SC Connect: Secure Server Access from Mobile Device

Wilfred Almeida
Department of Computers Engineering
Thakur College of Engineering and Technology
Mumbai, India
almeidawilfred642@gmail.com

***Abstract*— With the rise of technology, Server requirements have grown at an overwhelming pace. Accessing these servers and performing administrative tasks on them remotely has always been a great challenge. A popular utility used for this is the Secure Shell (SSH) which is being used extensively to get shell access on servers. SSH is however prone to vulnerabilities like Dictionary Attacks and Man-In-The-Middle (MITM) attacks. Further on, accessing servers from mobile devices is not yet feasible, SSH remains the de facto choice for it however SSH via third-party applications from mobile devices poses a security risk.**

In this paper, I've proposed an alternative system for accessing servers from mobile devices from operating systems Android and iOS. The main idea here is to keep commands pre-defined on servers and facilitate their execution from mobile applications securely.

Keywords*— *SSH, Servers, SSH Alternative, Server Access from Mobile, SSH Alternative

I. INTRODUCTION

Mechanisms to access servers from desktop environments are widely used reliably and efficiently, however, to access servers from mobile devices there aren't any viable solutions. Servers are critical backbones of businesses and any compromised security towards them can be catastrophic.

Well-established services such as Secure Shell (SSH) are an option for system administrators however from a mobile environment using third-party applications poses a security risk. Alongside that, to perform low-complexity tasks such as routine maintenance and health checks, a full-fledged SSH connection has to be established which is overkill for such minimal tasks.

The proposed system in this paper addresses the issue of access from a mobile device by setting up a Representational State Transfer (REST) Application Programming Interface (API) on the server which will be used to establish a secure connection channel between the mobile application and server. The server will have preconfigured shell commands whose identity will be sent to the mobile device. The mobile device will respond with the (Unique ID) UID of a command and the appropriate command will be executed and its result will be sent to the mobile device. The data exchange will be secured via public key cryptography using the Rivest, Shamir, Adleman (RSA) algorithm. Two sets of RSA 4096 bits keys must be maintained for usage by the client and server. These keys must be generated by the administrator. The mobile application will scan the Quick Response (QR) codes of these keys and store them securely in its local app storage.

II. LITERATURE REVIEW

There are various existing solutions that provide connection mechanisms. Some have industry-wide usage whereas some are still in the early adoption phase.

A. SSH

SSH was developed as a replacement for insecure platforms such as telnet by Tatu Ylonen in 1995. An SSH connection starts with an initial handshake. During this handshake, the server first attempts to establish the key exchange (KEX) algorithms to determine the encryption for the connection. Once the algorithm is agreed upon, the host key and cipher algorithms are agreed to followed by the exchange of the host keys. [1]

1. SSH Vulnerabilities

This section outlines some prominent issues with using SSH. It emphasizes the password-based and key-based login mechanisms of SSH.

- Honey Pot Attack Analysis [2]

The following table outlines the analysis of a honeypot attack study conducted to gain statistics on attacks targeted at SSH.

TABLE I. HONEYPOT ANALYSIS

Timeline	55 days
SSH Connection Attempts	16,558 times
Login Attempts	32,695 times
Unique IP's	683 addresses
Distinct Usernames	279 usernames
Most Used Username	*root* 53%
Most Executed Commands	*wget, curl*

From Table I it can be inferred that SSH connections are a prime targets of attackers.

- SSH Key Analysis [3]

The following table outlines the usage of SSH keys usage and their usage and access metrics.

TABLE II. SSH KEYS ANALYSIS

Key Usage	90% authorized keys unused
Key Lifetime	Extreme Cases 10-20 years

978-1-6654-9291-1/22/$31.00 ©2022 IEEE

Access	10% grant root access

From Table II it can be inferred that a significant amount of SSH keys granting elevated privilege are unused and difficult to manage.

B. *Mosh (Mobile Shell)*

The Mosh (Mobile Shell) is a terminal application for remote connectivity with features like intermittent connectivity, roaming connectivity, and safe echoes of user keystrokes over high latency network connections. [4]

Mosh utilizes State Synchronization Protocol (SSP) which is a secure object synchronization protocol on top of User Datagram Protocol (UDP) that synchronizes abstract state objects in the presence of roaming, intermittent connectivity, and marginal networks. [4]

Mosh conveys the most recent screen state to the client at a "frame rate" which allows avoidance of filling up of network buffers. Connection is initiated by running a script that logs in to the server via conventional means like SSH and runs the server. This program listens on a UDP port and generates a random shared symmetric encryption key. The SSH connection is terminated after this and the server talks directly over UDP. [4]

Mosh is relatively a new tool and has long release cycles with a lack of security audits. Server admins prefer its functionalities but are uncertain whether or not to adopt it due to a lack of credibility.

C. *Mobile SSH*

In [5] a mobile SSH protocol is proposed based on the application-layer handover approach. Whenever the mobile device is roaming and the session is broken, the server receives a reconnection request from the SSH mobile host with new IP configuration. The server then authenticates and updates the configuration of the session and rebuilds a secure connection. [5]

Due to network address change, the executing shell gets suspended by the SSH server. To distinguish a reconnection request and a general SSH session establishment request an extension flag is used in the header of the original SSH packet. [5]

The session key remains the same. If the reconnection request has the original SSH session key then the request mobile host gets authenticated. After authentication the socket descriptor is passed by the SSH daemon to the suspended shell. The suspended shell then wakes up and retransmits lost packets and the session continues. [5]

The identified issues are as follows:

1. Authentication

The daemon process maintains a copy of each session key in a shared memory table between the process and the shell. [5]

2. Synchronization

Mobile host's network switching leads to packet loss which must be recovered based on the sequence number carried by the reconnection request. [5]

3. Concurrency Control

Due to reconnection needing authentication, the session key table is implemented with semaphores. Due to this the server only processes one connection request at any given instance of time which leads to starvation of other requests and increases reconnection time. [5]

4. Attacks and Fault Tolerance

While the server is waiting for reconnection, malicious attackers may try to connect, even if the request gets denied the server resources will still be used. If the reconnection requests are flooded, the genuine request will starve. [5]

III. PROPOSED SYSTEM

The system is proposed in two sections. The first section consists of the server-side architecture and the second section consists of the client-side mobile app architecture.

A. *Server Side System*

The architecture of the server side system is discussed in this section.

Figure 1: Server Side System Flow

The above figure illustrates the architectural flow of the server side system. Its modules are discussed as follows.

1. Authentication Module

This module authenticates all incoming requests. It verifies the request origin device and then decrypts the body and passes it on for further processing.

2. Commands Parser Module

This module interacts with the file system and reads all commands and their metadata. It parses them into a standard form that's understandable by other components of the system.

3. Command Execution Module

This module executes the command provided to it and returns the result. It has libraries that interact with the Operating System (OS) to execute the specified command. It takes in the ID of the command to be executed and then fetches the command from the Commands Parser Module. Then it executes the command and returns the result.

a) Security

All communication between the client and server is encrypted. Public Key Cryptography using the RSA Public-Key Cryptography Standards (PKCS8) algorithm is used. The administrator generates two sets of RSA 4096 bits keys and provides their paths in the application. The description of the keys is as follows.

1. Server Keys

The RSA keys for the server. The public key is shared with the client. All incoming requests to the server are encrypted using its public key. The RSA public key of the client is stored with the server and all responses sent to the client are encrypted using this key.

2. Client Keys

The RSA keys for the client. The public key is shared with the server. All incoming responses to the client are encrypted using its public key. The RSA public key of the server is stored with the client and all requests sent to the server are encrypted using this key.

3. SSL

The API endpoints can have Secure Socket Layer (SSL) setup. This is optional however is strongly recommended for enhanced security.

B. Client Side Mobile Application

The architecture of the client-side mobile application is discussed in this section.

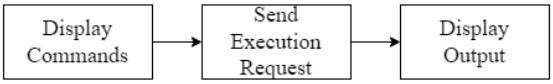

Figure 2. Client Side System Flow

The above figure illustrates the architectural flow of the client side system. Its modules are discussed as follows.

1. Display Commands Module

This module fetches available commands from the server and displays them on the device screen. Users can tap on the available commands and send an execution request.

2. Send Request Module

This module sends command execution requests to the server. The API calls are secure and the request body is encrypted.

3. Display Output Module

This module displays the execution result returned by the server. The response is first decrypted and then shown to the user.

C. Additional Features

This section suggests some additional security measures that could be implemented for enhancing security. However, these will add performance bottlenecks.

1. API Body Signing

For additional security, the API request and response bodies can be signed using a signing algorithm. RSA-based SHA-256 is recommended however is not mandatory. Any suitable algorithm can be used. Signing can increase CPU requirements and increase response times. Developers need to decide accordingly. The signing key must be transferred to the client by scanning a QR code.

2. Connection Security

To establish a secure tunnel between the client and server, a Virtual Private Network (VPN) service can be established using Wireguard. This must be set up by the administrator externally and only a VPN connection must be allowed to connect to the server.

3. TOTP Authentication

For accessing the mobile application, the user can be prompted to enter a Time-Based-One-Time-Password (TOTP) authentication code which must be verified by the application, and only after a successful verification the user should be allowed to use the app.

CONCLUSION

Accessing servers via SSH is widely used and has its own sets of issues and vulnerabilities. On top of that accessing servers via mobile devices relying on third-party applications elevates the risk associated with SSH. SC Connect aims to provide a platform wherein the admins can perform specific tasks like monitoring, health checks, reboots, etc. without establishing an SSH connection and facing the risks and difficulties associated with it. The process of command execution happens in various stages which makes it difficult for an attacker to cause any harm. In a worst-case scenario wherein an attacker has the capability to execute commands via SC Connect, the executable commands will be limited, and hence the caused damage if any won't be catastrophic.

ACKNOWLEDGMENT

I'd like to thank Dr. Vidyadhari Singh and Prof. Aniket Mishra for their guidance and help in formulating the idea and guiding me while writing this paper. The paper wouldn't have been possible without their guidance. They helped me bring clarity to the idea and guided me in approaching the problem, formulating the solution, and finally writing and reviewing the paper.

REFERENCES

[1] R. Andrews, D. A. Hahn, and A. G. Bardas, "Measuring the prevalence of the password authentication vulnerability in SSH," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, Jun. 2020, pp. 1–7, doi: 10.1109/ICC40277.2020.9148912.

[2] Andhra University CS&SE, Visakhapatnam, 530003, India, S. Z. Melese, and P. S. Avadhani, "Honeypot system for attacks on SSH protocol," *IJCNIS*, vol. 8, no. 9, pp. 19–26, Sep. 2016, doi: 10.5815/ijcnis.2016.09.03.

[3] T. Ylonen, "SSH key management challenges and requirements," in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Jun. 2019, pp. 1–5, doi: 10.1109/NTMS.2019.8763773.

[4] K. Winstein and H. Balakrishnan, "Mosh: An interactive remote shell for mobile clients.," 2012 USENIX Annual Technical Conference (USENIX ATC 12), p. 177, 2012.

[5] I.-H. Huang, W.-J. Tzeng, S.-W. Wang, and C.-Z. Yang, "Design and implementation of a mobile SSH protocol," in *TENCON 2006 - 2006 IEEE Region 10 Conference*, 2006, pp. 1–4, doi: 10.1109/TENCON.2006.343956.