The saltunnel protocol

Jay Sullivan

July 13, 2020

Abstract

The `saltunnel` *protocol* is a cryptographically secure transport protocol. It is used by the `saltunnel` *utility*, which allows one to augment a normally-insecure TCP session with state-of-the-art security, with minimal hassle and minimal impact on performance. This paper describes the protocol used between the `saltunnel` software's client and server. In most cases, the term "`saltunnel`" can be used to refer to either the software or the protocol.

1 The `saltunnel` protocol

1.1 Overview

A `saltunnel` *session* occurs between a `saltunnel` client and a `saltunnel` server, both which share, as a prerequisite, a 256-bit shared long-term key k. The normal flow of the protocol works as follows:

	client		server
(1)	$clienthi$	$\rightarrow$	
(2)		$\leftarrow$	$serverhi$
(3)	$message(0)$	$\rightarrow$	
(4)	$message(n)$	$\Rightarrow \Leftarrow$	$message(n)$
	...	$\Rightarrow \Leftarrow$	...

1. The client connects to the server, and immediately sends a `clienthi` (defined later).
2. The server receives the `clienthi`, and, if it passes verification, responds with a `serverhi` (also defined later). However, the server must wait until it has authenticated the client before performing any non-idempotent or resource-intensive actions (e.g., in some protocols, certain behavior may occur as soon as a connection is opened).
3. The client then receives the `serverhi`, and, if it passes verification, **the client has now authenticated the server:** this means the client is now permitted to read (or forward somewhere else to be read) any data received from the server. The client then sends its first message to the server.
4. The server receives the client's first message, and, if it passes verification, **the server has now authenticated the client,** and therefore the server may now considered the connection established. At this point, if the server needs to, it may perform any non-idempotent or resource-intensive behavior which needs to be performed immediately after connection establishment. At this point, both parties have authenticated each other. They may now both freely send each other messages.

1.2 The clienthi

The `clienthi` has the following structure:

$$clienthi = \begin{cases} \texttt{uint8 nonce[24]} \\ \texttt{uint8 auth[16]} \\ \boxed{\begin{array}{ll} \texttt{uint8} & \texttt{version[8]} \\ \texttt{uint8} & \texttt{public\_key[32]} \\ \texttt{uint64} & \texttt{timestamp} \\ \texttt{uint8} & \texttt{machine\_id[16]} \\ \texttt{uint64} & \texttt{machine\_counter} \\ \texttt{uint8} & \texttt{zeros[400]} \end{array}} \end{cases}$$

The entire `clienthi` structure is 512 bytes in length, and consists of a 24-byte random nonce (*nonce*), a 16-byte authenticator (*auth*), with remaining fields enclosed in a 472-byte encrypted "box".

The `clienthi` box is encrypted/authenticated (by the client, before sending) using the `xsalsa20poly1305` cipher [1] [2] [3] with a fresh random nonce *nonce* and the shared long-term key k, resulting in authenticator *auth*. The contents of the box are as follows:

1. `version` - The 16-byte string `0x060528849a6108c7` (for `saltunnel-1.0.0`).
2. `public_key` - A 32-byte ephemeral `curve25519` public key [4]; an ephemeral keypair is generated for use only within this connection, and erased as soon as the connection is terminated.
3. `timestamp` - A 64-bit, big-endian encoded, unsigned integer representing the epoch time in seconds (i.e., number of seconds since `1970-01-01T00:00:00Z`) according to the client's system clock.
4. `machine_id` - A 16-byte unique string which uniquely identifies this machine along with the last time it has rebooted. For example, a hash of both the machine's `/etc/machine-id` (or MAC address) and last boot timestamp.
5. `machine_counter` - A 64-bit, big-endian encoded, monotonic (always increasing, never repeating) unsigned integer which must be unique and monotonic (per `machine_id`) since (at least) the last boot. (For example, `clock_gettime(CLOCK_MONOTONIC, ...)`.)
6. `zeros` - Always set to zero.

As soon as the server receives the client's `clienthi`, it (the server) immediately performs the following validation steps:

7. Verify+decrypt the encrypted data with nonce *nonce*, authenticator *auth*, and key k.
8. Verify that `version` equals `0x060528849a6108c7` (for `saltunnel-1.0.0`).
9. Verify that `timestamp` is not older or newer than 1 hour according to the server's system clock.
10. Verify that, considering each `clienthi` ever received matching the current packet's `machine_id`, that this instance of `machine_counter` (a 64-bit big-endian unsigned integer) is *greater than* than all previous instances.

If any of the above steps fail, the server will immediately close the TCP connection—no error will be sent back to the client. Otherwise, the server responds with a `serverhi`.

1.3 The serverhi

The `serverhi` has the following structure:

$$serverhi = \begin{cases} \texttt{uint8} & \texttt{nonce[24]} \\ \texttt{uint8} & \texttt{auth[16]} \\ \boxed{\begin{array}{ll} \texttt{uint8} & \texttt{version[8]} \\ \texttt{uint8} & \texttt{public\_key[32]} \\ \texttt{uint8} & \texttt{proof[16]} \\ \texttt{uint8} & \texttt{zeros[416]} \end{array}} \end{cases}$$

The entire `serverhi` structure, like the `clienthi`, is also 512 bytes in length, and consists of a 24-byte random nonce (*nonce*), a 16-byte authenticator (*auth*), and a 472-byte encrypted "box".

The `serverhi` box is, similar to `clienthi`, is encrypted/authenticated (by the server, before sending) using the `xsalsa20poly1305` cipher with a fresh random nonce *nonce* and the shared long-term key k, resulting in authenticator *auth*. The contents of the box are as follows:

1. `version` - The 16-byte string `0x060528849a6108c7` (for `saltunnel-1.0.0`).
2. `public_key` - A 32-byte ephemeral `curve25519` public key; an ephemeral keypair is generated for use only within this connection, and erased as soon as the connection is terminated.

   Additionally, the *server session key* (not to be confused with the shared long-term key k) can now be calculated. The server session key is calculated by peforming an ECDH (elliptic-curve Diffie-Hellman) operation on both the client's and server's public keys, then using the resulting curve point as the first 32-bytes of input to (with the last 32-bytes of input being set to zero) the `salsa20` hash function, then taking the *last* 32 bytes of the output.
3. `proof` - The 16-byte value (`client_session_key[0..15]` $\oplus$ `server_session_key[16..31]`). This proves to the client that the server knows both session keys.
4. `zeros` - Always set to zero.

As soon as the client receives the server's `serverhi`, it (the client) immediately performs the following validation steps:

5. Verify+decrypt the encrypted data with nonce *nonce*, authenticator *auth*, and key k.
6. Verify that `version` equals `0x060528849a6108c7` (for `saltunnel-1.0.0`).
7. Calculate the *server session key* (as explained previously), and verify the `proof`.

If any of the above steps fail, the client will immediately close the TCP connection—no error will be sent back to the server. Otherwise, if all steps so far were successful, then the client now trusts the server.

The client can now calculate the *client session key* (not to be confused with the shared long-term key k or with the *server session key*). The *client session key* is calculated by peforming an ECDH (elliptic-curve Diffie-Hellman) operation on both the client's and server's public keys, then using the resulting curve point as the first 32-bytes of input to (with the last 32-bytes of input being set to zero) the `salsa20` hash function, then taking the *first* 32 bytes of the output.

The client then must send its first message before the server will consider the client trustworthy (i.e., before the server performs any non-idempotent actions).

1.4 Messages

Messages are always exactly 512-bytes in length, and consist of a 16-byte authenticator followed by a 496-byte encrypted box:

$$message = \begin{cases} \texttt{uint8 auth[16]} \\ \boxed{\begin{array}{l} \texttt{uint8 len[2]} \\ \texttt{uint8 data[494]} \end{array}} \end{cases}$$

Messages are encrypted using the `salsa20poly1305` cipher. Messages sent from the client are encrypted with the *client session key*, and messages sent from the server are encrypted with the *server session key*. The first message sent from the client and the first message sent from the server will use nonce `0`, the second message(s) will use nonce `1`, the third message(s) will use nonce `2`, etc.

Inside each message is up to 494-bytes of data ("`data`"), along with a 2-byte length ("`len`") which specifies the actual number of bytes used in `data`.

As stated previously, the client must send the first message, immediately after verifying the `serverhi`—the server will wait for the client's first message before trusting the client. If, when it is time for the client to send its first message, it does not have any data available to send (for example, in a server-sends-first protocol), then the client will immediately send a message with `len` set to `0`.

When the server receives the client's first message, the encrypted box will be verified+decrypted. If this successful, the server then trusts the client. If both the client and server made it this far without any errors, they have reached a mutual trust. They may now freely exchange messages, in addition to forwarding packets outside of the tunnel.

2 Alternative Ciphers

The choice of `xsalsa20poly1305`, `salsa20poly1305`, and `curve25519` were due to easy access to high-performance implementations at the time of this writing. The protocol need not be bound to these choices: it should be trivial to provide command-line options to a `saltunnel` implementation to allow alternate encryption and key exchange algorithms, without affecting the protocol.

3 Security Features

The **saltunnel** protocol provides the following security features:

Confidentiality and Integrity

The **saltunnel** protocol uses `xsalsa20poly1305` and `salsa20poly1305` as its primary symmetric ciphers, which provide strong confidentiality and integrity.

Denial-of-Service Protection

Additionally, `saltunnel` has considerable denial-of-service protection: an attacker attempting to overload a **saltunnel server** instance will find that his/her most cost-effective attacks are (probably) reduced to exploiting the well-known shortcomings of TCP itself.

The use of a `timestamp` in the `clienthi` allows the server to immmediately reject any obviously-stale connection requests. The use of `machine_id` and `machine_counter` allow the server to maintain

an in-memory hash table to keep track of all recent connection requests, and to detect (and block) any replay attacks.

Forward Secrecy

Forward secrecy is also provided. For each TCP connection, both the client and server generate ephemeral `curve25519` key pairs, followed by a shared session key used to encrypt all data within that connection. After each TCP connection terminates, all ephemeral and session keys which were used to encrypt that connection are permanently erased.

This means it is impossible for even an attacker who gains access to the shared long-term key to retroactively decrypt any data from any previous sessions. (With one exception: an attacker who holds both the shared long-term key k and a quantum computer can retroactively decrypt historical connections.)

Post-Quantum Security

Since `saltunnel` primarily relies on 256-bit symmetric-key cryptographically, it is not necessarily vulnerable to being broken by quantum computing attacks (e.g., Shor's algorithm). Admittedly, there are two pitfalls with this claim:

1. Confidentiality is broken by an attacker who holds *both* the shared long-term key k, *and* a quantum computer.
2. `saltunnel`'s preqrequisite that the client and server must share a long-term key side-steps the difficulty involved in exchange such a key in post-quantum-secure manner.

However, provided both client and server both manage to permanently keep their shared long-term key away from attackers, post-quantum security is retained.

Message-Length Quantization

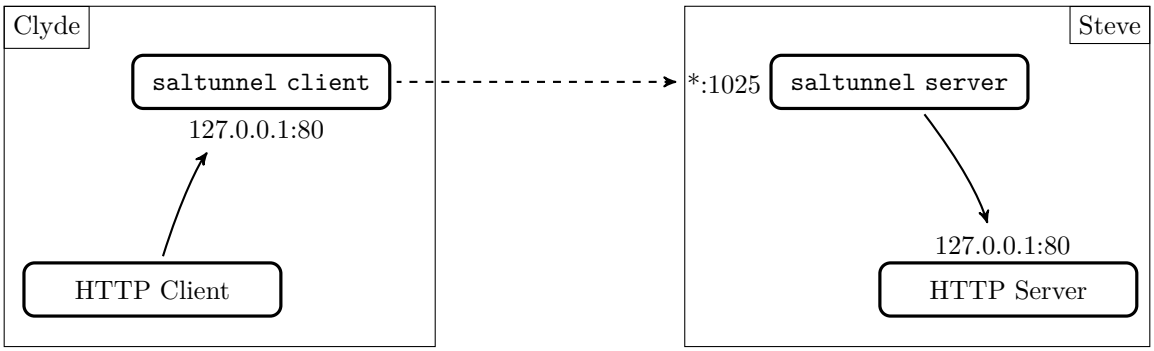
Data is sent over the network in chunks of 512 bytes. If one computer sends, for example, 7 bytes, the data will be sent as a 512-byte encrypted chunk. This greatly reduces the amount of information which can be inferred from network analysis.

Uniform Random TCP Data

An attacker who eavesdrops on a `saltunnel` connection will only see a TCP connection with same-sized chunks of uniform random data. There are no protocol header magic bytes or patterns within any given chunk of data. This complicates an attacker's ability to track/fingerprint users, and makes it difficult to write analysis tools that detect/censor the saltunnel protocol. (Obviously, tracking can still be done at the IP layer, and patterns may still be available through timing analysis.)

4 The `saltunnel` tunnel

So far, this document has described the protocol which takes place between a `saltunnel` server and `saltunnel` client. This protocol on its own is simply a symmetric-key based secure transport protocol. This protocol, in fact, is not a TCP tunnel on its own—it is the `saltunnel` *utility* which provides a TCP tunnel. In order to explain how the software works, consider the following scenario:



Imagine that Clyde and Steve, who live in different households, each have their own computer and that Clyde would like to establish a TCP session to port 80 on Steve's computer. However, the program Clyde is using to connect to Steve's computer does not encrypt its traffic and is therefore too insecure to use comfortably over a network. This is where `saltunnel` comes in.

First, Clyde and Steve must exchange a shared long-term key, which is placed at location on both of their computers.

Clyde then sets up a `saltunnel client` from `127.0.0.1:80` to `*:1025` (where `*` represents whatever Steve's public IP address is). This tells `saltunnel` to *listen* for plain, unencrypted traffic on Clyde's computer's local port `80`, then *forward* all traffic—while encrypting+authenticating it using the `saltunnel` protocol—to Steve's public port `1025`.[1]

At the same time, Steve sets up a `saltunnel server`, which both *listens* for encrypted traffic on its public port `1025`, and *forwards* all traffic—while verifying+decrypting it using the `saltunnel` protocol—to Steve's own local port `80`.

After all of this is complete, any program running on Clyde's computer can connect to local port `80`, and the connection will act as if it was directly connecting to Steve's computer's local port `80`.

5 Additional resources

The `saltunnel` utility is available at:

`https://github.com/notfed/saltunnel`

This software implements the protocol and utility as defined in this document, and provides a convenient, easy-to-use command-line interface, written in C. The code is public domain.

References

[1] Daniel J Bernstein. The Salsa20 family of stream ciphers. In *New stream cipher designs*, pages 84–97. Springer, 2008.

[2] Daniel J Bernstein. Extending the Salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop*, volume 2011, 2011.

[3] Daniel J Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.

[4] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[1] The ports used in this example are arbitrary. Any ports could be used for any of these steps.