Use offense to inform defense.
Find flaws before the bad guys do.

IP Masquerading Vulnerability for Linux 2.2.x - CVE-2000-0289

(Version 1.4a)

Tanya Baccam, CISSP, CISA, MCSE, CCNA, CCSE, CCSA, Oracle DBA

IP Masquerading Vulnerability

Exploit Details

This paper focuses on an IP Masquerading vulnerability that is present in multiple Linux versions. The vulnerability is identified as CVE-2000-0289 in the Common Vulnerabilities and Exposures database. The vulnerability can exploit the masquerading feature in the Linux kernel where the masquerading code will allow arbitrary backward connections to be opened. The vulnerability was first posted on BugTraqs on March 27, 2000, under the title "Security Problems with Linux 2.2.x IP Masquerading." SUSE published the vulnerability on May 20, 2000, as a security hole in the kernel. This vulnerability does have the potential to cause a Denial of Service (DOS) attack. At the time of this writing, no direct variants of this vulnerability exist. The following operating systems can be affected by this vulnerability:

- Debian Linux 2.2pre potato
- Debian Linux 2.2
- Debian Linux 2.1
- Linux kernel 2.2.14
- Linux kernel 2.2.12
- Linux kernel 2.2.10
- RedHat Linux 6.2 i386
- RedHat Linux 6.1 sparc
- RedHat Linux 6.1 i386
- RedHat Linux 6.1 alpha
- RedHat Linux 6.0 sparc
- RedHat Linux 6.0 i386
- RedHat Linux 6.0 alpha

This paper will focus specifically on this vulnerability as it relates to the UDP protocol.

Protocol Description

The UDP protocol is documented in RFC768. UDP provides access to many services which are supported by IP and assumes that IP is utilized as the underlying protocol. UDP packets are very similar to IP packets and are delivered in the same manner as IP packets. It should be noted that UDP is a connectionless protocol, unlike TCP which is connection-oriented. UDP is typically used when TCP would be too complicated, too slow or simply not necessary. In addition to the functions provide by IP, UDP also provides 16-bit port numbers and checksums. The 16-bit port numbers can be used to let multiple processes use UDP services on the same host. Data integrity can be ensured by utilizing the checksum field of the UDP protocol. The UDP protocol is

transaction oriented. Since UDP is connectionless, duplicated messages or messages not received can not be verified by UDP. The UDP protocol has the following fields:

- Source Port
- Destination Port
- Length
- Checksums
- Data Octets

The **source port** field is not required. When it is utilized it will identify the port that sent the packet. This port can be utilized to return any appropriate reply. If no value is provided, a value of zero is inserted into this field. The **destination port** identifies a particular Internet destination address. The **length** field identifies the length of the datagram including both the header and the related data recorded in octets. The **data octets** contain the actual data. The **checksum** is a count of the number of bits in a given transmission that is included in the packet so the receiver can check whether the same number of bits has arrived. Below, you will find a diagram of the UDP protocol:

Source Port	Destination Port
Length	Checksum
Data Octets ...	

The following contains a sample listing of applications that are implemented utilizing UDP, as well as the port number on which the application typically operates.

- Echo 7
- Discard 9
- Daytime 13
- Quote 17
- Chargen 19
- Nameserver 53
- Bootps 67
- Bootpc(DHCP) 68
- TFTP 69
- SunRPC 111
- NTP 123

- NetBios 137
- SNMP 161
- SNMP 162
- CORBA IIOP 535

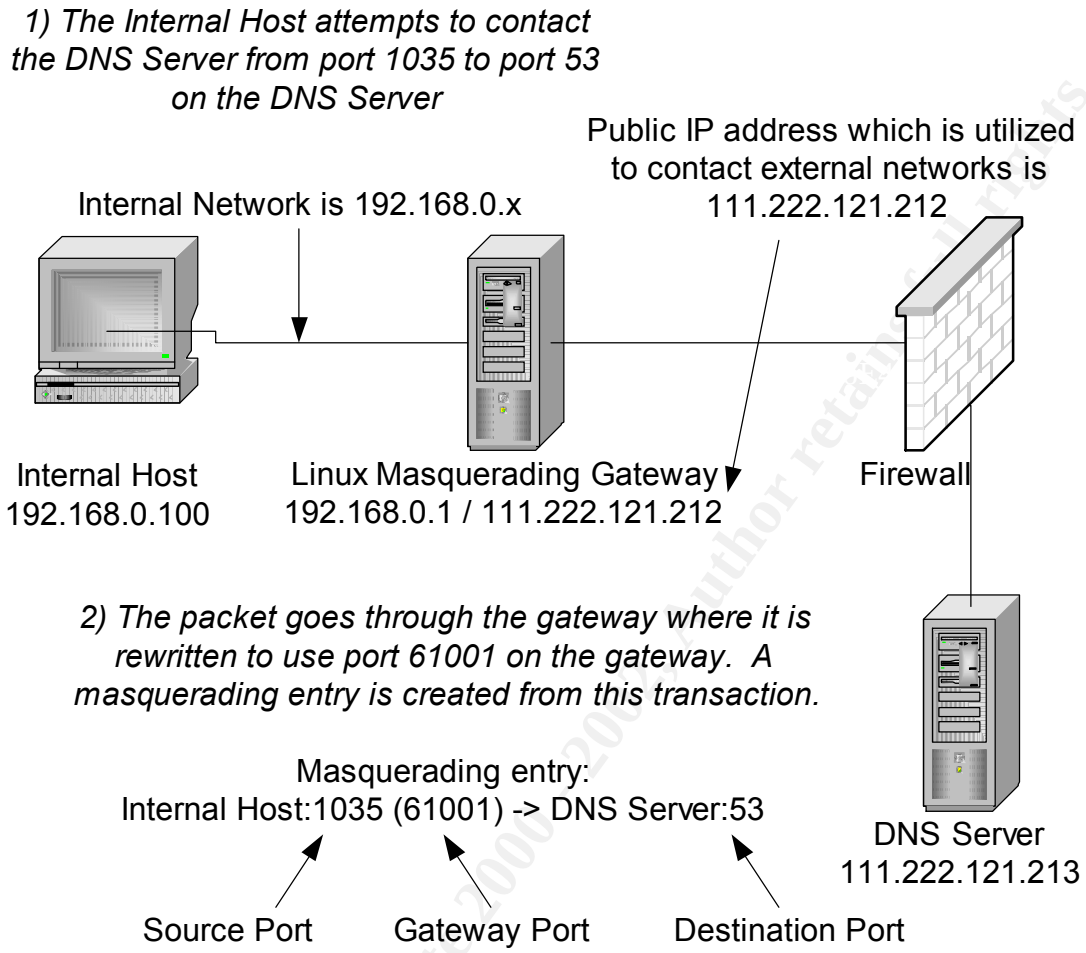
Description of Variants

No direct variants of this vulnerability exist; however, there are multiple Denial of Service vulnerabilities. To find more information about Denial of Service attacks, please refer to the listing in the "additional information" section.

How the Exploit Works

To understand how this exploit works, an understanding of IP masquerading is required. IP masquerading is a NAT (Network Address Translation) implementation specific to the Linux OS. IP masquerading allows multiple internal machines to connect to an external network (i.e. Internet) via a single IP address. The packets, whose source is the internal network, go through a gateway where their packet is rewritten to contain the IP address and port number, which the gateway will utilize to handle the connection to the external network. By default, the kernel on the gateway reserves 4096 ports for UDP and TCP connections. Specifically, ports 61000 to 65096 are reserved to handle these masqueraded connections. The diagram below outlines this process.

MASQUERADING DIAGRAM



The UDP protocol requires a source and destination port and address. The source port for a given UDP connection is typically selected from the first port available between 1024 and 65535 on the internal machine. When the packet is sent from the internal machine to an external address via the masquerading machine, an entry is entered into the masquerading table. The entries are in the following format:

- Internal Host: Internal port (masquerading port) -> External Host: External port

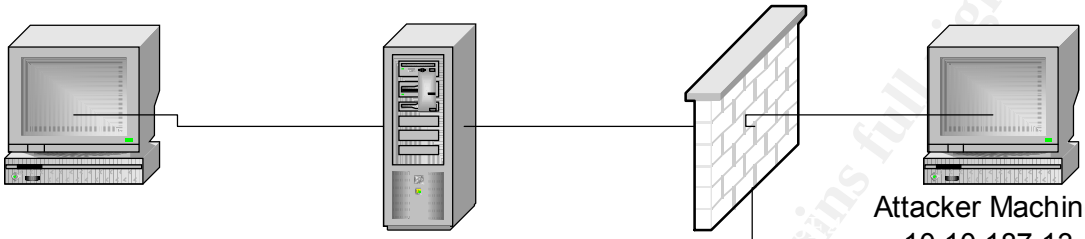
This exploit takes advantage of the lack of checking prevalent in the kernel code for masquerading. The masquerading code only checks the destination port in order to determine whether a packet should be forwarded into the network. A potential attacker can rewrite the UDP masquerading entries for a Linux masquerading gateway to make the external host and port whatever they may choose. This will create a tunnel from the attackers machine, or whatever host and port they specify, and the internal machine whose connection is being masqueraded. The exploit takes advantage of the fact that UDP is a

connectionless protocol, since the only way to determine whether or not a connection is still being utilized for a UDP connection is by the lack of activity or ICMP messages indicating that the port is closed. A five minute default time-out for the UDP a masqueraded connection is built in. During this five minute time-out, the attacker can find and exploit a given connection.

Diagram

EXPLOITATION DIAGRAM

1) The internal host sends a request from port 1035 to the DNS server on port 53 that goes through the masquerading gateway. The masquerading gateway uses port 63000 to forward this request. The following entry is made on the masquerading gateway: "UDP 03:39.21 192.168.1.100 10.0.0.25 1035 (63000) -> 53"



Internal Host
192.168.1.100

Linux Masquerading Gateway
192.168.1.1 / 10.0.0.1

Firewall

Attacker Machine
10.10.187.13

DNS Server
10.0.0.25

3) The internal host will send an ICMP unreachable reply back to the attacker's machine when port 63000 is scanned. The returned reply will have an IP ID which is from either the masquerading machine or the internal host. Since IP ID's are supplied in sequence in a TCP/IP stack, the IP ID's returned from the gateway will be in ascending order. When an IP ID is out of sequence, this signals that the IP ID came from an different machine, which identifies a masqueraded connection.

4) By sending the packet with a source IP address and port from the attacker machine, the masquerading entry gets modified with this new information. The masquerading entry has been changed to "UDP 04:35.12 192.168.1.100 10.10.187.13 1035 (63000) -> 12000" which is the destination IP address and port for the attacker's machine.

2) The attacker machine will send UDP packets from their machine to the masquerading gateway machine on ports 61000 to 65096 to identify masqueraded connections. The source port and destination in the packets will be from the attacker's machine. For illustration purposes, the source port will be 12000 in this example.

How to Use the Exploit

The following example was taken from www.securityfocus.com/bid/1078:

[tcpdump from attacker's machine]

(we picked source port 12345 for our packets just so the trace would be easier to follow)

[snip -- this starts at port 61000]

10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63762 unreachable [tos 0xd8] (ttl 245, id 13135)
10.10.187.13.12345 > 10.0.0.1.63763: udp 0 (DF) [tos 0x18] (ttl 254, id 23069)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63763 unreachable [tos 0xd8] (ttl 245, id 13136)
10.10.187.13.12345 > 10.0.0.1.63764: udp 0 (DF) [tos 0x18] (ttl 254, id 23070)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63764 unreachable [tos 0xd8] (ttl 245, id 13137)
10.10.187.13.12345 > 10.0.0.1.63765: udp 0 (DF) [tos 0x18] (ttl 254, id 23071)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63765 unreachable [tos 0xd8] (ttl 245, id 13138)
10.10.187.13.12345 > 10.0.0.1.63766: udp 0 (DF) [tos 0x18] (ttl 254, id 23074)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63766 unreachable [tos 0xd8] (ttl 245, id 13139)
10.10.187.13.12345 > 10.0.0.1.63767: udp 0 (DF) [tos 0x18] (ttl 254, id 23083)

10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63767 unreachable [tos 0xd8] (ttl 244, id 17205)
^^^
The above packet's ID is substantially different, we may have found a masq'd connection !!!

10.10.187.13.12345 > 10.0.0.1.63768: udp 0 (DF) [tos 0x18] (ttl 254, id 23084)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63768 unreachable [tos 0xd8] (ttl 245, id 13140)
10.10.187.13.12345 > 10.0.0.1.63769: udp 0 (DF) [tos 0x18] (ttl 254, id 23088)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63769 unreachable [tos 0xd8] (ttl 245, id 13141)
10.10.187.13.12345 > 10.0.0.1.63770: udp 0 (DF) [tos 0x18] (ttl 254, id 23090)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63770 unreachable [tos 0xd8] (ttl 245, id 13142)
10.10.187.13.12345 > 10.0.0.1.63771: udp 0 (DF) [tos 0x18] (ttl 254, id 23091)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63771 unreachable [tos 0xd8] (ttl 245, id 13143)

10.10.187.13.12345 > 10.0.0.1.63771: udp 0 (DF) [tos 0x18] (ttl 254, id 23092)
10.0.0.1 > 10.10.187.13: icmp: 10.0.0.1 udp port 63772 unreachable [tos 0xd8] (ttl 245, id 13144)

[snip -- all the way to the upper end of our masq ports]

The example above illustrates how a masqueraded connection can be identified and exploited. The attacker, here located at machine 10.10.187.13 sends packets to the gateway machine which is located at 10.0.0.1. A tool such as nmap can be utilized to scan the appropriate ports. For each packet that is sent, an ICMP reply is sent from the gateway to the attacker's machine noting that the port is unreachable. Included in each of these ICMP replies is a packet ID and ttl (time to live) field. Typically, the IP ID field on the gateway will be at least 1000 digits away from the IP ID field on the internal host machine, since each host's TCP/IP stack will sequentially increment the IP ID field. When the masqueraded port is scanned, an IP ID field will be returned from the internal machine, since this connection will be forwarded due to the masquerading entry. Another factor that can identify a masqueraded connection is that the time to live field will be one less when it is from the internal machine, since it had one additional hop to make. By observing the IP ID field or the ttl field, an attacker will be able to determine which ports are being masqueraded.

Because the UDP masquerading code only checks the destination port to determine whether a packet should be forwarded to the internal network, the source port can be modified. The code sets the remote host and port to that of the incoming connection. Therefore, the attacker only needs to know the port number on the gateway to be able to rewrite the masquerading entries. Noted in the discussion earlier, by default, ports 61000 to 65096 are utilized for this activity and are the only ports the attacker needs to scan.

Before the scan is conducted, an entry on the masqueraded gateway such as the following is present:

- UDP 03:39.21 192.168.1.100 10.0.0.25 1035 (63767) -> 53

This entry can be retrieved by executing "ipchains –L –M –n on the masqueraded gateway. After the scan is conducted, an entry such as the following will exist on the gateway:

- UDP 04:35.12 192.168.1.100 10.10.187.13 1035 (63767) -> 12345

In the second entry, the destination IP address and destination port number has been changed. This was accomplished by sending a packet to the masqueraded gateway containing a source IP address and source port from the attacker's machine.

Nmap can be used to implement this scan. Nmap allows the following type of scans to be conducted per the nmap documentation located at http://www.insecure.org/nmap/nmap_doc.html:

- Vanilla TCP connect() scanning
- TCP SYN (half open) scanning
- TCP FIN (stealth) scanning
- TCP ftp proxy (bounce attack) scanning
- SYN/FIN scanning using IP fragments (bypasses packet filters)
- UDP recvfrom() scanning
- UDP raw ICMP port unreachable scanning
- ICMP scanning (ping-sweep)
- Reverse-ident scanning

By conducting UDP raw ICMP port unreachable scans, the results above should be returned and can be captured by tcpdump. To implement a UDP raw ICMP port unreachable scans with nmap, the '-u' command should be utilized. During UDP ICMP port unreachable scanning, the UDP protocol is utilized to scan requested ports. The UDP protocol itself is a simpler protocol, but this makes scanning more difficult, since UDP open ports do not have to send an acknowledgement and closed ports are not required to reply with an error packet. However, most hosts do send an ICMP_PORT_UNREACH error when a packet is sent to a closed UDP port. Therefore, the ports that do not respond with an ICMP_PORT_UNREACH error are assumed to be open. This scan is retransmitted multiple times by nmap to ensure that open ports, are in fact open since both UDP packets and ICMP errors are not guaranteed to be returned to the host. It should be noted that this type of scan is actually quite slow. In the case of the Linux kernel, it limits destination unreachable message generation to 80 per 4 seconds. It should also be noted that root access is required to access the raw ICMP socket to read the port unreachable packets.

During this process, utilize tcpdump or windump, which is the Windows version of tcpdump, to capture all the packets which are being sent during this transmission. For example, the command "tcpdump host hostname", where "hostname" is the name of your host, can be utilized to capture all the packets coming from and going to the identified host machine. To send the packets to a file, the "-w" option can be utilized. For more detailed information, the "-v", "-vv", or "-vvv" options can be utilized which provide successively more information.

The process above makes it possible to capture the external side of a masqueraded connection. By capturing the external side of a connection, applications which utilize UDP can be exploited if they have vulnerabilities in allowing unrestricted external access to the source ports. (To view a listing of the most common applications which utilizes UDP, see the protocol description section.)

Signature of the Attack

If a company is running Linux 2.2 and has not upgraded to the patched 2.2.14 kernel listed below, the masquerading gateway should be continually monitored for this vulnerability. Exploitation of this vulnerability can be identified by observing incoming UDP traffic on ports 61000 to 65096 on the masquerading gateway. Probes, which are sent from the external network to the internal network on a masqueraded port, could potentially be an attempted exploitation. Monitoring for this type of potential exploitation seems more difficult then simply applying the patch. Therefore, the 2.2.14 kernel patch should be applied to all Linux 2.2 machines.

How to Protect Against the Vulnerability

To protect again this vulnerability, the kernel should be updated from one of SuSE's FTP servers. A patched 2.2.14 kernel was initially provided to ensure stability. The checksums have been included below and the checksums should be verified before implementing the patch.

- ftp://ftp.suse.com/pub/suse/i386/update/6.4/kernel/k_deflt.rpm 765e268875a7716f681c14389a1c9b9b
- ftp://ftp.suse.com/pub/suse/i386/update/6.4/kernel/k_eide.rpm be6ee213f0cafd4dac5c51a2a8d100f0
- ftp://ftp.suse.com/pub/suse/i386/update/6.4/kernel/k_i386.rpm b900eb9f47c94df5cc15721e5f96a58e
- ftp://ftp.suse.com/pub/suse/i386/update/6.4/d1/lx_suse-2.2.14.SuSE-24.i386.rpm 37deca6ee856c3242a13c2a24f32fc7f

If these sites are not available, the following sites have a listing of mirrors:

- http://www.suse.de/ftp.html
- http://www.suse.com/ftp_new.html

If the kernel is not updated, the possibility of putting a caching names server on the masquerading gateway and then disabling the masquerading of UDP packets becomes an option. Typically, it is not necessary to utilize UDP masquerading on a gateway such as this. The only UDP service typically needed on a gateway is DNS, and therefore it is better to put a caching names server on the gateway box instead.

Source Code/ Pseudo Code

No exploit code has been designed specifically for this vulnerability. However, code which conducts a UDP scan for a given IP is available and permits one to obtain the external access described above, which in affect does exploit this vulnerability. UDP scanning code, which was written by Wietse Venema and is included in the SATAN utility, is included in Appendix A. The code includes a lot of verifications including, among other things, probing for alive machines and handling retransmission and congestion, since UDP is a connectionless protocol. Here are short definitions of some of the more foundational functions in use in this code:

- scan_ports: Scans the ports for a given port range.
- monitor_ports: Monitors how much activity is occurring on the socket.
- receive_answers: Receives the answer from a given probe and may call receive_icmp for further information.
- receive_icmp: Receives and translates the ICMP messages that are returned from the probe.
- process_reply: Processes the reply and if the 'verbose' option was chosen, prints this information out, as well.
- report_and_drop_port: Reports the information that is known about the port that was probed and drops the port.
- average: Controls the average roundtrip time for a probe.
- add_port: Records the port that is being probed.
- write_port: Writes information to the port.
- drop_port: Releases the port information for the probed port.
- init_port_info: Makes the additional collection of port information.
- find_port_info: Retrieves information about the port being queried.

Additional Information

Articles About this Vulnerability

- http://linuxtoday.com/news_story.php3?ltsn=2000-04-27-033-04-SC - Story about this Vulnerability

- http://www.suse.de/de/support/security/suse_security_announce_48.txt - Security Announcement from SUSE
- http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0289 - Common Vulnerabilities and Exposures Database Documentation

- http://oliver.efri.hr/~crv/security/bugs/Linux/krnl122.html - Information on this Vulnerability

- http://www.linuxsecurity.com/advisories/redhat_advisory-413.html - Suggested Fix for the Vulnerability

- http://icat.nist.gov/icat.taf?_function=cve&_UserReference=2F6F0A5F515D40243AB6CB22&cvename=CVE-2000-0289 - Additional Information on this Vulnerability

UDP Protocol Information

- http://www.faqs.org/rfcs/rfc768.html - RFC Documentation for the UDP Protocol

Linux Information

- http://www.suse.de/security - Linux Security Announcements
- http://www.linuxsecurity.com/docs/SecurityAdminGuide/SecurityAdminGuide.txt - Linux Security Administrator's Guide
- http://www.linuxtoday.com - Newsletters for Linux
- http://securityportal.com/research/research.wsl.html - Weekly Linux Updates
- http://securityportal.com/research/research.wsl.html - Archives with Exploit Code
- http://securityportal.com/research/exploits/linux/ - Linux Exploits
- http://www.suse.de/patches/index.html - Patches for Linux Systems

Tools Documentation

- http://www.insecure.org/nmap/nmap_doc.html - Documentation on Nmap
- http://ciac.llnl.gov/ciac/notes/Notes07.shtml - Notes from CIAC about SATAN(System Administrators Tool for Administering Networks)
- http://www.cerias.purdue.edu/coast/satan.html - Information about SATAN
- http://www.tcpdump.org/ - Information about Tcpdump as well as the Tcpdump Tool
- http://netgroup-serv.polito.it/windump/ - Information on Windump

Denial of Service Attacks Information

- http://www.sans.org/ddos_roadmap.htm - "Consensus Roadmap for Defeating Distributed Denial of Service Attacks" by SANS
- http://www.infosyssec.com/infosyssec/secdos1.htm - Background Information on Denial of Service Attacks

Additional Security Resources

- http://www.infosyssec.net/ - Portal for Information System Security Professionals

- http://www.securitypanel.org/security_portals.html - Listing of Security Portals

Appendix A
UDP Scanning Code from SATAN

```
/*
 * udp-scan - determine available udp services
 *
 * Author: Wietse Venema.
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/socket.h>
#include <sys/time.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/udp.h>

#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

extern int errno;

#ifndef __STDC__
extern char *strerror();
#endif

extern char *optarg;
extern int optind;

#define offsetof(t,m)  (size_t)(&(((t *)0)->m))

#ifndef FD_SET
#include <sys/select.h>
#endif

#include "lib.h"

#define LOAD_LIMIT          100             /* default max nr of open sockets */
#define AVG_MARGIN          10              /* safety margin */

 /*
  * In order to protect ourselves against dead hosts, we first probe UDP port
  * 1. If we do not get an ICMP error (no listener or host unreachable) we
  * assume this host is dead. If we do get an ICMP error, we have an estimate
  * of the roundtrip time. The test port can be changed with the -p option.
  */
char   *test_port = "1";
int    test_portno;

#define YES    1
#define NO     0

int    verbose = 0;                     /* default silent mode */
int    open_file_limit;         /* max nr of open files */

 /*
  * We attempt to send as many probes per roundtrip time as network capacity
  * permits. With UDP we must do our own retransmission and congestion
  * handling.
  */
int    hard_limit = LOAD_LIMIT;         /* max nr of open sockets */
int    soft_limit;                      /* slowly-moving load limit */
```

```
struct timeval now;                     /* global time after select() */
int    ports_busy;                      /* number of open sockets */
int    want_err = 0;                    /* show reachable/unreachable */
int    show_all = 0;                    /* show all ports */

 /*
  * Information about ongoing probes is sorted by time of last transmission.
  */
struct port_info {
   RING    ring;                        /* round-robin linkage */
   struct timeval last_probe;           /* time of last probe */
   int     port;                        /* port number */
   int     pkts;                        /* number of packets sent */
};

struct port_info *port_info = 0;
RING    active_ports;                   /* active sockets list head */
RING    dead_ports;                     /* dead sockets list head */
struct port_info *find_port_info();     /* retrieve port info */

 /*
  * Performance statistics. These are used to update the transmission window
  * size depending on transmission error rates.
  */
double  avg_irt = 0;                    /* inter-reply arrival time */
double  avg_rtt = 0;                    /* round-trip time */
double  avg_pkts = 1;                   /* number of packets sent per reply */
int     probes_sent = 0;            /* probes sent */
int     probes_done = 0;                /* finished probes */
int     replies;                        /* number of good single probes */
struct timeval last_reply;              /* time of last reply */

int     send_sock;                      /* send probes here */
int     icmp_sock;                      /* read replies here */
fd_set  icmp_sock_mask;                                 /* select() read mask */
static struct sockaddr_in sin;

 /*
  * Helpers...
  */

#define time_since(t) (now.tv_sec - t.tv_sec + 1e-6 * (now.tv_usec - t.tv_usec))
#define sock_age(sp) time_since(sp->last_probe)
double  average();
struct port_info *add_port();

/* main - command-line interface */

main(argc, argv)
int    argc;
char   *argv[];
{
   int     c;
   struct protoent *pe;
   char  **ports;

   progname = argv[0];
   if (geteuid())
            error("This program needs root privileges");

   open_file_limit = open_limit();

   while ((c = getopt(argc, argv, "al:p:uUv")) != EOF) {
            switch (c) {
            case 'a':
               show_all = 1;
               break;
            case 'l':
               if ((hard_limit = atoi(optarg)) <= 0)
                        usage("invalid load limit");
```

```
                break;
            case 'p':
                test_port = optarg;
                break;
            case 'u':
                want_err = EHOSTUNREACH;
                break;
            case 'U':
                want_err = ~EHOSTUNREACH;
                break;
            case 'v':
                verbose = 1;
                break;
            default:
                usage((char *) 0);
                break;
            }
}
argc -= (optind - 1);
argv += (optind - 1);
if (argc < 3)
            usage("missing argument");

if (hard_limit > open_file_limit - 10)
            hard_limit = open_file_limit - 10;
soft_limit = hard_limit + 1;
init_port_info();

if ((pe = getprotobyname("icmp")) == 0)
            error("icmp: unknown protocol");
if ((icmp_sock = socket(AF_INET, SOCK_RAW, pe->p_proto)) < 0)
            error("icmp socket: %m");
FD_ZERO(&icmp_sock_mask);
FD_SET(icmp_sock, &icmp_sock_mask);

if ((send_sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
            error("socket: %m");

/*
 * First do a test probe to see if the host is up, and to establish the
 * round-trip time. This requires that the test port is not used.
 */
memset((char *) &sin, 0, sizeof(sin));
sin.sin_addr = find_addr(argv[1]);
sin.sin_family = AF_INET;

gettimeofday(&now, (struct timezone *) 0);
last_reply = now;

/*
 * Calibrate round-trip time and dead time.
 */
for (;;) {
            scan_ports(test_port);
            while (ports_busy > 0)
                monitor_ports();
            if (avg_rtt)
                break;
            sleep(1);
}
scan_ports(test_port);

/*
 * Scan those ports.
 */
for (ports = argv + 2; *ports; ports++)
            scan_ports(*ports);

/*
 * All ports probed, wait for replies to trickle back.
```

```
    */
    while (ports_busy > 0)
            monitor_ports();

    return (0);
}

/* usage - explain command syntax */

usage(why)
char   *why;
{
    if (why)
            remark(why);
    error("usage: %s [-apuU] [-l load] host ports...", progname);
}

/* scan_ports - scan ranges of ports */

scan_ports(service)
char   *service;
{
    char   *cp;
    int     min_port;
    int     max_port;
    int     port;
    struct port_info *sp;

    if (service == test_port)
            test_portno = atoi(test_port);

    /*
     * Translate service argument to range of port numbers.
     */
    if ((cp = strchr(service, '-')) != 0) {
            *cp++ = 0;
            min_port = (service[0] ? ntohs(find_port(service, "udp")) : 1);
            max_port = (cp[0] ? ntohs(find_port(cp, "udp")) : 65535);
    } else {
            min_port = max_port = ntohs(find_port(service, "udp"));
    }

    /*
     * Iterate over each port in the given range. Adjust the number of
     * simultaneous probes to the capacity of the network.
     */
    for (port = min_port; port <= max_port; port++) {
            sp = add_port(port);
            write_port(sp);
            monitor_ports();
    }
}

/* monitor_ports - watch for socket activity */

monitor_ports()
{
    do {
            struct port_info *sp;

            /*
             * When things become quiet, examine the port that we haven't looked
             * at for the longest period of time.
             */
            receive_answers();

            if (ports_busy == 0)
               return;

            sp = (struct port_info *) ring_succ(&active_ports);
```

```
            if (sp->pkts > avg_pkts * AVG_MARGIN) {
                report_and_drop_port(sp, 0);
            } else

                /*
                 * Strategy depends on whether transit times dominate (probe
                 * multiple ports in parallel, retransmit when no reply was
                 * received for at least one round-trip period) or by dead time
                 * (probe one port at a time, retransmit when no reply was
                 * received for some fraction of the inter-reply period).
                 */
                if (sock_age(sp) > (avg_rtt == 0 ? 1 :
                                    2 * avg_rtt < avg_irt ? avg_irt / 4 :
                                    1.5 * avg_rtt)) {
                    write_port(sp);
                }

                /*
                 * When all ports being probed seem to be active, send a test probe
                 * to see if the host is still alive.
                 */
                if (time_since(last_reply) > 3 * (avg_rtt == 0 ? 1 :
                                                  avg_rtt < avg_irt ? avg_irt : avg_rtt)
                    && find_port_info(test_portno) == 0) {
                    last_reply = now;
                    write_port(add_port(test_portno));
                }
    } while (ports_busy && (ports_busy >= hard_limit
                            || ports_busy >= probes_done
                            || ports_busy >= soft_limit));
}

/* receive_answers - receive reactions to probes */

receive_answers()
{
    fd_set  read_mask;
    struct timeval waitsome;
    double  delay;
    int     answers;

    /*
     * The timeout is less than the inter-reply arrival time or we would not
     * be able to increase the load.
     */
    delay = (2 * avg_rtt < avg_irt ? avg_irt / 3 : avg_rtt / (1 + ports_busy * 4));
    waitsome.tv_sec = delay;
    waitsome.tv_usec = (delay - waitsome.tv_sec) * 1000000;

    read_mask = icmp_sock_mask;
    if ((answers = select(icmp_sock + 1, &read_mask, (fd_set *) 0, (fd_set *) 0,
                          &waitsome)) < 0)
            error("select: %m");

    gettimeofday(&now, (struct timezone *) 0);

    /*
     * For each answer that we receive without retransmissions, update the
     * average roundtrip time.
     */
    if (answers > 0) {
            if (FD_ISSET(icmp_sock, &read_mask))
                receive_icmp(icmp_sock);
    }
    return (answers);
}

/* receive_icmp - receive and decode ICMP message */

receive_icmp(sock)
```

```
int    sock;
{
   union {
            char    chars[BUFSIZ];
            struct ip ip;
   }       buf;
   int     data_len;
   int     hdr_len;
   struct ip *ip;
   struct icmp *icmp;
   struct udphdr *udp;
   struct port_info *sp;

   if ((data_len = recv(sock, (char *) &buf, sizeof(buf), 0)) < 0) {
            error("error: recv: %m");
            return;
   }

   /*
    * Extract the IP header.
    */
   ip = &buf.ip;
   if (ip->ip_p != IPPROTO_ICMP) {
            error("error: not ICMP proto (%d)", ip->ip_p);
            return;
   }

   /*
    * Extract the IP payload.
    */
   hdr_len = ip->ip_hl << 2;
   if (data_len - hdr_len < ICMP_MINLEN) {
            remark("short ICMP packet (%d bytes)", data_len);
            return;
   }
   icmp = (struct icmp *) ((char *) ip + hdr_len);
   data_len -= hdr_len;

   if (icmp->icmp_type != ICMP_UNREACH)
            return;

   /*
    * Extract the offending IP header.
    */
   if (data_len < offsetof(struct icmp, icmp_ip) + sizeof(icmp->icmp_ip)) {
            remark("short IP header in ICMP");
            return;
   }
   ip = &(icmp->icmp_ip);
   if (ip->ip_p != IPPROTO_UDP)
            return;
   if (ip->ip_dst.s_addr != sin.sin_addr.s_addr)
            return;

   /*
    * Extract the offending UDP header.
    */
   hdr_len = ip->ip_hl << 2;
   udp = (struct udphdr *) ((char *) ip + hdr_len);
   data_len -= hdr_len;
   if (data_len < sizeof(struct udphdr)) {
            remark("short UDP header in ICMP");
            return;
   }

   /*
    * Process ICMP subcodes.
    */
   switch (icmp->icmp_code) {
   case ICMP_UNREACH_NET:
```

```
            error("error: network unreachable");
            /* NOTREACHED */
    case ICMP_UNREACH_HOST:
            if (sp = find_port_info(ntohs(udp->uh_dport)))
                process_reply(sp, EHOSTUNREACH);
            break;
    case ICMP_UNREACH_PROTOCOL:
            error("error: protocol unreachable");
            /* NOTREACHED */
    case ICMP_UNREACH_PORT:
            if (sp = find_port_info(ntohs(udp->uh_dport)))
                process_reply(sp, ECONNREFUSED);
            break;
    }
}

/* process_reply - process reply */

process_reply(sp, err)
struct port_info *sp;
int     err;
{
    double  age = sock_age(sp);
    int     pkts = sp->pkts;
    double  irt = time_since(last_reply);

    /*
     * Don't believe everything.
     */
    if (age > 5) {
            age = 5;
    } else if (age < 0) {
            age = 1;
    }
    if (irt > 5) {
            irt = 5;
    } else if (irt < 0) {
            irt = 1;
    }

    /*
     * We jump some hoops for calibration purposes. First we estimate the
     * round-trip time: we use this to decide when to retransmit when network
     * transit time dominates.
     *
     * Next thing to do is to estimate the inter-reply time, in case the sender
     * has a "dead time" for ICMP replies; I have seen this happen with some
     * Cisco routers and with Solaris 2.4. The first reply will come fast;
     * subsequent probes will be ignored for a period of up to one second.
     * When this happens the retransmission period should be based on the
     * inter-reply time and not on the average round-trip time.
     */
    last_reply = now;
    replies++;
    if (pkts == 1)
            avg_rtt = (avg_rtt == 0 ? age :                 /* adopt initial rtt */
                       average(age, avg_rtt));              /* normal processing */
    avg_irt = (avg_irt == 0 ? 1 :                   /* prepare for irt
                                                             * calibration */
               avg_irt == 1 ? irt :                 /* adopt initial irt */
               average(irt, avg_irt));              /* normal processing */
    avg_pkts = average((double) pkts, avg_pkts);
    if (verbose)
            printf("%d:age %.3f irt %.3f pkt %d ports %2d soft %2d done %2d avrtt %.3f avpkt %.3f avirt %.3f\n",
                sp->port, age, irt, pkts,
                ports_busy, soft_limit,
                probes_done, avg_rtt, avg_pkts, avg_irt);
    report_and_drop_port(sp, err);
}
```

```
/* report_and_drop_port - report what we know about this service */

report_and_drop_port(sp, err)
struct port_info *sp;
int     err;
{
   struct servent *se;

   if (probes_done == 0) {
            if (err == 0)
               error("are we talking to a dead host or network?");
   } else if (show_all || want_err == err || (want_err < 0 && want_err != ~err)) {
            printf("%d:%s:", sp->port,
                (se = getservbyport(htons(sp->port), "udp")) ?
                se->s_name : "UNKNOWN");
            if (err && show_all)
               printf("%s", strerror(err));
            printf("\n");
            fflush(stdout);
   }
   drop_port(sp);
}

/* average - quick-rise, slow-decay moving average */

double  average(new, old)
double  new;
double  old;
{
   if (new > old) {                                    /* quick rise */
            return ((new + old) / 2);
   } else {                                            /* slow decay */
            return (0.1 * new + 0.9 * old);
   }
}

/* add_port - say this port is being probed */

struct port_info *add_port(port)
int     port;
{
   struct port_info *sp = (struct port_info *) ring_succ(&dead_ports);

   ring_detach((RING *) sp);
   sp->port = port;
   sp->pkts = 0;
   ports_busy++;
   ring_append(&active_ports, (RING *) sp);
   return (sp);
}

/* write_port - write to port, update statistics */

write_port(sp)
struct port_info *sp;
{
   char    ch = 0;

   ring_detach((RING *) sp);
   sin.sin_port = htons(sp->port);
   sp->last_probe = now;
   sendto(send_sock, &ch, 1, 0, (struct sockaddr *) & sin, sizeof(sin));
   probes_sent++;
   sp->pkts++;
   ring_prepend(&active_ports, (RING *) sp);

   /*
    * Reduce the sending window when the first retransmission happens. Back
    * off when retransmissions dominate. Occasional retransmissons will keep
    * the load unchanged.
```

```
    */
    if (sp->pkts > 1) {
            replies--;
            if (soft_limit > hard_limit) {
                soft_limit = (ports_busy + 1) / 2;
            } else if (replies < 0 && avg_irt) {
                soft_limit = 0.5 + 0.5 * (soft_limit + avg_rtt / avg_irt);
                replies = soft_limit / 2;
            }
    }
}

/* drop_port - release port info, update statistics */

drop_port(sp)
struct port_info *sp;
{
    ports_busy--;
    probes_done++;
    ring_detach((RING *) sp);
    ring_append(&dead_ports, (RING *) sp);

    /*
     * Increase the load when a sufficient number of probes succeeded.
     * Occasional retransmissons will keep the load unchanged.
     */
    if (replies > soft_limit) {
            replies = soft_limit / 2;
            if (soft_limit < hard_limit)
                soft_limit++;
    }
}

/* init_port_info - initialize port info pool */

init_port_info()
{
    struct port_info *sp;

    port_info = (struct port_info *) mymalloc(hard_limit * sizeof(*port_info));
    ring_init(&active_ports);
    ring_init(&dead_ports);
    for (sp = port_info; sp < port_info + hard_limit; sp++)
            ring_append(&dead_ports, (RING *) sp);
}

/* find_port_info - lookup port info */

struct port_info *find_port_info(port)
int     port;
{
    struct port_info *sp;

    for (sp = (struct port_info *) ring_succ(&active_ports);
             sp != (struct port_info *) & active_ports;
             sp = (struct port_info *) ring_succ((RING *) sp))
            if (sp->port == port)
                return (sp);
    return (0);
}
```

Sources

"CVE-2000-0289 " 13 Oct 2000. URL: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0289 (13, March 2001).

Fyodor. "The Art of Port Scanning" 1997 Sept 6. URL: http://www.insecure.org/nmap/nmap_doc.html (20 March 2001).

JWS. "tcpdump/libpcap" 6 Feb 2001. URL: http://www.tcpdump.org/ (23 March 2001).

Moore, H D. "Multiple Linux Vendor 2.2.x Kernel IP Masquerading Vulnerabilities" 10 November 2000. URL: http://www.securityfocus.com/bid/1078 (11 March 2001).

Postel, J. "User Datagram Protocol " 28 August 1980. URL: http://www.faqs.org/rfcs/rfc768.html (18 March 2001).

Purdue University. "CERIAS - Security Archive" 14 June 2000. URL: ftp://coast.cs.purdue.edu/pub/tools/ (23 March 2001).

Ranch, David. "Linux IP Masquerade HOWTO" 14 Nov 2000. URL: http://www.europe.redhat.com/documentation/HOWTO/IP-Masquerade-HOWTO.php3 (18 March 2001).

"Red Hat, Inc. Bug Fix Advisory" 28 April 2000. URL: http://www.linuxsecurity.com/advisories/redhat_advisory-413.html (15 March 2001).

"SecurityFocus.com: Multiple Linux Vendor 2.2.x Kernel IP Masquerading Vulnerabilities." 27 April 2000. URL: http://linuxtoday.com/news_story.php3?ltsn=2000-04-27-033-04-SC (3 March 2001).

"SuSE Security Announcement" 17 May 2000. URL: http://www.suse.de/de/support/security/suse_security_announce_48.txt (13 March 2001).

Last Updated: June 24th, 2018

Upcoming SANS Penetration Testing

SANS Minneapolis 2018	Minneapolis, MN	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS Cyber Defence Canberra 2018	Canberra, Australia	Jun 25, 2018 - Jul 07, 2018	Live Event
SANS Vancouver 2018	Vancouver, BC	Jun 25, 2018 - Jun 30, 2018	Live Event
SANS London July 2018	London, United Kingdom	Jul 02, 2018 - Jul 07, 2018	Live Event
SANS Charlotte 2018	Charlotte, NC	Jul 09, 2018 - Jul 14, 2018	Live Event
SANS Cyber Defence Singapore 2018	Singapore, Singapore	Jul 09, 2018 - Jul 14, 2018	Live Event
Mentor Session - SEC504	Oklahoma City, OK	Jul 10, 2018 - Sep 11, 2018	Mentor
SANSFIRE 2018	Washington, DC	Jul 14, 2018 - Jul 21, 2018	Live Event
SANSFIRE 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
SANSFIRE 2018 - SEC542: Web App Penetration Testing and Ethical Hacking	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
SANSFIRE 2018 - SEC560: Network Penetration Testing and Ethical Hacking	Washington, DC	Jul 16, 2018 - Jul 21, 2018	vLive
SANS Pen Test Berlin 2018	Berlin, Germany	Jul 23, 2018 - Jul 28, 2018	Live Event
SANS vLive - SEC560: Network Penetration Testing and Ethical Hacking	SEC560 - 201807,	Jul 24, 2018 - Aug 30, 2018	vLive
SANS Pittsburgh 2018	Pittsburgh, PA	Jul 30, 2018 - Aug 04, 2018	Live Event
San Antonio 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	San Antonio, TX	Aug 06, 2018 - Aug 11, 2018	vLive
SANS San Antonio 2018	San Antonio, TX	Aug 06, 2018 - Aug 11, 2018	Live Event
Security Awareness Summit & Training 2018	Charleston, SC	Aug 06, 2018 - Aug 15, 2018	Live Event
SANS Boston Summer 2018	Boston, MA	Aug 06, 2018 - Aug 11, 2018	Live Event
Mentor Session - AW SEC560	Austin, TX	Aug 08, 2018 - Oct 10, 2018	Mentor
Northern Virginia- Alexandria 2018 - SEC542: Web App Penetration Testing and Ethical Hacking	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	vLive
Community SANS Ventura SEC560	Ventura, CA	Aug 13, 2018 - Aug 18, 2018	Community SANS
SANS Northern Virginia- Alexandria 2018	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	Live Event
Northern Virginia- Alexandria 2018 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Alexandria, VA	Aug 13, 2018 - Aug 18, 2018	vLive
SANS New York City Summer 2018	New York City, NY	Aug 13, 2018 - Aug 18, 2018	Live Event
SANS Virginia Beach 2018	Virginia Beach, VA	Aug 20, 2018 - Aug 31, 2018	Live Event
SANS Chicago 2018	Chicago, IL	Aug 20, 2018 - Aug 25, 2018	Live Event
SANS Prague 2018	Prague, Czech Republic	Aug 20, 2018 - Aug 25, 2018	Live Event
Community SANS Reno SEC504	Reno, NV	Aug 20, 2018 - Aug 25, 2018	Community SANS
SANS Krakow 2018	Krakow, Poland	Aug 20, 2018 - Aug 25, 2018	Live Event
Mentor Session - SEC504	Cincinnati, OH	Aug 21, 2018 - Oct 02, 2018	Mentor
Mentor Session - SEC542	Denver, CO	Aug 23, 2018 - Oct 25, 2018	Mentor