NMAP - Port-Scanning: A Practical Approach Modified for better


```
----Port-Scanning: A Practical Approach Modified for better
------------------------------------------------------------
I accept that when i got this file that was called nmapguide.txt it is written by Doug Hoyte a senior
programmer and i liked to add some information for the past years
that nmap has been a evolution on protscanning since 1997.I have added here the mos used commands for
penetesters and so on for hackers.Im not saying that im helping
hackers or the bad guys to get into systemsor get into troubles, this is a paper that should be read very
carefully.It has too many infos for you........

Thanks
Version 2.0
I took at about 7 days to edit, add, remove, and unduplicate the information.
Port Scanning is not a crime and will not be till the end.
Author(s):
------|Florian MINDZSEC|---------- - Skilled Programmer

------------------------------
* Introduction
* History
* What can nmap Do:
* Your arsenal
* Fundamentals
* Port scanning
* Practical Scanning
* Scanning with NEW Scripts
* Cheat Sheet
* Nmap in your hands
* Script Engine Scanning
------------------------------

Introduction
------------

Often times it is useful, even necessary, to gather as much information as possible
```

about a remote target. This includes learning all of their network "points of entry", the operating systems used, firewalling methods employed, services running, etc.

Note that while it certainly is possible to portscan with a windows machine, I will be focusing on using a unix machine with certain utilities installed. This is due to Windows' lack of raw socket access (pre Win2K) and the lack of decent, free, portscanners availble for the platform. In the next section I will share some useful pointers on portscanning. Note that root level access is required on your unix machine for many scans.

History

The nmap is first released in 1997 in Phrack Magazine issue 51, article 11.
Some information:[Abstract]

This paper details many of the techniques used to determine what ports (or similar protocol abstraction) of a host are listening for connections. These ports represent potential communication channels. Mapping their existence facilitates the exchange of information with the host, and thus it is quite useful for anyone wishing to explore their networked environment, including hackers. Despite what you have heard from the media, the Internet is NOT all about TCP port 80. Anyone who relies exclusively on the WWW for information gathering is likely to gain the same level of proficiency as your average AOLer, who does the same. This paper is also meant to serve as an introduction to and ancillary documentation for a coding project I have been working on. It is a full featured, robust port scanner which (I hope) solves some of the problems I have encountered when dealing with other scanners and when working to scan massive networks. The tool, nmap, supports the following:

- 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), and
- reverse-ident scanning.

The freely distributable source code is appended to this paper.

[Introduction]

Scanning, as a method for discovering exploitable communication channels, has been around for ages. The idea is to probe as many listeners as possible, and keep track of the ones that are receptive or useful to your particular need. Much of the field of advertising is based on this paradigm, and the "to current

resident" brute force style of bulk mail is an almost perfect parallel to what we will discuss. Just stick a message in every mailbox and wait for the responses to trickle back.

Scanning entered the h/p world along with the phone systems. Here we have this tremendous global telecommunications network, all reachable through codes on our telephone. Millions of numbers are reachable locally, yet we may only be interested in 0.5% of these numbers, perhaps those that answer with a carrier.

The logical solution to finding those numbers that interest us is to try them all. Thus the field of "wardialing" arose. Excellent programs like Toneloc were developed to facilitate the probing of entire exchanges and more. The basic idea is simple. If you dial a number and your modem gives you a CONNECT, you record it. Otherwise the computer hangs up and tirelessly dials the next one.

While wardialing is still useful, we are now finding that many of the computers we wish to communicate with are connected through networks such as the Internet rather than analog phone dialups. Scanning these machines involves the same brute force technique. We send a blizzard of packets for various protocols, and we deduce which services are listening from the responses we receive (or don't receive).

[Techniques]

Over time, a number of techniques have been developed for surveying the protocols and ports on which a target machine is listening. They all offer different benefits and problems. Here is a line up of the most common:

- TCP connect() scanning : This is the most basic form of TCP scanning. The connect() system call provided by your operating system is used to open a connection to every interesting port on the machine. If the port is listening, connect() will succeed, otherwise the port isn't reachable. One strong advantage to this technique is that you don't need any special privileges. Any user on most UNIX boxes is free to use this call. Another advantage is speed. While making a separate connect() call for every targeted port in a linear fashion would take ages over a slow connection, you can hasten the scan by using many sockets in parallel. Using non-blocking I/O allows you to set a low time-out period and watch all the sockets at once. This is the fastest scanning method supported by nmap, and is available with the -t (TCP) option. The big downside is that this sort of scan is easily detectable and filterable. The target hosts logs will show a bunch of connection and error messages for the services which take the connection and then have it immediately shutdown.

- TCP SYN scanning : This technique is often referred to as "half-open" scanning, because you don't open a full TCP connection. You send a SYN packet, as if you are going to open a real connection and wait for a response. A SYN|ACK indicates the port is listening. A RST is indicative of a non-listener. If a SYN|ACK is received, you immediately send a RST to tear down the connection (actually the kernel does this for us). The primary advantage to this scanning technique is that fewer sites will log it. Unfortunately you need root privileges to build these custom SYN packets. SYN scanning is the -s option of nmap.

- TCP FIN scanning : There are times when even SYN scanning isn't clandestine enough. Some firewalls and packet filters watch for SYNs to an unallowed port, and programs like synlogger and Courtney are available to detect these scans. FIN packets, on the other hand, may be able to pass through unmolested. This scanning technique was featured in detail by Uriel Maimon in Phrack 49, article 15. The idea is that closed ports tend to reply to your FIN packet with the proper RST. Open ports, on the other hand, tend to ignore the packet in question. This is a bug in TCP implementations and so it isn't 100% reliable (some systems, notably Micro$oft boxes, seem to be immune). It works well on most other systems I've tried. FIN scanning is the -U (Uriel) option of nmap.

- Fragmentation scanning : This is not a new scanning method in and of itself, but a modification of other techniques. Instead of just sending the probe packet, you break it into a couple of small IP fragments. You are splitting up the TCP header over several packets to make it harder for packet filters and so forth to detect what you are doing. Be careful with this! Some programs have trouble handling these tiny packets. My favorite sniffer segmentation faulted immediately upon receiving the first 36-byte fragment. After that comes a 24 byte one! While this method won't get by packet filters and firewalls that queue all IP fragments (like the CONFIG_IP_ALWAYS_DEFRAG option in Linux), a lot of networks can't afford the performance hit this causes. This feature is rather unique to scanners (at least I haven't seen any others that do this). Thanks to daemon9 for suggesting it. The -f instructs the specified SYN or FIN scan to use tiny fragmented packets.

- TCP reverse ident scanning : As noted by Dave Goldsmith in a 1996 Bugtraq post, the ident protocol (rfc1413) allows for the disclosure of the username of the owner of any process connected via TCP, even if that process didn't initiate the connection. So you can, for example, connect to the http port and then use identd to find out whether the server is running as root. This can only be done with a full TCP connection to the target port (i.e. the -t option). nmap's -i option queries identd for the owner of all listen()ing ports.

- FTP bounce attack : An interesting "feature" of the ftp protocol (RFC 959) is support for "proxy" ftp connections. In other words, I should be able to connect from evil.com to the FTP server-PI (protocol interpreter) of target.com to establish the control communication connection. Then I should be able to request that the server-PI initiate an active server-DTP (data transfer process) to send a file ANYWHERE on the internet! Presumably to a User-DTP, although the RFC specifically states that asking one server to send a file to another is OK. Now this may have worked well in 1985 when the RFC was just written. But nowadays, we can't have people hijacking ftp servers and requesting that data be spit out to arbitrary points on the internet. As *Hobbit* wrote back in 1995, this protocol flaw "can be used to post virtually untraceable mail and news, hammer on servers at various sites, fill up disks, try to hop firewalls, and generally be annoying and hard to track down at the same time." What we will exploit this for is to (surprise, surprise) scan TCP ports from a "proxy" ftp server. Thus you could connect to an ftp server behind a firewall, and then scan ports that are more likely to be blocked (139 is a good one). If the ftp server allows reading from and writing to a directory (such as /incoming), you can send arbitrary data to ports that you do find open.

For port scanning, our technique is to use the PORT command to declare that our passive "User-DTP" is listening on the target box at a certain port number. Then we try to LIST the current directory, and the result is sent over the Server-DTP channel. If our target host is listening on the specified port, the transfer will be successful (generating a 150 and a 226 response). Otherwise we will get "425 Can't build data connection: Connection refused." Then we issue another PORT command to try the next port on the target host. The advantages to this approach are obvious (harder to trace, potential to bypass firewalls). The main disadvantages are that it is slow, and that some FTP servers have finally got a clue and disabled the proxy "feature". For what it is worth, here is a list of banners from sites where it does/doesn't work:

Bounce attacks worked:

```
220 xxxxxxx.com FTP server (Version wu-2.4(3) Wed Dec 14 ...) ready.
220 xxx.xxx.xxx.edu FTP server ready.
220 xx.Telcom.xxxx.EDU FTP server (Version wu-2.4(3) Tue Jun 11 ...) ready.
220 lem FTP server (SunOS 4.1) ready.
220 xxx.xxx.es FTP server (Version wu-2.4(11) Sat Apr 27 ...) ready.
220 elios FTP server (SunOS 4.1) ready
```

Bounce attack failed:

```
220 wcarchive.cdrom.com FTP server (Version DG-2.0.39 Sun May 4 ...) ready.
220 xxx.xx.xxxxx.EDU Version wu-2.4.2-academ[BETA-12](1) Fri Feb 7
220 ftp Microsoft FTP Service (Version 3.0).
220 xxx FTP server (Version wu-2.4.2-academ[BETA-11](1) Tue Sep 3 ...) ready.
220 xxx.unc.edu FTP server (Version wu-2.4.2-academ[BETA-13](6) ...) ready.
```

The 'x's are partly there to protect those guilty of running a flawed server, but mostly just to make the lines fit in 80 columns. Same thing with the ellipse points. The bounce attack is available with the -b <proxy_server> option of nmap. proxy_server can be specified in standard URL format, username:password@server:port , with everything but server being optional.

- UDP ICMP port unreachable scanning : This scanning method varies from the above in that we are using the UDP protocol instead of TCP. While this protocol is simpler, scanning it is actually significantly more difficult. This is because open ports don't have to send an acknowledgement in response to our probe, and closed ports aren't even required to send an error packet. Fortunately, most hosts do send an ICMP_PORT_UNREACH error when you send a packet to a closed UDP port. Thus you can find out if a port is NOT open, and by exclusion determine which ports which are. Neither UDP packets, nor the ICMP errors are guaranteed to arrive, so UDP scanners of this sort must also implement retransmission of packets that appear to be lost (or you will get a bunch of false positives). Also, this scanning technique is slow because of compensation for machines that took RFC 1812 section 4.3.2.8 to heart and limit ICMP error message rate. For example, the Linux kernel (in net/ipv4/icmp.h) limits destination unreachable message generation to 80 per 4 seconds, with a 1/4 second penalty if that is exceeded. At some point I will add a better algorithm to nmap for detecting this. Also, you will need to be root for access to the raw ICMP socket necessary for reading the port unreachable. The -u (UDP) option of nmap implements this scanning method for root users.

Some people think UDP scanning is lame and pointless. I usually remind them of the recent Solaris rcpbind hole. Rpcbind can be found hiding on an undocumented UDP port somewhere above 32770. So it doesn't matter that 111 is blocked by the firewall. But can you find which of the more than 30,000 high ports it is listening on? With a UDP scanner you can!

- UDP recvfrom() and write() scanning : While non-root users can't read port unreachable errors directly, Linux is cool enough to inform the user indirectly when they have been received. For example a second write() call to a closed port will usually fail. A lot of scanners such as netcat and Pluvius' pscan.c does this. I have also noticed that recvfrom() on non-blocking UDP sockets usually return EAGAIN ("Try Again", errno 13) if the ICMP error hasn't been received, and ECONNREFUSED ("Connection refused", errno 111) if it has. This is the technique used for determining open ports when non-root users use -u (UDP). Root users can also use the -l (lamer UDP scan) options to force this, but it is a really dumb idea.

- ICMP echo scanning : This isn't really port scanning, since ICMP doesn't have a port abstraction. But it is sometimes useful to determine what hosts in a

network are up by pinging them all. the -P option does this. Also you might want to adjust the PING_TIMEOUT #define if you are scanning a large network. nmap supports a host/bitmask notation to make this sort of thing easier. For example 'nmap -P cert.org/24 152.148.0.0/16' would scan CERT's class C network and whatever class B entity 152.148.* represents. Host/26 is useful for 6-bit subnets within an organization.

[Features]

Prior to writing nmap, I spent a lot of time with other scanners exploring the Internet and various private networks (note the avoidance of the "intranet" buzzword). I have used many of the top scanners available today, including strobe by Julian Assange, netcat by *Hobbit*, stcp by Uriel Maimon, pscan by Pluvius, ident-scan by Dave Goldsmith, and the SATAN tcp/udp scanners by Wietse Venema. These are all excellent scanners! In fact, I ended up hacking most of them to support the best features of the others. Finally I decided to write a whole new scanner, rather than rely on hacked versions of a dozen different scanners in my /usr/local/sbin. While I wrote all the code, nmap uses a lot of good ideas from its predecessors. I also incorporated some new stuff like fragmentation scanning and options that were on my "wish list" for other scanners. Here are some of the (IMHO) useful features of nmap:

- dynamic delay time calculations: Some scanners require that you supply a delay time between sending packets. Well how should I know what to use? Sure, I can ping them, but that is a pain, and plus the response time of many hosts changes dramatically when they are being flooded with requests. nmap tries to determine the best delay time for you. It also tries to keep track of packet retransmissions, etc. so that it can modify this delay time during the course of the scan. For root users, the primary technique for finding an initial delay is to time the internal "ping" function. For non-root users, it times an attempted connect() to a closed port on the target. It can also pick a reasonable default value. Again, people who want to specify a delay themselves can do so with -w (wait), but you shouldn't have to.

- retransmission: Some scanners just send out all the query packets, and collect the responses. But this can lead to false positives or negatives in the case where packets are dropped. This is especially important for "negative" style scans like UDP and FIN, where what you are looking for is a port that does NOT respond. In most cases, nmap implements a configurable number of retransmissions for ports that don't respond.

- parallel port scanning: Some scanners simply scan ports linearly, one at a time, until they do all 65535. This actually works for TCP on a very fast local network, but the speed of this is not at all acceptable on a wide area network like the Internet. nmap uses non-blocking i/o and parallel scanning in all TCP and UDP modes. The number of scans in parallel is configurable

with the -M (Max sockets) option. On a very fast network you will actually decrease performance if you do more than 18 or so. On slow networks, high values increase performance dramatically.

- Flexible port specification: I don't always want to just scan all 65535 ports. Also, the scanners which only allow you to scan ports 1 - N sometimes fall short of my need. The -p option allows you to specify an arbitrary number of ports and ranges for scanning. For example, '-p 21-25,80,113, 60000-' does what you would expect (a trailing hyphen means up to 65536, a leading hyphen means 1 through). You can also use the -F (fast) option, which scans all the ports registered in your /etc/services (a la strobe).

- Flexible target specification: I often want to scan more then one host, and I certainly don't want to list every single host on a large network to scan. Everything that isn't an option (or option argument) in nmap is treated as a target host. As mentioned before, you can optionally append /mask to a hostname or IP address in order to scan all hosts with the same initial <mask> bits of the 32 bit IP address.

- detection of down hosts: Some scanners allow you to scan large networks, but they waste a huge amount of time scanning 65535 ports of a dead host! By default, nmap pings each host to make sure it is up before wasting time on it. It is also capable of bailing on hosts that seem down based on strange port scanning errors. It is also meant to be tolerant of people who accidentally scan network addresses, broadcast addresses, etc.

- detection of your IP address: For some reason, a lot of scanners ask you to type in your IP address as one of the parameters. Jeez, I don't want to have to 'ifconfig' and figure out my current address every time I scan. Of course, this is better then the scanners I've seen which require recompilation every time you change your address! nmap first tries to detect your address during the ping stage. It uses the address that the echo response is received on, as that is the interface it should almost always be routed through. If it can't do this (like if you don't have host pinging enabled), nmap tries to detect your primary interface and uses that address. You can also use -S to specify it directly, but you shouldn't have to (unless you want to make it look like someone ELSE is SYN or FIN scanning a host.

Some other, more minor options:

 -v (verbose): This is highly recommended for interactive use. Among other useful messages, you will see ports come up as they are found, rather than having to wait for the sorted summary list.

 -r (randomize): This will randomize the order in which the target host's ports are scanned.

```
 -q (quash argv): This changes argv[0] to FAKE_ARGV ("pine" by default).
It also eliminates all other arguments, so you won't look too suspicious in
'w' or 'ps' listings.

 -h for an options summary.


<++> nmap/Makefile
# A trivial makefile for Network Mapper
nmap: nmap.c nmap.h
        gcc -Wall -O6 -o nmap nmap.c -lm
<-->

<++> nmap/nmap.h
#ifndef NMAP_H
#define NMAP_H

/***********************INCLUDES***********************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <rpc/types.h>
#include <sys/socket.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>
#include <time.h>
#include <fcntl.h>
#include <signal.h>
#include <signal.h>
#include <linux/ip.h> /*<netinet/ip.h>*/
#include <linux/icmp.h> /*<netinet/ip_icmp.h>*/
#include <arpa/inet.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <asm/byteorder.h>
#include <netinet/ip_tcp.h>

/***********************DEFINES************************************/

/* #define to zero if you don't want to ignore hosts of the form
   xxx.xxx.xxx.{0,255} (usually network and broadcast addresses) */
#define IGNORE_ZERO_AND_255_HOSTS 1
```

```
#define DEBUGGING 0

/* Default number of ports in paralell.  Doesn't always involve actual
   sockets.  Can also adjust with the -M command line option.  */
#define MAX_SOCKETS 36
/* If reads of a UDP port keep returning EAGAIN (errno 13), do we want to
   count the port as valid? */
#define RISKY_UDP_SCAN 0
 /* This ideally should be a port that isn't in use for any protocol on our machine or on the target */
#define MAGIC_PORT 49724
/* How many udp sends without a ICMP port unreachable error does it take before we consider the port open?
*/
#define UDP_MAX_PORT_RETRIES 4
 /*How many seconds before we give up on a host being alive? */
#define PING_TIMEOUT 2
#define FAKE_ARGV "pine" /* What ps and w should show if you use -q */
/* How do we want to log into ftp sites for */
#define FTPUSER "anonymous"
#define FTPPASS "-wwwuser@"
#define FTP_RETRIES 2 /* How many times should we relogin if we lose control
                         connection? */

#define UC(b)   (((int)b)&0xff)
#define MORE_FRAGMENTS 8192 /*NOT a user serviceable parameter*/
#define fatal(x) { fprintf(stderr, "%s\n", x); exit(-1); }
#define error(x) fprintf(stderr, "%s\n", x);

/**********************STRUCTURES**********************************/

typedef struct port {
  unsigned short portno;
  unsigned char proto;
  char *owner;
  struct port *next;
} port;

struct ftpinfo {
  char user[64];
  char pass[256]; /* methinks you're paranoid if you need this much space */
  char server_name[MAXHOSTNAMELEN + 1];
  struct in_addr server;
  unsigned short port;
  int sd; /* socket descriptor */
};

typedef port *portlist;

/**********************PROTOTYPES**********************************/
```

```
/* print usage information */
void printusage(char *name);

/* our scanning functions */
portlist tcp_scan(struct in_addr target, unsigned short *portarray,
                  portlist *ports);
portlist syn_scan(struct in_addr target, unsigned short *portarray,
                  struct in_addr *source, int fragment, portlist *ports);
portlist fin_scan(struct in_addr target, unsigned short *portarray,
                  struct in_addr *source, int fragment, portlist *ports);
portlist udp_scan(struct in_addr target, unsigned short *portarray,
                  portlist *ports);
portlist lamer_udp_scan(struct in_addr target, unsigned short *portarray,
                        portlist *ports);
portlist bounce_scan(struct in_addr target, unsigned short *portarray,
                     struct ftpinfo *ftp, portlist *ports);

/* Scan helper functions */
unsigned long calculate_sleep(struct in_addr target);
int check_ident_port(struct in_addr target);
int getidentinfoz(struct in_addr target, int localport, int remoteport,
                  char *owner);
int parse_bounce(struct ftpinfo *ftp, char *url);
int ftp_anon_connect(struct ftpinfo *ftp);

/* port manipulators */
unsigned short *getpts(char *expr); /* someone stole the name getports()! */
unsigned short *getfastports(int tcpscan, int udpscan);
int addport(portlist *ports, unsigned short portno, unsigned short protocol,
            char *owner);
int deleteport(portlist *ports, unsigned short portno, unsigned short protocol);
void printandfreeports(portlist ports);
int shortfry(unsigned short *ports);

/* socket manipulation functions */
void init_socket(int sd);
int unblock_socket(int sd);
int block_socket(int sd);
int recvtime(int sd, char *buf, int len, int seconds);

/* RAW packet building/dissasembling stuff */
int send_tcp_raw( int sd, struct in_addr *source,
                  struct in_addr *victim, unsigned short sport,
                  unsigned short dport, unsigned long seq,
                  unsigned long ack, unsigned char flags,
                  unsigned short window, char *data,
                  unsigned short datalen);
```

```
int isup(struct in_addr target);
unsigned short in_cksum(unsigned short *ptr,int nbytes);
int send_small_fragz(int sd, struct in_addr *source, struct in_addr *victim,
                     int sport, int dport, int flags);
int readtcppacket(char *packet, int readdata);
int listen_icmp(int icmpsock, unsigned short outports[],
                unsigned short numtries[], int *num_out,
                struct in_addr target, portlist *ports);

/* general helper functions */
void hdump(unsigned char *packet, int len);
void *safe_malloc(int size);
#endif /* NMAP_H */
<-->

<++> nmap/nmap.c

#include "nmap.h"

/* global options */
short debugging = DEBUGGING;
short verbose = 0;
int number_of_ports  = 0; /* How many ports do we scan per machine? */
int max_parallel_sockets = MAX_SOCKETS;
extern char *optarg;
extern int optind;
short isr00t = 0;
short identscan = 0;
char current_name[MAXHOSTNAMELEN + 1];
unsigned long global_delay = 0;
unsigned long global_rtt = 0;
struct in_addr ouraddr = { 0 };

int main(int argc, char *argv[]) {
int i, j, arg, argvlen;
short fastscan=0, tcpscan=0, udpscan=0, synscan=0, randomize=0;
short fragscan = 0, finscan = 0, quashargv = 0, pingscan = 0, lamerscan = 0;
short bouncescan = 0;
short *ports = NULL, mask;
struct ftpinfo ftp = { FTPUSER, FTPPASS, "", { 0 }, 21, 0};
portlist openports = NULL;
struct hostent *target = 0;
unsigned long int lastip, currentip, longtmp;
char *target_net, *p;
struct in_addr current_in, *source=NULL;
int hostup = 0;
char *fakeargv[argc + 1];
```

```
/* argv faking silliness */
for(i=0; i < argc; i++) {
  fakeargv[i] = safe_malloc(strlen(argv[i]) + 1);
  strncpy(fakeargv[i], argv[i], strlen(argv[i]) + 1);
}
fakeargv[argc] = NULL;

if (argc < 2 ) printusage(argv[0]);

/* OK, lets parse these args! */
while((arg = getopt(argc,fakeargv,"b:dFfhilM:Pp:qrS:stUuw:v")) != EOF) {
  switch(arg) {
  case 'b':
    bouncescan++;
    if (parse_bounce(&ftp, optarg) < 0 ) {
      fprintf(stderr, "Your argument to -b is fucked up. Use the normal url style:  user:pass@server:port
or just use server and use default anon login\n  Use -h for help\n");
    }
    break;
  case 'd': debugging++; break;
  case 'F': fastscan++; break;
  case 'f': fragscan++; break;
  case 'h':
  case '?': printusage(argv[0]);
  case 'i': identscan++; break;
  case 'l': lamerscan++; udpscan++; break;
  case 'M': max_parallel_sockets = atoi(optarg); break;
  case 'P': pingscan++; break;
  case 'p':
    if (ports)
      fatal("Only 1 -p option allowed, seperate multiple ranges with commas.");
    ports = getpts(optarg); break;
  case 'r': randomize++; break;
  case 's': synscan++; break;
  case 'S':
    if (source)
      fatal("You can only use the source option once!\n");
    source = safe_malloc(sizeof(struct in_addr));
    if (!inet_aton(optarg, source))
      fatal("You must give the source address in dotted deciman, currently.\n");
    break;
  case 't': tcpscan++; break;
  case 'U': finscan++; break;
  case 'u': udpscan++; break;
  case 'q': quashargv++; break;
  case 'w': global_delay = atoi(optarg); break;
  case 'v': verbose++;
  }
```

```
}

/* Take care of user wierdness */
isr00t = !(geteuid()|geteuid());
if (tcpscan && synscan)
  fatal("The -t and -s options can't be used together.\
 If you are trying to do TCP SYN scanning, just use -s.\
 For normal connect() style scanning, use -t");
if ((synscan || finscan || fragscan || pingscan) && !isr00t)
  fatal("Options specified require r00t privileges.  You don't have them!");
if (!tcpscan && !udpscan && !synscan && !finscan && !bouncescan && !pingscan) {
  tcpscan++;
  if (verbose) error("No scantype specified, assuming vanilla tcp connect()\
 scan. Use -P if you really don't want to portscan.");
if (fastscan && ports)
  fatal("You can use -F (fastscan) OR -p for explicit port specification.\
  Not both!\n");
}
/* If he wants to bounce of an ftp site, that site better damn well be reachable! */
if (bouncescan) {
  if (!inet_aton(ftp.server_name, &ftp.server)) {
    if ((target = gethostbyname(ftp.server_name)))
      memcpy(&ftp.server, target->h_addr_list[0], 4);
    else {
      fprintf(stderr, "Failed to resolve ftp bounce proxy hostname/IP: %s\n",
              ftp.server_name);
      exit(1);
    }
  }  else if (verbose)
    printf("Resolved ftp bounce attack proxy to %s (%s).\n",
           target->h_name, inet_ntoa(ftp.server));
}
printf("\nStarting nmap V 1.21 by Fyodor (fyodor@dhp.com, www.dhp.com/~fyodor/nmap/\n");
if (!verbose)
  error("Hint: The -v option notifies you of open ports as they are found.\n");
if (fastscan)
  ports = getfastports(synscan|tcpscan|fragscan|finscan|bouncescan,
                       udpscan|lamerscan);
if (!ports) ports = getpts("1-1024");

/* more fakeargv junk, BTW malloc'ing extra space in argv[0] doesn't work */
if (quashargv) {
  argvlen = strlen(argv[0]);
  if (argvlen < strlen(FAKE_ARGV))
    fatal("If you want me to fake your argv, you need to call the program with a longer name.  Try the full
pathname, or rename it fyodorssuperdedouperportscanner");
  strncpy(argv[0], FAKE_ARGV, strlen(FAKE_ARGV));
  for(i = strlen(FAKE_ARGV); i < argvlen; i++) argv[0][i] = '\0';
```

```
    for(i=1; i < argc; i++) {
      argvlen = strlen(argv[i]);
      for(j=0; j <= argvlen; j++)
        argv[i][j] = '\0';
    }
  }

  srand(time(NULL));

  while(optind < argc) {

    /* Time to parse the allowed mask */
    target = NULL;
    target_net = strtok(strdup(fakeargv[optind]), "/");
    mask = (p = strtok(NULL,""))? atoi(p) : 32;
    if (debugging)
      printf("Target network is %s, scanmask is %d\n", target_net, mask);

    if (!inet_aton(target_net, ¤t_in)) {
      if ((target = gethostbyname(target_net)))
        memcpy(¤tip, target->h_addr_list[0], 4);
      else {
        fprintf(stderr, "Failed to resolve given hostname/IP: %s\n", target_net);
      }
    } else currentip = current_in.s_addr;

    longtmp = ntohl(currentip);
    currentip = longtmp & (unsigned long) (0 - pow(2,32 - mask));
    lastip = longtmp | (unsigned long) (pow(2,32 - mask) - 1);
    while (currentip <= lastip) {
      openports = NULL;
      longtmp = htonl(currentip);
      target = gethostbyaddr((char *) &longtmp, 4, AF_INET);
      current_in.s_addr = longtmp;
      if (target)
        strncpy(current_name, target->h_name, MAXHOSTNAMELEN);
      else current_name[0] = '\0';
      current_name[MAXHOSTNAMELEN + 1] = '\0';
      if (randomize)
        shortfry(ports);
  #ifdef IGNORE_ZERO_AND_255_HOSTS
      if (IGNORE_ZERO_AND_255_HOSTS
          && (!(currentip % 256) || currentip % 256 == 255))
        {
          printf("Skipping host %s because IGNORE_ZERO_AND_255_HOSTS is set in the source.\n",
  inet_ntoa(current_in));
          hostup = 0;
        }
```

```
      else{
  #endif
        if (isr00t) {
          if (!(hostup = isup(current_in))) {
            if (!pingscan)
              printf("Host %s (%s) appears to be down, skipping scan.\n",
                     current_name, inet_ntoa(current_in));
            else
              printf("Host %s (%s) appears to be  down\n",
                     current_name, inet_ntoa(current_in));
          } else if (debugging || pingscan)
            printf("Host %s (%s) appears to be up ... good.\n",
                   current_name, inet_ntoa(current_in));
        }
        else hostup = 1; /* We don't really check because the lamer isn't root.*/
      }

      /* Time for some actual scanning! */
      if (hostup) {
        if (tcpscan) tcp_scan(current_in, ports, &openports);

        if (synscan) syn_scan(current_in, ports, source, fragscan, &openports);

        if (finscan) fin_scan(current_in, ports, source, fragscan, &openports);

        if (bouncescan) {
          if (ftp.sd <= 0) ftp_anon_connect(&ftp);
          if (ftp.sd > 0) bounce_scan(current_in, ports, &ftp, &openports);
            }
        if (udpscan) {
          if (!isr00t || lamerscan)
            lamer_udp_scan(current_in, ports, &openports);

          else udp_scan(current_in, ports, &openports);
        }

        if (!openports && !pingscan)
          printf("No ports open for host %s (%s)\n", current_name,
                 inet_ntoa(current_in));
        if (openports) {
          printf("Open ports on %s (%s):\n", current_name,
                 inet_ntoa(current_in));
          printandfreeports(openports);
        }
      }
      currentip++;
    }
    optind++;
```

```
}

return 0;
}

__inline__ int unblock_socket(int sd) {
int options;
/*Unblock our socket to prevent recvfrom from blocking forever
  on certain target ports. */
options = O_NONBLOCK | fcntl(sd, F_GETFL);
fcntl(sd, F_SETFL, options);
return 1;
}

__inline__ int block_socket(int sd) {
int options;
options = (~O_NONBLOCK) & fcntl(sd, F_GETFL);
fcntl(sd, F_SETFL, options);
return 1;
}

/* Currently only sets SO_LINGER, I haven't seen any evidence that this
   helps.  I'll do more testing before dumping it. */
__inline__ void init_socket(int sd) {
struct linger l;

l.l_onoff = 1;
l.l_linger = 0;

if (setsockopt(sd, SOL_SOCKET, SO_LINGER,  &l, sizeof(struct linger)))
  {
   fprintf(stderr, "Problem setting socket SO_LINGER, errno: %d\n", errno);
   perror("setsockopt");
  }
}

/* Convert a string like "-100,200-1024,3000-4000,60000-" into an array
   of port numbers*/
unsigned short *getpts(char *origexpr) {
int exlen = strlen(origexpr);
char *p,*q;
unsigned short *tmp, *ports;
int i=0, j=0,start,end;
char *expr = strdup(origexpr);
ports = safe_malloc(65536 * sizeof(short));
i++;
i--;
for(;j < exlen; j++)
```

```
    if (expr[j] != ' ') expr[i++] = expr[j];
  expr[i] = '\0';
  exlen = i + 1;
  i=0;
  while((p = strchr(expr,','))) {
    *p = '\0';
    if (*expr == '-') {start = 1; end = atoi(expr+ 1);}
    else {
      start = end = atoi(expr);
      if ((q = strchr(expr,'-')) && *(q+1) ) end = atoi(q + 1);
      else if (q && !*(q+1)) end = 65535;
    }
    if (debugging)
      printf("The first port is %d, and the last one is %d\n", start, end);
    if (start < 1 || start > end) fatal("Your port specifications are illegal!");
    for(j=start; j <= end; j++)
      ports[i++] = j;
    expr = p + 1;
  }
  if (*expr == '-') {
    start = 1;
    end = atoi(expr+ 1);
  }
  else {
    start = end = atoi(expr);
    if ((q =  strchr(expr,'-')) && *(q+1) ) end = atoi(q+1);
    else if (q && !*(q+1)) end = 65535;
  }
  if (debugging)
    printf("The first port is %d, and the last one is %d\n", start, end);
  if (start < 1 || start > end) fatal("Your port specifications are illegal!");
  for(j=start; j <= end; j++)
    ports[i++] = j;
  number_of_ports = i;
  ports[i++] = 0;
  tmp = realloc(ports, i * sizeof(short));
    free(expr);
    return tmp;
  }

  unsigned short *getfastports(int tcpscan, int udpscan) {
    int portindex = 0, res, lastport = 0;
    unsigned int portno = 0;
    unsigned short *ports;
    char proto[10];
    char line[81];
    FILE *fp;
    ports = safe_malloc(65535 * sizeof(unsigned short));
```

```
    proto[0] = '\0';
    if (!(fp = fopen("/etc/services", "r"))) {
      printf("We can't open /etc/services for reading!  Fix your system or don't use -f\n");
      perror("fopen");
      exit(1);
    }

    while(fgets(line, 80, fp)) {
      res = sscanf(line, "%*s %u/%s", &portno, proto);
      if (res == 2 && portno != 0 && portno != lastport) {
        lastport = portno;
        if (tcpscan && proto[0] == 't')
          ports[portindex++] = portno;
        else if (udpscan && proto[0] == 'u')
          ports[portindex++] = portno;
      }
    }


  number_of_ports = portindex;
  ports[portindex++] = 0;
  return realloc(ports, portindex * sizeof(unsigned short));
  }

  void printusage(char *name) {
  printf("%s [options] [hostname[/mask] . . .]
  options (none are required, most can be combined):
     -t tcp connect() port scan
     -s tcp SYN stealth port scan (must be root)
     -u UDP port scan, will use MUCH better version if you are root
     -U Uriel Maimon (P49-15) style FIN stealth scan.
     -l Do the lamer UDP scan even if root.  Less accurate.
     -P ping \"scan\". Find which hosts on specified network(s) are up.
     -b <ftp_relay_host> ftp \"bounce attack\" port scan
     -f use tiny fragmented packets for SYN or FIN scan.
     -i Get identd (rfc 1413) info on listening TCP processes.
     -p <range> ports: ex: \'-p 23\' will only try port 23 of the host(s)
                    \'-p 20-30,63000-\' scans 20-30 and 63000-65535 default: 1-1024
     -F fast scan. Only scans ports in /etc/services, a la strobe(1).
     -r randomize target port scanning order.
     -h help, print this junk.  Also see http://www.dhp.com/~fyodor/nmap/
     -S If you want to specify the source address of SYN or FYN scan.
     -v Verbose.  Its use is recommended.  Use twice for greater effect.
     -w <n> delay.  n microsecond delay. Not recommended unless needed.
     -M <n> maximum number of parallel sockets.  Larger isn't always better.
     -q quash argv to something benign, currently set to \"%s\".
  Hostnames specified as internet hostname or IP address.  Optional '/mask' specifies subnet. cert.org/24 or
  192.88.209.5/24 scan CERT's Class C.\n",
```

```
        name, FAKE_ARGV);
  exit(1);
  }

  portlist tcp_scan(struct in_addr target, unsigned short *portarray, portlist *ports) {

  int starttime, current_out = 0, res , deadindex = 0, i=0, j=0, k=0, max=0;
  struct sockaddr_in sock, stranger, mysock;
  int sockaddr_in_len = sizeof(struct sockaddr_in);
  int sockets[max_parallel_sockets], deadstack[max_parallel_sockets];
  unsigned short portno[max_parallel_sockets];
  char owner[513], buf[65536];
  int tryident = identscan, current_socket /*actually it is a socket INDEX*/;
  fd_set fds_read, fds_write;
  struct timeval nowait = {0,0},  longwait = {7,0};

  signal(SIGPIPE, SIG_IGN); /* ignore SIGPIPE so our 'write 0 bytes' test
                               doesn't crash our program!*/
  owner[0] = '\0';
  starttime = time(NULL);
  bzero((char *)&sock,sizeof(struct sockaddr_in));
  sock.sin_addr.s_addr = target.s_addr;
  if (verbose || debugging)
    printf("Initiating TCP connect() scan against %s (%s)\n",
           current_name,  inet_ntoa(sock.sin_addr));
  sock.sin_family=AF_INET;
  FD_ZERO(&fds_read);
  FD_ZERO(&fds_write);

  if (tryident)
    tryident = check_ident_port(target);

  /* Initially, all of our sockets are "dead" */
  for(i = 0 ; i < max_parallel_sockets; i++) {
    deadstack[deadindex++] = i;
    portno[i] = 0;
  }

  deadindex--;
  /* deadindex always points to the most recently added dead socket index */

  while(portarray[j]) {
    longwait.tv_sec = 7;
    longwait.tv_usec = nowait.tv_sec = nowait.tv_usec = 0;

    for(i=current_out; i < max_parallel_sockets && portarray[j]; i++, j++) {
      current_socket = deadstack[deadindex--];
      if ((sockets[current_socket] = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
```

```
      {perror("Socket troubles"); exit(1);}
    if (sockets[current_socket] > max) max = sockets[current_socket];
    current_out++;
    unblock_socket(sockets[current_socket]);
    init_socket(sockets[current_socket]);
    portno[current_socket] = portarray[j];
    sock.sin_port = htons(portarray[j]);
    if ((res = connect(sockets[current_socket],(struct sockaddr *)&sock,sizeof(struct sockaddr)))!=-1)
      printf("WTF???? I think we got a successful connection in non-blocking!!@#$\n");
    else {
      switch(errno) {
      case EINPROGRESS: /* The one I always see */
      case EAGAIN:
        block_socket(sockets[current_socket]);
        FD_SET(sockets[current_socket], &fds_write);
        FD_SET(sockets[current_socket], &fds_read);
        break;
      default:
        printf("Strange error from connect: (%d)", errno);
        perror(""); /*falling through intentionally*/
      case ECONNREFUSED:
        if (max == sockets[current_socket]) max--;
        deadstack[++deadindex] = current_socket;
        current_out--;
        portno[current_socket] = 0;
        close(sockets[current_socket]);
        break;
      }
    }
  }
  if (!portarray[j]) sleep(1); /*wait a second for any last packets*/
  while((res = select(max + 1, &fds_read, &fds_write, NULL,
                      (current_out < max_parallel_sockets)?
                      &nowait : &longwait)) > 0) {
    for(k=0; k < max_parallel_sockets; k++)
      if (portno[k]) {
        if (FD_ISSET(sockets[k], &fds_write)
            && FD_ISSET(sockets[k], &fds_read)) {
          /*printf("Socket at port %hi is selectable for r & w.", portno[k]);*/
          res = recvfrom(sockets[k], buf, 65536, 0, (struct sockaddr *)
                         & stranger, &sockaddr_in_len);
          if (res >= 0) {
            if (debugging || verbose)
              printf("Adding TCP port %hi due to successful read.\n",
                     portno[k]);
            if (tryident) {
              if ( getsockname(sockets[k], (struct sockaddr *) &mysock,
                               &sockaddr_in_len ) ) {
```

```
        perror("getsockname");
        exit(1);
      }
      tryident = getidentinfoz(target, ntohs(mysock.sin_port),
                               portno[k], owner);
    }
    addport(ports, portno[k], IPPROTO_TCP, owner);
  }
  if (max == sockets[k])
    max--;
  FD_CLR(sockets[k], &fds_read);
  FD_CLR(sockets[k], &fds_write);
  deadstack[++deadindex] = k;
  current_out--;
  portno[k] = 0;
  close(sockets[k]);
}
else if(FD_ISSET(sockets[k], &fds_write)) {
  /*printf("Socket at port %hi is selectable for w only.VERIFYING\n",
    portno[k]);*/
  res = send(sockets[k], buf, 0, 0);
  if (res < 0 ) {
    signal(SIGPIPE, SIG_IGN);
    if (debugging > 1)
      printf("Bad port %hi caught by 0-byte write!\n", portno[k]);
  }
  else {
    if (debugging || verbose)
      printf("Adding TCP port %hi due to successful 0-byte write!\n",
             portno[k]);
    if (tryident) {
      if ( getsockname(sockets[k], (struct sockaddr *) &mysock ,
                       &sockaddr_in_len ) ) {
        perror("getsockname");
        exit(1);
      }
      tryident = getidentinfoz(target, ntohs(mysock.sin_port),
                               portno[k], owner);
    }
    addport(ports, portno[k], IPPROTO_TCP, owner);
  }
  if (max == sockets[k]) max--;
  FD_CLR(sockets[k], &fds_write);
  deadstack[++deadindex] = k;
  current_out--;
  portno[k] = 0;
  close(sockets[k]);
}
```

```
          else if ( FD_ISSET(sockets[k], &fds_read) ) {
            printf("Socket at port %hi is selectable for r only.  This is very wierd.\n", portno[k]);
            if (max == sockets[k]) max--;
            FD_CLR(sockets[k], &fds_read);
            deadstack[++deadindex] = k;
            current_out--;
            portno[k] = 0;
            close(sockets[k]);
          }
          else {
            /*printf("Socket at port %hi not selecting, readding.\n",portno[k]);*/
            FD_SET(sockets[k], &fds_write);
            FD_SET(sockets[k], &fds_read);
          }
        }
    }
  }

  if (debugging || verbose)
    printf("Scanned %d ports in %ld seconds with %d parallel sockets.\n",
           number_of_ports, time(NULL) - starttime, max_parallel_sockets);
  return *ports;
  }

  /* gawd, my next project will be in c++ so I don't have to deal with
     this crap ... simple linked list implementation */
  int addport(portlist *ports, unsigned short portno, unsigned short protocol,
              char *owner) {
  struct port *current, *tmp;
  int len;

  if (*ports) {
    current = *ports;
    /* case 1: we add to the front of the list */
    if (portno <= current->portno) {
      if (current->portno == portno && current->proto == protocol) {
        if (debugging || verbose)
          printf("Duplicate port (%hi/%s)\n", portno ,
                 (protocol == IPPROTO_TCP)? "tcp": "udp");
        return -1;
      }
      tmp = current;
      *ports = safe_malloc(sizeof(struct port));
      (*ports)->next = tmp;
      current = *ports;
      current->portno = portno;
      current->proto = protocol;
      if (owner && *owner) {
```

```
        len = strlen(owner);
        current->owner = malloc(sizeof(char) * (len + 1));
        strncpy(current->owner, owner, len + 1);
      }
      else current->owner = NULL;
    }
    else { /* case 2: we add somewhere in the middle or end of the list */
      while( current->next  && current->next->portno < portno)
        current = current->next;
      if (current->next && current->next->portno == portno
          && current->next->proto == protocol) {
        if (debugging || verbose)
          printf("Duplicate port (%hi/%s)\n", portno ,
                 (protocol == IPPROTO_TCP)? "tcp": "udp");
        return -1;
      }
      tmp = current->next;
      current->next = safe_malloc(sizeof(struct port));
      current->next->next = tmp;
      tmp = current->next;
      tmp->portno = portno;
      tmp->proto = protocol;
      if (owner && *owner) {
        len = strlen(owner);
        tmp->owner = malloc(sizeof(char) * (len + 1));
        strncpy(tmp->owner, owner, len + 1);
      }
      else tmp->owner = NULL;
    }
  }

  else { /* Case 3, list is null */
    *ports = safe_malloc(sizeof(struct port));
    tmp = *ports;
    tmp->portno = portno;
    tmp->proto = protocol;
    if (owner && *owner) {
      len = strlen(owner);
      tmp->owner = safe_malloc(sizeof(char) * (len + 1));
      strncpy(tmp->owner, owner, len + 1);
    }
    else tmp->owner = NULL;
    tmp->next = NULL;
  }
  return 0; /*success */
  }

  int deleteport(portlist *ports, unsigned short portno,
```

```
                unsigned short protocol) {
  portlist current, tmp;

  if (!*ports) {
    if (debugging > 1) error("Tried to delete from empty port list!");
    return -1;
  }
  /* Case 1, deletion from front of list*/
  if ((*ports)->portno == portno && (*ports)->proto == protocol) {
    tmp = (*ports)->next;
    if ((*ports)->owner) free((*ports)->owner);
    free(*ports);
    *ports = tmp;
  }
  else {
    current = *ports;
    for(;current->next && (current->next->portno != portno || current->next->proto != protocol); current =
current->next);
    if (!current->next)
      return -1;
    tmp = current->next;
    current->next = tmp->next;
    if (tmp->owner) free(tmp->owner);
    free(tmp);
}
  return 0; /* success */
}


void *safe_malloc(int size)
{
  void *mymem;
  if (size < 0)
    fatal("Tried to malloc negative amount of memmory!!!");
  if ((mymem = malloc(size)) == NULL)
    fatal("Malloc Failed! Probably out of space.");
  return mymem;
}

void printandfreeports(portlist ports) {
  char protocol[4];
  struct servent *service;
  port *current = ports, *tmp;

  printf("Port Number  Protocol  Service");
  printf("%s", (identscan)?"         Owner\n":"\n");
  while(current != NULL) {
    strcpy(protocol,(current->proto == IPPROTO_TCP)? "tcp": "udp");
```

```
      service = getservbyport(htons(current->portno), protocol);
      printf("%-13d%-11s%-16s%s\n", current->portno, protocol,
             (service)? service->s_name: "unknown",
             (current->owner)? current->owner : "");
      tmp = current;
      current = current->next;
      if (tmp->owner) free(tmp->owner);
      free(tmp);
    }
    printf("\n");
  }

  /* This is the version of udp_scan that uses raw ICMP sockets and requires
     root priviliges.*/
  portlist udp_scan(struct in_addr target, unsigned short *portarray,
                    portlist *ports) {
    int icmpsock, udpsock, tmp, done=0, retries, bytes = 0, res,  num_out = 0;
    int i=0,j=0, k=0, icmperrlimittime, max_tries = UDP_MAX_PORT_RETRIES;
    unsigned short outports[max_parallel_sockets], numtries[max_parallel_sockets];
    struct sockaddr_in her;
    char senddata[] = "blah\n";
    unsigned long starttime, sleeptime;
    struct timeval shortwait = {1, 0 };
    fd_set  fds_read, fds_write;

    bzero(outports, max_parallel_sockets * sizeof(unsigned short));
    bzero(numtries, max_parallel_sockets * sizeof(unsigned short));

     /* Some systems (like linux) follow the advice of rfc1812 and limit
      * the rate at which they will respons with icmp error messages
      * (like port unreachable).  icmperrlimittime is to compensate for that.
      */
    icmperrlimittime = 60000;

    sleeptime = (global_delay)? global_delay : (global_rtt)? (1.2 * global_rtt) + 30000 : 1e5;
  if (global_delay) icmperrlimittime = global_delay;

  starttime = time(NULL);

  FD_ZERO(&fds_read);
  FD_ZERO(&fds_write);

  if (verbose || debugging)
    printf("Initiating UDP (raw ICMP version) scan against %s (%s) using wait delay of %li usecs.\n",
  current_name,  inet_ntoa(target), sleeptime);

  if ((icmpsock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0)
    perror("Opening ICMP RAW socket");
```

```
  if ((udpsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    perror("Opening datagram socket");

  unblock_socket(icmpsock);
  her.sin_addr = target;
  her.sin_family = AF_INET;

  while(!done) {
    tmp = num_out;
    for(i=0; (i < max_parallel_sockets && portarray[j]) || i < tmp; i++) {
      close(udpsock);
      if ((udpsock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        perror("Opening datagram socket");
      if ((i > tmp && portarray[j]) || numtries[i] > 1) {
        if (i > tmp) her.sin_port = htons(portarray[j++]);
        else her.sin_port = htons(outports[i]);
        FD_SET(udpsock, &fds_write);
        FD_SET(icmpsock, &fds_read);
        shortwait.tv_sec = 1; shortwait.tv_usec = 0;
        usleep(icmperrlimittime);
        res = select(udpsock + 1, NULL, &fds_write, NULL, &shortwait);
         if (FD_ISSET(udpsock, &fds_write))
            bytes = sendto(udpsock, senddata, sizeof(senddata), 0,
                           (struct sockaddr *) &her, sizeof(struct sockaddr_in));
        else {
          printf("udpsock not set for writing port %d!",  ntohs(her.sin_port));
          return *ports;
        }
        if (bytes <= 0) {
          if (errno == ECONNREFUSED) {
            retries = 10;
            do {
              /* This is from when I was using the same socket and would
               * (rather often) get strange connection refused errors, it
               * shouldn't happen now that I create a new udp socket for each
               * port.  At some point I will probably go back to 1 socket again.
               */
              printf("sendto said connection refused on port %d but trying again anyway.\n",
  ntohs(her.sin_port));
              usleep(icmperrlimittime);
              bytes = sendto(udpsock, senddata, sizeof(senddata), 0,
                             (struct sockaddr *) &her, sizeof(struct sockaddr_in));
              printf("This time it returned %d\n", bytes);
            } while(bytes <= 0 && retries-- > 0);
          }
          if (bytes <= 0) {
            printf("sendto returned %d.", bytes);
            fflush(stdout);
```

```
            perror("sendto");
          }
        }
        if (bytes > 0 && i > tmp) {
          num_out++;
          outports[i] = portarray[j-1];
        }
      }
    }
    usleep(sleeptime);
    tmp = listen_icmp(icmpsock, outports, numtries, &num_out, target, ports);
    if (debugging) printf("listen_icmp caught %d bad ports.\n", tmp);
    done = !portarray[j];
    for (i=0,k=0; i < max_parallel_sockets; i++)
      if (outports[i]) {
        if (++numtries[i] > max_tries - 1) {
          if (debugging || verbose)
            printf("Adding port %d for 0 unreachable port generations\n",
                   outports[i]);
          addport(ports, outports[i], IPPROTO_UDP, NULL);
          num_out--;
          outports[i] = numtries[i] = 0;
        }
        else {
          done = 0;
          outports[k] = outports[i];
          numtries[k] = numtries[i];
          if (k != i)
            outports[i] = numtries[i] = 0;
          k++;
        }
      }
    if (num_out == max_parallel_sockets) {
    printf("Numout is max sockets, that is a problem!\n");
    sleep(1); /* Give some time for responses to trickle back,
                 and possibly to reset the hosts ICMP error limit */
    }
  }


  if (debugging || verbose)
    printf("The UDP raw ICMP scanned %d ports in  %ld seconds with %d parallel sockets.\n", number_of_ports,
  time(NULL) - starttime, max_parallel_sockets);
  close(icmpsock);
  close(udpsock);
  return *ports;
  }
```

```
int listen_icmp(int icmpsock,  unsigned short outports[],
                unsigned short numtries[], int *num_out, struct in_addr target,
                portlist *ports) {
  char response[1024];
  struct sockaddr_in stranger;
  int sockaddr_in_size = sizeof(struct sockaddr_in);
  struct in_addr bs;
  struct iphdr *ip = (struct iphdr *) response;
  struct icmphdr *icmp = (struct icmphdr *) (response + sizeof(struct iphdr));
  struct iphdr *ip2;
  unsigned short *data;
  int badport, numcaught=0, bytes, i, tmptry=0, found=0;

  while  ((bytes = recvfrom(icmpsock, response, 1024, 0,
                            (struct sockaddr *) &stranger,
                            &sockaddr_in_size)) > 0) {
  numcaught++;
  bs.s_addr = ip->saddr;
  if (ip->saddr == target.s_addr && ip->protocol == IPPROTO_ICMP
      && icmp->type == 3 && icmp->code == 3) {
    ip2 = (struct iphdr *) (response + 4 * ip->ihl + sizeof(struct icmphdr));
    data = (unsigned short *) ((char *)ip2 + 4 * ip2->ihl);
    badport = ntohs(data[1]);
    /*delete it from our outports array */
    found = 0;
    for(i=0; i < max_parallel_sockets; i++)
      if (outports[i] == badport) {
        found = 1;
        tmptry = numtries[i];
        outports[i] = numtries[i] = 0;
        (*num_out)--;
        break;
      }
    if (debugging && found && tmptry > 0)
      printf("Badport: %d on try number %d\n", badport, tmptry);
    if (!found) {
      if (debugging)
        printf("Badport %d came in late, deleting from portlist.\n", badport);
      if (deleteport(ports, badport, IPPROTO_UDP) < 0)
        if (debugging) printf("Port deletion failed.\n");
    }
  }
  else {
    printf("Funked up packet!\n");
  }
}
  return numcaught;
}
```

```
/* This fucntion is nonsens.  I wrote it all, really optimized etc.  Then
   found out that many hosts limit the rate at which they send icmp errors :(
   I will probably totally rewrite it to be much simpler at some point.  For
   now I won't worry about it since it isn't a very important functions (UDP
   is lame, plus there is already a much better function for people who
   are r00t */
portlist lamer_udp_scan(struct in_addr target, unsigned short *portarray,
                        portlist *ports) {
int sockaddr_in_size = sizeof(struct sockaddr_in),i=0,j=0,k=0, bytes;
int sockets[max_parallel_sockets], trynum[max_parallel_sockets];
unsigned short portno[max_parallel_sockets];
int last_open = 0;
char response[1024];
struct sockaddr_in her, stranger;
char data[] = "\nhelp\nquit\n";
unsigned long sleeptime;
unsigned int starttime;

/* Initialize our target sockaddr_in */
bzero((char *) &her, sizeof(struct sockaddr_in));
her.sin_family = AF_INET;
her.sin_addr = target;

if (global_delay) sleeptime = global_delay;
else sleeptime =  calculate_sleep(target) + 60000; /*large to be on the
                                                     safe side */

if (verbose || debugging)
  printf("Initiating UDP scan against %s (%s), sleeptime: %li\n", current_name,
         inet_ntoa(target), sleeptime);

starttime = time(NULL);

for(i = 0 ; i < max_parallel_sockets; i++)
  trynum[i] =  portno[i] = 0;

while(portarray[j]) {
  for(i=0; i < max_parallel_sockets && portarray[j]; i++, j++) {
    if (i >= last_open) {
      if ((sockets[i] = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
        {perror("datagram socket troubles"); exit(1);}
      block_socket(sockets[i]);
      portno[i] = portarray[j];
    }
    her.sin_port = htons(portarray[j]);
    bytes = sendto(sockets[i], data, sizeof(data), 0, (struct sockaddr *) &her,
                   sizeof(struct sockaddr_in));
```

```
      usleep(5000);
      if (debugging > 1)
        printf("Sent %d bytes on socket %d to port %hi, try number %d.\n",
               bytes, sockets[i], portno[i], trynum[i]);
      if (bytes < 0 ) {
        printf("Sendto returned %d the FIRST TIME!@#$!, errno %d\n", bytes,
               errno);
        perror("");
        trynum[i] = portno[i] = 0;
        close(sockets[i]);
      }
    }
    last_open = i;
    /* Might need to change this to 1e6 if you are having problems*/
    usleep(sleeptime + 5e5);
    for(i=0; i < last_open ; i++) {
      if (portno[i]) {
        unblock_socket(sockets[i]);
        if ((bytes = recvfrom(sockets[i], response, 1024, 0,
                              (struct sockaddr *) &stranger,
                              &sockaddr_in_size)) == -1)
          {
            if (debugging > 1)
              printf("2nd recvfrom on port %d returned %d with errno %d.\n",
                     portno[i], bytes, errno);
            if (errno == EAGAIN /*11*/)
              {
                if (trynum[i] < 2) trynum[i]++;
                else {
                  if (RISKY_UDP_SCAN) {
                    printf("Adding port %d after 3 EAGAIN errors.\n", portno[i]);
                    addport(ports, portno[i], IPPROTO_UDP, NULL);
                  }
                  else if (debugging)
                    printf("Skipping possible false positive, port %d\n",
                           portno[i]);
                  trynum[i] = portno[i] = 0;
                  close(sockets[i]);
                }
              }
            else if (errno == ECONNREFUSED /*111*/) {
              if (debugging > 1)
                printf("Closing socket for port %d, ECONNREFUSED received.\n",
                       portno[i]);
              trynum[i] = portno[i] = 0;
              close(sockets[i]);
            }
            else {
```

```
              printf("Curious recvfrom error (%d) on port %hi: ",
                     errno, portno[i]);
              perror("");
              trynum[i] = portno[i] = 0;
              close(sockets[i]);
            }
          }
        else /*bytes is positive*/ {
          if (debugging || verbose)
            printf("Adding UDP port %d due to positive read!\n", portno[i]);
          addport(ports,portno[i], IPPROTO_UDP, NULL);
          trynum[i] = portno[i] = 0;
          close(sockets[i]);
        }
      }
    }
    /* Update last_open, we need to create new sockets.*/
    for(i=0, k=0; i < last_open; i++)
      if (portno[i]) {
        close(sockets[i]);
        sockets[k] = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
        /*      unblock_socket(sockets[k]);*/
        portno[k] = portno[i];
        trynum[k] = trynum[i];
        k++;
      }
   last_open = k;
    for(i=k; i < max_parallel_sockets; i++)
      trynum[i] = sockets[i] = portno[i] = 0;
  }
  if (debugging)
    printf("UDP scanned %d ports in %ld seconds with %d parallel sockets\n",
           number_of_ports, time(NULL) - starttime, max_parallel_sockets);
  return *ports;
  }

  /* This attempts to calculate the round trip time (rtt) to a host by timing a
     connect() to a port which isn't listening.  A better approach is to time a
     ping (since it is more likely to get through firewalls.  This is now
     implemented in isup() for users who are root.  */
  unsigned long calculate_sleep(struct in_addr target) {
  struct timeval begin, end;
  int sd;
  struct sockaddr_in sock;
  int res;

  if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
    {perror("Socket troubles"); exit(1);}
```

```
sock.sin_family = AF_INET;
sock.sin_addr.s_addr = target.s_addr;
sock.sin_port = htons(MAGIC_PORT);

gettimeofday(&begin, NULL);
if ((res = connect(sd, (struct sockaddr *) &sock,
                   sizeof(struct sockaddr_in))) != -1)
  printf("You might want to change MAGIC_PORT in the include file, it seems to be listening on the target
host!\n");
close(sd);
gettimeofday(&end, NULL);
if (end.tv_sec - begin.tv_sec > 5 ) /*uh-oh!*/
  return 0;
return (end.tv_sec - begin.tv_sec) * 1000000 + (end.tv_usec - begin.tv_usec);
}

/* Checks whether the identd port (113) is open on the target machine.  No
   sense wasting time trying it for each good port if it is down! */
int check_ident_port(struct in_addr target) {
int sd;
struct sockaddr_in sock;
int res;

if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
  {perror("Socket troubles"); exit(1);}

sock.sin_family = AF_INET;
sock.sin_addr.s_addr = target.s_addr;
sock.sin_port = htons(113); /*should use getservbyname(3), yeah, yeah */
res = connect(sd, (struct sockaddr *) &sock, sizeof(struct sockaddr_in));
close(sd);
if (res < 0 ) {
  if (debugging || verbose) printf("identd port not active\n");
  return 0;
}
if (debugging || verbose) printf("identd port is active\n");
return 1;
}

int getidentinfoz(struct in_addr target, int localport, int remoteport,
                  char *owner) {
int sd;
struct sockaddr_in sock;
int res;
char request[15];
char response[1024];
char *p,*q;
```

```
  char  *os;

  owner[0] = '\0';
  if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)
    {perror("Socket troubles"); exit(1);}

  sock.sin_family = AF_INET;
  sock.sin_addr.s_addr = target.s_addr;
  sock.sin_port = htons(113);
  usleep(50000);   /* If we aren't careful, we really MIGHT take out inetd,
                      some are very fragile */
  res = connect(sd, (struct sockaddr *) &sock, sizeof(struct sockaddr_in));

  if (res < 0 ) {
    if (debugging || verbose)
      printf("identd port not active now for some reason ... hope we didn't break it!\n");
    close(sd);
    return 0;
  }
  sprintf(request,"%hi,%hi\r\n", remoteport, localport);
  if (debugging > 1) printf("Connected to identd, sending request: %s", request);
  if (write(sd, request, strlen(request) + 1) == -1) {
    perror("identd write");
    close(sd);
    return 0;
  }
  else if ((res = read(sd, response, 1024)) == -1) {
    perror("reading from identd");
    close(sd);
    return 0;
  }
  else {
    close(sd);
    if (debugging > 1) printf("Read %d bytes from identd: %s\n", res, response);
    if ((p = strchr(response, ':'))) {
      p++;
      if ((q = strtok(p, " :"))) {
        if (!strcasecmp( q, "error")) {
          if (debugging || verbose) printf("ERROR returned from identd\n");
          return 0;
        }
        if ((os = strtok(NULL, " :"))) {
          if ((p = strtok(NULL, " :"))) {
            if ((q = strchr(p, '\r'))) *q = '\0';
            if ((q = strchr(p, '\n'))) *q = '\0';
            strncpy(owner, p, 512);
            owner[512] = '\0';
          }
```

```
      }
    }
  }
}
return 1;
}

/* A relatively fast (or at least short ;) ping function.  Doesn't require a
   seperate checksum function */
int isup(struct in_addr target) {
  int res, retries = 3;
  struct sockaddr_in sock;
  /*type(8bit)=8, code(8)=0 (echo REQUEST), checksum(16)=34190, id(16)=31337 */
#ifdef __LITTLE_ENDIAN_BITFIELD
  unsigned char ping[64] = { 0x8, 0x0, 0x8e, 0x85, 0x69, 0x7A };
#else
  unsigned char ping[64] = { 0x8, 0x0, 0x85, 0x8e, 0x7A, 0x69 };
#endif
  int sd;
  struct timeval tv;
  struct timeval start, end;
 fd_set fd_read;
  struct {
    struct iphdr ip;
    unsigned char type;
    unsigned char code;
    unsigned short checksum;
    unsigned short identifier;
    char crap[16536];
  }  response;

sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);

bzero((char *)&sock,sizeof(struct sockaddr_in));
sock.sin_family=AF_INET;
sock.sin_addr = target;
if (debugging > 1) printf(" Sending 3 64 byte raw pings to host.\n");
gettimeofday(&start, NULL);
while(--retries) {
  if ((res = sendto(sd,(char *) ping,64,0,(struct sockaddr *)&sock,
                    sizeof(struct sockaddr))) != 64) {
    fprintf(stderr, "sendto in isup returned %d! skipping host.\n", res);
    return 0;
  }
  FD_ZERO(&fd_read);
  FD_SET(sd, &fd_read);
  tv.tv_sec = 0;
  tv.tv_usec = 1e6 * (PING_TIMEOUT / 3.0);
```

```
    while(1) {
      if ((res = select(sd + 1, &fd_read, NULL, NULL, &tv)) != 1)
        break;
      else {
        read(sd,&response,sizeof(response));
        if  (response.ip.saddr == target.s_addr &&  !response.type
             && !response.code   && response.identifier == 31337) {
          gettimeofday(&end, NULL);
          global_rtt = (end.tv_sec - start.tv_sec) * 1e6 + end.tv_usec - start.tv_usec;
          ouraddr.s_addr = response.ip.daddr;
          close(sd);
          return 1;
        }
      }
    }
  }
  close(sd);
  return 0;
  }


  portlist syn_scan(struct in_addr target, unsigned short *portarray,
                    struct in_addr *source, int fragment, portlist *ports) {
  int i=0, j=0, received, bytes, starttime;
  struct sockaddr_in from;
  int fromsize = sizeof(struct sockaddr_in);
  int sockets[max_parallel_sockets];
  struct timeval tv;
  char packet[65535];
  struct iphdr *ip = (struct iphdr *) packet;
  struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct iphdr));
  fd_set fd_read, fd_write;
  int res;
  struct hostent *myhostent;
  char myname[MAXHOSTNAMELEN + 1];
  int source_malloc = 0;

  FD_ZERO(&fd_read);
  FD_ZERO(&fd_write);

  tv.tv_sec = 7;
  tv.tv_usec = 0;

  if ((received = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0 )
    perror("socket trobles in syn_scan");
  unblock_socket(received);
  FD_SET(received, &fd_read);
```

```
/* First we take what is given to us as source.  If that isn't valid, we take
   what should have swiped from the echo reply in our ping function.  If THAT
   doesn't work either, we try to determine our address with gethostname and
   gethostbyname.  Whew! */
if (!source) {
  if (ouraddr.s_addr) {
    source = &ouraddr;
  }
  else {
  source = safe_malloc(sizeof(struct in_addr));
  source_malloc = 1;
  if (gethostname(myname, MAXHOSTNAMELEN) ||
      !(myhostent = gethostbyname(myname)))
    fatal("Your system is fucked up.\n");
  memcpy(source, myhostent->h_addr_list[0], sizeof(struct in_addr));
  }
  if (debugging)
    printf("We skillfully deduced that your address is %s\n",
           inet_ntoa(*source));
}

starttime = time(NULL);

do {
  for(i=0; i < max_parallel_sockets && portarray[j]; i++) {
    if ((sockets[i] = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0 )
      perror("socket trobles in syn_scan");
    else {
      if (fragment)
        send_small_fragz(sockets[i], source, &target, MAGIC_PORT,
                         portarray[j++], TH_SYN);
      else send_tcp_raw(sockets[i], source , &target, MAGIC_PORT,
                        portarray[j++],0,0,TH_SYN,0,0,0);
      usleep(10000);
    }
  }
  if ((res = select(received + 1, &fd_read, NULL, NULL, &tv)) < 0)
    perror("select problems in syn_scan");
  else if (res > 0) {
    while  ((bytes = recvfrom(received, packet, 65535, 0,
                              (struct sockaddr *)&from, &fromsize)) > 0 ) {
      if (ip->saddr == target.s_addr) {
        if (tcp->th_flags & TH_RST) {
          if (debugging > 1) printf("Nothing open on port %d\n",
                                    ntohs(tcp->th_sport));
        }
        else /*if (tcp->th_flags & TH_SYN && tcp->th_flags & TH_ACK)*/ {
          if (debugging || verbose) {
```

```
              printf("Possible catch on port %d!  Here it is:\n",
                     ntohs(tcp->th_sport));
              readtcppacket(packet,1);
            }
            addport(ports, ntohs(tcp->th_sport), IPPROTO_TCP, NULL);
          }
        }
      }
    }
    for(i=0; i < max_parallel_sockets && portarray[j]; i++) close(sockets[i]);

  } while (portarray[j]);
  if (debugging || verbose)
    printf("The TCP SYN scan took %ld seconds to scan %d ports.\n",
           time(NULL) - starttime, number_of_ports);
  if (source_malloc) free(source);  /* Gotta save those 4 bytes! ;) */
  close(received);
  return *ports;
  }


  int send_tcp_raw( int sd, struct in_addr *source,
                    struct in_addr *victim, unsigned short sport,
                    unsigned short dport, unsigned long seq,
                    unsigned long ack, unsigned char flags,
                    unsigned short window, char *data,
                    unsigned short datalen)
  {

  struct pseudo_header {
    /*for computing TCP checksum, see TCP/IP Illustrated p. 145 */
    unsigned long s_addr;
    unsigned long d_addr;
    char zer0;
    unsigned char protocol;
    unsigned short length;
  };
  char packet[sizeof(struct iphdr) + sizeof(struct tcphdr) + datalen];
   /*With these placement we get data and some field alignment so we aren't
     wasting too much in computing the checksum */
  struct iphdr *ip = (struct iphdr *) packet;
  struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct iphdr));
  struct pseudo_header *pseudo =  (struct pseudo_header *) (packet + sizeof(struct iphdr) - sizeof(struct
  pseudo_header));
  int res;
  struct sockaddr_in sock;
  char myname[MAXHOSTNAMELEN + 1];
  struct hostent *myhostent;
```

```
int source_malloced = 0;

/* check that required fields are there and not too silly */
if ( !victim || !sport || !dport || sd < 0) {
  fprintf(stderr, "send_tcp_raw: One or more of your parameters suck!\n");
  return -1;
}

/* if they didn't give a source address, fill in our first address */
if (!source) {
  source_malloced = 1;
  source = safe_malloc(sizeof(struct in_addr));
  if (gethostname(myname, MAXHOSTNAMELEN) ||
      !(myhostent = gethostbyname(myname)))
    fatal("Your system is fucked up.\n");
  memcpy(source, myhostent->h_addr_list[0], sizeof(struct in_addr));
  if (debugging > 1)
    printf("We skillfully deduced that your address is %s\n",
           inet_ntoa(*source));
}


/*do we even have to fill out this damn thing?  This is a raw packet,
  after all */
sock.sin_family = AF_INET;
sock.sin_port = htons(dport);
sock.sin_addr.s_addr = victim->s_addr;

bzero(packet, sizeof(struct iphdr) + sizeof(struct tcphdr));

pseudo->s_addr = source->s_addr;
pseudo->d_addr = victim->s_addr;
pseudo->protocol = IPPROTO_TCP;
pseudo->length = htons(sizeof(struct tcphdr) + datalen);

tcp->th_sport = htons(sport);
tcp->th_dport = htons(dport);
if (seq)
  tcp->th_seq = htonl(seq);
else tcp->th_seq = rand() + rand();

if (flags & TH_ACK && ack)
  tcp->th_ack = htonl(seq);
else if (flags & TH_ACK)
  tcp->th_ack = rand() + rand();

tcp->th_off = 5 /*words*/;
tcp->th_flags = flags;
```

```
if (window)
  tcp->th_win = window;
else tcp->th_win = htons(2048); /* Who cares */

tcp->th_sum = in_cksum((unsigned short *)pseudo, sizeof(struct tcphdr) +
                       sizeof(struct pseudo_header) + datalen);

/* Now for the ip header */
bzero(packet, sizeof(struct iphdr));
ip->version = 4;
ip->ihl = 5;
ip->tot_len = htons(sizeof(struct iphdr) + sizeof(struct tcphdr) + datalen);
ip->id = rand();
ip->ttl = 255;
ip->protocol = IPPROTO_TCP;
ip->saddr = source->s_addr;
ip->daddr = victim->s_addr;
ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));

if (debugging > 1) {
printf("Raw TCP packet creation completed!  Here it is:\n");
readtcppacket(packet,ntohs(ip->tot_len));
}
if (debugging > 1)
  printf("\nTrying sendto(%d , packet, %d, 0 , %s , %d)\n",
         sd, ntohs(ip->tot_len), inet_ntoa(*victim),
         sizeof(struct sockaddr_in));
if ((res = sendto(sd, packet, ntohs(ip->tot_len), 0,
                  (struct sockaddr *)&sock, sizeof(struct sockaddr_in))) == -1)
  {
    perror("sendto in send_tcp_raw");
    if (source_malloced) free(source);
    return -1;
  }
if (debugging > 1) printf("successfully sent %d bytes of raw_tcp!\n", res);

if (source_malloced) free(source);
return res;
}

/* A simple program I wrote to help in debugging, shows the important fields
   of a TCP packet*/
int readtcppacket(char *packet, int readdata) {
struct iphdr *ip = (struct iphdr *) packet;
struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct iphdr));
char *data = packet +  sizeof(struct iphdr) + sizeof(struct tcphdr);
int tot_len;
```

```
struct in_addr bullshit, bullshit2;
char sourcehost[16];
int i;

if (!packet) {
  fprintf(stderr, "readtcppacket: packet is NULL!\n");
  return -1;
    }
bullshit.s_addr = ip->saddr; bullshit2.s_addr = ip->daddr;
tot_len = ntohs(ip->tot_len);
strncpy(sourcehost, inet_ntoa(bullshit), 16);
i =  4 * (ntohs(ip->ihl) + ntohs(tcp->th_off));
if (ip->protocol == IPPROTO_TCP)
  if (ip->frag_off) printf("Packet is fragmented, offset field: %u",
                           ip->frag_off);
  else {
    printf("TCP packet: %s:%d -> %s:%d (total: %d bytes)\n", sourcehost,
           ntohs(tcp->th_sport), inet_ntoa(bullshit2),
           ntohs(tcp->th_dport), tot_len);
    printf("Flags: ");
    if (!tcp->th_flags) printf("(none)");
    if (tcp->th_flags & TH_RST) printf("RST ");
    if (tcp->th_flags & TH_SYN) printf("SYN ");
    if (tcp->th_flags & TH_ACK) printf("ACK ");
    if (tcp->th_flags & TH_PUSH) printf("PSH ");
    if (tcp->th_flags & TH_FIN) printf("FIN ");
    if (tcp->th_flags & TH_URG) printf("URG ");
    printf("\n");
    printf("ttl: %hi ", ip->ttl);
    if (tcp->th_flags & (TH_SYN | TH_ACK)) printf("Seq: %lu\tAck: %lu\n",
                                                  tcp->th_seq, tcp->th_ack);
    else if (tcp->th_flags & TH_SYN) printf("Seq: %lu\n", ntohl(tcp->th_seq));
    else if (tcp->th_flags & TH_ACK) printf("Ack: %lu\n", ntohl(tcp->th_ack));
  }
if (readdata && i < tot_len) {
printf("Data portion:\n");
while(i < tot_len)  printf("%2X%c", data[i], (++i%16)? ' ' : '\n');
printf("\n");
}
return 0;
}

/* We don't exactly need real crypto here (thank god!)\n"*/
int shortfry(unsigned short *ports) {
int num;
unsigned short tmp;
int i;
```

```
for(i=0; i < number_of_ports; i++) {
  num = rand() % (number_of_ports);
  tmp = ports[i];
  ports[i] = ports[num];
  ports[num] = tmp;
}
return 1;
}


/* Much of this is swiped from my send_tcp_raw function above, which
   doesn't support fragmentation */
int send_small_fragz(int sd, struct in_addr *source, struct in_addr *victim,
                     int sport, int dport, int flags) {

struct pseudo_header {
/*for computing TCP checksum, see TCP/IP Illustrated p. 145 */
  unsigned long s_addr;
  unsigned long d_addr;
  char zer0;
  unsigned char protocol;
  unsigned short length;
};
/*In this placement we get data and some field alignment so we aren't wasting
  too much to compute the TCP checksum.*/
char packet[sizeof(struct iphdr) + sizeof(struct tcphdr) + 100];
struct iphdr *ip = (struct iphdr *) packet;
struct tcphdr *tcp = (struct tcphdr *) (packet + sizeof(struct iphdr));
struct pseudo_header *pseudo = (struct pseudo_header *) (packet + sizeof(struct iphdr) - sizeof(struct
pseudo_header));
char *frag2 = packet + sizeof(struct iphdr) + 16;
struct iphdr *ip2 = (struct iphdr *) (frag2 - sizeof(struct iphdr));
int res;
struct sockaddr_in sock;
int id;

/*Why do we have to fill out this damn thing? This is a raw packet, after all */
sock.sin_family = AF_INET;
sock.sin_port = htons(dport);
sock.sin_addr.s_addr = victim->s_addr;

bzero(packet, sizeof(struct iphdr) + sizeof(struct tcphdr));

pseudo->s_addr = source->s_addr;
pseudo->d_addr = victim->s_addr;
pseudo->protocol = IPPROTO_TCP;
pseudo->length = htons(sizeof(struct tcphdr));
```

```
  tcp->th_sport = htons(sport);
  tcp->th_dport = htons(dport);
  tcp->th_seq = rand() + rand();

  tcp->th_off = 5 /*words*/;
  tcp->th_flags = flags;

  tcp->th_win = htons(2048); /* Who cares */

  tcp->th_sum = in_cksum((unsigned short *)pseudo,
                         sizeof(struct tcphdr) + sizeof(struct pseudo_header));

  /* Now for the ip header of frag1 */
  bzero(packet, sizeof(struct iphdr));
  ip->version = 4;
  ip->ihl = 5;
  /*RFC 791 allows 8 octet frags, but I get "operation not permitted" (EPERM)
    when I try that.  */
  ip->tot_len = htons(sizeof(struct iphdr) + 16);
  id = ip->id = rand();
  ip->frag_off = htons(MORE_FRAGMENTS);
  ip->ttl = 255;
  ip->protocol = IPPROTO_TCP;
  ip->saddr = source->s_addr;
  ip->daddr = victim->s_addr;
  ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));

  if (debugging > 1) {
    printf("Raw TCP packet fragment #1 creation completed!  Here it is:\n");
    hdump(packet,20);
  }
  if (debugging > 1)
    printf("\nTrying sendto(%d , packet, %d, 0 , %s , %d)\n",
           sd, ntohs(ip->tot_len), inet_ntoa(*victim),
           sizeof(struct sockaddr_in));
  if ((res = sendto(sd, packet, ntohs(ip->tot_len), 0,
                    (struct sockaddr *)&sock, sizeof(struct sockaddr_in))) == -1)
    {
      perror("sendto in send_syn_fragz");
      return -1;
    }
  if (debugging > 1) printf("successfully sent %d bytes of raw_tcp!\n", res);

  /* Create the second fragment */
  bzero(ip2, sizeof(struct iphdr));
  ip2->version = 4;
  ip2->ihl = 5;
  ip2->tot_len = htons(sizeof(struct iphdr) + 4); /* the rest of our TCP packet */
```

```
ip2->id = id;
ip2->frag_off = htons(2);
ip2->ttl = 255;
ip2->protocol = IPPROTO_TCP;
ip2->saddr = source->s_addr;
ip2->daddr = victim->s_addr;
ip2->check = in_cksum((unsigned short *)ip2, sizeof(struct iphdr));
if (debugging > 1) {
  printf("Raw TCP packet fragment creation completed!  Here it is:\n");
  hdump(packet,20);
}
if (debugging > 1)
  printf("\nTrying sendto(%d , ip2, %d, 0 , %s , %d)\n", sd,
         ntohs(ip2->tot_len), inet_ntoa(*victim), sizeof(struct sockaddr_in));
if ((res = sendto(sd, ip2, ntohs(ip2->tot_len), 0,
                  (struct sockaddr *)&sock, sizeof(struct sockaddr_in))) == -1)
  {
    perror("sendto in send_tcp_raw");
    return -1;
  }
return 1;
}

/* Hex dump */
void hdump(unsigned char *packet, int len) {
unsigned int i=0, j=0;

printf("Here it is:\n");

for(i=0; i < len; i++){
  j = (unsigned) (packet[i]);
  printf("%-2X ", j);
  if (!((i+1)%16))
    printf("\n");
  else if (!((i+1)%4))
    printf("  ");
}
printf("\n");
}


portlist fin_scan(struct in_addr target, unsigned short *portarray,
                  struct in_addr *source, int fragment, portlist *ports) {

int rawsd, tcpsd;
int done = 0, badport, starttime, someleft, i, j=0, retries=2;
int source_malloc = 0;
int waiting_period = retries, sockaddr_in_size = sizeof(struct sockaddr_in);
```

```
  int bytes, dupesinarow = 0;
  unsigned long timeout;
  struct hostent *myhostent;
  char response[65535], myname[513];
  struct iphdr *ip = (struct iphdr *) response;
  struct tcphdr *tcp;
  unsigned short portno[max_parallel_sockets], trynum[max_parallel_sockets];
  struct sockaddr_in stranger;


  timeout = (global_delay)? global_delay : (global_rtt)? (1.2 * global_rtt) + 10000 : 1e5;
  bzero(&stranger, sockaddr_in_size);
  bzero(portno, max_parallel_sockets * sizeof(unsigned short));
  bzero(trynum, max_parallel_sockets * sizeof(unsigned short));
  starttime = time(NULL);


  if (debugging || verbose)
    printf("Initiating FIN stealth scan against %s (%s), sleep delay: %ld useconds\n", current_name,
  inet_ntoa(target), timeout);

  if (!source) {
    if (ouraddr.s_addr) {
      source = &ouraddr;
    }
    else {
    source = safe_malloc(sizeof(struct in_addr));
    source_malloc = 1;
    if (gethostname(myname, MAXHOSTNAMELEN) ||
        !(myhostent = gethostbyname(myname)))
      fatal("Your system is fucked up.\n");
    memcpy(source, myhostent->h_addr_list[0], sizeof(struct in_addr));
    }
    if (debugging || verbose)
      printf("We skillfully deduced that your address is %s\n",
             inet_ntoa(*source));
  }

  if ((rawsd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0 )
    perror("socket trobles in fin_scan");

  if ((tcpsd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0 )
    perror("socket trobles in fin_scan");

  unblock_socket(tcpsd);
  while(!done) {
    for(i=0; i <  max_parallel_sockets; i++) {
      if (!portno[i] && portarray[j]) {
```

```
        portno[i] = portarray[j++];
      }
      if (portno[i]) {
      if (fragment)
        send_small_fragz(rawsd, source, &target, MAGIC_PORT, portno[i], TH_FIN);
      else send_tcp_raw(rawsd, source , &target, MAGIC_PORT,
                        portno[i], 0, 0, TH_FIN, 0, 0, 0);
      usleep(10000); /* *WE* normally do not need this, but the target
                        lamer often does */
      }
    }

    usleep(timeout);
    dupesinarow = 0;
    while ((bytes = recvfrom(tcpsd, response, 65535, 0, (struct sockaddr *)
                             &stranger, &sockaddr_in_size)) > 0)
      if (ip->saddr == target.s_addr) {
        tcp = (struct tcphdr *) (response + 4 * ip->ihl);
        if (tcp->th_flags & TH_RST) {
          badport = ntohs(tcp->th_sport);
          if (debugging > 1) printf("Nothing open on port %d\n", badport);
          /* delete the port from active scanning */
          for(i=0; i < max_parallel_sockets; i++)
            if (portno[i] == badport) {
              if (debugging && trynum[i] > 0)
                printf("Bad port %d caught on fin scan, try number %d\n",
                       badport, trynum[i] + 1);
              trynum[i] = 0;
              portno[i] = 0;
              break;
            }
          if (i == max_parallel_sockets) {
            if (debugging)
              printf("Late packet or dupe, deleting port %d.\n", badport);
            dupesinarow++;
            if (ports) deleteport(ports, badport, IPPROTO_TCP);
          }
        }
        else
          if (debugging > 1) {
            printf("Strange packet from target%d!  Here it is:\n",
                   ntohs(tcp->th_sport));
            if (bytes >= 40) readtcppacket(response,1);
            else hdump(response,bytes);
          }
      }

    /* adjust waiting time if neccessary */
```

```
    if (dupesinarow > 6) {
      if (debugging || verbose)
        printf("Slowing down send frequency due to multiple late packets.\n");
      if (timeout < 10 * ((global_delay)? global_delay: global_rtt + 20000)) timeout *= 1.5;
      else {
        printf("Too many late packets despite send frequency decreases, skipping scan.\n");
        if (source_malloc) free(source);
        return *ports;
      }
    }


    /* Ok, collect good ports (those that we haven't received responses too
       after all our retries */
    someleft = 0;
    for(i=0; i < max_parallel_sockets; i++)
      if (portno[i]) {
        if (++trynum[i] >= retries) {
          if (verbose || debugging)
            printf("Good port %d detected by fin_scan!\n", portno[i]);
          addport(ports, portno[i], IPPROTO_TCP, NULL);
      send_tcp_raw( rawsd, source, &target, MAGIC_PORT, portno[i], 0, 0,
                    TH_FIN, 0, 0, 0);
          portno[i] = trynum[i] = 0;
        }
        else someleft = 1;
      }

    if (!portarray[j] && (!someleft || --waiting_period <= 0)) done++;
  }

  if (debugging || verbose)
    printf("The TCP stealth FIN scan took %ld seconds to scan %d ports.\n",
           time(NULL) - starttime, number_of_ports);
  if (source_malloc) free(source);
  close(tcpsd);
  close(rawsd);
  return *ports;
  }

  int ftp_anon_connect(struct ftpinfo *ftp) {
  int sd;
  struct sockaddr_in sock;
  int res;
  char recvbuf[2048];
  char command[512];

  if (verbose || debugging)
```

```
  printf("Attempting connection to ftp://%s:%s@%s:%i\n", ftp->user, ftp->pass,
         ftp->server_name, ftp->port);

if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
  perror("Couldn't create ftp_anon_connect socket");
  return 0;
}

sock.sin_family = AF_INET;
sock.sin_addr.s_addr = ftp->server.s_addr;
sock.sin_port = htons(ftp->port);
res = connect(sd, (struct sockaddr *) &sock, sizeof(struct sockaddr_in));
if (res < 0 ) {
  printf("Your ftp bounce proxy server won't talk to us!\n");
  exit(1);
}
if (verbose || debugging) printf("Connected:");
while ((res = recvtime(sd, recvbuf, 2048,7)) > 0)
  if (debugging || verbose) {
    recvbuf[res] = '\0';
    printf("%s", recvbuf);
  }
if (res < 0) {
  perror("recv problem from ftp bounce server");
  exit(1);
}

snprintf(command, 511, "USER %s\r\n", ftp->user);
send(sd, command, strlen(command), 0);
res = recvtime(sd, recvbuf, 2048,12);
if (res <= 0) {
  perror("recv problem from ftp bounce server");
  exit(1);
}
recvbuf[res] = '\0';
if (debugging) printf("sent username, received: %s", recvbuf);
if (recvbuf[0] == '5') {
  printf("Your ftp bounce server doesn't like the username \"%s\"\n",
         ftp->user);
  exit(1);
}
snprintf(command, 511, "PASS %s\r\n", ftp->pass);
send(sd, command, strlen(command), 0);
res = recvtime(sd, recvbuf, 2048,12);
if (res < 0) {
  perror("recv problem from ftp bounce server\n");
  exit(1);
}
```

```
if (!res) printf("Timeout from bounce server ...");
else {
recvbuf[res] = '\0';
if (debugging) printf("sent password, received: %s", recvbuf);
if (recvbuf[0] == '5') {
  fprintf(stderr, "Your ftp bounce server refused login combo (%s/%s)\n",
         ftp->user, ftp->pass);
  exit(1);
}
}
while ((res = recvtime(sd, recvbuf, 2048,2)) > 0)
  if (debugging) {
    recvbuf[res] = '\0';
    printf("%s", recvbuf);
  }
if (res < 0) {
  perror("recv problem from ftp bounce server");
  exit(1);
}
if (verbose) printf("Login credentials accepted by ftp server!\n");

ftp->sd = sd;
return sd;
}

int recvtime(int sd, char *buf, int len, int seconds) {

int res;
struct timeval timeout = {seconds, 0};
fd_set readfd;

FD_ZERO(&readfd);
FD_SET(sd, &readfd);
res = select(sd + 1, &readfd, NULL, NULL, &timeout);
if (res > 0 ) {
res = recv(sd, buf, len, 0);
if (res >= 0) return res;
perror("recv in recvtime");
return 0;
}
else if (!res) return 0;
perror("select() in recvtime");
return -1;
}

portlist bounce_scan(struct in_addr target, unsigned short *portarray,
                     struct ftpinfo *ftp, portlist *ports) {
int starttime,  res , sd = ftp->sd,  i=0;
```

```
  char *t = (char *)⌖
  int retriesleft = FTP_RETRIES;
  char recvbuf[2048];
  char targetstr[20];
  char command[512];
  snprintf(targetstr, 20, "%d,%d,%d,%d,0,", UC(t[0]), UC(t[1]), UC(t[2]), UC(t[3]));
  starttime = time(NULL);
  if (verbose || debugging)
    printf("Initiating TCP ftp bounce scan against %s (%s)\n",
           current_name,  inet_ntoa(target));
  for(i=0; portarray[i]; i++) {
    snprintf(command, 512, "PORT %s%i\r\n", targetstr, portarray[i]);
    if (send(sd, command, strlen(command), 0) < 0 ) {
      perror("send in bounce_scan");
      if (retriesleft) {
        if (verbose || debugging)
          printf("Our ftp proxy server hung up on us!  retrying\n");
        retriesleft--;
        close(sd);
        ftp->sd = ftp_anon_connect(ftp);
        if (ftp->sd < 0) return *ports;
        sd = ftp->sd;
        i--;
      }
      else {
        fprintf(stderr, "Our socket descriptor is dead and we are out of retries. Giving up.\n");
        close(sd);
        ftp->sd = -1;
        return *ports;
      }
    } else { /* Our send is good */
      res = recvtime(sd, recvbuf, 2048,15);
      if (res <= 0) perror("recv problem from ftp bounce server\n");

      else { /* our recv is good */
        recvbuf[res] = '\0';
        if (debugging) printf("result of port query on port %i: %s",
                              portarray[i],  recvbuf);
        if (recvbuf[0] == '5') {
          if (portarray[i] > 1023) {
          fprintf(stderr, "Your ftp bounce server sucks, it won't let us feed bogus ports!\n");
          exit(1);
        }
        else {
          fprintf(stderr, "Your ftp bounce server doesn't allow priviliged ports, skipping them.\n");
          while(portarray[i] && portarray[i] < 1024) i++;
          if (!portarray[i]) {
            fprintf(stderr, "And you didn't want to scan any unpriviliged ports.  Giving up.\n");
```

```
            /*      close(sd);
            ftp->sd = -1;
            return *ports;*/
            /* screw this gentle return crap!  This is an emergency! */
            exit(1);
          }
        }
        }
      else  /* Not an error message */
        if (send(sd, "LIST\r\n", 6, 0) > 0 ) {
          res = recvtime(sd, recvbuf, 2048,12);
          if (res <= 0)  perror("recv problem from ftp bounce server\n");
          else {
            recvbuf[res] = '\0';
            if (debugging) printf("result of LIST: %s", recvbuf);
            if (!strncmp(recvbuf, "500", 3)) {
              /* fuck, we are not aligned properly */
              if (verbose || debugging)
                printf("misalignment detected ... correcting.\n");
               res = recvtime(sd, recvbuf, 2048,10);
            }
            if (recvbuf[0] == '1' || recvbuf[0] == '2') {
              if (verbose || debugging) printf("Port number %i appears good.\n",
                                  portarray[i]);
              addport(ports, portarray[i], IPPROTO_TCP, NULL);
              if (recvbuf[0] == '1') {
              res = recvtime(sd, recvbuf, 2048,5);
              recvbuf[res] = '\0';
              if ((res > 0) && debugging) printf("nxt line: %s", recvbuf);
              }
            }
          }
        }
      }
    }
  }
  if (debugging || verbose)
    printf("Scanned %d ports in %ld seconds via the Bounce scan.\n",
           number_of_ports, time(NULL) - starttime);
  return *ports;
  }

  /* parse a URL stype ftp string of the form user:pass@server:portno */
  int parse_bounce(struct ftpinfo *ftp, char *url) {
  char *p = url,*q, *s;

  if ((q = strrchr(url, '@'))) /*we have username and/or pass */ {
    *(q++) = '\0';
```

```
    if ((s = strchr(q, ':')))
      { /* has portno */
        *(s++) = '\0';
        strncpy(ftp->server_name, q, MAXHOSTNAMELEN);
        ftp->port = atoi(s);
      }
    else  strncpy(ftp->server_name, q, MAXHOSTNAMELEN);

    if ((s = strchr(p, ':'))) { /* User AND pass given */
      *(s++) = '\0';
      strncpy(ftp->user, p, 63);
      strncpy(ftp->pass, s, 255);
    }
    else { /* Username ONLY given */
      printf("Assuming %s is a username, and using the default password: %s\n",
             p, ftp->pass);
      strncpy(ftp->user, p, 63);
    }
  }
  else /* no username or password given */
    if ((s = strchr(url, ':'))) { /* portno is given */
      *(s++) = '\0';
      strncpy(ftp->server_name, url, MAXHOSTNAMELEN);
      ftp->port = atoi(s);
    }
    else  /* default case, no username, password, or portnumber */
      strncpy(ftp->server_name, url, MAXHOSTNAMELEN);

  ftp->user[63] = ftp->pass[255] = ftp->server_name[MAXHOSTNAMELEN] = 0;

  return 1;
  }



  /*
   *      I'll bet you've never seen this function before (yeah right)!
   *      standard swiped checksum routine.
   */
  unsigned short in_cksum(unsigned short *ptr,int nbytes) {

  register long           sum;            /* assumes long == 32 bits */
  u_short                 oddbyte;
  register u_short        answer;         /* assumes u_short == 16 bits */

  /*
   * Our algorithm is simple, using a 32-bit accumulator (sum),
   * we add sequential 16-bit words to it, and at the end, fold back
```

```
 * all the carry bits from the top 16 bits into the lower 16 bits.
 */

sum = 0;
while (nbytes > 1)  {
sum += *ptr++;
nbytes -= 2;
}

/* mop up an odd byte, if necessary */
if (nbytes == 1) {
oddbyte = 0;            /* make sure top half is zero */
*((u_char *) &oddbyte) = *(u_char *)ptr;   /* one byte only */
sum += oddbyte;
}

/*
 * Add back carry outs from top 16 bits to low 16 bits.
 */

sum  = (sum >> 16) + (sum & 0xffff);    /* add high-16 to low-16 */
sum += (sum >> 16);                     /* add carry */
answer = ~sum;          /* ones-complement, then truncate to 16 bits */
return(answer);
}
```

What can nmap Do:

Network Mapped (Nmap) is a network scanning and host detection tool that is very useful during several steps of penetration testing.

Nmap is not limited to merely gathering information and enumeration, but it is also powerful utility that can be used as a vulnerability detector or a security scanner.

So Nmap is a multipurpose tool, and it can be run on many different operating systems including Windows, Linux, BSD, and Mac.

Nmap is a very powerful utility that can be used to:

Detect the live host on the network (host discovery)

Detect the open ports on the host (port discovery or enumeration)

Detect the software and the version to the respective port (service discovery)

Detect the operating system, hardware address, and the software version

Detect the vulnerability and security holes (Nmap scripts)

Nmap is a very common tool, and it is available for both the command line interface and the graphical user interface.

How to use Nmap? You might have heard this question many times before, but in my opinion, this is not the right question to ask.

The best way to start off exploring Nmap is to ask: How can I use Nmap effectively? This article was written in an effort to answer that question.

Nmap uses different techniques to perform scanning including: TCP connect() scanning, TCP reverse ident

scanning, FTP bounce scanning and so on.
All these types of scanning have their own advantages and disadvantages, and we will discuss them as we go on.
It is a basic scan, and it is also called half-open scanning because this technique allows Nmap to get information from the remote host without the complete TCP handshake process, Nmap sends SYN packets to the destination, but it does not create any sessions, As a result, the target computer can't create any log of the interaction because no session was initiated, making this feature an advantage of the TCP SYN scan.

If there is no scan type mentioned on the command, then avTCP SYN scan is used by default, but it requires the root/administrator privileged.

```
# nmap -sS 192.168.1.1
```

TCP connect() scan (-sT)

This the default scanning technique used, if and only if the SYN scan is not an option, because the SYN scan requires root privilege. Unlike the TCP SYN scan, it completes the normal TCP three way handshake process and requires the system to call connect(), which is a part of the operating system. Keep in mind that this technique is only applicable to find out the TCP ports, not the UDP ports.

```
# nmap -sT 192.168.1.1
```

UDP Scan (-sU)

As the name suggests, this technique is used to find an open UDP port of the target machine. It does not require any SYN packet to be sent because it is targeting the UDP ports. But we can make the scanning more effective by using -sS along with –sU. UDP scans send the UDP packets to the target machine, and waits for a response—if an error message arrives saying the ICMP is unreachable, then it means that the port is closed; but if it gets an appropriate response, then it means that the port is open.

```
# nmap -sU 192.168.1.1
```

FIN Scan (-sF)

Sometimes a normal TCP SYN scan is not the best solution because of the firewall. IDS and IPS scans might be deployed on the target machine, but a firewall will usually block the SYN packets. A FIN scan sends the packet only set with a FIN flag, so it is not required to complete the TCP handshaking.

```
root@bt:~# nmap -sF 192.168.1.8

Starting Nmap 5.51 ( http://nmap.org ) at 2012-07-08 19:21 PKT

Nmap scan report for 192.168.1.8

Host is up (0.000026s latency).

Not shown: 999 closed ports
```

```
PORT STATE SERVICE

111/tcp open|filtered rpcbind
```

The target computer is not able to create a log of this scan (again, an advantage of FIN). Just like a FIN scan, we can perform an xmas scan (-sX) and Null scan (-sN). The idea is same but there is a difference between each type of scan. For example, the FIN scan sends the packets containing only the FIN flag, where as the Null scan does not send any bit on the packet, and the xmas sends FIN, PSH, and URG flags.

Ping Scan (-sP)

Ping scanning is unlike the other scan techniques because it is only used to find out whether the host is alive or not, it is not used to discover open ports. Ping scans require root access s ICMP packets can be sent, but if the user does not have administrator privilege, then the ping scan uses connect() call.

```
# nmap -sP 192.168.1.1
```

Version Detection (-sV)

Version detection is the right technique that is used to find out what software version is running on the target computer and on the respective ports. It is unlike the other scanning techniques because it is not used to detect the open ports, but it requires the information from open ports to detect the software version. In the first step of this scan technique, version detection uses the TCP SYN scan to find out which ports are open.

```
# nmap -sV 192.168.1.1
```

Idle Scan (-sI)

Idle scan is one of my favorite techniques, and it is an advance scan that provides complete anonymity while scanning. In idle scan, Nmap doesn’t send the packets from your real IP address—instead of generating the packets from the attacker machine, Nmap uses another host from the target network to send the packets. Let’s consider an example to understand the concept of idle scan:

```
nmap -sI zombie_host target_host
```

```
# nmap -sI 192.168.1.6 192.168.1.1
```

The idle scan technique (as mentioned above) is used to discover the open ports on 192.168.1.1 while it uses the zombie_host (192.168.1.6) to communicate with the target host. So this is an ideal technique to scan a target computer anonymously.

There are many other scanning techniques are available like FTP bounce, fragmentation scan, IP protocol scan. and so on; but we have discussed the most important scanning techniques (although all of the scanning techniques can important depending on the situation you are dealing with).

Your arsenal

Which unix you use is entirely up to you. I suggest you verify that nmap will run on your flavour of unix before deciding on one in particular. Linux, BSD, and SunOS are good choices. HP-UX, AIX, IRIX, SCO, XENIX, and the rest of the plethora of unix clones MAY work, but you'll be sailing untested waters.

Here's the list of other programs you might want to install:
* nmap (The defacto security scanner)
* nc (NetCat - The "IP Swiss Army Knife")
* tcpdump (The original sniffer. Personally, I prefer snort, but tcpdump will run on just about any unix out there...)
* nping (Nping is an open source tool for network packet generation, response analysis and response time measurement. Nping can generate network packets for a wide range of protocols, allowing users full control over protocol headers.)
* lynx (Excellent console based webbrowser. Always handy to have)
* ncat (The modified version of nc by FYODOR)
* Ncrack (Ncrack is a high-speed network authentication cracking tool. It was built to help companies secure their networks by proactively testing all their hosts and networking devices for poor passwords.)

For the Tiger Team member involved in physical audits, nothing is more valuable than a laptop with a network card in it, running unix. It can be carried around throughout the company building, and plugged into ethernet jacks and start sniffing/scanning immediatley. I personally would suggest NetBSD or OpenBSD for this purpose, because they are small, fast, mobile, and VERY capable unix systems.

Fundamentals

In order to understand how to make your scans efficient and effective, you have to grasp a few concepts about TCP/IP networking, and how the operating system accomplishes this.

All of the internet (which relies very heavily on TCP/IP) uses packets to send data back and forth. There is no direct stream of data like, say, a telephone connection. Instead, the computer sends packets, which are processed, filtered, fragmented, and routed throughout other computers, until this packet of data reaches its ultimate destination.

If this concept is completely foreign to you, you will probably have much difficulty understanding basic portscanning. In any case, you must understand how computers can cary on simultaneous connections to different computers at once. The explanation can actually get quite detailed, but is the basis of portscanning. Here are the basics:

Every internet connected computer has an IP address (either permanently designated to that very machine, or dynamically assigned upon connecting to the network) which uniquely identifies that machine to all other machines on the network. Think of the IP address as a name for the computer. All packets you want to send to a particular computer, you would slap that address onto the packets you want to send, and throw that packet out onto the internet, and let the internet take care of getting that packet to the machine with the destination's IP address.

But what if I want to have 2 connections to one IP address simultaneously? How will my computer know which packets are for which connection? Ports, my friend. Each internet connected computer has 65535 potential ports available to them. Keep in mind that usually only ports greater than 1024 are designated for general use, with the rest of the ports reserved for services on machines, like webservers (port 80), FTP servers (port 21), SSH servers (port 22). At least that's how it's supposed to work. There are to many exceptions to list. For instance, IRC servers (port 6667), MySQL (port 3306). Note: One of the reasons why I insisted you have root access on your unix machine is that if you want to listen on a port below 1024 in unix, you simply must have root privileges. Unix won't let you "bind" to that port otherwise. Note that this is not necessary for most portscanning techniques, but can be very valuable in certain situations.

Okay. So what happens when I, as a client, want to open a connection to a computer on the internet on a specific port?

This also is quite involved, so I'll summarize.

You slap together a small packet which has a special "flag" set on it: SYN (Synchronizing). Then you slap on your own IP address (the "source IP"). Then you pick a random, unused port between 1024 and 65535, and slap that on the packet (the "source port"). Next you slap on the destination's IP address (the "destination IP"), and the port you want to connect to ("the destination port"). Then you send the packet onto the internet and (hopefully) a few milliseconds later it will arrive at its destination.

From here on in, all TCP/IP packets for this connection will use the same ports and IP addresses, except, of course, when the server sends a packet, then the source becomes the destination, and vice versa.

So, upon recieving the packet, the server must make a decision. First off, does the server want to talk to this IP address that's knocking on its door, so to speak. If it does, it will check if the source port is indeed open for communication (SYNchronization, as it were). If so, it replies with another packet, except this one has not only the SYN flag set, but also the ACK. If, on the other hand, this port is NOT open, it sends an ICMP message (which is NOT, in fact TCP/IP, but operates over the internet nonetheless): an RST (reset). The user at the client's computer would probably get an error message like "the service you requested is not online at <insert IP address here>".

Assuming that all went well and a SYN/ACK was transmitted, the client will then reply with a packet with only the ACK flag set. Then, the 2 can begin sending data packets between them (which

all, as a matter of fact, have the ACK flag set). Now this may seem like an awful amount of trouble to go through to establish a connection... Why not just start sending packets? Well, if you think about it, this is the minimal amount of communication required for verifying to both parties that data can, in fact, be sent in 2 directions. Interestingly, there is a second method of sending data over the internet as well, called UDP/IP, which basically DOES just start sending packets. Unfortunatley, UDP/IP is renowned for its unreliability. Not only are packets NOT guaranteed to arrive in order and without data corruption, but their arrival isn't guaranteed at all! UDP does have its place, though, especially when you don't need full data integrity (Streaming audio is the classic exmple). nmap offers UDP/IP scanning techniques too, by the way.)

A couple notes before we proceed to actual port scanning:
* Here's what an IP address might look like: 192.168.24.53.
* All information in this manual deals with IPv4.
* Often, IPs and DNS names can be used interchangably. Think about DNS names, like hypervivid.com, as being turned into an IP by the operating system before being used by the networking code.
* TCP/IP connections are also closed via ICMP (ideally), or by timeout (as is often the case)
* If a client that DIDN'T request a connection ever recieves a SYN/ACK packet, it is supposed to reply with an ICMP message responding appropriately.

Port-Scanning

Basically, for this section, you can simply throw out all other port scanners, and learn how to use nmap. nmap is an extremely powerful, free security scanner that, in my opinion, beats the pants off even the most pricey commercial scanners on the market. A skilled nmap wielder can scan through firewalls, determine remote operating systems, preform literally dozens of different types of scans, and even bounce scans off of FTP servers, so the victim will think the FTP server is scanning them. Congratulations go to Fyodor, the author of nmap, and all the other hackers who helped make nmap the incredible beast that it is today.

This is by no means a complete nmap manual. The most complete documentation is, of course, the freely available source code. Next to that, you have to rely on the man page, even though the man page neglects to mention several interesting features of nmap that you could only ever find buried deep inside the source. I'll touch upon several of the different scan types, OS detection, tips, and a few of these "undocumented features".
Okay, here's a very simple nmap scan:

```
nmap 192.168.9.3
```

This uses many defaults. It defaults to a standard TCP scan. You could also have done it like so:

```
nmap -sT 192.168.9.3
```

Description of -sT:

Basically, this scan attempts a full TCP/IP connection as described above with every port listed in nmap's custom /etc/services file. It then reports all ports it finds open.

Note: If you're not, in fact, scanning over the network (for example, nmap localhost), this is the scan you want to use. It's VERY fast, and most IP loggers don't log TCP/IP connection attempts from 'localhost'.

Note: If you don't have root on the box you're scanning from, this is the only "standard" scan you can do as this scan doesn't use raw sockets, and instead relies on the ubiquitous connect() system call.

Advantages:

* Fairly fast scan
* DOESN'T require root privileges.

Disadvantages:

* VERY easily detectable.

Description of -sS:

So let's consider a "SYN scan", or "half-open scanning" as it is commonly called. Basically it works like this: Your machine injects a SYN packet of the appropriate port and IP address onto the network stream so that your OS doesn't even know it's sending out this packet, that way it won't be expecting a SYN/ACK packet back from the server. Then, nmap starts listening directly into the network stream until it sees either SYN/ACKs or the ICMP messages saying the port is closed. That's all nmap has to do. The OS, upon recieving this SYN/ACK packet from the scan victim, thinks "I didn't request this connection, I'd better send an ICMP error message...". It's important that your OS does this, otherwise the server will sit there expecting an ACK packet for quite some time, which eats up memory and such. This is referred to as SYN flooding, which is something you DON'T want to do if stealthiness is your game. (I'll touch briefly on how nmap can be used as a very powerful SYN flooding tool a bit later). It's called half-open scanning because a full connection is never made. nmap takes just enough information it needs, and never has to open a full connection.

Advantages:

* Fairly stealthy...
* Most firewalling software doesn't log these particular scans, although many do.
* Reasonably fast scan. Sometimes faster than -sT, sometimes not.

Disadvatages:
* You need to be root to preform this scan.
* I've found this scan can be signifigantly slower than -sT on older hardware with ISA ethernet cards for whatever reason.
* Still sends packets to the victim that have your IP on them.

Description of -sF, -sX, -sN:

There are 3 other types of "standard" TCP/IP scans: -sF, -sX, and -sN. Basically they have to do with setting various TCP/IP flags on the packets that you scan with, relying on the standard methods of handling these unusual packets set out in the networking standards: the RFCs. To be honest, I get very little use out of any of these scans. They aren't much more difficult to detect and log than -sS, and you can't be certain that all scanned OSs respond according to the RFCs, so chances are you'll have to use a different scan just to confirm the results of these ones! Of course, there are people that really like these scans. See the nmap man page for details.

Advantages:

* Debatably more stealthy than SYN scans.
* These scans will impress your friends if you do them right. :)

Disadvantages:

* Sometimes you get cryptic, misleading results.
* Still sends packets to the victim that have your IP on them.

Description of -sI:

This is an exciting new scan that has just recently been incorporated into nmap. Few people seem to know how it works, but Fyodor has promised us some documentation on this scan soon. If it works as advertised, this scan has an incredible amount of potential. I've played with

it a bit, but haven't got it working.

Advantages:

* No packets sent to the victim from your IP!
* Your friends will idol you if you do them right. :)

Disadvantages:

* Probably has some serious limitations. This document will be updated when I understand more of this scan.

Description of -sA:

This little doozy of a scan is highly underrated, in my opinion. It has a very large amount of legitimate network debugging uses. Basically it works like this: You want to find out if a firewall is filtering certain ports (filtering means not letting you see which ports are open behind the firewall by not returning ICMP messages saying you can't be reached). Good for mapping out firewall rulesets. Plus if you know a port is open behind a firewall, you can deduce wether or not you're dealing with a stateful packet filter. You can't find out which ports are open on internal hosts, but you can determine which hosts to scan when you get that short, 2 minute opportunity on the internal network after slipping through a window right before closing time. ;) Seriously, though, in the hands of somebody who understands firewalling principles, this scan can be incredibly valuable.

Note: When you're reading the nmap manpage, you'll see that it mentions the ACK packets have random sequence numbers and such. It seems everybodies favorite IDS (intrustion detection system), snort, was using a clever monitoring trick to see if somebody was mapping out the firewall ruleset. See, earlier versions of nmap had a fixed value, so it was trivial to pick them up, and find any GUARANTEED suspicious activity. That's some nice detective work by the people who make snort. Congrats go to Fyodor too, who fixed it, enabling us scanners to ACK scan free from worry. :)

Advantages:

* This stuff is probably never logged.
* Very useful for the skilled scanner.

Disadvantages:

* It isn't really a port scan.

Description of -sW:

This is an even cooler scan than -sA. Not only does it use ACK packets instead of SYN packets
to find your targets filtering rules, but it will also report open ports on the target!
It does this via some sort of TCP/IP window size anomalies. Unfortunatley, it only
works for some OSs... (It works for many popular ones, though, including VMS, SunOS 4.X, and BSD)

Advantages:

* Everything holds from -sA

Disadvantages:

* Doesn't work on all Operating Systems

Description of -sU:

This is a UDP scan. UDP, or Uniform Datagram Protocol, is a straightforward protocol. You
send a packet, if it gets there, good. Any applications bound to that UDP port on
the destination gets the data. If it's closed, they send back an ICMP message. That's basically
it. -sU just sends UDP packets to all ports in the services file.

UDP isn't that commonly used, but often times it's a good idea to do a quick UDP scan
on a host just to see whats up. For instance they might be running NFS or something.
%90 of the time, I'm simply not interested in UDP ports, so this feature doesn't get a lot of
use.

Advantages:

* Quite fast, I've noticed.
* It's good to have a UDP scanning feature.

Disadvantages:

* Not as useful as the other scans, usually.

Description of -sR:

Once there was a bug in a Beta version of nmap that killed -sR support temporarily... It went unnoticed for quite some time before Fyodor himself caught the bug. He was somewhat upset, because nobody had tested -sR on the Beta... I'm sorry Fyodor, but I've never once used -sR. Has to do with scanning for SunRPC ports, and finding their versions and whatnot. (See the manpage for info). Personally, I think this scan belongs in an application level scanner, NOT in nmap, but this is, of course, a matter for continuous debate.

Description of -sO:

This is an interesting scan. It doesn't scan for ports at all. It scans for open internet protocols that the target is accepting... I've only ever used this out of curiosity or to impress clients. :)

Description of -sL and -sP:

This would be a good time to bring up an excellent feature of nmap. It can scan more than one host from a single command line entry! Yes, that's right, you could scan your whole subnet with this command:

nmap -sS 192.168.0.0/24

or

nmap -sS 192.168.0.0-255

You can do all sorts of crazy combinations, like:

nmap -sS 1,3,9-11.3-9.-1.5-

That would scan a LOT of hosts... I REALLY like how Fyodor did this... It's so... useful!

Anyways, the -sL scan is handy when you have an extremely complicated IP range to scan, and you just want to make sure nmap is interpreting your command the way you want it. At least, that's the only use *I* can think of for it... See, -sL DOESN'T scan anything... It doesn't send any packets out on your network...

-sP is completely different. It pings hosts to see if they're up or not. This is a very

useful scan. Say I wanted to see which hosts are up on my subnet, I'd do a:

nmap -sP 192.168.0.1/24

nmap, by default uses both ICMP and ACK packets to identify wether a host is up or not, although you can change this behaviour with the -P switch, which will be discussed shortly.

-P<insert mode here>

nmap usually wants to confirm that a host is up before scanning it, so it sends out "pings" to the target and waits for its replies. Often, you'll want different methods of pinging hosts, or you don't want to ping at all. This is where the -P switch comes in:

See the man page for the different available modes. One of the most useful is -P0. This is for when you KNOW the host is up, but they are dropping pings.

-F

By default, nmap will scan all ports < 1024, plus the ports listed in nmap's special services file: /usr/local/share/nmap/nmap-services normally. The -F scan will only scan the ports listed in the services file. Actually, the "fast" scan isn't signifigantly faster than a normal scan. You save scanning 435 ports; maybe a few seconds. I almost never use this switch. If you really want a fast scan, think about which ports you are interested in, and use the -p switch (keep reading) to narrow down the scan. IF, however, you want to scan all 65535 ports on a host, you should use this scan:

nmap -p 1- target.com

Which leads us to our next switch...

-p <port range>

This is one of the most commonly used switches. It specifies what ports you actually want to scan. It uses similar syntax as specifying multiple host scanning, and is equally flexible.

Some examples:

```
nmap -p 2-500 target.com
nmap -p 2,4,8,29,500-9000 target.com
nmap -p -300,60000- target.com
nmap -sS -P0 -p 2-500 target.com
```

-O

Remote operating system identification is one of nmap's coolest features. The hacker community that develops nmap always sends in new OS "fingerprints" of new or obscure operating systems, so nmap has an incredible ability to remotley identify almost any internet connected computer.

Note: You must be root to use this feature.
Note: Usually, nmap must know about 1 open and 1 closed port, although there are exceptions.

-I

This option is designed to find out who "owns" the process that is listening on the open ports you've found. Many default unix installations actually come with an ident daemon running (port 113), so this option can be quite useful. Keep in mind, though, it is trivial to write an ident daemon that responds with any user for any process. Never fully rely on this scan.

-g <port>

This switch is an interesting one, and is rarely given the credit it deserves. If the target only responds to packets from a certain source port, this switch can help get your scan through, or if a firewall is doing something crazy, like only letting in port 53 as the source port. This is common on many poorly configured firewalls hiding a DNS server, I'm told. (Ideally, it would look at the IP too, not just the source port). As with any cool option, this switch requires root privileges.

-T <Paranoid|Sneaky|Polite|Normal|Aggressive|Insane>

This switch controls the speed that you want to portscan. Sometimes, spreading your port scan packets out will get by an IDS on the target system. Other times, you may just want to conserve (or saturate) network bandwidth. See the man page for details.

-v and -d

Verbose and debugging. It's often a good idea to use the -v switch, as you will often be able to determine more information about anything that went wrong. You can use as many -v switches as you like in the command, and each one cumulatively makes the output more and more verbose. This stops being effective after about 3 -v switches. -d is the debugging mode, and probably isn't necessary for everyday scans, but it helps untold amounts in the debugging process.

-iR

Picks random IP addresses to scan. nmap recently endured a massive overhaul to ensure that these random scans don't scan private (non-routable) networks, or government computers who probably won't take to kindly to your portscan. This switch is of limited usefulness, but may by valuable for statistical analysis or seeing how a outward-destined packet filtering firewall will filter a random array of IPs.

-M <sockets>

This useful option allows you to specify how many sockets you would like to limit nmap to use for scanning. See, nmap will scan multiple ports at once, and for a normal -sT scan, it must go through the BSD sockets interface to use the network. Needless to say, any other type of scan (except possibly -sU) is unaffected by this switch, as it doesn't use the sockets interface at all: It just directly injects packets onto the network, bypassing the operating system altogether. This is why root is required. This switch is especially useful on BSD machines, I've noticed. Some of them (especially on older hardware) seem to jam up and lag a bit while preforming -sT scans, although I've never seen one crash before. Listen to the man page, though: Use -sS over -sT whenever possible.

-o<logging method>

Generally, nmap will output its results in human readable form to stdout. Usually, when I want to save my logs, I'll do this:

nmap -sS target.com | tee /home/doug/scans/target.com-sS

This works just fine for me, but people seem to like using this switch to save output elsewhere. See the man page for details. One useful feature that I haven't experimented much with is the --resume switch that will let you resume a canceled nmap scan. Again, refer to the man page.

Practical Scanning

```
------------------

Many novice scanners don't recognize the importance of cataloging your scans. If you simply
port scan to standard output, and read the results, you will often forget your results
and be forced to repreform the scan, which is not stealthy in the least. Create a directory
for storing your scans, and name them appropriatley, with all the switches in the filename,
so as to avoid any confusion.

Always carefully consider what information you are actually after BEFORE you start scanning.
Nothing will give you away faster than a bunch of blind, thoughtless scans. This is what
most IDSs are designed to detect, afterall. For instance, if you want to find the operating
system of a webserver NOT running SSL, use this scan:

nmap -sS -P0 -O -p 80,443 www.target.com

The logic behind this scan is left as an exercise to the reader.

Always keep in mind that anything you do can, and often is, logged.

The only real way to become an expert portscanner is practise, practise, and more practise.
You'll see a lot of strange things if you scan enough computers, and often it is interesting
and educational to discover the causes of these anomalies. Scan your own machines,
scan your friends machines, and you will learn a lot about scanning and networking in
general.

Scanning with NEW Scripts
-----------------------------

Many hackers now dont have the time to search in google for the right exploit that exploits the service
he wants.Now with the help of the L1on at the https://forum.intern0t.org/hacking-tools-utilities/3553-
exploitdb-nse-nmap-script.html
he says "This is just a little nmap script I wrote. It searches the exploitdb archive for possible
exploits.
It is very verbose and can give you false positives, but I like it that way, and there was no other good
way I could think of to not skip over a possible exploit.
Currently it is set to use the archive on BT, if you would like to use it on something else just download
the archive and change the file location"
This tool can be only used in bactrack but with some fixes in directories you will find yourself using this
tool to all linux systems.
The script is:

description = [[Searches for exploits in the exploitdb on Backtrack. This archive can also be found at
http://www.exploitdb.com]]
author = "L10n"
license = "Same as Nmap--See http://nmap.org/book/man-legal.html"
categories = {"safe", "vuln"}

require("stdnse")
```

```
portrule = function(host, port)
    return port.state == "open"
end

action = function(host, port)
    local n = port.version.product
    local exploits = ""
    for line in io.lines ("/pentest/exploits/exploitdb/files.csv") do
         if string.match(line, n) and string.match(line, "remote") then
             local items = split(line, ",")
             local file  = items[2]
             local desc  = items[3]
             exploits    = exploits..file.." ---> "..desc.."\n"
         end
    end
    if not string.match(exploits, "\n") then
        exploits = nil
    end
    exploits = " \n"..exploits
    return exploits
end

function split(str, pat)
    local t = {}  -- NOTE: use {n = 0} in Lua-5.0
    local fpat = "(.-)" .. pat
    local last_end = 1
    local s, e, cap = str:find(fpat, 1)
    while s do
        if s ~= 1 or cap ~= "" then
            table.insert(t,cap)
        end
        last_end = e+1
        s, e, cap = str:find(fpat, last_end)
    end
    if last_end <= #str then
        cap = str:sub(last_end)
        table.insert(t, cap)
    end
    return t
end
```

To use this tool in linux systems you should change the path of io.lines that for the program is the line 647 here in my guide.
You set where the files.csv where it is and you scan the target with the syntax of:

```
nmap -sV --script=exploitdb.nse scanme.nmap.org
Starting Nmap 5.21 ( http://nmap.org ) at 2010-12-19 04:25 PST
NSE: Script Scanning completed.
```

```
Nmap scan report for scanme.nmap.org (64.13.134.52)
Host is up (0.071s latency).
Not shown: 995 filtered ports
PORT    STATE  SERVICE VERSION
22/tcp  open   ssh     OpenSSH 4.3 (protocol 2.0)
| exploitdb:
| platforms/linux/remote/25.c ---> "OpenSSH/PAM <= 3.6.1p1 Remote Users Discovery Tool"
| platforms/linux/remote/26.sh ---> "OpenSSH/PAM <= 3.6.1p1 Remote Users Ident (gossh.sh)"
| platforms/multiple/remote/3303.sh ---> "Portable OpenSSH <= 3.6.1p-PAM / 4.1-SUSE Timing Attack Exploit"
| platforms/multiple/remote/3303.sh ---> "Portable OpenSSH <= 3.6.1p-PAM / 4.1-SUSE Timing Attack Exploit"
|_platforms/linux/remote/6094.txt ---> "Debian OpenSSH Remote SELinux Privilege Elevation Exploit (auth)"
25/tcp  closed smtp
53/tcp  open   domain
80/tcp  open   http    Apache httpd 2.2.3 ((CentOS))
113/tcp closed auth

Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 37.84 seconds

Scanning hosts with perl script made by cypherround in https://forum.intern0t.org/hacking-tools-
utilities/3283-perl-script-simplify-nmap.html
This perl script automatize the work of the simple.
Here you just add the target and the port and you get what you want.
Here is the Script:
#! /usr/bin/perl

#################################################################################################
# This script was created by cypherround
# The purpose of the script is to simplify my favorite nmap scans
# without having to input the parameters of the specific scan.
# Feel free to use this scan for yourself and if you would like me
# to add any other types of scans let me know and I will input them
# into the script. Thanks happy hacking!
#################################################################################################

use strict;
use warnings;

print "\nScript created by cypherround\n\n";

my @a;         #aggressive scan
my @ao;        #aggressive scan with output to file
my @aa;        #aggressive scan of all 65535 ports
my @aao;    #aggressive scan of all 65535 ports with output
my @ap;        #aggressive scan with specific port
my @apo;    #aggressive scan with specific port and output
my @os;        #os fingerprint scan
my @oso;    #os fingerprint scan with output
```

```
my @oss;    #os fingerprint service scan
my @osso;    #os fingerprint service scan with output
my @osp;    #os fingerprint scan with specific port
my @ospo;    #os fingerprint scan with specific port and output
my @osps;    #os fingerprint service scan with specific port
my @ospso;    #os fingerprint service scan with specific port and output
my @r;        #random IP scan
my @ro;        #random IP scan with output to file
my @ra;        #random IP scan of all 65535 ports
my @rao;    #random IP scan of all 65535 ports with output
my @rp;        #random IP scan with specific port
my @rpo;    #random IP scan with specific port and output
my @s;        #stealth scan
my @so;        #stealth scan with output to file
my @sa;        #stealth scan of all 65535 ports
my @sao;    #stealth scan of all 65535 ports with output
my @sp;        #stealth scan with specific port
my @spo;    #stealth scan with specific port and output
my @sss;    #stealth service scan
my @ssso;    #stealth service scan with output to file
my @sssa;    #stealth service scan of all 65535 ports
my @sssao;    #stealth service scan of all 65535 ports with output
my @sssp;    #stealth service scan with specific port
my @ssspo;    #stealth service scan with specific port and output
my @u;        #udp scan
my @uo;        #udp scan with output to file
my @ua;        #udp scan of all 65535 ports
my @uao;    #udp scan of all 65535 ports with output
my @up;        #udp scan with specific port
my @upo;    #udp scan with specific port and output
my @ut;        #udp & tcp scan
my @uto;    #udp & tcp scan with output to file
my @uta;    #udp & tcp scan of all 65535 ports
my @utao;    #udo & tcp scan of all 65535 ports with output
my @utp;    #udp & tcp scan with specific port
my @utpo;    #udp & tcp scan with specific port and output
my $ip;        #specified ip
my $output;    #specified output file
my $port;    #specified port
my $random;    #specifide amount of random IPs
my $scan;    #specified type of scan to use

print "Here are your options for your nmap scan (enter the abbreviation in parenthesis):\n\n
aggressive(a)\n aggresive with output(ao)\n aggressive 65535 ports(aa)\n aggressive 65535 ports with
output(aao)\n aggressive with specific port(ap)\n aggressive specific port and output (apo)\n os
fingerprint scan(os)\n os fingerprint scan with output(oso)\n os fingerprint service scan(oss)\n os
fingerprint service scan with output(osso)\n os fingerprint scan with specific port(osp)\n os fingerprint
scan with specific port and output(ospo)\n os fingerprint service scan with specific port(osps)\n os
```

```
fingerprint scan with specific port and output(ospso)\n random IPs(r)\n random IPs with output(ro)\n random
IPs all 65535 ports(ra)\n random IPs all 65535 ports with output(rao)\n random IPs with specific port(rp)\n
random IPs with specific port and output(rpo)\n stealth(s)\n stealth with output(so)\n stealth scan of all
65535 ports(sa)\n stealth scan with all 65535 ports with output(sao)\n stealth scan with specific
port(sp)\n stealth scan with specfifi port and output(spo)\n stealth service scan(sss)\n stealth serv
scan with output(ssso)\n stealth service scan of all 65535 ports(sssao)\n stealth service scan of all
ports with output(sssao)\n stealth service scan with specific port(sssp)\n stealth service scan with
specific port and output(ssspo)\n udp(u)\n udp with output(uo)\n udp of all 65535 ports(ua)\n udp of all
65535 ports with output(uao)\n udp with specifc port(up)\n udp with specific port and output(upo)\n
udp/tcp(ut)\n udp/tcp with output(uto)\n udp/tcp of all 65535 ports(uta)\n udp/tcp if all 65535 ports with
output(utao)\n udp/tcp with specific port(utp)\n udp/tcp with specific port and output(utpo)\n" ;
$scan=<STDIN>;
chomp($scan);

if ($scan eq "r" or $scan eq "ro" or $scan eq "ra" or $scan eq "rao" or $scan eq "rp" or $scan eq "rpo") {
    print "How many random IPs would you like to scan? \n";
    $random=<STDIN>;
    chomp($random);
}

elsif ($scan eq "ao" or $scan eq "aao" or $scan eq "apo" or $scan eq "oso" or $scan eq "osso" or $scan eq
"ospo" or $scan eq "ospso" or $scan eq "ro" or $scan eq "rao" or $scan eq "rpo" or $scan eq "so" or $scan
eq "sao" or $scan eq "spo" or $scan eq "ssso" or $scan eq "sssao" or $scan eq "ssspo" or $scan eq "uo" or
$scan eq "uao" or $scan eq "upo" or $scan eq "uto" or $scan eq "utao" or $scan eq "utpo") {
    print "What would you like your file to be named? add .txt to the filenname (ex: aggressive.txt) \n";
    $output=<STDIN>;
    chomp($output);
}

elsif ($scan eq "a" or $scan eq "ao" or $scan eq "aa" or $scan eq "aao" or $scan eq "ap" or $scan eq "apo"
or $scan eq "os" or $scan eq "oso" or $scan eq "oss" or $scan eq "osso" or $scan eq "osp" or $scan eq
"ospo" or $scan eq "osps" or $scan eq "ospso" or $scan eq "s" or $scan eq "so" or $scan eq "sa" or $scan eq
"sao" or $scan eq "sp" or $scan eq "spo" or $scan eq "sss" or $scan eq "ssso" or $scan eq "sssa" or $scan
eq "sssao" or $scan eq "sssp" or $scan eq "ssspo" or $scan eq "u" or $scan eq "uo" or $scan eq "ua" or
$scan eq "uao" or $scan eq "up" or $scan eq "upo" or $scan eq "ut" or $scan eq "uto" or $scan eq "uta" or
$scan eq "utao" or $scan eq "utp" or $scan eq "utpo") {
    print "Enter the IP you are searching for: \n";
    $ip=<STDIN>;
    chomp($ip);
}

if ($scan eq "ap" or $scan eq "apo" or $scan eq "osp" or $scan eq "ospo" or $scan eq "osps" or $scan eq
"ospso" or $scan eq "rp" or $scan eq "rpo" or $scan eq "up" or $scan eq "upo" or $scan eq "utp" or $scan eq
"utpo" or $scan eq "sp" or $scan eq "spo" or $scan eq "sssp" or $scan eq "ssspo") {
    print "Enter the ports you would like to scan: (ex: 21,22,80,443)\n";
    $port=<STDIN>;
    chomp($port);
}
```

```
if ($scan eq "a") {
    @a = `nmap -v -A $ip`;
        print "@a\n";
}

if ($scan eq "ao") {
    @ao = `nmap -v -A $ip -oG $output`;
        print "@ao\n";
}

if ($scan eq "aa") {
    @aa = `nmap -v -A $ip -p-`;
        print "@aa\n";
}

if ($scan eq "aao") {
    @aao = `nmap -v -A $ip -p- -oG $output`;
        print "@aao\n";
}

if ($scan eq "ap") {
    @ap = `nmap -v -A $ip -p $port`;
        print "@ap\n";
}

if ($scan eq "apo") {
    @apo = `nmap -v -A $ip -p $port -oG $output`;
        print "@apo\n";
}

if ($scan eq "os") {
    @os = `sudo nmap -v -O $ip`;
        print "@os\n";
}

if ($scan eq "oso") {
    @oso = `sudo nmap -v -O $ip -oG $output`;
        print "@oso\n";
}

if ($scan eq "oss") {
    @oss = `sudo nmap -v -O -sV $ip`;
        print "@oss\n";
}

if ($scan eq "osso") {
    @osso = `sudo nmap -v -O -sV $ip -oG $output`;
```

```
        print "@osso\n";
}

if ($scan eq "osp") {
    @osp = `sudo nmap -v -O $ip -p $port`;
        print "@osp\n";
}

if ($scan eq "ospo") {
    @ospo = `sudo nmap -v -O $ip -p $port -oG $output`;
        print "@ospo\n";
}

if ($scan eq "osps") {
    @osps = `sudo nmap -v -O -sV $ip -p $port`;
        print "@osps\n";
}

if ($scan eq "ospso") {
    @ospso = `sudo nmap -v -O -sV $ip -p $port -oG $output`;
        print "@ospso\n";
}

if ($scan eq "r") {
    @r = `nmap -v -iR $random -PN`;
        print "@r\n";
}

if ($scan eq "ro") {
    @ro = `nmap -v -iR $random -PN -oG $output`;
        print "@ro";
}

if ($scan eq "ra") {
    @ra = `nmap -v -iR $random -PN -p-`;
        print "@ra\n";
}

if ($scan eq "rao") {
    @rao = `nmap -v -iR $random -PN -p- -oG $output`;
        print "@rao\n";
}

if ($scan eq "rp") {
    @rp = `nmap -v -iR $random -PN -p $port`;
        print "@rp\n";
}
```

```
if ($scan eq "rpo") {
    @rpo = `nmap -v -iR $random -PN -p $port -oG $output`;
        print "@rpo\n";
}


if ($scan eq "s") {
    @s = `nmap -v -v -PN $ip`;
        print "@s\n";
}

if ($scan eq "so") {
    @so = `nmap -v -PN $ip -oG $output`;
        print "@so\n";
}

if ($scan eq "sa") {
    @sa = `nmap -v -PN $ip -p-`;
        print "@sa\n";
}

if ($scan eq "sao") {
    @sao = `nmap -v -PN $ip -p- -oG $output`;
        print "@sao\n";
}

if ($scan eq "sp") {
    @sp = `nmap -v -PN $ip -p $port`;
        print "@sp\n";
}

if ($scan eq "spo") {
    @spo = `nmap -v -PN $ip -p $port -oG $output`;
        print "@spo\n";
}

if ($scan eq "sss") {
    @sss = `nmap -v -v -sV -PN $ip`;
        print "@sss\n";
}

if ($scan eq "ssso") {
    @ssso = `nmap -v -sV -PN $ip -oG $output`;
        print "@ssso\n";
}

if ($scan eq "sssa") {
    @sssa = `nmap -v -sV -PN $ip -p-`;
```

```
        print "@sssa\n";
}

if ($scan eq "sssao") {
    @sssao = `nmap -v -sV -PN $ip -p- -oG $output`;
        print "@sssao\n";
}

if ($scan eq "sssp") {
    @sssp = `nmap -v -sV -PN $ip -p $port`;
        print "@sssp\n";
}

if ($scan eq "ssspo") {
    @ssspo = `nmap -v -sV -PN $ip -p $port -oG $output`;
        print "@ssspo\n";
}

if ($scan eq "u") {
    @u = `sudo nmap -v -sU $ip`;
        print "@u\n";
}

if ($scan eq "uo") {
    @uo = `sudo nmap -v -sU $ip -oG $output`;
        print "@uo\n";
}

if ($scan eq "ua") {
    @ua = `sudo nmap -v -sU $ip -p-`;
        print "@ua\n";
}

if ($scan eq "uao") {
    @uao = `sudo nmap -v -sU $ip -p- -oG $output`;
        print "@uao\n";
}

if ($scan eq "up") {
    @up = `sudo nmap -v -sU $ip -p $port`;
        print "@up\n";
}

if ($scan eq "upo") {
    @upo = `sudo nmap -v -sU $ip -p $port -oG $output`;
        print "@upo\n";
}
```

```
if ($scan eq "ut") {
    @ut = `sudo nmap -v -sU -sS $ip`;
        print "@ut\n";
}

if ($scan eq "uto") {
    @uto = `sudo nmap -v -sU -sS $ip -oG $output`;
        print "@uto\n";
}

if ($scan eq "uta") {
    @uta = `sudo nmap -v -sU -sS $ip -p-`;
        print "@uta\n";
}

if ($scan eq "utao") {
    @utao = `sudo nmap -v -sU -sS $ip -p- -oG $output`;
        print "@utao\n";
}

if ($scan eq "utp") {
    @utp = `sudo nmap -v -sU -sS $ip -p $port`;
        print "@utp\n";
}

if ($scan eq "utpo") {
    @utpo = `sudo nmap -v -sU -sS $ip -p $port -oG $output`;
        print "@utpo\n";
}
```

Save it and chmod u+x the perl script to make it executable.

Cheat Sheet

Just founded out a complete reference to nmap in pdf version.This cheat sheet has almost the most used commands by pentesters
and sysadmins to secure the perimeter , there are a lots of command to get into the world of nmap scanner. So i would suggest to learn very good this tool becauze the part of scanning is the part where you have to find vuln
of an target to exploit it.
The cheat sheet is in this url:
https://scadahacker.com/library/Documents/Cheat_Sheets/Hacking%20-%20NMap%20Quick%20Reference%20Guide.pdf

Nmap in your hands

Everything on the Nmap command-line that isn't an option (or option argument) is treated as a target host specification. The simplest case is to specify a target IP address or hostname for scanning. Sometimes you wish to scan a whole network of adjacent hosts. For this, Nmap supports CIDR-style

addressing. You can append /numbits to an IP address or hostname and Nmap will scan every IP address for which the first numbits are the same as for the reference IP or hostname given. For example, 192.168.10.0/24 would scan the 256 hosts between 192.168.10.0 (binary: 11000000 10101000 00001010 00000000) and 192.168.10.255 (binary: 11000000 10101000 00001010 11111111), inclusive. 192.168.10.40/24 would do exactly the same thing. Given that the host scanme.nmap.org is at the IP address 205.217.153.62, the specification scanme.nmap.org/16 would scan the 65,536 IP addresses between 205.217.0.0 and 205.217.255.255. The smallest allowed value is /1, which scans half the Internet. The largest value is 32, which scans just the named host or IP address because all address bits are fixed.

CIDR notation is short but not always flexible enough. For example, you might want to scan 192.168.0.0/16 but skip any IPs ending with .0 or .255 because they are commonly broadcast addresses. Nmap supports this through octet range addressing. Rather than specify a normal IPaddress you can specify a comma separated list of numbers or ranges for each octet. For example, 192.168.0-255.1-254 will skip all addresses in the range that end in .0 and or .255. Ranges need not be limited to the final octects: the specifier 0-255.0-255.13.37 will perform an Internet-wide scan for all IP addresses ending in 13.37. This sort of broad sampling can be useful for Internet surveys and research.

IPv6 addresses can only be specified by their fully qualified IPv6 address or hostname. CIDR and octet ranges aren't supported for IPv6 because they are rarely useful.

Nmap accepts multiple host specifications on the command line, and they don't need to be the same type. The command nmap scanme.nmap.org 192.168.0.0/8 10.0.0,1,3-7.0-255 does what you would expect.

Decoys are used both in the initial ping scan (using ICMP, SYN, ACK, or whatever) and during the actual port scanning phase. Decoys are also used during remote OS detection (-O).

Decoys do not work with version detection or TCP connect() scan.

It is worth noting that using too many decoys may slow your scan and potentially even make it less accurate. Also, some ISPs will filter out your spoofed packets, but many do not restrict spoofed IP packets at all.

-S <IP_Address> (Spoof source address)

In some circumstances, Nmap may not be able to determine your source address (Nmap will tell you if this is the case). In this situation, use -S with the IP address of the interface you wish to send packets through.

Another possible use of this flag is to spoof the scan to make the targets think that someone else is scanning them. Imagine a company being repeatedly port scanned by a competitor!

The -e option would generally be required for this sort of usage, and -P0 would normally be advisable as well.

-e <interface> (Use specified interface)

Tells Nmap what interface to send and receive packets on. Nmap should be able to detect this automatically, but it will tell you if it cannot.

--source_port <portnumber>; -g <portnumber> (Spoof source port number)

One surprisingly common misconfiguration is to trust traffic based only on the source port number. It is easy to understand how this comes about. An administrator will set up a shiny new firewall, only to be flooded with complains from ungrateful users whose applications stopped working. In particular, DNS may be broken because the UDP DNS replies from external servers can no longer enter the network. FTP is another common example. In active FTP transfers, the remote server tries to establish a connection back to the client to transfer the requested file.

Secure solutions to these problems exist, often in the form of application-level proxies or protocol-parsing firewall modules. Unfortunately there are also easier, insecure solutions. Noting that DNS replies come from port 53 and active ftp from port 20, many admins have fallen into the trap of simply allowing incoming traffic from those ports. They often assume that no attacker would notice and exploit such firewall holes. In other cases, admins consider this a short-term stop-gap measure until they can implement a more secure solution. Then they forget the security upgrade.
Overworked network administrators are not the only ones to fall into this trap. Numerous products have shipped with these insecure rules. Even Microsoft has been guilty. The IPsec filters that shipped with Windows 2000 and Windows XP contain an implicit rule that allows all TCP or UDP traffic from port 88 (Kerberos). In another well-known case, versions of the Zone Alarm personal firewall up to 2.1.25 allowed any incoming UDP packets with the source port 53 (DNS) or 67 (DHCP).

Script Engine Scanning

Some Words from nmap WEBsite
The Nmap Scripting Engine (NSE) is one of Nmap's most powerful and flexible features. It allows users to write (and share) simple scripts to automate a wide variety of networking tasks. Those scripts are then executed in parallel with the speed and efficiency you expect from Nmap. Users can rely on the growing and diverse set of scripts distributed with Nmap, or write their own to meet custom needs.

We designed NSE to be versatile, with the following tasks in mind:

Network discovery
This is Nmap's bread and butter. Examples include looking up whois data based on the target domain, querying ARIN, RIPE, or APNIC for the target IP to determine ownership, performing identd lookups on open ports, SNMP queries, and listing available NFS/SMB/RPC shares and services.

More sophisticated version detection
The Nmap version detection system (Chapter 7, Service and Application Version Detection) is able to recognize thousands of different services through its probe and regular expression signature based matching system, but it cannot recognize everything. For example, identifying the Skype v2 service requires two independent probes, which version detection isn't flexible enough to handle.
Nmap could also recognize more SNMP services if it tried a few hundred different community names by brute force.
Neither of these tasks are well suited to traditional Nmap version detection, but both are easily accomplished with NSE. For these reasons, version detection now calls NSE by default to handle some tricky services. This is described in the section called “Version Detection Using NSE”.

Vulnerability detection
When a new vulnerability is discovered, you often want to scan your networks quickly to identify vulnerable systems before the bad guys do. While Nmap isn't a comprehensive vulnerability scanner, NSE is powerful enough to handle even demanding vulnerability checks. Many vulnerability detection scripts are already available and we plan to distribute more as they are written.

Backdoor detection

Many attackers and some automated worms leave backdoors to enable later reentry. Some of these can be detected by Nmap's regular expression based version detection. For example, within hours of the MyDoom worm hitting the Internet, Jay Moran posted an Nmap version detection probe and signature so that others c quickly scan their networks for MyDoom infections. NSE is needed to reliably detect more complex worm backdoors.

Vulnerability exploitation

As a general scripting language, NSE can even be used to exploit vulnerabilities rather than just find them. The capability to add custom exploit scripts may be valuable for some people (particularly penetration testers), though we aren't planning to turn Nmap into an exploitation framework such as Metasploit.

These listed items were our initial goals, and we expect Nmap users to come up with even more inventive uses for NSE.

Scripts are written in the embedded Lua programming language, version 5.2. The language itself is well documented in the books Programming in Lua, Second Edition and Lua 5.1 Reference Manual. The reference manual, updated for Lua 5.2, is also freely available online, as is the first edition of Programming in Lua. Given the availability of these excellent general Lua programming references, this document only covers aspects and extensions specific to Nmap's scripting engine.

NSE is activated with the -sC option (or --script if you wish to specify a custom set of scripts) and results are integrated into Nmap normal and XML output.

A typical script scan is shown in the Example 9.1. Service scripts producing output in this example are ssh-hostkey, which provides the system's RSA and DSA SSH keys, and rpcinfo, which queries portmapper to enumerate available services. The only host script producing output in this example is smb-os-discovery, which collects a variety of information from SMB servers. Nmap discovered all of this information in a third of a second.

NSE scripts are very powerful and have become one of Nmap's main strengths, performing tasks from advanced version detection to vulnerability exploitation.

Each NSE script belongs to a category based on what it does. Current categories are the following:

CATEGORY:

- auth: scripts that work with authentication credentials

- broadcast: scripts that discover active hosts by broadcasting on a local network and adding them to a target list

- brute: scripts that brute force the credentials of the remote service

- default: scripts that are automatically run with -sC or -A options

- discovery: scripts that try to acquire more information about the target network

- dos: scripts that may crash the target application and therefore cause a denial of service to the target

- exploit: scripts that may be able to exploit the target application

- external: scripts that send data to a third party server over the network (whois)

- fuzzer: scripts that send invalid random data to the target to find undiscovered bugs

- intrusive: scripts that can cause the target to fail

- malware: scripts that test whether the target is infected by malware or backdoors

- safe: scripts that can be run safely, so they will not crash a server

- version: scripts that can determine the version of the application running on a target (they are run only when -sV option is specified)

- vuln: scripts that can check whether the target is vulnerable to specific attacks

To include the title of the index document of a web server in your scan results, open your terminal and type the following command:

```
$ nmap -sV --script http-title
$ nmap --script http-headers,http-title
Run all the scripts in the vuln category:
$ nmap -sV --script vuln <target>
Run the scripts in the categories version or discovery:
$ nmap -sV --script="version,discovery" <target>
 Run all the scripts except for the ones in the exploit category:
$ nmap -sV --script "not exploit" <target>
Finding Geolocation of a IP
nmap --script ip-geolocation-* <target>
Whois
nmap --script WHOIS <target>
```

DNS records contains a lot of information about a particular domain which cannot be ignored.
Of course there are specific tools for brute forcing DNS records which can produce better results but the dns-brute script can perform also
this job in case that we want to extract DNS information during our Nmap scans.

```
nmap -p80 --script dns-brute <target>
```

Reversein IP

```
nmap --script http-reverse-ip <target>
```

Executing the Discovery Scripts

This category of scripts is ideal when we need to have as much information as possible for a specific target.
The next two images are a sample of what kind of information could be delivered to us when we run the Discovery Scripts.

```
nmap --script=discovery <target>                //discovery is a category
                goto CATEGORY;
I made those lines to tell that we can put every category in the finding bugs or vuln in a target host.
LAST:nmap –script-updatedb
```

Greetz to:
FYODOR:Nmap programmer
INFOSEC Institute
Str0ke(My friend who died, creator of milw0rm.com)
And All Who read this long paper
