire Iface ID is used to embed message. No extrraneous information.

```
bt source # ./m2a.pl
addr_net: undef input
DEBUG MODE: PRINTING SENDER SETTINGS...
Mode: msg
Interface: wlan0
Interval: 25
Mac Encode: 0
PACKET CRAFTING ATTRIBUTES:
Transport layer: tcp
TCP options:
Prefix Scope: link
Destination address: fe80::21a:70ff:fe14:8ac0
Destination MAC: 10:10:10:10:10:10
Destination Port: 4444
Source Port:
Source MAC:
Data File: /etc/m2aData
Encryption: 1
Keyfile: /etc/m2aKey
Message from file: /etc/m2aMessage
Message:[invade normandy tomorrow. dawn]
Message with metadata:[<30>invade normandy tomorrow. dawn]
Bytes with metadata: 3c33303e696e76616465206e6f726d616e647920746f6d6f72726f772e206461776e
Bytes after encryption:
9f:41:e9:b5:47:45:32:82:d6:b8:3b:34:7a:89:d1:1d:fc:3d:9c:bd:02:ff:65:17:72:9a:41:4b:87:7e:a2:c5:70:6d:72:67:71:ad:f7:80
Total Ifaces to send: 5
Sending: 9f41:e9b5:4745:3282
Sending: d6b8:3b34:7a89:d11d
Sending: fc3d:9cbd:02ff:6517
Sending: 729a:414b:877e:a2c5
Sending: 706d:7267:71ad:f780
bt source #
```

Figure 19: Direct Encoding: User A (Sender) – Program Output

Figure 20: Direct Encoding: User A - Packet Capture below shows a Wireshark capture of the packets created by the program and sent out over the IPv6 network. It can be seen that a single packet for each piece of the message is sent out, in contrast to the many per piece of message sent out in Scenario A. The figure also shows the cover message in the data portion of the packet, as well as the destination port of 4444 that is used to filter only packets of interest. Figure 20: Direct Encoding: User A - Packet Capture also shows the True MAC address of the interface used to inject packets, demonstrating that it is not changed to a piece of the message as in MAC Encoding mode. What is not shown in the picture, but is important to remember, is the fact that the true IPv6 source address of the interface is not actually changed either. These packets contain spoofed source addresses with the embedded message.

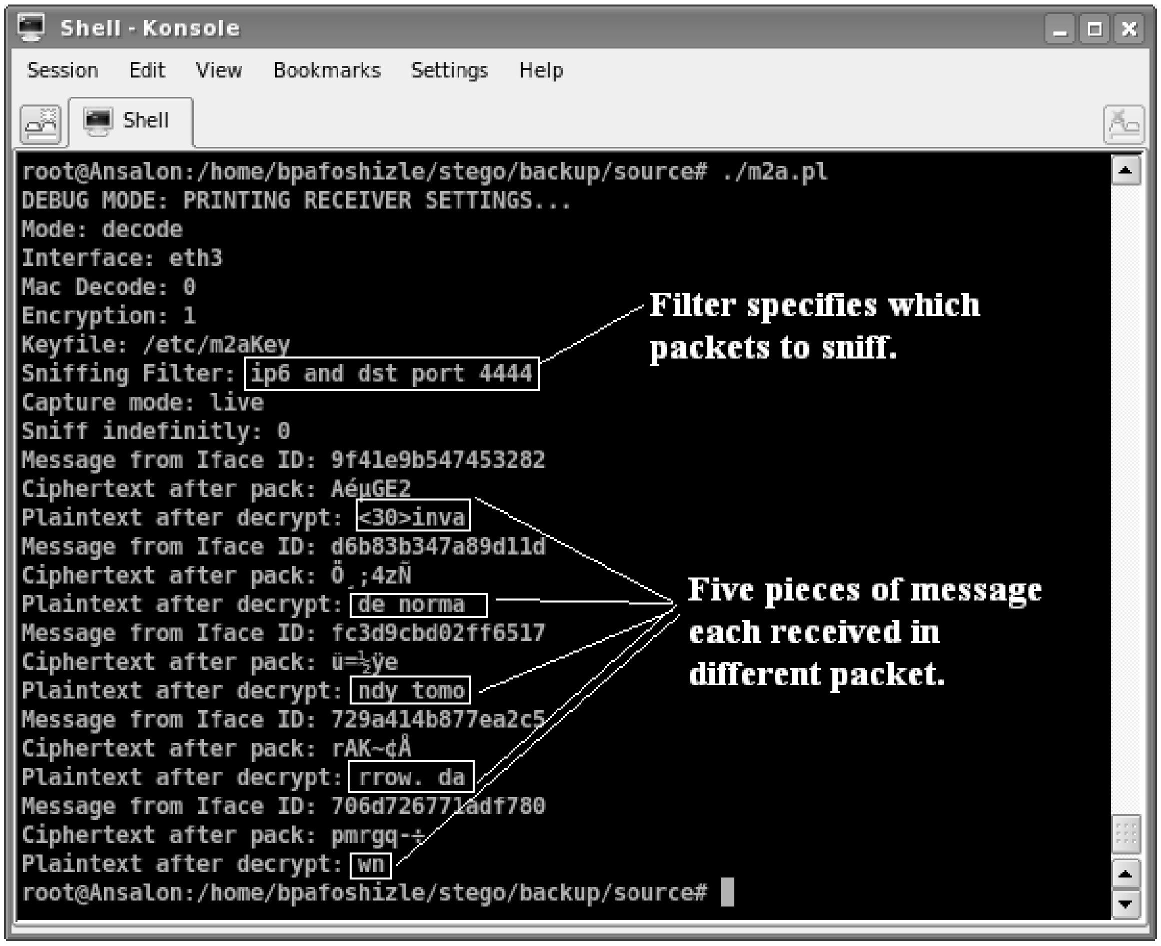

(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: ipv6.dst == fe80::21a:70ff:fe14:8ac0 and tcp.dstp
Expression... Clear Apply
Interface IDs with message
Link-local IPv6 Source addresses
Dst port to filter on
Actual MAC of sending interface doesn't change. Neither does the true IPv6 source address, but that info doesn't go out on wire.
Cover message containing disinformation.

```
No. .  Time        Source                          Destination               Protocol  Info
     6 8.960855    fe80::9f41:e9b5:4745:3282      fe80::21a:70ff:fe14:8ac0  TCP       55130 > 4444 [SYN] Seq=0 L
     8 33.729496   fe80::d6b8:3b34:7a89:d11d      fe80::21a:70ff:fe14:8ac0  TCP       17482 > 4444 [SYN] Seq=0 L
     9 58.729502   fe80::fc3d:9cbd:2ff:6517       fe80::21a:70ff:fe14:8ac0  TCP       8322 > 4444 [SYN] Seq=0 Le
    20 83.729504   fe80::729a:414b:877e:a2c5      fe80::21a:70ff:fe14:8ac0  TCP       5593 > 4444 [SYN] Seq=0 Le
    23 108.729407  fe80::706d:7267:71ad:f780      fe80::21a:70ff:fe14:8ac0  TCP       1402 > 4444 [SYN] Seq=0 Le

▷ Frame 6 (102 bytes on wire, 102 bytes captured)
▷ Ethernet II, Src: 00:1c:bf:90:c7:2b (00:1c:bf:90:c7:2b), Dst: 10:10:10:10:10:10 (10:10:10:10:10:10)
▷ Internet Protocol Version 6
▽ Transmission Control Protocol, Src Port: 55130 (55130), Dst Port: 4444 (4444), Seq: 0, Len: 28
    Source port: 55130 (55130)
    Destination port: 4444 (4444)
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 28    (relative sequence number)]
    Header length: 20 bytes
  ▷ Flags: 0x02 (SYN)
    Window size: 65535
    Checksum: 0xd780 [correct]
  Data (28 bytes)

0040  00 00 50 02 ff ff d7 80  00 00 69 6e 76 61 64 65   ..P..... ..invade
0050  20 6e 6f 72 6d 61 6e 64  79 20 69 6e 20 32 20 77    normand y in 2 w
0060  65 65 6b 73 2e 0a                                  eeks..

Data (data), 2...  P: 23 D: 5 M: 0 Drops: 0
```

Figure 20: Direct Encoding: User A - Packet Capture

Figure 21: Direct Encoding: User B (Receiver) – Program Output shows the output of the program in decode mode on the receiver side. The image shows the filter used for sniffing packets, and each piece of the message received in turn, as well as the interface ID that contained the piece of the message. It is notable that each piece of the message can be decrypted as soon as it is received. Again, this is a result of the useful coincidence of the block cipher working with eight bytes at a time, and the interface IDs containing exactly eight bytes. The decoder does not have to wait before decrypting each piece. Once the program has read 30 bytes (not including the metadata length information itself), it prints the message to a file and terminates as in Scenario A, because

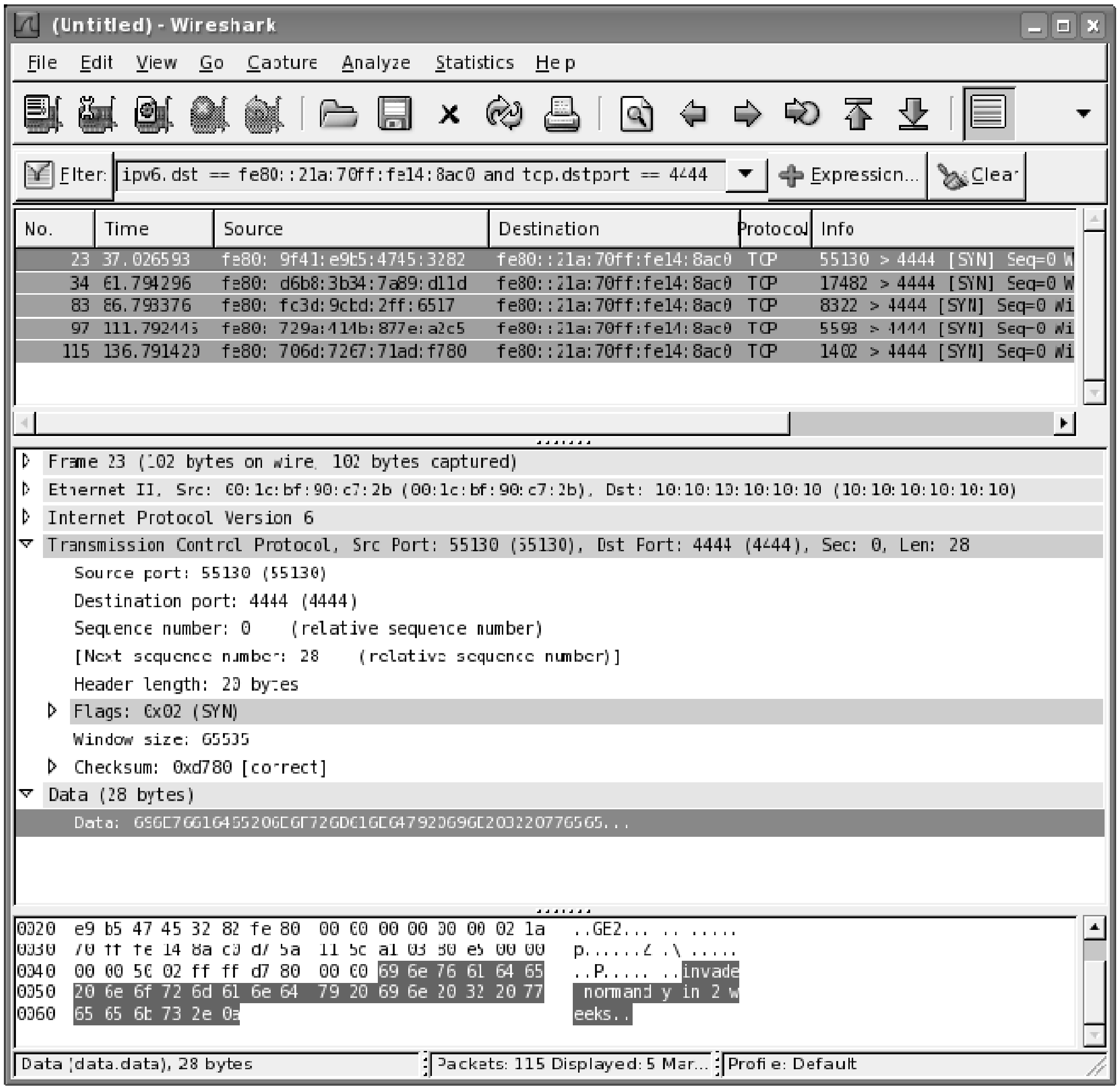
User B did not specify to capture indefinitely. The file and its contents are shown in Figure 23: Direct Encoding: Resulting Message File and Contents.


Shell - Konsole
Session
Edit
View
Bookmarks
Settings
Help
Shell
Filter specifies which packets to sniff.
Five pieces of message each received in different packet.

```
root@Ansalon:/home/bpafoshizle/stego/backup/source# ./m2a.pl
DEBUG MODE: PRINTING RECEIVER SETTINGS...
Mode: decode
Interface: eth3
Mac Decode: 0
Encryption: 1
Keyfile: /etc/m2aKey
Sniffing Filter: ip6 and dst port 4444
Capture mode: live
Sniff indefinitly: 0
Message from Iface ID: 9f41e9b547453282
Ciphertext after pack: AéµGE2
Plaintext after decrypt: <30>inva
Message from Iface ID: d6b83b347a89d11d
Ciphertext after pack: Ö¸;4zÑ
Plaintext after decrypt: de norma
Message from Iface ID: fc3d9cbd02ff6517
Ciphertext after pack: ü=½ÿe
Plaintext after decrypt: ndy tomo
Message from Iface ID: 729a414b877ea2c5
Ciphertext after pack: rAK~¢Å
Plaintext after decrypt: rrow. da
Message from Iface ID: 706d726771adf780
Ciphertext after pack: pmrgq-÷
Plaintext after decrypt: wn
root@Ansalon:/home/bpafoshizle/stego/backup/source# 
```

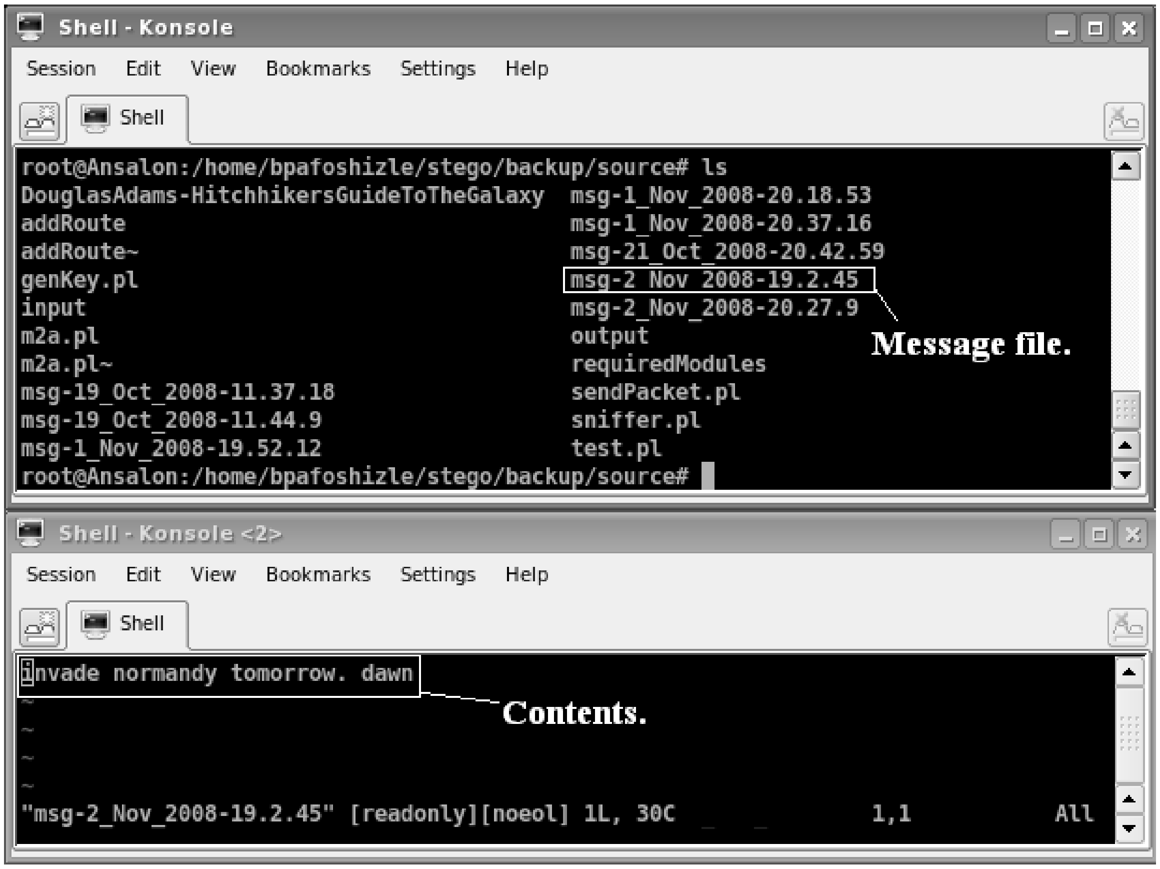
Figure 21: Direct Encoding: User B (Receiver) – Program Output

The next screenshot, Figure 22: Direct Encoding: User B – Packet Capture, shows the packets captured on the receiver side. This is the same information shown in Figure 20: Direct Encoding: User A - Packet Capture above for the User A, because the same filters are used to capture packets.



```
ipv6.dst == fe80::21a:70ff:fe14:8ac0 and tcp.dstport == 4444

No.   Time        Source                      Destination                 Protocol  Info
   23 37.026593   fe80::9f41:e9b5:4745:3282   fe80::21a:70ff:fe14:8ac0    TCP       55130 > 4444 [SYN] Seq=0 W
   34 61.794296   fe80::d6b8:3b34:7a89:d11d   fe80::21a:70ff:fe14:8ac0    TCP       17482 > 4444 [SYN] Seq=0 W
   83 86.793376   fe80::fc3d:9cbd:2ff:6517    fe80::21a:70ff:fe14:8ac0    TCP       8322 > 4444 [SYN] Seq=0 Wi
   97 111.792445  fe80::729a:414b:877e:a2c5   fe80::21a:70ff:fe14:8ac0    TCP       5593 > 4444 [SYN] Seq=0 Wi
  115 136.791420  fe80::706d:7267:71ad:f780   fe80::21a:70ff:fe14:8ac0    TCP       1402 > 4444 [SYN] Seq=0 Wi

Frame 23 (102 bytes on wire, 102 bytes captured)
Ethernet II, Src: 00:1c:bf:90:c7:2b (00:1c:bf:90:c7:2b), Dst: 10:10:10:10:10:10 (10:10:10:10:10:10)
Internet Protocol Version 6
Transmission Control Protocol, Src Port: 55130 (55130), Dst Port: 4444 (4444), Seq: 0, Len: 28
    Source port: 55130 (55130)
    Destination port: 4444 (4444)
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 28    (relative sequence number)]
    Header length: 20 bytes
  Flags: 0x02 (SYN)
    Window size: 65535
  Checksum: 0xd780 [correct]
Data (28 bytes)
    Data: 696e76616465206e6f726d616e647920696e203220776565...

0020  e9 b5 47 45 32 82 fe 80  00 00 00 00 00 00 02 1a   ..GE2... .. .....
0030  70 ff fe 14 8a c0 d7 5a  11 5c a1 03 30 e5 00 00   p......Z .\ .....
0040  00 00 50 02 ff ff d7 80  00 00 69 6e 76 61 64 65   ..P..... ..invade
0050  20 6e 6f 72 6d 61 6e 64  79 20 69 6e 20 32 20 77    normandy in 2 w
0060  65 65 6b 73 2e 0a                                  eeks..

Data (data.data), 28 bytes   Packets: 115 Displayed: 5 Mar...   Profile: Default
```

Figure 22: Direct Encoding: User B – Packet Capture


Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Message file.
Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Contents.

```
root@Ansalon:/home/bpafoshizle/stego/backup/source# ls
DouglasAdams-HitchhikersGuideToTheGalaxy  msg-1_Nov_2008-20.18.53
addRoute                                  msg-1_Nov_2008-20.37.16
addRoute~                                 msg-21_Oct_2008-20.42.59
genKey.pl                                 msg-2_Nov_2008-19.2.45
input                                     msg-2_Nov_2008-20.27.9
m2a.pl                                    output
m2a.pl~                                   requiredModules
msg-19_Oct_2008-11.37.18                  sendPacket.pl
msg-19_Oct_2008-11.44.9                   sniffer.pl
msg-1_Nov_2008-19.52.12                   test.pl
root@Ansalon:/home/bpafoshizle/stego/backup/source#
```

```
invade normandy tomorrow. dawn
~
~
~
~
"msg-2_Nov_2008-19.2.45" [readonly][noeol] 1L, 30C             1,1           All
```

Figure 23: Direct Encoding: Resulting Message File and Contents

As part of the testing for the Direct Encoding process, the entire 287056-byte e-book text file for *The Hitchhiker's Guide to the Galaxy* by Douglas Adams was sent out on a 1-second interval. The whole process took approximately 10 hours; however, the resulting file contents exactly matched the original.

6. CONCLUSIONS

6.1 Summary

As a result of the research and work done for this thesis, the possibility of a covert channel within the IPv6 protocol has been made reality, and debate around the issue can be discussed in the realm of the actual instead of the theoretical. The implementation of exploiting the covert channel inherent in IPv6 shown in this thesis is one way of many ways possible. Practical end uses for such an exploit will only be determined by the end user and their intentions, but the importance of the awareness of this possibility cannot be overstated. It is crucial for users of technology to understand different uses that the technology makes possible, whether the designers of the technology intended a given use or not. If the world wishes to responsibly implement a system on such a wide scale as IPv6 that will affect the lives of over one billion users [11] whether they are aware of it or not, then it is the duty of the implementers and maintainers of such a system to fully understand it. Full understanding is not hard to come by in an open system such as the Internet Protocol version 6, where anyone from the community that has a desire may closely inspect the inner workings. This thesis demonstrates the community acting responsibly to increase awareness and understanding of protocol that will soon be used by billions.

6.2 Risk Mitigation and Countermeasures

One way that this type of exploit could be partially prevented is by having all routers connected to end nodes use DHCPv6 instead of Stateless Address Autoconfiguration. When DHCPv6 is used, the person in control of the subnet has complete control over addresses assigned to end nodes on that subnet including the interface identifier portion of the addresses. This would prevent the MAC encoding method, because the message embedded into the MAC address would no longer be encoded into the interface identifier automatically by the operating system. This would not necessarily prevent the direct encoding (Active Injection) method from working, however. Because the direct encoding process spoofs the source address of packets it creates, independent of the actual source address of the interface, it is more difficult to prevent. One way to prevent this method of transmitting messages would be to configure all, or at the very least end routers, to check the source addresses of all packets being routed off of the subnet to make sure that the source addresses actually correspond to physical nodes on the network. This could be accomplished by a lookup in the routing table, perhaps, but the overhead involved is likely undesirable for most.

Another way to mitigate this risk would be to put in place an “active warden” at the edge of the network controlled by the entity wishing to prevent steganographic messages from leaving that network. This would be something similar to a network address translation (NAT) router or firewall that actually does not allow internal nodes to interface directly with nodes outside the network. In other words, nodes outside the network could not see the true addresses of any of the internal nodes, and could only communicate with the border node. The border router would have to keep state and perform lookups to correctly route information. This is similar to how a NAT would handle the translation of addresses from different internal ones to a common external one, and this process could cause unwanted delays from the overhead involved. Moreover, one of the goals of IPv6 is to eliminate the NAT to make the network layer more “pure” among other reasons.

6.3 Contributions

This thesis contributes to the information security field in several ways. Experience is the most important form of human knowledge, if not the only form, and things are most believable when they can be seen. As a result of the demonstration of a covert channel within IPv6, steps can be taken by the development community or by end users to mitigate this risk using a variety of methods, some of which are discussed above in Section 6.2. An awareness of the problem and a demonstration of how the process could work gives considerable power to those who wish to stop this type of activity, because stopping something becomes much easier if those wanting to do the stopping have a fuller understanding of that which would be stopped. At the very least, if no preventative measures are taken, then reactive measures can be, if a security incident involving secret messages steganographically embedded into IPv6 addresses is detected or suspected. Those monitoring a network, whether it be school officials, counterterrorist organizations, or corporate network administrators, will be one step closer to solving their security breach by being aware of the possibility of such an attack.

On the other side, people all over the world who are oppressed, censored, or persecuted by their government could find a tool such as the one designed for this thesis invaluable for communication. In places where standard communication channels are closely scrutinized and spied on by ruling regimes, people who publicly speak out against the status quo or the government are persecuted as is reportedly happening in China today [12]. In situations such as these, a secret and non-standard communication channel becomes all the more important for those who value freedom, privacy, and individual rights. In an information age, those with complete control of information become sole arbiters of truth.

6.4 Future Work

This thesis builds on the ideas of Jane Lindqvist to make the possibility of a covert channel that he foresaw in IPv6 into a reality [2]. While this thesis implements the basic framework for encoding and decoding messages across an IPv6 network as a proof of concept, more work could be done in the way of making the associated program more efficient, robust, or easy to use, or in implementing countermeasures for this kind of process. A few possible extensions are listed below.

- Add support for transmission and decoding of Binary Files
- Implement a log cleanup option that will erase any log history the program creates.
- Allow sending user to specify any arbitrary prefix to use for packets. This would be trivial to implement.
- Implement for IPv4. IPv6 is not the only way to embed secret messages into network metadata. MAC Encoding mode will embed into IPv4 just as easily for local area networks, because network layer packets wrap the lower layers including the link layer containing the MAC address. If deep packet information is extracted by a node instead of it just looking at the network layer information, then messages can be sent over IPv4 as well, though not in the network layer (at least, not as easily using the method already

implemented for this thesis). The underlying packet creation Perl framework Net::Packet allows for easy creation of IPv4 packets as well as IPv6. Since all packet properties may be specified, the message could be embedded in any field of any layer desired, and could easily go out over an IPv4 network.

- Implement Long MAC decoding. The main hindrance to this is presentational, and stems from the fact that there was no immediately clear solution for printing out a large tree representing the possible message. The display could be performed with a little effort or searching around for a library or module capable of printing tree data structures in some intuitive way.
- Another related interest to this project would be to perform statistical analysis on the covert channel of packets flowing across an IPv6 network to determine whether it is possible to detect when a secret message is being passed. This would probably work well for unencrypted messages because plaintext has predictable patterns such as letters that are used more than others.

REFERENCES

[1] I. van Beijnum, “Everything You Need to Know about IPv6.” *Ars Technica*, [Online Document], pp 2, (2007 Mar), [cited 2008 Nov 3] Available: http://arstechnica.com/articles/paedia/IPv6.ars/2

[2] J. Lindqvist. “IPv6 is Bad for Your Privacy,” presented at *Defcon 15*, Las Vegas, NV. 2007. [Online]. Available:http://www.defcon.org/images/defcon-15/dc15-presentations/Lindqvist/Whitepaper/dc-15-lindqvist-WP.pdf

[3] S. Deering and R. Hinden, “RFC 2460: Internet Protocol, Version 6 (IPv6) Specification,” Dec. 1998, Available: http://www.ietf.org/rfc/rfc2460.txt?number=2460

[4] S. Thompson, T. Narten, and T. Jinmei, RFC 4862: “IPv6 Stateless Address Autoconfiguration,” Sep. 2007, Available: http://www.ietf.org/rfc/rfc4862.txt?number=4862

[5] S. Deering and R. Hinden, RFC 4291: “IP Version 6 Addressing Architecture,” Feb. 2006, Available: http://www.ietf.org/rfc/rfc4291.txt?number=4291

[6] C. Schroder, “Getting Acquainted with IPv6,” in *Linux Networking Cookbook*, 1st Edition. Sebastopol, CA: O’Reilly Media, Inc., 2008, ch. 15, pp. 437-451

[7] B. Dunbar, “A Detailed Look at Steganographic Techniques and their use in an Open Systems Environment,” SANS Institute, Jan. 2002

[8] S.J. Murdoch, S. Lewis, “Embedding Covert Channels into TCP/IP”, Information Hiding Workshop, July 2005.

[9] T. Narten and R. Draves, RFC 3041: “Privacy Extensions for Stateless Address Autoconfiguration in IPv6,” Jan. 2001, Available: http://www.ietf.org/rfc/rfc3041.txt?number=3041

[10] V. Jacobson, C. Leres and S. McCanne, et al, “Tcpdump – Linux Man Page,” unpublished, Available: http://linux.die.net/man/8/tcpdump

[11] Miniwatts Marketing Group, “Internet Usage Statistics,” [Online Document], [cited 2008 Nov 3], Available: http://www.internetworldstats.com/stats.htm

[12] Amnesty International USA. (2004, Jan.). Peoples Republic of China Controls tighten as Internet Activism grows. [Online]. Available: http://www.amnestyusa.org/document.php?id=6219A12C7651806380256DFE00581835&lang=e

A. APPENDIX CONFIGURATION OPTIONS

The table below shows options that can be used when running the program. These options can be specified by either command line or in a configuration file '/etc/m2a.conf'.

Command Line	Config File Value	Description
-i	interval	Interval in seconds to send messages out on. If left 0, then the default preferred lifetime of IPv6 temporary addresses is used
-f	msgfile	Input File Name
-if	interface	Link layer interface to use
-clean	txfast	Specifies wither to use the faster transmit method with a more complicated decode, or the slower transmit method with a clean decode. On the decoding side, it tells the decode which mode to assume the message was transmitted as. Only applicaple when encoding with macencode set to 1 or decoding with macdecode set to 1.
-cs	cnctscript	User may specify a connect script to connect to the internet
-mac	macencode	User may specify to encode message into the Mac address in which case it will go out over IPv4 as well
-m	mode	Specifies mode to use: either msg, inject, testFormat, or decode. If msg the behavior is to embed a message into the source address field of packets in one of two main ways - either though manipulation of the mac address or through direct insertion into the interface ID portion of the source address field of a packet. If injection mode is used, no message is inserted and the user may set the source address field (among others) to whatever value desired.
-s	srcAddr	Specifies the value for the source address field (only used in injection mode)
-d	dstAddr	Specifies the destination address field's value (only used in direct msg mode or in injection mode)
-sp	srcPort	Specifies the source port value for a frame (only used in direct msg mode, or in injection mode)
-dp	dstPort	Specifies the destination port value for a frame (only used in direct message mode, or in injection mode)
-t	transport	Specifies the transport layer protocol to use only either UPD ('udp') or TCP ('tcp') currently.
-sm	srcMAC	Specifies the source MAC address to use. If none is specified, then the MAC for the specified interface is used. LSB of the first octet cannot be 1. Used only in direct message mode or in injection mode.
-dm	dstMAC	Specifies the destination MAC address to use. If none is specified, then a random one is generated. LSB of first octet cannot be 1. Used only in direct message mode or in injection mode.

-scope	scope	The scope of the prefix to use. i.e. link, site, global,or local. If none is specified, then link local (fe80) will be used.
-tcpOpts	tcpOpts	The TCP options to use as a hex string.
-dataFile	dataFile	A file that contains data to be used for the application layer object. It is assumed that if a data file is specified, then it will be inserted into the packet(s).
-e	encrypt	Flag to signal if the user wishes to encrypt the message before embedding it into the medium. The Blowfish encryption algorithm will be used.
-k	keyFile	The keyfile containing the key and initialization vector that will be used to encrypt and decrypt the data. This file is required for decryption.
-fs	filterString	Filter string that will be used by the sniffer to selectively grab packets.
-c	capture	Mode to capture from. Used by the dump object to know whether to capture from a live network device or a pcap file (pcap file must be specified if offline capture. Choices: 'live' or 'offline'
-pc	pcap	the pcap file to open and use by the dump object to search for a message.
-md	macDecode	Tells program to assume macencode mode was used. This attribute only applies to decode mode.
-cd	continuousDump	If set, then in decode mode, the program will run indefinitely searching for messages. Otherwise it will quit after the first message ending with a newline followed by the string 'end'.
-dbug	dbug	Specifies whether to go into debug mode and print very verbosely what is going on in the program.