Project Report: Physical Authentication System

– by Navneet Pandey

❖ Objective

The objective of this system is put automatic authentication system at home. This system uses two-step verification to determine whether guests at the door should be granted access or not. Two verification methods are 'Face Recognition' and Distance based Password'. Just to add extra security layer, a secondary verification is made by the owner by authorizing the guest using an android application.

❖ Project Requirements

- **Hardware**
  - Raspberry Pi – Raspberry Pi 2B was used for this project
  - Distance Sensor – VL53L0X Time of Flight distance sensor was used for this project
  - Webcam – Fosmon USB webcam was used for face recognition
  - Server – AWS EC2 instance for this project
  - Speakers
  - USB Wi-Fi Adapter
  - Memory Card
  - Android Device
- **Software**
  - Face Recognition API – Kairos API was used along with OpenCV Face Recognizer
  - Voice Synthesizer API – Google Text-to-Speech API was used
  - Android Studio – For the development of android application
  - SQLite Database – For keeping track of user and corresponding distance

❖ System Setup

Raspberry Pi is the central piece in this authentication system. Webcam, distance sensor and speakers are connected to the Raspberry Pi. An instance of AWS EC2 server needs to be created. The Android device needs to have the application installed.

❖ Working Mechanism

System starts by detecting some object in front of sensor. Guests are advised to follow the steps told via voice interaction. Guest is asked to place hands in front of sensor. Distance (being used as password) captured by sensor is matched with database. Next, the system initiates face recognition irrespective of the fact whether the password matched or not. Now, the system has 3 options depending upon the results of 'Distance Password Match' and 'Face Recognition'.

- **Password matches, and Face Recognition is successful**
  Pi awaits authorization and it also starts sending images from the webcam at regular intervals (decided by owner via app) via EC2. Owner needs to provide an appropriate answer to the authorization request.
- **Password match fails, and Face Recognition is successful**
  Guest is prompted again for entering the correct password. Guest is prompted thrice to provide the correct password before the program informs the guest to try again later.
- **Password match fails, and Face Recognition is unsuccessful**
  Pi starts sending images from webcam at regular intervals (decided by owner via app) to the android app (via EC2). The owner needs to provide an identity for the guest and provide an appropriate answer to the authorization request.

❖ Getting Started (Do It Yourself)

- Setup Raspberry Pi –
  - Installing OS Image: https://www.raspberrypi.org/documentation/installation/installing-images/
  - Copy an empty text file with the name "SSH" (no extension) to Memory Card
  - Insert Memory Card into Raspberry Pi and connect Pi to a modem using ethernet cable
  - Find the Pi's IP address and SSH to it either using putty or a Linux terminal
  - Login using the default credentials -> username: 'pi' and password: 'raspberry'
- Setup a Server – Choose one amongst Amazon AWS, Google Cloud or Microsoft Azure
  AWS: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html
- Setup Android Studio – https://developer.android.com/studio/index.html
- Raspberry Pi Installations –
  - OpenCV: https://github.com/Tes3awy/OpenCV-3.2.0-Compiling-on-Raspberry-Pi
    Also, save 'haarcascade_frontalface_default.xml' to project folder (from web)
  - VL53L0X Python Interface on Raspberry Pi:
    https://github.com/johnbryanmoore/VL53L0X_rasp_python
  - Python Libraries:
    - Pip (Python Package Mangement System) – sudo apt-get install python-pip
    - 'smbus' – 'sudo apt-get install i2c-tools libi2c-dev python-dev python3-dev'
    - 'numpy' – Installation available with OpenCV (see OpenCV link)
    - 'mplayer' – sudo apt-get install mplayer
    - 'sqlite3' – sudo apt-get install sqlite3

❖ System Performance

This section presents some insight related to network and memory use on both server and client end.

- Network Overhead – Each packet sent or received consists of header and payload. Payload is the actual data which needs to be transferred. The overhead is the additional data that you need to send the payload (i.e. header). To analyze the network overhead, packet capturing tool t-shark was used on both Raspberry Pi and server.

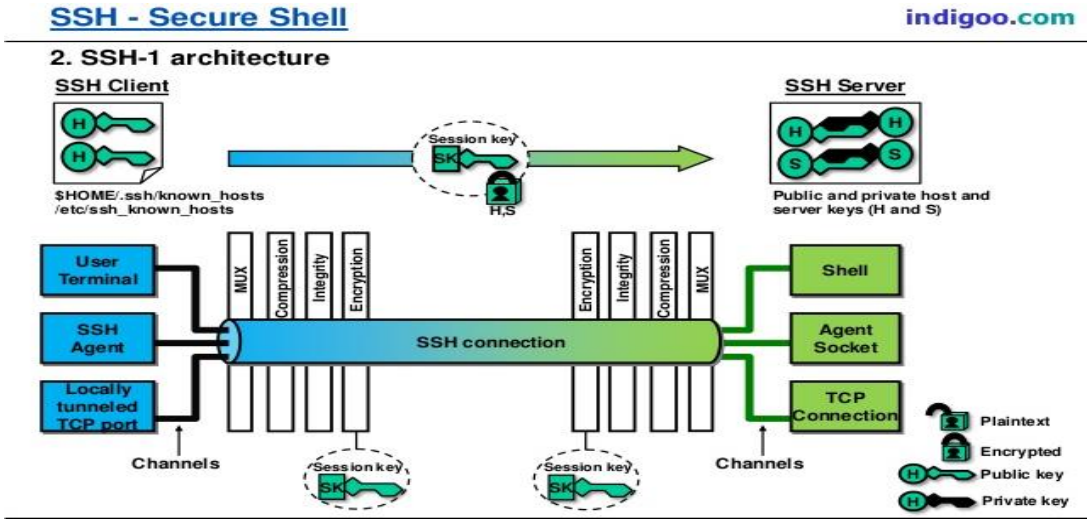

Figure 1: SSH Architecture (Image Source: Slide Share and indigo.com)

- **Sent/Received Byte Measurement –**
  - Sending 1 Image from Pi to Android app: 129KB (Payload: 120KB)
    Pi to Server:
    Bytes transferred for establishing a secure connection -> 6.46KB
    Bytes transferred when actual data was sent -> 122KB
    Bytes transferred for closing secure communication -> 1.01KB
    Server to Pi:
    Bytes transferred for establishing a secure connection -> 8.93KB
    Bytes transferred when actual data was sent -> 122KB
    Bytes transferred for closing secure communication -> 3.96KB
  - Sending ImageName.txt (helps app identify image file): 18.92KB (Payload: 22B)
    Pi to Server:
    Bytes transferred -> 7.18KB
    Server to Android:
    Bytes transferred -> 11.74KB
  - For 1 Authorization (from Android to Pi): 19.89 KB (Payload: 44B)
    Android to Server:
    Bytes transferred -> 12.77KB
    Server to Pi:
    Bytes transferred -> 7.12KB
- **Overhead Factors –** Actual data (Image) was sent using SSHv2 protocol. Thus, rest of the communication which happened between client and server can be considered as overhead. Also, the total number of bytes transferred using SSHv2 was 122KB, but the size of Image was 120 KB only. Thus, a total of 9KB as overhead for one image transfer. The overhead is because of the way communication happens (depending on the protocol) before and after the data transfer. Communication which takes place between server and client due to protocols (part of SSH communication) such as TCP, SSH(v1), ICMP are the factors responsible for overhead. Network Overhead:
  Sending an Image : 20.36KB
  Sending 'ImageName.txt' file : 18.92KB
  Sending Authentication : 19.89KB

- Face Recognition Tradeoffs

  Face recognition is a computer application can identify a person via a digital image or a video frame from a video source. There are 2 major ways to perform this function:
  - **Locally – Using OpenCV**
  - **Remotely – Face Recognition API (Amazon, Kairos, Microsoft, Google etc.)**

  Apart from positives and negatives of these application possess, there are some tradeoffs between the two methodologies. To analyze the pros and cons of the two methodology Linux OS based command 'free -m' was used to find memory consumption and t-shark was used for packet capturing.
  - **Networking** – A total of 149.502KB were sent over the network per API call (Kairos), whereas OpenCV has absolutely no network usage. Thus, if the connection is weak or a lot of recognition requests need to be sent, it is advised to use OpenCV.
  - **Latency** – OpenCV turned out to be faster than Kairos API. On an average Kairos API took about 8 seconds to figure the identity whereas OpenCV was much faster with 3 seconds.

- **Memory** – From memory perspective, local processing (OpenCV) will consume more memory than remote processing. Following are the stats of memory consumption:
  - OpenCV – 16 MB
  - Kairos API – 2 MB

  Thus, if the processor is short on memory, then it is advised to any of the available APIs (preferably Kairos because of its simplicity).

- Frame Rate – It is the count of frequency at which frames are displayed in the android application. The fastest frame rate I was able to achieve was displaying a new image almost every 2 seconds i.e. one image per 2 seconds. Here are some statistics to demonstrate this:

  50 Images Sent (from Pi) | 42 Retrieved (by Android) | Total Time - 1m 32s i.e. every 2.14 sec
  50 Images Sent (from Pi) | 39 Retrieved (by Android) | Total Time - 1m 35s i.e. every 2.44 sec

  I was able to achieve this when I was not providing any intentional system delay. In the original application I have a second delay (intentional) so that android app can retrieve every image which was sent i.e. I don't want a scenario where before android is able to fetch the current image, a new image is uploaded. Thus, retrieving a new image every 4 sec seems best. Bottlenecks:
  - **Transfer time** – It takes about a second to upload the image on server and similarly, it takes about a second to download the image to the android app. Thus, a minimum refresh interval of 2 seconds needs to be given.
  - **Pi Processing** – with limited RAM and absence of GPU it can only do so much processing. Also, the maximum speed it can achieve is also limited when compared with a computer.
  - **Android processing** – The android application cannot be bombarded with loads of images. Thus, reasonable amount images need to be sent to keep the application responsive.

- Polling vs Non-Polling –
  - **Polling** – Polling is the process where the client waits for an answer but instead of waiting for the same, it keeps asking or polling the server whether it received an answer or not. If server has an answer, client also receives the same. These processes can be as minute as only reading one bit. Pros and cons:
    - Control – In this process client has the control, thus, it is easier to manage.
    - Bandwidth Usage – Polling uses up a lot of bandwidth sending requests, and receiving responses. 7.12 KB of data is transferred each time the client (Pi) polls for an answer (i.e. authorization approval) to server (EC2). Thus, a multiple of 7.12 KB was transferred during polling depending upon the polling count.
    - Memory – About 1 MB of memory was used in this (client end).
    - Latency – Latency varies between 7 – 15 seconds
  - **Non-Polling** – Non-polling or server-push is the conveyance of information to the client by the server. Instead of continuously polling for the answer client decides to wait for the answer to be intimated by the server. Pros and cons:
    - Control – In this process server has all the control.

- Bandwidth Usage – Server push doesn't use a lot of bandwidth as the answer is notified to the client only once. Thus, a total of 7.12 KB was transferred during non-polling.
- Memory – Similar to polling, 1 MB of memory was used in this on the client end but server was also running a script to identify the answer, so another megabyte of memory was used.
- Latency – Latency varies between 10 – 15 seconds

Methodology: For this project Reverse SSH tunneling was used. Reverse SSH is a technique through which one can access systems that are behind a firewall/NAT. Instead of client doing a simple SSH, the server also does an SSH and through the port forwarding SSH gets back to the client machine.

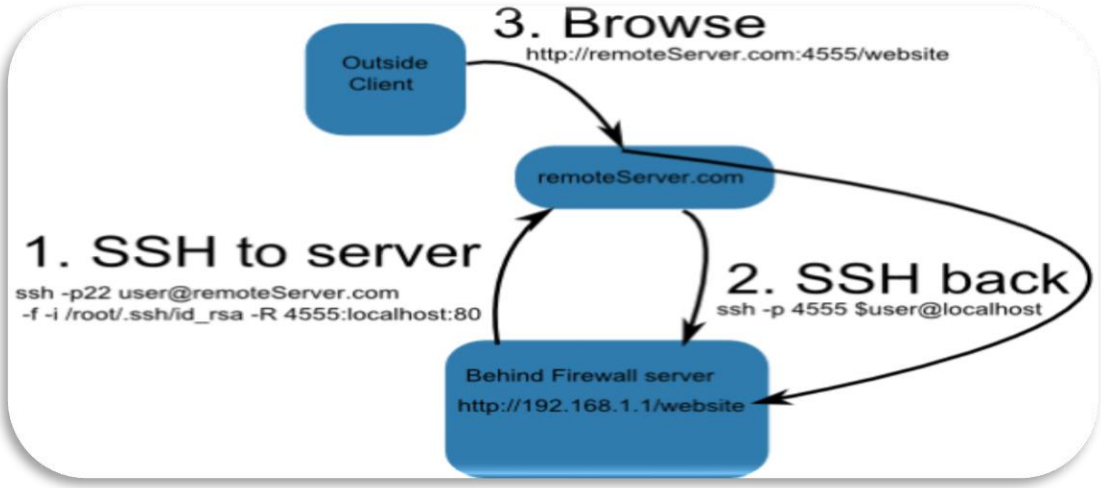

Figure 1: Reverse SSH

(Image Source: https://serverfault.com/questions/535075/using-ssh-tunneling-to-access-website-behind-firewall**)**

❖ What You Will Learn

This report will enable the reader to learn everything there is to learn about the project. This report starts with Objective, requirement and working mechanism. These will enable the reader to learn about the physical authentication system, get to know how it works and know the prerequisites (if the reader wants to pursue building such a system). Next, with basics in mind the reader is ready to implement this system and help the reader there is a next section called "Getting Starter or Do It Yourself". In this section, the reader is introduced with step by step implementation details of this system. Once the user has the setup, next section will provide some networking and processing related insight which will allow the reader to make informed decision regarding various aspect of the system.

❖ References

- https://blog.devolutions.net/2017/03/what-is-reverse-ssh-port-forwarding.html
- https://en.wikipedia.org
- https://www.webopedia.com/