SSH

Secure Shell Overview

Zanolin Lorenzo[1]

[1]DMIF
University of Udine

August 2023

UNIVERSITÀ
DEGLI STUDI
DI UDINE

hic sunt futura

Table of Contents

1. Introduction

2. Architecture

3. Attacks and Vulnerabilities

4. Conclusions

Introduction

What is SSH?

SSH (Secure Shell) is a cryptographic network protocol for secure remote access and data communication over an insecure network.

This tool is a great improvement with respect to ftp and rcp (for file transfers), telnet and rlogin (for remote logins), and rsh (for remote execution of commands) since it can be used to all these tasks while preserving

- confidentiality
- authentication
- integrity

Let us see an example.

Example

First, we can remote login as user lorenzo on the ssh server host.example.com:

```
$ ssh -l lorenzo@server.example.com
```

Now, a connection is established, which means that we are inside the remote machine. As example, we can run a command on the remote machine as it

would be on our local machine:

```
$ ssh lorenzo@server.example.com ls -l
```

Let us see now a brief history of the available versions currently out.

History

The first version of SSH was developed in the mid-1990s as a secure alternative to the insecure Telnet and rlogin protocols. It was called **SSH-1**. Important facts:

- It provided encrypted communication and authentication methods.
- The structure was monolithic, encompassing multiple functions in a single protocol.
- It was deprecated because of some security vulnerabilities and weaknesses. For instance, certain algorithms employed were susceptible to attacks, such as DES or CRC32.

In response to the security flaws in SSH-1, **SSH-2** was developed to address these issues and provide improved security.

History (cont’d)

The standard now has become SSH-2, which has a different structure and uses only robust algorithms.

The **architecture** of this protocol consists of three components:

- Transport Layer protocol: supplies server authentication to the client, confidentiality, and integrity.
- User Authentication protocol: provides client authentication to the server and runs over Transport layer.
- Connection Protocol: multiplexes the encrypted tunnel into several logical channels and runs over User Authentication layer.

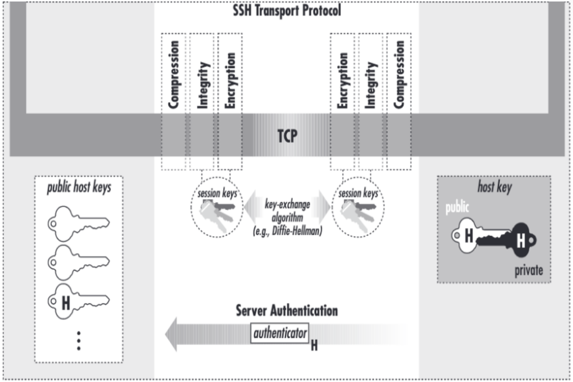
Keep in mind that all these components runs over TCP/IP connection, thus above ISO/OSI level 4.

Architecture

Transport Layer

First of all we will focus on the core of SSH, the transport layer.
It provides:

- Server authentication
- Confidentiality
- Integrity
- Compression



Let us analyze each of them one by individually.

Server Authentication

Server possess an **asymmetric** $host_{key}$, represented as a tuple

$$host_{key} = (PR_{srv}, PU_{srv})$$

it enables secure server authentication during the symmetric key exchange process. The two parts are stored locally in the server and PR_{srv} is saved encrypted with a password chosen by the administrator.

It’s crucial to keep in mind that PU_{srv} should be stored manually in the client’s database. Alternatively, automatic insertion during the initial connection is possible, but this leaves us vulnerable to a **man-in-the-middle attack**, as we’ll discuss.

Server Authentication (cont'd)

The real authentication happens during the **initial handshake**.
During this phase, each side sends a name-lists of supported algorithms (encryption, integrity, compression, key exchange), listed in order of preference.

After the negotiation concludes, the *Diffie-Hellman* key exchange is executed, during which the server authenticates itself.
This process produces two values: a shared secret K and an exchange hash H; these two values will be used later to build the session key.
It is important to remind that SSH-2 introduced **rekeying**. K and H will be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner.

We will now see more in detail the steps of the key-exchange.

SSH Diffie-Hellman

This procedure serves the dual purpose of verifying the server's authenticity and establishing a session key between the server and the client.
The steps are the following:

1. Client generates a random number x such that $1 < x < q$ and computes $e = g^x \ mod \ p$. Client sends e to Server.

$$C \rightarrow S : e = g^x \ mod \ p$$

2. Server generates a random number y such that $0 < y < q$ and computes $f = g^y \ mod \ p$. Server receives e and computes

$$K = e^y \ mod \ p$$

$$H = hash(V_C||V_S||I_C||I_S||K_S||e||f||K)$$

and the signature s on H with its private key.

SSH Diffie-Hellman (cont'd)

3. Server sends (K_S, f, s) to Client.

$$S \rightarrow C : (K_S, f, s)$$

4. The client verifies whether K_S is the $host_{key}$ for the server by checking if K_S belongs to his local database. If it is not present, the client has the option to either add it to the file or terminate the connection.
5. Client then computes

$$K = f^x \; mod \; p$$

$$H = hash(V_C||V_S||I_C||I_S||K_S||e||f||K)$$

and verifies the signature s on H.

Now the server is authenticated to the client, and both can build their session key.

Confidentiality

To ensure confidentiality we need **encryption**.There are a lot of supported symmetric algorithms, like 3des-cbc, blowfish-cbc, aes256-cbc, aes192-cbc, aes128-cbc,...

As already written, symmetric key is derived from two values: a shared secret K and an exchange hash H; there is a symmetric key for each direction:

$$K_{client \to server} = hash(K||H||"C"||session_{id})$$

$$K_{server \to client} = hash(K||H||"D"||session_{id})$$

Regarding the initial IV (Initialization Vector), a distinction is made for each direction:

$$IV_{client \to server} = hash(K||H||"A"||session_{id})$$

$$IV_{server \to client} = hash(K||H||"B"||session_{id})$$

Integrity & Compression

Data integrity is ensured by appending a Message Authentication Code (MAC) to each packet; as the other fields, authentication algorithm is negotiated during the initial handshake.

MAC is calculated as follows:

$$mac = MAC(K, sequence\_number || unencrypted\_packet)$$

In the packet structure, the $sequence\_number$ is not present, since it is implicit. It is initialized to zero for the first packet, and is incremented after every packet; it is never reset, even if keys/algorithms are renegotiated later and wraps around to zero after every 2^{32} packets.

Integrity & Compression (cont'd)

As the other keys, the integrity key is derived from two values: a shared secret K and an exchange hash H; there is an integrity key for each direction:

$$K_{client \rightarrow server} = hash(K||H||"E"||session_{id})$$

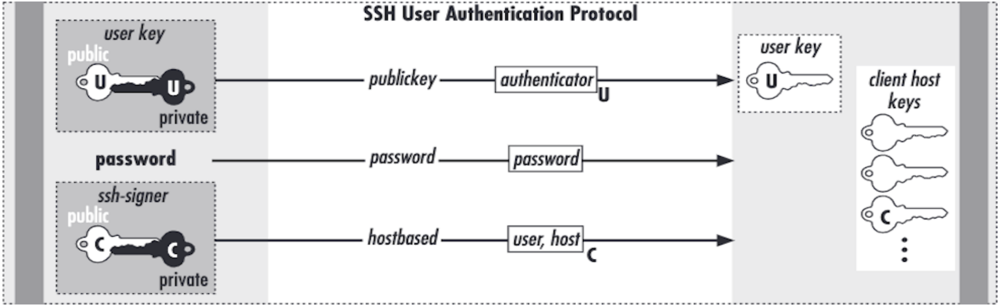
$$K_{server \rightarrow client} = hash(K||H||"F"||session_{id})$$

Supported MAC algorithms are: hmac-sha1, hmac-sha1-96, hmac-md5, hmac-md5-96.

If negotiated during the handshake, **compression** of the payload is possible; the only compression method currently implemented is $zlib$.

Authentication Layer

It provides **client authentication**. The server leads the process, notifying the client of available authentication methods for ongoing exchange.



Let us analyze all available methods.

Password Authentication Method

Regarded as the weakest, this method transmits the password in **plain text** (despite the entire packet being later encrypted by the transport layer).

The usual rules apply to password selection, such as using a complex one with a mix of characters and numbers.

After a fixed amount of time the password expires and Client need to set up a new one.

This method is higly unrecommended since it is vulnerable to password cracking attacks.

Public Key Authentication Method

This method is necessary for every implementation, and it depends on having a private key for authentication.

When a client attempts to authenticate:

1. Client receives a message from the Server and sends a signature s of it, created with his private key.
2. Server then validates whether the key is a legitimate authenticator for the client and checks the signature's validity.
3. If both checks pass successfully, the client's au- thentication request is accepted.

Crucial point: client's public key needs to be manually added to the server's local database; otherwise, authentication will not succeed.

Certificate Authentication Method

The previous method suffers of the key distribution problem, this one eliminates it.

A **certificate** contains not only the public key and the person’s name but also supplementary information like expiration dates and permissions; this data structure receives authentication through signing from a certificate authority (CA).

Now parties exchange their certificates, allowing validation through the shared CA signature without the necessity for the user and server to deploy each other’s public keys.
This method is more secure than public key authentication: in the second method, compromised keys might remain undetected, while the first method’s certificates expire.

Hostbased Authentication Method

In this method Client have a physical host key $host_{key} = (PR_{MAC}, PU_{MAC})$ and use it for authentication.

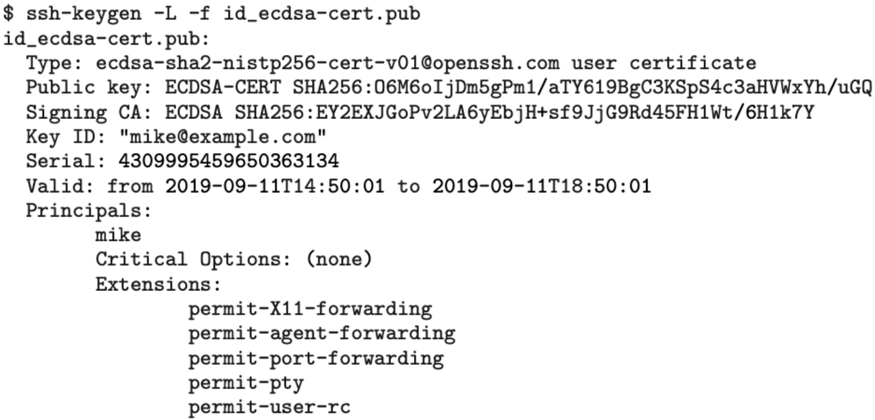
The process is similar to public key auth:

1. Client sends a signature created using the private key of its host (physical machine).
2. Server verifies this signature using the corresponding host's public key.

This method also has a deployment issue, as the $host_{key}$ needs to be manually saved in the server's local database.
Difference with public key authentication is that with this technique we authenticate the **host machine**, while with public one we authenticate the single user.

Certificate Authentication Method (cont’d)

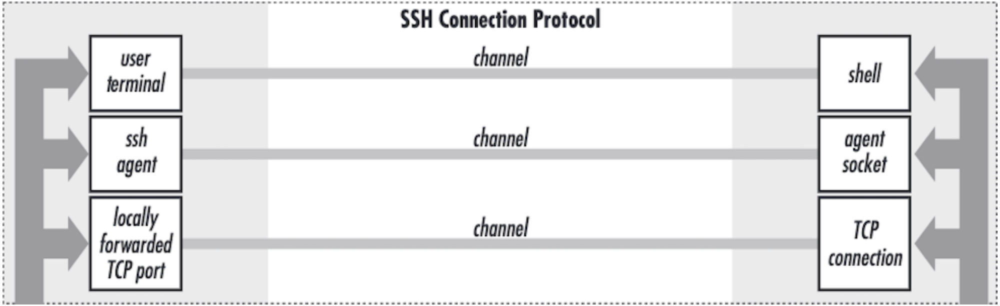
An example of certificate is the following.



```
$ ssh-keygen -L -f id_ecdsa-cert.pub
id_ecdsa-cert.pub:
  Type: ecdsa-sha2-nistp256-cert-v01@openssh.com user certificate
  Public key: ECDSA-CERT SHA256:O6M6oIjDm5gPm1/aTY619BgC3KSpS4c3aHVWxYh/uGQ
  Signing CA: ECDSA SHA256:EY2EXJGoPv2LA6yEbjH+sf9JjG9Rd45FH1Wt/6H1k7Y
  Key ID: "mike@example.com"
  Serial: 4309954596503631 34
  Valid: from 2019-09-11T14:50:01 to 2019-09-11T18:50:01
  Principals:
        mike
        Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

Connection Layer

This protocol provides interactive login sessions, remote command execution, forwarded TCP/IP connections, and forwarded X11 connections. These communication channels are securely multiplexed and transmitted within a single encrypted tunnel.



Functionalities

It’s crucial to recall that this protocol is designed to operate over a secure, authenticated transport. User authentication and protection against network-level attacks are expected to be provided by the underlying protocols. We will present popular operations done within this protocol:

Opening a channel
First of all we must open a new channel; every one is identified with a number at each end, thus we can have different numbers at each end.

Starting a program
Once the session is set up, a program can be launched at the remote end. This program could be a shell, an application, or a subsystem identified by a host-independent name.

Functionalities (cont’d)

Data Transfer

Sender uses send messages and the maximum amount of data allowed is determined by the maximum packet size for the channel and the current window size.

Window size indicates the number of bytes that the other party can send before needing to wait for window adjustment; to modify it, both parties utilize a specific message.

Closing a channel

Once all direction flows have been terminated, each side sends specific message, leading to the closure of the channel.

Now let us focus on the possible attacks to SSH.

Attacks and Vulnerabilities

Attacks SSH can counter (1)

We will first analyze some of the attacks that SSH can counter.

Eavesdropping
In this situation, a malicious actor secretly watches network traffic. SSH **encryption** stops these efforts by making intercepted SSH sessions incomprehensible, thus preserving confidentiality.

Connection Hijacking
The next concern is Connection Hijacking, where attackers can monitor and insert their own traffic into an ongoing TCP connection. SSH can't prevent hijacking because it's a weakness in the underlying TCP, which operates beneath SSH, however, it can spot altered packets by verifying their **integrity**.

Attacks SSH can counter (2)

IP spoofing
In this attack, a malicious actor takes on the identity of a valid host by using its IP address; this can unintentionally lead to connecting to a malicious server, risking the theft of login credentials.

To counter this, SSH meticulously confirms the host’s identity: when a session begins, the SSH client checks the **server’s host key** in its local database and any inconsistencies trigger warnings from SSH and lead to connection termination, protecting the connection’s integrity.

Attacks SSH can counter (3)

Man in the Middle Attack
In this situation, an attacker secretly intercepts and maybe changes the messages between two parties who think they're talking directly. The attacker comes between them, acting like a fake middleman in the conversation.
Let us consider two different scenarios:

First time connection
In this case there is always an instant where the attack can be done. When Server's public key is **not** stored in client's local database, he can decide wheter to add it (using trust) or deny the connection; if client connects for the first time to a new server and accept the host key, it is open to a man-in-the-middle attack.

Man in the Middle (cont’d)

Pre-existing connection
In this scenario, if the password authentication method is not used, SSH is immune to MITM attacks. The attacker perform an active attack in which he carries out separate exchanges with each side, obtaining separate keys of his own with the client and server.

The key exchange is so designed that if he does this, the session identifiers for each side will be diferent; when a client provides a digital signature for either public-key or trusted-host authentication, it includes the session identifier in the data signed. Thus, Eve can’t just pass on the client-supplied authenticator to the server, nor does she have any way of coercing the client into signing the other session ID.

Attacks SSH cannot counter (1)

We will now consider menaces that SSH cannot counter.

Password Cracking
This problem incomes when using simple password based authentication. To counter this problem it is mandatory to create a password that strikes a balance between being user-friendly, memorable, non-obvious to others, and resistant to easy guessing; users must also pay attention to prevent their passwords from being discovered, as mere possession of it grants unauthorized access to their accounts.

It’s preferable to use public key authentication rather than password-based authentication to avoid this problem.

Attacks SSH cannot counter (2)

SSH operates on top of TCP, so it is vulnerable to some attacks against weaknesses in TCP and IP, like **SYN flooding** and **TCP reset**.

Also, it is vulnerable to **Traffic analysis**.
SSH connections are easily identifiable as they generally go to a well-known port and SSH protocol does not incorporate measures to obscure or thwart traffic analysis attempts. Traffic patterns can also indicate backup schedules or times of day most vulnerable to denial-of-service attacks; if there is extended silence on an SSH connection originating from a system administrator's workstation, it might suggest their absence, creating an opportune moment for unauthorized access, either electronically or even physically.

Conclusions

Conclusions

In conclusion, SSH emerges as a powerful tool for managing sensitive information.
Its versatility shines in remote administration tasks, enabling system administrators to control servers, execute commands, transfer files, and establish secure communication channels.
It is a great improvement with respect to ftp and rcp (for file transfers), telnet and rlogin (for remote logins), and rsh (for remote execution of commands) since it can be used to all these tasks while preserving confidentiality, authentication and integrity.

The most important thing to remember is to use robust authentication method to avoid being attacked by some malicious party.

References I

[1] Daniel J Barrett, Richard E Silverman, and Robert G Byrnes. *SSH, The Secure Shell: The Definitive Guide: The Definitive Guide.* " O'Reilly Media, Inc.", 2005.

[2] Mark D. Baushke. *Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH).* RFC 9142. Jan. 2022. DOI: 10.17487/RFC9142. URL: https://www.rfc-editor.org/info/rfc9142.

[3] *CVE-1999-1085.* Available from MITRE, CVE-ID CVE-1999-1085. 1999. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=1999-1085.

[4] *CVE-2012-5975.* Available from MITRE, CVE-ID CVE-2012-5975. 1999. URL: https://www.cve.org/CVERecord?id=CVE-2012-5975.

[5] A. K. Dewdney. "COMPUTER RECREATIONS". In: *Scientific American* 260.3 (Mar. 1989), p. 4.

References II

[6] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Authentication Protocol*. RFC 4252. Jan. 2006. DOI: 10.17487/RFC4252. URL: https://www.rfc-editor.org/info/rfc4252.

[7] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. Jan. 2006. DOI: 10.17487/RFC4254. URL: https://www.rfc-editor.org/info/rfc4254.

[8] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. DOI: 10.17487/RFC4251. URL: https://www.rfc-editor.org/info/rfc4251.

[9] Chris M. Lonvick and Tatu Ylonen. *The Secure Shell (SSH) Transport Layer Protocol*. RFC 4253. Jan. 2006. DOI: 10.17487/RFC4253. URL: https://www.rfc-editor.org/info/rfc4253.

References III

[10] Eric Rescorla. *Diffie-Hellman Key Agreement Method*. RFC 2631. June 1999. DOI: 10.17487/RFC2631. URL: https://www.rfc-editor.org/info/rfc2631.

[11] Alan T. Sherman et al. “Cybersecurity: Exploring core concepts through six scenarios”. In: *Cryptologia* 42.4 (2018), pp. 337–377. DOI: 10.1080/01611194.2017.1362063. eprint: https://doi.org/10.1080/01611194.2017.1362063. URL: https://doi.org/10.1080/01611194.2017.1362063.

[12] Lei Zeng. “A study of accountability in operating systems”. PhD thesis. University of Alabama Libraries, 2014.

Thanks for the attention