UNIVERSITÉ MOHAMMED V of Rabat

Faculty of Sciences

Université Mohammed V
Faculté des Sciences
Rabat

Department of Computer Science

MSc: Software Engineering and System Security

Report on the Secure Shell Protocol (SSH)

Prepared by:

Bouzid Imane

Achtoun Fatima Zahra

Supervised by Fatima Ezzahra Ziani Professor at the FSR

Academic Year 2024-2025

Contents

1 Introduction and Historical Background **3**

2 Definition **4**

3 Cyptography in SSH **5**

3.1 List of Cryptographic Algorithms used in SSH 5
3.2 Symmetric Cryptography 5
3.3 Asymmetric Cryptography 5
3.4 Cryptographic Hashing 6

4 SSH Architecture **6**

4.1 Overview 7
4.2 Transport Layer Workflow 9
4.2.1 Version Exchange 9
4.2.2 Algorithm Negotiation 10
4.2.3 Generation of Asymmetric Keys 11
4.2.4 Client Transmits Asymmetic Public Key 11
4.2.5 Server Proceeds With Calculations 11
4.2.6 Server Transmits Public Key and Digital Signature 12
4.2.7 Proceeds With Calculations and Server Authentication 12
4.2.8 Switching to Encrypted Communication 13
4.3 Authentication Layer 13
4.4 Connection Layer 14

5 Limitations and Security Challenges of SSH **15**

6 Practical Implementation of SSH: Lab Setup and Analysis **15**

6.1 Tools 15
6.2 Kali Setup 16
6.3 Server Setup 16
6.3.1 Configure OpenSSH 16
6.3.2 Starting The SSH Server 17
6.4 Client setup 18
6.4.1 Key Generation 18
6.4.2 Public Key Uplaod 19
6.4.3 Connecting To the Server: 19
6.5 Capturing SSH Traffic with Wireshark 20
6.5.1 Version Exchange 20
6.5.2 ALgo Negotiation 21

6.5.3 Public key Upload By The Client . . . 22
6.5.4 Server Response . . . 22
6.5.5 Switching To new Keys . . . 23
6.6 Interacting With The Server Remotly: SSH Tunneling . . . 24
6.6.1 Local Tunnel . . . 24
6.6.2 Remote Tunnel . . . 24
6.7 Security Measures . . . 25
6.7.1 Fail2Ban . . . 25
6.7.2 Testing SSH Security with Different Methods . . . 26
6.7.3 SSH Security Hardening Techniques . . . 27
6.8 Automation . . . 27
6.8.1 Bash Script to Automate Tasks . . . 27
6.8.2 Using Ansible . . . 28

7 Conclusion **29**

1 Introduction and Historical Background

Figure 1: Tatu Ylönen

Figure 2: The University of Helsinki

In 1995, a single security breach changed the Internet forever. A researcher at the University of Helsinki (Finland), Tatu Ylönen, discovered that an attacker had intercepted login credentials transmitted over Telnet, which allowed them to gain unauthorized access to a university server. This incident was not just a wake-up call; it exposed a systemic flaw in the way legacy protocols handled authentication and data transmission. It became clear that as networks grew and became more interconnected, the risks of relying on outdated security assumptions were no longer acceptable.

For years, protocols such as Telnet, rlogin, and FTP had been used for remote access and file transfers, but they operated under a dangerous assumption: that network environments were inherently trustworthy. Designed in an era when computer networks were smaller and controlled, these protocols lacked the safeguards necessary for a rapidly expanding, increasingly public Internet. As a result, they transmitted sensitive information—including usernames, passwords, and commands—in plaintext.

This left systems vulnerable to even the simplest forms of attack. Anyone with access to the network, whether a malicious insider or an external adversary, could passively eavesdrop on traffic, steal credentials, and even manipulate active sessions without detection. As organizations and individuals relied more on remote access, the consequences of these weaknesses became impossible to ignore. The need for a secure alternative was urgent, not just for a single institution, but for the future of digital communication itself.

Thus, SSH was born.

2 Definition

Tatu Ylönen introduced SSH in 1995 as a replacement for legacy protocols. It is a cryptographic network protocol designed based on the client-server architecture, with the aim of providing a secure communication channel between the client and the server. The channel's main objective is to ensure confidentiality, authentication, and integrity by leveraging cryptographic techniques.

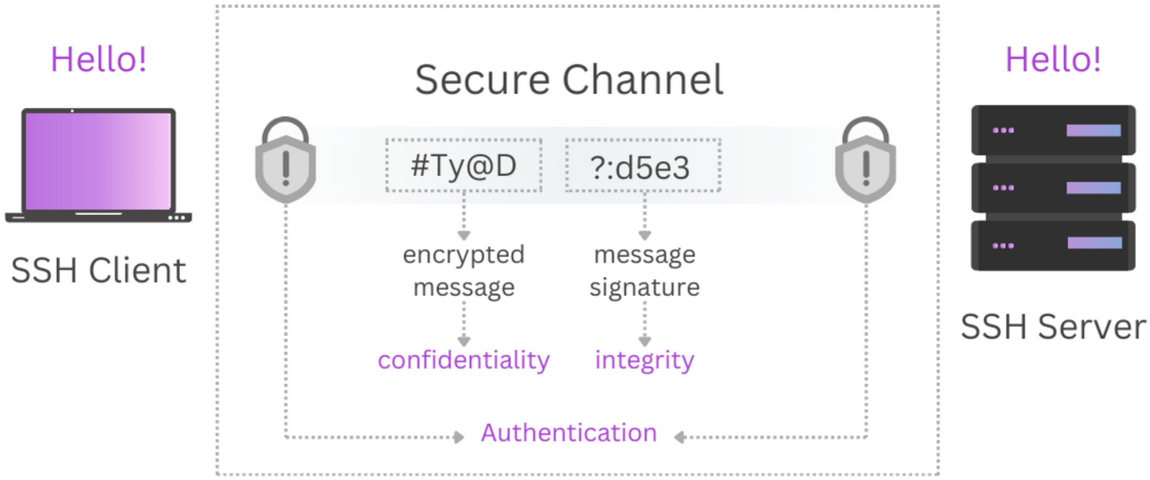


Figure 3: SSH key features

- **Confidentiality**: ensures that sensitive data, such as passwords or commands, remain private and are not readable by unauthorized parties. SSH achieves confidentiality through encryption (symmetric cryptograph).
- **Authentication**: ensures that legitimate communication parties are correctly identified, preventing unauthorized access to systems. One of SSH's authentication methods uses asymmetric cryptography.
- **Integrity**: guarantees that the data exchanged between communication parties is not altered during transmission. SSH ensures integrity through hash functions.

SSH has evolved over time, with SSH-1 laying the foundation and SSH-2 building upon it with refinements that shaped the protocol as we know it today. The following sections will explore SSH-2, the standard in modern implementations.

3 Cyptography in SSH

The Secure Shell (SSH) protocol employs symmetric, asymmetric, and cryptographic hashing techniques to ensure confidentiality, authentication, and integrity. Each type of cryptography serves a specific purpose and contributes to the overall security of SSH.

3.1 List of Cryptographic Algorithms used in SSH

Encryption Type	Algorithm
Asymmetric	RSA
	DSA
	ECDSA
	EdDSA
	Diffie-Hellman (Key Exchange)
Symmetric	AES (AES-128, AES-192, AES-256)
	Blowfish
	3DES (Triple DES)
	ChaCha20
	RC4
Hashing	SHA-1 (Legacy)
	SHA-256
	SHA-512
	MD5 (Used in legacy key fingerprints)

Table 1: Asymmetric, Symmetric, and Hashing Algorithms Used in SSH

3.2 Symmetric Cryptography

Symmetric cryptography is a cryptographic technique where the same key is used for both encryption and decryption. Symmetric algorithms are much faster and more efficient than their asymmetric counterparts, making them ideal for decrypting large amounts of data. Thus, SSH uses symmetric keys to encrypt the entire connection.

3.3 Asymmetric Cryptography

Asymmetric cryptography is a cryptographic system that uses two mathematically related keys: a public key and a private key.

The mathematical relationship between the public key and the private key allows the public key to encrypt messages that can only be decrypted by the private key. This is a one-way ability, meaning that the public key has no ability to decrypt the messages it writes, nor can it decrypt anything the private key may send it. By virtue of this fact, any entity capable decrypting these messages has demonstrated that they are in control of the private key.

The public key can be freely shared with any party, while the private key should be kept entirely secret.

This method is particularly useful when sensitive data must not be transmitted directly over the network. Instead of sending confidential information explicitly, one party shares its public key, allowing the other to verify or derive necessary secrets securely. Thus, SSH utilizes asymmetric encryption in a few different places. During the initial key exchange process used to set up the symmetrical keys and for authentication.

3.4 Cryptographic Hashing

Another form of data manipulation that SSH takes advantage of is cryptographic hashing. Cryptographic hash functions are methods of creating a succinct "signature" or summary of a set of information. Their main distinguishing attributes are that they are never meant to be reversed, they are virtually impossible to influence predictably, and they are practically unique.

Using the same hashing function and message should produce the same hash; modifying any portion of the data should produce an entirely different hash. A user should not be able to produce the original message from a given hash, but they should be able to tell if a given message produced a given hash.

Given these properties, hashes are mainly used for data integrity purposes and to verify the authenticity of communication.

4 SSH Architecture

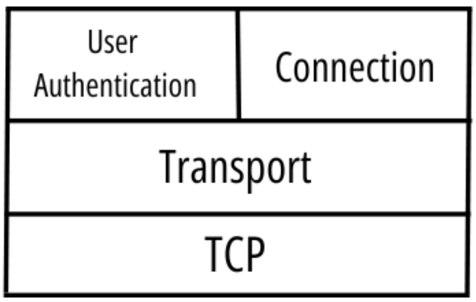


Figure 4: SSH Layer Model

The SSH protocol is structured into three distinct layers that typically run on top of TCP. Each layer serves a specific purpose in the secure communication process, ensuring modularity, scalability, and robust security.

4.1 Overview

Transport Layer The Transport Layer is responsable for establishing the secure channel between the client and for server authentication.
The key steps in this layer are as follows:

- **Version Exchange:** The client and server exchange their supported SSH protocol versions to ensure compatibility.
- **Algorithm Negotiation:** Once the protocol version is agreed upon, the next step is to negotiate the cryptographic algorithms that will be used during the session. These algorithms define how encryption, integrity verification, and authentication will be handled.
- **Key Exchange for Shared Secret Calculation:**
  - **Problem:** Encryption requires a key, and both sides need the same key for symmetric cryptography to work. However:
    - Sending the key over the network exposes it to interception.
    - Generating keys independently results in mismatched keys, making secure communication impossible.
  - **Solution:** This problem is solved by using asymmetric cryptography to establish a shared secret without transmitting it directly over the network. This shared secret is later used to derive session keys for secure communication.
- **Session Key Derivation:** These keys are symmetric and are used for:
  - Encrypting and decrypting data.
  - Generating and verifying message authentication codes (MACs) to ensure data integrity.

  Session keys are derived from the shared secret using a key derivation function (KDF).
- **Server Authentication (on the Client Side):** the server sends a digital signature that can be used by the client to verify its authenticity.

Independent Processing: Due to the nature of the SSH handshake:

- The server receives the client's public key first and proceeds to compute the shared secret, derive session keys, and prepare its response (its public key and signature).
- The client goes through afterward shared key calculation, session key deriviation, and server authentication.
- this design reduces the number of round trips required, as the server can send its public key, signature, and other necessary information in a single message.

User Authentication Layer The User Authentication Layer verifies the identity of the user attempting to access the server. This ensures that only authorized users can establish a session. The key steps in this layer are:

- **Choosing an Authentication Method:** The client selects an authentication method (e.g., password, public-key, or keyboard-interactive).
- **Verifying Identity:** The server validates the user's credentials based on the chosen method.
- **Granting Access:** If the authentication succeeds, the server grants access to the requested resources.
- These steps are designed to be flexible, allowing for multiple authentication methods while ensuring strong security.

Connection Layer The Connection Layer manages the actual communication between the client and server once the secure channel and user authentication are established. It provides mechanisms for multiplexing multiple logical channels over a single SSH connection. The key aspects of this layer include:

- **Channel Establishment:** Logical channels are created for different types of communication (e.g., shell sessions, file transfers, or port forwarding).
- **Data Transmission:** Encrypted data is transmitted over these channels, ensuring confidentiality and integrity.
- **Session Management:** The layer handles session initiation, termination, and reconnection as needed.
- By supporting multiple channels within a single connection, the Connection Layer enhances efficiency and flexibility in communication.

4.2 Transport Layer Workflow

This section is dedicated to a detailed walk trough of the channel establishment and server authentication.

Part 1 of the SSH handshake:

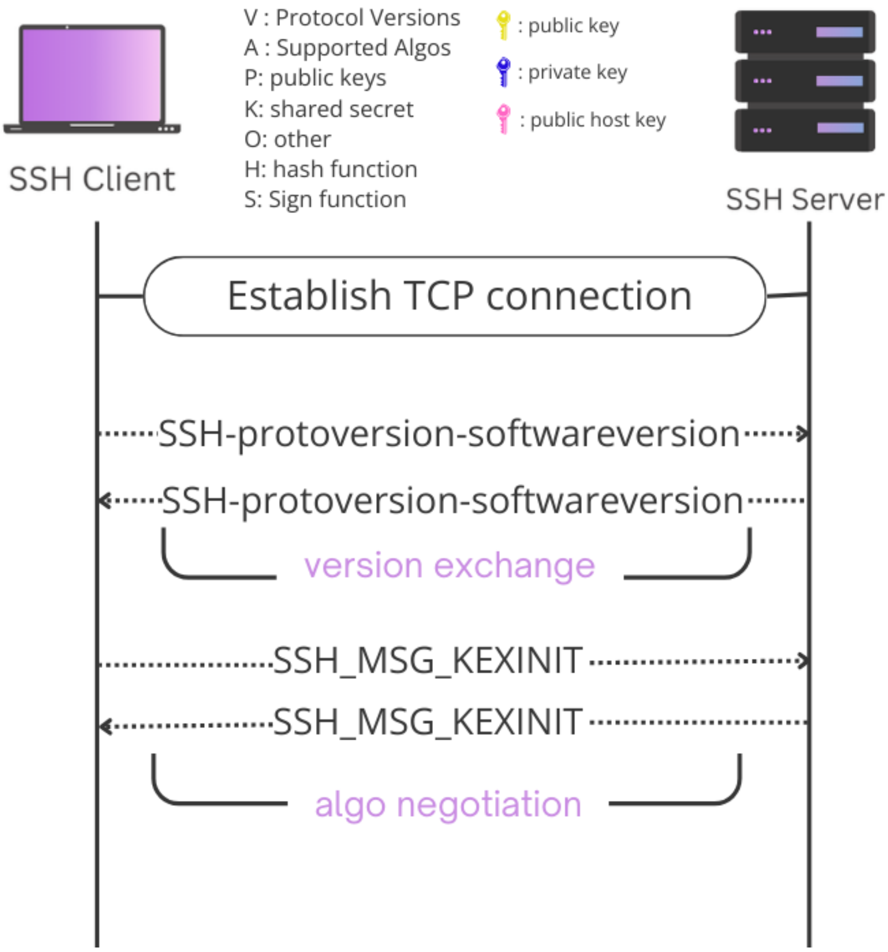


Figure 5: SSH handshake Part 1

4.2.1 Version Exchange

After a successful TCP connection:

- The client sends its supported SSH protocol version string (e.g., `SSH-2.0-OpenSSH_8.9`) to the server.
- The server responds with its supported version string.
- If the versions are compatible, the connection proceeds; otherwise, it terminates.

4.2.2 Algorithm Negotiation

- The client sends a list of supported cryptographic algorithms, ranked in order of preference. This includes:
  - Key exchange algorithms (e.g., Diffie-Hellman, ECDH).
  - Encryption algorithms (e.g., AES, ChaCha20).
  - Message authentication algorithms (e.g., HMAC-SHA256).
  - Compression algorithms (optional).
- The server selects the strongest mutually supported algorithms based on the client's preferences.
- Both parties confirm the selected algorithms.

Part 2 of the SSH handshake:

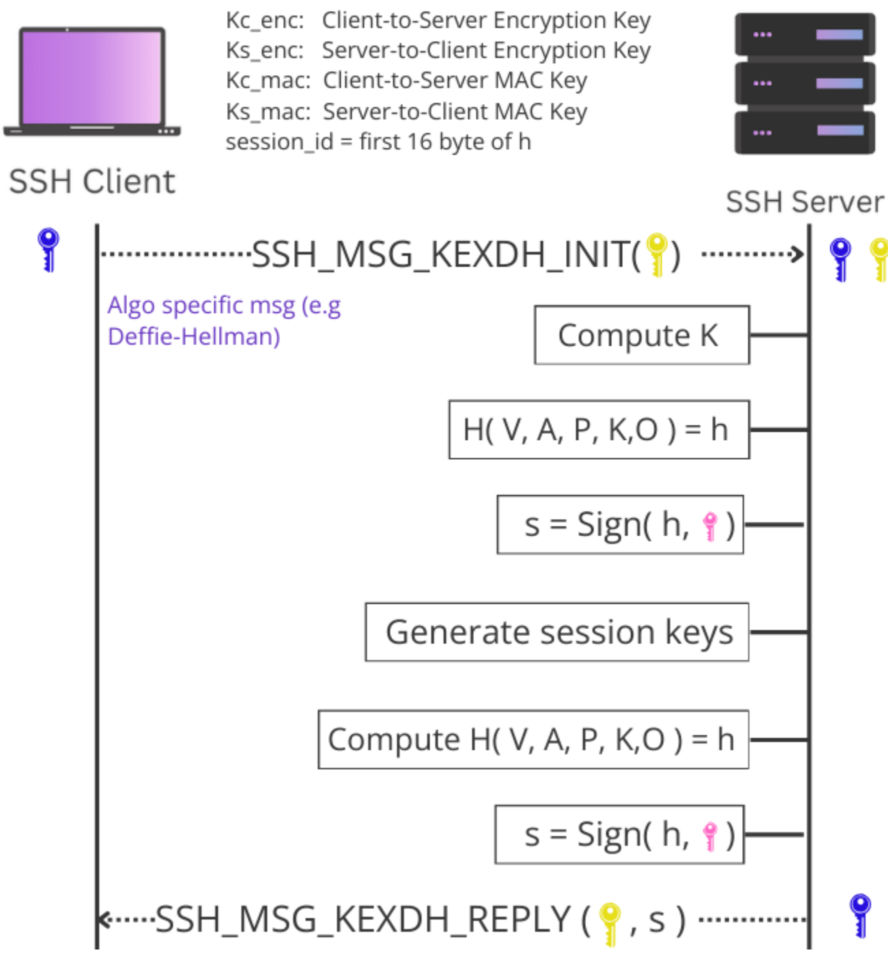


Figure 6: SSH handshake Part 2

4.2.3 Generation of Asymmetric Keys

After algorithm negotiation:

- The client generates a public-private key pair for the agreed-upon key exchange algorithm (e.g., Diffie-Hellman or ECDH).
- The server also generates its public-private key pair for the same key exchange algorithm.

4.2.4 Client Transmits Asymmetic Public Key

- The client sends its public key (e) to the server in an `SSH_MSG_KEXDH_INIT` message.

4.2.5 Server Proceeds With Calculations

Upon receiving the client's public key (e), the server:

- Computes the shared secret K using its private key and the received public key (e).
- Computes the hash H:

  $$H = \text{hash}(V_C||V_S||I_C||I_S||K_S||e||f||K)$$

  Where:

  - V_C and V_S: Protocol version strings.
  - I_C and I_S: `SSH_MSG_KEXINIT` messages.
  - K_S: Server's public host key.
  - e: Client's public key.
  - f: Server's public key.
  - K: Shared secret.
- Signs the hash H using its private host key:

  $$s = \text{Sign}(H, \text{private\_host\_key})$$

- Derives session keys independently using the shared secret K and the session ID (first 16 bytes of H).

4.2.6 Server Transmits Public Key and Digital Signature

The server sends a `SSH_MSG_KEXDH_REPLY` message containing:

- Its public key (f).
- The signature s.

Part 3 of the SSH handshake:

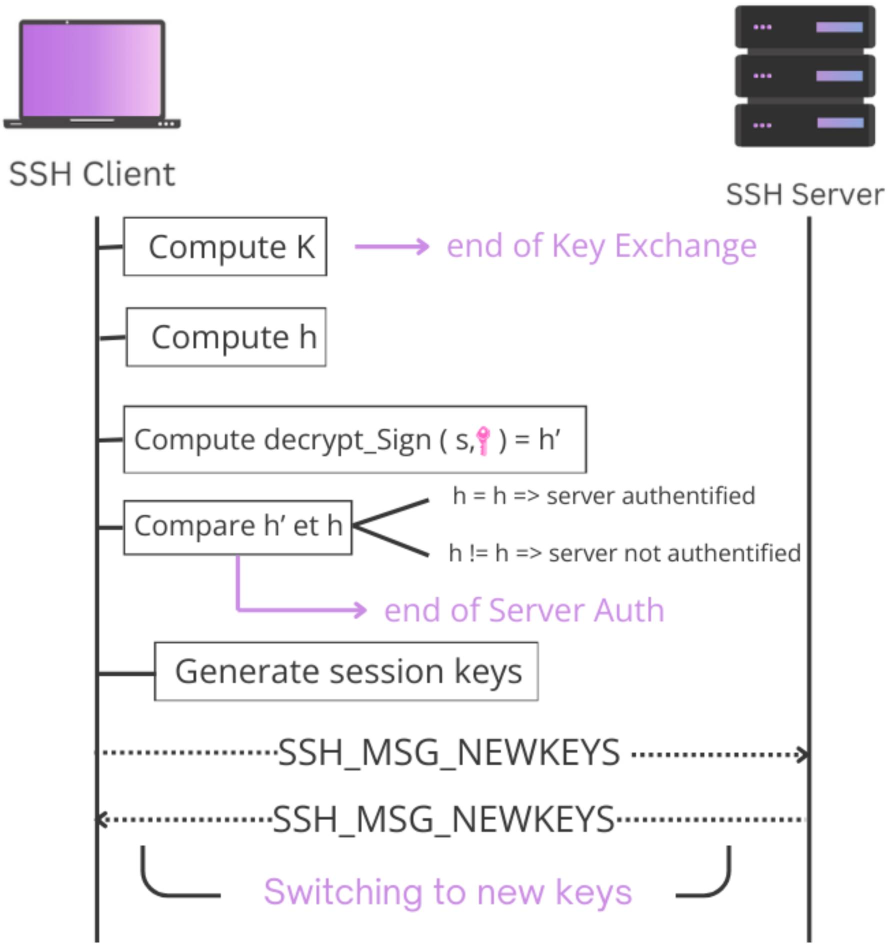


Figure 7: SSH handshake Part 2

4.2.7 Proceeds With Calculations and Server Authentication

Upon receiving the server's public key (f) and signature s, the client:

- Computes the shared secret K using its private key and the server's public key (f).
- Computes the same hash H as the server:

$$H = \text{hash}(V_C||V_S||I_C||I_S||K_S||e||f||K) = h$$

- decrypt the server's signature s using the server's public host key:

$$\text{decryptc\_sign}(s, \text{public\_host\_key})$$

- If the signature is valid (the result = h), the server is authenticated.
- finally the client derives session keys using the shared secret K and the session ID (first 16 bytes of H).

4.2.8 Switching to Encrypted Communication

To transition to encrypted communication:

- The client sends an `SSH_MSG_NEWKEYS` message to indicate it is ready to use the newly derived session keys.
- Upon receiving the client's `SSH_MSG_NEWKEYS`, the server responds with its own `SSH_MSG_NEWKEYS` message.
- From this point onward, all communication is encrypted using the agreed-upon algorithms and session keys.

4.3 Authentication Layer

The authentication layer verifies the identity of the client attempting to access the server. SSH supports multiple authentication methods, including:

- **Public Key Authentication**:
  - The client generates a public-private key pair (e.g., RSA, ECDSA, or Ed25519).
  - The public key is uploaded to the server and stored in the /.ssh/authorized_keys file of the target user account.
  - The private key is kept securely on the client's machine.
  - When the client connects to the server, the server challenges the client to prove ownership of the private key corresponding to the public key stored in authorized_keys by sending an encrypted message using that public key.
  - if the decrypted message sent by the client matches the original the client is authenticated.
- **Password Authentication**: The client provides a password, which is securely transmitted to the server for verification. For instance, a user might log in with their system password.

- **Keyboard-Interactive Authentication**: The server prompts the client for one or more authentication factors, such as a password and a one-time code. For example, a two-factor authentication system might use this method.

Public Key Authentication Workflow

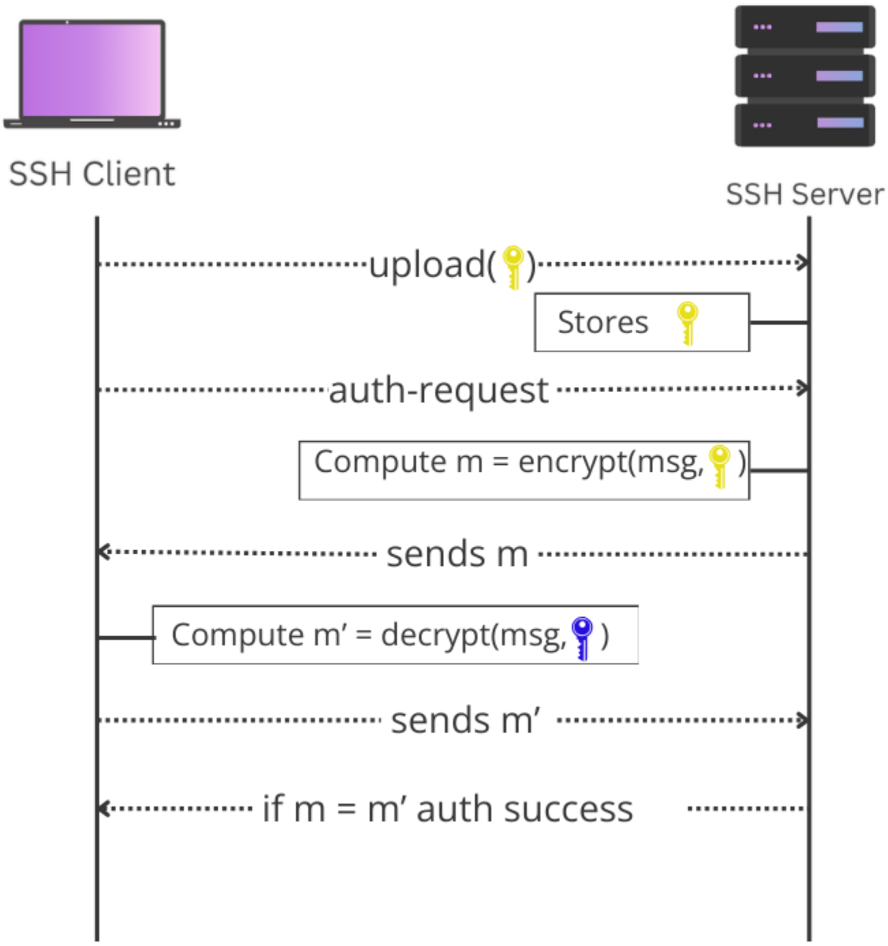


Figure 8: Public Key Authentication

4.4 Connection Layer

The Connection Layer enables multiplexing of multiple logical channels over a single SSH connection.
These channels support various functionalities, including:

- **Remote Command Execution**: Executing commands on the remote server. For example, a system administrator might use SSH to run diagnostic commands on a remote machine.

- **File Transfers**: Secure file transfer protocols like SFTP and SCP. For instance, a developer might use SFTP to upload code to a remote server.

- **Port Forwarding**: Tunneling of network traffic through the SSH connection. For example, a user might forward a local port to a remote database server for secure access.

5 Limitations and Security Challenges of SSH

Despite its robustness, SSH is not without limitations. Key challenges include:

- **Weak Cryptographic Algorithms**: The use of outdated algorithms like RSA-1024 or SHA-1 can compromise security. For example, RSA-1024 is vulnerable to brute-force attacks with modern computing power.
- **Man-in-the-Middle (MITM) Attacks**: Improper verification of server host keys can enable MITM attacks. For instance, if a client blindly accepts a server's host key, an attacker could impersonate the server.
- **Quantum Computing Threat**: Advances in quantum computing could render current cryptographic algorithms obsolete, necessitating the adoption of quantum-resistant alternatives. For example, RSA and ECDSA might be broken by quantum computers in the future.
- **Configuration Errors**: Misconfigurations, such as enabling weak encryption protocols or allowing password-based authentication, can introduce vulnerabilities. For instance, allowing password authentication increases the risk of brute-force attacks.

6 Practical Implementation of SSH: Lab Setup and Analysis

In this lab, simulates and visualizes client-server communication over the SSH protocol. We will set up an SSH server and client using virtual machines (Kali Linux machines), install OpenSSH, and analyze network traffic with Wireshark to observe the encrypted communication between the two systems.

6.1 Tools

We will utilize three primary tools: Kali Linux , OpenSSH , and Wireshark .

- **Kali Linux**: A powerful penetration testing distribution that serves as the operating system for both the client and server machines. Provides a robust environment for configuring and testing SSH setups, including server deployment and client connections.

- **OpenSSH**: An open-source implementation of the SSH protocol. Includes the ssh client for initiating connections and the sshd daemon for managing incoming SSH sessions on the server side.

- **Wireshark**: A network protocol analyzer used to capture and inspect SSH traffic in real-time. Allows us to visualize the encrypted communication between the client and server. Despite the encryption, Wireshark helps identify the structure of SSH packets and confirms that sensitive information remains protected during transit.

6.2 Kali Setup

Before installing any new software, Kali Linux system must be updated.

```
$ sudo apt update
```

Next, install the OpenSSH server application:

```
$ sudo apt install openssh-server
```

Finally, install the OpenSSH client application:

```
$ sudo apt install openssh-client
```

6.3 Server Setup

6.3.1 Configure OpenSSH

To configure the default behavior of the OpenSSH server application, sshd, edit the file /etc/ssh/sshd_config.

```
$ sudo nano /etc/ssh/sshd_config
```

We specify Diffie-Hellman (DH) Algorithm for key exchange by adding this line:
KexAlgorithms diffie-hellman-group-exchange-sha256, diffie-hellman-group14-sha256, diffie-hellman-group16-sha512
To the configuration file.

Optionally, for enhanced security we can:

- Change the default port 22 to a custom port (e.g. 2222):

  ```
  Port 2222
  ```

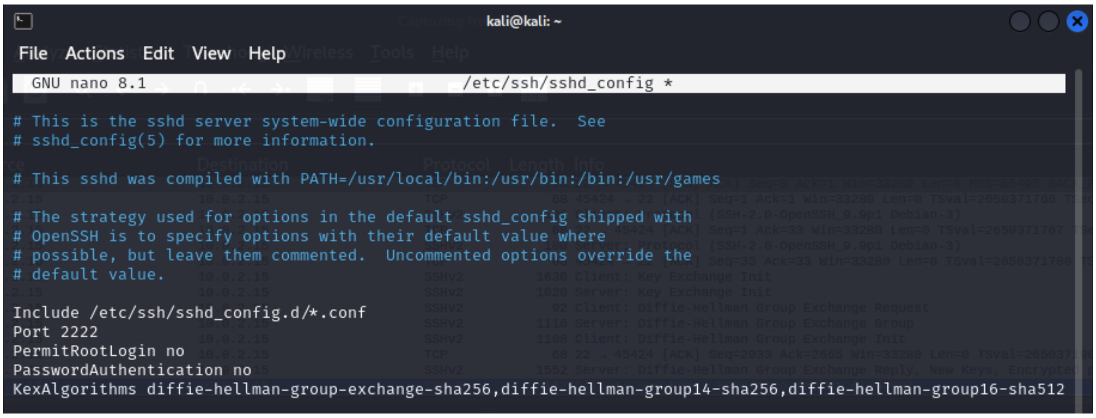
- Disabled direct root login:

```
PermitRootLogin no
```

- Disabled password authentication (to use SSH keys only):

```
PasswordAuthentication no
```

Here's a snapshot of the config file on the virtual machine



6.3.2 Starting The SSH Server

Starting the SSH Server:

```
$ sudo systemctl start ssh
```

If any new changes are made to the config file, restart the ssh server (Optional)

```
$ sudo systemctl restart ssh
```

If you want the SSH server to start automatically at system boot (Optional)

```
$ sudo systemctl enable ssh
```

To verify the server status:

```
$ sudo systemctl status ssh
```

If the SSH service is running correctly, you will be prompted with:

- **Active: active (running)**

Otherwise, in the case of failed to start:

- **Active : failed (Result: exit-code).**

Here's a snapshot of the server status on the virtual machine

```
┌──(kali㉿kali)-[~]
└─$ sudo systemctl status ssh

● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; preset: disabled)
     Active: active (running) since Sat 2025-02-15 11:31:48 EST; 3s ago
 Invocation: e2b23973fa724b5988e17afd7cdf9ed8
       Docs: man:sshd(8)
             man:sshd_config(5)
    Process: 2842 ExecStartPre=/usr/sbin/sshd -t (code=exited, status=0/SUCCESS)
   Main PID: 2844 (sshd)
      Tasks: 1 (limit: 2269)
     Memory: 1.1M (peak: 1.4M)
        CPU: 55ms
     CGroup: /system.slice/ssh.service
             └─2844 "sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups"

Feb 15 11:31:48 kali systemd[1]: Starting ssh.service - OpenBSD Secure Shell server...
Feb 15 11:31:48 kali sshd[2844]: Server listening on 0.0.0.0 port 2222.
Feb 15 11:31:48 kali sshd[2844]: Server listening on :: port 2222.
Feb 15 11:31:48 kali systemd[1]: Started ssh.service - OpenBSD Secure Shell server.
```

6.4 Client setup

6.4.1 Key Generation

```
$ ssh-keygen -t rsa -b 4096
```

- **ssh-keygen:** The ssh-keygen command is used to generate SSH key pairs for authentication.
- **-t rsa:** Specifies the type of key to generate. In this case, you are generating an RSA key. RSA (Rivest–Shamir–Adleman) is a widely supported and well-established cryptographic algorithm.
- **-b 4096:** Specifies the bit length of the key.

key-pair generation on the virtual machine:

```
┌──(kali㉿kali)-[~]
└─$ ssh-keygen -t rsa -b 4096
Generating public/private rsa key pair.
Enter file in which to save the key (/home/kali/.ssh/id_rsa): /home/kali/.ssh/id_rsa
Enter passphrase for "/home/kali/.ssh/id_rsa" (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/kali/.ssh/id_rsa
Your public key has been saved in /home/kali/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:2r68ghA5IMeC474KYIH9tpJZ3cRk0Dtgq2awGlG9gAI kali@kali
The key's randomart image is:
+---[RSA 4096]----+
|Eo .  .oo        |
|X.= . o+.        |
|=B.o o oo.       |
|..*....oo        |
|oo =+.. S.       |
|+.o=+. o         |
|.o=+...  .       |
|o. .. .o         |
|o     .=o        |
+----[SHA256]-----+
```

6.4.2 Public Key Uplaod

After the key-pair generation, the client must upload its public key to the ssh server:
If the server is listening on the default port:

```
$ ssh-copy-id user@host
```

Otherwise, If the server is listening on a custom port:

```
$ ssh-copy-id -p port user@host
```

Public Key upload on the virtual machine:

```
┌──(kali㉿kali)-[~]
└─$ ssh-copy-id -p 2222 kali@10.0.2.15
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/kali/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already insta
lled
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the ne
w keys

Number of key(s) added: 1

Now try logging into the machine, with: "ssh -p 2222 'kali@10.0.2.15'"
and check to make sure that only the key(s) you wanted were added.
```

6.4.3 Connecting To the Server:

Finnally, the client can connect to the server:

```
$ ssh user@host
```

Otherwise, If the server is listening on a custom port:

```
$ ssh -p port user@host
```

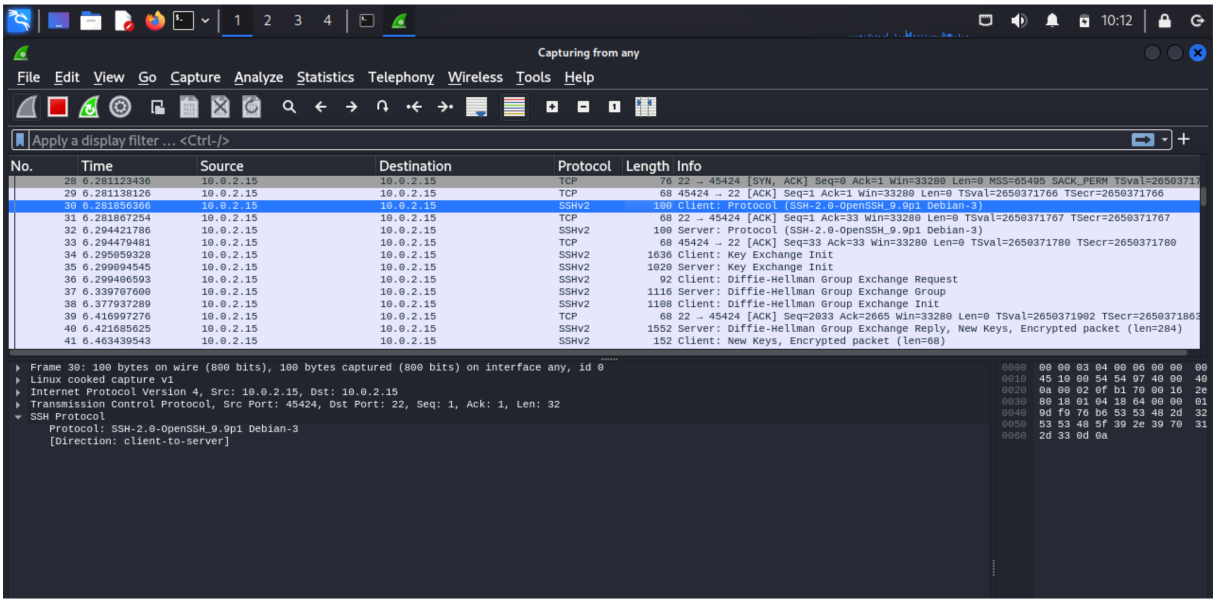
Successfull conection:

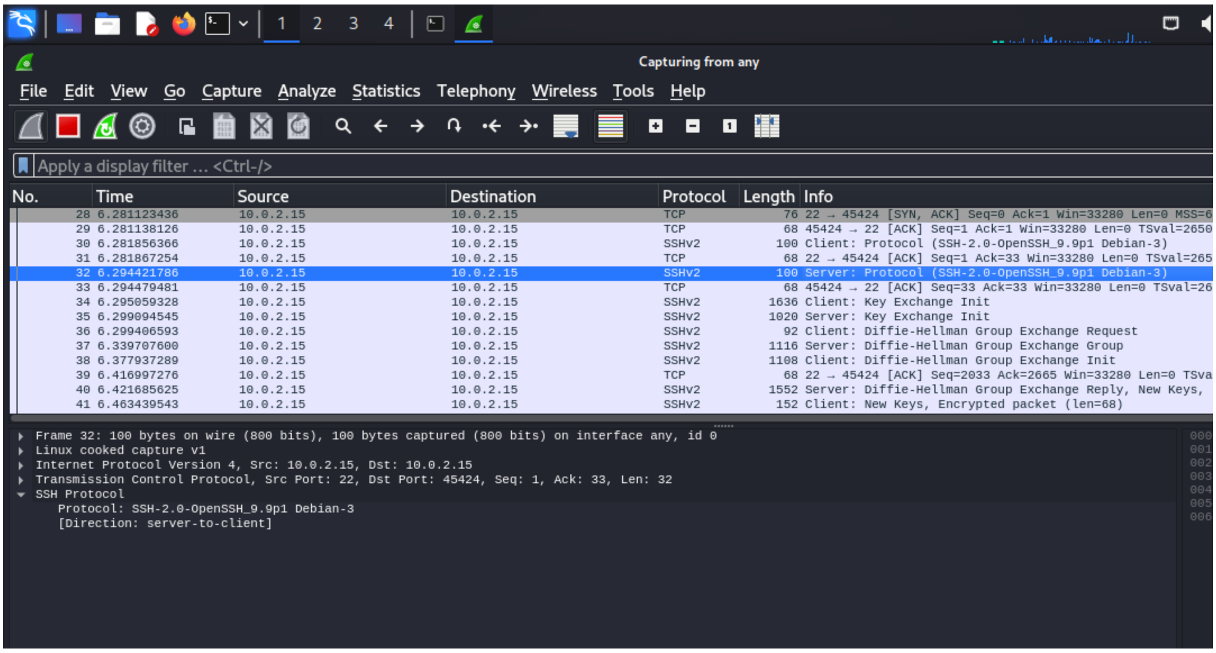
```
┌──(kali㉿kali)-[~]
└─$ ssh -p 2222 kali@10.0.2.15
Linux kali 6.8.11-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.8.11-1kali2 (2024-05-30) x86_64

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Feb 15 09:28:00 2025 from 10.0.2.15
```

6.5 Capturing SSH Traffic with Wireshark

When the client connects to the server by runnig the ssh user@host command (or any equivalent SSH command like ssh-copy-id). The SSH handshake kicks-off:

6.5.1 Version Exchange





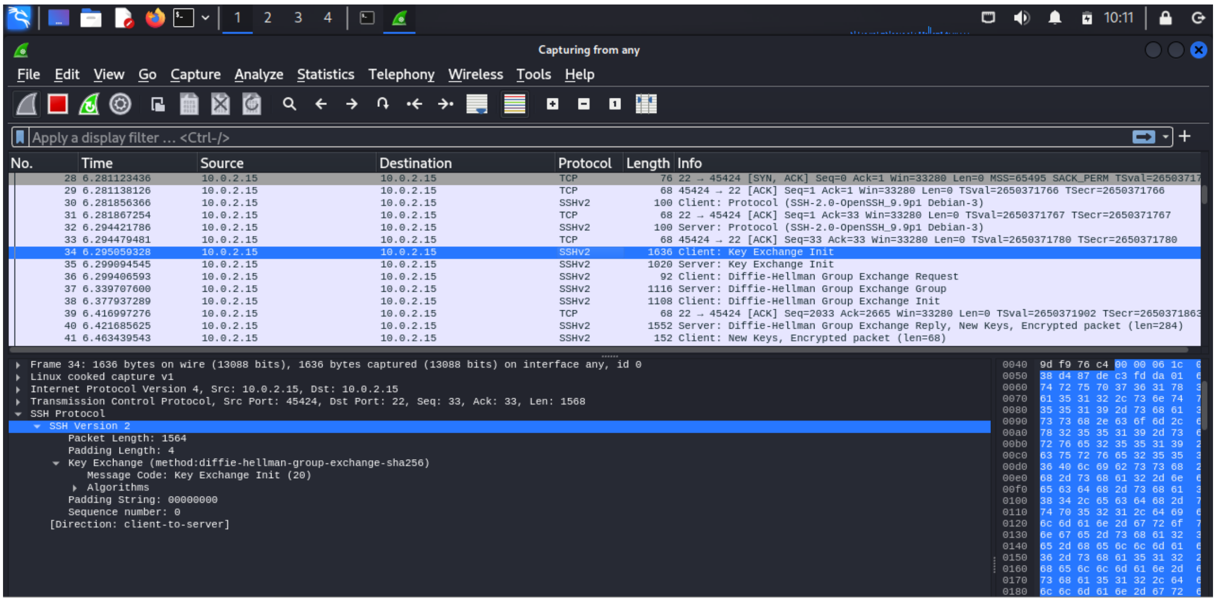
Once the TCP connection is established, the client and server exchange their supported SSH protocol versions.

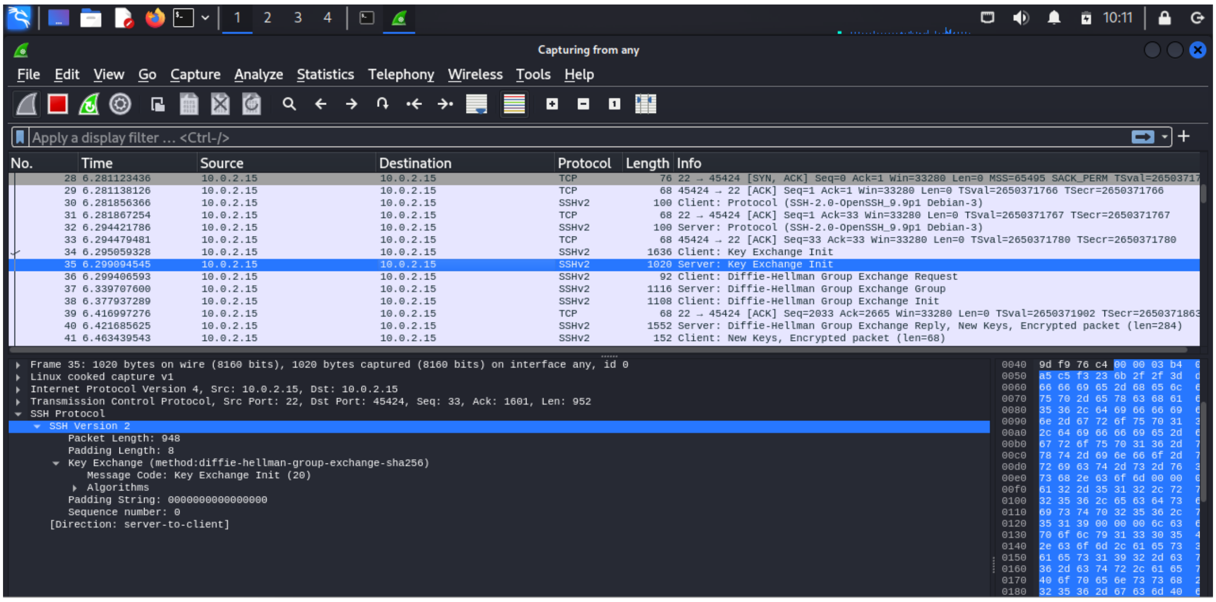
- **Client sends: SSH-2.0-OpenSSH_9.9p1 Debian-3**

- **Server responds: SSH-2.0-OpenSSH_9.9 Debian-3**

- Both are compatible

6.5.2 ALgo Negotiation





The client and server negotiate the cryptographic algorithms they will use for:

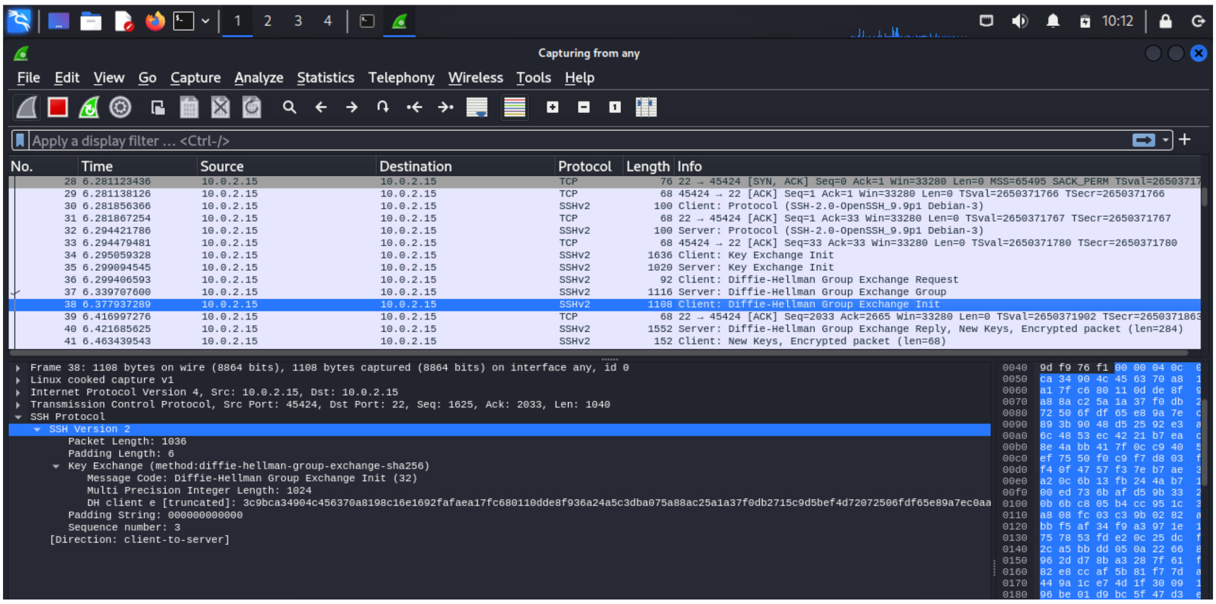
- **Key exchange:** diffie-hellman-group-exchange-sha256

- **Encryption:** aes256-gcm@openssh.com

- **Authentication:** public key

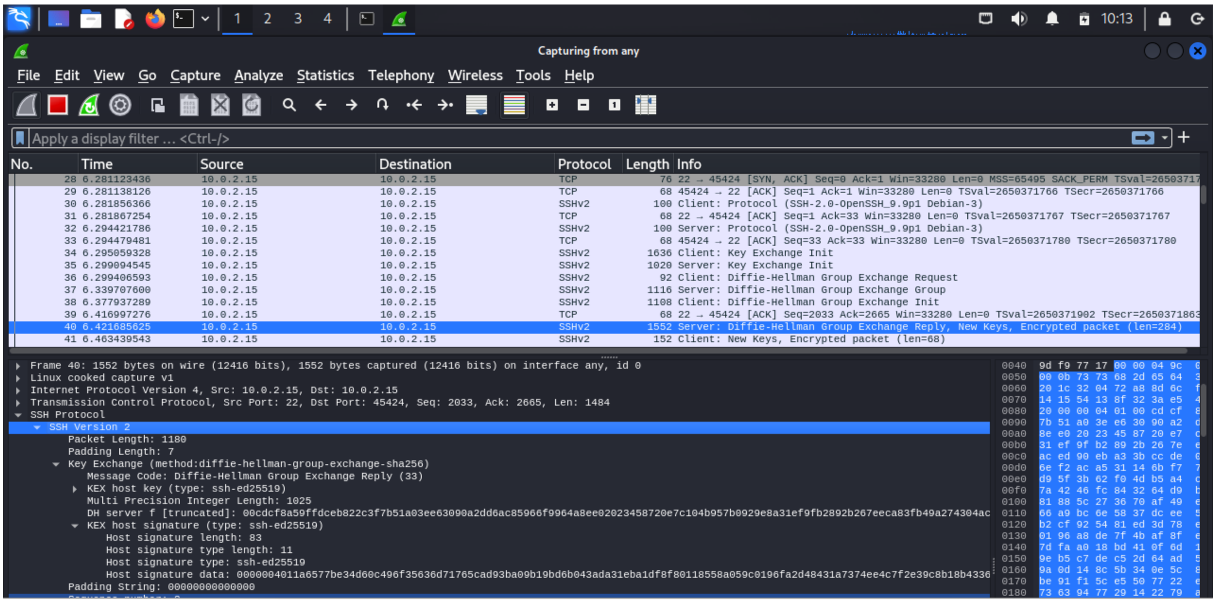
- **Integrity checks:** HMAC-SHA2-256

6.5.3 Public key Upload By The Client



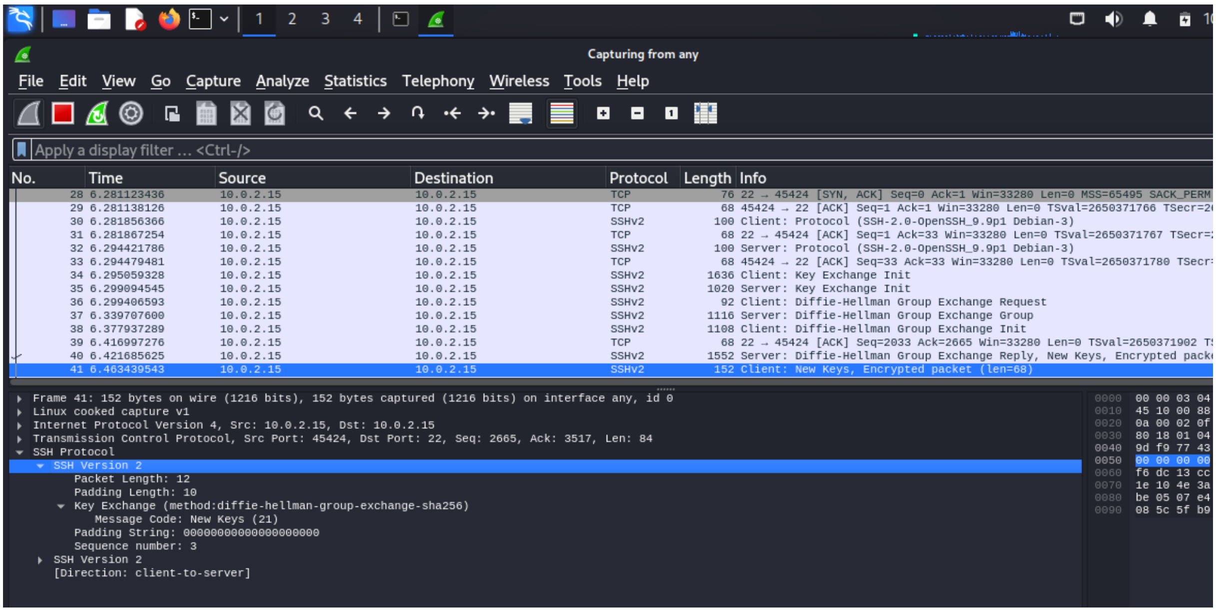
The client generates the asymetric key pairs using the agreed-upon algorithm (e.g., Diffie-Hellman) to securely establish a shared secret. Then Send the public key (e) to the server.

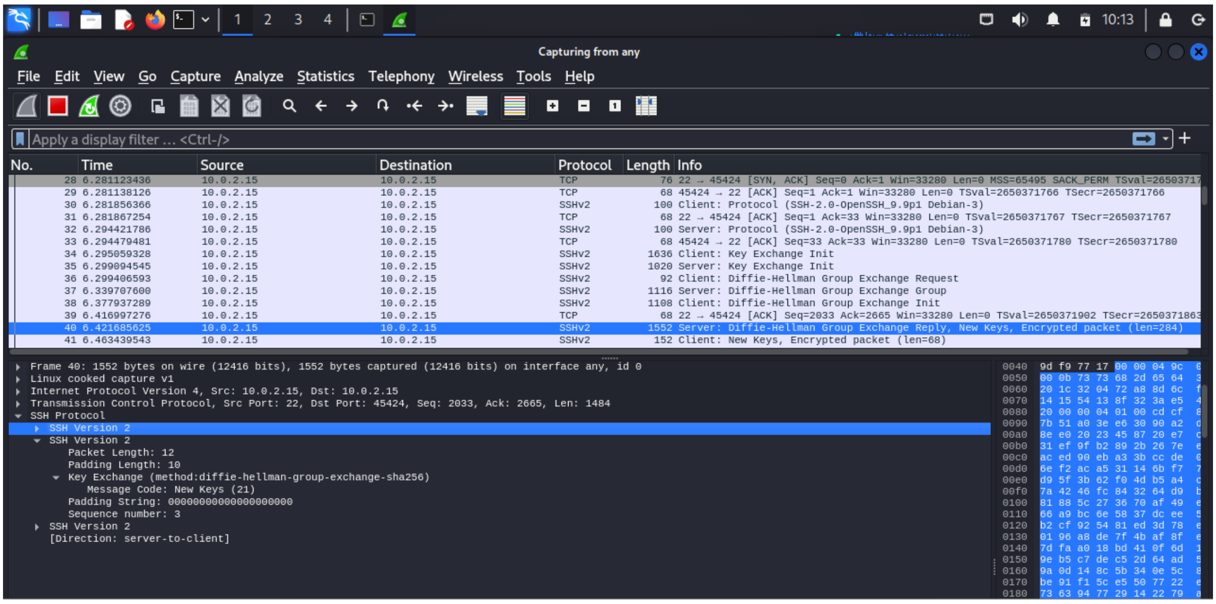
6.5.4 Server Response



Upon receive, the server generates its own asymetric key pairs using the same algorithm (e.g., Diffie-Hellman) and proceeds with calculation of the shared secret and digital signature it also generates the session keys. And send its public key and signature to the client.

6.5.5 Switching To new Keys





Upon receive, the client performs its own calculations, and if the server is authenticated, it sends a SSH_MSG_NEWKEYS message to switch to new keys. The server already sent its own SSH_MSG_NEWKEYS after sending the digital signature and public key.

6.6 Interacting With The Server Remotly: SSH Tunneling

6.6.1 Local Tunnel

We created a local tunnel to forward a local port to a remote port. For example, to access a remote web service on port 80 via local port 8080, we used:

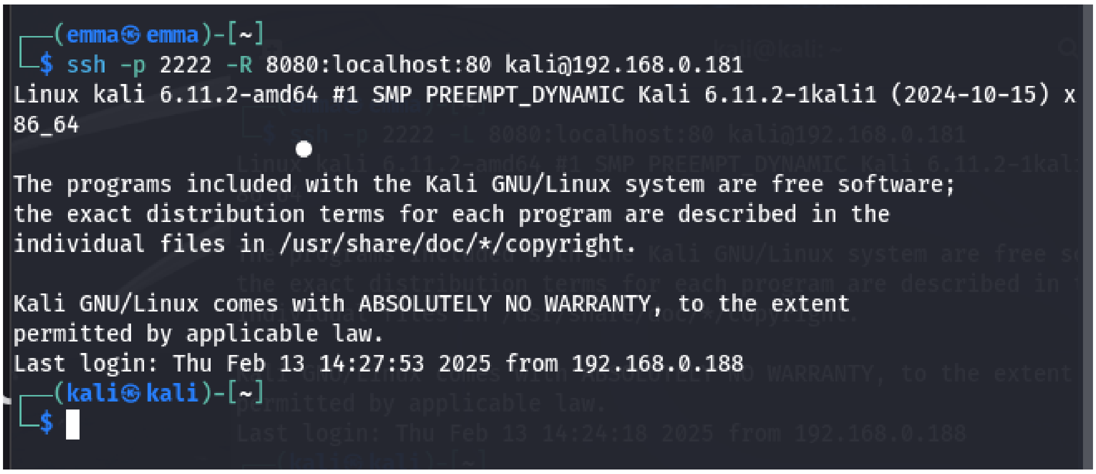
```
$ ssh -L 8080:localhost:80 user@ip_address
```

By opening a browser and navigating to `localhost:8080`, we were able to interact with the remote service.

6.6.2 Remote Tunnel

We also created a remote tunnel to forward a remote port to a local port. For example, to expose a local service on port 80 via remote port 8080, we used:

```
$ ssh -R 8080:localhost:80 user@ip_address
```



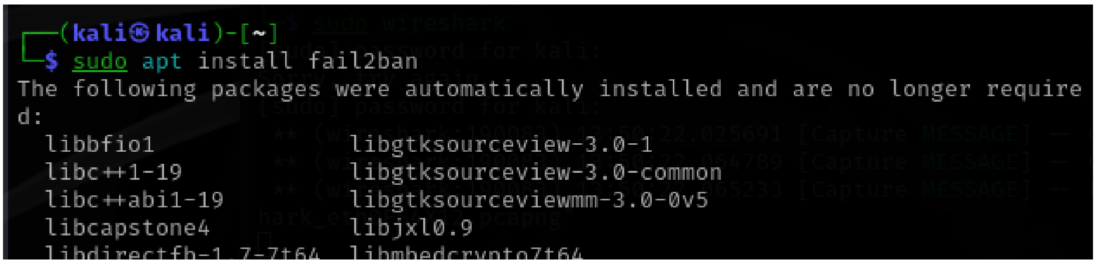
This allowed a remote user to access the local service by connecting to `ip_address:8080`.

6.7 Security Measures

6.7.1 Fail2Ban

To protect the SSH server against brute-force attacks, we installed and configured Fail2Ban:

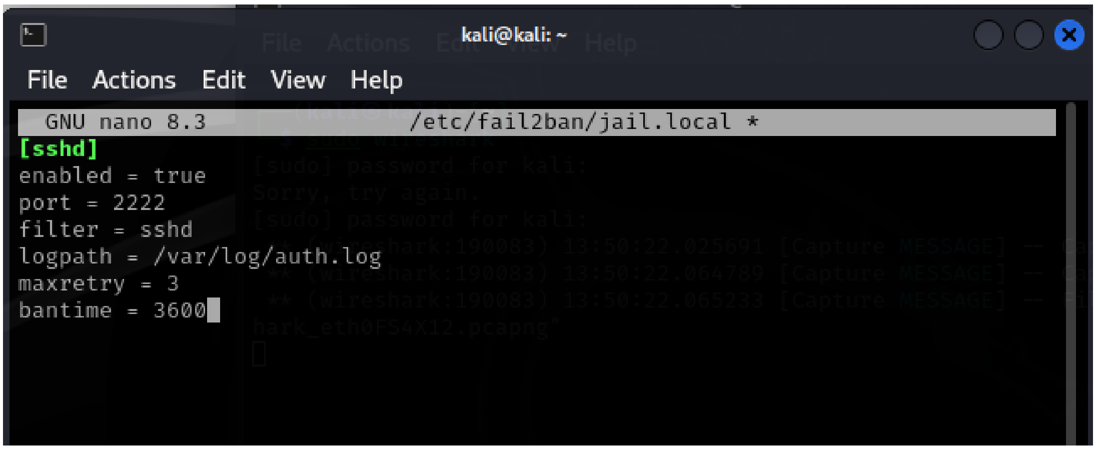
```
$ sudo apt install fail2ban
```



We then configured Fail2Ban to monitor SSH login attempts:

```
$ sudo nano /etc/fail2ban/jail.local
```

We added the following lines:



After restarting Fail2Ban, the SSH server was protected against brute-force attacks.

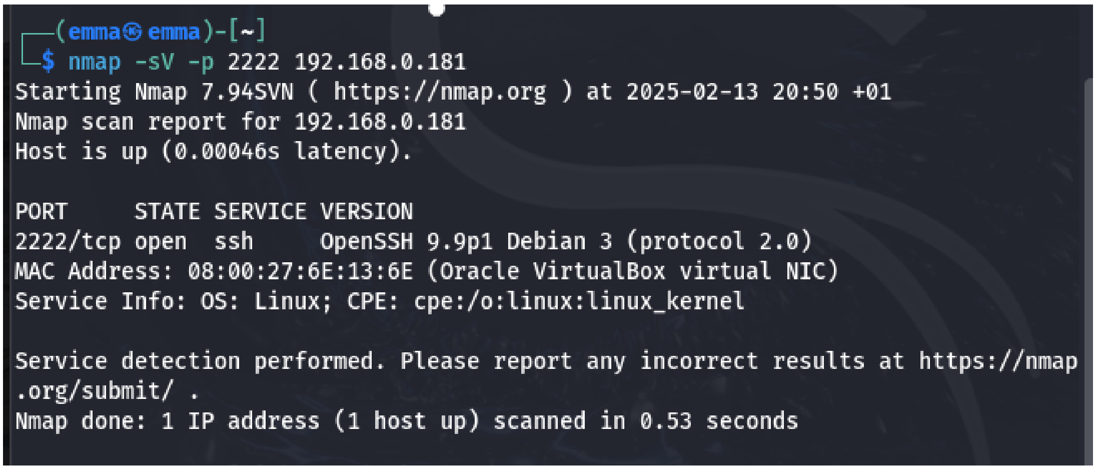
6.7.2 Testing SSH Security with Different Methods

We tested the security of our SSH server using tools like `nmap` and `hydra`:

Verify that the SSH port (2222) is well-open and accessible:

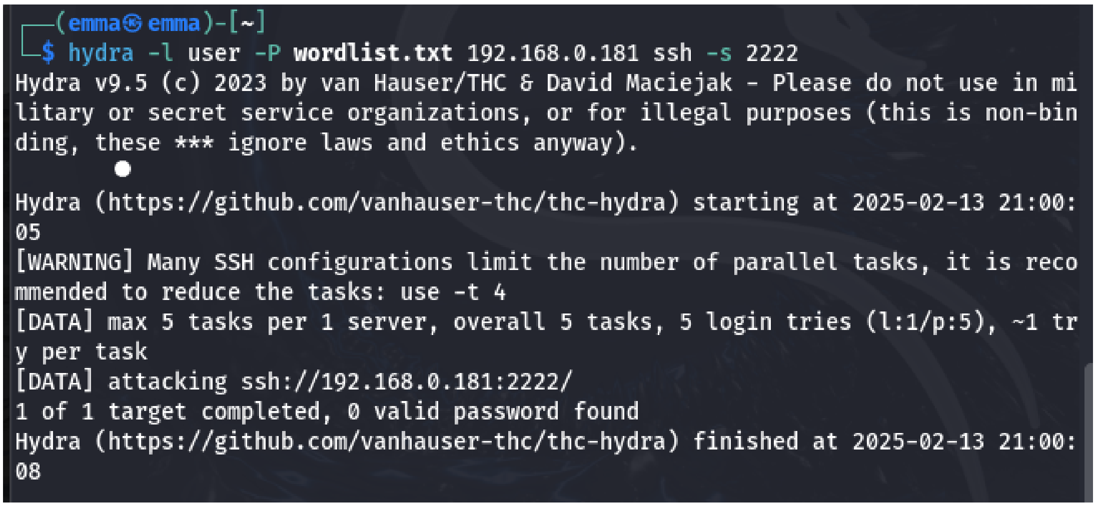
- Port Scanning with Nmap:

```
$ nmap -sV -p 2222 ip_address
```



- Brute-Force Attack with Hydra:

```
$ hydra -l user -P wordlist.txt ip_address ssh -s 2222
```



6.7.3 SSH Security Hardening Techniques

We applied the following techniques to harden the SSH server:

- Using SSH keys instead of passwords.
- Disabling root login.
- Changing the default SSH port.
- Limiting allowed users.
- Configuring Fail2Ban to block brute-force attacks.

6.8 Automation

6.8.1 Bash Script to Automate Tasks

We created a Bash script to execute remote commands on multiple servers. For example:

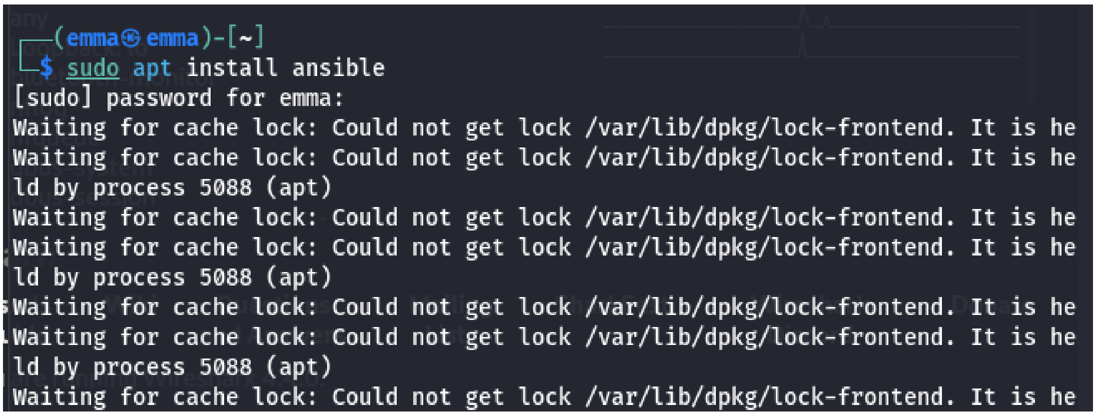
```
$ #!/bin/bash ssh user@ip_address "command1; command2"
```

This script allowed us to automate repetitive tasks.

6.8.2 Using Ansible

We installed Ansible to automate server management:

```
$ sudo apt install ansible
```



We created an inventory file to define the servers to manage:

```
$ [servers] server1 ansible_host=ip_address
```

We then used Ansible to execute commands on all servers:

```
$ ansible servers -m ping
```

7 Conclusion

The Secure Shell (SSH) protocol has fundamentally transformed the landscape of secure communication by offering a robust solution for encrypted data exchange, remote system administration, and secure file transfers. Its layered architecture ensures modularity and scalability, while its reliance on advanced cryptographic techniques guarantees confidentiality, authentication, and integrity in modern networked environments. SSH's versatility is further enhanced by its support for multiple authentication methods, making it an indispensable tool for both individual users and large-scale organizations.

However, as cyber threats continue to evolve and computational capabilities advance—particularly with the advent of quantum computing—the security of SSH deployments must be continuously reassessed and fortified. Best practices such as disabling password-based authentication, enabling fail2ban for brute-force protection, and regularly updating cryptographic algorithms are critical to maintaining SSH's effectiveness. By adhering to these principles and embracing emerging technologies, SSH will remain at the forefront of secure remote access solutions, adapting to meet the challenges of tomorrow's digital world.