2004

Limiting DNS covert channels and network validated DNS

Rex D. McCracken
West Virginia University

Limiting DNS Covert Channels and Network Validated DNS

Rex D. McCracken

Thesis submitted to the
College of Engineering and Mineral Resources
At
West Virginia University
In partial fulfillment of the requirements
For the degree of

Master of Science
In
Computer Science

Roy Nutter, Chair
John Atkins
Bojan Cukic
Todd Montgomery

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2004

Keywords: DNS, covert channels, real-time network analysis, packet validation, protocol validation, network validated DNS

Abstract
Limiting DNS covert channels and Network Validated DNS

Rex McCracken

Despite the variety and number of network security devices and policies available, sensitive data, such as intellectual property and business data, can still be surreptiously sent via the Internet to unscrupulous receivers. Furthermore, few security mechanisms address securing or limiting covert channels. This study defines a framework for determining a rule set to minimize covert channel capacity on the DNS protocol specifically. The information and techniques used in this study may be useful in aiding security professionals and developers with enforcing security policies on DNS and other Internet protocols.

This research resulted in the development of a rudimentary tool, referred to as NV-DNS, capable of detecting and effectively limiting the capability of covert channels in DNS communication packets.

Acknowledgements

First and foremost, I would like to thank God for giving me the strength, wisdom, and mental clarity to complete this task I undertook. Secondly I would like to thank my chairman, Dr. Nutter, for helping me choose a topic suitable for me and capable of being completed in a realistic timeframe. Without his guidance, I would still be working on an overly grandiose project.

I would also like acknowledge my committee and thank them for their belief that I can create a high-quality research work. Dr. Atkins guided me through my early days as an undergraduate and a graduate, helping me choose the courses I should take and steering me away from those he deemed detrimental. Dr. Cukic opened my eyes to computer security issues all around and security in general, giving me a much larger perspective on the role of security in daily operations. Todd Montgomery is also an excellent instructor and taught me the basics of networking and multicast, as well as being an all-around nice guy.

I want to thank my family and friends for understanding the stress I was under during the creation of this document. Mom and Dad, thanks for giving me the room to work and vent my frustrations out, as well as the unwavering faith that I would figure the problem at hand out and complete the task at hand. Ben, thanks for the suggestions for finding problems in my code. It helps to have a voice reminding you of the things that you already know but forget during stress. Julia, thank you for being zany; it is more of a stress-reliever than you know.

Danielle, I know it was anything but a joyride to be around me for the past few months, but I thank you for your patience and understanding. I won't be such a grouch to be around now that I don't have any impending deadlines beating the sanity out of me.

I'd also like to thank a few of the many students I have met during my time at WVU: Greg, Dave, Chris Delche, Jesse, John, Jay, and the many others that I don't have the space to mention.

Dedication

This thesis is dedicated to my parents, Ben and Marion McCracken, who have guided, supported, and encouraged me to the completion of one of the most difficult tasks of my life thus far. I wholeheartedly appreciate everything that both of you have done and continue to do to help me grow, mature, and achieve my goals in life. I love you.

Contents

List of figures

List of tables

Chapter 1
1 Introduction

Running a business in the 21st century is different from any other time in history. Years ago, merchants and vendors could assure themselves a piece of the market share by simply carrying a wanted product that no other vendor carried. Few businesses anymore have the fortune to be the only vendor marketing a unique product to the public as imitators and competing products are introduced to the masses through various advertising venues, hauled to a local shopping mall, or shipped directly to the consumer's home. Regardless of a business' financial classification or products, competing vendors, manufacturers, and even charities would all love to have something from their competition that is more valuable than a single product: intellectual property. [1]

Intellectual property is a major portion of the foundation on which every business grows and develops. Every business has some form of intellectual property, be it customer information lists, purchasing prices, diagrams for a new piece of equipment under development, or a new manufacturing process. Intellectual property therefore becomes the advantage that a particular business has over its competitors in order to continue to exist in the business place. Thus the largest threat to a company's competitive edge and existence is the loss or leakage of their intellectual property to a competitor. Internet access is available in nearly every business location and is used in daily operations of a business. While beneficial to the operation and expansion of a company, the speed and stability of the communications afforded by internet access also allows a malicious employee to send a company's intellectual property with relative ease to

someone external to the company without dealing with some type of physical security mechanism. [2]

E-mail, FTP, HTTP, and instant messaging programs are the easiest and most common methods of transferring large amounts of data across the Internet. E-mail accounts are typically given all company employees for company use and nearly every e-mail program available has the ability to send file attachments of any size, hampered only by the recipient's e-mail file size. FTP communications provide a direct means of efficiently transferring files of any size from client to server or vice versa and does not suffer from the size limitations e-mail is potentially subject to, but requires a valid IP address or domain name in order to successfully initiate communications.

HTTP communications allow for the same possibilities as FTP and E-mail and is commonly known as "web surfing." The newest communication class proliferating on network channels is instant messaging programs such as AOL Instant Messenger, MSN Messenger, and Yahoo Messenger, just to name a few. The basic function of these programs is to send and receive text messages from one username to another. Early versions of the messaging programs lacked file transfer and encryption capabilities, but recent revisions of the software have incorporated these features as well as video and audio capabilities. Spin-offs and clones of these software programs have similar capabilities.

E-mail, FTP, and HTTP traffic are the most commonly used methods of transferring data and are also the most commonly logged actions on a company network. User's e-mails are typically stored on the main e-mail server and archived for later retrieval or restoration. Employees sending out attachments of private company data

through e-mail can quickly be traced through a combination of username and Internet Protocol (IP) address tracing. Unauthorized FTP connections and traffic pushing data out of the company network can easily be traced through the router logs or denied in the router settings, thus limiting the effectiveness of FTP as a means of transferring files. HTTP connection pushing out large amounts of data from non-server computers raise their own questions, and are subject to the same requirements that FTP traffic falls under. Instant messaging programs are slightly more difficult to block or protect against, but most programs send messages and files without encryption over the network and can be reconstructed using packet sniffing software.

Routers and firewalls are typically the main defensive lines any network administrator utilizes to help minimize an intruder's foray into the business' LAN. Basic firewalls and routers can easily be set to forward, allow, or deny access to any combination of thousands of ports and many guides, websites, and other articles are available to help customize these settings based on the business' network topography. Advanced networking equipment and software can further enhance and extend the basic permit and deny rules using technologies like Stateful Packet Inspection, bandwidth throttling, and intrusion detection systems at an often significant financial investment for more capable hardware and software. Despite the range of technologies available to combat external intruders and malicious internal users, there is a basic and integral service of the Internet that is typically unmonitored by businesses and security professionals alike.

DNS, or domain name service, is an integral part of the workings of the internet. Every time a user attempts to connect to a site, be it google.com, wvu.edu, or any other of

a number of sites, a query is sent upwards through a hierarchy of computers and routers to try and find the IP address to which a specific domain name is mapped. Without DNS, users would be forced to remember a series of numbers for each site they wanted to visit. Attempts to remember Google (64.233.167.99), WVU (157.182.140.235), and ZDNet (216.239.115.140) would quickly devolve into random number guessing and make Internet connectivity little more than a passing interest, especially should the IP address of the domain change.

As a necessary service for the Internet to work, the DNS protocol specification, as approved by the Internet Engineering Task Force (IETF), uses an iterative hierarchical model to process domain name to IP address queries. DNS takes a domain name as an alphanumeric string and iteratively queries DNS name servers, returning a valid IP address of the form x.x.x.x, where x is a number from 0 to 255 when the DNS server returns a positive response. The protocol's specification answered a large number of questions concerning the handling of future growth of the Internet, but designers knew they could not foresee all the potential possibilities of DNS. With this in mind, designers chose to leave room for future revisions and additions to the protocol in the form of undefined or reserved bits, bytes, ranges, and experimental options. [3]

The protocol's specification calls for these unused bits and bytes to all be set to zero, but failure to properly set these bits will not result in failed responses as systems receiving DNS queries and responses ignore the undefined portions of the DNS packet. The unspecified bits coupled with an un-enforced request allows for data-stuffed packets to be created and sent on without disturbing the packet's validity as seen by the routers and DNS servers. A crafty employee could utilize these gaps in the protocol to send out

seemingly innocent DNS requests while simultaneously sending off bits and pieces of company data. The practice of hiding a communication within a medium is known as creating a covert channel.

The focus of this research is on the detection and logging of covert channels in DNS to aid security professionals and software development personnel faced with the daunting task of controlling unauthorized communications through design and application. This research is also being done in response to an article reported on August 2nd 2004 concerning the ability of DNS to contain covert channels and the lack of monitoring of this particular protocol. [4]

1.1 Definitions of Steganography and Covert Channels

1.1.1 Steganography

Steganography is the art of inconspicuously hiding data within data, literally meaning "covered writing". [5] The overall goal of steganography is to hide the message within the data well enough such that unintended recipients do not suspect a hidden message exists. [6] A simple text message using steganography to hide a message may be as follows:

```
Prepare and shutdown system workstations or remote
datalinks : Storms will offline realtime data feeds.
Interrupted several hours.
```

The body of the text hides the message "password : swordfish" using the first character of the initial word and the first character following a space. The steganography method is simple and for the most part, so is its detection. [6] notes that an important part

of steganography's success is the naïve attitude of human beings by not accepting the possibility that there is more than meets the eye occurring. Several other simple text-based steganography techniques include word spacing and invisible characters. Images, audio clips, and movies employ steganographic techniques such as least significant bit (LSB) alterations, color palette modifications, and manipulation of the compression algorithms. [5]

[5] states that "information hiding within electronic media requires alterations" that introduces degradation or unusual characteristics. These anomalies in the media can be viewed as signatures broadcasting a hidden message's existence within a communication; a point directly countering the goal of steganography. Steganalysis can be defined as "attacks and analysis on hidden information" and is the counter to steganography. The goal of steganalysis is to examine an image, message, or other medium that could potentially contain a hidden message, determine if a message exists, extract the message if possible, or disable or remove the hidden data. To determine if a hidden message exists, steganalysis relies heavily on statistical analysis of the medium's properties to determine if any anomalies exist and if these anomalies are naturally occurring or the result of tampering. [5]

Steganography is similar in nature to cryptography in several ways, but the two methods are exclusive. Both methods are concerned with passing a message from party A to party B without party C being able to understand the message, but the means by which the messages are passed is where the difference lies. Steganography's means of communication relies on stealth and naivety for all non-intended recipients to ensure the security of a private message. Cryptography's means of communication relies on the

strength of the encryption algorithm and key management to secure the message and makes no attempt to hide the existence of the communication.

Beyond that main difference, steganography and cryptography follow an algorithm and "key" to encode and later decode the data. For cryptographic messages, the key is the password, while steganography's key is the location of the hidden data. Cryptographic techniques are beyond the scope of this document and will not be discussed.

1.1.2 Covert Channels

Covert channels are the application of steganographic techniques to communications mediums. Digital communications protocols used on the Internet are filled with unused and reserved bit, bytes, open-ended options, and devoid of any type of validation mechanism to verify the correctness of these communications. Thus, a steganographer can piggyback their communications onto a valid communications protocol piece by piece and slowly cart off bits and pieces of data without much worry of being detected at all. The ease with which a steganographer can piggyback their message onto an existing protocol depends primarily upon the protocol's syntax and usage. This document discusses the DNS protocol and some network topology, making general references to the protocols upon which DNS traffic operates, TCP and UDP.

The Domain Name Service, or DNS, is an Internet Protocol (IP) utilizing the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP) for sending and receiving domain name translation requests. TCP is a "connection-oriented" protocol, designed to ensure the delivery and ordering of transmitted packets from sender to receiver. [7] UDP is a "connectionless" protocol that predates TCP and does not

ensure the delivery of packets or ordered reception of received packets. [8] UDP is the protocol primarily used to transmit DNS messages as UDP does not require the overhead setup that TCP uses to setup the reliable connection. TCP is used in zone and domain transfers where a large amount of data is passing from server to server and the data must be carefully preserved. [3], [9]

1.2 Statement of Problem

The goal of this research is to show the weaknesses in the DNS protocol and provide a methodology for protecting this and future communications protocols from subversion for use with unauthorized communications.

1.3 Organization

The remainder of this thesis will be organized as follows:

- Chapter 2 will be a literature review that will form the background for this research. The concepts of steganalysis, protocol validation, and real-time network analysis will be discussed regarding current research in the fields.
- Chapter 3 will discuss the application of steganalysis and protocol validation to the Domain Name Service and other per-packet protocols on a network.
- Chapter 4 will provide a description of the architecture of NV-DNS and its usage on a network segment.
- Chapter 5 will discuss the results of NV-DNS as applied to several example problems.

- Chapter 6 will be a final discussion and conclusion of this particular work and will include suggestions and extensions for the future to be even more flexible and usable in the real world.

Chapter 2
2 Literature Review

This chapter provides a review of previous research in the fields of steganographic detection, protocol validation, and real-time network analysis, which form the foundation for the work conducted in this thesis. This chapter's goal is to provide the reader with a basic understanding and background of each area by discussing several concepts related to each field.

2.1 Steganography

This section will provide a brief description of the current research in the area of steganography and its detection.

2.1.1 Steganography Overview

Individuals have long been concerned with keeping messages and other communications from prying eyes. This desire to secure communications fueled two different means of achieving the same end. One method commonly used to secure digital communications and data is encryption. With encryption, a private message is scrambled using some type of algorithm with a sequence of bits known as a key that serves to configure the algorithm. This methodology of securing a message hopes to guarantee privacy by the strength of the encryption algorithm and key complexity.

Steganography pursues a different means to achieve the goal of private communications. Message security in a steganographic model is achieved by hiding the existence of the message in the first place. [6] defines steganography as "the art of inconspicuously hiding data within data" such that unintended recipients do not suspect

the existence of hidden data. The exact process of hiding the information varies from medium to medium and type to type, but the basic steganographic process requires three items: cover medium, hidden message, and embedding algorithm. An optional fourth item required is a key.

The cover medium in the digital realm is some type of object, be it an image, audio, video, text file, data packet, or other file or grouped item or element. A hidden message can be text, a file, or any other data type, including single bits. The embedding algorithm is typically a cover medium-specific algorithm designed to take advantages of redundancies and unused bits and bytes specialized to that particular piece of data. The optional key can be used to encrypt the data, configure the embedding algorithm, or both.

The exact process of hiding the information varies from medium to medium, but the basic design starts with an analysis of the cover medium. The process finds the bits unnecessary to maintain the medium's integrity, named the redundant bits, and then the embedding process replaces the redundant bits with bits from the hidden message. [10] Based on the cover medium, this embedding process varies in its usage, analysis, and replacement.

The cover medium is also known as the carrier type, as it is the item which "carries" the steganographic message. The carrier type can literally be anything physical (wax on wood tablets) or intangible (bits of data in data communications). History tells of early steganography attempts by prisoners that hid messages on the wood of the wax-covered tablets they were given. Another account tells of how a Roman general shaved a slave's head, tattooed a message, and then sent him to deliver the message after the slave's hair had grown back. Suggested recent physical uses of steganography were

used in the quilts hanging outside the homes of those aiding slaves in the Underground Railroad, giving them directions and instructions in plain view while keeping their existence hidden from those unaware of the messages. [11]

Steganography has taken on a new dimension with the explosion of computing technology and the plethora of digital carrier types the computing growth has created. Digital cover mediums span a variety of categories and are continually expanding. Common carries of steganographic messages include the following types listed below.

- Images – Gifs, JPEGs, and other image file formats typically have some type of redundancy. Manipulation of these images relies on minute shifts that do not drastically alter the image.
- Audio – While images are effective for steganography, the human ear can be less discernable, especially when trying to listen to quiet background noises under a loud foreground sound.
- Video – The sheer size of video files as well as the encoding methods used permits a much larger amount of data to be stored.
- Text – Fonts, spacing, and even word and line ordering can hide a message.
- Network Communications – Many network protocols have unused bits and bytes reserved for future use and rely on honest implementation and "best practices"

The focus of this research is on network communications, but it is important to be aware of the other applications of steganography in a digital medium.

Regardless of the medium used, some type of encoding schema must be used to integrate the hidden data and the cover medium. There are a variety of encoding schemes available and discussed in the academic community, some of which are applicable to

multiple carriers, many of which are limited to a specific carrier type. Common steganographic encoding methods for images, audio, and video include Least Significant Bit (LSB) and wavelets.

The simplest of these methods is Least Significant Bit substitution where the low-order bit in all or some bytes is replaced by bits from the hidden or secret message. Since the human eyes and ears are only so discerning and sensitive, the least significant bits of the object can typically be modified without worry of a person being able to detect it visually or audibly. However, a computer program can look directly at these bits and reconstruct them to form a message or file quickly and easily.

The most complex and potentially rewarding method of steganography lies in a mathematical modeling concept known as a wavelet. Wavelets are functions satisfying specific mathematical requirements and are used in the representation of functions as well as other data. [12] discusses the foundations of wavelets and their ability to see "the forest and the trees." He also mentions the ability to choose a wavelet best suited for a particular data set or truncating the coefficients below a specific threshold to achieve data compression. Research by [13] shows that wavelet manipulation can compress an object to a maximum compression without threatening a single bit of the hidden message. This capability of wavelets is an important characteristic which will be discussed shortly.

Data masking and filtering techniques are not discussed here as these techniques extend data over a cover medium. These are commonly used as a form of digital watermarking and do hide information, but technically are not considered forms of steganography. They will not be discussed further in this document.

Medium- and format-specific steganographic algorithms are the most common techniques available used to encode data. GIF image files commonly have their color palette modified or reordered to hold hidden data. JPEG images can have hidden messages intertwined in the data coefficients describing the image pixels. MP3s and WAV files are susceptible to hidden messages being encoded quietly during loud portions of a musical crescendo or other loud noise. Text documents can hide steganographic data through line spacing, character fonts, text positioning, and even letter arrangements. Data packets and protocols with loosely defined or undefined bits and bytes can transmit information untouched by routers or end systems relaying the information. The list of steganographic mediums and algorithms continues on and on, including things such as redundant instruction sets in program executables and even file system tables and definitions.

Regardless of the medium chosen, steganographic techniques can usually be applied to the medium to encode a message within. While steganography itself is a powerful means of securing a message, it can be strengthened. Encryption algorithms can be used in conjunction with steganography methods to further scramble and more effectively hide the embedded message. Prior to the embedding the hidden data in the cover medium, the encryption algorithm of choice is run on the hidden message to encrypt its contents. Once encryption is complete, the scrambled message is embedded in the cover medium using the algorithm for the specific medium. Some encryption algorithms used include DES, 3DES, and AES.

The innocuous nature of steganography is also its main weakness. Embedded data in a cover medium is extremely fragile and is usually threatened or even destroyed

by any modifications to the cover medium. Text documents and GIF images are often irreparably altered simply by viewing the file and then saving it without making even a single edit. [5] Though there are some encoding methods which provide reliable robustness against destruction of the hidden message, many techniques are highly susceptible to attacks on message integrity. Despite this weakness, these mediums and others are still used to try and pass hidden messages.

2.1.2 Network Covert Channels

Covert channels encompass all types of communications, such as inter-process, input and output, and even hardware communications. Network covert channels are a specific focus area in steganography and covert channels using communications protocols as the medium for sending covert messages. Network and Internet communications are not the only types of covert channels available, though many of the same theories apply. For the purpose of this research, network covert channels will be the main type of covert channels discussed.

[14] defines network covert channels as exploits "by the manipulation of communications resources or transmission characteristics" of a communication protocol. Manipulation of the communications resources includes modification of any of the fields or values defined in the protocol specification to something other than the expected usage. Transmission characteristics include miniscule items such as timing between packet transmissions and the size of the packets being transmitted.

Most of the research found concerning covert channels was concerned with detecting and disrupting the timing of communications as a means to combat time-based communications. [14], [15], and [16] all discuss timing as the methodology of choice to

secretly transmit data. [17] briefly mentions a second means of secretly encoding data by placing it in network headers, one of the early examples of network covert channels.

The paper continues with a breakdown of covert channel research into four disciplines: Explanation, Identification, Measurement, and Mitigation. [14] also provides some methods of measuring a channel's capacity and provides a concept useful in combating not only timing channels, but a time-independent item as well: the introduction of noise into a channel.

2.1.3 Steganalysis

Directly countering the art and science of steganography is steganalysis. The goal of steganalysis is to detect, extract, or disable steganography in any type of medium. Regardless of how good the steganography method is, [10] says all steganographic methods are invasive and therefore leave evidence of their actions. The most difficult portion of steganalysis is reliably and accurately detecting an embedded message. There are two different theories driving steganalysis detection: statistical review and information theory.

Based on the medium, there are a number of characteristics that can be gleaned from analysis of said medium to create a statistical template. Some characteristics used in image analysis include luminance, coefficient values, and even image continuity. These characteristics are then compared to values and tolerances previously generated, and any values or characteristics exceeding the threshold are flagged and dealt with accordingly. [10] Depending on the threshold and characteristic values chosen, the number of erroneous responses (false negatives and false positives) may lead analysts on a wild goose chase or may keep them from analyzing objects containing hidden data.

Information theory was pioneered by a researcher named Shannon beginning in the 1940s. This area of research relates to message sources, communication channels, and theorems relating entropy to channel capacity. The fundamental concept of information theory is that information is regarded as only those symbols uncertain to the receiver in his body of knowledge. A message is analyzed using information theory, the redundancies and predictabilities are removed, and the resulting message is sent, minimizing the message's size while maximizing the number of simultaneous messages that can be transmitted. [18]

For example, the message "We have a large database to transmit containing secret agent records" could be minimized to "We hve lrg db 2 trnsmt cntaing scrt agt recs." While a simplistic example, it helps illustrate the broader uses of information theory when transmitting real-time video for streaming internet or cable or satellite TV, or real-time audio over cellular phones and landlines. Transmitting a studio-quality full-motion video TV signal to a home requires 70,000,000 bits per second, a rate which is economically impractical even using high bandwidth fiber optics connection. Using information theory, this rate can be compressed to 368, 192, or even 56 kilobytes per second. The same idea follows for phones, where voice data is compressed down as low as 64 kilobits per second, if not lower. [18]

Information theory plays a role in steganalysis because of its reducing and compressing nature. Using information theory, analysts can determine the object's capacity to store extra data or determine whether or not the object has redundancies that should have been removed during the original encoding process. By examining the results from each of these processes, analysts can more accurately predict the probability

that the object has information embedded in it. Information theory can be applied to all carrier types and cover mediums with algorithms specific to the type and medium structure or format. For communications protocols, information theory can be used to determine the protocol's capacity to hold hidden data and whether or not the protocol is being used to send extra data. [19]

2.2 Real-time Network Analysis

This section will review research regarding real-time network analysis and supporting areas.

2.2.1 Real-time Network Analysis

Humans have long had questions about the truth of what is really happening during an event. We may think that we know what is happening, but we still want proof that what we think is occurring is truly occurring. Ask any network administrator and he will tell you that one of the things that he wants to know and be sure of is that what he thinks is happening on his network IS happening on his network is happening RIGHT NOW. Like anyone responsible for safeguarding a location, item, or person, finding out that someone was trying to break into the network or was engaging in a denial-of-service attack even as few as five minutes ago is often unacceptable.

This is where real-time network analysis has its stake. The ideal of real-time network analysis is to examine every packet passing through a particular network segment to determine anomalies, show traffic patterns, and respond to changes or unauthorized behavior or traffic accordingly. On small, slow, and rarely-used home networks, this goal may be easily attained; large, fast and busy networks can end up

overwhelming a single system, causing a bottleneck and degrading network responsiveness, throughput, and productivity. The goals of real-time network analysis are to provide as clear a picture of the network and its utilization as possible without having a detrimental affect on any of the network's services, usage, or throughput.

[20] discusses a simple, efficient, and effective real-time network analysis strategy based on a queue structure that samples incoming packets for analysis. By examining packet frequencies, end points, and end ports, a clearer view of how the network is utilized is created at that instant. His simple model shows how a large network can quickly be analyzed for anomalies such as abnormally high traffic patterns or unexpected traffic types between systems. [20]'s case studies were performed on a network gateway with a bandwidth of 7 MB and an average of 1000 to 4000 packets passing through it per second.

Research by [21] on an OC-3 network running at 155 MB was published in 1997. This group's work gathered varying statistics from the network such as average packet size, source information, destination information, and the protocols used to transfer the information over IP. This statistical sampling was again a rather simple analysis of network traffic again centered on the basic information gleaned from the header information on each packet.

The largest hurdle of real-time network analysis is being able to perform an intensive analysis of every packet in real time without degrading the usability of the network. [20]'s research showed a node processing 1000 to 4000 packets per second on average, which translates into an average processing time of 0.00025 to 0.001 seconds to analyze a packet. When gathering statistics on a network, this limitation can be easily

met by a few lines of code and a few conditional statements, all of which can be optimized. The processing time complexity of such a statistical algorithm is low. Performing an in-depth analysis of the packets based on source, destination, protocol, and content validity and integrity has a much larger processing time complexity and can quickly lead to dropped packets and inefficient usage of network resources.

Optimizing analysis code and providing faster support hardware to execute the analysis code are two common methods for overcoming this boundary, but each has their own drawbacks. Optimized code is typically limited to the system architecture it is written for, and faster supporting hardware is often financially burdensome. It is also difficult to analyze gigabit plus data rates used in inter-backbone communications reliably.

The next theory is to split the analysis processing into smaller, more manageable sections such as network segments. While analysis of these smaller, slower segments can be done with older equipment, this benefit is offset by the necessity of the amount of equipment required to monitor an entire network. Instead of a single high-performance system at the entry point to an entire network, there would be multiple inexpensive systems utilized on the network.

[22] reports that network packet analysis requires too much human and hardware resources to be used on anything than a small network segment. Requirements for simply recording all of the traffic without loss of packet data through network saturation or storage delay are difficult enough to reliably achieve. This group's body of work suggests that data analysis post-event allows for a less time-restricted decision to be

rendered. This also gives the analyst a potentially large body of data to support and justify any claims made.

2.2.2 Application of Real-time Network Analysis

The applications of real-time network analysis to the problem of network steganography on the DNS protocol are straightforward. First, we are performing a real-time analysis of the network traffic and attempting to gather usage and packet characteristics. Our particular problem domain is a subset of the traffic monitored by a typical network monitor, traffic running to or from port 53. This is the port the Domain Name Service is commonly available on.

Traffic analysis will look at all packets with a source or destination port of 53 and perform an in-depth review of these packets to validate packet structure based on the protocol specifications and network topology. Here we are first concerned with whether or not the protocol is being utilized properly based on the protocol's specification. Traffic failing to conform to the standard or using undefined options are subject to review. Secondly we are concerned with the traffic's flow with regards to network topology based on services available on a specific network segment. There should not be any DNS requests going to or DNS responses coming from a network segment that does not have a DNS server. Evidence of the failures is likely to show an improperly configured machine, incorrect implementations of the DNS protocol, or rogue applications and servers.

The final application of network analysis to this problem area is a hands-on approach that modifies packet data. This intensive approach permits rogue packets to be caught and hopefully rendered inert as they are forwarded on. Unfortunately this

particular process is extremely invasive, but executed properly results in an entirely transparent process to honest users. Since our goal is to minimize, if not eliminate covert communications over DNS, modification of any weak or easily manipulated fields is an essential portion of our analysis that must be done in real-time.

2.3 Summary

The review of literature provided in this chapter has shown some of the more recent and prominent research regarding steganography, steganalysis, covert channels, and real-time network analysis. This body of work will attempt to unify these areas into a networking tool that will provide a contribution to the field of computer security. It has become apparent from the research described previously that there is very little attention paid to covert channels within network communications. Most research for steganography or covert channels returns some information on network-related fields, but little information was available beyond the basics concerning specific algorithms or communications. With the integral parts that communications protocols play in basic Internet usage, it is disturbing to see that very little research is available to counter the weaknesses inherent in many protocols. Malicious and rogue users are quick to employ and utilize such weaknesses to achieve their own goals, yet very little is done to minimize this risk.

It was also observed that there is a great deal of potential for continuing research and application of current real-time analysis work with regards to security. The current trends of analysis rely heavily on gathering basic statistics from network packets to gain a clearer view of network utilization. It has been shown that analysis can detect unusual

behaviors such as port scanning [20], but little work is listed about the network responding to anomalies such as this by bandwidth throttling, packet dropping, or other techniques. Since we can see the activities occurring in real-time, we should also be able to respond to them in real-time as well.

The remainder of this thesis will be focused on describing a specific algorithm for minimizing the capability of covert channels in DNS as well as a generic algorithm for the achieving the same goal on other protocols. This includes the Network Validated DNS tool, a tool implemented from this algorithm and drawing its inspiration from research discussing the capabilities and bandwidth of covert channels.

Chapter 3

3 Limiting DNS Covert Channels

In chapters 1 and 2 it was observed that there was very little formal research done to minimize the potential for covert channels in networked systems. Most of the research provided means of detection, if possible, and does not provide any methods of disabling or attacking the hidden data. In this chapter, several methods of detection and disabling are discussed with regards to the DNS protocol. This chapter will form the basis for tools applying this methodology that will be described in chapter 4.

3.1 Understanding the DNS protocol

The DNS protocol is an integral service of the internet responsible for translating domain name requests, such as www.wvu.edu, into their corresponding IP addresses (157.182.140.235). Without DNS, users would be forced to remember a series of numbers for each site they wanted to visit. Attempts to remember Google (64.233.167.99), WVU (157.182.140.235), and ZDNet (216.239.115.140) would quickly devolve into random number guessing and make Internet connectivity little more than a passing interest, especially should the IP address of the domain change. In chapter 1, it was mentioned that rogue users could subvert the DNS protocol to use it as a communications channel. Understanding how the protocol works is essential to all other aspects of our attempt to control covert channels within it. The following definitions come from [3] and [9], with definitions from other RFCs included.

3.1.1 Historical Perspective

Beginning in the 1960s with ARPAnet, hostname to IP address name mappings were collected, updated and distributed via FTP from the Network Information Center, a single location on ARPANET. This scheme originally worked well as there were few hosts and few changes to the network addresses. In the years prior to 1983, the number of hosts on the Internet had increased dramatically and threatened an explosion of growth, creating a need for a more efficient and dynamic method of performing domain name translations. The resulting proposals centered on the concept of a distributed hierarchical model roughly corresponding to the organizational structure of the domain names. The basic structure and operation of DNS was implemented in 1983 and was proposed as an RFC in 1987. [3]

DNS is designed to be as flexible as possible to allow a more general-purpose tool that can be used across multiple network types with varying domain name structures. It also is designed to be able to locate data beyond simple IP address mappings, thus requiring greater flexibility. This flexibility and expansive approach allows for a large amount of maneuvering room, enough to permit the subversion of the protocol.

The protocol runs primarily over UDP/IP to minimize connection overhead, but TCP/IP can also be used when data ordering is important or while transferring a large amount of data. Further historical information can be found in [3] and the following sections assume that the reader is somewhat familiar with [3] and [9], as well as network technologies and concepts. Figure 3.1 shows a high-level overview of a DNS packet. Each section of the DNS packet will be discussed in detail in the following subsections.

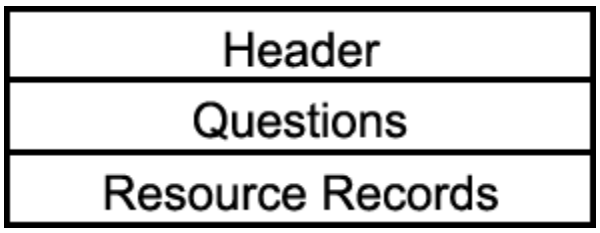

Figure 3-1 DNS Packet Sections Overview

Table 3.1 DNS Packet Sections Description

Field	Description
Header	Contains processing information and protocol-specific options
Questions	Contains the request for an unresolved domain name or Internet Address
Resource Records	Contains response information for the question

3.1.2 Header Definition

Each DNS packet sent and received is composed of a fixed-length header section, an optional question section, and a variable number of variable length resource records. The question section is described in section 3.1.3 and the resource records are described in section 3.1.4.

The header of each DNS packet is a fixed 12 byte length, beginning at the zero offset of the data section of packet payload. The header is show in figure 3.2 and a brief description of each field is listed in table 3.2.

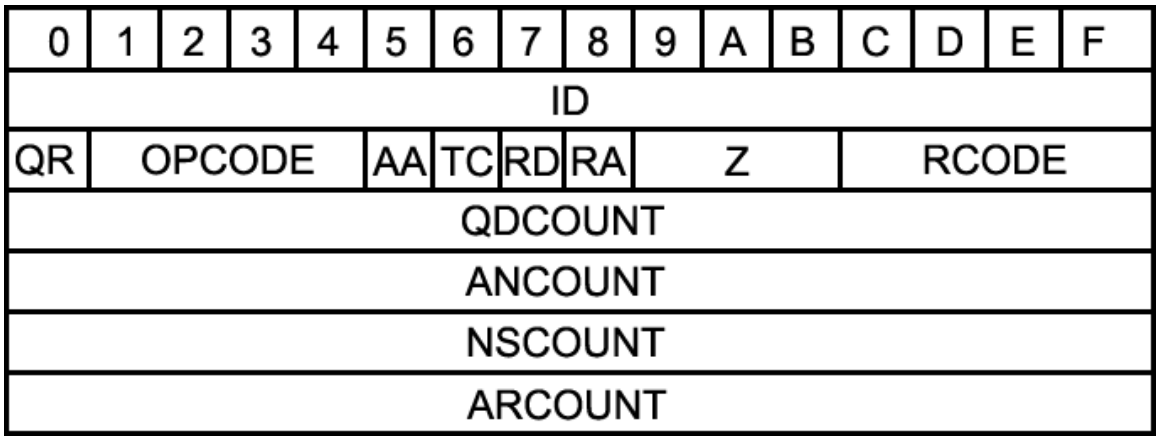

Figure 3-2 Packet Header

Table 3.2 Packet Header Field Definitions

Field	Type	Valid Values	Description
ID	16 bit unsigned int	0 - 2^16-1	ID assigned by program generating DNS query
QR	1 bit	0 - 1	Query (0) or Response (1)
Opcode	4 bit unsigned int	0 - 2	Describes the type of query
AA	1 bit	0 - 1	Authoritative Answer
TC	1 bit	0 - 1	TrunCated message indicator
RD	1 bit	0 - 1	Recursion Desired
RA	1 bit	0 - 1	Recursion Available
Z	3 bits	0	Reserved
Rcode	4 bit unsigned int	0 – 5	Response Code
QDCount	16 bit unsigned int	0 - 2^16-1	Number of questions encoded
ANCount	16 bit unsigned int	0 - 2^16-1	Number of answers encoded
NSCount	16 bit unsigned int	0 - 2^16-1	Number of Nameservers encoded
ARCount	16 bit unsigned int	0 - 2^16-1	Number of Additional Records encoded

The field definitions listed above are fairly straightforward and simple to understand, just as the values for each field based on a condition set are simple to understand. The definition of the header field includes a number of specific conditions that must be met for the values in each field to be valid. These specifics will be discussed further in sections 3.2 and 3.3.

3.1.3 Question Section Definition

Every DNS packet typically has a question included in its contents. Questions form the basis of the client aspect of the client-server relationship DNS functions. The question section of the DNS protocol is a variable length field followed by two fixed-length fields. Figure 3.3 shows the layout of the question section and table 3.3 contain descriptions of the field types and the values permitted for each field.

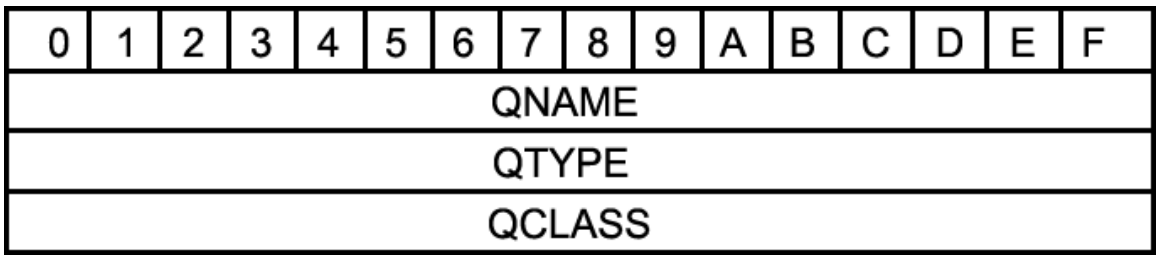

Figure 3-3 Question Format

Table 3.3 Question Field Definitions

Field	Type	Valid Values	Description
Qname	<Domain-name>	See syntax table	Domain Name of the question
Qtype	16 bit unsigned int	1-16, 252-255	Type of query requested
Qclass	16 bit unsigned int	1-4	Query class format

3.1.4 Resource Record Definitions

Each DNS packet optionally contains one or more variable length and structure items known as resource records. Resource records contain varying types of data such as IP addresses, start of authority records, and mailbox information. By far, this portion of the packet is the most important portion of the packet being transmitted as it handles numerous data types. Regardless of the data types being passed, each resource record has a header specifying the record's name, type, class, time to live, and data length. The data portion of the record immediately follows the data length field and is formatted according to the type and class fields specified. Figure 3.4 describes the resource record format while table 3.4 describes the field names and types.

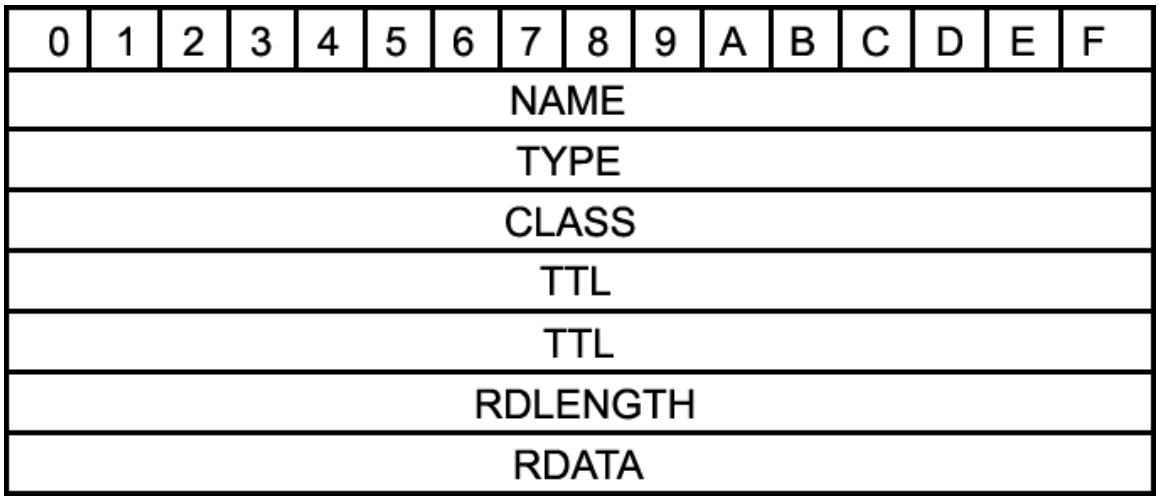

Figure 3-4 Resource Record Format

Table 3.4 Resource Record Field Definitions

Field	Type	Valid Values	Description
Name	<Domain-name>	See syntax table	Node to which this resource record pertains
Type	16 bit unsigned int	1 - 16	Type code determining structure of the Rdata
Class	16 bit unsigned int	1 – 4	Class code for determining format of the Rdata
TTL	32 bit unsigned int	0-(2^32)-1	Time To Live
RDLength	16 bit unsigned int	0-(2^16)-1	Size of the Rdata structure
Rdata	Varies by Type	See Type table	Data of the Resource Record, determined by type code

The type field is responsible for determining the structure of the data portion of the resource record. The class field is primarily used to specify the data format based on the addressing format used: Internet, CSNet, CHAOS, or Hesiod. Tables 3.5 and 3.6 list the valid types and classes per [3] and [9]. The following subsections will explain the structure of the resource record for each type value.

Table 3.5 Resource Record Types

Name	Value	Description
A	1	Answer
NS	2	Authoritative Name Server
MD	3	Mail Destination
MF	4	Mail Forwarder
CNAME	5	Canonical name for alias
SOA	6	Start of Zone Authority
MB	7	Mailbox domain name
MG	8	Mail group domain name
MR	9	Mail rename domain name
NULL	10	Any data valid

WKS	11	Well-Known Service
PTR	12	Domain Name Pointer
HINFO	13	Host Information
MINFO	14	Mailbox/Mail list information
MX	15	Mail Exchange
TXT	16	Text Strings
AFXR	252	Entire Zone Transfer
MAILB	253	Request for Mailbox-related records
MAILA	254	Request for all mail agents
*	255	Request for all records

Table 3.6 Resource Record Classes

Name	Value	Description
IN	1	Internet
CS	2	CSNet
CH	3	CHAOS Net
HS	4	Hesiod

3.1.3.1 – Type A

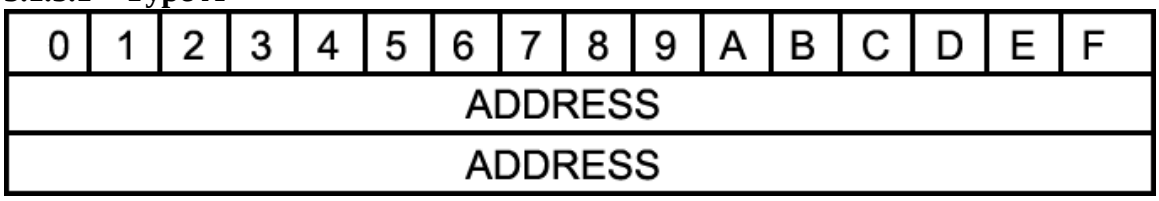

Figure 3-5 Type A Resource Record Format

Table 3.7 Type A Resource Record Field Definitions

Field	Type	Valid Values	Description
ADDRESS	32 bit IP Address	0-(2^32)-1	IP Address

3.1.3.2 – Type NS

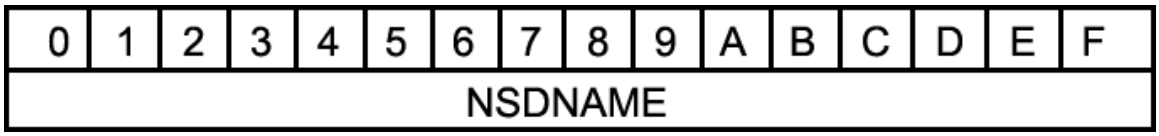

Figure 3-6 Type NS Resource Record Format

Table 3.8 Type NS Resource Record Field Definitions

Field	Type	Valid Values	Description
NSDNAME	<Domain-name>	See syntax table	Authoritative Nameserver

3.1.3.3 – Type MD

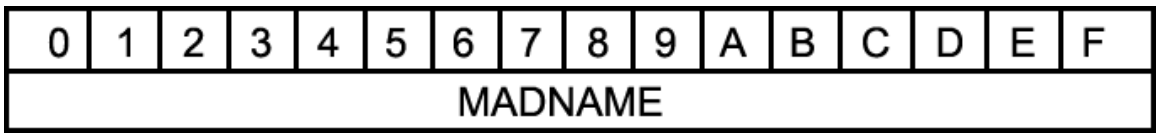

Figure 3-7 Type MD Resource Record Format

Table 3.9 Type MD Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
MADNAME	<Domain-name>	See syntax table	Mail Destination	Obsolete

3.1.3.4 – Type MF

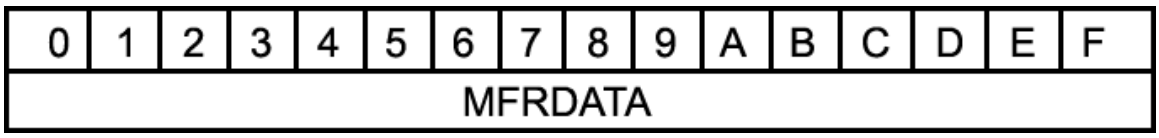

Figure 3-8 Type MF Resource Record Format

Table 3.10 Type MF Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
MFRDATA	<Domain-name>	See syntax table	Mail Forwarder	Obsolete

3.1.3.5 – Type CName

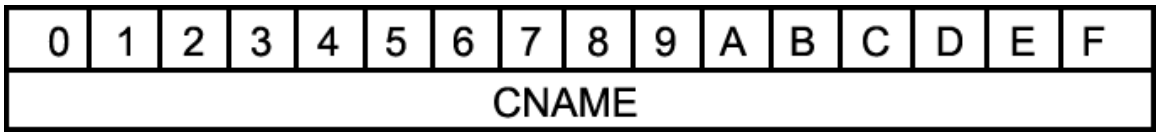

Figure 3-9 Type CNAME Resource Record Format

Table 3.11 Type CNAME Resource Record Field Definitions

Field	Type	Valid Values	Description
CNAME	<Domain-name>	See syntax table	Canonical name for an alias

3.1.3.6 – Type SOA

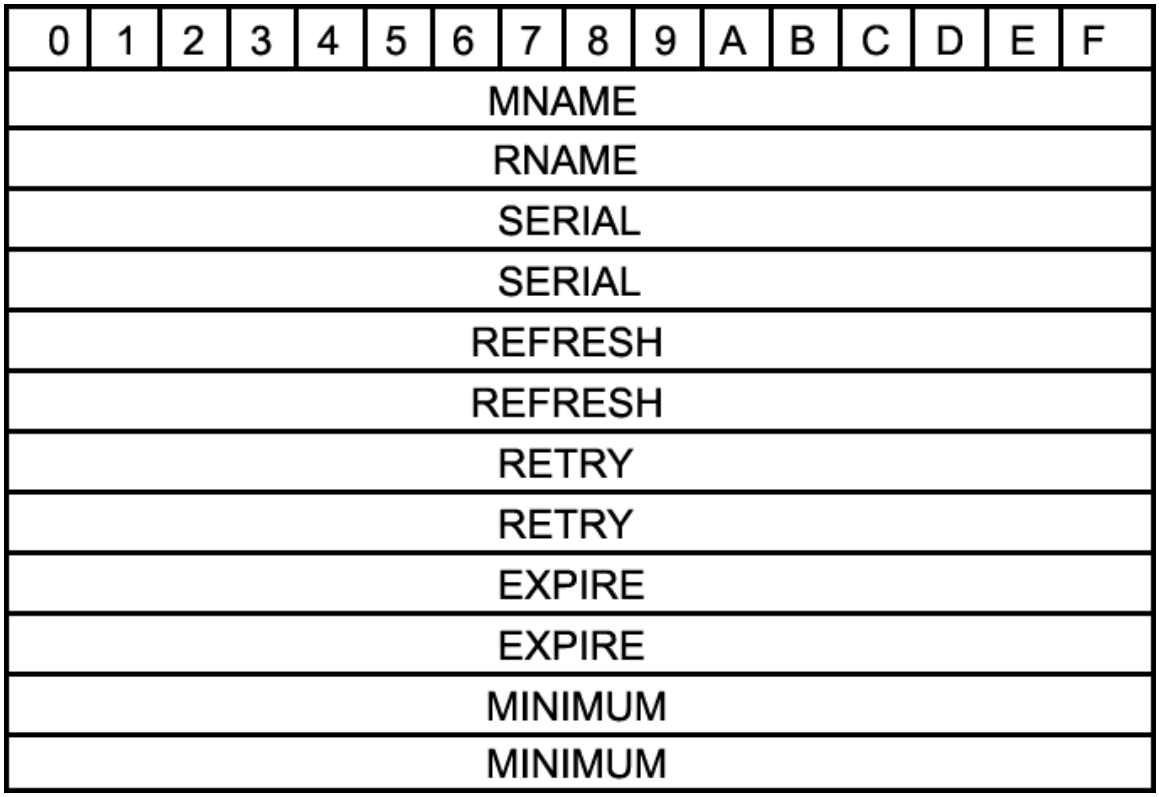

Figure 3-10 Type SOA Resource Record Format

Table 3.12 Type SOA Resource Record Field Definitions

Field	Type	Valid Values	Description
MNAME	<Domain-name>	See syntax table	Primary nameserver for zone data
RNAME	<Domain-name>	See syntax table	Zone administrator's Mailbox
SERIAL	32 bit unsigned int	0 - (2^32)-1	Original version number of zone
REFRESH	32 bit unsigned int	0 - (2^32)-1	Time interval before zone refresh
RETRY	32 bit unsigned int	0 - (2^32)-1	Time interval before zone refresh on failed
EXPIRE	32 bit unsigned int	0 - (2^32)-1	Time limit before zone no longer authoritative
MINIMUM	32 bit unsigned int	0 - (2^32)-1	Minimum TTL field exported from this zone

3.1.3.7 – Type MB

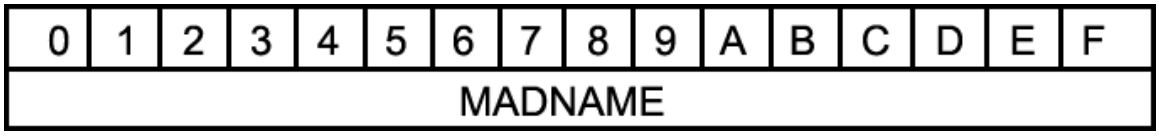

Figure 3-11 Type MB Resource Record Format

Table 3.13 Type MB Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
MADNAME	<Domain-name>	See syntax table	Host containing the specified mailbox	Experimental

3.1.3.8 – Type MG

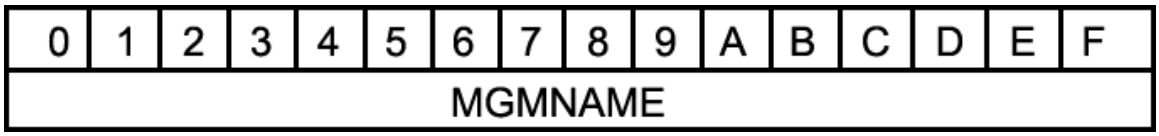

Figure 3-12 Type MG Resource Record Format

Table 3.14 Type MG Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
MGMNAME	<Domain-name>	See syntax table	Specifies member mailbox of mail group specified by domain	Experimental

3.1.3.9 – Type MR

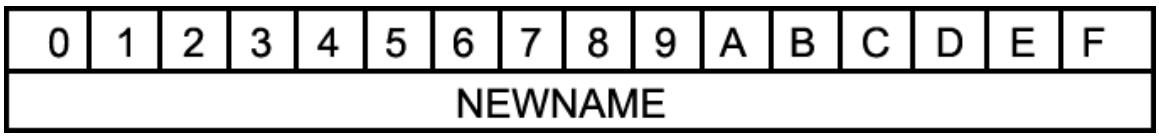

Figure 3-13 Type MR Resource Record Format

Table 3.15 Type MR Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
NEWNAME	<Domain-name>	See syntax table	Specifies mailbox which is the proper rename of specified mailbox	Experimental

3.1.3.10 – Type NULL

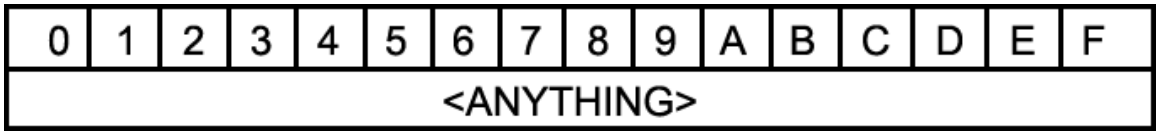

Figure 3-14 Type NULL Resource Record Format

Table 3.16 Type NULL Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
<ANYTHING>	<8 bit unsigned int>	Any value	Field permitting any kinds of values to be input. Used for testing.	Experimental

3.1.3.11 – Type WKS

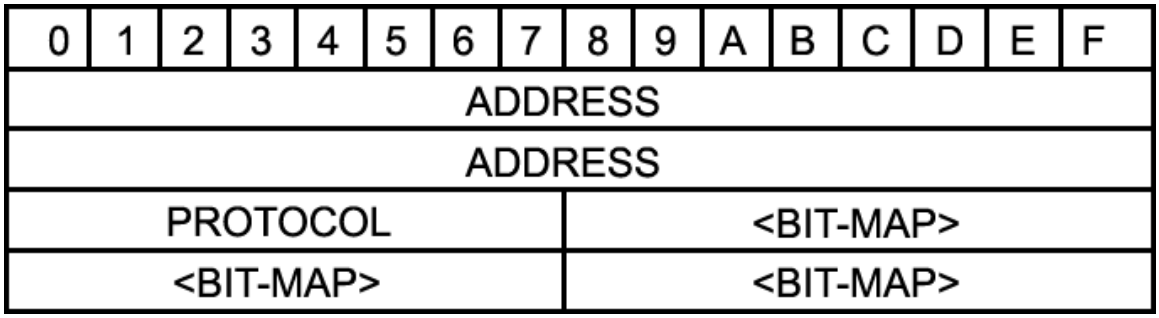

Figure 3-15 Type WKS Resource Record Format

Table 3.17 Type WKS Resource Record Field Definitions

Field	Type	Valid Values	Description
ADDRESS	32 bit IP Address	0 - (2^32)-1	IP Address
Protocol	8 bit unsigned int	6 (TCP), 17(UDP)	Protocol number as determined by IANA for IP
<BIT-MAP>	8 bit unsigned int	0 - 255	Bitmap showing available service ports on the specified IP Address

3.1.3.12 – Type PTR

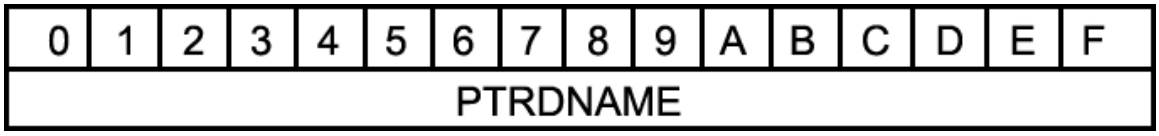

Figure 3-16 Type PTR Resource Record Format

Table 3.18 Type PTR Resource Record Field Definitions

Field	Type	Valid Values	Description
PTRDNAME	<Domain-name>	See syntax table	Pointer to some location in the domain name space

3.1.3.13 – Type HINFO

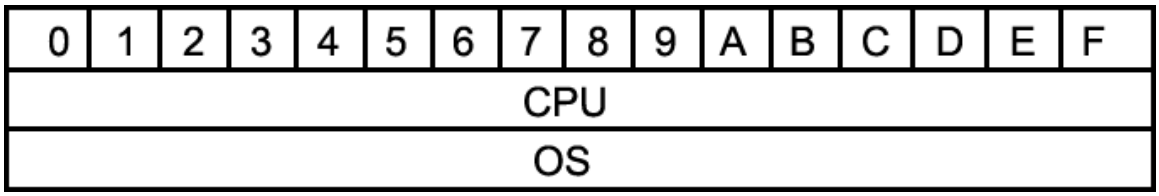

Figure 3-17 Type HINFO Resource Record Format

Table 3.19 Type HINFO Resource Record Field Definitions

Field	Type	Valid Values	Description	Notes
CPU	<character-string>	See syntax table	Specifies host CPU type	40 char max
OS	<character-string>	See syntax table	Specifies host OS	40 char max

3.1.3.14 – Type MINFO

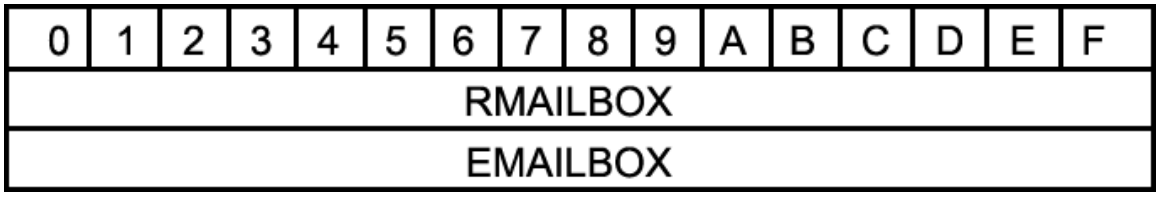

Figure 3-18 Type MINFO Resource Record Format

Table 3.20 Type MINFO Resource Record Field Definitions

Field	Type	Valid Values	Description
RMAILBOX	<Domain-name>	See syntax table	Specifies mailbox responsible for mailing list or mailbox
EMAILBOX	<Domain-name>	See syntax table	Specifies mailbox to receive error messages

3.1.3.15 – Type MX

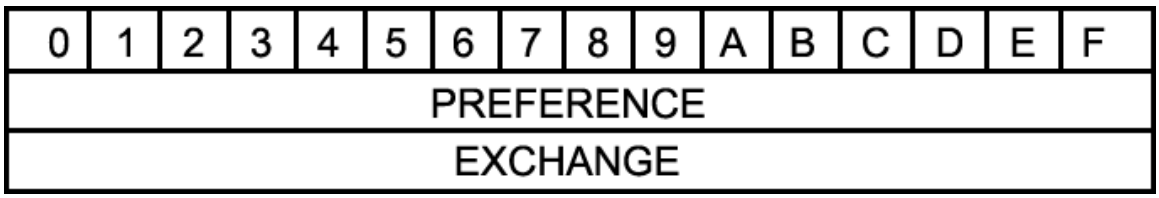

Figure 3-19 Type MX Resource Record Format

Table 3.21 Type MX Resource Record Field Definitions

Field	Type	Valid Values	Description
PREFERENCE	16 bit unsigned int	0 - (2^16)-1	Specifies mailbox responsible for mailing list or mailbox
EXCHANGE	<Domain-name>	See syntax table	Specifies mailbox to receive error messages

3.1.3.16 – Type TXT

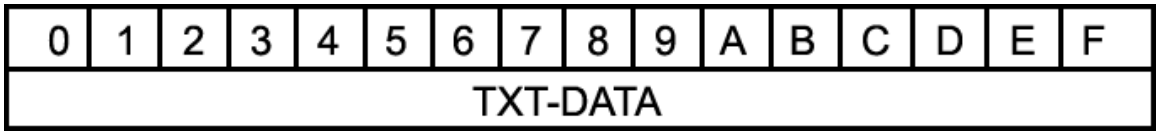

Figure 3-20 Type TXT Resource Record Format

Table 3.22 Type TXT Resource Record Field Definitions

Field	Type	Valid Values	Description
TXT-DATA	<character-string>	See syntax table	Holds descriptive text

Table 3.6 listed the legal class values permitted and defined in [3] and [9]. The most prevalent class value in use is class 1, Internet, but the protocol's designers wanted DNS to be a flexible service, so the values for CSNet, CHAOS, and Hesiod were included to allow for cross-addressing protocol communications. For the purpose of this research, non-Internet protocols were included, but specifics of the varying class types will not be discussed in the paper.

3.1.5 Syntax and Compression

There are two further pieces of information that must be understood to fully understand the DNS protocol: syntax and message compression.

3.1.5.1 Message Syntax

Throughout the definitions of the resource records, references were made to a data type known as <Domain-name>. Table 3.23 breaks this data type into its atomic pieces and defines the valid values.

Table 3.23 Message Syntax Definition

Field	Definition
<Domain-name>	<subdomain> | " "
<subdomain>	<label> | <subdomain> "." <label>
<label>	<let-dig>[[<ldh-str>]<let-dig>]
<ldh-str>	<let-dig-hyp> | <let-dig-hyp> <ldh-str>
<let-dig-hyp>	<let-dig> | "-"
<let-dig>	<letter> | <digit>
<letter>	A - Z, a – z
<digit>	0 – 9

Each label begins with an octet defining the length of the following label, and a series of labels is always terminated by a zero or null octet. In addition to the syntax listed above, there are several length requirements that must be met. The first is each label's length is limited to 63 bytes at a maximum. The reason for this limit will be discussed under message compression. The final requirement for a series of labels is that the maximum length of all of the labels, including the label length octets, is 255 octets.

IP addresses can also be encoded as part of a DNS request, but their format seems opposite that of a standard domain name. A request for 64.233.167.99 would be encoded as 99.167.233.64.IN-ADDR.ARPA.

The resource record definitions also made reference to a data type listed as <character-string>. Unless otherwise specified, this data type is characterized by a single label length octet followed by that number of characters, making the maximum permitted length of the character data and the length octet 256 characters. The information in the data type is treated as binary information. <Character-string> can be expressed in two ways: as a set of contiguous characters lacking spaces, or as a string encapsulated within a pair of quotation marks (“). Strings encapsulated within quotation marks containing quotation marks themselves must be quoted using the backslash.

3.1.5.2 Message Compression

In order to minimize the size of a message by removing redundant information, inline message pointers were implemented in the body of the DNS packet for the Name data type of the resource record, allowing data from one record to be utilized in another record. This concept is illustrated in figure 3.21 below. The top line is the offset value from the packet start, boldface characters are ASCII values, and regular characters are integer values.

40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
3	n	s	1	14	n	a	m	e	s	e	r	v	e	r	s	i	t	e	3	c	o	m	0
<Remainder of RR info>																							
3	n	s	2	192	44	<Remainder of RR info>																	
3	n	s	3	192	44	<Remainder of RR info>																	
3	n	s	4	192	44	<Remainder of RR info>																	
3	n	s	5	192	44	<Remainder of RR info>																	
3	n	s	6	192	44	<Remainder of RR info>																	
3	n	s	7	192	44	<Remainder of RR info>																	
3	n	s	8	192	44	<Remainder of RR info>																	

Figure 3-21 Message Compression Example

As we can see, ns1.nameserversite.com becomes the only fully domain name used in the entire packet, even though there are eight nameservers listed. Records ns2 through ns8 point to the offset in the packet where .nameserver.com originate, saving a significant amount of data per resource record. Message Compression is indicated by setting the first two high-order bits of a label length octet to 1. The remaining 6 bits of that octet and the following octet form the offset from the start of the packet where the next portion of the domain name can be found. It is possible for a single domain name to have multiple pointers, such as the described in figure 3-22.

29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
3	w	w	w	3	w	v	u	3	e	d	u	0	<Remainder of RR info>							
3	w	w	w	4	c	e	m	r	192	33	<Remainder of RR info>									
3	w	w	w	5	l	c	s	e	e	192	33	<Remainder of RR info>								
192	33	<Remainder of RR info>																		

Figure 3-22 Multiple Inline Pointers

Thus, we see that the resource record for www.lcsee.cemr.wvu.edu has two pointers for message compression. Pointers can also be used to start a domain name, as shown by the last entry in figure 3.22. Here we have a record that is part of the www.wvu.edu record, and to compress the record data, the record begins with a pointer to the original www.wvu.edu record. These are slightly simplistic examples, but we can see how message compression can be utilized to minimize the size of a DNS packet.

3.1.6 Summary

This subsection was intended to be an informational overview describing the DNS protocol at a slightly higher-level while maintaining enough technical content to allow a proper analysis of the protocol definition in section 3.2. In this subsection, all valid syntax and values were defined for each of the fields and a number of condition sets were created, all of which will be used in aiding our analysis and response.

3.2 Identification of Fields

In order for a protocol to be subverted and remain inconspicuous and valid to anyone weakly monitoring the data, a certain amount of weakness must be present in the protocol's definition. In this section, the protocol definitions are broken into several categories based on their requirements as part of the packet and whether the legal values are completely set. The four resulting categories are as follows:

- Required strong fields – Fields that must be present and must have a set value to be valid
- Required weak fields – Fields that must be present but do not require a specific value to function
- Optional strong fields – Fields that may be present and must have a set value to be valid
- Optional weak fields – Fields that may be present but do not require a specific value to function

The reason we are breaking these down is these fields all take up a certain portion of the transmission channel bandwidth and therefore affect the amount of data that can be transmitted in each packet. Table 3.24 below shows a breakdown of the four fields as each relates to the packet bandwidth.

Table 3.24 Packet Type Breakdown

Type	Values	Bits per Packet
Required	Strong	RS
	Weak	RW
Optional	Strong	OS
	Weak	OW
		RS+RW+OS+OW=Packet size

However, as this relates to covert channels, the weak fields represent the maximum bits per packet that can supposedly be transmitted without affecting the operation of the protocol. In effect, this value is lower as there are still some requirements placed on the syntax and format of the data.

3.2.1 Required fields

Research and understanding of the DNS protocol leads to the conclusion that the only required portion of the protocol that was to be present in every packet is the header section. From here, review of the fields within the header revealed that only one field with a required value: the Z field where all bits within this field must be zero. The rest of the fields were set according to direction and packet information and were therefore classified as required weak fields. Table 3.25 contains a full list of the findings.

Table 3.25 Required Strong and Weak Fields

Strong Fields	Value	Weak Fields	
Z	0	ID	OPCODE
		QR	RCODE
		AA	QDCOUNT
		TC	ANCOUNT
		RD	NSCOUNT
		RA	ARCOUNT

3.2.2 Optional fields

The question section and resource records were determined to be optional sections as they were not required in every packet like the header. The fields of the question and record portion of the protocol were found to be composed entirely of weak fields and were classified as such. Resource record structures based on class type were also examined for weak and strong fields. Table 3.26 contains a comprehensive list of the

varying type value structures and their definitions. Review of these fields show that no field has any strong fields, so all of the type value definitions were classified as optional weak fields since their parent, the resource record, is optional.

Table 3.26 Optional Weak Fields

Weak Fields				
QNAME	TTL	CNAME	EXPIRE	<BIT-MAP>
QTYPE	RDLENGTH	MNAME	MINIMUM	PTRDNAME
QCLASS	RDATA	RNAME	MGMNAME	CPU
NAME	ADDRESS	SERIAL	NEWNAME	OS
TYPE	NSDNAME	REFRESH	<ANYTHING>	RMAILBOX
CLASS	MADNAME	RETRY	PROTOCOL	EMAILBOX
PREFERENCE	EXCHANGE	TXT-DATA		

3.3 Network Topology Awareness

While format requirements and valid data ranges can help minimize the subversion of the protocol, the direction and types of traffic transmitted across the network can impact the usage of covert channels. DNS is a client-server protocol where a request is made to a server and the server responds with an answer. It is common for the DNS server for one client to itself become a client to another DNS server in order to resolve a particular request. The hierarchical nature of DNS paints a portrait of a network designed as shown in figure 3.23.

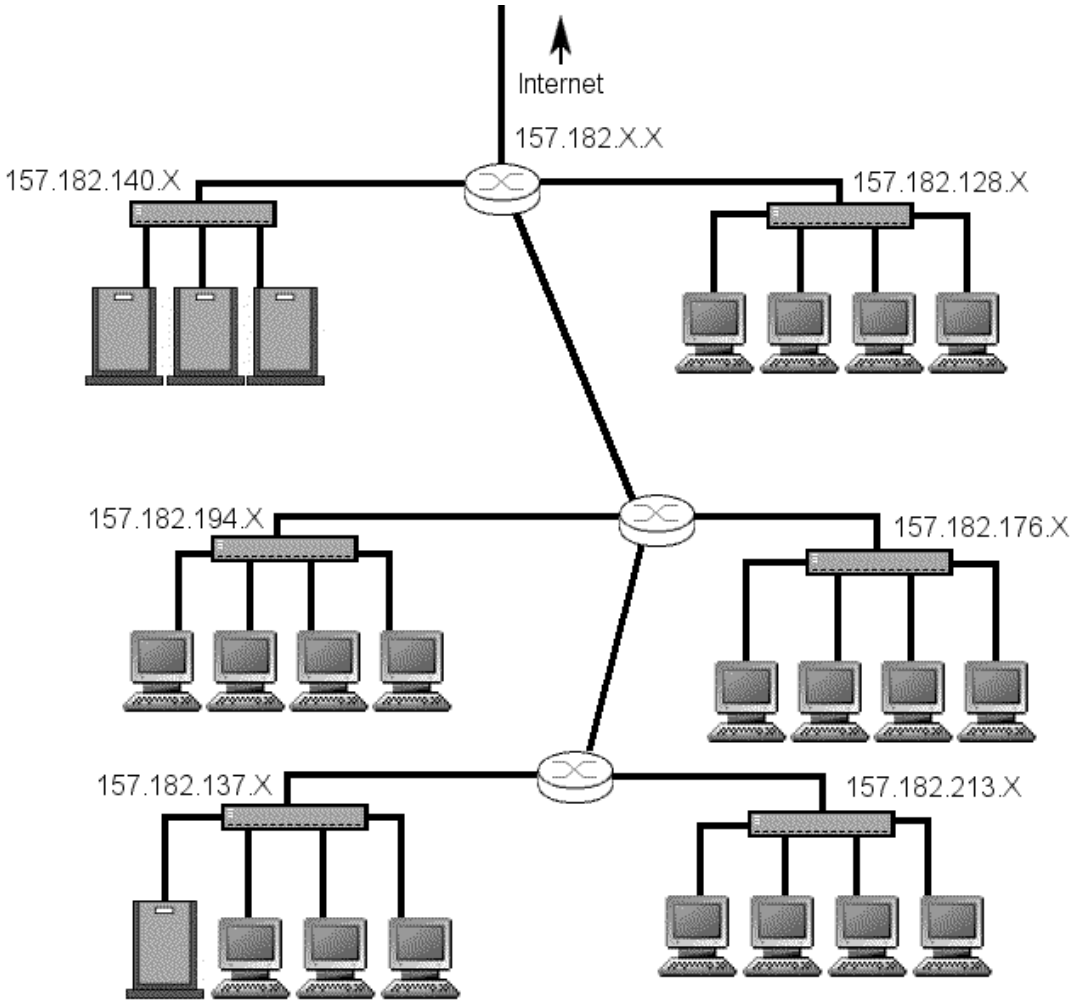

Figure 3-23 Example Network Layout

It is from the network's topology that we further pare down the permissible options and fields to a subset of valid options based on the network structure. For example, DNS traffic coming from the 157.182.213.X network does not have a DNS server in that particular collision domain, so we can effectively rule out specific values and options in a number of weak fields due to the expected types of traffic coming from a particular network segment. Due to the client-server nature of DNS, a second categorization of the fields into client and server functionality occurred.

It was noted that since servers could also act as clients, the categories would be divided into server-only, client-server, and client-only, where the client-only fields were for machines that were strictly end-systems, while servers potentially utilized the whole set of DNS fields and functionality. Server-only categories of functionality are intended for DNS servers on the absolute root of the domain hierarchy, such as .com, .org, and .net servers. However, due to the nature of the protocol, this server-only division is not guaranteed. It has been included as a category as theoretically it is possible to have a system that behaves as a pure server.

By being aware of the network's topology and of the expected services permissible in a segment, we can further reduce the capabilities of covert channels in DNS. Tables 3.27 and 3.28 contain a list of client-only fields and valid values for both incoming and outgoing traffic. Tables 3.29 and 3.30 contain a listing of server-only fields and the valid values for incoming and outgoing traffic. Certain inbound and outbound types have been excluded as the resource record definitions for those types are not permitted.

Table 3.27 Client-Only Expected Field Values

Field Name	Outbound	Inbound
ID	Any	Matches previous outbound
QR	0	1
OPCODE	0 - 2	0-2
AA	0	0,1
TC	0	Usually 0
RD	0,1	0
RA	0	0,1
Z	000	000
RCODE	0	0-5
QDCOUNT	1	1
ANCOUNT	0	0 - (2^16)-1
NSCOUNT	0	0 - (2^16)-1
ARCOUNT	0	0 - (2^16)-1
QNAME	<Domain-name>	Matches previous outbound
QTYPE	1-16, 252-255	Matches previous outbound
QCLASS	1-4	Matches previous outbound
NAME	-	<Domain-name>
TYPE	-	1 - 16
CLASS	-	1 - 4
TTL	-	0 - (2^32)-1
RDLENGTH	-	0 - (2^16)-1
RDATA	-	See Client RR table

Table 3.28 Client-Only Expected RR Values

RR Type	Field	Inbound
A	ADDRESS	32 bit IP address - Some exceptions
NS	NSDNAME	<Domain-name>
MD	MADNAME	<Domain-name>
MF	MADNAME	<Domain-name>
CN	CNAME	<Domain-name>
SOA	MNAME	<Domain-name>
	RNAME	<Domain-name>
	SERIAL	0 - (2^32)-1
	REFRESTH	0 - (2^32)-1
	RETRY	0 - (2^32)-1
	EXPIRE	0 - (2^32)-1
	MINIMUM	0 - (2^32)-1
MB	MADNAME	<Domain-name>
MG	MGMNAME	<Domain-name>
MR	NEWNAME	<Domain-name>
NULL	<ANYTHING>	Anything
WKS	ADDRESS	32 bit IP address - Some exceptions
	PROTOCOL	6 (TCP), 17 (UDP)
	<BIT-MAP>	0 - 255
PTR	PTRDNAME	<Domain-name>
HINFO	CPU	<character-string>
	OS	<character-string>
MINFO	RMAILBOX	<Domain-name>
	EMAILBOX	<Domain-name>
MX	PREFERENCE	0 - (2^16)-1
	EXCHANGE	<Domain-name>
TXT	TXT-DATA	<character-string>[<character-string>]

Table 3.29 Server-Only Expected Field Values

Field Name	Outbound	Inbound
ID	Matches previous inbound	Any
QR	1	0
OPCODE	0-2	0 - 2
AA	0,1	0
TC	Usually 0	0
RD	0	0,1
RA	0,1	0
Z	000	000
RCODE	0-5	0
QDCOUNT	1	1
ANCOUNT	0 - (2^16)-1	0
NSCOUNT	0 - (2^16)-1	0
ARCOUNT	0 - (2^16)-1	0
QNAME	Matches previous inbound	<Domain-name>
QTYPE	Matches previous inbound	1-16, 252-255
QCLASS	Matches previous inbound	1-4
NAME	<Domain-name>	-
TYPE	1 - 16	-
CLASS	1 - 4	-
TTL	0 - (2^32)-1	-
RDLENGTH	0 - (2^16)-1	-
RDATA	See Server RR table	-

Table 3.30 Server-Only Expected RR Values

RR Type	Field	Outbound
A	ADDRESS	32 bit IP address - Some exceptions
NS	NSDNAME	<Domain-name>
MD	MADNAME	<Domain-name>
MF	MADNAME	<Domain-name>
CN	CNAME	<Domain-name>
SOA	MNAME	<Domain-name>
	RNAME	<Domain-name>
	SERIAL	0 - (2^32)-1
	REFRESTH	0 - (2^32)-1
	RETRY	0 - (2^32)-1
	EXPIRE	0 - (2^32)-1
	MINIMUM	0 - (2^32)-1
MB	MADNAME	<Domain-name>
MG	MGMNAME	<Domain-name>
MR	NEWNAME	<Domain-name>
NULL	<ANYTHING>	Anything
WKS	ADDRESS	32 bit IP address - Some exceptions
	PROTOCOL	6 (TCP), 17 (UDP)
	<BIT-MAP>	0 - 255
PTR	PTRDNAME	<Domain-name>
HINFO	CPU	<character-string>
	OS	<character-string>
MINFO	RMAILBOX	<Domain-name>
	EMAILBOX	<Domain-name>
MX	PREFERENCE	0 - (2^16)-1
	EXCHANGE	<Domain-name>
TXT	TXT-DATA	<character-string>[<character-string>]

3.4 Identifying and Limiting Covert Channels

In sections 3.2 and 3.3, we identified a number of protocol- and network-specific constraints that were to be followed in order to maintain compliance with the protocol's standards. With these constraints in mind, a series of permitted operations based on the protocol and network layout from our lists can be derived. We can generate rule listings based on tables 3.27, 3.28, 3.29, and 3.30 that were in turn derived from the protocol and network definitions.

From tables 3.27 and 3.28, we can see that the rule set for the client side is fairly simple, depending on the functionality we want to permit on the network segment. However, the question of how we detect covert channels remains. The most obvious way to detect them is to see which, if any, of the rules a packet fails and then paying special attention to that packet. This is the most obvious way to detect covert channels, but does not mean that the other packets do not have covert transmissions hidden within.

Since we are not able to fully detect all types of covert communications, the option we have is to manage all fields not requiring an explicit value to function, such as the ID field. Limiting the potential for covert communications can be achieved by placing a layer of abstraction between the user's computer and the outgoing network interface from a segment. In this model, when a packet is received, all fields except the data payload field are rewritten or regenerated, and then the query is passed on to the DNS server. The ID field, which may previously have had a value of 1042, may now have a value of 249 courtesy of a random number generator. The response from the server is processed in a similar manner, removing any invalid response options, and the original field data with a reply is returned to the user. Regular users see nothing amiss while rogue users may be forced to utilize another method to transmit their data.

3.5 Response Options

Once identification of a covert channel has occurred, the question of how to respond to the channel remains. From the steganography model, there are three options we can glean from the portrayal of Alice, Bob, and the warden: Permit, Drop, and Reprocess.

The option to respond by permitting all traffic is a response that trusts the packets passing through without any question. Packets failing the filtering rule are permitted to continue on without any modifications or other response. Any type of network implementing this choice long-term does not require any type of validation service unless looking for unexpected services or monitoring traffic.

Choosing to drop any packets failing the filtering rules is the second of the three options derived from the warden's portrayal. With this choice, packets failing any of the filters are simply discarded as though they were dropped somewhere along the line by the network. This implementation choice can cause a buildup of DNS traffic on the segment as the protocol's definition suggests a number of times to retry a request following the lack of a response within an allotted time period. Small network segments without a large amount of DNS traffic can utilize this measure with little impact on network responsiveness.

Reprocessing a message is the equivalent of the active warden model, where the warden takes a message from Alice, types it up on the typewriter and performs some other analysis on it, and then gives it to Bob. The application of this model to the problem of DNS covert channels involves rewriting as much of the packet as possible without losing the meaning of the message being passed. In this application, reprocessing would entail overwriting every field failing the filtering rules and other fields whose modification does not impact the packet integrity.

The response options given by steganalysis are simple concepts that can be used to disrupt and combat steganography attempts in general. Covert channels are a specific field within steganography, but are still susceptible to these response techniques, albeit

varying from those of images or video. The difficulty with covert channels lies in the fact that the communications are not typically stored to disk like images and video. With covert communications, a decision must be rendered nearly instantaneously on a limited amount of data.

3.6 Additional Considerations

Minimizing and eliminating the capabilities of covert channels is a difficult process as there are many ways to hide data within legitimate communications. With this in mind, there are several additional considerations that must be taken by one attempting to limit covert channels.

3.6.1 Standard Revisions

The internet and its communication protocols are constantly evolving, adding new features, making others obsolete, and competing with other protocols for implementation. This document and its content used [3] and [9] to generate the rule listings and tables found within. There have been a number of proposed and accepted updates to these documents that add additional features like DNSSEC. Other revisions and clarifications are sure to follow in the future as demand for additional functionality from the protocol increases. Each new revision will require analysis, verification, and some means of permitting backwards compatibility to older protocol definitions.

3.6.2 Covert Channel Logging

A facet of communications protocols that is often overlooked is the packaging that holds the specific data desired. The packaging explains a number of things about the data encoded within, but is discarded once the packet reaches its endpoint and is decoded.

The choice to discard all of the communications except the specific data is potentially limiting as we cannot later review all the data in the packet should a new method of analysis become available. Consciously copying all of the original data on its way from the client and from the outside server allows for later review of the data and its packaging for covert transmissions. This storage of the whole packet allows for a thorough, long-term analysis of packet data for covert messages.

3.6.3 Network Issues

Two issues arise from the network structure that must be addressed to ensure validation of all DNS traffic: deployment location and packet routing.

The goal is to analyze all DNS packets coming and going through a particular network segment, so the most logical placement of a device to handle this task would be on the gateway. Here all traffic entering and leaving the segment would be forced to go through the single location leading to the next level of the network. This location presents a somewhat unexpected problem in that packets leaving the segment are monitored, but packets remaining in the segment that never attempt leave the segment fail to go through the gateway and therefore are not processed.

The second routing problem theoretically results from deviation from the expected routing structure of the DNS hierarchy. DNS queries are supposed go through a specific hierarchy of name servers to obtain a response. Typically, addresses for name servers are determined by DHCP or are set by a network administrator. In a typical model, queries would go to these servers first and then head out of the network. A rogue user specifying a DNS request to a name server outside of the network can avoid the hierarchy and theoretically validation as well.

With these problems in mind, we would want to ensure all traffic leaving or staying in the segment is validated. This suggests a means of forwarding all DNS packets received on a segment to first go through the validation mechanism prior to being sent to their intended destination.

Chapter 4
4 NV-DNS

NV-DNS is a Domain Name Service packet processor that checks and validates DNS packets for correctness, validity, and other implementation errors. The intent of this program is to evaluate DNS communications for hidden or covert channels within the contents of the packet. In addition, NV-DNS will aid security professionals and software designers in providing a more secure networked computing environment. Inspiration from this program came from the concept of real-time network analysis, where packets are examined and recorded for usage statistics.

NV-DNS processes incoming UDP-based DNS packets based on the specification of the Domain Name Service in RFC 1034 and RFC 1035. [3], [9] it utilizes a configuration file to determine the permissible DNS options and classes on a particular network segment. Based on this configuration, the program also executes a set of desired actions to perform on packets failing configuration rules. Ideally NV-DNS is a monitoring tool set on network gateways to determine the validity of the communications passing through. It can also be used to push software designers towards network-validating software to minimize the possibility of rogue programs subverting protocols for unauthorized use.

4.1 Architectural Overview

NV-DNS is composed of three main components:

- The Packet Parser
- The Validation Module
- The Packet Crafter

The Packet Parser is responsible for parsing DNS packets into its various fields to allow for simpler validation. The Validation Module processes the packet based on a rule filter to check for invalid values and fields, determining if the packet is a candidate for any type of covert communications. The Packet Crafter repackages the packet based on the rule filter and overwrites some fields based on the necessity of the field data, like the ID field.

The design of NV-DNS is intended to allow for a high degree of flexibility and maintainability. With these aspects of development in mind, the components of this tool were implemented in a modular fashion that permits simple upkeep of the current modules or replacement of the modules with their own without affecting program functionality. Figure 4.1 shows a simple diagram describing the flow of NV-DNS and will be used as the foundation for describing the tool's functionality.

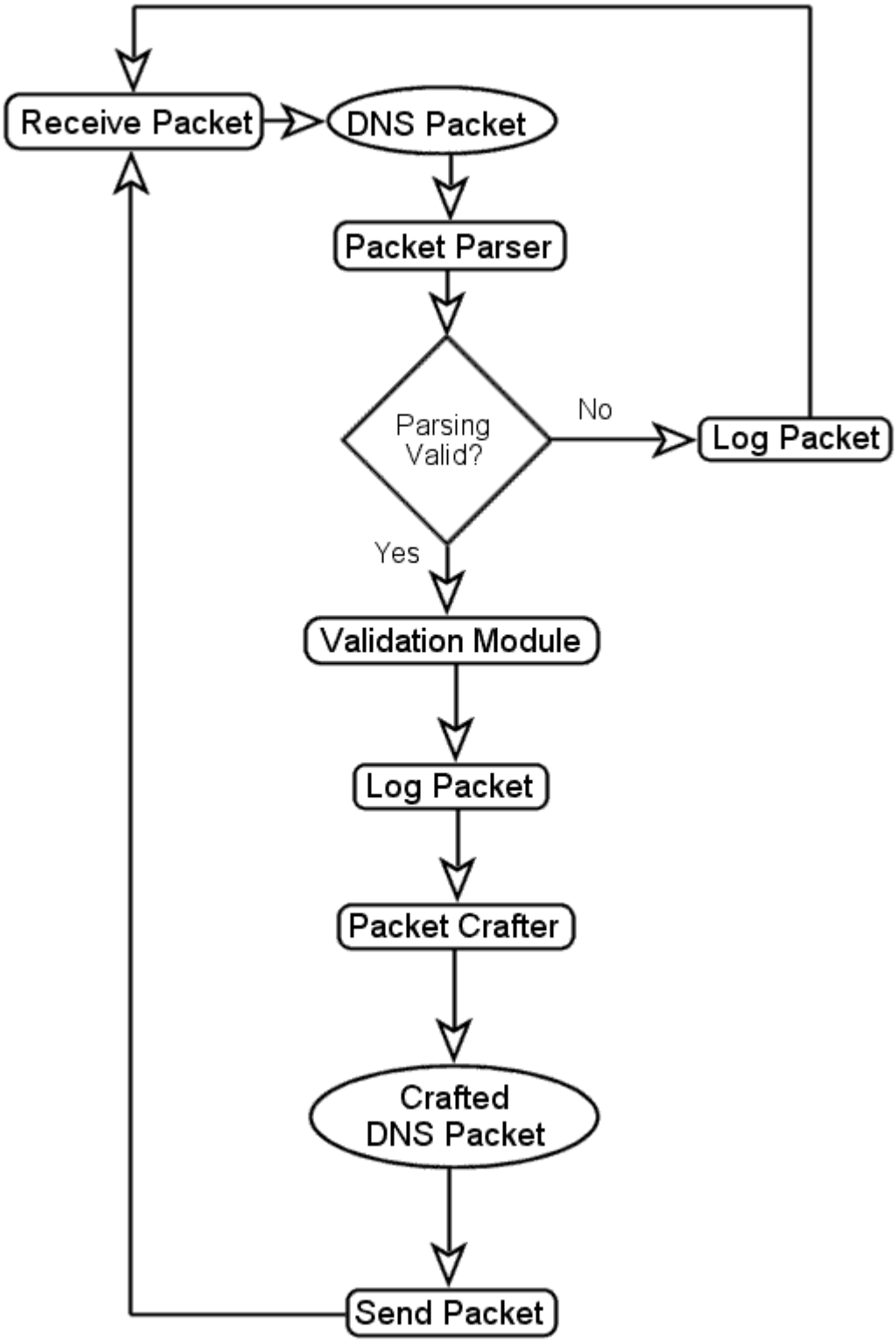

Figure 4-1 NV-DNS Flowchart

Connections for incoming DNS requests are received and the packet is fed into the Packet Parser, which simply breaks the packet into specific fields to allow for easier validation. Once the packet has been parsed, the Validation Module tests the packet structure based on the packet's direction, network topology of the segment, and adherence to protocol specifications. The validation rules are loaded from a configuration file previously generated by a network administrator. The resulting validation response is fed into the Packet Crafter and a new packet is crafted with valid data and sent off to the next-level DNS server. The program awaits a response and processes the response in the same way that the request was processed. The original request and response are also stored to disk with packet header information.

4.1.1 The Packet Parser

The Packet Parser is responsible for breaking the DNS packet into the varying fields to assist in analysis of the packet data. This includes pulling the specific bit fields out of the header and converting bits into numbers. The methodology chosen to create a structure suitable for validation is not intended to be the fastest or most efficient means of validation, but the simplest. Memory is allocated for a structure that contains the DNS fields and their values, and the data is parsed into the allocated fields. During this parsing, data is converted, packed, or unpacked into its respective fields before being returned to the calling routine. The newly allocated structure is then passed on to the validation module that performs the tedious task of reviewing the values in each of the fields.

4.1.2 The Validation Module

The Validation Module is the second major component of NV-DNS and is easily the most complex module of the system. It is also by far the most important component of NV-DNS. This particular module must validate all portions of the DNS protocol specification and must also account for network location rules. The concept of having some type of off-system validation mechanism was influenced heavily by the RFCs trusting implementation and utilization in "good faith" to the programmers. Secondary influences for off-system validation are found littered throughout research papers and articles on steganalysis and covert channels.

Validation of all aspects of a protocol based on its definition is a potentially difficult task. Every valid combination of the protocol definition must be processed as valid while any invalid combinations or definitions must be appropriately labeled as such. Implementing code to properly analyze and validate the protocol is the most difficult portion of the NV-DNS program. The validation of the protocol based on traffic direction is a much simpler validation to make in terms of complexity as there are only two directions to traffic. Most of the functions can be grouped into a few basic rule sets on either direction, thus minimizing the complexity of this type of validation. Once validation is complete and a response code has been generated by the module, the response code and packet are passed on to the final module.

4.1.3 The Packet Crafter

The last module being discussed is the Packet Crafter, which can be an optional module in the system. The purpose of the module is evident from its name, but for some reasons which may not be fully obvious at first glance. As the name implies, this

particular module crafts a new DNS request or response from the input it receives from the Validation Module and originally the Packet Parser. The main reason for crafting a new packet is to minimize the chance an undetected covert channel could survive intact. By removing and potentially restructuring the packet, we can further attack any potential covert channels by scrambling the data ordering of a query or reply.

4.2 Further Considerations

In this section, the components of NV-DNS were outlined and a reasonable scope of the prototype's functionality was described. There are a number of other challenges associated with this tool that are beyond the scope of this research. However, it is anticipated that the work done here will provide a foundation for future covert channel work.

In the next chapter, we discuss the implementation of NV-DNS and several case studies conducted to determine how reasonable the prototype is.

Chapter 5
5 Applications of NV-DNS

This chapter will discuss a rudimentary implementation of Network-Validated Domain Name Service (NV-DNS) designed with the purpose of applying steganalysis techniques of covert channels and real-time network analysis to the problem of identifying and limiting covert channels in the DNS protocol and being able to respond to these known and potentially unknown channels in real-time.

In the previous chapter, the architecture of NV-DNS was discussed in detail. The version of NV-DNS presented here is in no way intended to be complete and is only capable of handling a small amount of DNS traffic at a time. Further, only simple versions of the Packet Parser, Validation Module, and Packet Crafter are implemented.

5.1 Implementation Language

C was chosen as the implementation language due to the high level of control capabilities afforded by the language and a mostly standard implementation across platforms. The complete listing for the source code along with a user's guide is presented in Appendix A.

5.2 Caveats

NV-DNS is in no way a completely stand-alone protection method against covert channels in the DNS protocol. Some of the major problems encountered with implementing the tool are:

- NV-DNS must be able to recover from all errors without any side effects as a failure to recover results in inaccessibility of service to end hosts.

- The current implementation of NV-DNS is not time or usage sensitive and does not use a modified version of [20]'s analysis model to analyze the frequency of the requests.
- The current implementation is not meant for any type of high-performance analysis.
- The current implementation does not keep a cache of requests and replies to improve response time and localize covert channel communications.
- NV-DNS in its current implementation does not handle TCP-based DNS or most server communications.
- The current implementation is not multi-threaded and does not write data to the network.

5.3 Applying NV-DNS

Once NV-DNS was successfully able to continually run transparently and without error, the next step was to determine how well the program responded to a mixed group of valid and invalid DNS requests. A covert channel packet encoder/decoder was created to test the full capabilities of NV-DNS. A small text file was used as the data feed for the covert channel. The source code listing for the program is in Appendix B. User's guides for the programs are also available in their respective appendices. The results from the program are shown in Appendix C.

5.4 Summary

This chapter discussed the implementation of a very rudimentary version of Network-Validated Domain Name Service that was capable, to a degree, of identifying and limiting covert channels on a client-only network segment. The data generated by NV-DNS is a copy of the original request and reply received by the program that can be reviewed in the future to permit the application of future analysis techniques on the protocol.

Chapter 6
6 Summary and Conclusion

6.1 Summary and Conclusion

The research conducted for this thesis resulted in the design of a potentially useful framework for providing security professionals, network administrators, and developers with a means of effectively limiting the capability of covert channels within DNS communications.

The major drawback of the goal of limiting covert channels within DNS communications lies in the required functionality of the protocol. There is only so much information that can be specified independently of the user before some input from the user is required for the protocol to work. This leaves the potential for validated requests that contain some type of covert channel to be assembled at another location without any trouble. However, the work provided within this document is surely a significant step towards limiting covert channels within a protocol.

This research also focused on the construction of a rudimentary validation tool known as NV-DNS inspired by research in real-time network analysis to apply some type of real-time network validation of a packet's structure and information. NV-DNS as implemented and described in this work is only capable of handling a minimal number of concurrent DNS requests and replies. The tool could be used to identify and limit a number of covert channel requests but due to the simplicity of the detection algorithms, was unable to discover a broader range of covert channel attempts without further implementation.

6.2 Future Work

The following work may be considered as a future project:

- Providing full standard compliance for the DNS protocol.
- Implementing additional detection and response features, such as those suggested by [20].
- Development of a more efficient, multithreaded version of NV-DNS.
- Implementation of NV-DNS with a DNS server.
- Implementation of an analysis tool to review saved packets.

Appendix A
NV-DNS

A.1 Overview

This appendix provides a listing of the NV-DNS tool discussed in Chapters 4 and 5. C was the chosen implementation language for the tool due to its high level of control and mostly standard implementation across platforms.

A.2 Usage

To utilize NV-DNS, issue the command: *nvdns IP filename*. The *IP* argument indicates the DNS server or resolver for the current network segment. The argument *filename* indicates to the program the name of the file that the received packets should be written to. This represents the path and name of the file to be written.

A.3 Source Code Listing

A.3.1 defines.h

```
/*****************************************************/
/*  NV-DNS                                           */
/*  Author    : Rex McCracken                        */
/* Date Created : September 24 2004                  */
/*  Last Modified   : December 1 2004                */
/*  Description     : Program to remove covert       */
/*                  channels in DNS packets          */
/*****************************************************/
/*

  DNS-specific definitions

  Per RFCs 1034 and 1035

  Type values
  Class values
  Error codes

  */

/*  General defines  */
#define MAX_UDP_SIZE 512
#define DNS_PORT 53


/*  Network error defines  */
#define ERROR_SOCKET_UNBOUND -100
#define ERROR_BIND_ERROR -101

/*  DNS Header offsets  */
#define HEADER_QR 0x80
```

```
#define HEADER_OPCODE 0x78
#define HEADER_AA 0x04
#define HEADER_TC 0x02
#define HEADER_RD 0x01
#define HEADER_RA 0x80
#define HEADER_Z 0x70;
#define HEADER_RCODE 0x0F
/*  End DNS Header offsets  */

/* Error Codes */
#define INVALID_QR              0x80000001
#define INVALID_OPCODE          0x80000002
#define INVALID_AA              0x80000004
#define INVALID_TC              0x80000008
#define INVALID_RD              0x80000010
#define INVALID_RA              0x80000020
#define INVALID_Z               0x80000040
#define INVALID_RCODE           0x80000080
#define INVALID_QDCOUNT         0x80000100
#define INVALID_ANCOUNT         0x80000200
#define INVALID_NSCOUNT         0x80000400
#define INVALID_ARCOUNT         0x80000800
#define INVALID_QTYPE           0x80001000
#define INVALID_QCLASS          0x80002000
#define INVALID_TYPE            0x80004000
#define INVALID_CLASS           0x80008000
#define INVALID_ADDRESS         0x80010000
#define INVALID_PROTOCOL   0x80020000
#define INVALID_DOMAIN          0x80040000
#define INVALID_POINTER         0x80080000
#define INVALID_STRING          0x80100000
#define INVALID_PARSING         0x80200000
```

A.3.2 dnstype.h

```
/*************************************************/
/*  NV-DNS                                       */
/*  Author    : Rex McCracken                    */
/* Date Created : September 24 2004              */
/*  Last Modified   : December 1 2004            */
/*  Description     : Program to remove covert   */
/*                  channels in DNS packets      */
/*************************************************/



#include "defines.h"
/*  DNS type function prototypes  */

#define INBOUND 1
#define OUTBOUND 0

typedef struct dns_def_type {
   short unsigned int pid;
   char qr;
   short unsigned int opcode;
   char aa;
   char tc;
   char rd;
   char ra;
   char z;
   short unsigned int rcode;
   short unsigned int qdcount;
```

```
    short unsigned int ancount;
    short unsigned int nscount;
    short unsigned int arcount;
    char data[MAX_UDP_SIZE];
    int data_len;
} dns_def;


/* DNS Parse
  Inputs:
    data      character pointer to the packet received

  Outputs:
    dns_def*  Pointer to allocated memory structure
*/
dns_def *dnsparse(char *data);

/*  Process Packet
  Inputs:
    dns       Pointer to the parsed DNS packet
    direction Traffic flow direction

  Outputs:
    int       Error code
*/
int process_packet(dns_def *dns, int direction);

/* Packet Crafter
  Inputs:
    dns       Pointer to dns_def struct holding the DNS packet
    ndata     Pointer to new packet buffer
    idbuffer  Pointer to stored ID header
    fields    Error code for the fields that need overwritten
    direction Direction of the traffic flow

  Outputs:
    int       Number of characters the ndata buffer is
*/
int packet_crafter(dns_def *dns, char *ndata, char *idbuffer, int fields, int
direction);

/*  Display Errors
  Inputs:
    error     Error codes
    direction Traffic direction
    dns       Pointer to dns_def holding data
*/
void display_errors(int error, int direction, dns_def *dns);

/*  STRing All CoPY
  Inputs:
    dest   Pointer to destination buffer
    src    Pointer to source buffer
    len    Number of characters to overwrite

  Outputs:
    int    Number of characters written
*/
int stracpy(char *dest, const char *src, int len);
```

A.3.3 dnstype.c

```
/*************************************************/
/*  NV-DNS                                       */
/*  Author    : Rex McCracken                    */
/* Date Created : September 24 2004              */
/*  Last Modified   : December 1 2004            */
/*  Description     : Program to remove covert   */
/*                  channels in DNS packets      */
/*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "defines.h"
#include "dnstype.h"
#include <string.h>

/*  STRing All CoPY
  Inputs:
   dest   Pointer to destination buffer
   src    Pointer to source buffer
   len    Number of characters to overwrite

  Outputs:
   int    Number of characters written
*/
int stracpy(char *dest, const char *src, int len) {
   int i;
   for (i=0; i<len; i++) {
      dest[i]=src[i];
   }
   return i;
}

/*  Validate header
  Inputs:
   dns       Pointer to dns_def holding the packet data
   direction Traffic flow direction

  Outputs:
   int       Error code
*/
int validate_header(dns_def *dns, int direction) {
   int error=0;
   if (dns->opcode>2)
      error=error|INVALID_OPCODE;
   if (dns->z>0)
      error=error|INVALID_Z;
   if (dns->qdcount==0)
      error=error|INVALID_QDCOUNT;
   if ((dns->data_len<MAX_UDP_SIZE)&&(dns->tc)==1){
      error=error|INVALID_TC;
   }
   if (direction==OUTBOUND) {
      if (dns->qr==1)
         error=error|INVALID_QR;
      if (dns->aa==1)
         error=error|INVALID_AA;
      if (dns->rd==1)
         error=error|INVALID_RD;
      if (dns->ra==1)
         error=error|INVALID_RA;
      if (dns->rcode>0)
```

```
            error=error|INVALID_RCODE;
        if (dns->ancount>0)
            error=error|INVALID_ANCOUNT;
        if (dns->nscount>0)
            error=error|INVALID_NSCOUNT;
        if (dns->arcount>0)
            error=error|INVALID_ARCOUNT;
    }
    if (direction==INBOUND) {
        if (dns->qr==0)
            error=error|INVALID_QR;
        if (dns->rcode>5)
            error=error|INVALID_RCODE;
    }
    return error;
}

/* DNS Parse
  Inputs:
    data      character pointer to the packet received

  Outputs:
    dns_def*  Pointer to allocated memory structure
*/
dns_def* dnsparse(char *data) {
    dns_def *dns_packet;
    //Load packet header into memory
    dns_packet=(dns_def *)malloc(sizeof(dns_def));
    dns_packet->pid=data[0];
    dns_packet->pid=dns_packet->pid<<8;
    dns_packet->pid=dns_packet->pid|data[1];
    dns_packet->qr=data[2]&HEADER_QR;
    dns_packet->qr=dns_packet->qr>>7;
    dns_packet->qr=dns_packet->qr&0x01;
    dns_packet->opcode=data[2]&HEADER_OPCODE;
    dns_packet->opcode=dns_packet->opcode>>3;
    dns_packet->aa=data[2]&HEADER_AA;
    dns_packet->aa=dns_packet->aa>>2;
    dns_packet->tc=data[2]&HEADER_TC;
    dns_packet->tc=dns_packet->tc>>1;
    dns_packet->rd=data[2]&HEADER_RD;
    dns_packet->ra=data[3]&HEADER_RA;
    dns_packet->ra=dns_packet->ra>>7;
    dns_packet->ra=dns_packet->ra&0x01;
    dns_packet->z=data[3]&HEADER_Z;
    dns_packet->z=dns_packet->z>>4;
    dns_packet->rcode=data[3]&HEADER_RCODE;
    dns_packet->qdcount=data[4];
    dns_packet->qdcount=dns_packet->qdcount<<8;
    dns_packet->qdcount=dns_packet->qdcount|data[5];
    dns_packet->ancount=data[6];
    dns_packet->ancount=dns_packet->ancount<<8;
    dns_packet->ancount=dns_packet->ancount|data[7];
    dns_packet->nscount=data[8];
    dns_packet->nscount=dns_packet->nscount<<8;
    dns_packet->nscount=dns_packet->nscount|data[9];
    dns_packet->arcount=data[10];
    dns_packet->arcount=dns_packet->arcount<<8;
    dns_packet->arcount=dns_packet->arcount|data[11];
    return dns_packet;
}
```

```
/*  Process Packet
  Inputs:
    dns       Pointer to the parsed DNS packet
    direction Traffic flow direction

  Outputs:
    int       Error code
*/

int process_packet(dns_def *dns, int direction) {
    char *s;
    int current;
    int err_code=0, temp=0;

    /* Validate the header */
    temp=validate_header(dns, direction);
    err_code=err_code|temp; //Add the error code

    s=dns->data; //Get start of data
    current=12;//Jump to end of data
    //Check the count size to the packet size
    if ((dns->qdcount*6)+(dns->ancount*12)+(dns->nscount*12)+(dns-
>arcount*12)+current>=dns->data_len){

    err_code=err_code|INVALID_QDCOUNT|INVALID_ANCOUNT|INVALID_NSCOUNT|INVALID_AR
COUNT;
    }
    /* Parse the Question section */
    if (direction==OUTBOUND){
        if (dns->qdcount>1)
            err_code=err_code|INVALID_QDCOUNT;
        if (dns->ancount>0)
            err_code=err_code|INVALID_ANCOUNT;
        if (dns->nscount>0)
            err_code=err_code|INVALID_NSCOUNT;
        if (dns->arcount>0)
            err_code=err_code|INVALID_ARCOUNT;
    }

    return err_code;
}

/*  Display Errors
  Inputs:
    error     Error codes
    direction Traffic direction
    dns       Pointer to dns_def holding data
*/

void display_errors(int error, int direction, dns_def *dns) {
    int start=0x00000001;
    int val=0, i;
    if (error<0) {
        if (direction==OUTBOUND)
            printf("Client mode errors:\n");
        else
            printf("Server mode errors:\n");
        for (i=0; i<32; i++) {
            val=error&start;
            val=val|0x80000000;
            switch(val) {
            case INVALID_QR:
                printf("  Query - %d\n", dns->qr);
```

```
            break;
         case INVALID_OPCODE:
            printf("  Opcode - %d\n", dns->opcode);
            break;
         case INVALID_AA:
            printf("  AA - %d\n", dns->aa);
            break;
         case INVALID_TC:
            printf("  TC - %d\n", dns->tc);
            break;
         case INVALID_RD:
            printf("  RD - %d\n", dns->rd);
            break;
         case INVALID_RA:
            printf("  RA - %d\n", dns->ra);
            break;
         case INVALID_Z:
            printf("  Z - %d\n", dns->z);
            break;
         case INVALID_RCODE:
            printf("  Rcode - %d\n", dns->rcode);
            break;
         case INVALID_QDCOUNT:
            printf("  QDcount - %d\n", dns->qdcount);
            break;
         case INVALID_ANCOUNT:
            printf("  ANcount - %d\n", dns->ancount);
            break;
         case INVALID_NSCOUNT:
            printf("  NScount - %d\n", dns->nscount);
            break;
         case INVALID_ARCOUNT:
            printf("  Arcount - %d\n", dns->arcount);
            break;
         default:
            ;
         }
         start=start<<1;
      }
   }
}


/* Packet Crafter
  Inputs:
   dns       Pointer to dns_def struct holding the DNS packet
   ndata     Pointer to new packet buffer
   idbuffer  Pointer to stored ID header
   fields    Error code for the fields that need overwritten
   direction Direction of the traffic flow

  Outputs:
   int       Number of characters the ndata buffer is
*/

int packet_crafter(dns_def *dns, char *ndata, char *idbuffer, int fields, int
direction) {
   int idnew=0;
   int code=0;

   stracpy(ndata, dns->data, dns->data_len);
   //Handle ID
   if (direction==OUTBOUND) {    //Extract & save original request
```

```
        //ID field
        idnew=dns->pid;
        idbuffer[1]=(char)idnew;
        idbuffer[0]=(char)(idnew>>8);
        idnew=rand()%256;
        ndata[0]=idnew&0x0000ff00; //Put new data into packet
        ndata[1]=idnew&0x000000ff;
        code=fields&0x7fffffff;


        if ((code&INVALID_QR)>0){  //Invalid QR type
           ndata[2]=ndata[2]&0x7f;//Clear bit
        }
        if ((code&INVALID_OPCODE)>0){
           ndata[2]=ndata[2]&0x87;//Clear 4 bits
        }
        if ((code&INVALID_AA)>0){
           ndata[2]=ndata[2]&0xfb;//Clear 1 bit
        }
        if ((code&INVALID_TC)>0){
           ndata[2]=ndata[2]&0xfd;//Clear bit 2
        }
        if ((code&INVALID_RD)>0) {
           ndata[2]=ndata[2]&0xfe;//For always desired, |0x01;
        }
        if ((code&INVALID_RA)>0){
           ndata[3]=ndata[3]&0x7f;
        }
        if ((code&INVALID_Z)>0){
           ndata[3]=ndata[3]&0x8f;
        }
        if ((code&INVALID_RCODE)>0){
           ndata[3]=ndata[3]&0xf0;
        }
        if ((code&INVALID_QDCOUNT)>0){
           ndata[4]=0;
           ndata[5]=1;
        }
        if ((code&INVALID_ANCOUNT)>0){
           ndata[6]=0;
           ndata[7]=0;
        }
        if ((code&INVALID_NSCOUNT)>0){
           ndata[8]=0;
           ndata[9]=0;
        }
        if ((code&INVALID_ARCOUNT)>0){
           ndata[10]=0;
           ndata[11]=0;
        }
    }
    else { //Revert original request
        ndata[0]=idbuffer[0];
        ndata[1]=idbuffer[1];
    }
    return dns->data_len;
}
```

A.3.4 nvdns.c

```
/*************************************************/
/*  NV-DNS                                       */
/*  Author    : Rex McCracken                    */
/* Date Created : September 24 2004              */
/*  Last Modified   : December 1 2004            */
/*  Description     : Program to remove covert   */
/*                  channels in DNS packets      */
/*************************************************/

#include <winsock.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

//Local includes
#include "defines.h"
#include "dnstype.h"

/*  Write packet
  Inputs:
   o         File pointer to output data to
   data      Char pointer to data to write
   size      Size of the data buffer
   direction Direction of the data flow
   error     Error code from the packet processor

  Outputs:
   int       Number of characters written from packet
*/
int write_packet(FILE *o, char *data, int size, int direction, int error) {
   int i;
   if (direction==OUTBOUND)
      fprintf(o, "Outbound ");
   else
      fprintf(o, "Inbound ");
   fprintf(o, "%d %d\n", error, size);
   for (i=0; i<size; i++) {
      fprintf(o, "%c", data[i]);
   }
   fprintf(o, "\n\n");
   return i;
}

void usage(char *name){
   printf("Usage: %s server filename\n", name);
   printf("\n  Server  \tIP address of the next level DNS server - x.x.x.x\n");
   printf("\n  Filename\tName of the file to store received DNS packets to\n");
}


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


   //Network functionality
   struct sockaddr_in server_sin, client_sin, dns_server;
   int server_size, dns_size, client_size;
   SOCKET server_socket, dns_socket;
   WSADATA wsa_data;
   int status;
```

```
int client_len, len, i, valid;
unsigned int d;
struct timeval timeout;
fd_set sockets, readable;
int fdmax, retry, poll;
int out_bx=0, in_bx=0;
int out_rq=0, in_rq=0;
int error;

char data[MAX_UDP_SIZE], crafted[MAX_UDP_SIZE], idbuf[2];
FILE *f=NULL;
dns_def *dns_data_ptr;

if (argc==1){
   usage(argv[0]);
   exit(1);
}
if ((argc>=2)&& (argc<4)){
   if (strcmp(argv[1], "/?")==0){
      usage(argv[0]);
      exit(0);
   }
   else {
      d=inet_addr(argv[1]);
      if (d==INADDR_NONE){
         usage(argv[0]);
         exit(1);
      }
      f=NULL;
      if (argc==3) {
         if ((f=fopen(argv[1], "w"))==NULL){
            printf("Error creating the file.  Check permissions!\n");
            exit(11);
         }
      }
   }
}

//Startup communications
if ((status = WSAStartup(MAKEWORD(1,1),&wsa_data)) != 0) {
   fprintf(stderr,"%d is the WSA startup error\n",status);
   exit(1);
}
timeout.tv_sec=0;
timeout.tv_usec=500000;
FD_ZERO(&sockets);
FD_ZERO(&readable);

/* Setup the server socket */
server_socket = socket(PF_INET, SOCK_DGRAM, 0);
if (server_socket==INVALID_SOCKET) {
   printf("Error, unable to create server socket!\n");
   exit(ERROR_SOCKET_UNBOUND);
}
memset((void*)&server_sin, 0, sizeof(server_sin));
server_sin.sin_family = AF_INET;
server_sin.sin_port = htons(DNS_PORT);
server_sin.sin_addr.s_addr = htonl(INADDR_ANY);
server_size=sizeof(server_sin);
status=bind(server_socket, (struct sockaddr*)&server_sin, server_size);
if (status==SOCKET_ERROR) {
   printf("Error binding the server socket... try another port.\n");
   exit(ERROR_BIND_ERROR);
```

```
    }
    /* Server socket setup */
    if((f=fopen("packets.txt", "w"))==NULL) {
       printf("error opening file\n");
       exit(2);
    }

    /* Setup the gateway */
    dns_socket = socket(PF_INET, SOCK_DGRAM, 0);
    if (dns_socket==INVALID_SOCKET) {
       printf("Error, unable to bind socket!\n");
       exit(ERROR_SOCKET_UNBOUND);
    }
    memset((void *)&dns_server, 0, sizeof(dns_server));
    dns_server.sin_family = AF_INET;
    dns_server.sin_port = htons(DNS_PORT);
    dns_server.sin_addr.s_addr = d;
    if (dns_server.sin_addr.s_addr == INADDR_NONE) {
       printf("Invalid IP address\n");
       exit(3);
    }
    dns_size=sizeof(dns_server);
    /* Gateway setup */

/* Check for errors (Winsock does this slightly differently) */
/* Clear the structure so that we don't have garbage around */
/* AF means Address Family - same as Protocol Family for now */
/* Fill in port number in address (careful of byte-ordering) */
/* Fill in IP address (careful of byte-ordering) */
/* Bind the sockets for communications */

    FD_SET(dns_socket, &sockets);
    fdmax=dns_socket;
    srand(time(NULL));
    while (1) {
       error=0;
       readable=sockets;
       for (len=0; len<MAX_UDP_SIZE; len++) {  //Clear out the data buffer
          data[len]=0;
       }
       client_len=sizeof(client_sin);
       valid=1;
       while (valid) {
          printf("%d packets out: %d bytes\t%d packet in: %d bytes\n",out_rq,
out_bx, in_rq, in_bx );
          if((in_rq>0)&&(out_rq>0)){
             printf("Avg out:%d\tAvg In:%d\t", (int)out_bx/out_rq,
(int)in_bx/in_rq);
             printf("Ratio:%2.2f/%2.2f\n",
((float)out_bx/(float)(out_bx+in_bx))*(float)100,
((float)in_bx/(float)(out_bx+in_bx))*(float)100);
          }
          len=recvfrom(server_socket, data, MAX_UDP_SIZE, 0, (struct
sockaddr*)&client_sin, &client_len);
          if (len>=0){
             break;
          }
       }
       out_rq++; //Update statistics
       out_bx+=len;
       data[len]=0;
       dns_data_ptr=dnsparse(data);
       stracpy(dns_data_ptr->data, data, len);
```

```
       dns_data_ptr->data_len=len;
       //Do my processing steps
       error=process_packet(dns_data_ptr, OUTBOUND);
       display_errors(error, OUTBOUND, dns_data_ptr);
       if ((len>0)&&(f!=NULL))
          write_packet(f, data, len, OUTBOUND, error);
       packet_crafter(dns_data_ptr, crafted, idbuf, error, OUTBOUND);
       idbuf[2]=0;
       status=sendto(dns_socket, crafted, len, 0, (struct sockaddr*)&dns_server,
dns_size);
       retry=0;
       do {   //Timeout handling
          poll=select(fdmax+1, &readable, NULL, NULL, &timeout);
          for (d=0; d<=(unsigned int)fdmax; d++) {
             if ((FD_ISSET(d, &readable))&&(d==dns_socket)) {
                len=recvfrom(dns_socket, data, MAX_UDP_SIZE, 0, (struct
sockaddr*)&dns_server, &dns_size);
             }
          }
          switch(poll) {
          case 0:   //Handling if the packet is dropped
             printf("Timeout... Retransmitting\n");
             status=sendto(dns_socket, crafted, len, 0, (struct
sockaddr*)&dns_server, dns_size);
             if (status==SOCKET_ERROR) {
                closesocket(dns_socket);
                dns_socket=socket(PF_INET, SOCK_DGRAM, 0);
                status=sendto(dns_socket, crafted, len, 0, (struct
sockaddr*)&dns_server, dns_size);
             }
             retry++;
             break;
          case SOCKET_ERROR:     //Handling if the socket dies
             printf("Socket error #%d\n", WSAGetLastError());
             i=closesocket(dns_socket);
             printf("closesocket=%d\n", i);
             dns_socket=socket(PF_INET, SOCK_DGRAM, 0);
             status=sendto(dns_socket, crafted, len, 0, (struct
sockaddr*)&dns_server, dns_size);
             break;
          default:
             ;//printf("%d sockets available\n", poll);
          }
       } while ((poll==0)&&(retry<3));

       free((dns_def *)dns_data_ptr);

       data[len]=0;
       in_bx+=len;
       in_rq++;
       dns_data_ptr=dnsparse(data);
       stracpy(dns_data_ptr->data, data, len); //Copy all the data over
       dns_data_ptr->data_len=len;
       client_size=sizeof(client_sin);

       error=process_packet(dns_data_ptr, INBOUND);
       display_errors(error, INBOUND, dns_data_ptr);
       if ((len>0)&&(f!=NULL))
          write_packet(f, data, len, INBOUND, error);
       packet_crafter(dns_data_ptr, crafted, idbuf, error, INBOUND);
       //Send the crafted packet
       status=sendto(server_socket, crafted, len, 0, (struct
sockaddr*)&client_sin, client_size);
```

```
        free((dns_def *)dns_data_ptr);
        printf("\n"); //Give us some spacing for the next time
    }
    return -1;
}
```

Appendix B
Encode/Decode DNS

B.1 Overview

This appendix provides a listing of the Encode/Decode DNS tool used to test the effectiveness of the NV-DNS implementation. C was the chosen implementation language for the tool due to its high level of control and mostly standard implementation across platforms.

B.2 Usage

To use Encode/Decode DNS, issue the command: ***eddns mode field filename [IP]***. The argument ***mode*** tells the program to run in either client (-c) or server (-s) mode. The argument ***field*** determines the DNS field the program will encode and decode the data from. ***filename*** is the name of the file to be read in client mode or written in server mode. ***IP*** is only valid for the client mode and is the IP address of the computer running the server portion of Encode/Decode DNS with the same field option. Running ***eddns*** by itself will give a list of all valid options.

B.3 Source Code Listing

B.3.1 defines.h

```
/*****************************************************/
/*  Encode/Decode DNS                                */
/* Author       : Rex McCracken                      */
/* Date Created : November 21 2004                   */
/*  Last Modified   : December 1 2004                */
/*  Description     : Definitions for edDNS          */
/*                                                   */
/*****************************************************/

/*  Operation Mode definitions */
#define SERVER 1
#define CLIENT 0
#define UNDEFINED -1

/*  Encoding Mode definitions */
#define ID 1
#define QR 2
#define OPCODE 3
#define AA 4
#define TC 5
#define RD 6
#define RA 7
#define Z 8
#define RCODE 9
#define QDCOUNT 10
#define ANCOUNT 11
#define NSCOUNT 12
#define ARCOUNT 13
#define QNAME 14
#define QTYPE 15
#define QCLASS 16
#define NAME 17
#define TYPE 18
#define CLASS 19
#define TTL 20
#define RDLENGTH 21
```

```
#define RR_A 31
#define RR_NS 32
#define RR_MD 33
#define RR_MF 34
#define RR_CNAME 35
#define RR_SOA 36
#define RR_MB 37
#define RR_MG 38
#define RR_MR 39
#define RR_NULL 40
#define RR_WKS 41
#define RR_PTR 42
#define RR_HINFO 43
#define RR_MINFO 44
#define RR_MX 45
#define RR_TXT 46

#define DOMAIN_NAME 0
#define RR_TYPE 2
```

B.3.2 setup.h

```
/*************************************************/
/*  Encode/Decode DNS                            */
/* Author        : Rex McCracken                 */
/* Date Created  : November 21 2004              */
/*  Last Modified   : December 1 2004            */
/*  Description     : Header defining functionality */
/*                  text over DNS packets        */
/*************************************************/

/* Operating Mode
  Inputs:
   mode   String with the choice of modes

  Outputs:
   int    Number designating the operating mode
*/
int operating_mode(char *mode);

/* Encoding Mode
  Inputs:
   mode   String with the choice of encoding fields

  Output:
   int    Number designating the encoding mode
 */
int encoding_mode(char *mode);

/* Encoding Size
  Input:
   Field  Field used for encoding

  Output:
   int    Size of the field in bits
*/
int encoding_size(int field);

/*Client mode functions*/

/* File size
  Input:
   in     File pointer to file needing size check
```

```
  Output:
    int    Size of the file
*/
int file_size(FILE *in);

/* Bit Encode
  Inputs:
    Packet Packet to encode the data into
    Size   Size of the packet
    Field  Field # to encode data into
    Data   Data to encode into field
    Start  Current bit needed encoding

  Output:
    int    Contains the number of bits used in encoding
*/
int bitencode(char packet[], int size, int field, char *data, int start);

/*  Generate Packet
  Input:
    packet Pointer to char array to contain the packet
    field  Number designator for the field being encoded

  Output
    int    Size of the packet generated
*/
int generate_packet(char *packet, int field);


/*Server mode functions*/
/* Bit Decode
  Inputs:
    packet    Packet as received
    size      Packet size
    field     Encoding type
    response  Buffer containing current slice
    rsize     Size of the response
    used      Number of bits used so far in the response

  Outputs:
    char      Updated buffer with newest slice
*/
char* bitdecode(char *packet, int size, int field, char *response, int
*bits_found, int used, int *r_size);

/*  Generate Response
  Inputs:
    packet Pointer to char array containing the packet

  Outputs:
    int    Size of the packet created
*/
int generate_response(char *packet);
```

B.3.3 setup.c

```
/*************************************************/
/*  Encode/Decode DNS                            */
/* Author       : Rex McCracken                  */
/* Date Created : November 21 2004               */
/*  Last Modified   : December 1 2004            */
/*  Description     : Support functions for edDns*/
/*                                               */
/*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "defines.h"
#include "setup.h"

/* Operating Mode
  Inputs:
   mode   String with the choice of modes

  Outputs:
   int    Number designating the operating mode
*/
int operating_mode(char *mode) {
   int a=UNDEFINED;
   if (strcmp(mode, "-s")==0)
      a=SERVER;
   if (strcmp(mode, "-c")==0)
      a=CLIENT;
   return a;
}

/* Encoding Mode
  Inputs:
   mode   String with the choice of encoding fields

  Output:
   int    Number designating the encoding mode
 */
int encoding_mode(char *mode) {
   int a=UNDEFINED;
   if (strcmp(mode, "id")==0)
      a=ID;
   if (strcmp(mode, "qr")==0)
      a=QR;
   if (strcmp(mode, "opcode")==0)
      a=OPCODE;
   if (strcmp(mode, "aa")==0)
      a=AA;
   if (strcmp(mode, "tc")==0)
      a=TC;
   if (strcmp(mode, "rd")==0)
      a=RD;
   if (strcmp(mode, "ra")==0)
      a=RA;
   if (strcmp(mode, "z")==0)
      a=Z;
   if (strcmp(mode, "rcode")==0)
      a=RCODE;
   if (strcmp(mode, "qdcount")==0)
      a=QDCOUNT;
   if (strcmp(mode, "ancount")==0)
```

```
       a=ANCOUNT;
    if (strcmp(mode, "nscount")==0)
       a=NSCOUNT;
    if (strcmp(mode, "arcount")==0)
       a=ARCOUNT;
    return a;
}

/* Encoding Size
  Input:
   Field  Field used for encoding

  Output:
   int    Size of the field in bits
*/
int encoding_size(int field) {
    int a=UNDEFINED;
    switch(field) {
    case QR://QR
    case AA://AA
    case TC://TC
    case RD://RD
    case RA://TA
       a=1;
       break;
    case Z://Z
       a=3;
       break;
    case OPCODE://OPCODE
    case RCODE://RCODE
       a=4;
       break;
    case ID://ID
    case QDCOUNT://QDCOUNT
    case ANCOUNT://ANCOUNT
    case NSCOUNT://NSCOUNT
    case ARCOUNT://ARCOUNT
    case QTYPE://QTYPE
    case QCLASS://QCLASS
    case TYPE://TYPE
    case CLASS://CLASS
    case RDLENGTH://RDLENGTH
       a=16;
       break;
    case TTL://TTL
       a=32;
       break;
    case QNAME://QNAME
    case NAME://NAME
       a=DOMAIN_NAME;
       break;
    default://RR_ types
       a=RR_TYPE;
       break;
    }
    return a;
}

//Client-specific functions

/*  Generate Packet
  Input:
   packet Pointer to char array to contain the packet
```

```
    field  Number designator for the field being encoded

  Output
    int    Size of the packet generated
*/
int generate_packet(char *packet, int field) {
    int a;
    char name[13]={ 3, 'w', 'w', 'w', 3, 'w', 'v', 'u', 3, 'e', 'd', 'u', 0 };
    char *p;
    for (a=0; a<12; a++) {
        packet[a]=0;  //Clear out the packet header for easy viewing
    }
    packet[5]=1; //QDcount
    p=&packet[12];   //Qname start
    strncpy(p, name, 13);
    a+=13;
    packet[25]=0;//Qtype
    packet[26]=1;
    packet[27]=0;//Qclass
    packet[28]=1;
    a=29;
    if (field>16) {  //We have resource records
        ;
    }
    return a;
}

/* File size
  Input:
    in     File pointer to file needing size check

  Output:
    int    Size of the file
*/
int file_size(FILE *in) {
    int a=0;
    char c;
    while (!feof(in)) {
        fscanf(in, "%c", &c);
        a++;
    }
    a--;
    rewind(in);  //Move pointer back to file start
    return a;
}

/*  Bit Slice
  Inputs:
    Data      Data to be encoded
    bit_req       Bits requested for slice
    cur_bit       Offset from data start to be encoded

  Outputs:
    int       Number of bits encoded
    slice     Low-bit packed character
*/
int bitslice(char *data, int bit_req, int cur_bit, int *slice) {
    int byte=0, len=0;
    int bit=0, a=0;
    char *b_slice=0, temp=0;
    int val=0;

    len=strlen(data);
```

```
   bit=cur_bit%8;
   byte=(cur_bit-bit)/8;
   for (a=0; a<bit_req; a++) {
      /* Pack into temp */
      if (bit==0){
         temp=data[byte]&0x80;
         temp=temp>>7;
      }
      if (bit==1) {
         temp=data[byte]&0x40;
         temp=temp>>6;
      }
      if (bit==2){
         temp=data[byte]&0x20;
         temp=temp>>5;
      }
      if (bit==3){
         temp=data[byte]&0x10;
         temp=temp>>4;
      }
      if (bit==4){
         temp=data[byte]&0x08;
         temp=temp>>3;
      }
      if (bit==5){
         temp=data[byte]&0x04;
         temp=temp>>2;
      }
      if (bit==6){
         temp=data[byte]&0x02;
         temp=temp>>1;
      }
      if (bit==7){
         temp=data[byte]&0x01;
      }
      /*Place into slice*/
      val=val<<1;
      val=val|temp;
      bit++;
      if (bit==8){
         bit=0;
         byte++;
      }
      if (byte>len){
         a++;
         break;
      }
   }
   *slice=val;
   return a;
}


/* Bit Encode
  Inputs:
   Packet Packet to encode the data into
   Size   Size of the packet
   Field  Field # to encode data into
   Data   Data to encode into field
   Start  Current bit needed encoding

  Output:
   int    Contains the number of bits used in encoding
```

```
*/
int bitencode(char *packet, int size, int field, char *data, int start) {
    int enc_size=UNDEFINED, esize=0;
    int bits_returned=0;
    int slice=0;
    int bitcount=start;
    int max_bitlen=0, bitlen=0;

    max_bitlen=strlen(data)*8;
    enc_size=encoding_size(field);
    if (enc_size>7) {
        esize=8;
    }
    else
        esize=enc_size;
    while ((bitcount-start)<enc_size) {
        bits_returned=bitslice(data, esize, bitcount, &slice);
        bitcount+=bits_returned;
        switch(field){
        case ID:
            packet[0]=(char)slice;
            bits_returned=bitslice(data, esize, bitcount, &slice);
            bitcount+=bits_returned;
            packet[1]=(char)slice;
            break;
        case QR://QR
            slice=slice<<7;
            packet[2]=packet[2]|slice;
            break;
        case OPCODE://OPCODE
            slice=slice<<3;
            packet[2]=packet[2]|slice;
            break;
        case AA://AA
            slice=slice<<2;
            packet[2]=packet[2]|slice;
            break;
        case TC://TC
            slice=slice<<1;
            packet[2]=packet[2]|slice;
            break;
        case RD://RD
            packet[2]=packet[2]|slice;
            break;
        case RA://RA
            slice=slice<<7;
            packet[3]=packet[3]|slice;
            break;
        case Z://Z
            slice=slice<<4;
            packet[3]=packet[3]|slice;
            break;
        case RCODE://RCODE
            packet[3]=packet[3]|slice;
            break;
        case QDCOUNT://QDCOUNT
            packet[4]=slice;
            bits_returned=bitslice(data, esize, bitcount, &slice);
            bitcount+=bits_returned;
            packet[5]=slice;
            break;
        case ANCOUNT://ANCOUNT
            packet[6]=slice;
```

```
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[7]=slice;
    break;
case NSCOUNT://NSCOUNT
    packet[8]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[9]=slice;
    break;
case ARCOUNT://ARCOUNT
    packet[10]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[11]=slice;
    break;
case QTYPE://QTYPE
    packet[25]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[26]=slice;
    break;
case QCLASS://QCLASS
    packet[27]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[28]=slice;
    break;
case TYPE://TYPE
    packet[42]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[43]=slice;
    break;
case CLASS://CLASS
    packet[44]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[45]=slice;
    break;
case TTL://TTL
    packet[46]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[47]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[48]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[49]=slice;
    break;
case RDLENGTH://RDLENGTH
    packet[50]=slice;
    bits_returned=bitslice(data, esize, bitcount, &slice);
    bitcount+=bits_returned;
    packet[51]=slice;
    break;
default:
    printf("Shouldn't be here\n");
    bitcount=bitlen;
    break;
}
```

```
    }
    return bitcount;
}

//  Server specific functions

/*  Generate Response
  Inputs:
   packet Pointer to char array containing the packet

  Outputs:
   int    Size of the packet created
*/
int generate_response(char *packet) {
    int a;
    char *p, name[13]={3, 'w', 'w', 'w', 3, 'w', 'v', 'u', 3, 'e', 'd', 'u', 0};

    packet[2]=(char)0x84;
    for (a=3; a<12; a++) {
       packet[a]=0;
    }
    packet[5]=1;
    packet[7]=1;
    p=&packet[26];
    strncpy(p, name, 13);
    a=26+13;
    packet[39]=0;
    packet[40]=1;
    packet[41]=0;
    packet[42]=1;
    a=43;
    return a;
}


/* Bit Offset
  Inputs:
   Field  # of the encoding field

  Outputs:
   int    Bits from the start of the packet
*/
int bitoffset(int field) {
    int a;
    switch(field){
    case ID:
       a=0;
       break;
    case QR:
       a=16;
       break;
    case OPCODE://OPCODE
       a=17;
       break;
    case AA://AA
       a=21;
       break;
    case TC://TC
       a=22;
       break;
    case RD://RD
       a=23;
       break;
```

```
    case RA://RA
        a=24;
        break;
    case Z://Z
        a=25;
        break;
    case RCODE://RCODE
        a=28;
        break;
    case QDCOUNT://QDCOUNT
        a=32;
        break;
    case ANCOUNT://ANCOUNT
        a=48;
        break;
    case NSCOUNT://NSCOUNT
        a=64;
        break;
    case ARCOUNT://ARCOUNT
        a=80;
        break;
    case QTYPE://QTYPE
        a=25*8;
        break;
    case QCLASS://QCLASS
        a=27*8;
        break;
    case TYPE://TYPE
        a=42*8;
        break;
    case CLASS://CLASS
        a=44*8;
        break;
    case TTL://TTL
        a=46*8;
        break;
    case RDLENGTH:
        a=50*8;
        break;
    default:
        printf("Error!\n");
        a=-1;
        break;
    }
    return a;
}

/*  Create Mask
  Inputs:
    Field  Number designator of the encoding field

  Outputs:
    char   Character containing a bitmask for that field
*/
char create_mask(int field) {
    char a;
    switch (field){
    case ID:
    case QDCOUNT:
    case ANCOUNT:
    case NSCOUNT:
    case ARCOUNT:
    case QTYPE:
```

```
    case QCLASS:
    case TYPE:
    case CLASS:
    case TTL:
    case RDLENGTH:
       a=(char)0xff;
       break;
    case QR:
    case RA:
       a=(char)0x80;
       break;
    case OPCODE:
       a=0x78;
       break;
    case AA:
       a=0x04;
       break;
    case TC:
       a=0x02;
       break;
    case RD:
       a=0x01;
       break;
    case Z:
       a=0x70;
       break;
    case RCODE:
       a=0x0f;
       break;
    }
    return a;
}

/*  BitExtraction
  Inputs:
    Packet        Packet with data encoded
    Prev_offset      Offset in bits from the last returned octet
    Field         Field # containing the encoded data

  Outputs:
    char          Low order packed byte containing the extracted data
*/
char bitextract(char *packet, int prev_offset, int field){
    int bit=0, byte=0, a=0;
    char extract=0;
    int esize=-1;
    char mask;
    int offset;

    offset=bitoffset(field);
    mask=create_mask(field);//Returns a mask for the field we're working with
    bit=(offset+prev_offset)%8;
    byte=(offset+prev_offset-bit)/8;
    extract=packet[byte]&mask;
    switch(field){   //Extract Data
    case QR:
    case RA:
       extract=extract>>7; //Slide data to low-order
       extract=extract&0x01;      //Clear extra data
       break;
    case OPCODE:
       extract=extract>>3;        //Slide data to low-order
       extract=extract&0x0f;      //Clear extra data
```

```
      break;
   case AA:
      extract=extract>>2; //Slide data to low-order
      extract=extract&0x01;      //Clear extra data
      break;
   case TC:
      extract=extract>>1; //Slide data to low-order
      extract=extract&0x01;      //Clear extra data
      break;
   case Z:
      extract=extract>>4; //Slide data to low-order
      extract=extract&0x07;      //Clear extra data
      break;
   default:
      ;
   }
   return extract;
}

/* Bit Decode
  Inputs:
   packet    Packet as received
   size      Packet size
   field     Encoding type
   response  Buffer containing current slice
   rsize     Size of the response
   used      Number of bits used so far in the response

  Outputs:
   char      Updated buffer with newest slice
*/

char* bitdecode(char *packet, int size, int field, char response[], int
*bits_found, int used, int *rsize) {
   int enc_size=0, r_size=0, e_size=0;
   char slice;
   char *alldata=NULL, p=0;
   int bit=0, byte=0, bitcount=0;

   //Response size
   if (response==NULL) {  //If we don't have a response size, allocate one
      switch (field){
      case ID://ID
      case QDCOUNT://QDcount
      case ANCOUNT://ANcount
      case NSCOUNT://NScount
      case ARCOUNT://ARcount
      case QTYPE://Qtype
      case QCLASS://Qclass
      case TYPE://Type
      case CLASS://Class
      case RDLENGTH://Rdlength
      case Z://Z - Z overflow
         r_size=2;
         break;
      case TTL://TTL
         r_size=4;
         break;
      default://QR, OPCODE, AA, TC, RD, RA, RCODE
         r_size=1;
         break;
      }
      alldata=(char*)calloc(sizeof(char), r_size);
```

```
    }
    else {
       alldata=response;
       r_size=*rsize;
    }
    enc_size=encoding_size(field);
    if (enc_size>7)
       e_size=8; //Max number of bits to get per loop
    else
       e_size=enc_size;    //Bits to get per loop
    while (bitcount<enc_size) {   //Loop to get large fields & small fields
       slice=bitextract(packet, bitcount, field);//Get the appropriate bits
       bitcount+=e_size;   //Add the encoding size to the bits
       if (field==Z) {     //Check for overflow - Applies to Z field ONLY
          bitcount+=used;
          if (alldata==NULL){
             printf("ERROR!!!!!!!!!!!!\n\n");
             exit(1);
          }
          if ((bitcount-8)>0){    //Handle 9 & 10 bits
             if ((bitcount-8)==1) {  //Handle 2 bits over
                alldata[0]=alldata[0]<<2;
                alldata[0]=alldata[0]|((slice&0x06)>>1);   //Mask slice and
align
                alldata[1]=slice&0x01;;  //Mask for final bit
             }
             if ((bitcount-8)==2) {     //Handle 1 bit over
                alldata[0]=alldata[0]<<1;     //Move data over
                alldata[0]=alldata[0]|((slice&0x04)>>2);  //Mask slice and
align
                alldata[1]=slice&0x03;;//Mask for 2 bits
             }
          }
          else { //Handle 0-8 bits
             alldata[0]=alldata[0]<<e_size;   //Move data over
             alldata[0]=alldata[0]|slice;     //Add data
          }
       }
       if ((e_size<8)&&(field!=Z)){     //Handle small fields
          alldata[0]=alldata[0]<<e_size;
          alldata[0]=alldata[0]|slice;
       }
       if (e_size==8) {
          alldata[byte]=slice;
          byte++;
       }
    }
    response=alldata;             //Set / return our response
    *bits_found=enc_size;  //Return the encoding size
    *rsize=r_size;                //Return the buffer size
    return response;       //Return the response
}
```

B.3.4 main.c

```
/*************************************************/
/*  Encode/Decode DNS                            */
/* Author       : Rex McCracken                  */
/* Date Created : November 21 2004               */
/*  Last Modified   : December 1 2004            */
/*  Description     : Program to encode and decode   */
/*                  text over DNS packets        */
/*************************************************/

//Standard Includes
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <winsock.h>

//Local Includes
#include "defines.h"
#include "setup.h"

/* Usage
  Input:
   name   Name of this program

  Output:
   None
*/

void disp_header(char *packet) {
   printf("ID: %c%c\n",packet[0], packet[1]);
   printf("QR: %d\n", (packet[2]&0x80)>>7);
   printf("Op: %d\n", (packet[2]&0x78)>>3);
   printf("AA: %d\n", (packet[2]&0x04)>>2);
   printf("TC: %d\n", (packet[2]&0x02)>>1);
   printf("RD: %d\n", (packet[2]&0x01));
   printf("RA: %d\n", (packet[3]&0x80)>>7);
   printf("Z : %d\n", (packet[3]&0x70)>>4);
   printf("Rc: %d\n", (packet[3]&0x0f));
   printf("QD: %c%c\n", packet[4], packet[5]);
   printf("AN: %c%c\n", packet[6], packet[7]);
   printf("NS: %c%c\n", packet[8], packet[9]);
   printf("AR: %c%c\n", packet[10], packet[11]);
}

void usage(char *name) {
   printf("Usage instructions for %s\n\n", name);
   printf("%s (OP_MODE) (ENCODE_MODE) (FILENAME) [IP] \n\n", name);
   printf("Where:\n");
   printf("OP_MODE\n\t -s \tServer mode\n");
   printf("\t -c \tClient mode\n\n");
   printf("ENCODE_MODE\n");
   printf("\t id\n");
   printf("\t opcode\n");
   printf("\t aa\n");
   printf("\t tc\n");
   printf("\t rd\n");
   printf("\t ra\n");
   printf("\t z\n");
   printf("\t rcode\n");
   printf("\t qdcount\n");
   printf("\t ancount\n");
   printf("\t nscount\n");
```

```
    printf("\t arcount\n");
    printf("\n");
    printf("FILENAME\n");
    printf("\tName of the file to be read (client) or written (server)\n\n");
    printf("IP\n");
    printf("\tIP address NV-DNS is running on\n");
    printf("\n");
}


int main(int argc, char *argv[]) {
    //Network-related Variables
    struct sockaddr_in comm_sin, client_sin, server_sin;
    int comm_size, client_size, server_size;
    int client_len, packet_cnt=0;
    SOCKET sock;
    WSADATA wsa_data;
    int status, len, valid=1, started=0, resp_size=0;
    char msg[512];
    fd_set sockets, readable;
    struct timeval time;
    int fdmax, poll, retry;

    //Local variables
    FILE *f=NULL;
    char name[20], *p=NULL, *s=NULL;
    char ip[35];
    char *data=NULL;
    int op_mode=UNDEFINED, choice=UNDEFINED, enc_mode=UNDEFINED;
    int enc_size=UNDEFINED, i=0;
    int d_size=-1, bitcnt=0, bits_returned=0;
    char c;

    for (i=0; i<=512; i++) {
        msg[i]=0;
    }
    if ((argc<4)||(argc>5)) {
        p=argv[0];
        usage(p);
        exit(1);
    }

    /* Initialize communications */
    if ((status = WSAStartup(MAKEWORD(1,1),&wsa_data)) != 0) {
        fprintf(stderr,"%d is the WSA startup error\n",status);
        exit(1);
    }
    /*  Parse command line */
    s=argv[0];
    p=argv[1];
    op_mode=operating_mode(p);
    if (op_mode==UNDEFINED){
        usage(s);
        exit(1);
    }
    if ((op_mode==CLIENT)&&(argc==4)){
        usage(s);
        exit(1);
    }
    p=argv[2];    //Encoding mode
    enc_mode=encoding_mode(p);
    if (enc_mode==UNDEFINED) {
        usage(s);
```

```
        exit(1);
    }
    enc_size=encoding_size(enc_mode);    //Determine the encoding scheme size
    strcpy(name, argv[3]);//Filename determination
    if (argc==5) {//Client Destination IP
        if (op_mode==CLIENT) {
            strcpy(ip, argv[4]);
        }
        else {
            usage(s);
            exit(1);
        }
    }
    if (op_mode==SERVER) {//File handling
        if ((f=fopen(name, "w"))==NULL) {
            printf("Error opening file for writing!\n");
            usage(s);
            exit(2);
        }
    }
    else { //Client functions
        if ((f=fopen(name, "r"))==NULL) {
            printf("File not found!\n");
            usage(s);
            exit(2);
        }
    }

    //Polling timeout selection
    if (op_mode==CLIENT) {
        time.tv_sec=(long)1;
        time.tv_usec=(long)0;
    }
    else {
        time.tv_sec=(long)5;
        time.tv_usec=(long)0;
    }

    /* Client specific code */
    if (op_mode==CLIENT) {
        //Set socket /network options
        sock=socket(PF_INET, SOCK_DGRAM, 0);
        if (sock==INVALID_SOCKET) {
            printf("Error!  Unable to create a socket!\n");
            exit(10);
        }
        memset((void*)&comm_sin, 0, sizeof(comm_sin));
        comm_sin.sin_family = AF_INET;
        comm_sin.sin_port = htons(53);
        comm_sin.sin_addr.s_addr= inet_addr(ip);
        comm_size=sizeof(comm_sin);

        FD_ZERO(&sockets);
        FD_ZERO(&readable);
        FD_SET(sock, &sockets);
        fdmax=sock;

        printf("Client mode\t");
        printf("Port:%d\tDest:%s:%d\n", sock, inet_ntoa(comm_sin.sin_addr),
htons(comm_sin.sin_port));
        d_size=file_size(f);             //Load the file into memory
        if (d_size>0) {
            data=(char*)calloc(sizeof(char), d_size);
```

```
            for (i=0; i<d_size; i++) {
               fscanf(f, "%c", &c);
               data[i]=c;
            }
            data[d_size]=0;
            for (i=0; i<d_size; i++) {
               printf("%c", data[i]);
            }
            printf("\n");
            printf("Sending this data will require %d packets\n",
d_size*8/enc_size);
            bitcnt=0;
            while (bitcnt<d_size*8) {
               //Generate packet
               len=generate_packet(msg, enc_mode);
               //Insert data
               i=bitencode(msg, len, enc_mode, data, bitcnt);
               bitcnt+=(i-bitcnt);
               retry=0;
               sendto(sock, msg, len, 0, (struct sockaddr*)&comm_sin, comm_size);
               do {
                  readable=sockets;
                  poll=select(fdmax+1, &readable, NULL, NULL, &time);
                  switch (poll) {
                  case 0:
                     printf("Packet timeout, resending.\n");
                     sendto(sock, msg, len, 0, (struct sockaddr*)&comm_sin,
comm_size);
                     retry++;
                     break;
                  case SOCKET_ERROR:
                     printf("Socket Error!\n");
                     bitcnt=d_size*8;
                     break;
                  default:
                     ;
                  }
               } while ((poll==0)&&(retry<3));
               //If we time out too many times
               if (retry>=3) {
                  printf("Server connection unavailable\n");
                  bitcnt=d_size*8;
                  break;
               }
               len=recvfrom(sock, msg, 512, 0, (struct sockaddr*)&comm_sin,
&comm_size);
               msg[len]=0;
               printf("%s packet %d Received a response\n", argv[2], packet_cnt);
               packet_cnt++;
            }
         }
         else {
            printf("Empty file %s!\n", name);
            exit(3);
         }
      }

      /*  Server specific code  */
      else {
         //Set socket / network options
         sock=socket(PF_INET, SOCK_DGRAM, 0);
         if (sock==INVALID_SOCKET) {
            printf("Error!  Unable to create a socket!\n");
```

```
            exit(10);
        }
        memset((void*)&client_sin, 0, sizeof(client_sin));
        memset((void*)&server_sin, 0, sizeof(server_sin));
        server_sin.sin_family = AF_INET;
        server_sin.sin_port = htons(53);
        server_sin.sin_addr.s_addr=htonl(INADDR_ANY);
        server_size=sizeof(server_sin);
        status=bind(sock, (struct sockaddr*)&server_sin, server_size);
        if (status==SOCKET_ERROR) {
            printf("Error binding socket!\n");
            exit(12);
        }
        listen(sock, 10);

        FD_ZERO(&sockets);
        FD_ZERO(&readable);
        FD_SET(sock, &sockets);
        fdmax=sock;

        printf("Server mode: %s:%d\n", inet_ntoa(server_sin.sin_addr),
htons(server_sin.sin_port));
        bitcnt=0;
        client_size=16;
        client_len=0;
        while (valid) {
            if(started) {
                readable=sockets;
                poll=select(fdmax+1, &readable, NULL, NULL, &time);
                switch(poll){
                case 0:
                    printf("Data feed ended.\n");
                    valid=0;
                    break;
                default:
                    ;
                }
            }
            if (valid==0)
                break;
            len=recvfrom(sock, msg, 500, 0, (struct sockaddr*)&client_sin,
&client_size);
            msg[len]=0;
            started=1;
            if (len==SOCKET_ERROR) {
                printf("The last error:%d\n",WSAGetLastError());
                exit(1);
            }
            //Recv packet until timeout
            data=bitdecode(msg, len, enc_mode, data, &bits_returned, bitcnt,
&resp_size);
            data[resp_size]=0;
            bitcnt+=bits_returned;
            if (bitcnt>7) {  //If we have a full packet
                bits_returned=bitcnt%8;
                if ((bits_returned>0)||(enc_mode==Z))//If we don't have a full
byte - Z field
                    bitcnt=resp_size-1;    //Limit by one
                else
                    bitcnt=resp_size;
                for (i=0; i<bitcnt; i++) {
                    fprintf(f, "%c", data[i]);    //Write this to file
                }
```

```
            if (bits_returned>0) {
               data[0]=data[bitcnt];  //Move the data from unfully used to
start
               i=1;
            }
            else
               i=0;
            for (i; i<resp_size; i++) {
               data[i]=0;       //Clear out the rest of the field
            }
            bitcnt=bits_returned;      //Reset the used bits field
            printf("\n");
         }
         len=generate_response(msg);
         len=sendto(sock, msg, len, 0, (struct sockaddr*)&client_sin,
client_size);
         packet_cnt++;
         printf("%s packet %d\n", argv[2], packet_cnt);
         //Sleep(250);
      }
      i=0;
      while(data[i]>0) {
         fprintf(f, "%c", data[i]);
         i++;
      }
   }
   fclose(f);

   if (op_mode==SERVER) {
      printf("Retrieved data: ");
      f=fopen(name, "r");
      while(!feof(f)){
         fscanf(f, "%c", &c);
         if (c>0)
            printf("%c", c);
         c=0;
      }
      printf("\n");
      fclose(f);
   }
   return 0;
}
```

Appendix C
Results from NV-DNS and Encode/Decode DNS

The file used in the encoding process was a simple text file with the text "i'm just a long text file that i'm transmitting across myself." The results of the transmission between the client and server aspects of Encode/Decode DNS alone and with NV-DNS are included in the following subsections.

C.1 ID field

C.1.1 No NV-DNS

```
i'm just a long text file that i'm transmitting across myself.
```

C.1.1 NV-DNS

```
□C□¸□ö□ □ □
□N□¯□½□î□€□Í□‹□R□)□ □O□&□’□ç□Ç□}□Œ□9□W□|□¼□_□ÿ□w□7□ð□‹
```

C.2 QR field

C.2.1 No NV-DNS

```
i'm just a long text file that i'm transmitting across myself.
```

C.2.2 NV-DNS

```
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

C.3 OPCODE field

C.3.1 No NV-DNS

```
i'm just a long text file that i'm transmitting across myself.
```

C.3.2 NV-DNS

```
□ □ □□□□ □ □□□□ □□□□ □□□□ □□□□ □ □ □□□□□□□□□□□□ □□□□□□ □□□□□□ □
```

C.4 AA field

C.4.1 No NV-DNS

```
i'm just a long text file that i'm transmitting across myself.
```

C.4.2 NV-DNS

```
□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

C.5 TC field

C.5.1 No NV-DNS

```
i'm just a long text file that i'm transmitting across myself.
```

C.5.2 NV-DNS

```
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

C.6 RD field

C.6.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.6.2 NV-DNS

□□□

C.7 RA field

C.7.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.7.2 NV-DNS

□□□

C.8 Z field

C.8.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.8.2 NV-DNS

□□□

C.9 RCODE field

C.9.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.9.2 NV-DNS

□□□

C.10 QDCOUNT field

C.10.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.
□

C.10.2 NV-DNS

□□□

C.11 ANCOUNT field

C.11.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.11.2 NV-DNS

C.12 NSCOUNT field

C.12.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.12.2 NV-DNS

C.13 ARCOUNT field

C.13.1 No NV-DNS

i'm just a long text file that i'm transmitting across myself.

C.13.2 NV-DNS

Bibliography

[1] Vericept, “Preventing Identity Theft and the Loss of Intellectual Property”, February 2004.
URL:
http://www.vericept.com/Downloads/WhitePapers/Vericept_Fraud_IdentityTheft_WP.pdf

[2] Anderson, K., “Criminal Threats to Business on the Internet”, February 1, 1999.
URL: http://www.aracnet.com/~kea/Papers/White_Paper.shtml

[3] Mockapetris, P., “Domain Names – Concepts and Facilities”, ISI, November 1987
URL: http://www.ietf.org/rfc/rfc1034.txt?number=1034

[4] “DNS hack leaves corporate networks wide open”, August 2, 2004.
URL: http://software.silicon.com/security/0,39024655,39122803,00.htm

[5] Johnson, N.F.; Jajodia, S., “Steganalysis: The Investigation of Hidden Information” Information Technology Conference, 1998. IEEE, Vol., Iss., 1-3 Sep 1998. Pages: 113-116
URL:
http://ieeexplore.ieee.org/iel4/5774/15421/00713394.pdf?isNumber=15421&prod=STD&arnumber=713394&arNumber=713394&arSt=113&ared=116&arAuthor=Johnson%2C+N.F.%3B+Jajodia%2C+S.

[6] Artz, D., “Digital Steganography: Hiding Data Within Data”, Internet Computing, IEEE, Vol.5, Iss.3, May/Jun 2001. Pages: 75-80
URL:http://ieeexplore.ieee.org/iel5/4236/20242/00935180.pdf?isNumber=20242&prod=STD&arnumber=935180&arNumber=935180&arSt=75&ared=80&arAuthor=Artz%2C+D.

[7] “Transmission Control Protocol”, Information Sciences Institute, University of Southern California, Sep 1981.
URL: http://www.ietf.org/rfc/rfc793.txt?number=793

[8] Postel, J., “User Datagram Protocol”, ISI, 28 Aug 1980
URL: http://www.ietf.org/rfc/rfc768.txt?number=768

[9] Mockapetris, P., “Domain Names – Implementation and Specification”, ISI, November 1987
URL: http://www.ietf.org/rfc/rfc1035.txt?number1035

[10] Provos, N.; Honeyman, P., “Hide and Seek: An Introduction to Steganography” Security & Privacy Magazine, IEEE, Vol.1, Iss.3, May-June 2003 Pages: 32- 44
URL:
http://ieeexplore.ieee.org/iel5/8013/27102/01203220.pdf?isnumber=27102&prod=STD&arnumber=1203220&arnumber=1203220&arSt=+32&ared=+44&arAuthor=Provos%2C+N.%3B+Honeyman%2C+P.

[11] Breneman, J., “Underground Railroad Quilts & Abolitionist Fairs”
URL: http://www.womenfolk.com/quilting_history/abolitionist.htm

[12] Graps, A., “An Introduction to Wavelets”, Computational Science and Engineering, IEEE, Vol.2, Iss.2, Summer 1995 Pages: 50-61
URL: http://ieeexplore.ieee.org/iel4/99/8829/00388960.pdf?isnumber=8829&prod=STD&arnumber=388960&arnumber=388960&arSt=50&ared=61&arAuthor=Graps%2C+A.

[13] Xu, J.; Sung, A.H.; Shi, P.; Liu, Q., “JPEG Compression Immune Steganography Using Wavelet Transform”, Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on, Vol.2, Iss., 5-7, April 2004 Pages: 704- 708
URL: http://ieeexplore.ieee.org/iel5/9035/28683/01286737.pdf?isnumber=28683&prod=STD&arnumber=1286737&arnumber=1286737&arSt=+704&ared=+708+Vol.2&arAuthor=Xu%2C+J.%3B+Sung%2C+A.H.%3B+Shi%2C+P.%3B+Liu%2C+Q.

[14] Venkatraman, B.R.; Newman-Wolfe, R.E., “Capacity estimation and Auditability of Network Covert Channels”,
Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on, Vol., Iss., 8-10 May 1995 Pages: 186-198
URL:
http://ieeexplore.ieee.org/iel2/3181/9013/00398932.pdf?isnumber=9013&prod=STD&arnumber=398932&arnumber=398932&arSt=186&ared=198&arAuthor=Venkatraman%2C+B.R.%3B+Newman-Wolfe%2C+R.E.

[15] Moskowitz, I.S.; Kang, M.H., “Covert channels - Here to Stay?”, Computer Assurance, 1994. COMPASS '94 'Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security'. Proceedings of the Ninth Annual Conference on, Vol., Iss., 27 Jun-1 Jul 1994 Pages: 235-243,
URL:
http://ieeexplore.ieee.org/iel2/1114/7667/00318449.pdf?isnumber=7667&prod=STD&arnumber=318449&arnumber=318449&arSt=235&ared=243&arAuthor=Moskowitz%2C+I.S.%3B+Kang%2C+M.H.

[16] Ogurtsov, N.; Orman, H.; Schroeppel, R.; O'Malley, S.; Spatscheck, O., "Experimental Results of Covert Channel Limitation in One-way Communication Systems", Network and Distributed System Security, 1997. Proceedings., 1997 Symposium on, Vol., Iss., 10-11 Feb 1997 Pages: 2-15
URL:
http://ieeexplore.ieee.org/iel3/4421/12557/00579214.pdf?isnumber=12557&prod=STD&arnumber=579214&arnumber=579214&arSt=2&ared=15&arAuthor=Ogurtsov%2C+N.%3B+Orman%2C+H.%3B+Schroeppel%2C+R.%3B+O%27Malley%2C+S.%3B+Spatscheck%2C+O.

[17] Millen, J., "20 Years of Covert Channel Modeling and Analysis", Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on, Vol., Iss., 1999 Pages: 113-114
URL:
http://ieeexplore.ieee.org/iel5/6220/16605/00766906.pdf?isnumber=16605&prod=STD&arnumber=766906&arnumber=766906&arSt=113&ared=114&arAuthor=Millen%2C+J.

[18] "Bell Labs Celebrates 50 years of Information Theory: An Overview of Information Theory"
URL:http://www.lucent.com/minds/infotheory/docs/history.pdf

[19] Pierce, J., "The Early Days of Information Theory", Information Theory, IEEE Transactions on, Vol.19, Iss.1, Jan 1973 Pages: 3- 8
URL:
http://ieeexplore.ieee.org/iel5/18/22667/01054955.pdf?isnumber=22667&prod=STD&arnumber=1054955&arnumber=1054955&arSt=+3&ared=+8&arAuthor=+Pierce%2C+J.

[20] Xu Gang; Zhang Hui, "Advanced Methods for Detecting Unusual Behaviors on Networks in Real-Time", Communication Technology Proceedings, 2000. WCC - ICCT 2000. International Conference on, Vol.1, Iss., 2000 Pages: 291-295
URL:
http://ieeexplore.ieee.org/iel5/7138/19245/00889216.pdf?isNumber=19245&prod=STD&arnumber=889216&arNumber=889216&arSt=291&ared=295+vol.1&arAuthor=Xu+Gang%3B+Zhang+Hui

[21] Thompson, K.; Miller, G.J.; Wilder, R., "Wide-Area Internet Traffic Patterns and Characteristics", Network, IEEE, Vol.11, Iss.6, Nov/Dec 1997 Pages: 10-23
URL:
http://ieeexplore.ieee.org/iel4/65/13911/00642356.pdf?isnumber=13911&prod=STD&arnumber=642356&arnumber=642356&arSt=10&ared=23&arAuthor=Thompson%2C+K.%3B+Miller%2C+G.J.%3B+Wilder%2C+R.

[22] Corey, V.; Peterman, C.; Shearin, S.; Greenberg, M.S.; Van Bokkelen, J., "Network Forensics Analysis", Internet Computing, IEEE, Vol.6, Iss.6, Nov/Dec 2002 Pages: 60 - 66
URL:
http://ieeexplore.ieee.org/iel5/4236/22924/01067738.pdf?isnumber=22924&prod=STD&arnumber=1067738&arnumber=1067738&arSt=+60&ared=+66&arAuthor=Corey%2C+V.%3B+Peterman%2C+C.%3B+Shearin%2C+S.%3B+Greenberg%2C+M.S.%3B+Van+Bokkelen%2C+J.