# MobXcess: Secure Server Access from Mobile Devices

Wilfred Almeida, Rohan Dalvi, Faraz Ahmed, Lydia Suganya
*Department of Computers Engineering*
*Thakur College of Engineering and Technology*
Mumbai, India
almeidawilfred642@gmail.com
rohanjdalvi1402@gmail.com
farazahmed06@gmail.com
lydia.suganya@tcetmumbai.in

***Abstract*— Accessing remote servers has always been a difficult task. In this paper, we offer an alternative to SSH to access servers from mobile devices. This option is based on a secure and reliable protocol that ensures data privacy and integrity.**

**The proposed protocol allows users to securely connect to remote servers over the Internet. It provides an easy-to-use interface that allows users to execute commands with extreme security. The protocol encrypts all data transmissions, which prevents eavesdropping and unauthorized use. The essence of the idea is to keep the allowed executable commands predefined on the server and facilitate their execution through the mobile application.**

**The proposed protocol is very efficient and can be used for all types of servers, including servers with different operating systems. The protocol is also easy to configure and can be implemented on most mobile devices with little effort. In addition, the protocol can be used on servers in various networks, including public and private networks. Due to its secure and reliable features, the proposed protocol is an ideal solution for accessing remote servers from mobile devices.**

## I. Introduction

This paper proposes a system that solves mobile device access problems by defining a Representational State Transfer (REST). It creates a secure connection channel between the mobile application and the server. The server is preconfigured with shell commands that are detected and sent to the mobile device. The mobile device responds with the unique identifier (UID) of the command, and the corresponding command is executed and its result returned to the mobile device. Data exchange is protected by public key encryption using the Rivest, Shamir, Adleman (RSA) algorithm. Two sets of RSA 4096-bit keys are maintained for client and server use. Keys are generated by the system administrator, and the mobile application scans the quick response (QR) codes of these keys and stores them securely in its local application.

## II. Literature Review

Various solutions exist that provide connection mechanisms. Some are widely used in the industry, while others are still in the early stages of adoption.

### *A. SSH*

In 1995, Tatu Ylonen developed SSH as an alternative to vulnerable systems such as telnet. To initiate an SSH connection, the server first attempts to establish a key exchange algorithm (KEX) to decide the encryption of the connection. After the algorithm is accepted, the master key and encryption algorithms are confirmed and the master keys are then exchanged. [1]

### *B. Mosh (Mobile Shell)*

Mosh is a terminal program used to establish a remote connection with advantages such as roaming, irregular networks, and the security of replaying user keystrokes for high-latency connections. It uses the Synchronization State Protocol (SSP), a secure object synchronization protocol built on top of the User Datagram Protocol (UDP), which enables synchronization of abstract state objects even during roaming, intermittent networks and weak connections. [2]

## III. Proposed System

Figure 1 depicts the overall architecture of the system. The system is proposed in two sections. The first section consists of the server-side architecture and the second section consists of the mobile app client-side architecture.

### *A. Server Side System*

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

#### *a) Commands*

The commands are defined in a JSON file as an array of objects. Each object consists of *title* and *command* properties. Following is a sample of file *commands.json*

```
{
 "commands": [
   {
    "title": "Get Docker Images",
    "command": "docker image ls"
   },
   {
    "title": "Echo Hi",
    "command": "echo 'hi'"
   },
   {
    "title": "Git Status",
    "command": "git status"
   }
 ]
}
```

The administrator takes care of access and user permissions for the commands execution. The commands are

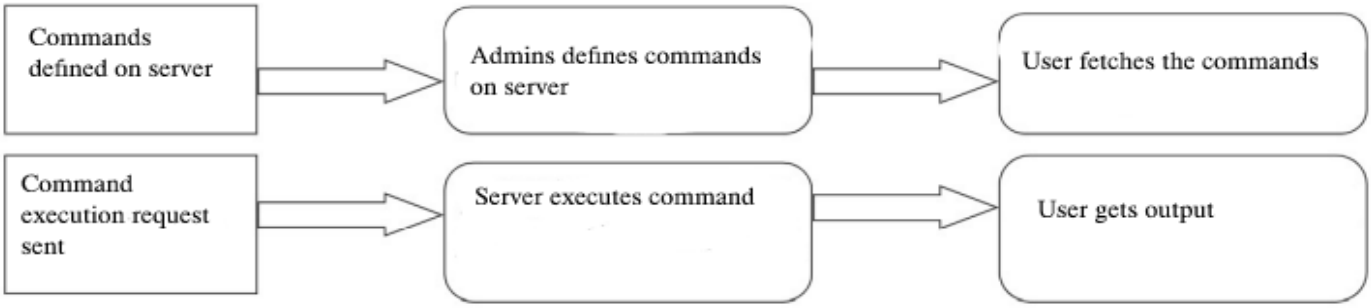


*Figure 1. System Architecture*

interpreted from here and executed using the development language's provided APIs.

*b) Commands Parsing*

The *commands.json* file is parsed on application start and a metadata file for the commands is generated. This file consists of command information as well as generated metadata like UID for the command. Administrators modify the application code to generate any more additional metadata. Following is a sample of the *commands-generated.json* file. This file is auto parsed on changes without needing to restart the application.

```
{
 "commands": [
  {
   "title": "Get Docker Images",
   "command": "docker image ls",
   "id": "de43tg7",
   "parseTime": 1659811532
  },
 ]
}
```

The UID is a unique randomly generated string of 6 characters that refreshes after a defined time interval, when file changes or on application restart. Additionally, if the administrator wants to serve a static UID for a command, it can be defined in the *commands-generated.json* file. However, this is less secure and by default, the static UID gets overwritten if the *commands.json* file is reparsed.

*c) Endpoints*

The system exposes a port for REST API endpoints. The security of the port is the responsibility of the administrator. The system listens for requests on the endpoints and responds accordingly. The response from the endpoints follows a fixed format. Following is a sample of a response:

```
{
 "status": 1,
 "message": "Some Message",
 "payload": [
  {
   "key": "value"
  }
 ]
}
```

Following are the response fields explained:

1. **status**: This indicates the success of the operation. Possible values:
   - **0**: Indicates Failure of the operation
   - **1**: Indicates Success of the operation

2. **message**: A human-readable message string providing some information about the operation.

3. **payload**: The response payload from the endpoint. Possible values:
   - **null**: If no response is to be returned.
   - **list**: A list of objects containing the responses.
   - **string**: A string stack trace of any error that occurred.

The following endpoints are exposed:

1. **getCommands**:

This endpoint returns the title and UID of parsed commands. The request to this endpoint is an HTTP POST request without any request body. The response returned is of type *application/json*. Following is a sample of the expected response from this endpoint:

```
{
 "status": 1,
 "message": "Data Fetched Successfully",
 "payload": [
  {
   "commands": [
    {
     "title": "Get Docker Images",
     "id": "fhg73g6"
    }
   ]
  }
 ]
}
```

Note that only the title and id should be returned in the response.

2. **runCommand**:

This endpoint takes in the UID of the command to be executed and executes the command and returns its result. The request to this endpoint must be an HTTP POST request with the request body type as *application/json*. The response of this endpoint must be of type *application/json.* Following are samples of request and response objects:

Request Object:

```
{
   "commandId":"ek68cd"
}
```

Response Object:

```
{
 "status": 1,
 "message": "Data Fetched
Successfully",
 "payload": [
  {
   "result": "hello"
  }
 ]
}
```

*d) Security*

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

- **Server Keys**

A directory contains the server keys. The public key is sent to the client. The client encrypts all data to be sent to the server using the public key. The private key in this directory is used to decrypt the data sent by the client.

- **Client Keys**

A directory contains the client keys. The private key is sent to the client. The server encrypts all data to be sent to the client using the public key. The private key is used by the client to decrypt the data sent by the client.

- **SSL**

The administrator can set up SSL to the endpoint for enhanced security. Setting SSL is optional, however, it is strongly recommended to do so.

*e) Key Exchange*

The keys are not exchanged over any network. The administrator is providing QR codes of the keys that are being scanned by the client and the keys are being transferred. The QR code generation utility must be trusted by the administrator.

- **Authentication**

Requests from trusted devices are only allowed. To add a trusted device, the administrator has to define a file which contains an UID of the device. The UID is shown by the client mobile application and should be unique for each device. Following is a sample of the trusted-devices.json file:

```
{
  "trustedDevices":[
    {
      "name":"My Device",
      "uid": "SMSNG35428J30P"
    }
  ]
}
```

The hash value of this UID is sent from client to server. Requests from mobile devices whose UID is specified here are only allowed. This is the first step of authentication. The request body is not attempted to be decrypted before the UID from the header is verified. After the UID is verified successfully, the request body is decrypted and the request is processed. Authentication fails in event of the following conditions:

1. The UID hash sent by the client does not match the UID hash specified in the *trusted-devices.json*.
2. The decryption of the request body fails

A response of failed authentication is sent to the client with the appropriate HTTP status code, which is usually 401. Here is a sample of failed authentication responses:

```
Unauthorized
```

The administrator can send additional information about the phase in which authentication fails however it is not recommended to do so since it might provide additional information to an attacker too.

*f) Commands Execution*

To execute commands and send a response, the following steps need to be followed:

1. Obtain command id from the request body
2. Fetch command string from the *commands.json* file
3. Execute the command using APIs provided by the language
4. Convert the result to *application/json*
5. Encrypt the response object using the public key of the client
6. Send the response to the client

Whatever output is obtained from the command execution is sent to the client. Administrators can modify the actions to be taken based on the response however these actions should not require any interaction from the client and notifying the client about such an event should be avoided.

*g) Files Reloading*

JSON files containing commands and device UIDs reload automatically without needing to restart the application,

though this might increase CPU requirements and require multiple threads. Administrators provide an option to auto-reload files or restart the application.

Another scenario administrators address is whether to load data from configuration files in memory or reopen the files every time a request is made, depending on the server's resource limitations. It's a tradeoff between response times and memory.

### B. *Client Side Mobile Application*

The mobile application is built using reliable and stable technologies. It follows the Model-View-View-Model (MVVM) architecture.

#### *a) Server Connection*

To communicate with the server, the application needs the following parameters:

1. **Server URI:** Can be an IP address or domain endpoint. All requests will be sent to this URI.
2. **Public Key of Server:** All data sent to the server must be encrypted using this key.
3. **Private Key of Mobile:** This is the private key of the client's mobile application. All data sent from the server to the client will be decrypted using this key.

#### *b) Keys Transfer*

All necessary cryptography keys are provided to the application in form of QR codes. The application implements an appropriate QR code scanning mechanism that scans the QR code of keys provided by the administrator. If the scanning needs a third-party utility/package to be included, it is trusted by the developer. All scanned keys are stored encrypted in the persistent storage of the application using appropriate databases.

#### *c) Persistent Storage*

The keys are stored in an encrypted database. A popular database solution for mobile devices offers encryption services. No session data is stored in the application, every request sent is authenticated. Keys and app access password hash are the only data is stored persistently.

#### *d) App Access*

The app is secured with a password and the user is prompted for the password every time the app is in focus. If the device supports biometric authentication or any other secured app access mechanisms, they are implemented. If the app loses focus for more than 15 seconds, the responses visible on the screen are cleared. The command execution results from the server are only present in the memory while the application is focused on the screen and the user is using the application.

After 15 seconds of app inactivity, the app closes and upon restart, it reauthenticates the user. Additionally, if a network change is detected, the application terminates itself and the user must restart it manually.

#### *e) Core Functionality*

Commands are fetched from the *getCommands* API and the titles of these fetched commands are shown to the user. Upon clicking a command, an API call to the *runCommands* API is made using the appropriate body. The response received is displayed on the screen. The response received is not persisted anywhere. For the utmost security, it is recommended that screenshot clicking be disabled too. However, for user convenience screenshots can be allowed along with a button to copy the response text to the keyboard.

## CONCLUSION

However, SSH is a battle-tested technology and we need community support and opinions to bring the project to production. We need to do extensive research and explore the benefits of using SSH in our project. We should consider the cost of implementation, ease of use and security features provided by SSH. In addition, we should analyze the possible risks associated with the use of MobXcess, such as possible security holes that can be exploited. We should reach out to the community to get their opinion on using SSH in our project and whether it is a viable solution. Once we have gathered all the information we need, we can make an informed decision as to whether or not SSH is the right choice for our project.

## ACKNOWLEDGMENT

We'd like to thank our guide Prof. Lydia Syganya for her guidance and help in formulating the idea and guiding us while writing this paper. The paper wouldn't have been possible without her guidance.

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