The Secure Shell (SSH) Overview

Zanolin Lorenzo

August 23, 2023

Abstract

SSH is a powerful, convenient approach to protecting communications on a computer network. Through secure authentication and encryption technologies, SSH supports secure remote logins, secure remote command execution, secure file transfers, access control, TCP/IP port forwarding and other important features.

Contents

1 Introduction

SSH (Secure Shell) is a widely used cryptographic network protocol for secure remote access and data communication, offering strong encryption, authentication, and integrity. It enables users to remotely control systems (remote login), execute commands, transfer files (using SFTP), and forward TCP/IP connections securely. SSH has become a standard for secure network administration, ensuring confidentiality and protecting sensitive data.

2 Threat model

In this part, we're examining the threat model to understand exactly who the authorized parties are, what valuable things need safeguarding, who could potentially attack, and how they might do it. This helps us build a clear picture of who's involved, what needs protection, and how to anticipate and prevent potential security issues.

- Assets: in this context, the assets refer to all the data that's transmitted over the network. Some of this information is sensitive, like the actual contents of messages, so it needs to be scrambled for protection; but there's also information that's sent as-is, without encryption, so that it can guide the packets to where they need to go.
- Attackers:Attackers might pretend to be innocent coworkers or anyone on the network. They can watch and change the information being sent, and they could have different reasons for doing it, like economical or acting based on their beliefs.
- Attack vectors: as we will see in Section 5, considered attacks are Eavesdropping, IP Spoofing, Connection Hijacking, Man in the Middle, Password Cracking, Traffic analysis and the famous Insertion attack.

3 Security goals

The security objectives are typically described by the *C.I.A.A.A.* paradigm [11], however, achieving this framework is challenging since, as stated in [5], a system can never be completely isolated from intrusions. The only solution is to impose limitations that deny various services offered by the machine (hence the notion that a system is *secure* only when it is powered off and disconnected from the network). Let us analyze the meaning of the acronym C.I.A.A.A.

- Confidentiality: protecting sensitive data from unauthorized access that could compromise the confidentiality of information;
- Integrity: ensuring that modifications are only performed by authorized individuals and through authorized mechanisms;

- Availability: ensuring that information is always accessible and available to authorized personnel;
- Accountability: ensuring that every action or event within the system can be traced uniquely to the entity or user who performed it;
- Authenticity: ensuring that users are who they say they are and that each input arriving at the system came from a trusted source.

4 Security service and implementation

The Secure Shell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network. The first release of this protocol was SSH-1; as described in [1]. The approach was monolithic, encompassing multiple functions in a single protocol and also some algorithms used were vulnerable to attacks, as example DES or CRC32; another important thing is that there was no rekeying in SSH1, thus there was a single key used during all the session. SSH1 is no longer supported. SSH2, which we will consider, is a complete rewriting of the past protocol and it is more secure. SSH2 guarantees:

- Confidentiality: it is obtained using symmetric encryption algorithms;
- Integrity: it is obtained using integrity algorithms;
- Authenticity: of both server and client; it is obtained with password or public key authentication algorithms;
- Access control: it is obtained using Access Control List.
- Accountability: obtained using logging, as said in [12].

As explained in [8], the architecture of this protocol consists of three components:

- Transport Layer protocol: supplies *server* authentication to the client, confidentiality, and integrity; it usually operates over a TCP/IP connection (above ISO/OSI level 4), ensuring secure data exchange between the server and the client. Optionally, SSH can also provide compression to enhance performance during data transmission.
- 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.

In figure 2 it is represented the full stack. We will briefly discuss about each single layer.

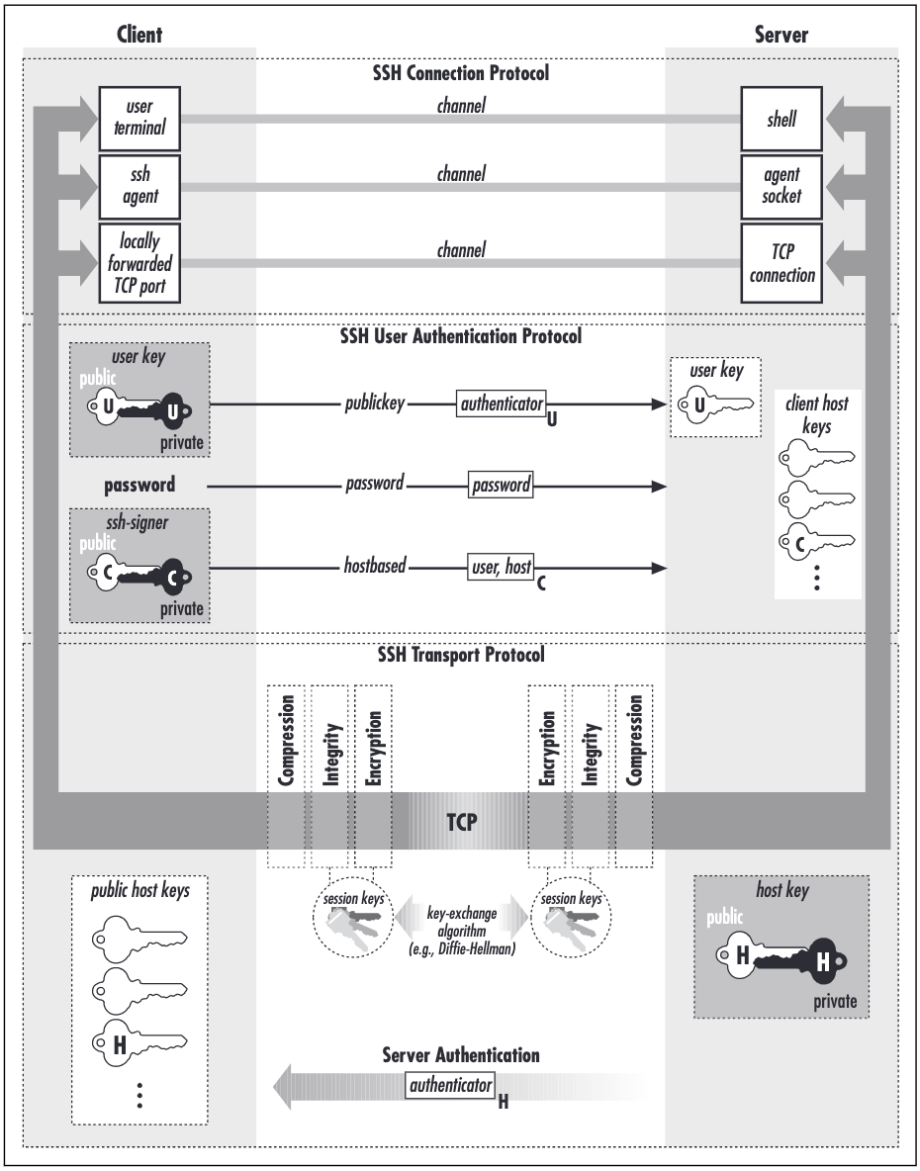

Figure 1: SSH2 Architecture.

4.1 Transport Layer

Let's take a brief look at each point that characterizes this layer.

Initial handshake

In this phase each side sends a name-lists of supported algorithms (encryption, integrity, compression, key exchange), listed in order of preference. After the SSH_MSG_KEXINIT message exchange, the *Diffie-Hellman* key exchange algorithm is run; throughout this process, a negotiation takes place where each party decides on algorithms for each field that are compatible with the other party's requirements and it's computed a key [10].
As reported in [9], Client and Server shares some common values: p is a large safe prime; g is a generator for a subgroup of $GF(p)$ and q is the order of the subgroup. Server authenticates during this process; 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. Finally, Server sends (K_S, f, s) to Client.

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

3. The client verifies whether K_S is the $host_{key}$ for the server by checking if K_S belongs to the `/etc/ssh/known_hosts` file. If it is not present, the client has the option to either add it to the file or terminate the connection. 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.

This process produces two values: a shared secret K and an exchange hash H; these two values will be used later to build each key. Key exchange ends by each side sending an SSH_MSG_NEWKEYS message. Keep in mind that keys will be changed after each gigabyte of transmitted data or after each hour of connection time, whichever comes sooner. However, since the re-exchange is a public key operation, it requires a fair amount of processing power. Supported algorithms for key exchange are summarized in [2]; however as written there, all key exchange methods using the SHA-1 hash are to be considered as deprecated, thus it is preferrable to use another hash function.

Asymmetric key

To begin with, *server authentication* is essential, requiring the server to possess an asymmetric $host_{key}$, represented as a tuple (PR_{srv}, PU_{srv}); it's crucial to note that each service of the server has its own unique $host_{key}$. This asymmetric key pair (private key PR_{srv} and public key PU_{srv}) enables secure server authentication during the symmetric key exchange process. This key is created during the installation of openssh on the server, using $ssh - keygen2$ with RSA/DSA; the total length is 1024 bits and usually the two parts are stored in `/etc/ssh` as `ssh_host_rsa_key` and `ssh_host_rsa_key.pub`. The private one is saved encrypted with a password chosen by the user; however the downside is that the public key needs to be pre-stored on each server beforehand for every connection.

Compression

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

Encryption

Since we want to guarantee *confidentiality*, we need to encrypt our data using symmetric encryption. When encryption is enabled, it is required that the packet length, padding length, payload, and padding fields of each packet are encrypted using the specified algorithm; since it is required robustness, an effective symmetric key has to be long at least 128 bits. As written in [9], the supported encryption algorithms are: 3des-cbc, blowfish-cbc, twofish256-cbc, twofish-cbc, twofish192-cbc, twofish128-cbc, aes256-cbc, aes192-cbc, aes128-cbc, serpent256-cbc, serpent192-cbc, serpent128-cbc, arcfour, idea-cbc, cast128-cbc. As the other keys, the 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:

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

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

Data Integrity

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:

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

Supported *mac* algorithms are: hmac-sha1, hmac-sha1-96, hmac-md5, hmac-md5-96. 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:

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

Packet Structure

Each packet has five fields:

- *packet_length* : length in bytes of the packet, not including *mac*;
- *padding_length*;
- *payload* : array of length $packet\_length - padding\_length - 1$;
- *padding* : of length padding_length. Maximum length is 32768 bytes;
- *mac*

As we can read, there is no explicit sequence_number. The sequence_number 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.

4.2 Authentication Layer

Next, we proceed to analyze the layer that aims to authenticate the client. It is important to remind that when this protocol starts, it receives the session_identifier (represented by the exchange hash H from the first key exchange) from Transport layer. The authentication process is driven by the server, as it informs the client about the available authentication methods that can be used to continue the exchange at any given time. We will analyze each authentication method.

Public Key Authentication Method

As stated in [6], this method is the only one that must be present in every implementation. Using this technique, authentication relies on the possession of a private key. When a client attempts to authenticate, he receives a message from the Server and sends a signature of it, created with his private key; the server then validates whether the key is a legitimate authenticator for the client and checks the signature's validity. If both checks pass successfully, the client's authentication request is accepted. A crucial point to remember is that the client's public key needs to be manually added to the server's `~/.ssh/authorized_keys` file; otherwise, authentication will not succeed. As for the server, private keys are generated using $ssh - keygen2$ and stored in an encrypted form at the client

host, and the user must supply a passphrase before accessing. Using public key cryptography for authentication requires copying the public key from every client to every server that the client intends to log into; this system does not scale well and can be an administrative burden.

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. If the password has expired, the server indicates this by responding with SSH_MSG_USER AUTH_PASSWD_CHANGEREQ, and the user must then send a SSH_MSG_ USERAUTH_REQUEST packet containing both the old and new passwords.

Hostbased Authentication Method

In this method, the client sends a signature created using the private key of its host; the server verifies this signature using the corresponding host's public key. Once the client host's identity is established, authorization is performed based on the user names on both the server and the client, as well as the client host's name. However, further authentication beyond the initial host identity verification is not required in this method. The 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

Certificate authentication eliminates key approval and distribution. Instead of scattering public keys across static files, you bind a public key to a name with a certificate. A certificate serves as a data structure containing 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). This mechanism removes the necessity for the user and server to deploy each other's public keys. Instead, they exchange their certificates, allowing validation through the shared CA signature. It is vital to emphasize that this CA must originate from a highly trusted Certificate Authority. To implement this authentication approach, simply configure clients and hosts to authenticate certificates using the public key of your CA. This method offers better security than public key authentication: in the second one keys are trusted permanently and a compromised private key or illegitimate key binding may go unnoticed or unreported for a long time; in the first one, certificates expire.

4.3 Connection Layer

This protocol offers interactive login sessions, remote execution of commands, forwarded TCP/IP connections, and forwarded X11 connections; to achieve this, all these communication channels are multiplexed and transmitted securely

within a single encrypted tunnel. We will briefly explore each phase of the communication between two terminals. It is important to remeber that this protocol is assumed to run on top of a secure, authenticated transport. User authentication and protection against network-level attacks are assumed to be provided by the underlying protocols.

Opening a Channel

Every single channel is identified with a number at each end, thus we can have different numbers at each end. According to [7], the communication begins with a SSH_MSG_ CHANNEL_OPEN message, which contains of three key fields: the *sender channel* indicating the channel number from the sender's perspective, the *initial window size* representing the allowed bytes of channel data that can be sent without window adjustment, and the *maximum packet size* setting the upper limit for packet size. If the remote side chooses to open the channel, it responds with a SSH_MSG_CHANNEL_OPEN_CONFIRMATION message, containing the *recipient channel* field.

Starting a Program

After the session has been established, a program is initiated at the remote end, which can be a shell, an application program, or a subsystem identified by a host-independent name. The SSH_MSG_CHANNEL_REQUEST message allows for the selection of a program to run on the other side; for instance, to run a shell, the packets will contain the text "shell".

Data Transfer

The window size indicates the number of bytes that the other party can send before needing to wait for window adjustment; to modify the window size, both parties utilize the SSH_MSG_CHANNEL_WINDOW_ADJUST message. After receiving this message, the recipient sends the given number of bytes more than it was previously allowed to send; the window size is incremented. The maximum window size is $2^{32} - 1$ bytes. Finally, to transfer a file, the sender uses SSH_MSG_ CHANNEL_DATA messages; the maximum amount of data allowed is determined by the maximum packet size for the channel and the current window size.

Closing a Channel

When one side has completed its communication, it sends an SSH_MSG_ CHANNEL_EOF message to signal the closure of the channel. No further response is expected after this message. Once all direction flows have been terminated, each side sends an SSH_MSG_ CHANNEL_CLOSE message, leading to the closure of the channel.

5 Attacks and Vulnerabilities

As mentioned in [1], SSH has particular threats against which it is effective and others that it doesn't address; we will briefly analyze all of them.

Attacks SSH can counter

Let us delve into the threats that SSH effectively mitigates, starting with the concept of *Eavesdropping*. In this scenario, an attacker surreptitiously monitors network traffic; however, SSH encryption thwarts such attempts by ensuring that any intercepted SSH sessions remain inscrutable, thus preserving Confidentiality.
Moving forward, let's consider *IP Spoofing*. This form of attack involves a malicious actor assuming the identity of a legitimate host by usurping its IP address; this can inadvertently lead to connecting with a malevolent server, potentially resulting in stolen login credentials. To counteract this, SSH rigorously verifies the host's identity. When a session starts, the SSH client searches the server's host key within his local database. Any discrepancies trigger warnings from SSH and result in connection termination, safeguarding the integrity of the connection.
Next one is *Connection Hijacking* which consists of attackers capable of both monitoring and injecting their traffic into an existing TCP connection. In this case SSH can't prevent hijacking, since this is a weakness in TCP, which operates below SSH; however, SSH can detect modified packets by checking their integrity.
Continuing, we encounter the *Man-in-the-Middle Attack*. This scenario describes an attacker clandestinely intercepting and potentially modifying the communications between two parties who are under the impression that they are engaged in direct communication; the attacker positions himself between these parties, creating a deceptive intermediary role in the exchange. In this case there is always an instant where the attack can be done, that is the initial connection. As we can see in Figure 3, when Server's public key is not stored in *known_host*, client can decide wheter to add it (using trust) or deny the connection; if the 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. Also it is important to limit the use of password authentication since it is vulnerable.

Finally, if public key/hostbased authentication is used and it is not the first time connecting to the server, then SSH is immune to MITM attacks. More specifically, Eve can't discover the session key simply by observing the key exchange; she must perform an active attack in which she carries out separate exchanges with each side, obtaining separate keys of her own with the client and server. The key exchange is so designed that if she 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

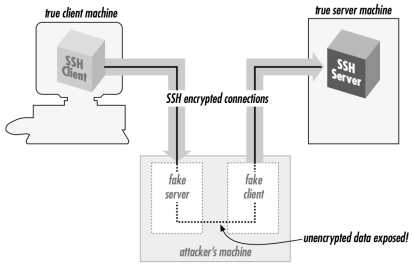

Figure 2: Man in the middle attack.

into signing the other session ID.
Finally, there is the famous *Insertion Attack* which exploited a vulnerability of CRC-32 used in SSH-1 to insert arbitrary data into the plaintext data stream bound for either the client or server [3]. More specifically, the use of encryption algorithms in cbc (cipher block chaining) or cfb (cipher feedback 64 bits) modes with the CRC-32 integrity check allows to perform a known plaintext attack (with as few as 16 bytes of known plaintext) that permits the insertion of encrypted packets with any choosen plaintext in the client to server stream that will subvert the integrity checks on the server and decrypt to the given plaintext, thus allowing an attacker to execute arbitrary commands on the server. SSH-2 makes use of cryptographycally strong message authentication codes for integrity checks that wont fail to these attacks.

Attacks SSH cannot counter

Primarily, a persistent issue emerges concerning *Password Cracking*, particularly in the context of simple password authentication. The challenge resides in creating 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. In fact, it is almost mandatory to use public-key authentication instead, since it is two-factor: a stolen passphrase is useless without the private key file, so an attacker needs to steal both. Also, it was founded a vulnerability when old-style password authentication is enabled, that allows remote attackers to bypass authentication via a crafted session involving entry of blank passwords [4].
SSH operates on top of TCP, so it is vulnerable to some attacks against weaknesses in TCP and IP, such as: *SYN flooding* and *TCP reset*. Conversely, SSH's encryption and host authentication offer formidable protection against attacks involving improper routing, which could potentially expose sensitive data or redirect connections towards compromised servers, and attempts to hijack or manipulate TCP data are thwarted by SSH's detection mechanisms, but they inadvertently lead to the cessation of the SSH connection due to SSH's response to these anomalies.

Continuing, we have *Traffic analysis*. SSH doesn't address traffic-analysis attacks and SSH connections are easily identifiable as they generally go to a well-known port; importantly, the SSH protocol does not incorporate measures to obscure or thwart traffic analysis attempts. As example, a sudden increase in traffic with another company might alert the attacker that an impending business deal is in the works. 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.

6 Conclusions

In conclusion, the paper aim was the exploration into the architecture of SSH and its functionalities. During the analysis of the layers that compose SSH, we have unveiled its robust security mechanisms, ranging from authentication and encryption to integrity checks and protection against a variety of network threats. Through this examination, it becomes apparent that SSH stands as a great tool to use when operating with sensitive informations, guaranteeing both confidentiality and integrity. SSH's versatility is particularly highlighted when considering remote administration tasks; it can be used by system administrators to remotely control servers, execute commands, transfer files, and establish secure communication channels. This dynamic functionality significantly enhances operational efficiency and flexibility, especially when dealing with geographically distributed systems or in scenarios where direct physical access is unfeasible, always guaranteeing security. As already written above, it is important to use robust authentication method to avoid being attacked by some malicious party. 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 and integrity.

A Miscellaneous

An example of execution could be the following.

```
# Login as user lorenzo on host.example.com
$ ssh -l lorenzo@server.example.com

# Run command 'ls -l' on a remote computer after logging into it
$ ssh lorenzo@server.example.com ls -l

# LOCAL Port forwarding: local port 8080 is connected to remote port 80
# SERVER is running on port 80 a web server and the client connects to it using port 8080
$ ssh -L localhost:8080:server.example.com:80 lorenzo@server.example.com

# REMOTE Port forwarding: remote port 8080 is connected to remote port 80
```

```
# CLIENT is running on port 80 a web server and the server connects to it using port 8080
$ ssh -R server.example.com:8080:localhost:80: lorenzo@server.example.com

# Copy local file video.mp4 to path uploads/videos on the server
$ scp video.mp4 lorenzo@server.example.com:uploads/videos
```

An example of certificate could be 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: 4309995459650363134
  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
```

References

[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.

[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.

[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.

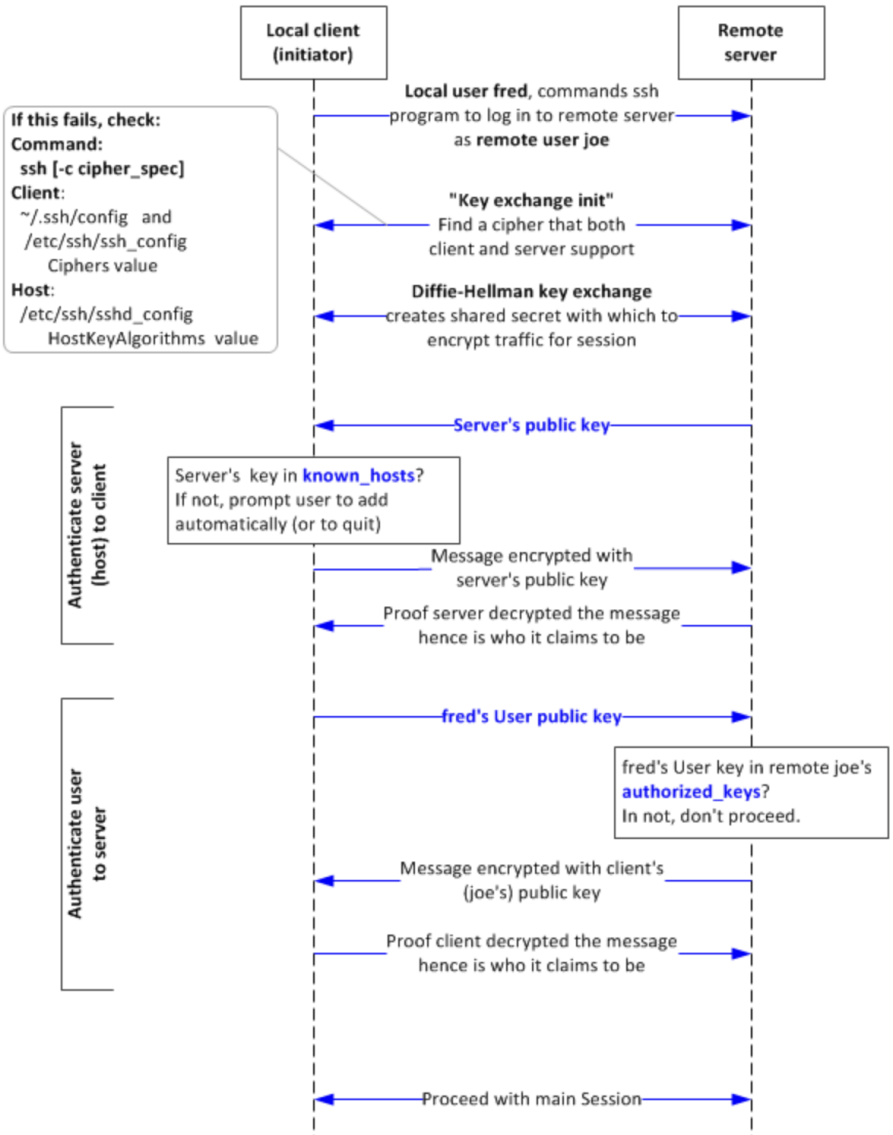

Figure 3: SSH2 Connection steps with public key authentication.