Abstract

This memo defines a mechanism for SSH clients and servers to counter some types of traffic analysis, especially keystroke analysis, in SSH terminal session channels.

Status

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Copyright

Copyright (c) 2016 IETF Trust and the persons identified as the document authors.  All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document.  Please review these documents carefully, as they describe your rights and restrictions with respect to this document.  Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Overview and Rationale

Secure Shell (SSH) is a common protocol for secure communication on the Internet. Despite encryption, naive SSH terminal implementations are vulnerable to keystroke traffic analysis. When a user is typing, each keystroke results in a small packet sent to the server, and the server echoes the key back. The sizes and timings of packets betray how many keys the user has pressed, as well as what key combinations are most likely, because different key combinations cause subtle but consistently different delays.

It is also possible to analyze non-keystroke traffic in the direction from the server to client. When a client makes a directory listing, the resulting encrypted TCP stream is distinguishable from viewing a man page, or running a SQL statement.

This memo defines a mechanism intended to counter keystroke analysis, and to generally improve terminal session privacy.

1.1. Requirements Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "MUST NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Mechanism

2.1. Entering Fixed-Bandwidth Mode

Because the SSH server will generally echo received keystrokes to the client, defending against keystroke analysis requires cooperation between the client and server.

To enable fixed-bandwidth mode, the client sends the following channel request at any point after opening the channel [RFC4254]:

```plaintext
byte    SSH_MSG_CHANNEL_REQUEST
uint32  remoteChannelNr
string  "fixed-bandwidth"
bool    wantReply
uint32  burstWindowSeconds
uint32  millisecondsBetweenPackets
uint32  bytesPerPacket
```

This mechanism is mainly intended for channels of type "session" that are used for interactive terminal, but a client MAY use it for other channels. A server that responds positively to this request MUST enable fixed-bandwidth mode for that channel. A server MAY internally constrain parameters of this request to values it deems sensible.
If a server responds negatively to this request, a client MAY still enter fixed-bandwidth mode for data sent in the direction from client-to-server. However, such a server is likely to echo keystrokes back to the client, defeating protection.

When fixed-bandwidth mode is requested by the client and confirmed by the server, the channel starts in the resting state.

2.1. Resting State

In resting state, the channel waits either for an outgoing channel message to send, or for an incoming channel message received. Each of these is a triggering event.

When a triggering event occurs – either a channel message is ready to send, or a channel message is received – this causes the channel to enter two or more consecutive burst windows.

2.2. Burst Window

When a triggering event occurs, the channel enters a burst window of duration burstWindowSeconds. A channel returns to resting state at the end of a burst window – if, and only if, no triggering event occurred during the last burst window. If any channel message was sent or received during a burst window; even if this was only the initial event that started the burst; then the channel continues with another burst window of duration burstWindowSeconds.

When the channel is in a burst window; instead of sending messages as it would outside this mechanism; the channel operates as follows:

(1) The SSH implementation operates a timer. It checks if a message should be sent on the channel every millisecondsBetweenPackets.

(2) If a channel message is to be sent; and is larger than what it takes to construct a packet of final size bytesPerPacket; the implementation sends it as a packet of size bytesPerPacket. Any further channel messages or data are left in queue, to be considered for sending after millisecondsBetweenPackets.

(3) If a channel message is to be sent; but is smaller than what it takes to construct a packet of final size bytesPerPacket; then the implementation sends the message; followed back-to-back by an SSH_MSG_IGNORE packet such that its size, and the size of the preceding channel message, add up to exactly bytesPerPacket.

(4) If no channel message is to be sent, the implementation sends an SSH_MSG_IGNORE packet such that its final size is bytesPerPacket.
The preceding treatment is applied to the following channel messages:

- SSH_MSG_CHANNEL_DATA
- SSH_MSG_CHANNEL_EXTENDED_DATA
- SSH_MSG_CHANNEL_WINDOW_ADJUST
- SSH_MSG_CHANNEL_REQUEST
- SSH_MSG_CHANNEL_SUCCESS
- SSH_MSG_CHANNEL_FAILURE
- SSH_MSG_CHANNEL_EOF
- SSH_MSG_CHANNEL_CLOSE

The size bytesPerPacket refers to the final externally observable size visible to an attacker, including padding and integrity tag. An SSH implementation MAY perform adjustments and fuzzing on this target size, but MUST avoid this revealing information to an attacker.

3. Security Considerations

3.1. TCP Sending

When multiple packets are sent back-to-back; such as a channel message followed by SSH_MSG_IGNORE to conceal its size; it is important that the encrypted back-to-back messages are delivered to the TCP layer in a way indistinguishable from a single SSH_MSG_IGNORE of same size.

3.2. Unencrypted Packet Lengths

The goals of this mechanism are more difficult to achieve if the SSH session negotiates an encryption method in either direction that does not encrypt packet lengths. When data to be sent is too short to produce a packet of size bytesPerPacket, it is impossible to hide this by appending an SSH_MSG_IGNORE packet back-to-back. If this mechanism is used with unencrypted packet lengths, implementations MUST target a final packet size small enough to be consistently achievable using internal padding only. Implementations MAY compensate by sending multiple small packets per each socket send. However, in this regime - any large packets that cannot be broken up; such as a large SSH_MSG_CHANNEL_REQUEST; will be plainly observable.

4. IANA Considerations

4.1. Additions to existing tables

IANA is requested to insert the following entry into the table Connection Protocol Channel Request Names under Secure Shell (SSH) Protocol Parameters [RFC4250]:

<table>
<thead>
<tr>
<th>Request Type</th>
<th>Reference</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-bandwidth</td>
<td>[this document]</td>
<td>Section 2.1</td>
</tr>
</tbody>
</table>
6. References

6.1. Normative References


Author’s Address

Denis Bider
Bitvise Limited
Suites 41/42, Victoria House
26 Main Street
GI

Phone: +506 8315 6519
EMail: ietf-ssh3@denisbider.com
URI:  https://www.bitvise.com/