Enhanced Secure DNS:

A Defense Against DDOS Attacks

by

DAVID BOYD WILKINSON

B.S., University of Colorado, 1993

A thesis submitted to the Graduate Faculty of the

University of Colorado at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2003

This thesis for the Master of Science degree by

David B. Wilkinson

has been approved for the

Department of Computer Science

by

____

C. Edward Chow

____

Maria Augusteijn

____

Jugal Kalita

____

Date

Wilkinson, David Boyd (M.S., Computer Science)

Enhanced Secure DNS: A Defense Against DDOS Attacks

Thesis directed by Professor C. Edward Chow

The success of distributed denial of service (DDOS) attacks carried out against e-business websites in February 2000 pointed out the deficiencies of an Internet where millions of dollars are transacted daily. Many experts call for the cooperation of the Internet community as a whole to follow safe network administation practices in order to foil the DDOS threat, but such advice mostly goes unheeded.

A novel strategy to fight DDOS attacks, called intrusion tolerance, argues that such attacks are impossible to stop, so it is better to find a way to successfully tolerate them. One such proposal, termed the Secure Collective Defense (SCOLD), uses next-generation domain name system (DNS) Berkeley Internet Name Domain (BIND) software to enable preferred clients to communicate with computers on a network that is being attacked.

The features of this new BIND code include a program, nsreroute, that securely installs an authoritative zone for the victim domain onto a client DNS server. This BIND software sets up an IP tunnel that forces queries from the client DNS server to the victim DNS server to traverse a SCOLD-aware proxy server, which forwards the query through a secret alternate gateway.

This thesis chronicles the addition of code made to the latest version of BIND to satisfy the requirements of SCOLD. The results of experiments using the enhanced DNS BIND in the SCOLD architecture is detailed. This paper ends with the many lessons I have learned from this project.

DEDICATION

To Louyi, for your encouragement and patience.

ACKNOWLEGEMENTS

Many thanks go to Dr. C. Edward Chow at the University of Colorado at Colorado Springs, who introduced me to computer networks. He always took the time to answer my silly questions. Without his generous help my task would have been much more onerous.

Thanks also to Yu Cai, graduate student at UCCS, for taking the time to rig up the SCOLD testbed. He saved me untold amounts of hassle in keeping the virtual machines running. Yu Cai also helped me understand IP tunnel.

Thanks also to Ganesh Godivari, graduate student at UCCS and resident OpenSSL guru, who helped me understand some of the finer points of OpenSSL.

CONTENTS

CHAPTER

I. INTRODUCTION 1

II. DISTRIBUTED DENIAL OF SERVICE (DDOS) 6

Attacker's Modus Operandi 6

DDOS Architecture 7

Types of DDOS Attacks 9

III. PROPOSED DEFENSES AGAINST DDOS ATTACKS 11

Community Prevention .... .11

Secure Collective Defense (SCOLD) 14

SCOLD Architecture 15

Need for Enhanced DNS 22

IV. DOMAIN NAME SYSTEM 23

Brief History of DNS 23

DNS Name Space 24

DNS Zones 27

DNS Zone Files 28

named Configuration File 30

Dynamic Update (nsupdate) 32

V. ENHANCING DNS BIND FOR SCOLD 34

Understanding Client and Server Components of BIND 35

The GNU GDB Debugger and DDD Graphical User Interface 36

VI. NEW CLIENT PROGRAM: NSREROUTE 41

Overview 41

Execution Flow of nsreroute 44

reroutemsg Data Structure 47

Code for nsreroute 51

Adding the New ALT rdatatype 61

Using OpenSSL to Authenticate Server 63

VII. ENHANCEMENT OF BIND FOR HANDLING NSREROUTE MESSAGES 69

Execution Flow of Named Daemon After Receiving nsreroute Message 69

Code Additions 74

Using OpenSSL to Authenticate Client 82

Resulting Zone File and Zone Statement Written to Client DNS Server 87

VIII. ENHANCEMENT OF BIND FOR CLIENT QUERY OF HOST IN VICTIM DOMAIN 90

Normal Query Handling by Non-Authoritative DNS Server 91

New Query Handling by Server with Desired Zone Installed from Reroutemsg 95

Implementing IP Tunnel for Rerouting Query 98

Code for IP Tunnel Setup, Destroy Scripts 99

Other Code Additions 103

In *query_find* 103

Implementation of Caching ALT rdataset
in Query Reply Message 108

Implementation of Retrieval of ALT rdataset in
Zone Database or Cache Database 111

IX. RESULTS OF TESTING ENHANCED BIND
ON SCOLD TESTBED 116

Comparisons of Times in nsreroute & Query Tests 120

X. CONCLUSIONS 123

Lessons Learned 123

Future Work 128

BIBLIOGRAPHY 130

APPENDIX

USER'S GUIDE TO INSTALLING ENHANCED BIND
ONTO SCOLD VIRTUAL MACHINES 132

CHAPTER I

INTRODUCTION

Nasty Internet traffic such as viruses, worms, and other devilish surprises has become more prevalent as hackers try to "one up" each other to see who can write the most destructive piece of code. In 1999 it was the Melissa virus using holes in Microsoft Outlook to propagate itself; in 2001 the Code Red worm made headlines by defacing web sites with the message, "Welcome to http://www.worm.com! Hacked by Chinese!" Early this year the Slammer worm played havoc with Microsoft SQL Server products, and in the summer the latest strain of the Sobig virus and the Blaster worm slithered into thousands of unsuspecting corporate and home computers to run amok in operating systems and user applications.

After one glance at the website of Network Associates, Inc., it is obvious that new Internet threats are emerging almost daily. In response, Congress is entertaining new legislation, similar to the Sarbanes-Oxley Act of 2002 for accounting practices, that would require public companies to report efforts taken to strengthen network security [Gro03]. This measure would make corporations accountable for operating insecure networks and any mischief arising therefrom.

Even Microsoft, trying to stem the rising tide of anger against their products that have been so easy to compromise, has undertaken several steps to enhance security. Last

year they launched the "Trustworthy Computing Initiative," whereby 8,500 Microsoft engineers analyzed and revamped millions of lines of code for the Windows Server 2003, placing security front and center [HGEK03]. In August Microsoft even issued advertisements in several major newspapers, imploring computer users to "Protect Your PC" and install updates to software as they become available [Del03].

However, one security expert called Microsoft's public cries for increased vigilance "pathetic" [Del03]. He dourly noted, "People who are not susceptible to viruses and worms don't need the advice and those who are susceptible just aren't teachable [Del03]."

Unfortunately, the current lack of cohesion across the Internet on adopting safe network security practices is helping make one kind of attack much more deadly. The distributed denial of service (DDOS) attack has recently enjoyed success in shutting down popular websites, depriving these businesses of millions of dollars of revenue.

In February 2000 DDOS attacks made Yahoo, Amazon.com, eBay, and other prominent e-businesses disappear for up to a few hours [Wil00]. During this time Yahoo lost an estimated $500,000 in advertising and other revenue; Amazon.com's loss was around $600,000 [DGLRS02]. The total amount of revenue lost by online companies from these and other less spectacular denial of service attacks in 2000 exceeded $1.7 billion [DGLRS02].

More recent DDOS assaults have targeted critical Internet systems that, if shut down, would severely limit the ability of computers worldwide to communicate. In October 2002 the 13 root Domain Name System (DNS) servers that form the Internet's "backbone" were assailed [Kre02]. Although the attack failed, information concerning

the amount of data that would be required for success was broadcast by the media, giving potential malefactors valuable knowledge for launching future attacks [KM02].

A DDOS attack simply overwhelms the victim's network, making it inaccessible to normal customers. The DDOS tool, available in a variety of flavors, can be downloaded and used with devastating results by any malicious hacking enthusiast with only a modicum of computing savvy.

The attack works this way: An attacker using a stolen account creates a program that remotely breaks into hundreds of computers, setting up each system to launch a later attack on a selected network. Each of these compromised systems in turn breaks into a hundred more. All traces of the unauthorized entry are carefully hidden.

Once the attacker is ready, he issues a remote command, which causes his network of "zombie" hosts to send an avalanche of Internet traffic to a targeted network. Unless the victim has an enormous amount of bandwidth and can handle the large influx of traffic, he is shut down. Once the victim network is flooded, no legitimate traffic, inbound or outbound, can get through.

Various software and hardware enhancements have been proposed to defend and/or trace back DDOS attacks. One such strategy, the Secure Collective Defense (SCOLD), assumes such attacks will occur and promotes intrusion tolerance, i.e., keeping networks open to preferred customers during a DDOS flood [Cho03]. At the heart of the SCOLD architecture lives the domain name system (DNS) servers, which use the popular Berkeley Internet Name Domain (BIND) software. Currently, however, DNS BIND does not have the capabilities required by SCOLD.

This thesis details the SCOLD architecture and my implementation of new functionality in the BIND software that meets the needs of the SCOLD defense system.

Specifically, this new BIND code does a couple of new things. It creates a new program, nsreroute, which sends a message from a proxy server acting on behalf of the victim to the client DNS server. This new message requires that each side verifies the identity of the other. After successful authentication, the client DNS server creates an authoritative zone for the network being victimized by a DDOS attack.

Thereafter, any query handled by the client DNS server for the victim's network retrieves resource records of type A and ALT from the victim's own DNS server through an indirect connection. This is also new DNS behavior that I have implemented in new BIND code. The new ALT-type IP addresses refer to proxy servers that are collaborating in the SCOLD system. IP tunneling forces the client DNS server to send the query through a proxy server (whose address is given by an ALT resource record in the previously created zone file) to get the victim's IP addresses.

Client packets subsequently sent to the victim will go to the intermediate proxy server, which forwards them through a secret alternate gateway that is not being attacked. This way the client avoids the flooding at the main gateway and remains in contact with the victim.

The rest of this thesis paper is organized as follows:

- Chapter II gives an overview of DDOS attacks.
- Chapter III describes a couple of proposed solutions to the DDOS threat.
- Chapter IV deals with fundamental components of the domain name system.

- Chapter V discusses important elements in tracing BIND code, including some prominent files and the GNU GDB debugger with DDD GUI.
- Chapter VI details the development of the new client program in BIND, nsreroute, including the implementation of a new data type, ALT, and OpenSSL code that implements server authentication.
- Chapter VII describes the addition of code in the server component of BIND that handles the new message generated by the nsreroute program, including using OpenSSL to authenticate the nsreroute sender.
- Chapter VIII details how the BIND server was enhanced for handling queries from a client for the address of a host in the victim domain, after the reroutemsg has been processed and installed. Code additions in BIND for setting up and destroying the IP tunnel, as well as retrieving and caching the new ALT datatype in the function query_find, will also be shown.
- Chapter IX displays the results of tests performed with the enhanced DNS BIND software in the SCOLD testbed.
- Chapter X compares test times on nsreroute and queries. It concludes with lessons I have learned in undertaking this project, and ideas on future development of next-generation DNS BIND.

CHAPTER II

DISTRIBUTED DENIAL OF SERVICE (DDOS)

In this chapter I discuss the typical DDOS attacker's method of operation, and examine the DDOS attack architecture.

Attacker's Modus Operandi

An attacker bent on setting up a DDOS network must first secure a "home base" for operations, as it were. This involves using a stolen account on a system with, probably, many users and a fast Internet connection. A university is the ideal setting for this. On the account the attacker stores his pre-compiled software, which would generally include scanning tools to find open ports, root kits to hide his presence, breaking and entering tools, and the DDOS tools [Dit99a].

From this home base the attacker locates vulnerable computers by conducting a comprehensive scan across the Internet [Dit99a]. He looks for services running on open ports that can be compromised remotely through a buffer overflow exploit. On average it takes only five seconds to break in and install the "handler" software. The handler daemon is programmed to go out looking for more hosts to ensnare in the growing web of computers secretly controlled by the attacker.

This lowest level of enslaved systems, the "agents," are the ones that do the attacker's dirty work, sending out a synchronized flood to the targeted web site.

Depending on the kind of DDOS tool that is to be used, or at what level of the hierarchy, sometimes a hijacked system opens a TCP port for listening for commands from the machine that "owns" [Dit99a] it. The compromised system may send back an acknowledgment of its status using this port, or might even email the intruder at a free email account [Dit99a]. Connectionless protocols, such as UDP or ICMP, are also used in control communication between handler and agent [Dit99b].

Seizing upon a moment of opportunity, the intruder issues the command to attack. This is generally given during a telnet session between attacker and master hosts, where the attacker supplies the required password [Dit99a]. The network of agents gets the call from the handlers, and dutifully carry out their assigned tasks.

DDOS Architecture

The network of commandeered computer equipment now directs a withering blizzard of Internet traffic toward one unlucky IP address, or an entire subnet. Actually, the packets sent out from this army of zombies usually has one very important characteristic: spoofed source IP addresses. A fake source IP address adds a magnitude of difficulty to computer forensic investigators trying to trace the attack back to point of origin. Of course, even getting back to the agent hosts will not necessarily tell them anything about the handlers, and trying to trace communication between handlers and attacker will not get them

anywhere close to the identity of the attacker, unless he has kindly left them his name, address and phone number in the stolen account.

Moreover, the source IP address on the attack packets are usually those of the victim. The DDOS attack, in most cases, maximizes its distributed power by bouncing the traffic off intermediate networks, whereby all machines reply en masse to the victim. This is possible if the packet is, say, an ICMP echo request directed at the broadcast address of the intermediate network (whereby the subnet portion of the address is all binary 1's). Unless the network is actually practicing some sort of safe security policy and drops all packets destined for the broadcast address, the number of ICMP echo replies multiply the attack geometrically, if not exponentially.

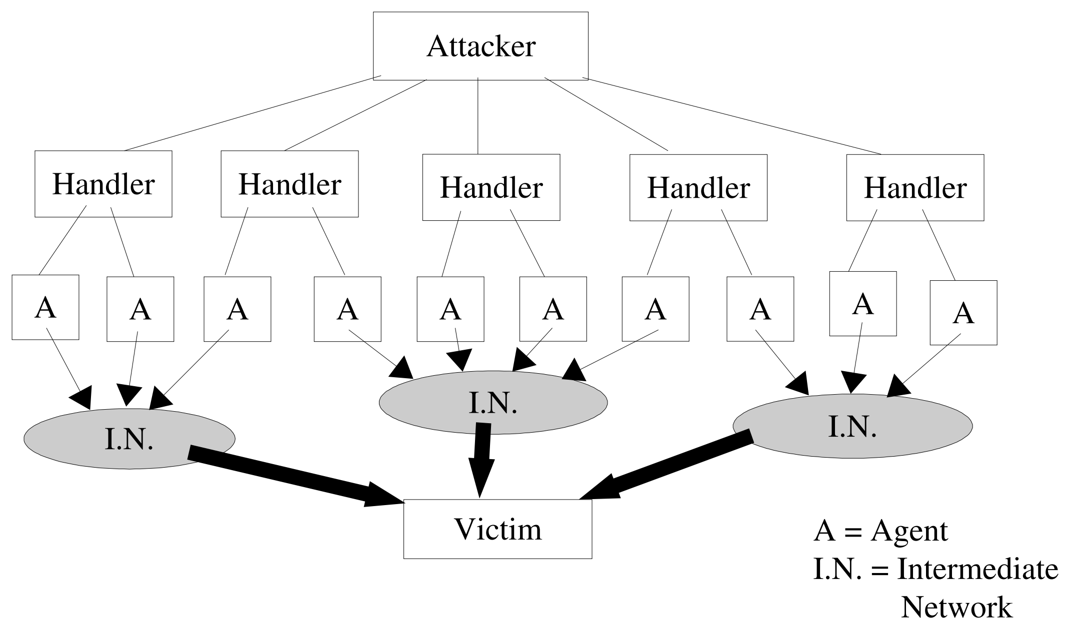


Figure 2.1: Typical DDOS attack architecture.

In Figure 2.1, control communication between the attacker and the layers of the DDOS network is designated by a solid line. Attack traffic is shown with arrows.

Figure 2.1 shows the basic setup and dictatorial nature of the DDOS attack architecture. What Figure 1 does not show adequately is the extremely large number of computers absorbed into the attacker's network. By one analysis, each handler can control up to 1,000 agents [Dit99b].

Types of DDOS Attacks

DDOS attackers use a colorful assortment of protocols and ports in order to achieve the element of surprise and impress investigators with their networking acumen (if only true for the DDOS tool authors, not the "script kiddies" who download their software). Attacks generally fall into one of five categories:

- **SYN Flooding**. The agents send the victim a large number of SYN packets, constituting the first part of the three-way handshake for TCP communication. However, the SYN-ACK reply by the victim is never responded to, forcing the victim to keep memory allocated for this open connection. In the face of the sheer volume of SYN TCP requests, the victim quickly exhausts all memory allotted to opening TCP connections. Legitimate clients are denied service.
- In a variation of SYN Flooding, the SYN packets from the agents simply overwhelm the victim's available bandwidth, rendering the victim unavailable to all traffic.
- **Smurf Attack**. Agents send a flood of ICMP echo request ("ping") packets to the broadcast address of an intermediate network. As mentioned previously, if no safeguards on the intermediary exist, every host sends an echo response to the sender. Because the sender is using the victim's source IP address, all responses from the

intermediate network's hosts are directed to the victim, filling up its network links and knocking it offline.

- **Fraggle Attack**. In a variation of Smurf, agents send datagrams to the UDP echo port 7 of every host in the intermediate network. The victim cannot handle the resulting cascade of echo replies [Sko02] and closes up shop.
- **SYN-ACK Attack** [Gib02]. Agents generate TCP SYN requests to multiple, random servers on the Internet with the victim's IP address as their spoofed source. The huge volume of SYN-ACK replies to the victim website shuts it down.

CHAPTER III

PROPOSED DEFENSES AGAINST DDOS ATTACKS

In this chapter I will examine two different approaches for dealing with the DDOS threat. The first, community prevention, requires anybody with a significant presence on the Internet follow a set of accepted security practices for keeping their networks safe from intruders. If everybody helped out in making their networks more secure, DDOS attacks would cease to exist.

The other defense I will present, termed the Secure Collective Defense (SCOLD), uses new features on standard Internet software programs to allow preferred clients to keep in contact with a victim, even during a DDOS assault.

Although other novel defense strategies exist, such as IP traceback and IDIP [Jel03], due to space limitations this paper will contrast only the simple community prevention technique with the SCOLD defense.

Community Prevention

Except for the SYN flooding attack, which relies on a weakness in the TCP protocol to exhaust the victim's resources, all the attacks use massive flooding to overload the victim's network bandwidth. This is the essence of the DDOS attack: brute force.

How does one guard against such an attack? Your network might have a great intrusion detection system set up, with no vulnerable services running on any open ports (of which only the most necessary are open), and your network administrator could be competent and well-trained, yet you could get a gigantic flood at any time that you could do virtually nothing about. If you do a lot of business online it could muck up your finances, at least for a day.

Widening network bandwidth is one option for keeping an attack from doing any harm. When the thirteen root DNS servers were attacked last year, the attack failed to fill up the large amount of excess bandwidth of the DNS network, and was not successful. However, it is just not economically feasible for most companies to excessively increase their network bandwidth as a solution to a problem they may never face.

As Congress debates on what network security practices should be required for businesses, it is clear that the solution to DDOS does not lie with just one company or entity. Since DDOS attacks harvest the energy of hundreds or thousands of computer systems, it seems some sort of network "neighborhood watch" program could be the answer. Or just like when all the citizens of a small town build a dike together with sand bags in the face of an imminent flood, so should the citizens of the Internet come together and erect a cyberdike to blunt and defeat a DDOS flood.

Whichever way you want to look at it, fighting the DDOS threat requires that everybody follow a set of reasonable networking practices [SANS00]:

- **Egress filtering**. All DDOS assaults have a common thread: spoofed source IP addresses. Usually the source address is that of the victim, but sometimes it is illegal,

e.g., a private address such as 127.0.0.1 or 192.168.0.1. All networks and ISPs should ensure that every packet that leaves a network contains a source IP address consistent with that network. This still leaves open the possibility of spoofing within a network, but of course the perpetrator would be much easier to catch.

- **Ingress filtering**. In addition to checking outbound messages, ISPs should also check the source address of inbound messages to ensure their legitimacy (again, 127.0.0.1 or 192.168.0.1 should not be on the Internet).
- **Broadcast Amplification**. Most DDOS attacks "amplify" the attack by sending packets to the broadcast address of a network, where all hosts reply to the victim. There is a simple remedy to this: Drop all packets destined for the broadcast address! There is rarely any good reason to allow such packets in. If broadcast-directed messages are needed to some function, use a firewall to let in only authorized senders.
- **Computer System Hardening**. Make the computers on a network harder to break into. This is one of the fundamental prerequisites for a DDOS attack. Without an attacker controlling hundreds of zombie systems from which to launch an attack, he is finished before he even gets started. Hardening involves closing unnecessary ports and/or services that are vulnerable to the buffer overflow attack exploit, as well as installing the latest security patches and communicating the risks to employees of opening email attachments (the preferred entrance of viruses and worms).

This set of guidelines makes sense, is achievable, and would not be hard to implement, but it will probably fail to reduce the DDOS threat. Why? Because, unfortunately, the probability of all entities (businesses, universities, individuals, etc.) on the Internet

following such reasonable recommendations is practically zero. With security expenditures being fairly low for most companies, having a compentent administrator implement these safeguards would probably be a luxury.

It will probably take an act of Congress, ever more destructive DDOS attacks, and heavy fines to start waking companies up to the problem.

Secure Collective Defense (SCOLD)

A novel approach to foiling distributed denial of service attacks has been devised by Dr. C. Edward Chow at the University of Colorado at Colorado Springs. This approach, the Secure Collective Defense (SCOLD), utilizes new capabilities in existing software packages to keep communication channels open between clients and a server that has been targeted with a DDOS attack. SCOLD represents a philosophy called "intrusion tolerance" and is a new area of research in network security.

SCOLD works from the assumption that DDOS attacks are going to happen. From the current perspective, this philosophy is realistic, because the Internet is sort of like the "Wild West" and few regulations of its use have been standardized or implemented. Although some states have outlawed the sending of spam, and, as mentioned previously, Congress is hammering out legislation on security for businesses, most hackers can still get away with almost anything. It seems that once guidelines have been established on network security practices then the underground of misfits will rise up again with a new way of attack that had not been thought of before. So the basis of the SCOLD system should remain on solid ground for awhile.

SCOLD Architecture

Before taking a look at how SCOLD works, let's look at another, more detailed diagram of a DDOS attack, involving routers and the domain name system (DNS) servers (Figure 3.1 below). In this illustration, the attacker has already successfully loaded the DDOS tool onto his handler and agent zombies; here, only the agents are shown because they deliver the blow. The routers handle the traffic flow from agents to victim network. The intermediate (amplification) networks are not included here in order to help clarify the essential aspects of the DDOS attack framework.

The DNS servers are included in the picture because of their important position in the architecture. Namely, they return answers to queries from systems in their zone of control. In this instance, they return the address of the victim to a system (malicious or not) immediately before the attack commences. Since the victim's domain is not located in their zone of authority, the DNS server will issue a query to the victim's own DNS server to get this information (unless the address has been cached from a recent query). I will delve more deeply into the role of the DNS server in the next chapter.

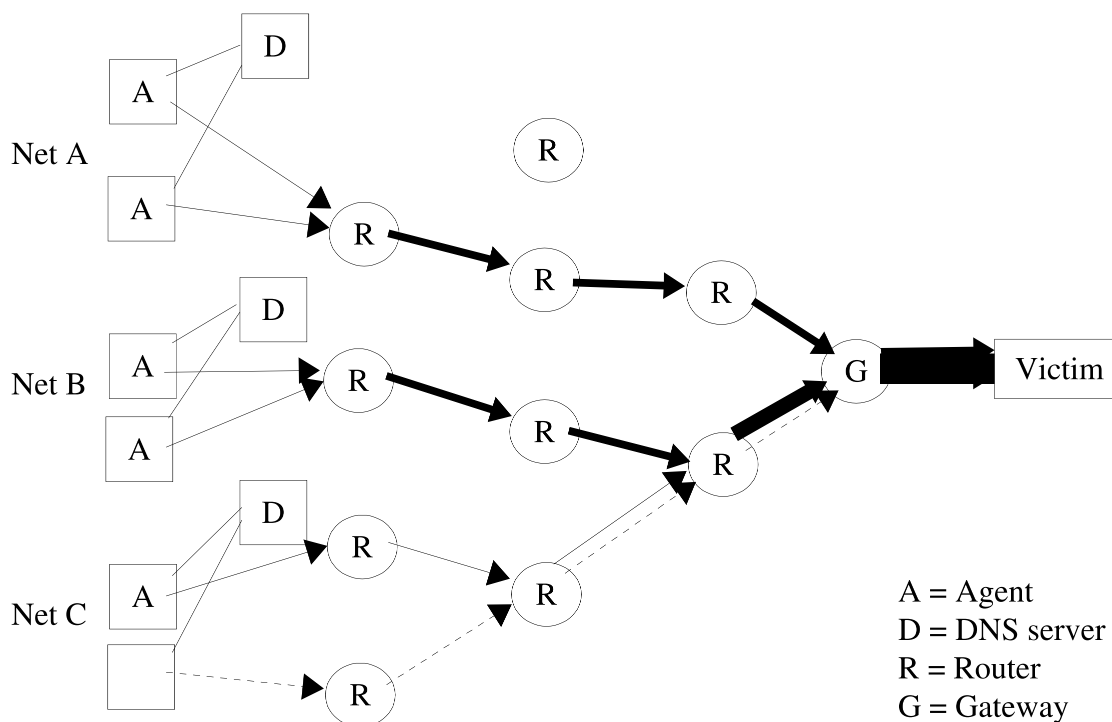


Figure 3.1: Detailed view of DDOS attack.

As with the earlier DDOS diagram (Figure 2.1), attack traffic is designated by an arrow; communication between computers and DNS servers is shown by a solid line. Traffic from a non-agent system to the victim is shown by a dashed line. Net A, Net B, and Net C designate three distinct networks (each having only two computers for simplicity).

As is plainly evident from Figure 2.1, once attack traffic from one network merges with that from another, it uses up more bandwidth of the incoming transmission lines. Arriving at the victim's doorstep, the traffic consumes bandwidth to the limits of the network's capacity, slowing traffic to a crawl. Thus the denial of service attack, distributed across many networks, achieves its purpose.

Now let us consider what happens when the DDOS attack has been interceded and defeated with the SCOLD system. The following is a narrative of an attack in progress

and the steps that SCOLD takes to counter it. Each action described is labeled with a number in parentheses, which corresponds to the same number in parentheses on the diagram shown below (Figure 3.2).

Once an attack is detected, the victim alerts **(1)** the SCOLD "coordinator," which, in turn, alerts **(2)** another machine, "proxy server 1" (PS1). With a list of preferred clients, PS1 sends **(3)** a DNS message to the DNS server that is authoritative for each client on the list. In this example, only the Net C network has a client on this list, so the only message sent is to Net C's DNS server. After mutual authentication, the message instructs the DNS server to create a new zone file on its hard drive for the victim's domain. Also, an entry for this new zone is appended to its configuration file (/etc/named.conf). The DNS server reloads all zones into memory.

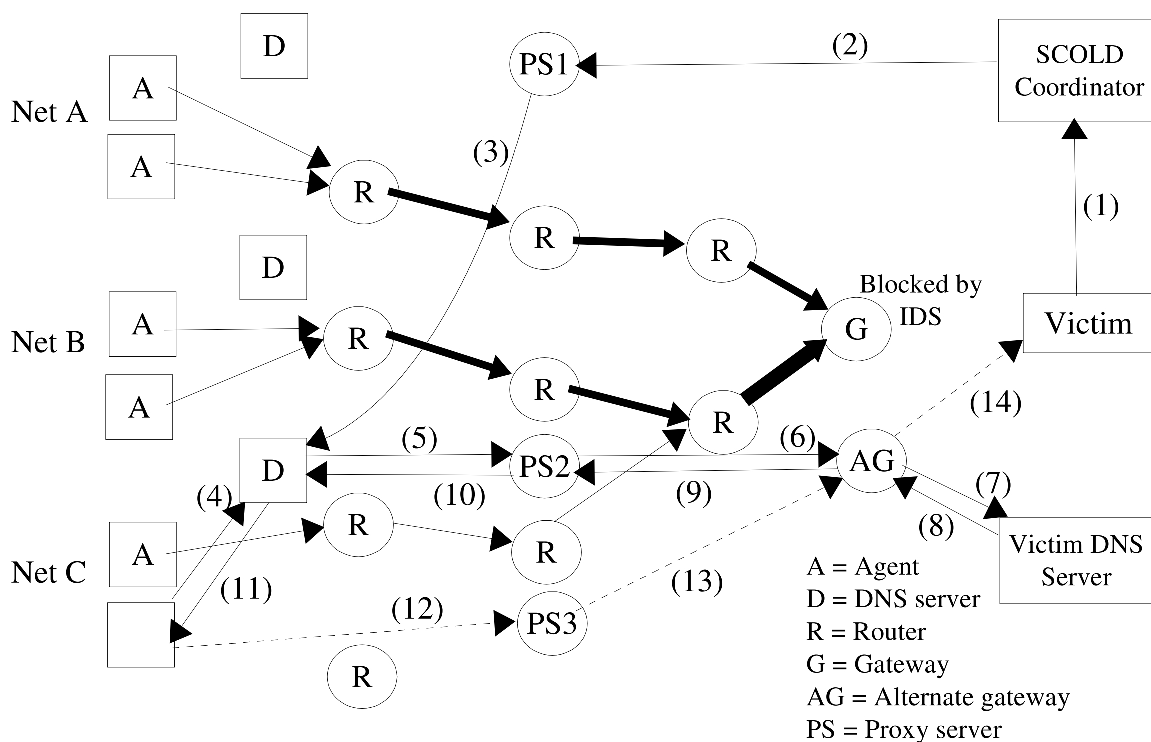


Figure 3.2: SCOLD system foils DDOS attack.

When a client wants to communicate with the victim, he queries **(4)** his authoritative DNS server for the victim's Internet Protocol (IP) address. Note that, if the victim's address resides in the client's memory because of a recent query, the client uses that address and does not query his DNS server. In this scenario, however, the client will not be able to communicate with the victim, who is under attack. The success of this SCOLD strategy hinges upon clients querying their DNS server after the victim's zone has been added. (An alternative SCOLD strategy has proxy server PS1 update the routing entries of the client so that the packets from the client will be sent over an indirect connection.) Because the client DNS server is now technically authoritative for the victim's zone, the client DNS server does not process the query as it normally would.

Instead, the client's DNS server looks in the zone file that was created by the DNS message from proxy server 1. There it finds a nameserver with a type A resource record, which is the IP address of the victim's DNS server. The client DNS server also finds another resource record, an IP address of the new ALT type. The ALT type (referring to an "alternate" address), designates the IP address of a SCOLD-aware proxy server (which we will call PS2).

With the addresses of the victim's DNS server and PS2, the client DNS server sends **(5)** a query to the victim DNS server for the address of the victim host. However, the query is not sent directly to the victim DNS server, because the main gateway is flooded. Instead, the client DNS server sends **(5)** the query to the victim's DNS server through PS2, via IP tunnel. When PS2 gets the packet, it forwards **(6)** it to the victim's secret, secondary gateway, which is not being attacked. This packet is also conveyed via IP tunnel. When it comes to the new gateway the packet is routed **(7)** to the victim in normal fashion. IP tunnel is not needed between gateway and victim because they are connected with the victim's internal network.

Once the victim DNS server receives the query, it replies with the normal list of (type A) IP addresses for the victim. However, the response also includes type ALT addresses in its response. These ALT-type addresses refer to other SCOLD-aware proxy servers that will be able to forward packets from the client to the victim through an undisclosed gateway router.

The return packet follows the same route it originally took, only in reverse. It goes back **(8)** through the alternate gateway, then is funneled via IP tunnel **(9)** through PS2 and

back **(10)** to the client DNS server. The client DNS server returns **(11)** the lookup response to the client application.

The client application selects one of the SCOLD-aware proxy servers from the list of ALT-type addresses in the DNS query result. It sends **(12)** packets to the victim, using a SCOLD-aware proxy server (PS3) as intermediary. PS3 forwards **(13)** the packets to the alternate gateway, which routes **(14)** them on to the victim. SCOLD sets up IP tunnels between client and PS3, and between PS3 and the victim's secret gateway, much like the traffic between client and victim DNS server.

The actions of the SCOLD defensive shield occur silently and unobtrusively, without knowledge of the client application or client user. In other words, the user can access the victim's website without knowing the victim is getting hammered by a DDOS attack.

A few notes on the SCOLD framework. The SCOLD coordinator has two purposes. One is to tell a proxy server to send out the DNS messages to clients, using a new program called *nsreroute*, detailed in chapter 6. The other purpose is to activate the secret gateways and control IP tunnels among the various proxy servers sprinkled throughout the Internet. The coordinator sends the proxy servers a remote command that instructs them to set up one end of an IP tunnel and listen for traffic; the other end of the tunnel is set up either by the client DNS server or the alternate gateway.

The SCOLD-aware proxy servers are owned by different branches of an organization, or different organizations (businesses, universities, etc.) who have agreed to join the SCOLD network.

Intrusion detection systems (IDSs) are attached to the SCOLD proxy servers, as well as to the victim's public gateway. When the attack begins, the IDS at the gateway blocks all incoming traffic. The IDSs stationed at each proxy server begin to ferret out malicious traffic destined for the victim. Because the DDOS attack activates the SCOLD defenses, most of the bad traffic will probably be directed toward the victim's public gateway, but it would be an easy feat for the attacker to every so often re-resolve the victim's IP address during the attack.

If the DDOS agents have infiltrated one of the networks whose DNS server was updated with the nsreroute message, then packets from these agents will traverse the IP tunnels to the proxy servers. However, now the flow of attack traffic is divided significantly, as each proxy server in the SCOLD apparatus is responsible for only a handful of networks. This attenuated stream enables each proxy server IDS to be more accurate in discriminating between good and bad packets.

It is certainly possible that the volume of traffic through one or two proxy servers could be high enough to significantly degrade communication between victim and some clients. However, the SCOLD architecture has essentially pushed the attack back from the edge of the victim's network to the origin of attack, the client networks. In essence, SCOLD employs a "divide and conquer" philosophy in releasing the DDOS chokehold on the victim. So most friendly clients that are on the victim's list should be able to talk to the victim. A machine on a network whose DNS server is not updated with the victim's zone at the outset of attack probably will not be able to get through until after the attack has stopped.

Need for Enhanced DNS

The functionality displayed by the client DNS server in authenticating the PS1 client, creating a new zone file, appending the zone entry to the configuration file, reloading the zone into memory, and carrying out the recursive query for the victim's IP address through an IP tunnel, is completely new. This behavior, including the new DNS message (nsreroute) sent by PS1 to the client DNS server, will be shown, in the context of the SCOLD architecture, to defeat a DDOS attack by keeping open the lines of communication between preferred clientele and the victim. The success of the enhanced capability of the DNS BIND software will be proven through experimental trials.

The steps to add the necessary code to the DNS BIND software to implement the new behavior needed by the SCOLD system will be detailed in upcoming chapters, and is the main focus of my thesis.

CHAPTER IV

DOMAIN NAME SYSTEM

Because my work on the SCOLD project involves adding functionality to the domain name system (DNS) server software, I will present a brief history of DNS and an overview on what DNS servers do. Then I will examine the Berkeley Internet Name Domain (BIND) nsupdate (dynamic update) command, which is similar to one of the new features that I will add to BIND, namely the nsreroute command.

Brief History of DNS

The U.S. Department of Defense's Advanced Research Projects Agency (DARPA) created the first wide-area computer network, the ARPAnet, in the late 1960s to connect their research labs [AL01]. For this small network, each computer stored a file, HOSTS.TXT, on its hard drive that would map the host names of other computers on the network with their Internet Protocol (IP) addresses. This file allowed the computers to find each other on the network and communicate.

However, as ARPAnet expanded and began using the TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite in the early 1980s, the HOSTS.TXT file was found to be inadequate for a variety of reasons. Most importantly, it was too often out of date and could not prevent the occurrence of two machines on the network

with identical host names [AL01]. These inadequacies spurred the development of a new technology, the domain name system.

The domain name system is a database that stores Internet host names and their IP addresses. It is distributed across many machines on the Internet in order to improve efficiency. DNS sprung up in 1984 when Paul Mockepetris (then of the University of Southern California's Information Sciences Institute; now chairman and chief scientist of Nominum, Inc.) wrote two Request for Comments (RFCs, papers published on new developments in Internet-related technology) on the DNS architecture in 1984 [AL01]. Mockepetris superseded these with RFC 1034 and 1035, which define the current specifications of the DNS.

The most popular incarnation of DNS is the Berkeley Internet Name Domain (BIND) server software, originally written by Kevin Dunlap for the Berkeley 4.3 BSD UNIX operating system [AL01]. BIND is currently the de facto standard and is used by most UNIX operating systems as well as Windows NT [AL01]. It is maintained by the Internet Software Consortium (ISC) and can be downloaded from ISC's web site at www.isc.org. The latest version of BIND as of this writing is v. 9.2.2. My research and this paper deal only with the BIND v. 9.2.2 DNS software.

DNS Name Space

The DNS database has an upside-down tree structure, as shown in Figure 4.1. Called the DNS name space, the tree defines all possibilities of host names on the Internet because

any host name can be found by descending from the top to a leaf node. The tree shown in Figure 4.1, of course, represents only a very small fraction of the total DNS namespace.

The root resides at the top and is designated by a dot ("."). Immediately below the root are the top-level domains that divide the Internet into different categories (originally there were only eight, shown in Figure 4.1, but many more have since been added, including domains for countries) [AL01]. For instance, "edu" is for education; "mil" stands for military; and "com" is for commercial.

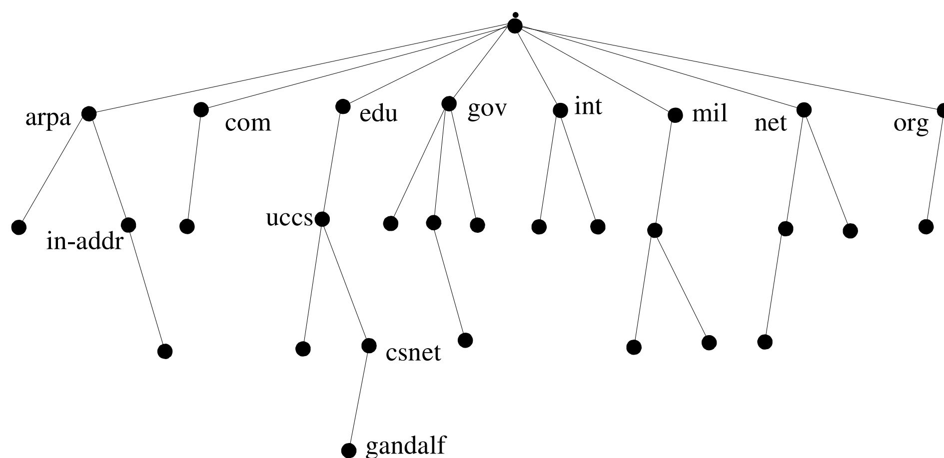


Figure 4.1: Structure of DNS name space.

For example, traversing one possible set of nodes on the tree in Figure 4.1 yields the path: .-->edu-->uccs-->csnet-->gandalf. This represents one host on the Internet with the Fully Qualified Domain Name (FQDN) of gandalf.csnet.uccs.edu. A FQDN means the entire name of the host is specified, including its domain.

The in-addr.arpa domain is special. This is the portion of the DNS database that holds all possible combinations of IP addresses [AL01]. Currently addresses follow the IPv4 format, where each address is 32 bits long, and every eight bits is separated by a dot

(the "dotted quad"). Although the rest of the world is slowly moving toward the IPv6 addressing format, where an IP address is 128 bits long, the United States still has an abundance of unused addresses so will not be switching over to IPv6 anytime soon [Cha03]. (I concentrate only on addresses that follow IPv4 in this paper.)

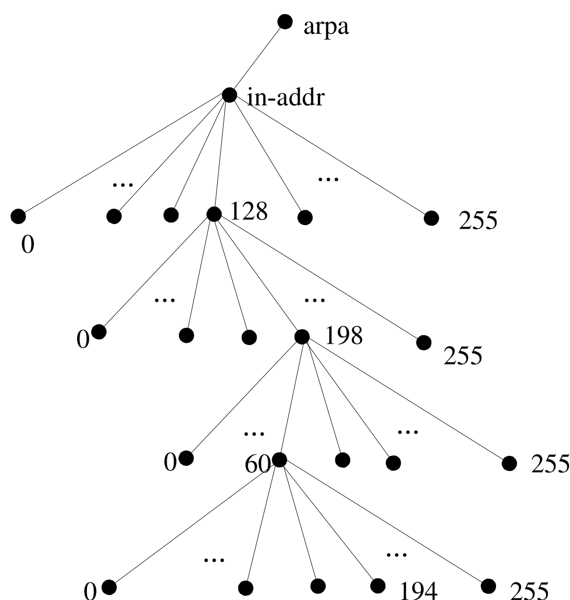


Figure 4.2: IP addresses are located in the in-addr.arpa domain.

Shown in Figure 4.2, arpa-->in-addr-->128-->198-->60-->194 corresponds to IP address 128.198.60.194 and represents one possible traversal down the in-addr.arpa subtree of the DNS name space.

A traversal of the DNS name space tree from top to bottom always starts at the most general and goes to the most specific. For cs.uccs.edu, the most general node is "edu", and the most specific is "gandalf". For 128.198.60.194, the most general node is "128", and the most specific is "194". Notice that the way the host name is written

(specific host followed by domain) is the reverse of the address (domain followed by specific host).

DNS Zones

The DNS server gandalf is authoritative for the csnet.uccs.edu zone, which means gandalf is the DNS server that should be queried to get the most accurate, up-to-date resource records available for a particular host name in the csnet.uccs.edu domain. A resource record refers to a host name, an IP address, the name of a DNS server, or other kind of DNS data type stored in a DNS server.

If another subnet of csnet.uccs.edu exists, say clientnet.csnet.uccs.edu, there might be another DNS server that is authoritative for the clientnet zone. Then gandalf would just refer queries pertaining to hosts on clientnet.csnet.uccs.edu to be answered by the DNS server that has authority for the clientnet subdomain. Referring queries to be answered by a subdomain's DNS server is called "delegation."

For instance, klingon.uccs.edu is the server of authority for the uccs.edu zone. When a machine somewhere on the Internet wants to know the IP address of shavano.uccs.edu, klingon will answer this query directly. However, when a query comes into klingon for the address of gamma.csnet.uccs.edu, klingon cannot answer this because the uccs.edu zone has delegated authority for the csnet.uccs.edu zone to gandalf.csnet.uccs.edu. So klingon responds that gandalf is the server to talk to. When gandalf gets the query, it responds with the resource records it finds in its zone file for the csnet.uccs.edu domain.

DNS Zone Files

When a DNS server is booted up, it loads the contents of all its zone files into memory for fast response to queries. Each zone file is kept in a directory specified in the name server's configuration file (at /etc/named.conf). The BIND name server daemon that handles query requests is called "named" (pronounced as "name-dee" [AL01]).

Zone files all follow a standard format. One example is shown below:

```
$TTL 86400
@    IN   SOA       gandalf.csnet.uccs.edu.dbwilkin.cs.uccs.edu.   (
              1         ; serial
              28800     ; refresh
              7200      ; retry
              604800    ; expire
              86400     ; ttl
              )
              IN   NS   gandalf.csnet.uccs.edu.
ace           IN   A    128.198.60.198
alpha              IN   A    128.198.61.15
beta          IN   A    128.198.61.16
delta              IN   A    128.198.61.18
gamma         IN   A    128.198.61.17
gandalf       IN   A    128.198.60.194
lamar              IN   A    128.198.60.168
narnia             IN   A    128.198.60.19
race          IN   A    128.198.60.178
wait          IN   A    128.198.60.202
```

Figure 4.3: Zone file for csnet.uccs.edu domain.

The zone file in Figure 4.3 contains the following elements. The first line, `$TTL 86400`, refers to a suggested "time to live" value in seconds of the names and addresses in the file. In other words, the entries may only be cached for 86400 seconds (one day) in whatever machine has asked for them. After this time has elapsed, a new query must be sent to the DNS server to get fresh records.

28

The "@" symbol means that for all A-type resource records, unless terminated with

a dot ("."), are part of the domain the zone file refers to. For instance, if the zone file' s associated entry in /etc/named.conf refers to the "csnet.uccs.edu" domain, then the presence of "@" means all host names without a terminating dot should have "csnet.uccs.edu" appended. A presence of a terminating dot means the host name is fully qualified.

"SOA" is a resource record and means "start of authority." It always goes at the top of the zone file. The rest of the SOA record includes the name of the server of authority, an email address (without the "@" symbol), and time values in parentheses. These values are for slave name servers and are not relevant here. All words placed after a semicolon are comments.

The subsequent lines in the zone file are the most important. They define either name servers (designated by the NS type) or IP addresses of host names (designated by the A type). All type A and type NS resource records shown in Figure 4.3 are also of type IN, which refers to the class "Internet." It is the only class we will see in this paper. (Two other classes, the Chaos class and Hesiod class, are obsolete [AL01].)

The NS type designates the host name of an authoritative DNS server, primary or secondary. In Figure 4.3 the DNS server of authority for the csnet.uccs.edu domain is gandalf.csnet.uccs.edu.

All following lines show host names and corresponding addresses. Each line maps one host name to one IP address. If the following line only has an address, then it also maps to the previous host name. Type A resource records always refer to IP addresses.

For example, host "ace" (which has the FQDN of ace.csnet.uccs.edu) has the IP address 128.198.60.198.

named Configuration File

Each zone file in a DNS server must have a corresponding entry in the named configuration file, named.conf, located in the /etc directory. At bootup time the server reads named.conf first to determine what zone files it should load into memory. If a zone file exists in the proper directory, usually /var/named, but does not have a zone statement in named.conf that maps this file to a domain, the file will not be loaded into memory and the DNS server cannot adequately respond to queries for this domain.

The named.conf file used by the DNS server gandalf.csnet.uccs.edu file is shown in Figure 4.4 below:

```
options {
     directory "/var/named";
};
controls {
     inet * allow { any; } keys { "rndc-key"; };
};
include "/etc/rndc.key";
zone "." IN {
     type hint;
     file "db.cache";
};
zone "localhost" IN {
     type master;
     file "db.localhost";
};
zone "0.0.127.in-addr.arpa" IN {
     type master;
     file "db.127.0.0";
};
zone "60.198.128.in-addr.arpa" IN {
     type master;
     file "db.128.198.60";
};
```



```
zone "csnet.uccs.edu" IN {
     type master;
     file "db.csnet.uccs.edu";
     allow-update { 128.198.61.15; };
};
```

Figure 4.4: A named.conf file.

The named.conf file shown above has a few important features. Each statement in named.conf begins with a key word, is followed by information between braces, and is terminated by a semicolon. The first statement, beginning with `options`, defines the directory where zone files are kept. Here the server will look for zone files in the /var/named directory.

The next statement, beginning with `controls`, specifies how the name server listens for control messages [AL01]. With the absence of a specified port, the name server listens on port 953 by default. The rndc-key is the cryptographic key that rndc users must use for authentication [AL01]. rndc is a utility that enables users to locally or remotely control the named daemon process from the command line [AL01].

The `include` statement specifies the directory where the rndc-key is defined [AL01].

The rest of the statements in the file, beginning with the word `zone`, each define a domain that the name server is authoritative for, and the corresponding zone file that maps hosts in the domain to their IP addresses. All zone files, in this example, begin with the two letters, "db". For instance, the zone "." refers to the root domain, and is specified with the file db.cache. (Note: zone file names may be whatever the DNS administrator wishes.) db.cache contains a list of the thirteen root name servers.

The zones "localhost" and "0.0.127.in-addr.arpa" give the DNS server authority to handle queries directed to itself. db.localhost is the zone file that handles queries about

localhost, and db.127.0.0 handles queries about the 127.0.0.1 local loopback IP address. (Queries for a host name for a given IP address are called "pointer queries" [AL01].)

Finally, zones "60.198.128.in-addr.arpa" and "csnet.uccs.edu" handle queries for the 128.198.60.0 network and the csnet.uccs.edu domain, respectively. The zone file for csnet.uccs.edu is called db.csnet.uccs.edu and is described in the previous section.

All zone statements contain the substatement `type master` which means that the DNS server is the primary master name server for that zone.

DNS Dynamic Update (nsupdate)

The last zone statement in Figure 4.4 contains a new substatement, `allow-update`. This substatement enables the machine with IP address 128.198.61.15 (which happens to be alpha.csnet.uccs.edu) send a remote command to the DNS server to add or delete resource records in the csnet.uccs.edu zone file. This feature is called DNS dynamic update, and can be run from a shell with the command "nsupdate."

Invoking nsupdate modifies the zone file on disk, and updates the records cached in memory. Dynamic update cannot entirely delete an existing zone, nor can it create a new zone [AL01]. It can also use secret cryptographic keys to mutually authenticate an updater, as well as public key cryptography to sign zones for a querying machine that wants to make sure the resource records retrieved from the DNS server are authentic.

The following commands illustrate the use of nsupdate to add a host:

```
# nsupdate
> update add viva.csnet.uccs.edu. 86400 A 128.198.60.192
>
```

The first line, nsupdate, tells the machine to accept the arguments on the following linesfor processing nsupdate. (Alternatively, an input file may be specified on the same line after nsupdate instead of entering the update commands manually.) The next line tells the server to add the host viva.csnet.uccs.edu to the csnet.uccs.edu zone, and to map this name to the address 128.198.60.192. The number 86400 is the time-to-live value, and A refers to the datatype of the IP address. The last line in the input must be blank. Entering this blank line tells the system that no more input is forthcoming and to execute nsupdate.

Likewise, nsupdate can be used to delete a host from a zone:

```
# nsupdate
> update delete viva.csnet.uccs.edu.
>
```

Here, all resource records (except the SOA record and one NS record) associated with host viva.csnet.uccs.edu are deleted from the csnet.uccs.edu zone.

Dynamic update can use transaction signatures (TSIG), which are secret keys shared between a server and an updater to ensure the updater is authorized to add or delete records from a server's zone.

Although the nsupdate program has proven invaluable as a starting point in developing a new program for BIND, I used OpenSSL instead of TSIG in the implemention of client-server authentication.

CHAPTER V

ENHANCING DNS BIND FOR SCOLD

In this chapter we discuss the implementation of the new, enhanced features for the DNS BIND software that are needed by the SCOLD anti-DDOS network security system. The preceding chapter concerning important aspects of BIND were presented because these areas are immediately relevant to my project, for the following reasons:

- BIND's nsupdate program is used as a starting point for a new program, nsreroute. This new client program will allow DNS servers that are authoritative for certain clients become authoritative for the domain of a victim under attack.
- The DNS servers that receive this new DNS message will create a new zone file for the victim's domain and load this file into memory.
- As part of adding the victim's domain to their zone of control, the DNS servers will also append the corresponding statement to the named.conf configuration file, so that the update will be persistent after the DNS server reboots.

Other parts of the DNS server software that have been enhanced include:

- BIND server-side code for handling the nsreroute DNS message.
- BIND server-side code for handling queries for victim's domain.

In addition, BIND was augmented with OpenSSL code for client-server authentication.

Understanding Client and Server Components of BIND

For my thesis I worked with the latest version of BIND, version 9.2.2. The base directory is bind-9.2.2. To get an idea of the size of the BIND software package, I created an archive of the files in the bind-9.2.2 directory (using the UNIX "tar" command), leaving out object files (using the "make distclean" command). After compressing the resulting archive (using the gzip command), I ended up with a file with a .tar.gz extension that is over five megabytes in size.

The two most important directories off the base directory bind-9.2.2 are lib and bin. These two directories contain most of the files relevant to client and server which were modified for my thesis. The two most important subdirectories in the lib directory are dns and isc. The dns directory mostly contains files that are concerned with implementing client actions for whatever client needs them. The client, for instance, could be the nsupdate program, or it could be nsreroute, or it could be a query tool, such as dig. Some of the most important client files in bind-9.2.2/lib/dns are message.c (responsible for manipulating DNS messages), rdata.c (manipulates rdata structures), rdataset.c (manipulates rdataset structures), and resolver.c (implements much of the resolver portion of BIND), among many others.

The bind-9.2.2/lib/isc directory is sort of a grab bag of files that do almost anything that is needed by either client or server. For instance, here you will find functions concerned with IP addresses, buffers, memory allocation, error checking, string processing, timers, writing to/deleting from logs, etc.

Off the isc directory lies the pthreads, nothreads, and unix subdirectories. The pthreads directory is utilized if BIND is configured with threads enabled (i.e., entering the –enable-threads option at the command line); otherwise the nothreads directory is relevant. The unix subdirectory contains socket programming code.

Many functions in the isc directory, or any subdirectories of isc, are wrapper functions of standard UNIX functions. The wrapper function calls a UNIX function and returns an ISC-defined error code, depending on the result of the call. For example, isc_socket_create() in the lib/isc/unix directory creates a socket by calling the UNIX function socket(), which returns a file descriptor to a socket. isc_socket_create() returns a value of type isc_result_t, which is an integer that ranges anywhere between 0 and 56. (A successful operation returns 0; otherwise the returned integer refers to one of 56 different kinds of error.)

In the bind-9.2.2/bin directory resides subdirectories for both client programs and server. Here exists the client directories dig, nsupdate, and nsreroute. The named directory implements code for the named server. The check, dnssec, rndc, tests, and win32 subdirectories are relatively unimportant and will not be examined further.

The GNU GDB Debugger and DDD Graphical User Interface

Understanding the guts of nsupdate, nsreroute, and the named server would not have been possible without the GNU GDB debugger. Originally authored by Richard Stallman, GDB is available for free download from the Free Software Foundation at www.gnu.org.

I used GDB with the excellent graphical user interface DDD (the "Data Display Debugger") v. 3.3.1, written by Dorothea Lutkehaus and Andreas Zeller. DDD is also available for free download at www.gnu.org.

With DDD it is easy to trace execution of a program that runs on the command line. Just do the following (in all descriptions of actions taken, "click" means "click on the left mouse button"):

- Start DDD by entering "ddd" on the command line.
- Click on File-->Open Program from the toolbar menu and choose the program you want to trace.
- Set any breakpoints before running the program by double-clicking on the far left side of the line of code you want to stop at. A red stop sign symbol will appear. (A breakpoint is easily cleared by clicking once on the breakpoint stop sign, then clicking on the toolbar menu item "Clear".)
- On the toolbar menu click on Program-->Run to run the program. If arguments should be supplied on the command line, enter them in the box "Run with Arguments". Click on the button "Run" to begin execution.
- DDD defaults to displaying source code in one window, and the results of execution in another window at the bottom. To close the source code window, click on View-->Source Window from the toolbar. To close the window showing execution results, click on View-->GDB Console from the toolbar.
- Another window can be opened in DDD to show variables or function arguments. The following actions can be taken in displaying or undisplaying their contents:

    - Double-clicking on the desired variable or argument will automatically display it in a new window. If the variable or argument is a pointer, then the address it points to must be double-clicked to show the contents of that address. Double-clicking on different fields contained within the variable will display those fields. To remove a data display, right-click on the variable and click on Undisplay.
    - Alternately, clicking on the toolbar menu item Data-->Display Local Variables will display all local variables, and clicking on Data-->Display Arguments will display all current function arguments. Clicking on Data-->Displays will bring up a dialog box with a list of all variables or arguments currently displayed (in the current section of the program or in other sections). To clear an item from this list, click on the item, then click on the menu choice Undisp (with the skull and crossbones picture). The display of the item will disappear.
- At the start of program execution a new menu box appears on the right side of the screen. It contains stepping commands. The most useful of these commands are described below:
    - Step (Step through instructions until reaching a different source line. This command will descend into a function, if possible.)
    - Stepi (execute one instruction only)
    - Next (step through the program, proceeding through function calls)
    - Cont (continue with program execution until end or next breakpoint is encountered)
    - Undo (undo last command)
    - Redo (after an undo, return to current line in program)

    - Interrupt (quit trace)

- Once program execution has terminated, the session (including all breakpoints) can be saved by clicking on the toolbar File-->Save Session As and entering the name of the saved session.
- To exit DDD, click on File-->Exit, or click on the X located in the upper right-hand corner of the DDD titlebar.
- To Retrieve a saved session, click on File-->Open Session; click on the session to open; and click on the button "Open" at the bottom of the dialog box.

Tracing processes already running, like the named server daemon, requires a slightly different method of attack using GDB/DDD:

- Start DDD by entering "ddd" on the command line.
- Open the program to be traced using the same methods as outlined above (either open a new program by clicking on File-->Open Program, or open a previously saved session by clicking on File-->Open Session).
- Click on File-->Attach to Process; click on the process to attach to (it may already be highlighted); then click on the "Attach" button.
- Step through the program using the stepping methods described above. Actions to add/clear breakpoints, display variables and/or arguments, quit tracing, save session, and exit are also the same as above. Notice that the toolbar menu choice Program-->Run is NOT chosen for tracing already running processes.

The GDB debugger can be run from the command line but I found the DDD GUI to be very convenient and easy to learn.

For a long time, maybe seven months or so, after I began working on this project I was unable to figure out how to trace the named daemon. I wasn't sure if I would be able to modify named successfully for the SCOLD project. I absolutely needed to trace it as it processes an incoming DNS message. Just adding printf statements everywhere (which I tried) doesn't cut it because the code is so big, and named prints these error messages repeatedly (since it essentially runs in a big loop as it waits for the next query).

Even understanding how to trace named as it starts up, initializing data structures, loading zone files into memory, etc., was a little difficult for awhile because early on the named process forks. I could trace it to the fork, and then it terminated. I solved this by reading the named man page, where it states that invoking named with the -f option runs named in the foreground, which means named is not "daemonized," or forked.

One day I had an epiphany and realized that tracing named as it handles DNS messages could only be accomplished by not running named explicitly inside the GDB, but by somehow attaching to the already running daemon. Reading the GDB user's manual ([SPS02]) and following the steps outlined above confirmed my theory. It wasn't until that moment that I realized I could successfully complete this project.

CHAPTER VI

NEW CLIENT PROGRAM: NSREROUTE

In this chapter I describe the steps I took in creating a new client-side program, nsreroute. After presenting an overview of what this program does, I will show how it runs by showing what important functions are executed (trying to show all the functions that it executes would be virtually impossible). I will try to describe all additions made to the BIND code as accurately and thoroughly as possible. Important data structures will be illustrated, and code implementing new functions will be included. The implementation of the new ALT rdatatype is also discussed.

The chapter will also show the implementation of a secure socket layer using OpenSSL that authenticates the server.

Overview

The Secure Collective Defense (SCOLD), as described in chapter 3, requires that the DNS BIND software process a new kind of command, one that will enable clients to communicate with the victim of a distributed denial of service (DDOS) attack. This new command is sent in a DNS message from a proxy server in the SCOLD network to a number of DNS servers that are authoritative for a list of clients.

Specifically, the new, enhanced BIND server software does the following:

- Authenticates sender of message. This means the receiving server checks the sender's certificate against a list of authorized senders. If successful, the server processes the message, which means it does the following:
  - Creates a zone file for the victim's domain. This zone file maps two of the victim's DNS servers to type A and type ALT resource records (RRs). The type A records are the IP addresses of the victim DNS servers. The type ALT records are also IP addresses. They refer to one or more proxy servers that are configured to reroute traffic to one of the victim's DNS servers through an alternate gateway.
  - Adds a zone statement to the receiving server's named configuration file (/etc/named.conf).
  - Reloads all zones into memory.
- Subsequent queries by the client for a host in the victim's domain will be sent to one of the two victim's DNS servers for resolution. While the victim is being attacked, this query is sent through an IP tunnel to one of the proxy DNS servers (with an address given by the ALT record). The proxy server forwards the traffic through a secret gateway to the victim's DNS server. Replies from the victim DNS server return through the same route.

To run the nsreroute program, the proxy server invokes it through a routine. This routine executes the following command

```
nsreroute input_file
```

"input_file" is a file that contains one or more commands, with the format shown in the figure below:

reroute client.clientnet1.com. victimDNSserver1.victimnet.com. victimDNSserver2.victimnet.com. <victim DNS 1 address> <victimDNS 2 address> <proxy server 1 address> <proxy server 2 address> ... <proxy server N address>
reroute client.clientnet2.com. victimDNSserver1.victimnet.com. victimDNSserver2.victimnet.com. <victim DNS 1 address> <victimDNS 2 address> <proxy server 1 address> <proxy server 2 address> ... <proxy server N address>
.
.
.
reroute client.clientnetX.com. victimDNSserver1.victimnet.com. victimDNSserver2.victimnet.com. <victim DNS 1 address> <victimDNS 2 address> <proxy server 1 address> <proxy server 2 address> ... <proxy server N address>

Figure 6.1: Example of input file of commands for nsreroute program.

Each command begins with the word "reroute", followed by the name of a client in a list. The next two elements are always two of the victim's authoritative DNS servers. These are followed by each server's respective IP address. The last items are all type ALT-type addresses for the various cooperating proxy servers in the SCOLD network. To simplify our implementation, each command can consist of only two type A addresses and anywhere from one to ten type ALT addresses. A carriage return separates commands.

The only limitation on the number of messages that can be sent out is the number of child processes that the parent process can create in the nsreroute program. For each client in a unique domain, the nsreroute program sends the message to each DNS server that is authoritative for that domain. These name servers are designated by type NS records in the primary master name server's zone file. A query over the Internet for NS records for a particular domain generally yields between two to four name servers. So between two to four messages, more or less, is sent out for each unique domain in the list of clients.

Multiple clients in the same domain should be weeded out from the input file to optimize the program's efficiency.

Refer to Figure 3.2 to see an example of a DNS server running the enhanced BIND software as part of a SCOLD response to a DDOS attack.

Execution Flow of nsreroute

In understanding how nsreroute works, it is useful to compare it with the the dynamic update command. Although much of the functionality of nsreroute is derived from nsupdate, there are important differences:

- Commands from input are processed differently.
- nsupdate seeks to add or delete to pre-existing zone files, whereas nsreroute proposes to add an entirely new zone file.
- nsreroute sends messages to all authoritative servers for a zone, both primary and secondary; nsupdate only sends the update to the primary master name server, and relies on zone transfers to the slave name servers to propagate changes in the zones. The reason nsreroute differs here is because zone transfers may not occur immediately. Thus, just sending the reroute message to the primary master DNS server would leave many of the client zone's DNS servers without the victim's new reroute zone, thereby depriving clients in the zone from communicating with the victim until the zone transfer takes place.
- To increase speed and efficiency, nsreroute creates a new process for each DNS server, whereas there is no forking in nsupdate.

- nsreroute employs OpenSSL and public key cryptography for authentication. This is totally different than what nsupdate does. Dynamic update may (or may not) use transaction signatures (TSIG) for identity verification. And TSIG uses symmetric key cryptography, or shared secret keys, not public keys and certificates.

In this example of nsreroute, the input file ("input_file") to be processed contains a single command line, for simplicity. This will add a new zone file in the client's authoritative DNS server for the victim's domain. We assume here that the client's DNS server successfully authenticates the sending machine. Also, a zone statement for this new zone will be added to the named configuration file. The input file looks like:

> reroute client.clientnet.com. victimdns1.victimnet.com. victimdns2.victimnet.com. 133.41.96.7 133.41.96.8 203.55.5.10 222.4.11.81 221.16.25.3

The major functions that nsreroute uses in executing this line of input are illustrated in the figure below (Figure 6.2). (Note: Functions are terminated by "()". Indentations represent the function above calling the function below. Words in **bold** explain the action that is carried out. Most of the functions listed reside in bin/nsreroute/nsreroute.c.)

```
main()
     isc_app_run()
          isc_taskmgr_dispatch()
               dispatch()
                    getinput()
                         user_interaction()
                              get_next_command()
                                   reroute_parse()
                                        Parse command in input file; put
into
                                        DNS reroute message
("reroutemsg")
                    start_reroute()
                         Create another DNS message ("soaquery"); this
is a
                         recursive query to find servers of authority
for
                         client.clientnet.com
                         Send soaquery
```



```
               dispatch()
```

```
                    recvns()
                         Receive query reply with authoritative servers
for
                         client.clientnet.com

                         For each server, retrieve its address (embedded
                         in reply), call fork() and do send_reroute()

                         /* Begin child process */

                         send_reroute()
                              Wait until parent process dies (i.e., all
DNS servers
                              for all clients in input file have been
retrieved and
                              parent has forked for each one)

                              Add authoritative zone name clientnet.com
to
                              Question section of reroutemsg

                              dns_request_createvia()
                                   Establish TCP connection with DNS
server at
                                   address found in recvns()
                              dns_request_verify_server()
                                   request-->must_verify_server =
ISC_TRUE;
               dispatch()
                    req_connected()
                         req_send()
                              Send reroutemsg through TCP connection
                              established in dns_request_createvia()
               dispatch()
                    req_senddone()
                         if (request-->must_verify_server) {
                              request-->must_verify_server = ISC_FALSE;
                              ssl_authenticate_server(request);
                                   Create new TCP connection with server
at
                                   port 5300; verify its identity using
SSL
                              exit(0);
                         }
                         /* End child process */
               dispatch()
                    reroute_completed()
                         Free memory; destroy data structures
               dispatch()
                    getinput()
                         user_interaction()
                              get_next_command()
                                   No more input; return STATUS_QUIT
                         isc_app_shutdown()
                              Kill nsreroute process
```

Figure 6.2: Execution flow of nsreroute program with one command line of input.

Important actions of Figure 6.2 can be summarized as follows:

- Parse command in input file; insert command into DNS message ("reroutemsg").

- Create another DNS message ("soaquery") for finding the authoritative servers for the client client.clientnet.com. Send soaquery using UDP.
- Receive reply that contains authoritative DNS servers. For each server in reply, find IP address and fork. Wait until parent process finds no more input and is terminated.
- For each child process, insert authoritative zone into Question section of reroutemsg. Establish TCP connection with authoritative server.
- Set must_verify_server field in request structure to true.
- Send reroutemsg to authoritative server over established TCP connection.
- Open up a new TCP connection with the server on the server's port #5300, and exchange certificates over a secure socket layer (SSL) so that the server can verify the sender's identity and to make sure the sender is authorized to create a new zone for the victim's domain on the server.
- After authentication, the SSL connection is closed and the child process terminates.
- The parent process continues, freeing used memory and looking for the next command line in input. In this example, there are no more commands.
- The nsreroute process terminates.

reroutemsg Data Structure

For the domain name system protocol, all messages between two DNS servers follow the same format [Moc87], shown in Figure 6.3:

Header
Question
Answers

Authority
Additional

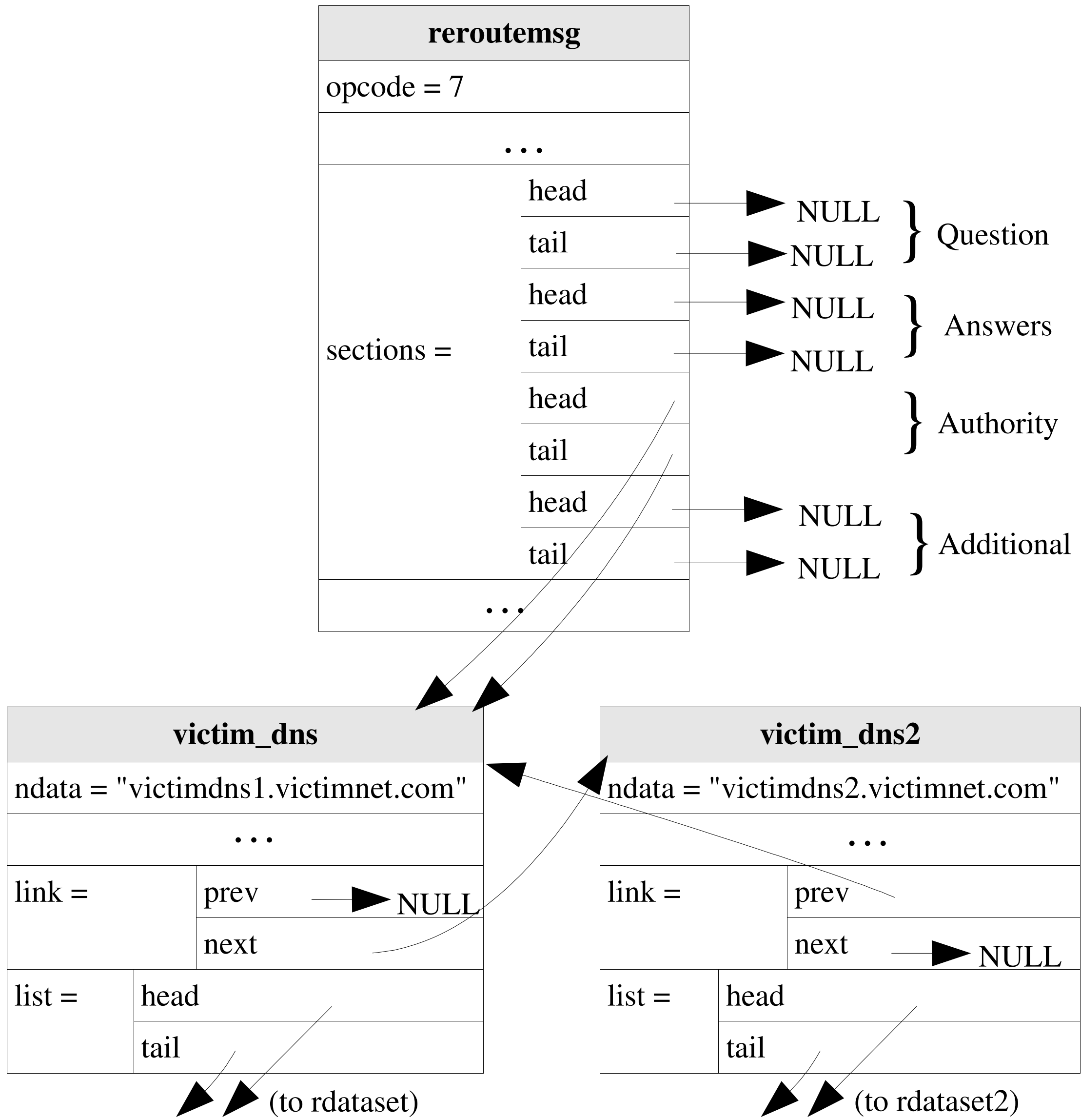
Figure 6.3: DNS Message Format

The header section is 12 bytes long [Ste94]. It contains a number of important values, such as the number of elements in the other four sections, and control flags. The control flags indicate the nature of the message (query or response); the opcode (operation code, a four-bit field indicating DNS query, DNS notify, status, update, and the new reroute); whether the answer is authoritative; whether recursion is desired; whether recursion is available; and other fields [Ste94].

The Question, Answers, Authority, and Additional sections may or may not contain resource records (RRs). It depends on the type of message. For instance, for queries (the most common type of DNS message), the Question section includes the host name, which is the object of a query. The Answers section contains one or more RRs that answer the query. The Authority section contains zero or more DNS name servers that are authoritative for the hosts in the Answer section. The Additional section contains more RRs that relate to the query [Moc87]. Often this section will contain addresses for the name servers in the Authority section. If the Authority section is blank, then typically this section is also. The DNS messages used by nsupdate and nsreroute programs utilize the Question, Answer, Authority, and Additional sections differently than the query type.

Figure 6.4 below shows how reroutemsg is constructed, along with its accompanying subordinate structures victim_dns, victim_dns2, rdataset, rdataset2,

rdataset_proxy, rdataset_proxy2, rdatalist, rdatalist2, rdatalist_proxy, rdatalist_proxy2, rdata_victim, rdata_victim2, rdata_proxy[] and rdata_proxy2[].



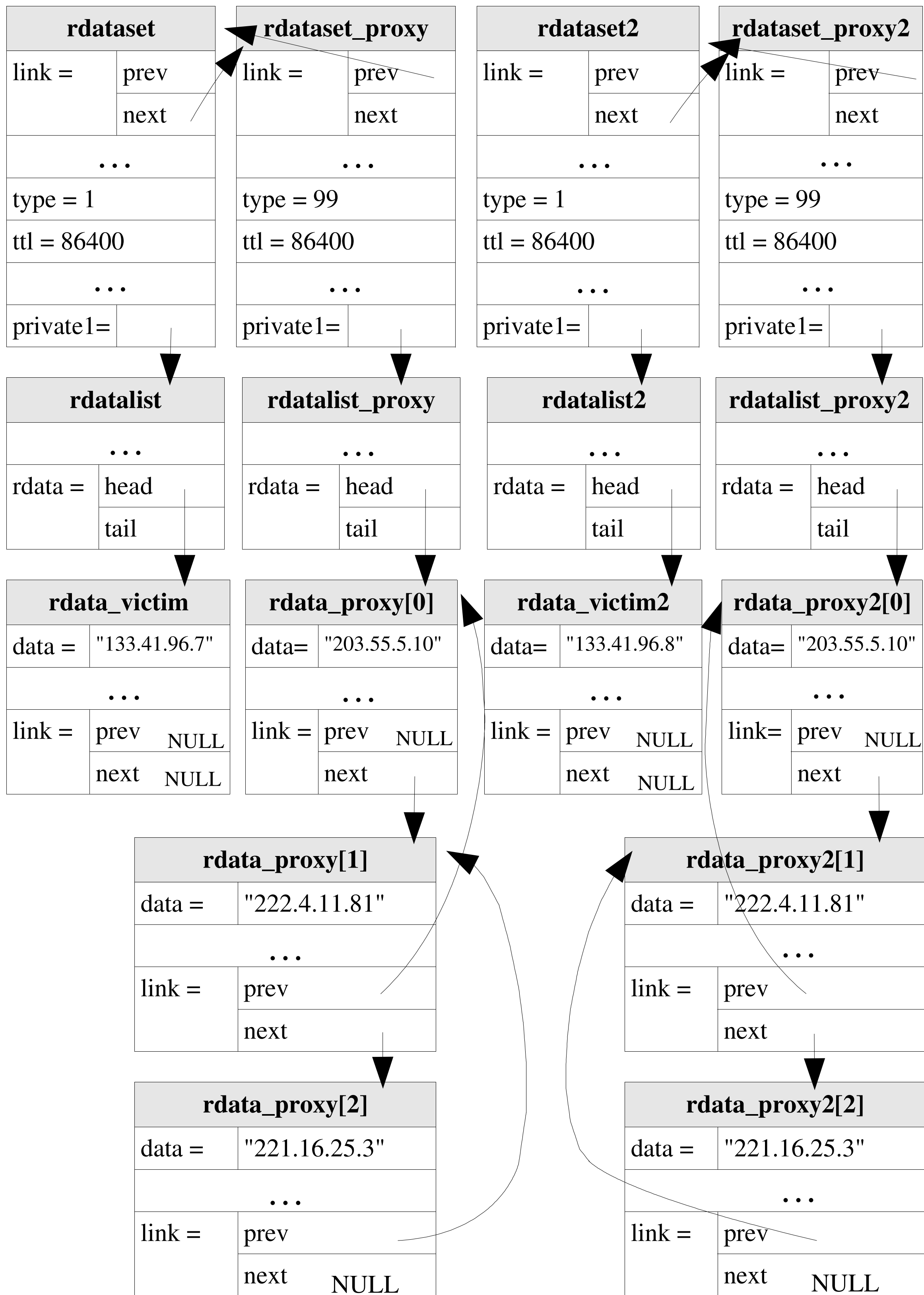


Figure 6.4: DNS message reroutemsg, including lower data structures.

Code for nsreroute

Included below is important code in the nsreroute.c file (partially derived from nsupdate.c, written by the Internet Software Consortium), in which resides most of the functionality for the nsreroute program.

The first bit of code is the function *reroute_parse*, which reads a line of command from input and stores it in the reroutemsg, as graphically depicted in the previous section.

```
static isc_uint16_t
reroute_parse(char *cmdline) {
     isc_result_t result;
     dns_name_t *victim_dns = NULL;
     dns_name_t *victim_dns2 = NULL;
          unsigned long ttl = 86400;  // one day
          char *address = NULL;
     int index;
     int iteration;
          dns_rdataclass_t rdataclass = 1;
          dns_rdatatype_t rdatatype_victim = 1;
     dns_rdatatype_t rdatatype_proxy = 99;
     dns_rdatalist_t *rdatalist_proxy = NULL;
     dns_rdatalist_t *rdatalist_proxy2 = NULL;
     dns_rdataset_t *rdataset_proxy = NULL;
     dns_rdataset_t *rdataset_proxy2 = NULL;
          dns_rdata_t *rdata_victim = NULL;
     dns_rdata_t *rdata_victim2 = NULL;
     dns_rdata_t *rdata_proxy[10];
     dns_rdata_t *rdata_proxy2[10];
          dns_rdatalist_t *rdatalist = NULL;
     dns_rdatalist_t *rdatalist2 = NULL;
          dns_rdataset_t *rdataset = NULL;
     dns_rdataset_t *rdataset2 = NULL;
          isc_uint16_t retval;

     ddebug("reroute_parse()");

     /*
      *   Read the client's host name and store in client_hostname.
      *   client_hostname is a global variable because it is used
      *   in start_reroute() to get the client's domain name that goes
      *   into the Question section of soaquery.
      */
     retval = parse_name(&cmdline, reroutemsg, &client_hostname);
     if (retval != STATUS_MORE)
```



```
          return (retval);

     /*
      *   Initialize array of proxy server rdata for both DNS servers;
      *   max of 10 proxy servers.
      */
     for (index = 0; index < 10; index++)
          rdata_proxy[index] = NULL;
```

```
     for (index = 0; index < 10; index++)
          rdata_proxy2[index] = NULL;
     /*
      *   Read the first victim's DNS server and store in victim_dns
      */
     retval = parse_name(&cmdline, reroutemsg, &victim_dns);
          if (retval != STATUS_MORE)
          goto failure;
          /*
                *   Read the second victim's DNS server and store in
victim_dns2
                */
          retval = parse_name(&cmdline, reroutemsg, &victim_dns2);
          if (retval != STATUS_MORE)
          goto failure;

     index = 0;       /*  re-initialize  */
     iteration = 0;
     address = nsu_strsep(&cmdline, " \t\r\n");
     while (cmdline != NULL && iteration < 12) {
          if (iteration == 0) {

               /*    For the first iteration, create a new rdata
structure to store the
                *    victim's first DNS server IP address in
                */
               result = dns_message_gettemprdata(reroutemsg,
&rdata_victim);
               check_result(result, "dns_message_gettemprdata");
               rdata_victim->rdclass = 0;
                    rdata_victim->type = 0;
                    rdata_victim->data = NULL;
                    rdata_victim->length = 0;
                    retval=parse_rdata(&address, rdataclass,
rdatatype_victim,
                              reroutemsg, rdata_victim);
               if (retval != STATUS_MORE)
                    goto failure;

               if ((rdata_victim->flags & DNS_RDATA_REROUTE) != 0) {
                                   fprintf(stderr, "could not read
rdata\n");
                                   goto failure;
               }
               /*    Create the rdatalist and rdataset that will attach
rdata_victim to the
                *    victim's DNS server, and attach the whole linked
list to reroutemsg,
                *    under the DNS_SECTION_REROUTE section.
                */
```



```
                         result = dns_message_gettemprdatalist
(reroutemsg, &rdatalist);
                         check_result(result,
"dns_message_gettemprdatalist");
                         result = dns_message_gettemprdataset
(reroutemsg, &rdataset);
                         check_result(result,
"dns_message_gettemprdataset");
                    dns_rdatalist_init(rdatalist);
                         rdatalist->type = rdatatype_victim;
                         rdatalist->rdclass = rdataclass;
                         rdatalist->covers = rdatatype_victim;
                         rdatalist->ttl = (dns_ttl_t)ttl;
```

```
              ISC_LIST_INIT(rdatalist->rdata);
                   dns_rdataset_init(rdataset);
                        dns_rdatalist_tordataset(rdatalist, rdataset);
                             ISC_LIST_INIT(victim_dns->list);
                             ISC_LIST_APPEND(victim_dns->list, rdataset,
link);
                   ISC_LIST_APPEND(rdatalist->rdata, rdata_victim,
link);
         } else if (iteration == 1) {

              /* Store the address for the 2nd DNS server in
rdata_victim2,
               * and create the 2nd rdatalist and rdataset that will
attach
               * rdata_victim2 to this DNS server.  Attach the entire
linked
               * list to the first DNS server, through the name
structure's
               * link.  Both DNS servers and their addresses will be
linked
               * to reroutemsg.
               */
                        result = dns_message_gettemprdata(reroutemsg,
&rdata_victim2);
                             check_result(result,
"dns_message_gettemprdata");
                             rdata_victim2->rdclass = 0;
                             rdata_victim2->type = 0;
                             rdata_victim2->data = NULL;
                             rdata_victim2->length = 0;
                             retval=parse_rdata(&address, rdataclass,
rdatatype_victim,
                                reroutemsg, rdata_victim2);
              if (retval != STATUS_MORE)
                             goto failure;
                             if ((rdata_victim2->flags &
DNS_RDATA_REROUTE) != 0) {
                                  fprintf(stderr, "could not read
rdata\n");
                              goto failure;
              }
                        result = dns_message_gettemprdatalist
(reroutemsg, &rdatalist2);
                        check_result(result,
"dns_message_gettemprdatalist");
                             result = dns_message_gettemprdataset
(reroutemsg, &rdataset2);
                             check_result(result,
"dns_message_gettemprdataset");
                             dns_rdatalist_init(rdatalist2);
                             rdatalist2->type = rdatatype_victim;
                             rdatalist2->rdclass = rdataclass;
                             rdatalist2->covers = rdatatype_victim;
                             rdatalist2->ttl = (dns_ttl_t)ttl;
                             ISC_LIST_INIT(rdatalist2->rdata);
                             dns_rdataset_init(rdataset2);
                             dns_rdatalist_tordataset(rdatalist2,
rdataset2);
                             ISC_LIST_INIT(victim_dns2->list);
                             ISC_LIST_APPEND(victim_dns2->list,
rdataset2, link);
```



```
         ISC_LIST_APPEND(rdatalist2->rdata, rdata_victim2, link);
         } else {
                   if (iteration == 2) {
                   /*    All the rest of the IP addresses are for proxy
```

```
servers.
                    *    Create rdataset_proxy that will link to both
rdataset
                    *    and rdataset2.  Create rdatalist_proxy that
will attach to
                    *    rdataset_proxy.
                    */
                           result = dns_message_gettemprdatalist
(reroutemsg, &rdatalist_proxy);
                   check_result(result,
"dns_message_gettemprdatalist");
                                result = dns_message_gettemprdatalist
(reroutemsg, &rdatalist_proxy2);
                   check_result(result,
"dns_message_gettemprdatalist");
                   result = dns_message_gettemprdataset(reroutemsg,
&rdataset_proxy);
                   check_result(result, "dns_message_gettemprdataset");
                                result = dns_message_gettemprdataset
(reroutemsg, &rdataset_proxy2);
                   check_result(result, "dns_message_gettemprdataset");
                   dns_rdatalist_init(rdatalist_proxy);
                   dns_rdatalist_init(rdatalist_proxy2);
                   rdatalist_proxy->type = rdatatype_proxy;
                   rdatalist_proxy->rdclass = rdataclass;
                   rdatalist_proxy->covers = 1;
                   rdatalist_proxy->ttl = (dns_ttl_t)ttl;
                   rdatalist_proxy2->type = rdatatype_proxy;
                   rdatalist_proxy2->rdclass = rdataclass;
                                  rdatalist_proxy2->covers = 1;
                                  rdatalist_proxy2->ttl =
(dns_ttl_t)ttl;
                                  ISC_LIST_INIT(rdatalist_proxy-
>rdata);
                   ISC_LIST_INIT(rdatalist_proxy2->rdata);
                                dns_rdataset_init(rdataset_proxy);
                   dns_rdataset_init(rdataset_proxy2);
                   rdataset_proxy->type = rdatatype_proxy;
                   rdataset_proxy2->type = rdatatype_proxy;
                   dns_rdatalist_tordataset(rdatalist_proxy,
rdataset_proxy);
                   dns_rdatalist_tordataset(rdatalist_proxy2,
rdataset_proxy2);
                   rdataset->link.next = rdataset_proxy;
                   rdataset->link.prev = NULL;
                   rdataset_proxy->link.prev = rdataset;
                   rdataset_proxy->link.next = NULL;
                   rdataset2->link.next = rdataset_proxy2;
                                  rdataset2->link.prev = NULL;
                                  rdataset_proxy2->link.prev =
rdataset2;
                                  rdataset_proxy2->link.next =
NULL;
              }

              /*    Create a new rdata structure for each proxy IP
address.  Store the
               *    address.  Attach to the list of rdata_proxy's,
headed by
               *    rdatalist_proxy or rdatalist_proxy2.
               */
              result = dns_message_gettemprdata(reroutemsg,
&rdata_proxy[index]);
              check_result(result, "dns_message_gettemprdata");
              result = dns_message_gettemprdata(reroutemsg,
&rdata_proxy2[index]);
              check_result(result, "dns_message_gettemprdata");
```

```
                    rdata_proxy[index]->rdclass = 0;
                    rdata_proxy[index]->type = 0;
                    rdata_proxy[index]->data = NULL;
                    rdata_proxy[index]->length = 0;
                         rdata_proxy2[index]->rdclass = 0;
                         rdata_proxy2[index]->type = 0;
                         rdata_proxy2[index]->data = NULL;
                         rdata_proxy2[index]->length = 0;

               retval = parse_rdata(&address, rdataclass,
rdatatype_proxy,
                              reroutemsg, rdata_proxy[index]);
               if (retval != STATUS_MORE)
                                   goto failure;
               if ((rdata_proxy[index]->flags & DNS_RDATA_REROUTE) != 0)
{
                                   fprintf(stderr, "could not read
rdata\n");
                                   goto failure;
                    }
                    ISC_LIST_APPEND(rdatalist_proxy->rdata, rdata_proxy
[index], link);
               dns_rdata_clone(rdata_proxy[index], rdata_proxy2[index]);
               ISC_LIST_APPEND(rdatalist_proxy2->rdata,
                              rdata_proxy2[index], link);
               index++;
          }
               address = nsu_strsep(&cmdline, " \t\r\n");
          iteration++;
     }

     if (cmdline != NULL && iteration >= 12) {
          fprintf(stderr, "too many arguments in command line\n");
          goto failure;
     }
     else if (cmdline == NULL && iteration <= 2) {
          fprintf(stderr, "too few arguments in command line\n");
          goto failure;
     }

     /* Attach first victim DNS server to message, attach second DNS
server to first, and return */
     dns_message_addname(reroutemsg, victim_dns, DNS_SECTION_REROUTE);
     victim_dns->link.next = victim_dns2;
          victim_dns->link.prev = NULL;
          victim_dns2->link.next = NULL;
          victim_dns2->link.prev = victim_dns;

     return (STATUS_SEND);
failure:
     /* clean up */
     index = 0;
     while (index < 10 && rdata_proxy[index] != NULL) {
                    dns_message_puttemprdata(reroutemsg, &rdata_proxy
[index]);
          index++;
     }
     index = 0;
```



```
          while (index < 10 && rdata_proxy2[index] != NULL) {
                    dns_message_puttemprdata(reroutemsg, &rdata_proxy2
[index]);
```

```
                    index++;
          }
     if (rdata_victim != NULL)
          dns_message_puttemprdata(reroutemsg, &rdata_victim);
          if (rdata_victim2 != NULL)
                    dns_message_puttemprdata(reroutemsg,
&rdata_victim2);
          if (rdatalist != NULL)
                    dns_message_puttemprdatalist(reroutemsg,
&rdatalist);
     if (rdatalist2 != NULL)
                    dns_message_puttemprdatalist(reroutemsg,
&rdatalist2);
          if (rdatalist_proxy != NULL)
                    dns_message_puttemprdatalist(reroutemsg,
&rdatalist_proxy);
     if (rdatalist_proxy2 != NULL)
                    dns_message_puttemprdatalist(reroutemsg,
&rdatalist_proxy2);
     if (rdataset != NULL) {
          dns_rdataset_disassociate(rdataset);
                    dns_message_puttemprdataset(reroutemsg, &rdataset);
     }
     if (rdataset2 != NULL) {
                    dns_rdataset_disassociate(rdataset2);
               dns_message_puttemprdataset(reroutemsg, &rdataset2);
     }
          if (rdataset_proxy != NULL) {
          dns_rdataset_disassociate(rdataset_proxy);
               dns_message_puttemprdataset(reroutemsg, &rdataset_proxy);
     }
     if (rdataset_proxy2 != NULL) {
          dns_rdataset_disassociate(rdataset_proxy2);
               dns_message_puttemprdataset(reroutemsg,
&rdataset_proxy2);
     }
     if (victim_dns != NULL)
                    dns_message_puttempname(reroutemsg, &victim_dns);
     if (victim_dns2 != NULL)
                    dns_message_puttempname(reroutemsg, &victim_dns2);
     if (client_hostname != NULL)
                    dns_message_puttempname(reroutemsg,
&client_hostname);

     return (STATUS_SYNTAX);
}
```

Figure 6.5: Function *reroute_parse* in nsreroute.c

The next piece of important code comes from the function *start_reroute*, in nsreroute.c. This code separates the client from his domain name. This domain name is inserted into

the Question section of soaquery. The soaquery message is sent to this domain, asking for all resource records of type NS (authoritative DNS servers).

```
.
.
.
/* Get domain name of client_hostname */
result = dns_message_gettempname(soaquery, &client_domainname);
check_result(result, "dns_message_gettempname");
result = isc_buffer_allocate(soaquery->mctx, &client_domain_buf, len);
check_result(result, "isc_buffer_allocate");
dns_name_setbuffer(client_domainname, client_domain_buf);
labels = dns_name_countlabels(client_hostname);
result = dns_name_splitatdepth(client_hostname, labels-1, NULL, client_domainname);
check_result(result, "dns_name_splitatdepth");

/* Destroy client_hostname */
dns_message_puttempname(reroutemsg, &client_hostname);

/* Put client's domainname into Question section and send */
dns_name_init(name, NULL);
dns_name_clone(client_domainname, name);
ISC_LIST_INIT(name->list);
ISC_LIST_APPEND(name->list, rdataset, link);
dns_message_addname(soaquery, name, DNS_SECTION_QUESTION);

if (userserver != NULL)
        sendrequest(localaddr, userserver, soaquery, &request);
else {
        ns_inuse = 0;
        sendrequest(localaddr, &servers[ns_inuse], soaquery, &request);
}

isc_buffer_free(&client_domain_buf);
dns_message_puttempname(soaquery, &client_domainname);
.
.
.
```

Figure 6.6: Portion of code from *start_reroute* in nsreroute.c.
Lines in **bold** highlight the most important function calls.

The next code snippet comes from the function *recvns.* In this function, for each DNS server that is found in the Answer section of the message (rcvmsg) received from the

client, its IP address is located and a child process is created to deal with sending this

message. The parent process will continue looking for other DNS servers in rcvmsg and forking when the next one is found. When no more servers are found in rcvmsg the parent process goes to the next line of input. Only when the parent processes all lines of input and terminates, will all the reroute messages be sent (otherwise the large amount of traffic from the messages might interfere with nsreroute's ability to process the next command, and the program may not behave correctly, may not get all input, etc.).

```
.
.
.
/* Only look in Answer section */
section = DNS_SECTION_ANSWER;
result = dns_message_firstname(rcvmsg, section);
if (result != ISC_R_SUCCESS)
        fatal("response to NS query didn't contain an NS");

while (result == ISC_R_SUCCESS) {
        name = NULL;
        dns_message_currentname(rcvmsg, section, &name);
        soaset = NULL;

        /* Get all NS records in rcvmsg */
        result = dns_message_findtype(name, dns_rdatatype_ns, 0, &soaset);
        if (result == ISC_R_SUCCESS) {
                result = dns_rdataset_first(soaset);
                check_result(result, "dns_rdataset_first");
                dns_rdata_init(&soarr);
                dns_rdataset_current(soaset, &soarr);
                result = dns_rdata_tostruct(&soarr, &soa, NULL);
                check_result(result, "dns_rdata_tostruct");
                if (debugging) {
                        char namestr[DNS_NAME_FORMATSIZE];
                        dns_name_format(name, namestr, sizeof(namestr));
                        fprintf(stderr, "Found zone name: %s\n", namestr);
                }
                dns_name_init(&master, NULL);
                dns_name_clone(&soa.origin, &master);
                if (userzone != NULL)
                        zonename = userzone;   /* domain name of name server */
                else
                        zonename = name;
                if (debugging) {
```



```
                        char namestr[DNS_NAME_FORMATSIZE];
                        dns_name_format(&master, namestr, sizeof(namestr));
                        fprintf(stderr, "The master is: %s\n", namestr);
```

```
                }

                /* Get the address of the name server */
                if (userserver != NULL)
                        serveraddr = userserver;
                else {
                        char serverstr[DNS_NAME_MAXTEXT+1];
                        isc_buffer_t buf;
                        isc_buffer_init(&buf, serverstr, sizeof(serverstr));
                        result = dns_name_totext(&master, ISC_TRUE, &buf);
                        check_result(result, "dns_name_totext");
                        serverstr[isc_buffer_usedlength(&buf)] = 0;
                        get_address(serverstr, DNSDEFAULTPORT, &tempaddr);
                        serveraddr = &tempaddr;
                }

                /* Spawn child to send message */
                pid = fork();
                if (pid == 0) {                         /* child */
                        /* Display DNS address */
                        printf("pid %d: NS address is %s\n",
                                getpid(), inet_ntoa(serveraddr->type.sin.sin_addr));

                        /* Send message to one NS, clean up, and return */
                        send_reroute(zonename, serveraddr, localaddr);
                        dns_message_destroy(&soaquery);
                        dns_request_destroy(&request);
                        dns_rdata_freestruct(&soa);
                        dns_message_destroy(&rcvmsg);
                        return;
                }
                else if (pid == -1) {
                        printf("Client cannot fork; exiting\n");
                        exit(1);
                }
                result = dns_message_nextname(rcvmsg, section);
        }
}
.
.
.
```

Figure 6.7: Portion of code from function *recvns* in nsreroute.c

In *send_reroute* (shown below), the child process waits for the parent to die, and then processes the message to be sent. *dns_request_createvia* essentially sets up the TCP

connection with the server, and the call to *dns_request_verify_server(&request)* (defined

in lib/dns/include/dns/request.h) sets the boolean variable *must_verify_server* (a new field added to the definition of the *dns_request* structure in lib/dns/request.c) to true in the *request* variable. A value of true in *must_verify_server* causes authentication to take place between DNS server and sender.

```
.
.
.
/* Wait for parent to get NS addresses from all domains in
 * the client list.  Once parent has processed all clients, it
 * will terminate, and all children will send their
 * reroute messages.
 */
while (getppid() == starting_pid) { }
result = dns_message_gettempname(reroutemsg, &name);
check_result(result, "dns_message_gettempname");
dns_name_init(name, NULL);
dns_name_clone(zonename, name);
result = dns_message_gettemprdataset(reroutemsg, &rdataset);
check_result(result, "dns_message_gettemprdataset");
dns_rdataset_makequestion(rdataset, dns_rdataclass_in, dns_rdatatype_soa);
ISC_LIST_INIT(name->list);
ISC_LIST_APPEND(name->list, rdataset, link);
dns_message_addname(reroutemsg, name, DNS_SECTION_ZONE);
if (usevc)       /* always use TCP */
        options |= DNS_REQUESTOPT_TCP;
result = dns_request_createvia(requestmgr, reroutemsg, srcaddr,
                               master, options, key,
                               FIND_TIMEOUT, global_task,
                               reroute_completed, NULL, &request);
check_result(result, "dns_request_createvia");
dns_request_verify_server(&request);   /* authenticate server */
requests++;
.
.
.
```

Figure 6.8: Portion of code from *send_reroute* in nsreroute.c.

Being able to trace the inside of BIND without the normal execution flow being interrupted by a timeout was only possible after finding out where these timeout values

were and changing them to an arbitrarily large number. For using the *dig* query tool, the

following timeout values were increased in bin/dig/include/dig.h to allow for more effective code tracing:

```
#define TCP_TIMEOUT 100000  /* was 10 */
#define UDP_TIMEOUT 100000  /* was 5 */
#define SERVER_TIMEOUT 100000  /* was 1*/
```

Printing messages to screen in various parts of the program is obviously a vital diagnostic tool. This ability is removed in BIND v. 9, however, by code written in *ns_os_daemonize* in bin/named/unix/os.c. Here, stdout and stderr are closed (*close (STDOUT_FILENO);* and *close(STDERR_FILENO);*), so no *printf* statements make it to screen. I commented this piece of code out, and from then on could better see what was happening inside the BIND software.

Adding the New ALT rdatatype

Figuring out how to add the new rdatatype, ALT, for the proxy server addresses was a little tricky. The file enumtype.h (located in the directory bind-9.2.2/lib/dns/include/dns/) has a list of all rdatatypes that are used by BIND, along with each type's numerical equivalent. It would seem that all that is needed to add a new rdatatype is just to add it to this file, and re-configure, re-make, and re-install the software. However, after trying this and re-installing, the program still doesn't recognize the new rdatatype. What's more, the new ALT type disappears from enumtype.h after re-installing!

At the top of enumtype.h is the message, "THIS FILE IS AUTOMATICALLY

GENERATED BY gen.c. DO NOT EDIT!" So I looked for gen.c, and found it in lib/dns/. However, this file is even more cryptic than enumtype.h. I tried adding the new ALT type (with the new rdatatype value 99) to this file, and re-compiling, but still the software does not recognize the ALT type.

In the directory lib/dns/rdata/ there resides some interesting subdirectories: in_1, hs_4, and generic. In the in_1 subdirectory lies the files a_1.c, a_1.h, a6_38.c, a6_38.h, and others. In the generic subdirectory resides the files ns_2.c, ns_2.h, ptr_12.c, ptr_12.h, soa_6.c, soa_6.h, and others. After examining these files it dawned on me that the file a_1.c, for instance, has various routines concerned with handling IP addresses. These functions compare two addresses; check attributes of IP addresses; allocate memory; perform network address-to-text conversions; and others.

The name of the file, "a_1.c", begins with the rdatatype representation of IP addresses in BIND, which is the lowercase form of 'A'. The '1' refers to the constant value of the A rdatatype. Same with "ns_2.c": "ns" is the NS rdatatype that designates the name of a DNS server; '2' is its numerical equivalent. Ditto for "soa_6.c", and all the others.

This is the way that gen.c finds all the rdatatypes in BIND and constructs the enumtype.h file. Reading the rdatatypes in through the lib/dns/rdata/in_1 directory (where "in_1" refers to the standard "IN", or "Internet", class, with a corresponding value of 1), gen.c parses each file contained therein. Each rdatatype has a source and header file starting with the rdatatype letter abbreviation and ending with the value.

In lib/dns/rdata, I created alt_99.c and alt_99.h based on a_1.c and a_1.h,

respectively, since the ALT type essentially uses the same operations as the A rdatatype. Inside alt_99.c and alt_99.h, I changed all procedures to reflect this new rdatatype.

After re-configuring, compiling, and installing the BIND software, the new ALT type appeared in enumtype.h. The BIND software now accepted this type.

However, when I sent out a query for the address of a host with an ALT-type address, the query reply did not include the actual "ALT" letters. Instead, the words "TYPE99" appeared. In order to enable the user to see the designation "ALT" in front of a proxy server address in a query reply, I needed to add the ALT type and its value to other files.

I solved this problem by adding the ALT rdatatype and its numerical value 99 to lib/bind/include/arpa/nameser.h and nameser_compat.h. Now, when ALT addresses are present in the query response, the three-letter designation "ALT" appears in front of each address.

Using OpenSSL to Authenticate Server

After sending the reroutemsg to the server, control branches to the function *ssl_authenticate_server* in the file lib/dns/openssl_client.c. *ssl_authenticate_server* is called from the function *req_senddone* (in the file lib/dns/request.c) after the *must_verify_server* field of *request* is found to be true.

ssl_authenticate_server verifies the identity of the server. This entails verifying the legitimacy of the server's certificate. More rigorous checking of the server's identity is

not necessary. A new TCP connection on which to create the SSL (secure socket layer) is created, so that the sender and server certificates can be exchanged securely.

On the other hand, the server needs to know if the sender is legitimate, and takes the additional step of ensuring the sender is authorized to install the new zone on the server. For this the server checks that the sender's certificate is on a list of authorized certificates. I will describe the server's processing of the reroutemsg and its authentication of the sender in chapter seven.

Beforehand, sender and server must possess certificates that have been signed by a trusted certification authority (CA). Both sender and server must also possess the public key of this certification authority, so that the other's certificate can be authenticated.

For testing the program I created a self-signed root CA certificate, using the following OpenSSL commands [VMC02], [Res01]. (With all commands the default OpenSSL configuration file openssl.cnf is used.)

```
$ openssl req -newkey rsa:1024 -sha1 -keyout rootkey.pem -out rootreq.pem
$ openssl x509 -req -in rootreq.pem -sha1 -extensions v3_ca -signkey rootkey.pem
> -out rootcert.pem
$ cat rootcert.pem rootkey.pem > root.pem
```

These OpenSSL commands create the CA's certificate (rootcert.pem), the CA's private key (rootkey.pem), and a file with both (root.pem). The root CA signs its own certificate with its private key. The RSA algorithm is used to create the private key, which is 1024 bits long.

Next I created the server certificate, signing it with the root CA's private key:

```
$ openssl req -newkey rsa:1024 -sha1 -keyout serverkey.pem -out serverreq.pem
$ openssl x509 -req -in serverreq.pem -sha1 -extensions usr_cert -CA root.pem -CAkey
> root.pem -CAcreateserial -out servercert.pem
$ cat servercert.pem serverkey.pem rootcert.pem > server.pem
```

The result is the server's certificate (servercert.pem), the server's private key (serverkey.pem), and a concatenation of these with the CA's certificate (server.pem). The private key is again created using the RSA 1024-bit encryption algorithm. The server's certificate is signed by the root CA's private key.

In much the same way I created the client's (sender's) certificate:

```
$ openssl req -newkey rsa:1024 -sha1 -keyout clientkey.pem -out clientreq.pem
$ openssl x509 -req -in clientreq.pem -sha1 -extensions usr_cert -CA root.pem -CAkey
> root.pem -CAcreateserial -out clientcert.pem
$ cat clientcert.pem clientkey.pem rootcert.pem > client.pem
```

The sender's certificate is clientcert.pem, and is signed by the CA's private key. An RSA encryption algorithm generates the client's 1024-bit private key, as for the server and root CA keys.

The root certificate is resident on both client and server machines, because it is used by both to verify each other's certificate.

ssl_authenticate_server does the following:

- Loads a random file (client_seed) that seeds the pseudo random number generator (PRNG). This enables each session to be encrypted with unique keys.
- Initializes the SSL context. This loads the sender's certificate, loads error messages, and locates all trusted CAs.
- Creates a new TCP socket, and tries to connect to port 5300 of the server with whom the sender already has an established TCP connection (at the server's port 53).
- If successful, the sender establishes an SSL connection with the server on top of this TCP connection.

- Through this encrypted SSL connection the sender receives the server's certificate and

verifies that it is legitimate. (The sender is not overly concerned about the identity of the server. Once the sender sends the reroutemsg to the server, the sender has effectively lost control of the message, and it does not matter whether the server can be verified or not.)

- Closes the SSL connection and the underlying TCP connection.
- As indicated in Figure 6.2, the client process is immediately terminated.

The code of *ssl_authenticate_server* is shown below:

```
void ssl_authenticate_server(dns_request_t *request) {
        SSL_CTX *ctx;
        SSL *ssl;
        BIO *sbio;
        int sock;
        int result;
        isc_socket_t *server_socket;
        isc_sockaddr_t address;

        RAND_cleanup();

        /* Load randomness */
        if (!(RAND_load_file(RANDOM_CLIENT, 1024)))
                berr_exit("Couldn't load randomness");

        /* Create SSL context */
        ctx = initialize_client_ctx();

        /* Get the file descriptor for the current TCP socket */
        server_socket = dns_request_socket(&request);
        address = isc_socket_address(server_socket);
        address.type.sin.sin_port = htons(PORT);
        sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

        if (sock < 0) {
                SSL_CTX_free(ctx);
                err_exit("Couldn't create socket");
        }

        printf("pid %d: sending connect to %s\n", getpid(),
                inet_ntoa(address.type.sin.sin_addr));
        printf("at port #%d\n", ntohs(address.type.sin.sin_port));
```



```
        if (connect_timeo(sock, (struct sockaddr *)&address.type.sin,
                                        sizeof(address.type.sin), 2) < 0) {
```

```
                close(sock);
                SSL_CTX_free(ctx);
                printf("pid %d: Couldn't connect socket\n", getpid());
                exit(0);
        }

        /* Connect the SSL socket */
        sbio = BIO_new(BIO_s_socket());
        BIO_set_fd(sbio, sock, BIO_NOCLOSE);
        ssl = SSL_new(ctx);
        SSL_set_bio(ssl, sbio, sbio);
        sleep(1);       /* Let the server catch up before attempting to connect */
        result = SSL_connect(ssl);

        if (result <= 0) {
                printf("pid %d: SSL_connect error, result = %d\n", getpid(), result);
                goto cleanup;
        }

        /* Verify authenticity of server's certificate */
        if (SSL_get_verify_result(ssl) != X509_V_OK)
                printf("Certificate doesn't verify\n");

cleanup:
        SSL_shutdown(ssl);
        SSL_free(ssl);
        SSL_CTX_free(ctx);
        close(sock);
        ERR_remove_state(0);
}
```

Figure 6.9: The function *ssl_authenticate_server*, in the lib/dns/ directory.

One note about the call to `sleep(1)` in *ssl_authenticate_server*. During testing I found that sometimes the client would reach the call to `connect` sooner than the server could reach the call to `accept`. This is most likely due to the fact that the server needs to load the list of authorized clients before he gets to `accept`. If the client tries to connect without a waiting accept on the server side, the client does not wait and declares an error has been

committed ("Couldn't connect socket"). Therefore, to increase the reliability of the program, I make the client sleep for one second before attempting the connect.

The call to *connect_timeo* is a wrapper around the standard *connect* call. *connect_timeo*, written by W. Richard Stevens [Ste98], raises an alarm if the client has not yet received a SYN/ACK from the server after two seconds. The alarm causes the connect to fail, and the client process immediately terminates. This helps make the client program more efficient, because incomplete TCP connections waste resources and slow program execution.

After receiving the reroutemsg, the server verifies the sender and makes sure he is authorized to install the zone that includes the victim's DNS servers. Upon successful authentication the server processes the message, installing the new zone file, updating the configuration file, and reloading all zones into memory. The enhancement of BIND that enables the server to accomplish these actions is described next.

CHAPTER VII

ENHANCEMENT OF BIND FOR HANDLING NSREROUTE MESSAGES

This section concentrates on the code added to the BIND DNS software that processes the new reroute message, as generated by the nsreroute program. The execution of named, when it receives the nsreroute message, is described. Then the code additions to the BIND software that implements the server's actions is presented. Before closing this section it is shown how the server verifies the sender using OpenSSL. The authentication involves checking the sender's certificate against a list of approved certificates.

Execution Flow of named Daemon After Receiving nsreroute Message

The named daemon is the most important part of the BIND DNS package. It is the process that continually waits for a message (usually a query), processes it, and waits for the next message. It never terminates, unless intentionally shut down by the administrator for maintenance.

As it has been previously described, when the victim is attacked, he alerts a proxy server (the "sender") to issue the reroute message to a list of known clients. The proxy server in turn runs the nsreroute program, with each client on one line of input. The sender first gets the NS resource records, which include the names and addresses of all the authoritative DNS servers for the client's zone. These are pulled from the client's

primary master DNS server, which is the one defined by the SOA record at the start of all zone files.

Once the proxy server gets this list of NS RRs, it constructs a reroutemsg and forks a child process for each one. When all NS records for all clients in the list are received, the parent process is terminated, and all the child processes send their messages to the intended DNS servers via TCP. During this TCP connection, another TCP connection is initiated by the sender, where the client DNS server and the sender exchange certificates and verify each other's identity using SSL.

When the client's primary master DNS server responds to the NS query with a list of NS RRs, it is not new behavior for the named daemon, so I will not describe it further. However, when the DNS servers in the list receive the reroutemsg, the code that named uses to process this message is new, and is depicted in the function flow diagram below.

```
dispatch()
        client_request()
                switch (client->message->opcode) {
                .
                .
                .
                case dns_opcode_reroute:
                        ns_reroute_start()     /* Opcode == 7 */
                                Check legality of DNS message (i.e., it must contain only name
                                in the Question section, and that name must have only one
                                rdataset of type soa)

                                Check database to make sure server is authoritative for zone
                                in DNS message's Question section
                .
                .
                .
                }
dispatch()
        reroute_action()
```



```
                result = ssl_authenticate_client()
```

```
                Authenticate client and check if he is an authorized reroutemsg sender
        if (result == not successful) goto failure

        result = write_zone()
                Create new zone file for victim's domain on disk
        if (result == not successful) goto failure

        result = append_toconfig()
                To configuration file (named.conf), add new zone statement
                corresponding to newly written zone file
        if (result == not successful) goto failure
dispatch()
    reroutedone_action()
        If a new zone has been added, signal to reload all zones into memory
dispatch()
    ns_server_reload()
        reload zones
dispatch()
    timer_cleanup()
dispatch()
    Wait for next message
```

Figure 7.1: named server daemon processing of reroutemsg.

Here is a summary of the actions of named in Figure 7.1:

- Get message, check message's opcode. The new reroutemsg opcode = 7. Branch to *ns_reroute_start* to begin processing the reroutemsg.
- In *ns_reroute_start*, message is checked for correct number of questions (one) and correct attached rdataset (type soa). Server database is checked to see if it is the authoritative DNS server for the zone indicated by the name in the Question section of reroutemsg. If the message is legal, it is passed to *reroute_action*.
- In *reroute_action*, three important actions are performed:
    - Verify sender's identity through *ssl_authenticate_client*. In this function the server listens on a new port #5300 for the sender, so that the sender and the DNS server can exchange certificates and the sender's certificate can be checked against a list of

    approved senders. If the sender doesn't check out, the function exits and the message is discarded.

  - Go to *write_zone*, and write a new zone file to the directory where the server keeps zone files. This is determined by the options statement in named.conf. This zone file displays the contents of the reroutemsg: namely, the victim DNS server's two host names are written, along with their regular A-type addresses as well as ALT-type addresses. For the new zone's SOA record, the server writes its own fully-qualified domain name. If the zone file was created and successfully written, go to the next step in *reroute_action*; otherwise, clean up and exit.
  - Execute *append_toconfig*, where a zone statement specifying the new domain and the corresponding zone file is appended to the end of the named.conf configuration file (see Figure 4.4). If there was a problem opening this file or another error occurred, print the error, clean up and exit.
- Go to *reroutedone_action* and, if a new zone has been created, signal the system to reload zones.
- If zone files should be reloaded into memory, execute *ns_server_reload.*
- Clean up data structures and wait for next message.

A couple of points are worth mentioning about named's handling of the reroutemsg. In *reroute_action*, if one of the three actions fails, then nothing needs to be undone. If the authentication fails, then clean up and exit. If authentication succeeds but there was an error creating the zone file, then still nothing needs to be undone; just clean up and exit. If authentication succeeds and the new zone file is created, but the zone statement cannot

be appended to the configuration file, then the zone file does not get loaded into memory. Although there is an unused zone file on the hard drive, its presence does not interfere with anything. Only if all three actions are completed successfully does the new zone get loaded into memory; otherwise, it doesn't and nothing needs to be undone for the server to operate correctly.

A second point to ponder is that, during *ssl_authenticate_client*, a new TCP connection is established with the sender in order to create an secure socket layer so that certificates can be exchanged securely. What would happen if the client suddenly was not interested in issuing a connect call to the server's port #5300? In the current implementation, the server blocks indefinitely, waiting for the connect from the sender. Obviously, this could be an invitation for mischief. An unscrupulous user could potentially send a fake reroutemsg and hang up the server's TCP port #53 (while it waits for a connect on TCP port #5300).

This avenue of abuse may be shut down by making the server's TCP accept call on port 5300 non-blocking. Since the connect call is also non-blocking, the server may also need to call *sleep()* and wait one or two seconds for the connect call to come. This way the server always waits two seconds, even when the connect call arrives immediately. This is the downside, but the upside is the TCP port does not wait for very long in the event of an absent connect call.

Earlier in this project I attempted to fork the server before the call to *ssl_authenticate_client.* This let the parent continue on to answer the next query, while

the child took care of the events in *reroute_action*. It seemed like the perfect way to handle delays in processing the reroutemsg.

However, when the child finished *reroute_action* and reloaded the zones, this reloading occurred just for the child process, not the parent. The parent process did not have any knowledge of the new zone for the victim. When the child process exited, any new zones that were reloaded while it was alive also vanished. One way to let the server know of the new zone would be to have it wait until the child finished, and then call for a zone reload itself. But then, what good would it be to fork in the first place?

The other way is to have parent and child processes access to the shared memory. In this case forking in the server would be very useful. My test system does not allow for this sharing of memory, however.

Code Additions

The file reroute.c, residing in bind-9.2.2/bin/named/, includes most of the functionality required to handle the reroutemsg in the named server. In reroute.c can be found the functions *ns_reroute_start*, *reroute_action*, *write_zone*, *append_toconfig*, and *reroutedone_action*, which are described in Section 7.1 above. The code for *ns_reroute_start* and *reroutedone_action* is mostly derived from code in update.c. The three remaining functions are my own and are shown below, with comments. Code implementing *ssl_authenticate_client* is located in openssl_server.c and will be shown in the next section.

```
static void
reroute_action(isc_task_t *task, isc_event_t *event) {
	reroute_event_t *uev = (reroute_event_t *) event;
	ns_client_t *client = (ns_client_t *)event->ev_arg;
	isc_result_t result;
	isc_mem_t *mctx = client->mctx;
	dns_message_t *request = client->message;

	INSIST(event->ev_type == DNS_EVENT_REROUTE);

	/* Authenticate client */
	result = ssl_authenticate_client();
	if (result != ISC_R_SUCCESS)
		goto failure;

	/* Create new zone file */
	result = write_zone(request, mctx);
	if (result != ISC_R_SUCCESS)
		goto failure;

	/* Create new zone entry in /etc/named.conf */
	result = append_toconfig(request, mctx);
	if (result != ISC_R_SUCCESS)
		goto failure;

 failure:
	isc_task_detach(&task);
	uev->result = result;
	uev->ev_type = DNS_EVENT_REROUTEDONE;
	uev->ev_action = reroutedone_action;
	isc_task_send(client->task, &event);
	INSIST(event == NULL);
}
```

Figure 7.2: Function *reroute_action* in bin/named/reroute.c.

The code in *reroute_action* needs little explanation. It contains calls to *ssl_authenticate_client*, *write_zone*, and *append_toconfig*. If any one of these functions returns a value other than 0 (ISC_R_SUCCESS), any remaining functions are skipped and the victim's zone is not installed on the server.

The following code implements *write_zone*. It extracts the elements of reroutemsg that originally resided as a line of input to the client's nsreroute program, and creates a

legal zone file. This function uses a wide assortment of routines culled from various files in BIND. The most useful ones in terms of manipulating pointers to structures of type dns_name_t, dns_rdataset_t, dns_message_t, and isc_buffer_t are defined in message.h, name.h, rdataset.h, rdata.h, master.h and buffer.h, but many other header files in BIND were consulted as well. It would have been very difficult, if not impossible, for me to manipulate these structures effectively without the use of these pre-defined routines. Even finding and understanding how these routines worked took quite a bit of time in itself.

```
static isc_result_t
write_zone(dns_message_t *msg, isc_mem_t *mctx) {
        isc_result_t result;
        dns_name_t *victim_name = NULL;
        dns_name_t *victim_name2 = NULL;
        dns_master_style_t *style;
        dns_rdataset_t *rdataset;
        dns_rdataset_t *rdataset2;
        dns_rdataset_t *rdataset_proxy;
        dns_rdataset_t *rdataset_proxy2;
        dns_name_t *server_domain_name = NULL;
        dns_name_t *victim_suffix = NULL;
        isc_buffer_t *victim_suffix_buf = NULL;
        isc_buffer_t *victim_dns_buf = NULL;
        isc_buffer_t *victim_dns_buf2 = NULL;
        isc_buffer_t *buf = NULL;
        isc_buffer_t *buf2 = NULL;
        isc_buffer_t *proxy_buf = NULL;
        isc_buffer_t *proxy_buf2 = NULL;
        unsigned int len = OUTPUTBUF;
        unsigned int labels = 0;
        char victim_domain_ptr[DNS_NAME_FORMATSIZE];
        char server_domain_ptr[DNS_NAME_FORMATSIZE];
        char filename_ptr[DNS_NAME_FORMATSIZE];
        char host_name[256];
        int fd;

        style = &dns_master_style_default;
        result = dns_message_firstname(msg, DNS_SECTION_REROUTE);
        if (result != ISC_R_SUCCESS)
                return result;
```

```
/* Put the first victim DNS server name and its IP address into buf */
dns_message_currentname(msg, DNS_SECTION_REROUTE, &victim_name);
rdataset = victim_name->list.head;
result = isc_buffer_allocate(mctx, &buf, len);
if (result != ISC_R_SUCCESS)
        return result;
result = dns_master_rdatasettotext(victim_name, rdataset, style, buf);
if (result != ISC_R_SUCCESS)
        return result;

/* Put the second victim DNS server name and its IP address into buf2 */
result = dns_message_nextname(msg, DNS_SECTION_REROUTE);
dns_message_currentname(msg, DNS_SECTION_REROUTE, &victim_name2);
rdataset2 = victim_name2->list.head;
result = isc_buffer_allocate(mctx, &buf2, len);
if (result != ISC_R_SUCCESS)
        return result;
result = dns_master_rdatasettotext(victim_name2, rdataset2, style, buf2);
if (result != ISC_R_SUCCESS)
        return result;

/* Put the first victim DNS server name and a list
 * of proxy server IP addresses into proxy_buf
 */
rdataset_proxy = victim_name->list.head->link.next;
result = isc_buffer_allocate(mctx, &proxy_buf, len);
if (result != ISC_R_SUCCESS)
        return result;
result = dns_master_rdatasettotext(victim_name, rdataset_proxy, style, proxy_buf);
if (result != ISC_R_SUCCESS)
        return result;

/* Put the second victim DNS server name and a list
 * of proxy server IP addresses into proxy_buf2
 */
rdataset_proxy2 = victim_name2->list.head->link.next;
result = isc_buffer_allocate(mctx, &proxy_buf2, len);
if (result != ISC_R_SUCCESS)
        return result;
result = dns_master_rdatasettotext(victim_name2, rdataset_proxy2, style, proxy_buf2);
if (result != ISC_R_SUCCESS)
        return result;

/* Get domain name of victim.  Put into victim_suffix. */
result = dns_message_gettempname(msg, &victim_suffix);
if (result != ISC_R_SUCCESS)
        return result;

result = isc_buffer_allocate(mctx, &victim_suffix_buf, len);
if (result != ISC_R_SUCCESS) {
        dns_message_puttempname(msg, &victim_suffix);
        return result;
```

```
}

dns_name_setbuffer(victim_suffix, victim_suffix_buf);
labels = dns_name_countlabels(victim_name);
result = dns_name_splitatdepth(victim_name, labels-1, NULL, victim_suffix);
if (result != ISC_R_SUCCESS) {
        dns_message_puttempname(msg, &victim_suffix);
        return result;
}

/* Point victim_domain_ptr to domain name of victim DNS server */
dns_name_format(victim_suffix, victim_domain_ptr, DNS_NAME_FORMATSIZE);

/* This host is the server of authority */
result = gethostname(host_name, 255);     /* get FQDN of current host */

/* Put the FQDN of first victim's DNS server into victim_dns_buf */
result = isc_buffer_allocate(mctx, &victim_dns_buf, len);
result = dns_name_tofilenametext(victim_name, ISC_FALSE, victim_dns_buf);

/* Put the FQDN of second victim's DNS server into victim_dns_buf2 */
result = isc_buffer_allocate(mctx, &victim_dns_buf2, len);
result = dns_name_tofilenametext(victim_name2, ISC_FALSE, victim_dns_buf2);

/* Point server_domain_ptr to domain name of SOA server */
result = dns_message_firstname(msg, DNS_SECTION_QUESTION);
if (result != ISC_R_SUCCESS) {
        dns_message_puttempname(msg, &victim_suffix);
        return result;
}

dns_message_currentname(msg, DNS_SECTION_QUESTION, &server_domain_name);
dns_name_format(server_domain_name, server_domain_ptr, DNS_NAME_FORMATSIZE);

/* Construct name of zone file using "db." followed by domain name of victim */
strcpy(filename_ptr, ZONE_DIR);   /* ZONE_DIR is where zone files reside */
strcat(filename_ptr, "/db.");
strncat(filename_ptr, victim_domain_ptr, strlen(victim_domain_ptr));

/* Open new file with exclusive write access */
fd = open(filename_ptr, O_CREAT|O_WRONLY|O_EXCL,
                S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);
if (fd == -1) {
        printf("Could not create new zone file\n");
        isc_buffer_free(&buf);
        isc_buffer_free(&proxy_buf);
        isc_buffer_free(&victim_suffix_buf);
        isc_buffer_free(&victim_dns_buf);
        dns_message_puttempname(msg, &victim_suffix);
        return ISC_R_IOERROR;           /* file already exists or other error */
}
```

```
	/* Write contents of new zone file */
	write(fd, "@ IN SOA ", 9);
	write(fd, host_name, strlen(host_name));
	write(fd, ". root.", 7);
	write(fd, server_domain_ptr, strlen(server_domain_ptr));
	write(fd, ". (\n", 4);
	write(fd, "\t\t\t\t1\t; Serial\n", 15);
	write(fd, "\t\t\t\t3h\t; Refresh after 3 hours\n", 31);
	write(fd, "\t\t\t\t1h\t; Retry after 1 hour\n", 28);
	write(fd, "\t\t\t\t1w\t; Expire after 1 week\n", 29);
	write(fd, "\t\t\t\t1h\t; Negative caching TTL of 1 hour\n", 40);
	write(fd, "\t\t\t\t)\n\n", 7);
	write(fd, "\t\t\tIN NS\t", 9);
	write(fd, (char *)isc_buffer_base(victim_dns_buf), (int)isc_buffer_usedlength(victim_dns_buf));
	write(fd, "\n", 1);
	write(fd, "\t\t\tIN NS\t", 9);
	write(fd, (char *)isc_buffer_base(victim_dns_buf2), (int)isc_buffer_usedlength(victim_dns_buf2));
	write(fd, "\n\n", 2);
	write(fd, (char *)isc_buffer_base(buf), (int)isc_buffer_usedlength(buf));
	write(fd, (char *)isc_buffer_base(proxy_buf), (int)isc_buffer_usedlength(proxy_buf));
	write(fd, "\n\n", 1);
	write(fd, (char *)isc_buffer_base(buf2), (int)isc_buffer_usedlength(buf2));
	write(fd, (char *)isc_buffer_base(proxy_buf2), (int)isc_buffer_usedlength(proxy_buf2));
	write(fd, "\n", 1);

	if (close(fd) == -1) {
		dns_message_puttempname(msg, &victim_suffix);
		return ISC_R_IOERROR;   /* error closing file */
	}

	/* Clean up */
	isc_buffer_free(&buf);
	isc_buffer_free(&buf2);
	isc_buffer_free(&proxy_buf);
	isc_buffer_free(&proxy_buf2);
	isc_buffer_free(&victim_suffix_buf);
	isc_buffer_free(&victim_dns_buf);
	isc_buffer_free(&victim_dns_buf2);
	dns_message_puttempname(msg, &victim_suffix);
	return ISC_R_SUCCESS;
}
```

Figure 7.3: Function *write_zone* in reroute.c.

The last function of note in reroute.c, *append_toconfig*, is displayed below.

```
static isc_result_t
append_toconfig(dns_message_t *msg, isc_mem_t *mctx) {
	isc_result_t result;
	dns_name_t *victim_dns = NULL;
```

```
dns_name_t *prefix = NULL;
dns_name_t *suffix = NULL;
isc_buffer_t *suffix_buf = NULL;
int fd;
unsigned int len = OUTPUTBUF;
unsigned int labels = 0;
char victim_domain_ptr[DNS_NAME_FORMATSIZE];

/* Get first name of Reroute section (the Authority section) of msg */
result = dns_message_firstname(msg, DNS_SECTION_REROUTE);
if (result != ISC_R_SUCCESS)
        return result;
dns_message_currentname(msg, DNS_SECTION_REROUTE, &victim_dns);

result = dns_message_gettempname(msg, &suffix);
if (result != ISC_R_SUCCESS)
        return result;

result = isc_buffer_allocate(mctx, &suffix_buf, len);
if (result != ISC_R_SUCCESS) {
        dns_message_puttempname(msg, &suffix);
        return result;
}

/* Store domain of victim_dns in suffix */
dns_name_setbuffer(suffix, suffix_buf);
labels = dns_name_countlabels(victim_dns);
result = dns_name_splitatdepth(victim_dns, labels-1, prefix, suffix);
if (result != ISC_R_SUCCESS) {
        dns_message_puttempname(msg, &suffix);
        return result;
}

/* Point victim_domain_ptr to domain name of victim DNS server */
dns_name_format(suffix, victim_domain_ptr, DNS_NAME_FORMATSIZE);

/* Append entry to named.conf */
fd = open(NAMED_CONF, O_APPEND|O_RDWR);
if (fd == -1) {
        dns_message_puttempname(msg, &suffix);
        return ISC_R_IOERROR;   /* error opening file */
}
if (flock(fd, LOCK_EX) == -1) {
        dns_message_puttempname(msg, &suffix);
        return ISC_R_IOERROR;   /* couldn't get exclusive lock on file */
}

/* Write contents of zone statement */
write(fd, "\n", 1);
write(fd, "zone \"", 6);
write(fd, victim_domain_ptr, strlen(victim_domain_ptr));
write(fd, "\" IN {\n", 7);
```

```
        write(fd, "\ttype master;\n", 14);
        write(fd, "\tfile \"db.", 10);
        write(fd, victim_domain_ptr, strlen(victim_domain_ptr));
        write(fd, "\";\n", 3);
        write(fd, "};\n", 3);

        if (flock(fd, LOCK_UN) == -1) {
                dns_message_puttempname(msg, &suffix);
                return ISC_R_IOERROR;   /* couldn't unlock file */
        }
        if (close(fd) == -1) {
                dns_message_puttempname(msg, &suffix);
                return ISC_R_IOERROR;   /* error closing file */
        }

        /* Clean up */
        isc_buffer_free(&suffix_buf);
        dns_message_puttempname(msg, &suffix);
        return ISC_R_SUCCESS;
}
```

Figure 7.4: Code for function *append_toconfig* in reroute.c.

append_toconfig contains more file I/O code and manipulation of data structures, similar to the code found in *write_zone.*

Other bits of code were inserted in various and sundry places in BIND to help make everything work. This includes the following small additions:

1) Added

```
#define DNS_SECTION_REROUTE       DNS_SECTION_AUTHORITY
```

to lib/dns/include/message.h.

2) Added

```
dns_opcode_reroute = 7
#define dns_opcode_reroute          ((dns_opcode_t)dns_opcode_reroute)
```

to lib/dns/include/types.h.

3) Added

```
#define DNS_RDATA_REROUTE         0x0002
```

to lib/dns/include/rdata.h.

4) Added

```
#define DNS_EVENT_REROUTE                  (ISC_EVENTCLASS_DNS + 98)
#define DNS_EVENT_REROUTEDONE         (ISC_EVENTCLASS_DNS + 99)
```

to lib/dns/include/events.h.

5) Added

```
"REROUTE"
```

in the seventh position (starting from zero) in the list of strings in the definition of *static const char *opcodetext[]* in bin/named/dig.c.

Using OpenSSL to Authenticate Client

The last important function to describe in more detail is *ssl_authenticate_client*, found in the bin/named/openssl_server.c file. This function implements the verification of the client by establishing a new TCP connection on port 5300, exchanging certificates, and checking if the client's certificate is in a list of approved reroutemsg senders.

This function uses certificates created by the same methods as described in Chapter 6, so I will not repeat them. Suffice it to say that both client and server, for purposes of testing this program, have been successfully created, and both machines possess the certificate of the trusted certification authority needed to verify each other's public certificates.

The implementation of *ssl_authenticate_client* is shown below. As with *ssl_authenticate_server*, this function uses OpenSSL functions I learned about in [VMC02] and [Res01].

```
isc_result_t ssl_authenticate_client(void) {
	SSL_CTX *ctx;
	SSL *ssl;
	BIO *sbio;
	int sock, s;
	isc_result_t ret;
	int result;
	struct sockaddr_in sin;
	int val = 1;
	access_list_t clientAccessList;

	/* Load list of authorized clients from file */
	clientAccessList = load_access_list();
	if (clientAccessList.clientListSize == 0)
		return ISC_R_FAILURE;

	RAND_cleanup();

	/* Seed random number generator */
	if (!(RAND_load_file(RANDOM_SERVER, 1024))) {
		printf("Couldn't load randomness\n");
		return ISC_R_FAILURE;
	}

	/* Create SSL context */
	ctx = initialize_server_ctx();

	SSL_CTX_set_session_id_context(ctx, (void *)&s_server_session_id_context,
						sizeof s_server_session_id_context);

	/* Get file descriptor and create socket */
	sock = socket(AF_INET, SOCK_STREAM, 0);
	if (sock < 0) {
		printf("pid %d: Error, couldn't make socket\n", getpid());
		SSL_CTX_free(ctx);
		return ISC_R_FAILURE;
	}
	memset(&sin, 0, sizeof(sin));
	sin.sin_addr.s_addr = INADDR_ANY;
	sin.sin_family = AF_INET;
	sin.sin_port = htons(PORT);
	setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
```

```
	/* Bind to socket */
	if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
		printf("pid %d: Error, couldn'  bind\n", getpid());
		close(sock);
		SSL_CTX_free(ctx);
		return ISC_R_FAILURE;
	}

	/* Listen for connection */
	listen(sock, 5);

	s = accept(sock, 0, 0);

	if (s < 0) {
		printf("pid %d: Error, problem accepting\n", getpid());
		close(sock);
		SSL_CTX_free(ctx);
		return ISC_R_FAILURE;
	}

	/* Create ssl object and accept ssl connection */
	sbio = BIO_new(BIO_s_socket());
	BIO_set_fd(sbio, s, BIO_NOCLOSE);
	ssl = SSL_new(ctx);
	SSL_set_accept_state(ssl);
	SSL_set_bio(ssl, sbio, sbio);

	result = SSL_accept(ssl);

	if (result <= 0) {
		printf("pid %d: SSL_accept error, result = %d\n", getpid(), result);
		ret = ISC_R_FAILURE;
		goto cleanup;
	}
	/* Check to see if client is authorized to perform nsreroute */
	ret = checkClientCert(ssl, clientAccessList);
	if (ret == ISC_R_FAILURE)
		printf("Client not authorized for nsreroute\n");

cleanup:
	/* Close ssl and underlying connection, and return error code */
	SSL_shutdown(ssl);
	SSL_free(ssl);
	SSL_CTX_free(ctx);
	close(s);
	close(sock);
	ERR_remove_state(0);
	return ret;
}
```

Figure 7.5: *ssl_authenticate_client* in bin/named/openssl_server.c.

The server checks that the client is authorized to install a new zone on the server in the call to *checkClientCert*. This is something the client doesn't do to the server, because it isn't needed. However, it is very important for the server.

Before invoking *checkClientCert*, the server calls *load_access_list()*, which loads a list of approved reroutemsg senders from a file into the following data structure:

```
typedef struct clientDN {
	char		data[256];	/* distinguished name (DN) */
	int		size;		/* sizeof DN */
} clientDN_t;

typedef struct access_list {
	int			clientListSize;		/* number of authorized DNs */
	clientDN_t	client[MAX_ACCESS_FILE_SIZE];	/* structure containing DN
									 * size, DN string, and
									 * access level */
} access_list_t;
```

Figure 7.6: Data structure for storing contents of client access file.

The following code implements *load_access_list*:

```
static access_list_t load_access_list(void) {
	access_list_t clientAccessList;
	isc_result_t result;
	FILE *f = NULL;
	char DN[256];
	int i = 0;

	/* Initialize clientAccessList */
	initializeAccessList(clientAccessList);

	/* Open input file */
	result = isc_stdio_open(ACCESS_FILE, "r", &f);
	if (result != ISC_R_SUCCESS) {
		printf("Error opening access file\n");
		return clientAccessList;
	}

	/* Load file into clientAccessList structure */
	while (i < MAX_ACCESS_FILE_SIZE && fgets(DN, 256, f) != NULL) {
```



```
		strncpy(clientAccessList.client[i].data, DN, strlen(DN) - 1);
```

```
		clientAccessList.client[i].size = strlen(DN) - 1;
		i++;
	}
	if (i == MAX_ACCESS_FILE_SIZE)
		printf("Warning: File size may be larger than maximum allowed\n");

	clientAccessList.clientListSize = i;

	/* Close input file */
	result = isc_stdio_close(f);
	if (result != ISC_R_SUCCESS) {
		printf("Error closing access file\n");
		clientAccessList.clientListSize = 0;
		return clientAccessList;
	}
	return clientAccessList;
}
```

Figure 7.7: *load_access_list* from openssl_server.c.

After the list of authorized clients has been loaded into the clientAccessList data structure, and an SSL connection with the sender is established on server's port #5300, the server gets the client's public certificate and verifies his identity by invoking *checkClientCert* (*checkClientCert* code courtesy of University of Texas, *Keystone* project):

```
static isc_result_t
checkClientCert(void *ssl_v, access_list_t clientAccessList) {
	SSL *ssl = (SSL *)ssl_v;
	X509 *peerCert;
	char peerName[256];
	int i;
	int peerNameSize = 0;

	for (i = 0; i < 256; i++)
		peerName[i] = '  \0'  ;

	/* Get client certificate */
	if ((peerCert = SSL_get_peer_certificate(ssl)) == NULL) {
		fprintf(stderr, "Error: no peer cert\n");
		return ISC_R_FAILURE;
	}
```



```
	/* Extract the client's Distinguished Name */
```

```
        X509_NAME_oneline(X509_get_subject_name(peerCert),
                                        peerName, sizeof(peerName));
        peerNameSize = strlen(peerName);

        /* Search clientAccessList for a match */
        for (i = 0; i < clientAccessList.clientListSize; i++) {
                if (clientAccessList.client[i].size != peerNameSize)
                        continue;
                if (strncmp(clientAccessList.client[i].data, peerName, peerNameSize) == 0)
                        break;
        }

        /* Return failure if no match found */
        if (i >= clientAccessList.clientListSize)
                return ISC_R_FAILURE;

        return ISC_R_SUCCESS;
}
```

Figure 7.8: *checkClientCert* in openssl_server.c.

Resulting Zone File and Zone Statement Written to Client DNS Server

From the previous chapter, the sending machine used the following command as input for the nsreroute program:

```
reroute client.clientnet.com. victimdns1.victimnet.com. victimdns2.victimnet.com. 133.41.96.7
133.41.96.8 203.55.5.10 222.4.11.81 221.16.25.3
```

Now, after the server has successfully authenticated the client and processes this message, the following zone file appears on the server's hard drive (where we assume the client's DNS server is called "clientdns"):

```
@ IN SOA clientdns.clientnet.com. root.clientnet.com. (
                                1       ; Serial
                                3h      ; Refresh after 3 hours
                                1h      ; Retry after 1 hour
                                1w      ; Expire after 1 week
                                1h      ; Negative caching TTL of 1 hour
                                )
```



```
                        IN NS           victimdns1.victimnet.com.
```

```
                    IN NS          victimdns2.victimnet.com.

victimdns1.victimnet.com. 86400  IN A 133.41.96.7
victimdns1.victimnet.com. 86400  IN ALT 203.55.5.10
                    86400 IN ALT 222.4.11.81
                    86400 IN ALT 221.16.25.3

victimdns2.victimnet.com. 86400  IN A 133.41.96.8
victimdns2.victimnet.com. 86400  IN ALT 222.4.11.81
                    86400 IN ALT 221.16.25.3
                    86400 IN ALT 203.55.5.10
```

Figure 7.9: Resulting zone file on client's DNS server after processing reroutemsg.

The zone statement for this zone file (called "db.victimnet.com"), appended to /etc/named.conf, looks like

```
zone "victimnet.com" IN {
        type master;
        file "db.victimnet.com";
};
```

Figure 7.10: Resulting zone statement appended to named.conf.

Taken together, the zone file and zone statement as shown above will now help reroute querying clients to the victim during a distributed denial of service (DDOS) attack on the victim.

A query for any host on domain victimnet.com will be sent to the victim's DNS server through one of the proxy servers at an IP address designated by the ALT datatype. The received reply from the victim's name server also includes at least one ALT address for the desired victim host, so that the client can use this alternate path to communicate with the victim.

The details of enabling the client's DNS server (after the victim's zone has been installed through the reroutemsg) to forward a query to the victim's DNS server, using an IP tunnel, is described in the next chapter.

CHAPTER VIII

ENHANCEMENT OF BIND FOR CLIENT QUERY OF HOST IN VICTIM DOMAIN

In this chapter we discuss why the nsreroute program is called what it is: It enables the name server to literally reroute queries through a proxy server to the victim name server. This is important because the proxy server knows of a secret gateway for the victim, one that is not deluged with a flood of DDOS traffic. Using this secret gateway makes all the difference in a client being able to communicate with the victim during a DDOS blitz, when usually it would be impossible.

To illustrate how this new behavior in the named server is implemented, consider the normal way the BIND software would process such a query in the absence of an authoritative zone. In this case the client DNS server would issue an iterative query to the closest authoritative name server for the victim domain that is known to the client server [AL01]. If the client server does not know of any DNS servers that have authority over "victimnet.com", then it will see if it knows of a DNS server that is authoritative for the "com" domain. If not, then it queries one of the 13 root DNS servers for the closest authoritative name server for the victim's domain.

The root DNS server will point it to an authoritative server for the "com" domain, which the client server will then query. The "com" server will point the client server to

the DNS server authoritative for "victimnet.com", and the client has found the server of authority for the desired domain.

The problem with this normal behavior of named is that, at the last stage of iteration, when the client server gets the address of the victim DNS server, the client server's query to the victim server cannot get through. This query is delivered to the victim server's gateway, which is flooded with packets from the DDOS attack. So the client gets a very slow response, and the query may time out.

In this chapter I will show the "normal" route that a query would take through a client DNS server with no victim zone, and then what the client server would do if the victim's zone is present, but the NS records in the zone do not map to at least one proxy server address.

This is followed by an examination of the new behavior of the server when the victim's zone is present and NS records also have ALT-type addresses. New code, including additions to data structures, will be presented. This implementation creates an IP tunnel, which will also be described in some detail.

Normal Query Handling by Non-Authoritative DNS Server

When an application wants to communicate with a web site, it invokes a library routine called a resolver that issues a recursive query to its authoritative DNS server to retrieve the address of the desired web site. This resolver is usually referred to as a "stub" resolver because it relies on the name server to do all the work in resolving the address of a host name. The resolver in BIND is a full resolver but is built into the named server.

Assume a DNS query is issued from a client for the address of klingon.uccs.edu. The client's name server is not authoritative for the uccs.edu domain, nor does it have an answer cached from a previous query. Once it gets this query from the resolver, the client DNS server sends the query to one of the 13 root DNS servers on the Internet. In essence, these 13 name servers are authoritative for the "" domain, representing the root of all domains. One of these root servers responds with a list ("referral") of name servers that is authoritative for the "edu" domain.

If the reply isn't immediately received, the server may issue the query to another DNS server in the list. This behavior may be repeated until a response is received.

An authority for "edu" responds with a referral of DNS servers authoritative for "uccs.edu". The client sends the query to one of these DNS servers. The reply will contain either the answer or an error (e.g., "the desired host name does not exist"). Once the answer arrives the client's DNS server returns it to the resolver, which supplies the address to the calling application.

The behavior of named in handling this recursive query is illustrated below:

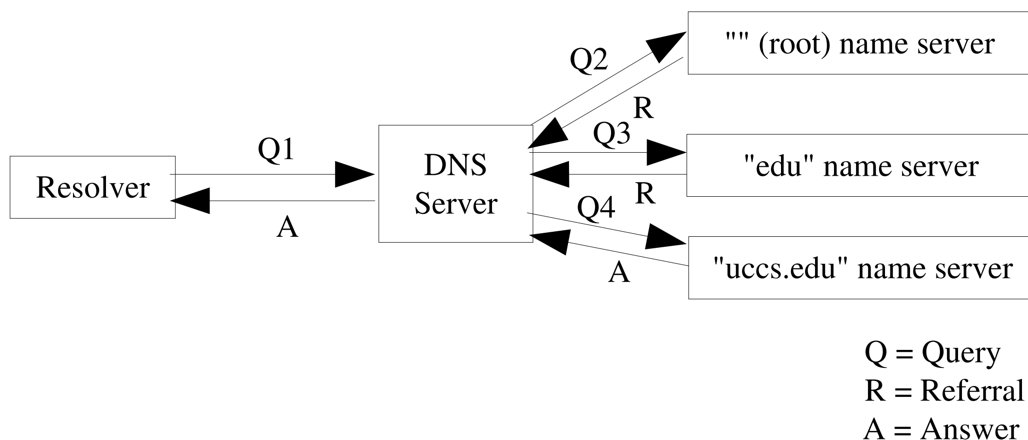


Figure 8.1: DNS server retrieving the IP address for "klingon.uccs.edu".

The following function flow diagram shows how BIND implements the behavior captured in Figure 8.1. Again, the query from the resolver is for the address of "klingon.uccs.edu". (Note: In my description of the server's actions, I refer to executing or repeating a "step." Here, a "step" refers to the branch of code starting from the previous *dispatch()* and ending before the next *dispatch()* is encountered.)

```
dispatch()
        Receive recursive query for address of "klingon.uccs.edu" from resolver on
        behalf of client application
dispatch()
        client_request()
                ns_query_start()
                        query_find()
                                query_getdb()
                                        Get database for "uccs.edu" zone.  It isn't there.
                                result = dns_db_find()
                                        Look for the root zone database.  It also isn't there.
                                switch (result) {
                                .
                                .
                                .
                                case ISC_R_NOTFOUND:
                                        Get the 13 root name servers from hints database (this is
                                        from the "db.cache" file)
                                /* fall through */

                                case DNS_R_DELEGATION:
                                        query_recurse()
                                                dns_resolver_createfetch()
                                                        fctx_create()
                                                                Create new resolver fetch data structure
                                .
                                .
                                .
                                }
dispatch()
        fctx_start()
                fctx_try()
                        fctx_query()
                                resquery_send()
                                        Send iterative query to one of the 13 root DNS servers
Always do this step twice

The next step is executed only if no replies have been received.
```

(dispatch()
    fctx_try()
        fctx_query()
            resquery_send()
                Send a query to another root DNS server

If still no reply, repeat this step, sending a query to a different root DNS server.
)
dispatch()
    udp_recv()
        A reply to one of the previous queries comes in, containing a list ("referral") of one or more authoritative DNS servers for the domain "edu". We want to query one of these DNS servers next.
dispatch()
    fctx_try()
        fctx_query()
            resquery_send()
                Send a query to one of the "edu" DNS servers listed in the last reply

If no reply to this query is received, repeat this step, sending a query to another authoritative server for the "edu" domain

dispatch()
    udp_recv()
        A referral containing one or more authoritative DNS servers for the "uccs.edu" domain is received
dispatch()
    fctx_try()
        fctx_query()
            resquery_send()
                Send a query to one of these "uccs.edu" name servers

If no reply to this query is received, repeat, sending a query to a different authoritative DNS server for the "uccs.edu" domain

dispatch()
    udp_recv()
        Receive answer
dispatch()
    query_resume()
        query_find()
                Returning from recursion; restore query context and resume
                result = event-->result;
                goto resume;
            resume:
                switch (result) {
                case ISC_R_SUCCESS:
                    break;
                .

                .

```
                    .
                    }
                cleanup:
                    query_send()
                        ns_client_send()
                            client_sendpkg()
                                Send answer back to resolver
dispatch()
    fctx_doshutdown()
dispatch()
    req_senddone()
dispatch()
    Ready for next query
```

Figure 8.2: named's normal handling of query for address of "klingon.uccs.edu".

New Query Handling by Server with Desired Zone Installed from Reroutemsg

After the client DNS server processes the reroutemsg and installs the new zone for the victim's domain, it no longer will handle queries for hosts on the victim's domain in the manner pictured above in Figure 8.1 or Figure 8.2. Now the server will answer such queries in a much different way.

Below is a function flow diagram showing the basic processing of a query for the address of the host "victim.victimnet.com", after the DNS server has installed the victimnet.com zone as a result of receiving and processing the reroutemsg.

```
dispatch()
    Receive query for address of host "victim.victimnet.com"
dispatch()
    client_request()
        ns_query_start()
            query_find()
                    query_getdb()
                        Get database for "victimnet.com" zone.  We have
                        it, so we're authoritative.

                    result = dns_db_find()
```



```
                        Look for host "victim" in "victimnet.com" database
```

```
resume:
        switch (result) {
        .
        .
        .
        case DNS_R_NXDOMAIN:
                Host "victim" does not exist in authoritative zone.
                Usually we would reply to the resolver with a
                "NXDOMAIN" ("host does not exist") response
                code.  Now, however, we do something different.

                result = dns_db_find()
                        Look for an NS resource record

                if (result == ISC_R_SUCCESS) {
                        There is at least one NS record in the
                        "victimnet.com" zone

                        result = dns_db_find()
                                See if at least one NS resource
                                record maps to a type A RR
                                in the "victimnet.com" database

                        if (result == ISC_R_SUCCESS) {
                                if (dns_rdataset_isassociated(rdataset) &&
                                dns_rdataset_isassociated(nsaltrdataset)) {
                                At least one NS record maps to
                                        BOTH A- and ALT-type records;
                                        this is a "reroute" zone

                                        result = DNS_R_DELEGATION;
                                        goto resume;
                                }
                        }
                }
        .
        .
        .
        }
resume:
        switch (result) {
        .
        .
        .
        case DNS_R_DELEGATION:
                Check cache for victim name and addresses; if
                found send reply; if not found, continue

                if (is_reroute_zone)
                        Create IP tunnel between this machine and
```



```
                        one of the proxy server addresses in
```

```
                                                  "victimnet.com" zone file; all queries now
                                                  will go through this IP tunnel

                                              query_recurse()
                                                  dns_resolver_createfetch()
                                                      fctx_create()
                                                          Create new resolver fetch data
                                                          structure
                                          .
                                          .
                                          .
                                          }
dispatch()
    fctx_start()
        fctx_try()
            fctx_query()
                resquery_send()
                    Send iterative query to one of the two victim name servers
```

The next step is executed only if no reply has been received.

```
(dispatch()
    fctx_try()
        fctx_query()
            resquery_send()
                Send a query to the other victim DNS server
```

If still no reply, repeat this step, sending the query to the first DNS server, then second, etc., alternating until either an answer is returned or the query times out

```
)
dispatch()
    udp_recv()
        Receive answer
dispatch()
    query_resume()
        query_find()
                Returning from recursion; restore query context and resume
                result = event-->result;
                goto resume;

            resume:
                switch (result) {
                case ISC_R_SUCCESS:
                    break;
                .
                .
                .
                }
            cleanup:
                query_send()
                    ns_client_send()
```



```
                                              client_sendpkg()
```

Send answer back to resolver
if (is_reroute_zone && ip_tunnel_up)
Destroy IP tunnel
dispatch()
fctx_doshutdown()
dispatch()
req_senddone()
dispatch()
Ready for next query

Figure 8.3: named's new handling of query for address of "victim.victimnet.com" with "victimnet.com" zone installed.

As is obvious from analyzing Figures 8.2 and 8.3, the query that is sent to the victim's DNS server through a proxy server is much faster than the usual way of querying successively closer DNS servers until the victim's DNS server is located.

A comparison of the elapsed time taken for the two approaches will be performed in Chapter 9.

Implementing IP Tunnel for Rerouting Query

In the algorithm shown in Figure 8.3, after it is determined that the victim's zone on the client DNS server has NS resource records that also map to both A- and ALT-type records, and the victim is not found in cache, an IP tunnel is created between the client DNS server and one of the ALT-type addresses of a SCOLD-aware proxy server. Any subsequent sending of a packet will automatically go through this tunnel to the specified proxy server, unless the proxy server has not created his end of the IP tunnel.

Both ends of the tunnel must be up for the packet to traverse it. If the proxy server on the other end has not set up his side, then the packet will be dropped by the proxy

server. The whole reason for using the proxy server in the first place is so that the proxy

server can reroute the query to the victim DNS server through a secret gateway unknown to clients, including the attacker.

In the SCOLD framework, the proxy servers whose addresses are sent in the reroutemsg to a list of client DNS servers will automatically set up their end of the IP tunnel with these DNS servers during an assault on the victim. So when a query from a client network arrives through the tunnel, it can be forwarded to the victim DNS server' s alternate gateway through another IP tunnel created between the proxy server and the alternate gateway. This setup is shown below in Figure 8.4, the DNS server testbed.

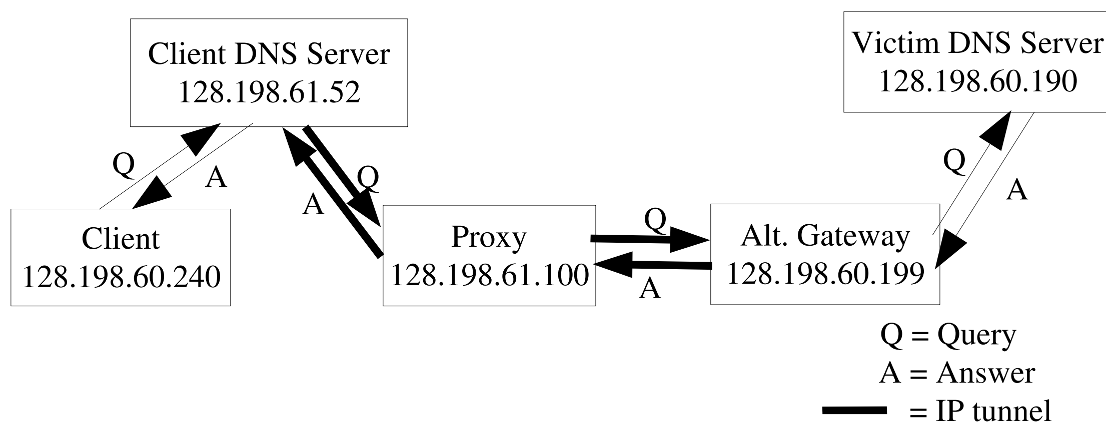


Figure 8.4: SCOLD Testbed for testing DNS server functionality.

Code of IP Tunnel Setup, Destroy Scripts

When the IP tunnels are created and destroyed, two scripts need to be created and executed. These are implemented in file query.c as two functions, *query_setupiptunnel* and *query_killiptunnel*, shown below in Figures 8.5 and 8.6.


```
static isc_result_t
query_setupiptunnel(ns_client_t *client, dns_rdataset_t *nsaltrdataset) {
```

```
isc_result_t result;
struct hostent *hostinfo = NULL;
char *hostname = NULL;
char name[256], address[256];
char **addrs;
int out, len;
char buf[256];
dns_rdata_t *nsrdata;
isc_sockaddr_t nssockaddr;
dns_rdata_in_alt_t nsrdata_struct;
char proxy_address[256];
char tunnelfile[256];

nsrdata = NULL;
result = dns_rdataset_first(nsaltrdataset);
if (result != ISC_R_SUCCESS)
        return ISC_R_FAILURE;

/* Get an ALT-type data record */
result = dns_message_gettemprdata(client->message, &nsrdata);
if (result != ISC_R_SUCCESS)
        return ISC_R_NOMEMORY;

dns_rdataset_current(nsaltrdataset, nsrdata);
result = dns_rdata_tostruct(nsrdata, &nsrdata_struct, NULL);
isc_sockaddr_fromin(&nssockaddr, &nsrdata_struct.in_addr, 53);

/* Convert ALT RR into string */
strcpy(proxy_address, inet_ntoa(nssockaddr.type.sin.sin_addr));

dns_message_puttemprdata(client->message, &nsrdata);

/* Get FQDN of current host */
result = gethostname(name, 255);
if (result != ISC_R_SUCCESS)
        return ISC_R_FAILURE;

hostname = name;

/* Get structure that contains addresses of host */
hostinfo = gethostbyname(hostname);
if (!hostinfo)
        return ISC_R_FAILURE;

if (hostinfo->h_addrtype != AF_INET)
        return ISC_R_FAILURE;

addrs = hostinfo->h_addr_list;

/* Retrieve first one in list */
```



```
strcpy(address, inet_ntoa(*(struct in_addr *)*addrs));
```

```
        /* ZONE_DIR is specified in /etc/named.conf */
        strcpy(tunnelfile, ZONE_DIR);

        /* Get exclusive access to new file for writing */
        out = open(strcat(tunnelfile, "/ip_tunnel.sh"),
                O_CREAT|O_WRONLY|O_TRUNC,
                S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH);
        if (out == -1)
                return ISC_R_IOERROR;
        if (flock(out, LOCK_EX) == -1)
                return ISC_R_IOERROR;

        /* Write script */
        write(out, "#!/bin/sh\n\n", 11);
        write(out, "#define variables\n", 18);
        write(out, "client_ip=", 10);
        len = strlen(address);
        write(out, address, len);
        write(out, "\nproxy_ip=", 10);
        len = strlen(proxy_address);
        write(out, proxy_address, len);
        write(out, "\ntunl=tunl1\n\n", 13);
        strcpy(buf, "#configure tunnel between client and proxy\n");
        len = strlen(buf);
        write(out, buf, len);
        strcpy(buf, "ip tunnel add $tunl mode ipip remote $proxy_ip dev eth0\n");
        len = strlen(buf);
        write(out, buf, len);
        strcpy(buf, "ifconfig $tunl $client_ip\n");
        len = strlen(buf);
        write(out, buf, len);
        strcpy(buf, "ip link set $tunl up\n");
        len = strlen(buf);
        write(out, buf, len);

        if (flock(out, LOCK_UN) == -1)
                return ISC_R_IOERROR;
        if (close(out) == -1)
                return ISC_R_IOERROR;

        /* Execute script */
        system(tunnelfile);
        return ISC_R_SUCCESS;
}
```

Figure 8.5: Script for creating IP tunnel on client DNS server.


```
static isc_result_t
query_killiptunnel(void) {
```

```
        int out;
        char tunnelfile[256];

        /* Open file for writing */
        strcpy(tunnelfile, ZONE_DIR);
        out = open(strcat(tunnelfile, "/ip_tunnel_kill.sh"),
                O_CREAT|O_WRONLY|O_TRUNC,
                S_IRUSR|S_IWUSR|S_IXUSR|S_IRGRP|S_IXGRP|S_IROTH|S_IXOTH);
        if (out == -1)
                return ISC_R_IOERROR;

        /* Get exclusive lock on file */
        if (flock(out, LOCK_EX) == -1)
                return ISC_R_IOERROR;

        /* Create script to destroy ip tunnel */
        write(out, "ifconfig tunl1 down\n", 20);

        if (flock(out, LOCK_UN) == -1)
                return ISC_R_IOERROR;
        if (close(out) == -1)
                return ISC_R_IOERROR;

        /* Execute script */
        system(tunnelfile);
        return ISC_R_SUCCESS;
}
```

Figure 8.6: Script for destroying IP tunnel on client DNS server.

When *query_setupiptunnel* is called in *query_find,* which is located in bin/named/query.c, a file called ip_tunnel.sh is created in the same directory as the zone files are kept, usually /var/named/. If the client DNS server's IP address is 128.198.61.52 and the proxy server's address is 128.198.61.100, this script would look like Figure 8.7 below.

```
#!/bin/sh

#define variables
client_ip=128.198.61.52
proxy_ip=128.198.61.100
tunl=tunl1
```



```
#configure tunnel between client and proxy
ip tunnel add $tunl mode ipip remote $proxy_ip dev eth0
```

```
ifconfig $tunl $client_ip
ip link set $tunl up
```

Figure 8.7: Script ip_tunnel.sh to set up IP tunnel.

Created in /var/named/ by *query_killiptunnel*, the script that destroys the IP tunnel created by *query_setupiptunnel* is comprised of one line:

```
ifconfig tunl1 down
```

Other Code Additions

In *query_find*

The code that is most important in implementing the new behavior of BIND as discussed in this chapter resides in bin/named/query.c, specifically in the function *query_find*, originally written by the Internet Software Consortium. Since this function is very large, I will display just the most important sections and my code additions. I will highlight these new lines of code in **bold**, as shown in Figure 8.8 below.

```
static void
query_find(ns_client_t *client, dns_fetchevent_t *event,
dns_rdatatype_t qtype) {
     dns_rdataset_t *rdataset, *trdataset, *nsrdataset, *nsaltrdataset;
     dns_rdataset_t *zaltrdataset, *nameservers;
     static isc_boolean_t is_reroute_zone, ip_tunnel_up;
     isc_buffer_t *dbuf, *nsbuf;
     dns_rdataset_t *altrdataset;
     dns_rdataset_t **altrdatasetp;
     dns_rdata_t *nsrdata;
     dns_name_t *nsname;
     dns_name_t *ns_fname;
     isc_region_t nsrdata_dataregion;
     .
     .
```



```
     .
     altrdataset = NULL;
     zaltrdataset = NULL;
     nsrdataset = NULL;
     nsaltrdataset = NULL;
```

```
    if (event != NULL) {
         /* Returning from recursion */
         altrdataset = event->altrdataset;
         .
         .
         .
         goto resume;
    }

    /*
     * Not returning from recursion.
     */
    is_reroute_zone = ISC_FALSE;
    ip_tunnel_up = ISC_FALSE;
    .
    .
    .
db_find:
    altrdataset = query_newrdataset(client);
    if (fname == NULL || rdataset == NULL || altrdataset == NULL) {
         QUERY_ERROR(DNS_R_SERVFAIL);
         goto cleanup;
    }
    .
    .
    .
    /*
     * Now look for an answer in the database.
     */
    result = dns_db_find(db, client->query.qname, version, type,
                   client->query.dboptions, client->now,
                   &node, fname, rdataset, sigrdataset,
                   altrdataset);
    .
    .
    .
resume:
    switch (result) {
    case ISC_R_SUCCESS:
         break;
    .
    .
    .
    case DNS_R_DELEGATION:
         authoritative = ISC_FALSE;
```



```
         if (is_zone) {
              .
              .
              .
              } else {
                   zaltrdataset = altrdataset;
                   .
                   .
                   .
              }
         } else {
                   query_putrdataset(client, &altrdataset);
                   altrdataset = zaltrdataset;
                   zaltrdataset = NULL;
              }
              if (RECURSIONOK(client)) {
                   if (is_reroute_zone) {
                        /* Set up IP tunnel between this machine and
```

```
proxy
                         * server, which will forward this query to the
target
                         * DNS server.
                         */
                        result = query_setupiptunnel(client,
nsaltrdataset);
                                    if (result != ISC_R_SUCCESS)
{
                             /* IP tunnel could not be established */
                                         QUERY_ERROR
(DNS_R_NXDOMAIN);
                                    goto cleanup;
                               }
                        ip_tunnel_up = ISC_TRUE;
                   }
     .
     .
     .
     case DNS_R_NXDOMAIN:
          INSIST(is_zone);

          /* Look for a nameserver record.  If present, and having type
'A' and 'ALT'
           * IP addresses, send a recursive query to this DNS server.
This is part
           * of reroute addition.
           */
          if (node != NULL)
               goto cleanup;
          nsrdataset = rdataset;
          rdataset = query_newrdataset(client);
               result = dns_db_find(db, fname, NULL, dns_rdatatype_ns,
                                       0, client->now, &node, fname,
                                       rdataset, NULL, NULL);

                 if (result == ISC_R_SUCCESS) {
               /* Although the desired host name was not found in the
zone file,
                * the file does have a nameserver record.  See if this
nameserver
                * also has mappings for type 'A' and 'ALT' IP addresses.
```



```
                */
               ns_fname = NULL;
               result = dns_message_gettempname(client->message,
&ns_fname);
                         if (ns_fname == NULL) {
                                   QUERY_ERROR(DNS_R_SERVFAIL);
                                   goto cleanup;
                              }
               nsbuf = NULL;
                    result = isc_buffer_allocate(client->mctx, &nsbuf,
NAMEBUF);
               if (nsbuf == NULL) {
                    QUERY_ERROR(DNS_R_SERVFAIL);
                    goto cleanup;
               }
                    dns_name_init(ns_fname, NULL);
                    dns_name_setbuffer(ns_fname, nsbuf);
                    dns_message_takebuffer(client->message, &nsbuf);

               result = dns_rdataset_first(rdataset);
               if (result != ISC_R_SUCCESS)
```

```
                goto cleanup;
           nsrdata = NULL;
           result = dns_message_gettemprdata(client->message,
&nsrdata);
           if (result != ISC_R_SUCCESS)
                goto cleanup;
           dns_rdataset_current(rdataset, nsrdata);
           dns_rdata_toregion(nsrdata, &nsrdata_dataregion);
           nsname = NULL;
           result = dns_message_gettempname(client->message,
&nsname);
           if (result != ISC_R_SUCCESS)
                goto cleanup;
           dns_name_fromregion(nsname, &nsrdata_dataregion);
           nameservers = NULL;
           nameservers = rdataset;
           rdataset = query_newrdataset(client);
           nsaltrdataset = query_newrdataset(client);
           nssigrdataset = NULL;
           nssigrdataset = query_newrdataset(client);
           if (node != NULL)
                dns_db_detachnode(db, &node);
           result = dns_db_find(db, nsname, NULL, dns_rdatatype_a,
                          0, client->now, &node, ns_fname,
                             rdataset, nssigrdataset,
nsaltrdataset);

           if (result == ISC_R_SUCCESS) {
                /* The nsname is in the database, and at least one
rdataset
                 * was found with it.  For us to send a query to
this name
                 * server, it must have mappings for BOTH type 'A'
and
                 * type 'ALT' IP addresses.
                 */
```



```
                if (dns_rdataset_isassociated(rdataset) &&
                    dns_rdataset_isassociated(nsaltrdataset)) {

                     /* Yes, both kinds of IP addresses are mapped
                      * to the nameserver.  This is a reroute zone,
                      * so we will send a query to this DNS server
                      * at one of these ALT-type addresses.
                      */
                     dns_message_puttempname(client->message,
&ns_fname);
                     dns_message_puttemprdata(client->message,
&nsrdata);
                     dns_message_puttempname(client->message,
&nsname);
                     query_putrdataset(client, &nssigrdataset);
                     query_putrdataset(client, &rdataset);
                     query_putrdataset(client, &nsrdataset);
                     rdataset = nameservers;
                     nameservers = NULL;
                     ns_fname = NULL;
                     is_reroute_zone = ISC_TRUE;
                     result = DNS_R_DELEGATION;
                     goto resume;
                }
           }

           /* Either no IP addresses mapped to the nameserver, or
```

```
only type 'A'
                * was found.  In either case, do not send a query to
this nameserver.
                */
               dns_message_puttempname(client->message, &ns_fname);
               dns_message_puttemprdata(client->message, &nsrdata);
               dns_message_puttempname(client->message, &nsname);
               query_putrdataset(client, &nssigrdataset);
               query_putrdataset(client, &nameservers);
                    query_putrdataset(client, &rdataset);
               rdataset = nsrdataset;
               nsrdataset = NULL;
               ns_fname = NULL;
               is_reroute_zone = ISC_FALSE;
               result = DNS_R_NXDOMAIN;
          } else {
                    query_putrdataset(client, &rdataset);
               rdataset = nsrdataset;
               nsrdataset = NULL;
               is_reroute_zone = ISC_FALSE;
               result = DNS_R_NXDOMAIN;
          }
     .
     .
     .
     }
     if (altrdataset != NULL)
          altrdatasetp = &altrdataset;
```



```
     else
          altrdatasetp = NULL;
     query_addrrset(client, &fname, &rdataset, sigrdatasetp, dbuf,
                DNS_SECTION_ANSWER, altrdatasetp);
     .
     .
     .
     query_addbestns(client, is_reroute_zone);
cleanup:
     if (altrdataset != NULL)
          query_putrdataset(client, &altrdataset);
     if (nsrdataset != NULL)
          query_putrdataset(client, &nsrdataset);
     if (zaltrdataset != NULL)
          query_putrdataset(client, &zaltrdataset);
     if (nsaltrdataset != NULL)
                    query_putrdataset(client, &nsaltrdataset);
     .
     .
     .
     } else if (!RECURSING(client)) {
          .
          .
          .
          query_send(client);
          if (is_reroute_zone && ip_tunnel_up)
               result = query_killiptunnel();
     }
}
```

Figure 8.8: Code additions (in **bold**) to function *query_find* in query.c.

Implementation of Caching ALT rdataset in Query Reply Message

Another segment of code I added to BIND implements caching of an ALT (type = 99) rdataset in a server that receives the answer to an iterative query. Now both A (type = 1) and ALT addresses are stored in cache, so that subsequent queries will pull all resource records up immediately without needing to send out another iterative query.

The code added is shown in **bold** below (Figure 8.9) in a function trace diagram. This diagram shows what the client server does when it receives a query reply that contains an ALT rdataset.

```
dispatch()
        fctx_try()
                fctx_query()
                        resquery_send()
                                Client server sends out iterative query
dispatch()
        udp_recv()
                Receive DNS message containing query reply
dispatch()
        resquery_response()
                .
                .
                .
                result = answer_response()
                        FOR all rdatasets in ANSWER section of DNS reply message DO
                                .
                                .
                                .
                                if (rdataset-->type == dns_rdataset_alt) {
                                        found = ISC_TRUE;
                                        aflag = DNS_RDATASET_ATTR_ANSWERALT
                                }
                                .
                                .
                                .
```

```
                    if (aflag == DNS_RDATASET_ATTR_ANSWER ||
                        aflag == DNS_RDATASET_ATTR_ANSWERALT)
                            have_answer = ISC_TRUE;
          .
          .
          .
          if (WANTCACHE(fctx)) {
               result = cache_message()
                    FOR msg section == ANSWER to ADDITIONAL DO
                         result = cache_name()
                              dns_rdataset_t *aaltrdataset;
                              aaltrdataset = NULL;
                              .
                              .
                              .
                              aaltrdataset = event-->altrdataset;
                              /* Contains ALT rdataset RRs */
```



```
                              .
                              .
                              .
                              FOR each rdataset attached to name DO
                                   .
                                   .
                                   .
                                   if (ANSWERALT(rdataset))
                                        addedrdataset = aaltrdataset;
                              .
                              .
                              .
                              result = dns_db_addrdataset()
                                   Add ALT rdataset to cache database
          }
          fctx_done()
dispatch()
     query_resume()
          query_find()
               Retrieve contents of DNS query reply message; send back to user
```

Figure 8.9: New code (in **bold**) in BIND implementing caching of ALT rdatasets retrieved from iterative query.

The code shown in Figure 8.9 relies on a few more additions to BIND, namely:

1) Adding

```
#define DNS_RDATASETATTR_ANSWERALT  0x2000
```

to lib/dns/include/dns/rdataset.h.

2) Adding

```
#define ANSWERALT(r) (((r)->attributes &
DNS_RDATASETATTR_ANSWERALT) != 0)
```

to /lib/dns/resolver.c.

3) Adding

```
dns_rdataset_t*        altrdataset;
```

to the *dns_fetchevent* structure (or *dns_fetchevent_t* type) defined in lib/dns/include/dns/resolver.h.

4) In adding the new altrdataset field to *dns_fetchevent_t*, I also had to add altrdataset to the parameter list of the following functions located throughout BIND:

```
dns_resolver_createfetch()
fctx_join()
dns_validator_create()
```

5) Because of the similarity of the *dns_fetchevent_t*, *dns_lookupevent_t*, and *dns_validatorevent_t* structures, and because all three data structures use some of the same functions, it required that **dns_rdataset_t *altrdataset** be added to the latter two structures as well. *dns_lookupevent_t* is defined in lookup.h; *dns_validatorevent_t* is defined in validator.h. These additions also necessitated adding code to lookup.c and validator.c for properly initializing and checking these variables.

Implementation of Retrieval of ALT rdataset in Zone Database or Cache Database

The previous section showed the implementation of caching ALT resource records retrieved from an iterative query reply. However, the original BIND v. 9 code does not

allow a subsequent query to retrieve these cached ALT addresses. Similarly, the original BIND code also cannot retrieve ALT addresses for a host in a zone file, which is stored in memory in a separate database. This section illustrates the code added to BIND for extracting ALT RRs from the cache and zone databases.

When the named daemon is booted up all the server's zone files are stored in memory in the zone database. This database is a red-black tree where each node of the tree is a zone. Descending into one of these nodes, the resource records contained therein

are arranged in another red-black tree structure. So in the zone database there exists one or more zones, each of which is comprised of a database of records.

Another database, the cache database, contains a tree of all domain names with associated resource records that have been cached after the server receives the answer to an iterative query.

The following trace of code shows, in **bold**, what was added to BIND to enable the server to retrieve an ALT rdataset that exists in a zone for which the server is authoritative. In this example, the query is for "grace.csnet.uccs.edu", and the server is authoritative for the "csnet.uccs.edu" domain.

query_find()
    query_getdb()
        Look for "csnet.uccs.edu" database. We have it.
    dns_db_find(..., **altrdataset**);
        Require that altrdataset is either NULL, or it is valid and all its fields are NULL
        return ((db-->methods-->find)(..., **altrdataset**));
            zone_find()
                rdatasetheader_t *altheader;
                .
                .
                .
                result = dns_rbt_findnode();

```
        Search "csnet.uccs.edu" for "grace"
    .
    .
    .
    found:          /* Found node */
        for (header = node-->data;
             header != NULL;
             header = header_next) {
                .
                .
                .
                else if (header-->type == dns_rdataset_alt)
                        altheader = header;
        }
```



```
        .
        .
        .
        if (altheader != NULL)
                bind_rdataset(..., altheader, ..., altrdataset);
                        Get ALT rdata records from "grace" node
                        and attach to altrdataset

/* Return to query_find after call to dns_db_find() */                    .
.
.
.
if (altrdataset != NULL)        /* altrdataset contains ALT rdata RRs */
        altrdatasetp = &altrdataset;
.
.
.
query_addrrset(client, ..., DNS_SECTION_ANSWER, altrdatasetp);
        /* This function adds ALT rdataset to query reply (in client-->message) */
        dns_rdataset_t *altrdataset;
        .
        .
        .
        if (altrdatasetp != NULL)
                altrdataset = *altrdatasetp;
        else
                altrdataset = NULL;

        Add "grace.csnet.uccs.edu" name structure to ANSWER section of client-->message
        .
        .
        .
        query_addrdataset(client, ..., rdataset);
                /* Attach type A (= 1) rdataset to name structure here */
```

```
        if (altrdataset != NULL && rdataset_isassociated(altrdataset))
                Append altrdataset to type A rdataset, which is attached to name structure

/* Back to query_find */
Add name servers (type NS = 2 RRs) to AUTHORITY section of
client-->message, and add their IP addresses to ADDITIONAL section
(client-->message contains the RRs for the query reply)
.
.
.
query_send()
        ns_client_send()
                client_sendpkg()
                        Send query reply to client application
```

Figure 8.10: Code (in **bold**) that enables retrieval of ALT rdataset from zone database.

In Figure 8.10, the client's DNS server was authoritative for the "csnet.uccs.edu" domain, so the call to *query_getdb* at the start of *query_find* retrieves the "csnet.uccs.edu" zone. Then the subsequent call to *dns_db_find* looks for the node "grace" in this database. Since the server found the desired domain in the database, the function that is called by *((db-->methods-->find)(..., **altrdataset**))* in *dns_db_find* is *zone_find*.

When the server is not authoritative for the desired domain, then the call to *query_getdb* yields nothing. This tells the server that the desired node is not in the binary search tree containing its zones. However, it can still search another tree which stores resource records cached from previous queries. In this circumstance, when *dns_db_find* is called in *query_find*, the *((db-->methods-->find)(..., **altrdataset**))* function calls *cache_find*, not *zone_find*.

The code in *cache_find* that retrieves ALT records is virtually identical to that in *zone_find*. The code that appends the ALT rdataset to the A rdataset, which is connected to the name structure in the client-->message ANSWER section, is the same as for retrieval of ALT rdatasets from the zone database. Therefore, tracing the code used for

retrieving ALT resource records from the cache database is mostly redundant, and is omitted.

Successfully implementing the code additions illustrated by Figure 9.10 required one extra minor addition:

- Added

```
dns_rdataset_t *altrdataset;
```


to the last spot in the parameter list of method *(*find)* in the declaration of *typedef struct dns_dbmethods* in lib/dns/include/dns/db.h. This method is used by both *zone_find* and *cache_find*, which also have added altrdataset to the last position in their parameter lists.

CHAPTER IX

RESULTS OF TESTING ENHANCED BIND ON SCOLD TESTBED

Using the testbed as illustrated in Figure 9.1 below, I tested the efficiency of the nsreroute program. Specifically, it shows:

1) How long it takes the nsreroute program on proxy server (128.198.61.100) to find authoritative servers for 1, 10, 25, and 50 different clients in a list (using mostly domain names of companies on the Internet). The result is shown in Figure 9.2.
2) How long it takes the proxy server and client DNS server (128.198.61.52) to process a reroutemsg. The client DNS server daemon starts timing after it has parsed the DNS message it has received, determines it has an opcode of 7 (the reroutemsg opcode), and enters *ns_reroute_start* in reroute.c. It stops timing in the procedure *reroutedone_action* in reroute.c, after it has issued a signal to reload zones. The proxy

server starts timing as soon as it begins executing, and stops timing immediately before the child process exits in request.c. See Figure 9.3 for the results.

I also tested differences in query times (using *dig*) from the client to the client DNS server for the address of target.targetnet.csnet.uccs.edu. The two cases I tested were:

1) Client DNS server does not have the zone for "targetnet.csnet.uccs.edu", so query must start at root and eventually be directed to the target DNS server through normal gateway (128.198.60.129). The results are shown in Figure 9.4.

2) Client DNS server has installed the zone for targetnet.csnet.uccs.edu as a result of receiving the reroutemsg from the proxy. The query from the client now is rerouted through the proxy to the alternate gateway (128.198.60.199), and on to the target DNS server. This route avoids the main gateway at 128.198.60.129, which would be bombarded in the event of a DDOS attack. See Figure 9.5 for these times.

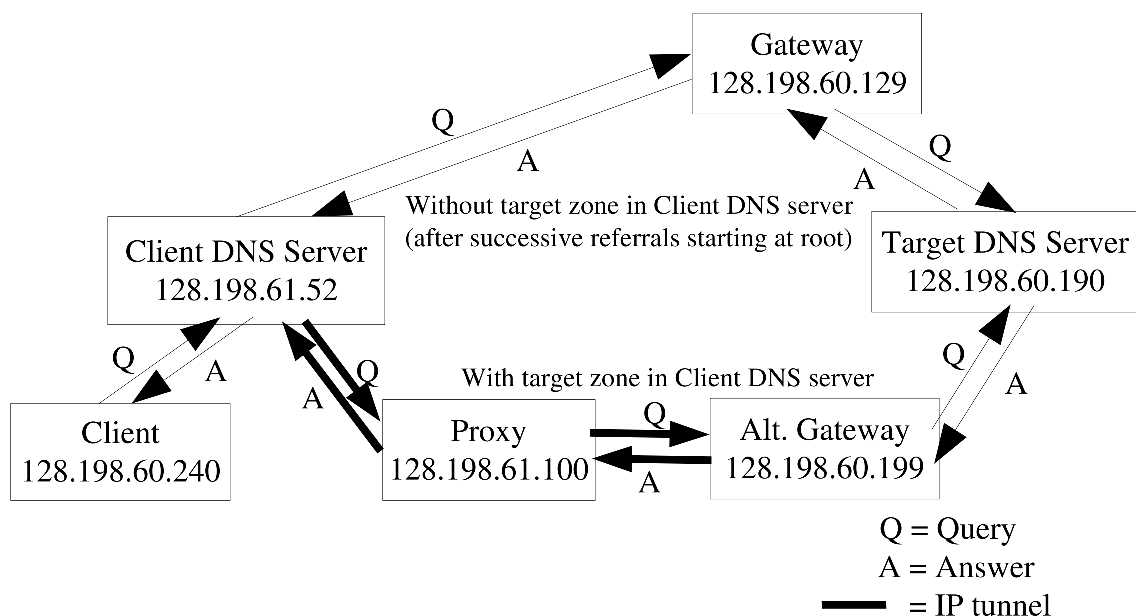


Figure 9.1: SCOLD Testbed for testing DNS reroutemsg and queries from Client

Time (in seconds) for nsreroute to Get Authoritative DNS Servers for Clients in Input List				
Trial	Number of Clients in List			
	1	10	25	50
1	0.20	20.03	89.48	96.61
2	0.18	20.65	92.10	398.43
3	0.19	19.81	91.89	410.23
4	0.20	19.44	91.28	201.49
5	0.18	16.08	88.17	428.56
6	0.19	20.06	326.80	444.26
7	0.15	20.14	95.19	83.51
8	0.71	25.35	90.99	109.26
9	0.22	19.66	94.12	171.55
10	0.17	23.26	95.18	60.14

Figure 9.2

Time (in seconds) for Proxy Server and Client DNS Server to Completely Process reroutemsg		
Trial	Proxy Server	Client DNS Server
1	2.97	2.55

Time (in seconds) for Proxy Server and Client DNS Server to Completely Process reroutemsg		
2	2.36	2.22
3	2.33	2.17
4	2.30	2.16
5	2.38	2.15
6	2.33	2.15
7	2.29	2.17
8	2.35	2.15
9	2.31	2.19
10	2.29	2.19

Figure 9.3

Time (in seconds) for Query from Client to Client DNS Server for Address of target.targetnet.csnet.uccs.edu. (Client Server does not have target's Zone.)	
Trial	Time
1	0.62
2	2.53
3	2.41
4	2.26
5	4.44
6	14.50
7	0.28
8	0.45
9	4.40
10	12.32

Figure 9.4

Time (in seconds) for Query from Client to Client DNS Server for Address of target.targetnet.csnet.uccs.edu. (Client Server has target's Zone from reroutemsg.)	
Trial	Time
1	0.33
2	0.29
3	0.15
4	0.13
5	0.22
6	0.22
7	0.22
8	0.22
9	0.14
10	0.22

Figure 9.5

Comparisons of Times in nsreroute and Query Tests

From Figure 9.2, it is evident how the nsreroute program gets disproportionally slower as more clients are added to the input list. With one client in the input file, the average time (throwing out the outlier 0.71) to get all the client's authoritative name servers is only 0.19 seconds. But increasing the number of clients by a factor of 10 results in an increase of about a factor of 100 for the elapsed time. Then increasing the number of clients 2 ½ times (to 25) results in approximately a 4 ½-time increase in total time (to about 92 seconds). Doubling the number of clients in input from 25 to 50 again results in about a 2.6-time increase in elapsed time to about 240 seconds average.

It does seem that, however, as more clients are added to the list, nsreroute gets

more efficient in processing the input.

Before analyzing the processing times in Figure 9.3, it is necessary to show the code added to BIND that is responsible for calculating these times. The server's processing time was started in the function *ns_reroute_start* with the following two lines:

```
rtn = gettimeofday(&tp, NULL);          /* get current time */
t1 = (double)tp.tv_sec+(1.e-6)*tp.tv_usec;
```

The server's stop time was calculated in reroute.c in the function *reroutedone_action* by adding the following lines of code:

```
rtn = gettimeofday(&tp, NULL);          /* get current time */
t2 = (double)tp.tv_sec+(1.e-6)*tp.tv_usec;
elapsed = t2 - t1;
printf("server processing time is %g seconds\n", elapsed);
```


For the proxy server, which sends the reroutemsg, the start time is calculated in function *main* in bin/nsreroute/nsreroute.c with the code

```
rtn = gettimeofday(&tp, NULL);          /* get current time */
t1 = (double)tp.tv_sec+(1.e-6)*tp.tv_usec;
```

The stop time for the client nsreroute program is calculated in lib/dns/request.c in the function *req_senddone* by adding the following four lines:

```
rtn = gettimeofday(&tp, NULL);          /* get current time */
t2 = (double)tp.tv_sec+(1.e-6)*tp.tv_usec;
elapsed = t2 - request->start_time;
printf("pid %d: total program time is %g seconds\n", getpid(), elapsed);
```

The field *start_time*, type *double*, was also added to the *dns_request_t* structure. In *send_reroute* in nsreroute.c, the start time for running nsreroute is saved in the *request* variable with the following line:

```
dns_request_start_time(&request, t1);  /* store program start time */
```

Figure 9.3 shows that the time for the proxy server to completely process an input file of one client is about 2.33 seconds, which is a little slower than the total time (about 2.17 seconds) the client DNS server expends. This makes sense, since we're not taking into account the time the client server spends in receiving the message from proxy and parsing it, and any other shutdown routines that the client server typically calls after processing any DNS message, including timer cleanup, memory deallocation, etc.

The final two tables (Figures 9.4 and 9.5) show a substantial difference between query times when the client DNS machine does not have the zone of the target (or victim) domain loaded into memory, and when it does. When it does not have the target zone, a query sent from the client must be sent to a root DNS server first, and after successive

iterations the client DNS server finally queries the target DNS server directly. For our experimental testbed, observing 10 trials yielded an average query time of 4.42 seconds.

The wide variation in times observed in Figure 9.4 is due to the fact that the authoritative server for the uccs.edu domain, klingon (at 128.198.1.250), had inaccurate name server referrals in its uccs.edu zone file when these trials were conducted. All times over 1 second result from klingon referring the client DNS server to one of these out-of-date DNS servers. These dubious DNS servers took a lot of time referring client DNS back to klingon, which eventually chose the right name server for csnet.uccs.edu – gandalf. The times in trials 1, 7, and 8 result when all DNS servers being referred to the client DNS machine are correct. This represents an average time of about 0.45 seconds.

I kept the trials that included querying these out-of-date name servers in Figure 9.4

because I wanted to show that sometimes a DNS server in the chain of referred name servers will have incorrect information, and demonstrate how that impacts on the overall query time. Obviously the impact can be quite significant.

When the client DNS server has installed the target zone after receiving and processing the reroutemsg, the average time for a query for the target machine is only 0.21 seconds. This is about 21-times better than the average evidenced by Figure 9.4. Even when times resulting from incorrect referrals are eliminated from the average, the average time of a query when the client DNS server has installed the rerouting zone is still better than half, 0.21 seconds vs. 0.45 seconds.

CHAPTER X

CONCLUSIONS

The new enhanced BIND v. 9.2.2 has some great features. These include:

- An nsreroute program that remotely installs victim zones on client DNS machines
- New zones help clients communicate with another member of the SCOLD consortium that is under attack
- Intrusion tolerance works
- New ALT data type results in faster queries over multiple dynamic paths

- Multiple-path routing capability results in larger aggregate bandwidth for the server

Lessons Learned

I spent over a year on this masters thesis, from understanding DNS BIND to tracing the BIND code, and finally to implementing changes in BIND that were necessary as part of the SCOLD defense system. No part of this was easy for me, as I have no prior experience in programming other than a summer job and what I've learned in school. I began research on this topic in an introductory graduate-level computer communications course.

In researching DNS BIND and OpenSSL I've had to overcome a few difficulties. These include:

- Unable to obtain help from the Internet Software Consortium (ISC).
- Understanding ISC's BIND code. Most of the code that must be used inside BIND should come from ISC's own functions. A lot of these are wrappers around standard C functions, but otherwise they are absolutely essential to properly manipulating the data structures inside BIND. Without using them, BIND does not account for memory allocations correctly, and the program will crash or be unstable. Also, coding would be much harder without having these functions at one's disposal. I printed a stack of BIND header files (rdata.h, rdataset.h, message.h, request.h, server.h, client.h, etc.) for handy reference as I implemented my code.
- "Clock skew" problems. Installing the enhanced version of BIND v. 9 onto the machines in the computer science lab sometimes resulted in warning messages of

"clock skew" during the compilation process. This had to do with the lab computers' clock not being set to a reasonable time (usually a few days behind), which conflicted with the correct timestamp on many files in the BIND software (which had been modified on my machine at home). This condition dramatically increased BIND's compile time. Clock skew also interfered with the authentication between the proxy server and the client DNS server. Their certificates were considered invalid by OpenSSL because the "good from" dates were set in the future from the machine's perspective. Clock skew was easily remedied by setting the correct time on the computers (using "ntpdate <IP address of NTP server>") and putting this line in the crontab file that would synchronize the clock with the NTP server once every hour.

- Testing and tracing difficulties. These include:
  - Timeouts. Tracing the normal execution of a dig query from the client or the server side would be nearly impossible with the short timeout values originally in the code. I found these constants and made them much bigger, as follows:

```
#define TCP_TIMEOUT 100000  /* was 10 */
#define UDP_TIMEOUT 100000  /* was 5 */
#define SERVER_TIMEOUT 100000  /* was 1*/
```

  - Tracing named daemon. For over half a year I was not able to fully understand the workings of the BIND server daemon, because of its nature (it isn't a program to be started up; it is already running). Finally I understood that I needed to "attach" the GNU GDB debugger to the server process.

- Printing to screen. BIND v. 9 has code in bin/named/unix/os.c in the function *ns_os_daemonize* that closes standard output. I commented this out.
- Caching "NS 128.198.60.194". During testing on the SCOLD testbed I sent out a query for a machine in the csnet.uccs.edu domain, and received an answer from gandalf, including gandalf's NS record (for the name server). Subsequent queries for other machines in the csnet.uccs.edu domain, however, yielded a response with the error code "SERVFAIL", which means the DNS server does not respond. I realized it was because in gandalf's zone file for the csnet.uccs.edu domain, somebody had written "NS 128.198.60.194", which is incorrect. (*RFC-1035* [Moc87] defines the NS type as "NS <host name>", not "NS <IP address>". Another record, "<host name> A <IP address>", maps the NS record to an address.) When I received the reply for the first query answered by gandalf, my machine was

  caching the name server "128.198.60.194" as a host name, not IP address. So the next query for a host in the csnet.uccs.edu domain was sent to the "128.198.60.194" DNS server, which doesn't exist.

- Forking in server. To keep the server free to answer queries while it is processing a reroutemsg, I tried to implement a call to *fork* before authenticating the client. The child process would take care of authentication and the rest of the processing while the parent process continues to answer queries. After the child process finishes writing a new zone file to disk, it reloads the zones into memory and dies. The problem is that the new zone is only known to the child process. The parent process is not aware of the new zone at all. So when the child process dies, the parent process still does not

know about the new zone for the victim. The parent cannot reload zones while the child is writing the new zone to disk; the victim zone may not be included or may only be partially included. Maybe after waiting a certain amount of time the server could issue a signal to reload zones. Because of this issue I took the fork call out.

- "Dereferencing pointer to incomplete type." This compilation error message was very hard to figure out, but I finally determined that it was because I was trying to assign a value to a private field in a data structure in a file other than the source file where the data structure was defined. I solved this problem by creating a new function in the structure's source file and performing the necessary assignments there.
- DNS referrals. The main UCCS DNS server, klingon.uccs.edu, up until only a few days ago would not refer queries for hosts in csnet.uccs.edu to gandalf or vinci. klingon either tried to answer them or replied with a NXDOMAIN ("not in domain")

error message. This problem made it harder to properly test the behavior of queries. To get gandalf to refer queries for the "targetnet" subdomain, I added the lines "targetnet NS targetdns.targetnet.csnet.uccs.edu" and "targetdns.targetnet A 128.198.60.190" to gandalf's csnet.uccs.edu.zone file.
- Threading. Before implementing forking in the client (and unsuccessfully implementing it in the server) I experimented with using threading. To enable threading BIND must be configured with the "--enable-threads" option. I tried this but it slowed execution times drastically, and was very hard to trace.
- Robustness. I spent a good deal of time trying to make the nsreroute program immune from bad input, including illegal IP addresses, not enough IP addresses, too few

addresses, not enough host names, etc. I think I was successful in this respect. If it encounters a problem with a command, it issues an error message and moves to the next command.

- Reliability of my OpenSSL application code. For a long time I had problems getting client and server to authenticate every time. Maybe every 4th or 5th authentication would fail with an unidentified error code. I re-read my OpenSSL books ([VMC02] and [Res01]) and added code from [VMC02] (which was not in [Res01]) to both ssl_authenticate_client and ssl_authenticate_server. Now authentication succeeds every time. This includes the following lines added to various places in *ssl_authenticate_client,* located in openssl_server.c:

```
RAND_cleanup();
SSL_set_accept_state(ssl);
ERR_remove_state(0);
```


Similarly, the following lines were added in various places in *ssl_authenticate_server*, located in openssl_client.c:

```
RAND_cleanup();
ERR_remove_state(0);
```

Future Work

In creating this addition to the DNS BIND v. 9.2.2 software, I tried my best to make the new code work as flawlessly as possible. After testing it innumerable times, I think it does exactly what it should, and never crashes. That being said, the enhanced BIND software package needs to be subjected to tests that utilize other, more advanced aspects of BIND, such as:

- TSIG and DNSSEC security features
- DNS Dynamic Update
- DNS Notify
- Using different views
- Adding different kinds of resource records to a DNS server's zone file (e.g., CNAME, MX, etc.) and testing queries to this server

In general, the enhanced BIND software described in this paper could possibly be improved in the following ways:

- Improve speed of nsreroute program in getting NS records for the client domains from the input file.

- Reduce connection time between the client DNS server and the sender of reroutemsg. Examine ways to remove call to *sleep(1)* in *ssl_authenticate_server* while maintaining program reliability.
- Fork DNS server and make the parent process reload zones after a certain period.
- For nsreroute program, only retrieve the primary master name server's SOA record,

instead of numerous NS records. Then send reroutemsg to just this DNS server. Rely on the zone transfer from the primary master DNS server to the zone's other name servers, and analyze tradeoff in time consumed.

BIBLIOGRAPHY

[AL01] Paul Albitz and Cricket Liu. *DNS and BIND*. Fourth Edition, O'Reilly & Associates, Inc., April 2001.

[Cha03] Ben Charny. *U.S. shrugs off world's address shortage.* At http://news.com.com, July 28, 2003.

[Cho03] C. Edward Chow. “UCCS Network Security Research.” At http://cs.uccs.edu/~chow/research/security/uccsSecurityResearch.ppt, 2003.

[Del03] Michelle Delio. *Geeks Grapple With Virus Invasion.* At http://www.wired.com, Aug. 21, 2003.

[DGLRS02] David Durham, Priya Govindarajan, Dylan Larson, Priya Rajagopal, Ravi Sahita. *Elimination of Distributed Denial of Service Attacks using Programmable Network Processors, Version 1.0.* Intel Corporation, June 2002.

[Dit99a] David Dittrich. *The DoS Project's "trinoo" distributed denial of service attack tool.* University of Washington, at http://staff.washington.edu/dittrich/misc/trinoo.analysis, Oct. 21, 1999.

[Dit99b] David Dittrich. *The "stacheldraht" distributed denial of service attack tool.* University of Washington, at http://staff.washington.edu/dittrich/misc/stacheldraht.analysis, Dec. 31, 1999.

[Gib02] Steve Gibson. *DRDoS: Distributed Reflection Denial of Service.* At http://grc.com/dos/drdos.htm, Feb. 22, 2002.

[Gro03] Grant Gross. *Cybersecurity legislation may go to Congress.* At http://www.computerworld.com, Sept. 4, 2003.

[HGEK03] Steve Hamm, Jay Greene, Cliff Edwards, and Jim Kerstetter. *Epidemic: Crippling computer viruses and spam attacks threaten the information economy. Can they be stopped?* At http://www.businessweek.com, Sept. 8, 2003.

[Jel03] Sarah Jelinek. *IP Traceback and IDIP.* Masters thesis proposal, University of Colorado at Colorado Springs, 2003.

[KM02] Brian Krebs and David McGuire. *Attacks Exposed Internet's Vulnerabilities.* At http://www.washingtonpost.com, Oct. 31, 2002.

[Kre02] Brian Krebs. *Root-Server Attack Traced to South Korea, U.S.* At http://www.washingtonpost.com, Oct. 31, 2002.

[Moc87] Paul Mockepetris. *RFC-1035: Domain Names – Implementation and Specification*. Network Working Group, Nov. 1987.

[Res01] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2001.

[SANS00] SANS Institute. *Consensus Roadmap for Defeating Distributed Denial of Service Attacks*. At http://www.sans.org, Feb. 23, 2000.

[Sko02] Ed Skoudis. *Counter Hack*. Prentice-Hall, Inc., 2002.

[SPS02] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, Inc., 2002.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[Ste98] W. Richard Stevens. *UNIX Network Programming, Volume 1*. 2nd Edition, Prentice-Hall, Inc., 1998.

[VMC02] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly & Associates, Inc., 2002.

[Wil00] Martyn Williams. *eBay, Amazon, Buy.com hit by Attacks*. At htt://www.cw.com.hk, Feb. 10, 2000.

APPENDIX

USER'S GUIDE TO INSTALLING ENHANCED BIND ONTO SCOLD VIRTUAL MACHINES

Figure A.1 below shows the SCOLD testbed:

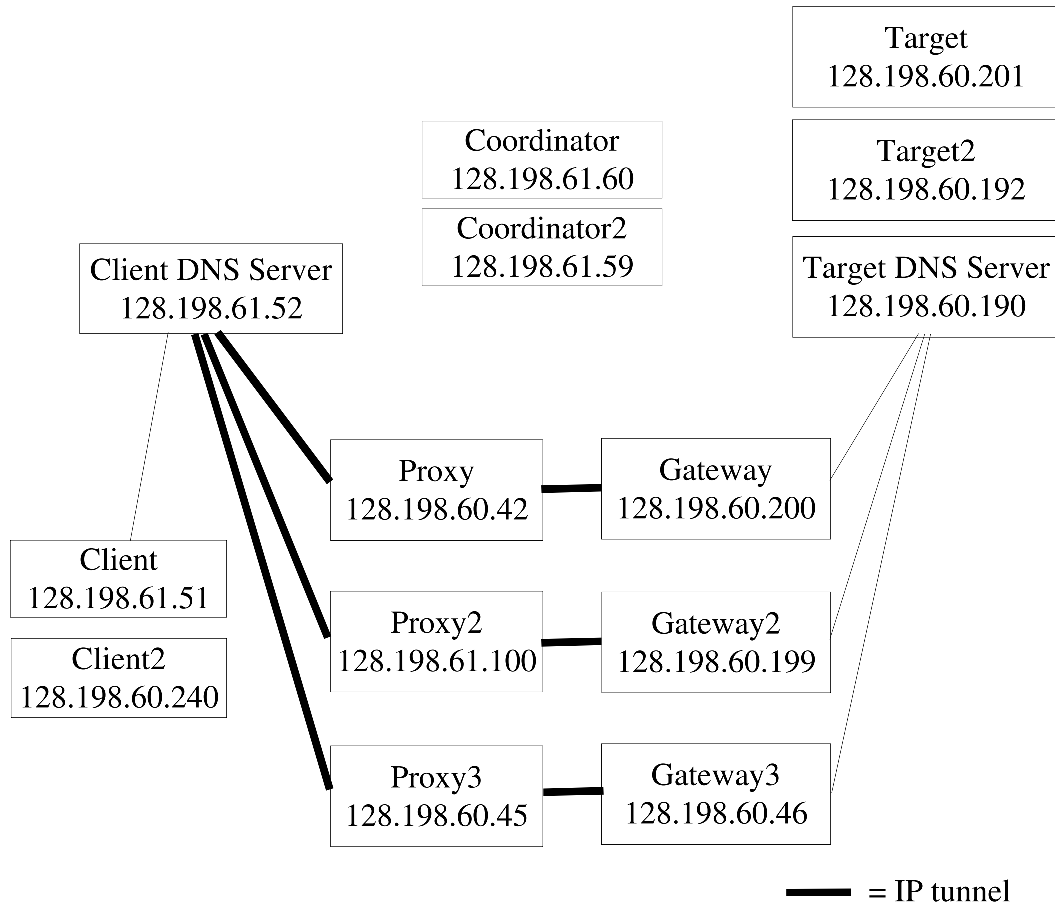


Figure A.1: SCOLD Testbed.

The machines in the SCOLD testbed are actually virtual machines running on the host machine athena.uccs.edu. To boot up the SCOLD virtual machines, execute ./ *run_vmware.sh* in the /root/ directory of athena. It will take 5-10 minutes for all machines to boot up.

To install enhanced BIND onto a SCOLD virtual machine, follow these steps:

- As root, log into the machine using ssh.
- If the directory /root/sdns/ does not exist, create it. Download the enhanced BIND software package bind-9.2.2-scold.tar.gz, located at

  http://cs.uccs.edu/~chow/pub/master/dbwilkin/src/

  and move the software package to /root/sdns/.
- Unzip the file using the command

  gunzip bind-9.2.2-scold.tar.gz
- Extract the bind-9.2.2 directory from the resulting tar file using the command

  tar xf bind-9.2.2-scold.tar
- Move to /root/sdns/bind-9.2.2/bin/nsreroute/ and copy Makefile from make_file:

  cp make_file Makefile

  Note: This step is needed so that nsreroute.c will be compiled during the *make* process. I spent a good deal of time trying to figure out how to get BIND to automatically add a Makefile to the bin/nsreroute/ directory during the *configure* step (shown next), but I was not successful.

- Move to /root/sdns/bind-9.2.2 and configure BIND:

  ./configure –with-openssl
- Run the following commands in order:

  make

  make depend

make install

If installing BIND onto either client DNS server or target DNS server, the named script file located in the /etc/init.d/ directory needs to be modified in the following way:

- Comment out the line

  # named -u named ${OPTIONS}

- Add the line

  /usr/local/sbin/named

The directory /var/named/, as well as all files residing there, need to be owned by root. This can be accomplished by executing the command

chown root named

in the /var/ directory, and executing the command

*chown root **

in the /var/named/ directory.

Executing the following command starts up the named server daemon:

/etc/init.d/named restart

Samples files, including named.conf, db.csnet.uccs.edu , db.targetnet.csnet.uccs.edu, and db.clientnet.csnet.uccs.edu, are all included in the src archive at

http://cs.uccs.edu/~chow/pub/master/dbwilkin/src/

Copy named.conf to the /etc/ directory, and copy db.csnet.uccs.edu, db.targetnet.csnet.uccs.edu, and db.clientnet.csnet.uccs.edu to the /var/named/ directory.