Using Network Exploration and Service Detection Techniques in Stork-Data Placement Scheduler

Mehmet Balman  
Center for Computation & Technology  
Louisiana State University, Baton Rouge, LA  
balman@cct.lsu.edu

December, 2007

Introduction

Data transfer over a network connection is the most basic data placement type. Before initiating a data transfer in which source host will connect a file transfer service running on a remote server and transmit data over a network channel, it is important to get prior knowledge in order to decrease error detection time. In addition, it is also useful at the time of scheduling to know whether destination host and service is available or not; such that, a data transfer job which would fail because destination host or service is not reachable will not be processed until that error condition is recovered. Besides the advantage of prior error detection, information about active services in the target machine would help data scheduler discover and use alternative transfer protocol.

This study has two aspects:

- making Stork aware whether destination host/service is available,
- making Stork able to select suitable data transfer services (any-to-any).

This paper explains techniques in network scanning and discusses applicability of those mechanisms in data placement schedulers. First, network exploration and port scanning has been explained in detail. Then, information about Nmap [2], a network scanner tool, and experiments with network scanner implementations have been given. A preliminary network scanner is implemented. The details about integrating host and service detection functions to Stork [3] are explained and initial coding and testing have been provided.

Motivation

The following layers have been proposed in Phoenix paper [1] as basic structure to make data intensive applications fault tolerant:

- DNS resolve
- Host Alive
- Port Open
- Service Available
• Test service (transfer test data before starting the actual data placement)

Port Alive and Port Open steps can be combined since firewalls usually block Ping scans. Normally, only root privileged users have access to ICMP layer which is used to detect whether remote host is up or down (ping utility). On the other hand, network scanners like Nmap, are able to manage by directly accessing the Ethernet using specialized libraries and not using using the underlying network layer.

Another important fact is that, port scanners (e.g. Nmap) are basically designed not to be detected by Intrusion Detection Systems (IDS). Therefore, they use some tricks which are supposed to mislead firewalls or IDSs. This features lead to delays in port scan and service detection operations especially when machines with active firewalls are involved.

Thus, our first goal is to implement a service detection mechanism in which the following steps will be focused on:
  • DNS resolve
  • Port Open
  • Service Available (a simple test)

Moreover, an initial attempt to discover available transfer protocol can be implemented in Port Scan phase. A list of all available services would provide scheduler to make better selections.

**Network Exploration**

Network exploration is a commonly used methodology in system and network administration for network inventory, host and service monitoring, and especially for security audits. On a computer network with the fact that services are not advertising themselves by a service discovery protocol, it is very much valuable to discover computers available on the network, and to determine what services are running. Moreover, today’s security scanners are able gather lots of useful information from remote network computers; such that, network discover tools can determine name and version of offered services, operating system version of available hosts on the network, presence of firewalls, type of packet filtering methods, filtered ports, device types, and even vendor of network cards in local area networks [4].

We can basically classify network mapping into the following sub-categories [8]:
  • Host Discovery
  • Port Scanning
  • Version Detection
  • OS Detection

Host discovery is defined by detecting available hosts on a network. Most common approach is to use “ping” utility to determine whether a host is reachable or not across an IP network [6]. Ping, which has been first implemented by Mike Muuss[12], sends ICMP echo request packets and waits for a corresponding ICMP echo response from target
computer, then calculates the round-trip time. Although, it is useful in diagnosing networks, ICMP echo request (Type 8) packets are usually filtered out by many ISPs due to the fact that ping has quietly been used to leak information by attackers and also it has created unnecessary load on network for routers across the Internet.

Operating system detection is accomplished by TCP/IP stack fingerprints [9]. A combination of data is sent to the system in order to discover its version according to the response. TCP/IP flag settings may vary from one stack implementation to another and systems usually response same with correct data but not always respond same way with wrong data.

In service detection, we first connect once an open port found and wait for the initial welcome banner because common services like FTP, SSH, SMTP, telnet, POP3, IMAP identify themselves [13]. If port is on a secure channel, we connect with SSL and scan the service running behind it. Basically, service scanners probe the port and compare returned data in order to determine the program using that port.

**Port Scanning**

Port scanning is one basic technique used by system administrators to find out open ports in order to check the security of the computer. There are 65535 port numbers in TCP/IP, and they have been categorized in three ranges [14]:

- Well known ports (0-1023)
- Registered Ports (1024-49151)
- Dynamic/Private Ports (49152-65535)

A port scan to a specific port results in generally three states [6]. If host sends back a reply, it shows port is open and a there is a service listening on that port. If reply is stating that the connection will be denied, port status is closed and there is no service listening on that port. The third port status is filtered or blocked in which no reply from the target host is sent back. Port scanning is also used to compromise security of the system. Gathering information about list of open and closed ports would be used by an attacker to exploit vulnerabilities of the service and the operating system running on the host. Therefore, IDPSs provides detection tool for port scanners; besides, transparent ports and packet filters prevent port scanners to reach actual port.

Port scanning has been classified as portscan, which searches a single host for open ports, and portsweep, which searches multiple hosts on a network to find a specific open port [7]. The simplest port scanning technique is TCP scan in which network function of the underlying operating system is used such that user does not require special privileges. In TCP scan, connect() system call returns success if port is open and listening, otherwise port is not accessible. However, this method will easily will be detected and logged by the system. Therefore, there are also many other techniques to proceed not be detected by auditing utilities.
Port scanning methodology has been classified as follows [5]:

- TCP connect() scanning
- TCP SYN (half-open) scanning
- TCP FIN (stealth) scanning
- TCP ftp proxy (bounce attach scanning)
- SNY/FIN (fragmented IP packets) scanning
- UDP recvfrom() scanning
- UDP raw ICMP port unreachable scanning
- ICMP scanning (ping-sweep)
- Reverse-ident scanning

In TCP SYN-scan, which is usually known as half-open scan, we do not rely on provided network functions such that port scanner send a raw IP packet and gets the response. Instead of using TCP three-way handshake protocol, a SYN packet is sent pretending that a real connection will be opened [5,6]. If port is open and listening, a SIN|ACK packet would be received. Otherwise, RST packet will be sent back from target host. When port scanner receives a SIN|ACK packet and determines port is open, it immediately closes the connection by sending a RST packet back. The advantage of half-open scan is that this access to the port is not logged by many systems.

Using raw IP packet gives full control in TCP stack, but root privileges are required to prepare custom IP packets. On the other hand, sophisticated scanning tools provide specialized network libraries and novel ways to use of raw IP packets [4].

Another stealth scanning technique is sending a FIN packet and attempting to close a connection which is not open. It is expected that operating system will generate an error and will reply back an RST packet if the target port is closed. If port is open, the sent packet will just be ignored and no response to the scanner indicates there is a service listening on that port. However, operating system may not behave as expected and reply back always with RST packets to FIN requests [5,6].

In fragmented packet scanning, TCP header is divided into several IP fragments [5,6]. Since packet filters do not usually queue all IP fragments, port scanner can bypass the firewall. In ftp bounce attack scanning, port scanner take advantage of the vulnerability of ftp server which supports proxy ftp connections. Ftp servers accept a request which ask the server to open a data connection to a third party host on a given port. Generated response code will help port scanner to identify the port and hide where scan attach is coming from.

In UDP scan, it is expected that system will response with ICMP port unreachable message if port is not open [5]. Non-privileged users cannot read unreachable error message, but operating system can indirectly inform the user
and scanner can determine port availability according to the difference in returned error messages. ICMP protocol
does not have a port abstraction. ICMP echo packets are used to determine host availability. In ident scan, we take
advantage of the ident service which gives information about the user that owns the service running on a specific
port, so scanner can determine whether port is open and active [5].

Nmap

Nmap is an open source tool for network mapping and security scanning. It uses raw IP packages in a “novel way”
for network exploration and service detection [4]. Nmap was first published in 1997 by Fyodor [11] and many new
features such as better detection algorithms, new scan types and supported protocols have been developed in the
following versions [10].

Besides dozens of useful characteristics, Nmap provides XML output format to be easily interpreted. A detailed
summary of Nmap options are given in Appendix C-1. Some modules and projects utilizing nmap:

- Nmap::Scanner, Perl Module [19]
- Nmap-Parse, Perl Module [20]
- NmapFE, official Nmap GUI [2]
- Nmap-web: Web interface to nmap [18]
- Nmap-cgi project [15]
- LOCALSCAN: a front-end for Nmap [16]
- NmapWIN: Windows front-end for Nmap [17]

A typical Nmap scan is shown below:

[balan@is ~]$ nmap -A -p1-10000 -d gridhub.cct.lsu.edu

Starting Nmap 4.03 ( http://www.insecure.org/nmap/ ) at 2007-11-02 10:31 CDT
Machine 130.39.128.148 MIGHT actually be listening on probe port 80
Hostupdate called for machine 130.39.128.148 state UNKNOWN/COMBO -> HOST_UP (trynum 0, dotimeadj:
yes time: 335)
Finished block: srtt: 376 rttvar: 5000 timeout: 100000 block_tries: 1 up_this_block: 1
down_this_block: 0 group_sz: 1
massping done: num_hosts: 1 num_responses: 1
mass_rdns: Using DNS server 130.39.3.5
mass_rdns: Using DNS server 130.39.254.30
mass_rdns: Using DNS server 130.39.244.30
mass_rdns: Using DNS server 130.39.254.33
mass_rdns: 0.00s 0/1 [#: 4, OK: 0, NX: 0, DR: 0, SF: 0, TR: 1]
DNS resolution of 1 IPs took 0.00s. Mode: Async (#: 4, OK: 1, NX: 0, DR: 0, SF: 0, TR: 1, CN: 0)
Initiating Connect() Scan against gridhub.cct.lsu.edu (130.39.128.148) [10000 ports] at 10:31
Discovered open port 80/tcp on 130.39.128.148
Discovered open port 443/tcp on 130.39.128.148
Discovered open port 22/tcp on 130.39.128.148
Discovered open port 3306/tcp on 130.39.128.148
Discovered open port 2119/tcp on 130.39.128.148
Discovered open port 111/tcp on 130.39.128.148
Discovered open port 2222/tcp on 130.39.128.148
Discovered open port 2135/tcp on 130.39.128.148
Discovered open port 2811/tcp on 130.39.128.148
Discovered open port 7512/tcp on 130.39.128.148
Discovered open port 3570/tcp on 130.39.128.148

The Connect() Scan took 0.39s to scan 10000 total ports.

Initiating service scan against 12 services on gridhub.cct.lsu.edu (130.39.128.148) at 10:31

Got nsock CONNECT response with status ERROR - aborting this service

The service scan took 38.51s to scan 12 services on 1 host.

Starting RPC scan against gridhub.cct.lsu.edu (130.39.128.148) at 10:31

Lamer on port 111 closed RPC socket on me in get_rpc_results

The RPCGrind Scan took 0.01s to scan 1 ports on gridhub.cct.lsu.edu (130.39.128.148).

Interesting ports on gridhub.cct.lsu.edu (130.39.128.148):

<table>
<thead>
<tr>
<th>PORT</th>
<th>STATE</th>
<th>SERVICE</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>22/tcp</td>
<td>open</td>
<td>ssh</td>
<td>OpenSSH 3.6.1p2 (protocol 2.0)</td>
</tr>
<tr>
<td>80/tcp</td>
<td>open</td>
<td>http</td>
<td>Apache httpd 2.0.46 (Apache)</td>
</tr>
<tr>
<td>111/tcp</td>
<td>open</td>
<td>rpc</td>
<td></td>
</tr>
<tr>
<td>443/tcp</td>
<td>open</td>
<td>ssl/http</td>
<td>Apache httpd 2.0.46 (Apache)</td>
</tr>
<tr>
<td>2119/tcp</td>
<td>open</td>
<td>ssl/unknown</td>
<td></td>
</tr>
<tr>
<td>2222/tcp</td>
<td>open</td>
<td>ssh</td>
<td>OpenSSH 3.9p1 NCSE_GSSAPI_3.5 GSI (protocol 1.99)</td>
</tr>
<tr>
<td>2135/tcp</td>
<td>open</td>
<td>ldap</td>
<td>(Anonymous bind OK)</td>
</tr>
<tr>
<td>3306/tcp</td>
<td>open</td>
<td>mysql</td>
<td>MySQL 5.0.27-standard-log</td>
</tr>
<tr>
<td>3570/tcp</td>
<td>open</td>
<td>ssl/unknown</td>
<td></td>
</tr>
<tr>
<td>5433/tcp</td>
<td>open</td>
<td>postgresql</td>
<td></td>
</tr>
</tbody>
</table>

Nmap is a very powerful tool and it is extremely fast. However, it relies on the expected behavior of network services such that Nmap performance was poor like other simpler scanners when target host was a well configured computer for security:

[balman@gridhub balman]$ nmap dsl-turtle06.csc.lsu.edu
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Note: Host seems down. If it is really up, but blocking our ping probes, try -P0
Nmap run completed -- 1 IP address (0 hosts up) scanned in 1 second

[balman@gridhub balman]$ nmap -P0 dsl-turtle06.csc.lsu.edu
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )

[balman@gridhub balman]$ nmap dsl-condor.csc.lsu.edu
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )

Interesting ports on dsl-condor.csc.lsu.edu (130.39.225.155):

<table>
<thead>
<tr>
<th>PORT</th>
<th>STATE</th>
<th>SERVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Practice in Port Scanning

We have scanned through a simple port scanner [21] implementation which supports TCP connect() scan and SYN scan. In SYN scan, root privileges are required in order to use raw IP packets. For some test results refer to Appendix C-2 and for the relevant code refer to Appendix C-3.

Also, we have implemented a simple port scanner (called zport) using TCP connect scanning technique. Here are some test results:

```
$ ./zport virtdev1
starting TCP connect() scan ...
  can not resolve host !

$ ./zport
  port 22 (ssh) open
  port 111 (sunrpc) open
  port 32768 (filenet-tms) open

$./zport fake
starting TCP connect() scan ...
  port 1 closed!
  port 2 closed!
```

Our initial effort to write a network exploration function, zport source code, is given in Appendix B-1.

We have also integrated host exploration feature with Stork – given in Appendix A-1. Here are some test results with stork.transfer.globus-url-copy:

```
$./stork.transfer.globus-url-copy ftp://virtdev/tmp/a gsiftp://virtdev/tmp/b
  Transfering from: srb://virtdev/tmp/a to: gsiftp://virtdev/tmp/b with arguments:
  $cat out.7474
  Network error: destination port virtdev:2811 is not open !

$./stork.transfer.globus-url-copy ftp://virtdev/tmp/a gsiftp://virtdev/tmp/b
  Transfering from: ftp://virtdev/tmp/a to: gsiftp://virtdev/tmp/b with arguments:
  $cat out.7478
  Network error: can not resolve destination host - virtdev !

$./stork.transfer.globus-url-copy srb://virtdev/tmp/a gsiftp://virtdev/tmp/b
  Transfering from: srb://virtdev/tmp/a to: gsiftp://virtdev/tmp/b with arguments:
  can not determine port number - protocol srb !
  $cat out.7485
  Network error: destination port virtdev:2811 is not open !

$ ./stork.transfer.globus-url-copy file://virttest/tmp/a gsiftp://virtdev/tmp/b
  Transfering from: file://virttest/tmp/a to: gsiftp://virtdev/tmp/b with arguments:
  $cat out.7585
  Network error: destination port virtdev:2811 is not open !

$ ./stork.transfer.globus-url-copy file://virttest/tmp/a ftp://virtdev/tmp/b
  $cat out.7595
  GLOBUS error: sh: line 1: globus-url-copy: command not found
```
**Practice with Nmap**

We have implemented a special function (nmap_network_status_checkHost), to be called inside Stork, which is capable of using Nmap techniques to resolve host name and to explore given hosts also return available services. This utility (myNmap), which is basically calling Nmap 4.20 objects, is dependent on many external Nmap network libraries.

While compiling Stork with the new nmap_network_status.o object, we got the following error:

Error message: conlict in nbase library with condor_util_lib.

```plaintext
++ Error message: conlict in nbase library with condor_util_lib

../condor_util_lib/util_lib.a(get_random_num.o)(.text+0x36): In function `get_random_int':
/condor_CVS/Stork_CCT/Condor/src/condor_util_lib/get_random_num.c:53:
multiple definition of `get_random_int'
/condor_CVS/Stork_CCT/Condor/externals/bundles/nmap/nbase/libnbase.a(nbase_rnd.o)(.text+0x1a8):
/usr/bin/ld: Warning: size of symbol `get_random_int' changed from 22 in /condor_CVS/Stork_CCT/Condor/src/condor_util_lib/get_random_num.c:53:
to 50 in (6y
../condor_util_lib/util_lib.a(get_random_num.o)(.text+0x1a8):/condor_CVS/Stork_CCT/Condor/externals/bundles/nmap/nbase/libnbase.a(nbase_rnd.o)
to 50 in (6y
.

../condor_util_lib/util_lib.a(get_random_num.o)(.text+0x1a8): In function `get_random_uint':
/condor_CVS/Stork_CCT/Condor/src/condor_util_lib/get_random_num.c:83:
multiple definition of `get_random_uint'

Moreover, we have realized that some critical error messages (if a problem has been encountered) are generated and printed to stdout inside related cc files using fatal function. Since nmap is designed as a command line tool, we cannot guarantee that nmap_network_status_module will catch all fatal errors.

We have looked at the first implemented code of Nmap. After that, there has been many enhancements in OS version scanning, protocol version detection, fingerprints for OS and protocol exploration, etc. Basically, latest nmap (version 4) that we have been working on does not use the underlying network layer in the operating system. That is the reason why nmap is so fast and efficient.

On the other hand, parsing XML output (with -oX output_file) option looks like an easier way of implementation, and also Stork would be able to use all features of a port scanner.

For the source code of nmap_network_status.cc, refer to Appendix B-2.

The initial effort to integrate nmap_network_status.o object to Stork data transfer jobs:

```plaintext
++ from: dap_transfer_globus-url-copy.C

status = nmap_network_status_checkHost(src_host, dest_host, error_str);

//-- output to a temporary file....
if (status != DAP_SUCCESS){
  unsigned int mypid;
  FILE *f;
  char fname[MAXSTR];
  mypid = getpid();
```
snprintf(fname, MAXSTR, "out.%d", mypid);

f = safe_fopen_wrapper(fname, "w");
fprintf(f, "Network error: %s", error_str);
fclose(f);
return DAP_ERROR;
}

References

16. LOCALSCAN: A front-end to NmapP. http://staff.washington.edu/dgreene/localscan/
Appendix A-1   [ modification in the source code of dap_transfer_globus-url-copy ]

Makefile:

+ UTIL_LIB_OBJS = ../condor_c++_util/condor_open.o dap_utility.o dap_client_interface.o zport.o
+ SERVER_OBJS = zport.o dap_server.o dap_daemon.o dap_scheduler.o dap_logger.o
  dap_classad_reader.o $(STORK_UTIL)

dap_transfer_globus-url-copy.C:

+ #include "zport.h"

parse_url(src_url, src_protocol, src_host, src_file);
parse_url(dest_url, dest_protocol, dest_host, dest_file);

f printf(stdout, "Transfering from: %s to: %s with arguments: %s\n", src_url, dest_url, arguments);

// -MB 12/12/07
// NMAP
// status = nmap_network_status_checkHost(src_host, dest_host, error_str);
// ZPORT
// status = zport_network_status_checkPort(src_host, src_protocol, dest_host, dest_protocol, error_str);
  status = checkPort(src_host, src_protocol, dest_host, dest_protocol, error_str);

// -- output to a temporary file....
if (status != DAP_SUCCESS){
  unsigned int mypid;
  FILE *f;
  char fname[MAXSTR];
  mypid = getpid();
  snprintf(fname, MAXSTR, "out.%d", mypid);
  f = safe_fopen_wrapper(fname, "w");
  fprintf(f, "Network error: %s", error_str);
  fclose(f);
  return DAP_ERROR;
}

// -MB 12/12/07

status = transfer_globus_url_copy(src_url, dest_url, arguments, error_str);
### Appendix B-1 [ source code for zport ]

+++ Makefile:
CC=gcc
CFLAGS=-Wall -O2

zport: main.c
main.c: zport.o
clean:	rm -rf zport zport.o

+++ zport.h:
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#define __Z_TCP_CONNECT  0
#define __Z_TCP_SYC    1
#define __Z_TCP_UDP    2
#define __Z_PORT_START 1
#define __Z_PORT_END   65535
#define __Z_TIMEOUT    3
#define __Z_RET_SUCCESS  0
#define __Z_RET_RESOLVE  1
#define __Z_RET_SOCKET  2
#define __Z_RET_PORT_CLOSED  3
#define __VERBOSE  _z_verbose

typedef struct _z_servEnt{

// to be implemented later
} servEnt;

/* -- functions -- */
int z_getPortNum( char * protocol ); // should be re-implemented
int tcpConnectScan( char * host , int timeout, int postStart, int portEnd );
int zport_network_status_checkPort(char * src_host, char * src_protocol, char * dest_host, char * dest_protocol, char * error_str);

++ zport.C
#include "zport.h"

/* -- global variables -- */

int _z_verbose = 0;

/* -- zport_network_status_checkPort */
int zport_network_status_checkPort(char * src_host, char * src_protocol, char * dest_host, char * dest_protocol, char * error_str){
    int portNum = 0, _socket = 0;
struct hostent * hostEnt;
struct sockaddr_in _socketDest;
struct timeval _timeout;

_timeout.tv_sec=_timeout.tv_usec=__Z_TIMEOUT;

/* src */
if(!( hostEnt = gethostbyname( src_host ))) {
    if (__VERBOSE) sprintf( error_str, "can not resolve source host - %s ! ",
                               src_host);
    return __Z_RET_RESOLVE;
}
if(!(_socket = socket(AF_INET, SOCK_STREAM, 0))){
    sprintf(error_str, "can not create socket!");
    return __Z_RET_SOCKET;
}
portNum= z_getPortNum(src_protocol);
if ( portNum) {
    setsockopt(_socket, SOL_SOCKET, SO_RCVTIMEO, ( const char* ) & _timeout, sizeof(_timeout ));
    setsockopt(_socket, SOL_SOCKET, SO_SNDTIMEO, ( const char* ) & _timeout, sizeof(_timeout ));

    _socketDest.sin_family = AF_INET;
    _socketDest.sin_addr = *(( struct in_addr * )hostEnt->h_addr );
    _socketDest.sin_port = htons(portNum);
    if ( connect(_socket, (struct sockaddr *)& _socketDest, sizeof( struct sockaddr)) == -1 ){
        sprintf( error_str, "source port %s:%d is not open !", src_host, portNum);
        return __Z_RET_PORT_CLOSED;
    }
}
close (_socket);
/* src */

/* dest */
if(!( hostEnt = gethostbyname( dest_host ))) {
    if (__VERBOSE) sprintf( error_str, "can not resolve destination host - %s ! ",
                               dest_host);
    return __Z_RET_RESOLVE;
}
if(!(_socket = socket(AF_INET, SOCK_STREAM, 0))){
    sprintf(error_str, "can not create socket!");
    return __Z_RET_SOCKET;
}
portNum = z_getPortNum( dest_protocol );
if ( portNum) {
    setsockopt(_socket, SOL_SOCKET, SO_RCVTIMEO, ( const char* ) & _timeout, sizeof(_timeout ));
    setsockopt(_socket, SOL_SOCKET, SO_SNDTIMEO, ( const char* ) & _timeout, sizeof(_timeout ));

    _socketDest.sin_family = AF_INET;
    _socketDest.sin_addr = *(( struct in_addr * )hostEnt->h_addr );
    _socketDest.sin_port = htons(portNum);
    if ( connect(_socket, (struct sockaddr *)& _socketDest, sizeof( struct sockaddr)) == -1 ){
        sprintf( error_str, "destination port %s:%d is not open !", dest_host, portNum);
        return __Z_RET_PORT_CLOSED;
    }
}
close (_socket);
/* dest */
return __Z_RET_SUCCESS;

}/* zport_network_status_checkPort */

/* -- z_getPortNum -- */
int z_getPortNum( char * protocol ){         // should be re-implemented
    if (!strcmp(protocol, "file")) return 0;
    if (!strcmp(protocol, "ftp"))   return 21;
    if (!strcmp(protocol, "file"))  return 2811;

    if (__VERBOSE) printf(" can not determine port number - protocol %s !\n", protocol);
    return 0;
}/* z_getPortNum */

/* -- z_tcpConnectScan -- */
int z_tcpConnectScan( char * hostName, int timeout , int portStart, int portEnd ){
    int i, _socket = 0;
    struct hostent * hostEnt;
    struct sockaddr_in _socketDest;
    struct servent * _servEnt;
    struct timeval _timeout;

    if (__VERBOSE) printf("starting TCP connect() scan ...
");
    _timeout.tv_sec=_timeout.tv_usec=timeout;

    if(!( hostEnt = gethostbyname( hostName ))) {
        if (__VERBOSE) printf("can not resolve host !\n");
        return __Z_RET_RESOLVE;
    }

    for ( i = portStart; i <= portEnd ; i ++ ){
        if(!(_socket = socket(AF_INET, SOCK_STREAM, 0))){
            if (__VERBOSE)
                printf("can not create socket!
");
            return __Z_RET_SOCKET;
        }

        setsockopt( _socket, SOL_SOCKET, SO_RCVTIMEO, ( const char* ) & _timeout, sizeof( _timeout ));
        setsockopt( _socket, SOL_SOCKET, SO_SNDTIMEO, ( const char* ) & _timeout, sizeof( _timeout ));

        _socketDest.sin_family = AF_INET;
        _socketDest.sin_addr = *(( struct in_addr * )hostEnt->h_addr );
        _socketDest.sin_port = htons(i);

        if ( connect( _socket , (struct sockaddr *)& _socketDest, sizeof( struct sockaddr)) == -1 ){
            if (__VERBOSE) printf("port %d closed!
",i);
        } else {
            _servEnt = getservbyport( htons(i), "tcp");
            //if (__VERBOSE)
                //printf(" port %d (%s) open 
", i, ( _servEnt == NULL) ? "?" : _servEnt->s_name );
            close ( _socket);
        }/* for */

    }/* for */

    return __Z_RET_SUCCESS;
}/* z_tcpConnectScan */
host_exp_group = (char **) safe_malloc(o.ping_group_sz * sizeof(char *));

num_host_exp_groups = 0;
host_exp_group[num_host_exp_groups++] = src;

hstate = new HostGroupState(o.ping_group_sz, o.randomize_hosts, host_exp_group, num_host_exp_groups);

hstate->current_scantype = HOST_DISCOVERY;
currenths = nexthost(hstate, exclude_group, ports, &(o.pingtype));
if (!currenths) {
    snprintf(error_str, MAXSTR, "can not resolve src_host (%s) \n", src); status=1;
    for(i=0; i < num_host_exp_groups; i++)
        free(host_exp_group[i]);
    num_host_exp_groups = 0;
    delete hstate;
}
else{
    o.numhosts_scanned++;

    if (currenths->flags & HOST_UP && !o.listscan)
        o.numhosts_up++;

    if (o.pingscan || o.listscan) {
        if (!(currenths->flags & HOST_UP ))
            snprintf(error_str, MAXSTR, "src_host (%s) is possibly DOWN\n", src); status=1;
            delete currenths;
    }
}

num_host_exp_groups = 0;
host_exp_group[num_host_exp_groups++] = dest;

hstate = new HostGroupState(o.ping_group_sz, o.randomize_hosts, host_exp_group, num_host_exp_groups);

hstate->current_scantype = HOST_DISCOVERY;
currenths = nexthost(hstate, exclude_group, ports, &(o.pingtype));
if (!currenths) {
    snprintf(error_str, MAXSTR, "can not resolve dest_host (%s) \n", dest); status=1;
    for(i=0; i < num_host_exp_groups; i++)
        free(host_exp_group[i]);
    num_host_exp_groups = 0;
    delete hstate;
}
else{
    o.numhosts_scanned++;

    if (currenths->flags & HOST_UP && !o.listscan)
        o.numhosts_up++;

    if (o.pingscan || o.listscan) {
        if (!(currenths->flags & HOST_UP ))
            snprintf(error_str, MAXSTR, "dest_host (%s) is possibly DOWN\n", dest); status=1;
            delete currenths;
    }
}

// Set the variable for status printing
o.numhosts_scanning = Targets.size();
........................................
Appendix C-1 [ Nmap: Option Summary ]

Nmap 4.52 ( http://insecure.org )
Usage: nmap [Scan Type(s)] [Options] {target specification}

TARGET SPECIFICATION:
Can pass hostnames, IP addresses, networks, etc.
Ex: scanme.nmap.org, microsoft.com/24, 192.168.0.1; 10.0.0-255.1-254
-IR <num hosts>: Input from list of hosts/networks
-iL <inputfilename>: Input from list of hosts/networks
-iR <num hosts>: Choose random targets
--exclude <host1[,host2[,host3],...]>: Exclude hosts/networks
--excludefile <exclude_file>: Exclude list from file

HOST DISCOVERY:
-sL: List Scan - simply list targets to scan
-sP: Ping Scan - go no further than determining if host is online
-sT: Treat all hosts as online -- skip host discovery
-sS/PA/PU [portlist]: TCP SYN/ACK or UDP discovery to given ports
-sE/PP/PM: ICMP echo, timestamp, and netmask request discovery probes
-sO [protocol list]: IP Protocol Ping
-n/-R: Never do DNS resolution/Always resolve [default: sometimes]
--dns-servers <serv1[,serv2],...>: Specify custom DNS servers
--system-dns: Use OS's DNS resolver

SCAN TECHNIQUES:
-sS/sT/sA/sW/sM: TCP SYN/Connect()/ACK/Window/Maimon scans
-sU: UDP Scan
-sN/sF/sX: TCP Null, FIN, and Xmas scans
-sC/SA/PU [portlist]: Customize TCP scan flags
-sI <zombie host[:probeport]>: Idle scan
-sO: IP protocol scan
-b <FTP relay host>: FTP bounce scan
--traceroute: Trace hop path to each host
--reason: Display the reason a port is in a particular state

PORT SPECIFICATION AND SCAN ORDER:
-p <port ranges>: Only scan specified ports
-F: Fast mode - Scan fewer ports than the default scan
-r: Scan ports consecutively - don't randomize
-t:<port ranges>: Top ports (number) most common ports
-t: <ratio>: Scan ports more common than <ratio>

SERVICE/VERSION DETECTION:
-sV: Probe open ports to determine service/version info
--version-intensity <level>: Set from 0 (light) to 9 (try all probes)
--version-light: Limit to most likely probes (intensity 2)
--version-all: Try every single probe (intensity 9)
--version-trace: Show detailed version scan activity (for debugging)

SCRIPT SCAN:
-sC: equivalent to --script=safe,intrusive
--script=<Lua scripts>: is a comma separated list of directories, script-files or script-categories
--script-args=<n1=v1,[n2=v2,...]>: provide arguments to scripts
--script-trace: Show all data sent and received
--script-updatedb: Update the script database.

OS DETECTION:
-O: Enable OS detection
--osscan-limit: Limit OS detection to promising targets
--osscan-guess: Guess OS more aggressively

TIMING AND PERFORMANCE:
Options which take <time> are in milliseconds, unless you append 's' (seconds), 'm' (minutes), or 'h' (hours) to the value (e.g. 30m).
-T[0-5]: Set timing template (higher is faster)
--min-hostgroup/max-hostgroup <size>: Parallel host scan group sizes
--min-parallelism/max-parallelism <time>: Probe parallelization
--min-rtt-timeout/max-rtt-timeout/initial-rtt-timeout <time>: Specifies probe round trip time.
--max-retries <tries>: Caps number of port scan probe retransmissions.
--host-timeout <time>: Give up on target after this long
--scan-delay/--max-scan-delay <time>: Adjust delay between probes

FIREWALL/IDS EVASION AND SPOOFING:
-f; --mtu <val>: fragment packets (optionally w/given MTU)
-D <decoy1,decoy2[,ME,]...>: Cloak a scan with decoys
-S <IP_Address>: Spoof source address
```
-e <iface>: Use specified interface
-g/--source-port <portnum>: Use given port number
--data-length <num>: Append random data to sent packets
--ip-options <options>: Send packets with specified ip options
--ttl <val>: Set IP time-to-live field
--spoof-mac <mac address/prefix/vendor name>: Spoof your MAC address
--badsum: Send packets with a bogus TCP/UDP checksum

OUTPUT:
-ON/-O/-O/-O <file>: Output scan in normal, XML, s|<rIpt kIddi3,
 and Grepable format, respectively, to the given filename.
-oA <basename>: Output in the three major formats at once
-v: Increase verbosity level (use twice for more effect)
-d[<level>]: Set or increase debugging level (Up to 9 is meaningful)
-open: Only show open (or possibly open) ports
--packet-trace: Show all packets sent and received
--iflist: Print host interfaces and routes (for debugging)
--log-errors: Log errors/warnings to the normal-format output file
--append-output: Append to rather than clobber specified output files
--resume <filename>: Resume an aborted scan
--stylesheet <path/URL>: XSL stylesheet to transform XML output to HTML
--webxml: Reference stylesheet from Insecure.Org for more portable XML
--no-stylesheet: Prevent associating of XSL stylesheet w/XML output

MISC:
-6: Enable IPv6 scanning
-A: Enables OS detection and Version detection, Script scanning and Traceroute
--datadir <dirname>: Specify custom Nmap data file location
--send-eth/--send-ip: Send using raw ethernet frames or IP packets
--privileged: Assume that the user is fully privileged
--unprivileged: Assume the user lacks raw socket privileges
-V: Print version number
-h: Print this help summary page.

EXAMPLES:
 nmap -v -A scanme.nmap.org
 nmap -v -sP 192.168.0.0/16 10.0.0.0/8
 nmap -v -iR 10000 -PN -p 80

SEE THE MAN PAGE FOR MANY MORE OPTIONS, DESCRIPTIONS, AND EXAMPLES

Appendix C-2 [ test results with rootscan ]

Rootscan was written by shaunige@yahoo.co.uk,
Eckz - mrx@netlane.com - http://freewebs.com/bh_x,
InvisibleGhost : i_t_rules@hotmail.com,
and Ozzy.
Bug testers: Threadhead and Odins_Son, p4n_n0s.
   -u : Scan for UDP Ports
   -s : Scan using SYN scanning (stealthy).
   -t : Scan using TCP connect() scanning (default).
   -p : Scan in parallel mode, using threads (faster in some cases)
   -b number: start scanning at port number. (default = 1)
   -e number: stop scanning at port number. (default = 65535)
   -c number: Set connect() timeout (default = 3,
currently only affects tcp connect() scan.)
   -v: Be verbose (mostly for debugging or checking speed)

$ ./portscan -t virtdev -b 1 -e 1000
  Port   State  Service
  22     Open   ssh
  111    Open   sunrpc
  32768  Open   filenet-tms
Scan complete!

$ ./portscan virtdev -s
  Port   State  Service
socke: Operation not permitted
```
Appendix C-3 [ code of rootscan ] [ from ref. 21 ]

RootScan.C

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <pthread.h>

struct timeval timeout;
#define MAX 1000
#define GREEN ""
#define RED ""
#define BLUE ""
#define TCP_SCAN 1
#define UDP_SCAN 2
#define SYN_SCAN 3
#define NO 0
#define YES 1

/* End of includes and defines */

/* Defining global variables, arrays and functions */

FILE *fp;
char file_to_open[60];
int count = 0;
int n_threads = 0;
int start_port = 1;
int end_port = 65535;
int parallel = NO;
int verbose = NO;
int timeout_sec;
char *host_addr;

struct pseudohdr {
  unsigned long saddr;
  unsigned long daddr;
  char zero;
  unsigned char protocol;
  unsigned short length;
};

/* Set prototypes: */
void usage(char progname[]);
void tcp_scan();
void udp();
void syn();
void *try_udp_port (void *);
void *try_tcp_port(void *);
void *try_syn_port(void *);
unsigned short in_cksum(unsigned short *addr,int len);

/* End of function prototypes */

OpenFiles()
void OpenFiles()
{
  fp = fopen("rootscan.log", "w");
  if( fp == NULL )
  {
    printf(RED "File Open Error\n");
    exit(1);
  }
  fprintf(fp,"Rootscan was written by shaunige@yahoo.co.uk\n");
  fprintf(fp,"\nPorted to windows by : Eckz - mrx@netlane.com - http://eckz.cjb.net\n");
  fprintf(fp,"\nOptions added by : InvisibleGhost : i_t_rules@hotmail.com\n");
  fprintf(fp,"\nTesters/Bug Testers: Threadhead and Odins_Son\n");
  fprintf(fp,"\n******************************************************************\n");
  fprintf(fp,"\n ROOTSCAN LOG \n");
  fprintf(fp,"\n******************************************************************\n");
}

/* End of OpenFiles() function */
/* CloseFiles() function, closes files */
void CloseFiles()
{
    fclose( fp );
}

/* End of global variables, arrays and functions */
/* Main function */
int main(int argc, char *argv[])
{
    char ch;
    int scan_type;
    /* Check args, and print a message if wrong */
    if(argc < 2) {
        usage(argv[0]);
        exit(-1);
    }
    /*Set host address*/
    host_addr = argv[1];
    /* Check command line arguments, and set variables appropriately */
optarg = NULL;
timeout_sec = 3;
    while ((ch = getopt(argc, argv, "sutpvhb:e:c:")) != -1)
        switch (ch)
        {
            case 's':
                scan_type = SYN_SCAN;
                break;
            case 'u':
                scan_type = UDP_SCAN;
                break;
            case 't':
                scan_type = TCP_SCAN;
                break;
            case 'b':
                start_port = atoi(optarg);
                break;
            case 'e':
                end_port = atoi(optarg);
                break;
            case 'p':
                parallel = YES;
                break;
            case 'c':
                timeout_sec = atoi(optarg);
                break;
            case 'v':
                verbose = YES;
                break;
            case 'h':
                usage(argv[0]);
                break;
            default:
                break;
        }
    if (verbose == YES) printf("Scanning host: %s\n", host_addr);
    if (verbose == YES && parallel == YES) printf("Going into parallel mode.\n");
    switch (scan_type)
    {
        case TCP_SCAN:
            tcp_scan();
            break;
        case UDP_SCAN:
            udp();
            break;
        case SYN_SCAN:
            syn();
            break;
        default:
            tcp_scan();
            break;
    }
}
void udp() {
    if (verbose == YES) printf("Beginning udp scan from: %d to: %d\n", start_port, end_port);
    if ((gethostbyname(host_addr)) == NULL) {
        printf("Couldn't resolve %s\n", host_addr);
        exit(-1);
    }
    OpenFiles();
    for(count = start_port; count <= end_port; count++) {
        if (parallel == YES) {
            pthread_t thread_t;
            pthread_detach(thread_t);
            n_threads++;
            if (pthread_create(&thread_t, NULL, try_udp_port, (void *)count)) {
                count--;
                n_threads--;
            }
        } else {
            try_udp_port((void *)count);
        }
        if (verbose == YES) printf("\rPort: %d\r", count);
    }
    CloseFiles();
}

void tcp_scan() {
    if (verbose == YES) printf("Beginning tcp connect() scan from: %d to: %d\n", start_port, end_port);
    if ((gethostbyname(host_addr)) == NULL) {
        printf("Couldn't resolve %s\n", host_addr);
        exit(-1);
    }
    printf(\t\tPort\t	State\t	Service\n\n";\n    for(count = start_port; count <= end_port; count++) {
        if (parallel == YES) {
            pthread_t thread_t;
            pthread_detach(thread_t);
            n_threads++;
            if (pthread_create(&thread_t, NULL, try_tcp_port, (void *)count)) {
                count--;
                n_threads--;
            }
        } else {
            try_tcp_port((void *)count);
        }
    }
    printf(\n\n"Scan complete!\n\n";\n    CloseFiles();
} /* End function */
void syn() {
    if (verbose == YES) printf("Beginning syn stealth scan from: %d to: %d\n", start_port, end_port);
    if((gethostbyname(host_addr)) == NULL) {
        printf(RED "Couldn't resolve %s!\n", host_addr);
        exit(-1);
    }
    OpenFiles();
    printf(BLUE "\tPort\t\tState\t\tService\n\n");
    for(count = start_port; count <= end_port; count++) {
        if (parallel == YES)
        {
            pthread_t thread_t;
            pthread_detach(thread_t);
            n_threads++;
            if (pthread_create(&thread_t, NULL, try_syn_port, (void *)count))
            {
                count--;
                n_threads--;
            }
        }
        else
        {
            try_syn_port((void *)count);
        }
    } /* End for() */
    } /* End function */
}

void *try_udp_port(void *tmp)
{
    int port = (int)(tmp);
    int sock;
    struct hostent *udp_host;
    struct sockaddr_in udp_dest;
    char udp_data[] = "hello\0";
    int udp_len = strlen(udp_data);
    char udp_buf[20];
    int sin_len = sizeof(struct sockaddr);
    if((udp_host = gethostbyname(host_addr)) == NULL)
    {
        printf(RED "Couldn't resolve %s\n", host_addr);
        exit(-1);
    }
    /* Create a SOCK_DGRAM socket instead, SOCK_DGRAM is UDP socket */
    if((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        printf(RED "Couldn't create datagram socket!\n");
        exit(-1);
    }
    /* Fill in address structs. */
    udp_dest.sin_family = AF_INET;
    udp_dest.sin_port = htons(port);
    udp_dest.sin_addr = *(struct in_addr *)&udp_host->h_addr;
    /* Send the datagram. */
    sendto(sock, udp_data, udp_len, 0, (struct sockaddr *)&udp_host, sizeof(struct sockaddr));
    /* If we get a reply, the port is probably closed. */
    if(recvfrom(sock, udp_buf, 19, 0, (struct sockaddr *)&udp_host, &sin_len) < 0) {
        close(sock);
        fprintf(fp, "Port %d \t Closed\n", count);
    } /* If we received no response, the port is probably open. */
    else {
        printf(RED "Port %d \t Open\n", count);
        close(sock);
        fprintf(fp, "Port %d \t Open\n", count);
    }
    close(sock);
}
void *try_tcp_port (void *tmp)
{
    int port = (int)(tmp);
    int sock = 0;
    char http;
    char httpsend[18] = "HEAD / HTTP/1.0\n\n";
    char httpbuf[MAX];
    struct hostent *tcp_host;
    struct sockaddr_in tcp_dest;
    struct servent *serv;
    if (verbose == YES) printf("Port: %d\r", port);
    if((tcp_host = gethostbyname(host_addr)) == NULL)
    {
        printf(RED "Couldn't resolve %s
", host_addr);
        exit(-1);
    }
    /* Creating the socket, with the integer variable called sock, checking if it succeeded, socket()
     * returns -1 on error */
    if((sock = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        printf(RED "Couldn't make socket!
");
        exit(-1);
    }
    setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout, sizeof(timeout));
    setsockopt(sock, SOL_SOCKET, SO_SNDTIMEO, (const char*)&timeout, sizeof(timeout));
    /* Setting up the sockaddr_in struct with connection details, port, 'family', hostname/IP address */
    tcp_dest.sin_family = AF_INET;
    tcp_dest.sin_port = htons(port);
    tcp_dest.sin_addr = *((struct in_addr *)tcp_host->h_addr);
    /* Connecting the sock to the host on the port the for loop is up to */
    if (connect(sock , (struct sockaddr *)&tcp_dest, sizeof(struct sockaddr)) == -1)
    {
        fprintf( fp ,"Port %5d Closed
", port);
        close(sock);
    }
    else {
        /* Get the service name the port is likely to be. */
        serv = getservbyport(htons(port), "tcp");
        printf(RED "\n\n\t%d \tOpen \t%\n\n", port, (serv == NULL) ? "UNKNOWN" : serv->s_name);
        fprintf( fp ,"Port %5d Open \ %s\n", port, (serv == NULL) ? "UNKNOWN" : serv->s_name);
        /* If the variable the for loop is using equals 80, they might be running a web server, get the
         * version? */
        if(port == 80)
        {
            printf(GREEN "\n\nThe host is running a HTTP server, get HTTPD version? [y/n]"");
            scanf("%c", &http);
            if(http == 'y')
            {
                fprintf(fp, "\nHTTP version response:\n");
                /* Sending HEAD / HTTP/1.0\n\n to get the version. */
                send(sock, httpsend, strlen(httpsend), 0);
                /* Receiving the result, store it in httpbuf */
                recv(sock, httpbuf, MAX-1, 0);
                /* Print it to stdout (monitor) */
                printf("%s", httpbuf);
                /* Print it to the file */
                fprintf(fp, "%s", httpbuf);
            }
        }
        /* Close the socket */
        close(sock);
    }
}

void *try_syn_port(void *tmp)
{
    int port = (int)(tmp);
    int sock;
struct hostent *h = gethostbyname(host_addr);
/* the variables */
int on=1;
int ssize = sizeof(struct sockaddr_in);
int packet_size = (sizeof(struct tcphdr)+sizeof(struct iphdr));
char *packet = malloc(packet_size);
char *received = malloc(packet_size);
/* The headers */
struct tcphdr *tcph = (struct tcphdr *)(packet+sizeof(struct iphdr));
struct pseudohdr *pseudo = (struct pseudohdr *)(packet+sizeof(struct iphdr)+sizeof(struct
tcphdr));
struct iphdr *iph   = (struct iphdr  *)(packet);
struct iphdr *iphr;
struct sockaddr_in local;
struct sockaddr_in remote;
struct in_addr saddr, daddr;
struct servent *serv;
/* making socket, and telling kernel we fill in the ip header */
if( (sock = socket( PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0 )
    { perror("socket");  exit(1); }
if( (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on))) < 0 )
    { perror("setsockopt");  exit(1); }
daddr = *((struct in_addr *)h->h_addr);
saddr.s_addr = inet_addr("62.254.68.38"); /* Change the IP address here. */
/***********************/
/* The pseudo header for the checksum */
pseudo->saddr = saddr.s_addr;
pseudo->daddr = daddr.s_addr;
pseudo->protocol = IPPROTO_TCP;
pseudo->zer0 = 0;
pseudo->length = htons(sizeof(struct tcphdr));
bzero( packet, packet_size );
tcph->source = htons(8901);
tcph->dest   = htons(port);
tcph->seq    = htonl(random()%time(NULL));
tcph->ack_seq= htonl(random()%time(NULL));
tcph->doff = 5;
tcph->res1 = 0;
tcph->fin   = 0;
tcph->syn = 1;
tcph->rst = 0;
tcph->psh = 0;
tcph->ack = 0;
tcph->urg = 0;
tcph->window = htons(12000);
tcph->check = (unsigned short)in_cksum((unsigned short *)tcph, sizeof(struct
tcphdr)+sizeof(struct pseudohdr));
bzero(packet, sizeof(struct iphdr));
iph->ihl = 5;
iph->version = 4;
iph->tos = 0;
iph->tot_len = htons(packet_size);
iph->frag_off = 0;
iph->ttl = IPDEFTTL;
iph->protocol = IPPROTO_TCP;
iph->check = (unsigned short)in_cksum((unsigned short *)iph, sizeof(struct iphdr));
iph->saddr = saddr.s_addr;
iph->daddr = daddr.s_addr;
remote.sin_family = PF_INET;
remote.sin_addr = daddr;
remote.sin_port = htons(port);
if ( sendto(sock, packet, packet_size, 0x0, (struct sockaddr *)&remote, sizeof(remote)) < 0 )
    { perror("sendto"); exit(1); } 

bzero(packet, packet_size);
if( (recvfrom(sock, received, packet_size, 0x0, (struct sockaddr *)&local, &ssize )) < 0 )
    { perror("recvfrom"); exit(1); } 

iphr = (struct iphdr *)(received);
tcphr = (struct tcphdr *)(received+(int)((iphr->ihl)*4)); /* using fixed sizes can be tricky, so
i dont; */ 
if( tcphr->syn == 1 && tcphr->ack == 1 ) /* SYN/ACK */
    {
        serv = getservbyport(htons(port), "tcp");
        printf("\t\%d \t\ Open \t\ %s\n", ntohs(tcphr->source), (serv == NULL) ? "UNKNOWN" : serv->s_name);
    }
else if( tcphr->rst == 1 ) /* RST */
    { /* anyone a suggestion what could be done here? */
    }
else /* This shouldn't happen */
    {
        printf("Protocol violation :P\n");
        exit(-2);
    }

close(sock);

/* The checksum function from the raw ip faq */
unsigned short in_cksum(unsigned short *addr,int len)
{
    register int sum = 0;
    u_short answer = 0;
    register u_short *w = addr;
    register int nleft = len;

    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    if (nleft == 1) {
        *(u_char *)(&answer) = *(u_char *)w;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}

void usage(char *progname) {
    printf(RED "Usage: %s <host/ip> [-tsu] [-p] [-b number] [-e number] [-c number] [-v]\n", progname);
    printf(RED "\n\nRootscan was written by shaunige@yahoo.co.uk,\nEckz - mrx@netlane.com -\nhttp://freewebs.com/bh_x,\n\nInvisibleGhost : i_t_rules@hotmail.com,\nand Ozzy.\nBug testers: Threadhead and\nOdins_Son, p4n_n0s.\n\nUDP scanning option is currently experimental.\n\n" tragedy );
    printf(RED "\n\t\-u : Scan for UDP Ports\n" tragedy );
    printf(RED "\n\t\-tsu : Scan using SYN scanning (stealthy).\n" tragedy );
    printf(RED "\n\t\-t : Scan using TCP connect() scanning (default).\n" tragedy );
printf(RED "\t-p : Scan in parallel mode, using threads (faster in some cases)\n");
printf(RED "\t-b number: start scanning at port number. (default = 1)\n");
printf(RED "\t-e number: stop scanning at port number. (default = 65535)\n");
printf(RED "\t-c number: Set connect() timeout (default = 3, \ncurrently only affects tcp\nconnect() scan.)\n");
printf(RED "\t-v: Be verbose (mostly for debugging or checking speed)\n\n");
exit(-1);)
/* End function */