Global Information Assurance Certification Paper

Understanding Buffer Overflow Exploits

Examining the BIND 8.2 NXT Overflow Exploit

Submitted for the SANS/GIAC
Advanced Incident Handling and Hacker Exploits
Practical Assignment Option #2

by

Michael V. Pelletier

EXPLOIT DETAILS

Name:

DNS NXT Record Overflow
a.k.a. the "ADMROCKS" bug.

Variants:

t666.c - main published exploit

Operating System:

Any OS running BIND versions 8.2, 8.2.1, or 8.2.2. OS' that shipped with a vulnerable named include:

- FreeBSD 3.2
- NetBSD 1.4.1
- OpenBSD 2.6
- RedHat Linux 6.0 and 6.1

The shell-creating exploit is OS-specific, due to the need for properly constructed assembly language to be written in the overrun, but any version of BIND can be corrupted and crashed by feeding bogus data.

Protocols/Services Used:

The Domain Name Service protocol and daemon.

Brief Description:

BIND version 8.2 introduced new features for establishing cryptographic security between master and slave nameservers, and their resolvers. One such feature, the NXT record (originally specified in RFC-2065, updated in RFC-2535), was not subjected to proper bounds checking. and a malformed NXT record response to a query by an affected nameserver was capable of overrunning the end of an allocated memory buffer for the storage of this response, and either corrupting the running executable or inserting malicious code that will execute with the priveleges of the owner of the namserver process.

DESCRIPTION OF THE DNS PROTOCOL

The Domain Name Service protocol is arguably the single most important service on the internet. It puts a human-usable face on the stream of numbers that makes up the internet's host addresses, providing a broadly-distributed approach to translating human-readable host names into their computer-readable IP addresses.

Originally defined by RFC-883 by P. Mockapetris in 1983, this service grew out of the increasing impracticality of maintaining a single large "/etc/hosts" file, containing a list of all the hostnames and IP addresses on one's local network and the Internet. Insuring timely updates with a few computers on a local network was one thing, but spreading this information to the growing internet of thousands of computers was entirely another, and it was clear that it would only get worse as the worldwide Internet grew.

Where the old-fashioned, centrally-managed /etc/hosts file was a "flat-file" database, the DNS protocol set forth a way to build an entirely distributed, locally-managed database of hostname and IP address mappings. The authority for each segment of the database was defined by each portion of the domain name. Beginning with the "root" domain, represented by an implicit trailing "dot" on all domain names, authority is delegated to each subdomain in turn, with increasing specificity, right down to the individual hostname.

So, for example, in order to look up the IP address for the hostname "arrakis.engin.umich.edu," one's local nameserver would first query the root nameserver. Working from right to left, this root server would return the authoritative nameservers for the "edu" domain as resource records of type "NS." The response will also typically contain the "A" address records for the list of nameservers. A query to one of the "edu" servers would return the authoritative servers for the "umich.edu" domain, again in NS records. A successive query to one of the "umich.edu" servers would then return the authoritative servers for the "engin.umich.edu" domain, and finally, a query to one of the servers in that final list of NS records would return the "A" or "address" record for "arrakis.engin.umich.edu."

In order to avoid the considerable network traffic and delays that would result from each and every hostname lookup going through the above process, each record has a "time-to-live" or "TTL" value associated with it. This is the maximum time that a nameserver may retain this record in its local memory cache. So, on subsequent queries to hosts in the "edu" domain, the root servers would not need to be checked - the cached response from the first lookup would provide the list of nameservers for the "edu" domain. Likewise, lookups of other hosts in the "engin.umich.edu" domain could go straight to the authoritative servers for that domain, based on the cached response from the previous queries.

It may strike one as odd that the DNS works from right to left, when most modern languages are parsed

from left to right. Indeed, at one time the UK domain was specified in the opposite manner, and great pains had to be undertaken in sendmail configuration files and the like to translate between "churchill.leeds.ac.uk" and "uk.ac.leeds.churchill," both specifying the host "churchill" at the University of Leeds in Great Britain. Eventually, the right to left format won out.

In addition to providing IP address information in the "A" records, the DNS protocol is capable of returning many other types of records, and is extensable within the definition of the protocol to allow the addition of new resource record types. Some common record types are "HINFO," for host type and operating system data, "MX," for mail exchanger lists, "PTR" for looking up a hostname based on the IP address, and so on.

And in order to provide redundancy in this important service, a master name server containing the official databases for a given domain may have one or more "slave" nameservers, that simply copy the master's entire set of databases and provide responses to querying resolvers as if they were the master nameserver. When the serial number of each zone is updated, the slaves are notified and subsequently download a new copy of the zone data files.

The Need for DNS Security

Candid and generous and just. Boys care but little whom they trust. An error soon corrected - for who but learns in riper years. That man, when smoothest he appears, is most to be suspected? ~ William Cowper

The Internet started out with the same kind of attitudes seen in a small village, where everyone trusted their neighbors and left their doors unlocked. As the network grew and more and more people got connected, the shortcomings of this approach became apparent.

In earlier versions of DNS implementations, if a remote nameserver offered resource records that your nameserver did not specifically request, your nameserver would happily accept them, add them to its local cache, and start handing them out to anyone who asked. In an environment of trust, this is a reasonable and efficient thing to do - why waste time querying for those records when someone else just handed them to you without any effort on your part?

But this attitude of trust led to an exploit called "DNS cache poisoning" - fake address records were fed into an unsuspecting server's cache, redirecting a commonly accessed site to a different host controlled by the hacker, allowing abuse of the trust of the end users, or to nowhere at all, leading to a denial of service.

Updated versions of the DNS software stopped accepting unsolicited resource records, but there was still no way to conclusively prove that the response your local nameserver received came from the actual authoritative source for that domain - maybe another server somewhere else on the net, running an old version of the software, has been exploited? And how would a slave server know that it hadn't been tricked into downloading a zone data file from someone other than its real master server?

The solution to the problem of trust and authenticity came through the application of asymmetric key cryptography and cryptographic hash algorithms, and was put forth in RFC-2065 as a proposed set of protocol extensions and added resource records.

Hashes and Asymmetric Cryptography

Secrecy is the first essential in affairs of the State. ~ Cardinal De Richelieu

Briefly, a cryptographic hash is a short summary of a block of data that has the following characteristics: 1) derivation of the original data is difficult or impossible, and 2) a very small change in the original data yields a large change in the resulting hash. A hash is calculated by the sender of a block of data, and if that

hash matches the one calculated by the recipient of the data, it can be reasonably assured that the data received exactly matches what was originally sent.

Asymmetric, or "public-key" cryptography, is a technique whereby data encrypted with one key can only be decrypted by another, related key, and vice versa. One key is kept private, and the other is made public. Used in conjunction with a cryptographic hash, a "digital signature" can be constructed by encrypting a hash of the delivered data using the *private* key. If the hash can be successfully decrypted with the corresponding, trusted public key, and matches the locally-calculated hash, it is reliably established that 1) the data has not been changed in transit, and 2) the data comes from the owner of the secret key corresponding to the public key used to decrypt the hash.

Application to DNS Security

Sincerity and truth are the basis of every virtue. ~ Confucius

Through the distribution server public keys through the DNS server network with the **KEY** record type, and cryptographically signing the resource records with the server's private key with signatures for those records delivered in the **SIG** record type, resolvers and nameservers can establish, through layers of signatures leading back to trusted or statically-configured public keys, the authenticity of records delivered among security-aware nameservers and the resolvers.

So for example, a slave server might have a statically-configured file containing the public key of its master server, and upon receiving a signed zone transfer from the master, it can check the signature against the trusted public key in its local configuration files by using that key to decrypt the master server's digital signature of the delivered records.

The use of the **SIG** record type provides this signature, and a way to strongly authenticate records that *do* exist in a zone, but a different approach is needed in order to strongly authenticate the denial of the existence of a name in a zone, or to deny the existence of a certain record type for an existing name, such as when a domain name has an "MX" record, but no "A" record, and an "A" record has been requested by the resolver. If there's no such record, there's nothing to sign!

This is accomplished using the **NXT** record type for a name interval containing the nonexistent name. So for example, if one looked up nonexistent host "b.foo.com" in a domain containing both hosts "a.foo.com" and "c.foo.com," the server might return a response such as:

```
a.foo.com. NXT c.foo.com.
```

signed with the secret key of the foo.com nameserver. It is essentially saying that "a.foo.com exists, but the next record in order is c.foo.com, and since b.foo.com falls between a and c, b.foo.com does not exist." It's rephrasing the response in an affirmative form, giving a response that can be signed and authenticated.

And as you've guessed from the title of this paper, the **NXT** resource record is the crux of this exploit.

DESCRIPTION OF VARIANTS

Since arbitrary malicious code may be inserted by the exploitation of this bug, there may be an arbitrary number of possible variants. However, the main variant is the "t666.c" code, written by "ADM," and available at http://www.hack.co.za/daem0n/named/t666.c. It uses the bug to create the most important of all hacking tools, a shell running as a priveleged user.

It is conceivable that this exploit could be used to write a worm to hop from one vulnerable nameserver to

another, but there have been no indications of such.

HOW THE EXPLOIT WORKS

If the human race wants to go to hell in a basket, technology can help it get there by jet. - Charles M. Allen

At its heart, this exploit is a classic buffer overrun, with its intellectual heritage in Phrack 49 Article 14, "*Smashing the Stack for Fun and Profit.*" This technique has been used to good effect on a wide variety of programs, ranging from the mundane `syslog`, `mount`, and `at`, to large complex applications such as `sendmail` and X Windows libraries.

Without sending you off to read the entire article reference above, a basic explanation of the concepts involved in a buffer overrun exploit is called for.

Understanding Processes

The "stack" of a process is a memory structure that operates much like a stack of plates at a salad bar, with the last item placed on the stack being the first one taken off. Function calls, a common feature of all but the most rudimentary C programs, make use of the stack to handle argument passing and the `return()` from the function call: First, the address to which the function should return when it is completed is "pushed" onto the stack in the form of a "stack frame," which contains both the data and a pointer to the next lower element of the stack. This is then followed by "pushes" of the values of the arguments to be passed to the function being called. After each "push," the stack pointer is updated to point to the memory location containing the topmost element of the stack. In a virtual sense, the stack might wind up looking something like this:

```
 STACK TOP
+---------+
|  arg 3  |  <--[stack pointer]--
+---------+
|  arg 2  |
+---------+
|  arg 1  |
+---------+
| return  |
| address |
+---------+
   [...]
```

The return address winds up on the bottom, since it was "pushed" first, followed by the arguments to the function.

Next, the program jumps its execution to the memory address of the function being called, and the code at that location first "pops" the arguments it needs off the top of the stack. Each "pop" uses the pointer in the popped stack frame to redirect the stack pointer to the next lower element of the stack. The most recently pushed values are the first to be popped off of the stack. This concept is commonly referred to as "last in, first out," or "LIFO." When the arguments are popped, the return address remains in the stack while the function proceeds.

As the function executes its instructions, it may call another function, `f2()`. This process of pushing a return address and arguments is simply repeated, leaving the calling function's return address on the bottom

of the stack, like so:

```
 STACK TOP
+---------+
| f2 arg2 | <--[stack pointer]--
+---------+
| f2 arg1 |
+---------+
| f2 ret  |
| address |
+---------+
| f1 ret  |
| address |
+---------+
   [...]
```

As each function executes the `return()` call, the address to which the execution flow should be returned is popped off the stack, and the flow resumes from that address, which under *normal* circumstances is the address immediately following the one from where the function was originally called. This is the important part. Remember this.

A "buffer" is simply a range of memory locations that hold a series of values of the same type. A variable defined as a "`char[10]`" would be a character buffer of length 10, for example.

In order to understand how a buffer overrun works to exploit the system, it's necessary to understand how a typical UNIX process is laid out in memory by the operating system. The executable code, or "text," of the process is stored in the lowest memory addresses, followed by the segment for storage of initialized and uninitialized variables and data. Finally, in the highest realm of memory addresses, and above the variables, lies the memory where the stack data structure is stored.

Borrowing a diagram from the Phrack article:

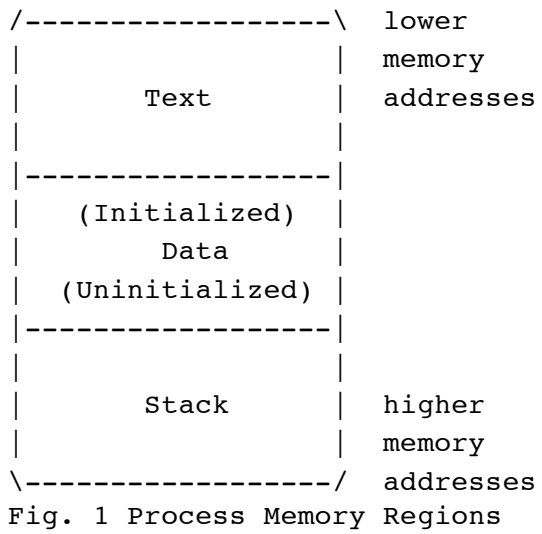


Fig. 1 Process Memory Regions

The Elements of Buffer Overrun

A buffer overrun exploit first requires a modicum of carelessness on the part of the programmer, coupled

with the characteristic of computers to do exactly what you tell them to do, no matter what.

If you carelessly write a series of 20 bytes to a 10-byte buffer, the 10 bytes making up the buffer will be written, followed by 10 additional bytes merrily written beyond the range of memory addresses allocated for the 10-byte buffer, overwriting anything that happened to be present in that memory.

Even if what was present in that memory was the stack itself!

Unless careful bounds checking is done by the author of the software, to insure that the length of what is proposed to be written into a range of memory is actually equal to or less than the size of that memory range, then by stuffing a large batch of data into the right input of the software it can be induced to "*smash the stack*" by writing that entire large batch of data right off the end of the buffer, off the end of the data area in the middle of the process' memory, and right over the beginning of the stack.

Cleverness Applied to Stack-Smashing

Find enough clever things to say, and you're a Prime Minister. Write them down and you're Shakespeare. ~ George Bernard Shaw

Sure, smashing the stack with a bunch of garbage would be a good way to crash a program - write a bunch of garbage characters into the stack and blow away the running executable as it attempts to return from a function call straight into unaligned never-never land. However, by carefully constructing the data written past the end of the buffer, one can align it just so, in such a way that it overwrites the return address for the currently-running function with a modified return address. And that modified address corresponds to the beginning of a bit of malicious assembly language code that is written along with the overrun data.

An important thing to note about this and all buffer overrun exploits is that since assembly language is required, this exploit must be customized to each CPU on which it is to be applied. A series of SPARC assembly language instructions would appear as nothing but gibberish to an x86 CPU, and would likely result in an illegal instruction exception and a core dump, rather than the successful creation of a shell. In addition, different operating systems on the same CPU platform, such as Linux vs. FreeBSD, use different formats for their stack frame, necessitating multiple versions of the malicious code for the same CPU.

When the function returns by popping the now-bogus address off of the overwritten stack, it jumps to the address of the malicious code, which executes as the owner of the program.

The NXT Generation

And this series of events is exactly what takes place with the NXT buffer overrun exploit, creating a shell for the attacker.

The bogus data containing the exploit code is introduced by tricking the target nameserver into making a query to a hacker-controlled fake nameserver. The code then overruns the buffer used to store the NXT resource record response as it is being prepared for evaluation and processing. This tricking of the target nameserver requires control of a different master nameserver to redirect the target nameserver, and control of a different machine that will be used to deliver the bogus data. Naturally, an alternative attack must be used to comopromise the "patsy" master nameserver system, but since there's no shortage of ways in which to exploit root on a UNIX system, this requirement is not a show-stopper.

As mentioned in the above description of the protocol, each subdomain is defined by an "NS" record in the parent domain. By adding a bogus "NS" record to a controlled master name server which points to another controlled machine, any queries for that domain will be redirected to the server listed in that NS record. So

for example, you might have a zone file that looks like this:

```
; company.tld zone file - Aug 15 2000
@               IN      SOA     company.tld. dns.company.tld. (
                2000081501      ; Serial
                10800           ; Refresh
                3600            ; Retry
                604800          ; Expire
                86400 )         ; Minimum TTL
                IN      NS      ns.company.tld.
                IN      NS      ns.upstreamisp.net.
                IN      MX      0 mail.company.tld.
ns              IN      A       172.20.99.45
mail            IN      A       172.20.99.47
```

Assuming that the other controlled machine's IP address is 192.168.45.97, by adding the following line to this file:

```
haxdomain       IN      NS      192.168.45.97
```

then any queries to this compromised master server for any host in "haxdomain.company.tld." zone will be redirected to the hacker's machine on port 53, the nameserver port.

But what will be listening on that hacker port 53 will not be a normal nameserver, but a compiled executable designed to deliver the malicious code in the form of an NXT response, built from the t666.c source code examined below.

The next step is to have the target nameserver query a host in that hacked domain. This, of course, is a very simple matter. The "nslookup" software can be set to resolve hostnames from any arbitrary server on the Internet. This, of course, is a very useful debugging feature, allowing sysadmins to compare responses from multiple servers to identify the origin and extent of a nameserver configuration problem.

So, in order to force a query of the hacked domain, only the following is necessary:

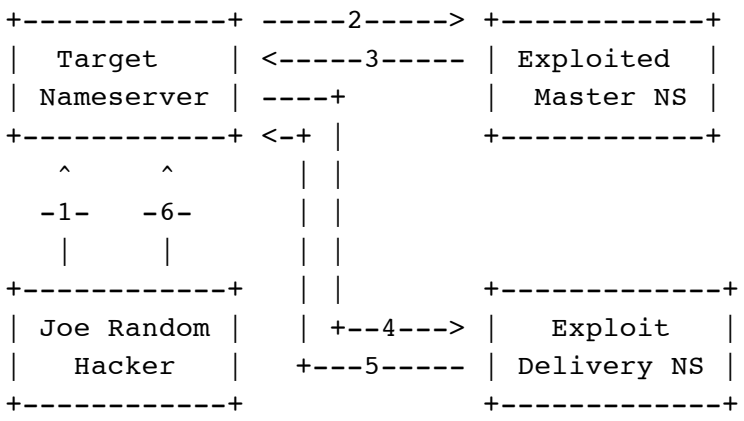
```
% nslookup - targetns.victimcompany.com
Default Server:  targetns.victimcompany.com
Address:  10.27.95.142

> fakehost.haxdomain.company.tld.
```

As was explained in the protocol description above, this request will cause the target nameserver to eventually query the company.tld nameserver, which will dutifully return the NS record for haxdomain.company.tld that points to the other controlled box, and immediately upon asking that controlled box for an "A" record for fakehost.haxdomain.company.tld, the fake "t666.c" fake nameserver running on that controlled box will deliver the buffer overflow data to the target nameserver in the form of an NXT reply. And finally, the target nameserver will begin executing the malicious code, which in the case of t666.c will create a shell running as the owner of the "named" process, often root, and execute the command supplied to the program.

The details of the specific source code and the exploit itself will be examined below.

DIAGRAM OF OPERATION



1. Joe Random Hacker uses `nslookup` or another DNS query tool to send a request for a modified subdomain in a zone controlled by the Exploited Master Nameserver.
2. The Target Namserver sends a query to the Exploited Master Nameserver for the modified subdomain in an attempt to fulfull Joe Random Hacker's request.
3. The Exploited Master Namserver returns an NS record for the modified subdomain which points to the address of the Exploit Delivery Nameserver, another box controlled by Joe Random Hacker. It may even be the same box used to make the initial request, but is separated here for the sake of clarity.
4. The Target Nameserver sends a query to the Exploit Delivery Nameserver in an attempt to resolve the request from Joe Random Hacker.
5. The Exploit Delivery Nameserver sends back the malformed NXT response, smashing the stack of the Target Nameserver, and causing it to execute the malicious code which runs a specified command as root on the system. By default, it creates a root shell listening at port 1524.
6. Joe Random Hacker uses this shell to further exploit the system, installing a rootkit, perhaps.
7. The Target Nameserver crashes because of its corrupted stack.
8. Joe Random Hacker, covering his tracks, uses his rootkit back door to restart the nameserver.

HOW TO USE THIS EXPLOIT

A fairly comprehensive "HowTo" document, written by "E-Mind," is available at this URL: http://packetstorm.securify.com/0003-exploits/NXT-Howto.txt

The essential approach follows the steps laid out in the section immediately above.

1. A master nameserver must be exploited.
2. A bogus NS record must be inserted to point to another exploited system
3. The exploit delivery code must be installed on the exploited system pointed to by the bogus NS record.
4. The target nameserver must be caused to query the exploited system running the exploit delivery code.
5. Using the shell created in inetd.conf on the target nameserver, a rootkit must be installed to maintain access to the system and cover the tracks of the exploit.

The HowTo also includes instructions on removing an obscure, slight bug deliberately inserted into the distributed t666.c source code, purportedly introduced in order to prevent script kiddies from using the exploit.

SIGNATURE OF THE ATTACK, DETECTION & BLOCKING

The t666.c script, written by ADM creates an easily-detected subdirectory called "ADMROCKS" in the directory where the target DNS server is running. Hubris got the better of discretion in this case, it would appear.

Short of examining the fingerprints of every DNS query and response looking for the exploit code, there is no good protocol-level way to detect the attack, since the queries and responses are part of the normal operation of the network of Internet nameservers.

Blocking the attack at a network level is less practical than simply upgrading the BIND software, since the problem stems from a bug in the processing of a legitmate nameserver response message. The blocking comes when the BIND software performs the proper bounds checking on the data returned from the query.

PROTECTION AGAINST THIS ATTACK

Protection is very simple - upgrade BIND to version 8.2.2-P5 or higher. The NXT record returned by remote servers in this version is checked against the size of the buffer into which it is intended to be stored, correcting the bug that allowed the buffer overrun.

Because this exploit is made possible by poorly designed code within the operation of the nameserver itself, upgrade of this software to fix the bug is the only practical protection.

SOURCE CODE

First, let's examine the aspects of the nameserver source code that lead to this vulnerability. Using BIND version 8.2 from the Internet Software Consortium, we need to examine the part of the code that handles replies from remote nameservers to locally-issued queries, and in particular the handling of the NXT response. Incoming responses are handled in the `ns_resp.c` file, within the `rrextract()` function, which pulls the resource record out of the wire-formatted response data stream and converts it into the internal database format.

Each DNS response packet consists of a name, a two-byte type specification, a two-byte class specification, a signed 32-bit integer specifying the time-to-live for this record, an unsigned 16-bit integer specifying the length of the resource record data, RDLENGTH, and finally the data itself, RDATA.

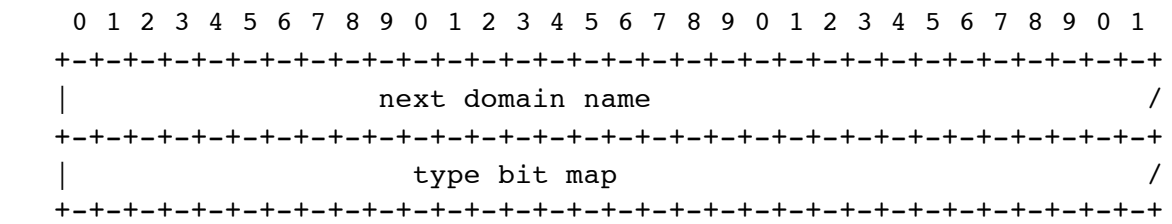
In the first section of the `rrextract()` function, the resource record is identified based on its type, as indicated in the header of the response packet. Next, a `switch()` function is executed against the resource record type, and each resource record is handled differently based on its type, evaluating the RDATA according to the format set forth by the definition of the RR type.

And in particular, the RDLENGTH section of the resource record header is pulled out into the dlen variable, by the following code:

```
GETSHORT(dlen, cp);
```

The NXT resource record RDATA, as set forth in RFC 2035, has the following format:

```
1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
```



The domain name is simply a canonical form of the next record, "c.foo.com" from the example above, and the type bit map is a list of bits that are set to indicate which resource records exist for the domain name, each bit position corresponding to a type number as specified in RFC 1035, Section 3.2.2 - 1 is "A," 2 is "NS," 5 is "CNAME," and so on.

The following is the section of the `rrextract()` function's `switch()` statement dealing with the NXT record type:

```
1.  case T_NXT:
2.      n = dn_expand(msg, eom, cp, (char *)data, sizeof data);
3.      if (n < 0) {
4.          hp->rcode = FORMERR;
5.          return (-1);
6.      }
7.      if (!ns_nameok((char *)data, class, NULL, response_trans,
8.              domain_ctx, dname, from.sin_addr)) {
9.          hp->rcode = FORMERR;
10.         return (-1);
11.     }
12.     cp += n;
13.     cp1 = data + strlen((char *)data) + 1;
14.     memcpy(cp1, cp, dlen - n);
15.
16.     cp += (dlen - n);
17.     cp1 += (dlen - n);
18.
19.     /* compute size of data */
20.     n = cp1 - (u_char *)data;
21.     cp1 = (u_char *)data;
22.     break;
```

First, the `dn_expand()` function reconstitutes the full-length domain name from the DNS-compressed format used on the wire for avoiding redundancy when transmitting domain names. A negative return value, detected on line 3, indicates an error in the compression or a problem in the format of the domain name, whereas a positive return value indicates both success, and the length of the domain name as found in the raw resource record. The expanded domain name is stored in the `data` char buffer.

Given a successful decompression from the wire format, the expanded name, in `(char *)data`, is checked within the appropriate context in accordance with the class in which it falls. Normally this will be the "IN" class. A properly formed domain name will result in a successful completion of this check.

The cp variable is an unsigned char pointer, which is set in the very beginning of the `rrextract()` function to the `rrp` variable - the resource record pointer - passed in by the calling function, and pointing to the beginning of the resource record. As the resource record is parsed, `cp` is incremented to point to the next

area of interest in the block of resource record data. When the switch() statement begins, the `cp` pointer is positioned at the beginning of the RDATA block.

On line 12, the current position pointer is incremented by the length of the domain name as returned by the `dn_expand()` function on line 2, positioning cp at the beginning of the type bit map in the raw resource record.

On line 13, cp1, another char pointer, is set to one character past the end of the expanded domain name, by adding the `data` pointer, positioned at the beginning of the expanded domain name, and the length of that domain name, and one additional character, which strictly speaking should be `sizeof(char)` instead of simply "1."

Then, the rubber meets the road: the memcpy() function is executed, copying bytes from the current position in the RDATA into the memory address specified in cp1, for up to dlen bytes. The function cleans up the pointers, breaks, and returns, right into the arms of the exploit code.

Say goodnight, Gracie.

The blocks of hexadecimal code in the beginning of the t666.c exploit are written starting at cp1 off into the stack area, and the maliciouis code is executed as its address is popped right off the stack.

Looking at some of the hexadecimal codes in the t666.c source code, most of which represent assembly language instructions or stack frame modifications, you can spot one of the signatures of the attack - the ADMROCKS string, rendered in hex as 0x41, 0x44, 0x4d, 0x52, 0x4f, 0x43, 0x4b, 0x53.

The source code to the t666 exploit of the NXT buffer overflow can be found here:
http://www.hack.co.za/daem0n/named/t666.c

REFERENCES

Internet Software Consortium, *BIND Source Code,* ftp://ftp.isc.org/isc/bind/src/DEPRECATED/8.2/
E-Mind, *t666.c Source Code*, http://www.hack.co.za/daem0n/named/t666.c
Aleph One, "*Smashing the Stack for Fun and Profit,*" Phrack Volume 49Article 14, November 1996 http://www.phrack.com/search.phtml?view&article=p49-14
E-Mind, "*BIND 8.2 - 8.2.2 *Remote root Exploit How-To**", http://packetstorm.securify.com/0003-exploits/NXT-Howto.txt
Mockapetris, P., "*Domain Names: Implementation Specification*", RFC 883, November 1983 (Obsoleted by RFC 1035)
Mockapetris, P., "*Domain Names: Implementation & Specification*", RFC 1035, November 1987
Eastlake, D., and Kaufman, C., "Domain Name System Security Extensions", RFC 2065, January 1997 (Obsoleted by RFC 2535.)
Eastlake, D., "Domain Name System Security Extensions", RFC 2535, March 1999