Ryan Sullivan, Maniadarsh Sivaram

EC544: Networking the Physical World

29 April 2022

Final Project Report

Introduction:

RSA public-private key encryption is the most utilized asymmetric form of encryption throughout the world today, offering users the ability to securely distribute and decrypt data using their public-private key pairs. The main problem, however, is the frailty of the encryption if the private key is discovered prior to the encrypted message being transmitted. Even with every precaution being taken, a singular pass can be lost in a moment, compromising the entire process. Also, with a singular point of failure, if a key is lost without a means of generating another, the information encrypted can be lost to everyone, even the original node (only if they were to destroy their private key copy after distribution), such is the danger of asymmetric encryption.

Our proposed solution to this problem would involve a central node, being a Raspberry Pi in this instance, capable of generating an RSA key pair, encrypting, sending and receiving data, and splitting the pass randomly between a series of child nodes, clients, irrespective of their platform and software processes. These clients, prior to pass creation, would organize between themselves and the Raspberry Pi as to which holds what part of the pass. Later, on a connection and signal from all client parties, if the clients are present and discoverable, they would transmit the private key to any Raspberry Pi acting as a central node from which the pass can be reconstructed and used to decrypt any data encrypted by the other nodes. The amount of clients

present for the reconstruction is fewer than are present at the initial splitting, allowing for some amount of leniency and guarding against absolute failure given a singular client is compromised.

Clarity and Organization

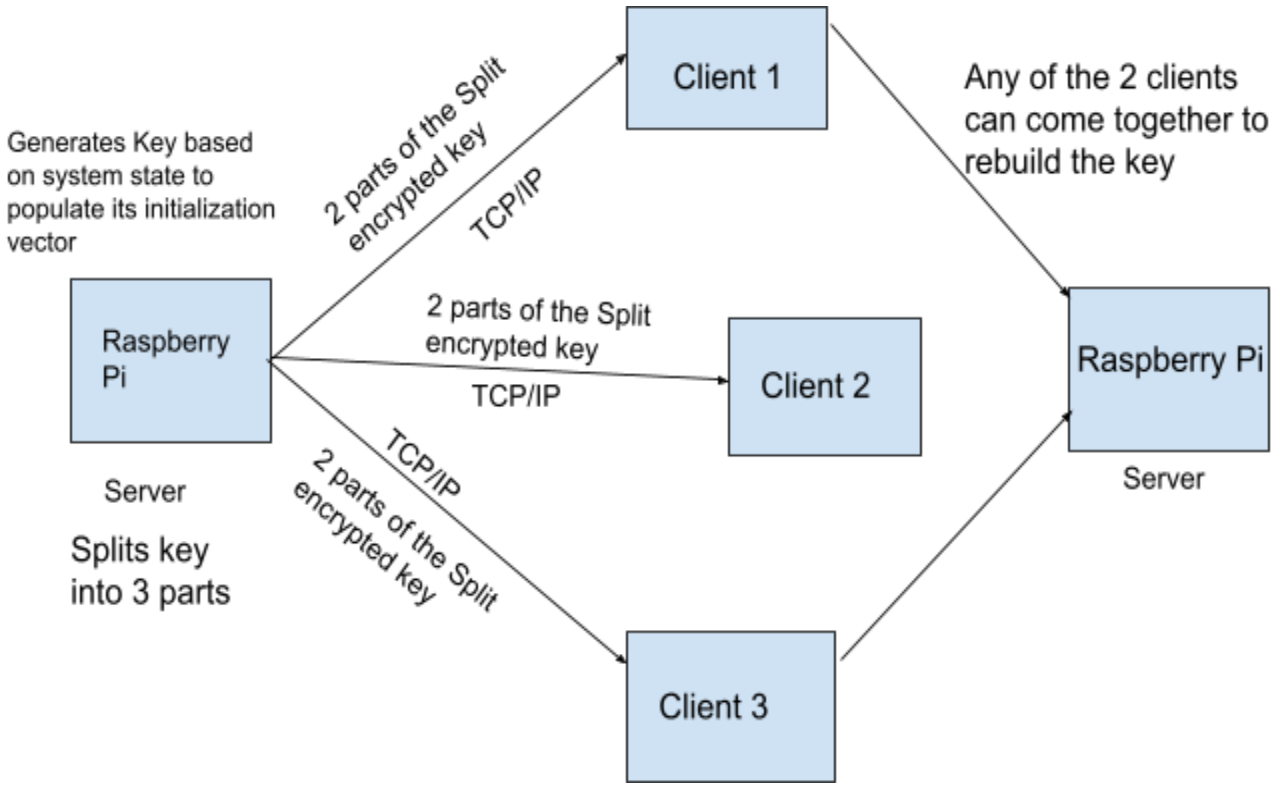

Figure 1. Overview of the system

Figure 1 shows the final system overview as previously described. First the Server, Raspberry Pi, generates an RSA key-pair using openssl. Openssl seeds its RSA keypair with an initialization vector using the system's entire state, gathered from a multitude of registers so as to better obfuscate the key generation process. We found this to be much more secure and efficient than using an outside GPIO sensor, the DHT22, as it only had a sampling rate of 1 sample per second and was only able to generate up to a 16 bit number. The RSA key is first split into its

public and private keys, with the public being stored on the server as well as a full copy of the private key. The private key is then split between the clients for later reconstruction on the secondary pi, where once it is reconstructed it is saved as key_reconstructed.pem.

Project failings

Using MQTT:

Initially the idea was to create a M2M connection using MQTT protocol. MQTT is a messaging protocol for the Internet of Things (IoT). It is a lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a minimal network bandwidth and can especially be used on small microcontrollers. So it was ideal to use it in this project.

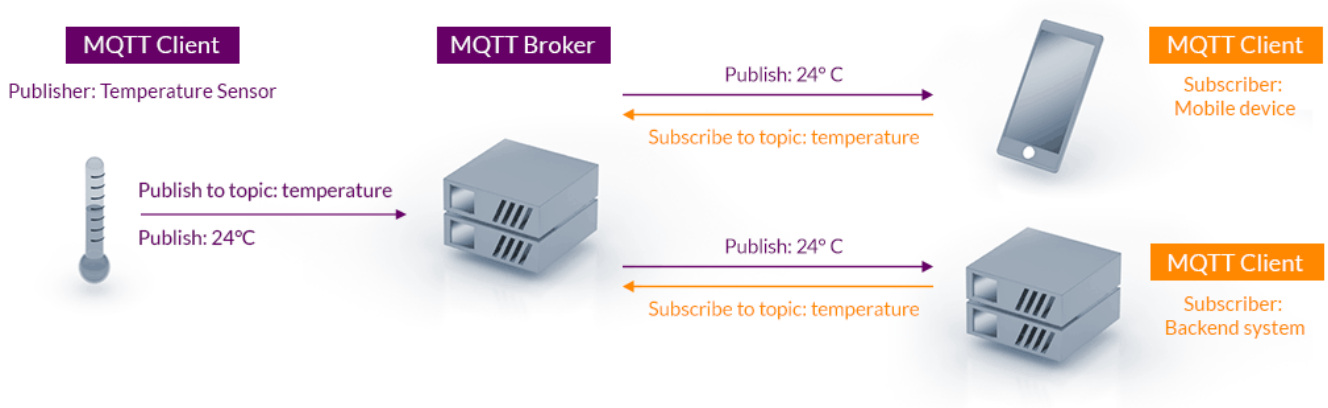

Figure 2. (taken from mqtt.org)

Some things to note while using MQTT protocol are[1]:

- Messages are published to a broker on a topic.

- The job of an MQTT broker is to filter messages based on topic, and then distribute them to subscribers.
- A client can receive these messages by subscribing to that topic on the same broker
- There is no direct connection between a publisher and subscriber.
- All clients can publish (broadcast) and subscribe (receive).
- MQTT brokers do not normally store messages.

We were able to connect our raspberry pi to the AWS IoT platform and were also able to publish messages to it. Problems occured while trying to figure out how to communicate those messages to another controller or Raspberry pi. In this project much two-way communication between the controllers takes place such as giving instructions like 'Generate' and modeling this communication under a publish subscribe model did not prove to be effective. As seen in the lecture on M2M, establishing a two way communication between Raspberry Pi's via MQTT is difficult to do. From the above image, it is observed that all MQTT clients need a broker through which the clients can subscribe to other clients or publish their messages, MQTT uses TCP/IP to connect to the broker. In this case communicating it to another controller through a broker gave a good set of problems which could not be handled as this was not foreseen. So the communication between the controllers was done using direct socket communication.

FRDM-K64F Board:

Initial goal was to use the FRDM Board as one of the microcontrollers to which the split part of the key was going to be sent. Problem was with burning the code to the board. Some of

the errors that occurred while trying to flash the code were “Remote Communication Error: Target Disconnected: Connection Reset by peer”, “IOexception”. Etc. This problem was handled by directly generating a .bin file, and this was pasted into the board's folder, or disk. The inbuilt demo “Led_blinky” ran successfully using this method but when the same method was used to execute a “Hello World” program it failed, the output on the terminal was viewed as just “Terminated”. Therefore the microcontroller had to be changed as we were unable to flash the program onto the chip.

DHT22 Temperature and Humidity sensor to create Randomness:

The sensor integration in the Raspberry Pi was successful, temperature and humidity values could be obtained. The output value of the sensor is only 16 bits whereas the value required to seed the OpenSSL RSA key generation algorithm was so large that it utilized the entire machine state, therefore the sensor was removed from the project

Project Successes

The project, at its core, was able to be executed in its full form, with key generation and splitting between nodes going through multiple iterations prior to the end product. First we explored using a simple XOR cipher with the DHT22 sensor seeding the rand() function and generating a byte string of length len. Using this method an initial idea was to split the string into parts and then distribute the parts to the clients in a way that allows for reconstruction without all initial clients present. This method is eventually what we finished with but not after considering

distributing random numbers between the clients that would multiply to the initial seeding number gathered from the DHT22. However, when we found that the DHT22 only generated 16 bit numbers, we decided to look to OpenSSL as a way to generate a more cryptographically sound key. After exploring the idea of simply creating a randomized hash of length len we eventually went with the final solution of generating an RSA key pair, and distributing the private key to the clients. Communication-wise the project utilizes low level socket communication over the TCP/IP layer that allows us to more intimately modify the exact communication method used by all of the parties involved in this cryptographic transaction. Initial difficulties with this method was mainly due to the fact that coordination between multiple parties throughout a complex transaction often leads to unintelligible data being sent into the communication buffer. The solution to this problem was to simplify the communication between client and server, with minimal back and forth save an acknowledgement and ready byte being sent between the two. These optimizations greatly reduced the presence of garbage data being sent into the communication buffer and thus allowed for the final solution to be implemented successfully.

OpenSSL:

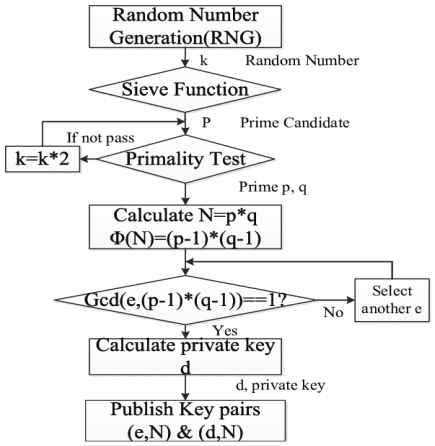

Figure 3. (taken from semanticscholar.com)

Throughout this project we utilize OpenSSL to handle the generation of RSA key pairs. OpenSSL is an open-source project meant to provide "a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication." To do so the project has compiled a series of cryptography techniques at the ready for any user that would so need them. These include the ability to generate MD5 hashes, encrypting files with generated keys, generating a randomized byte string of a set length, and many more. The main features we utilized in our project were the installed C libraries that allowed us to more directly interact with these cryptographic functions. These functions are also what allowed us to seed our RSA generation function with the entire system state, as opposed to a single 16-bit number as provided by the external sensor.

TCP/IP Sockets:

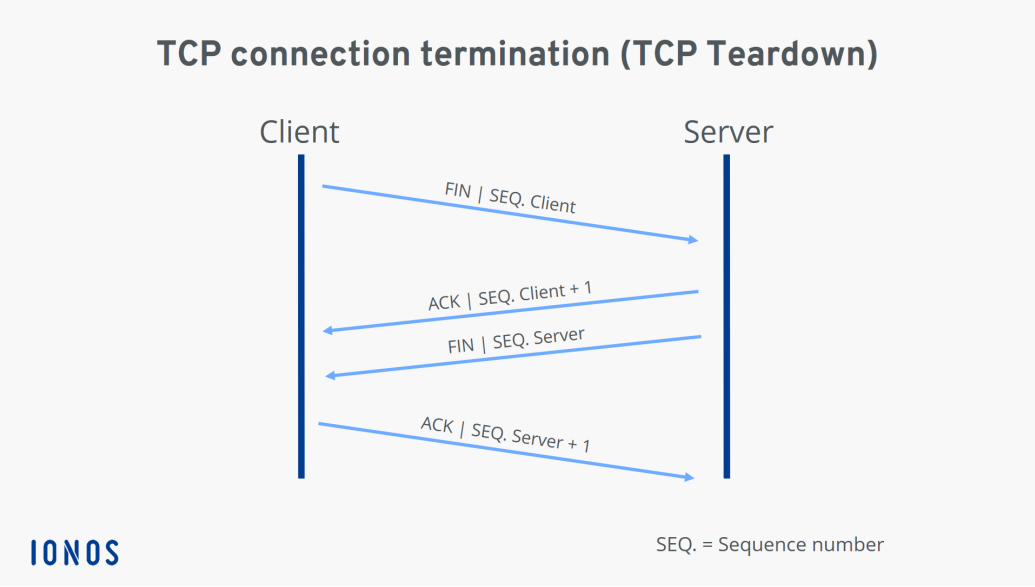

Figure 4. (taken from ionos.com)

Our project's other main technology is low level socket communication. This technology is built upon the Transmission Control Protocol (TCP), a communication protocol that requires a connection prior to any data transmission. This is in contrast to User Datagram Protocol (UDP), which allows for data to be sent to any location regardless of an established connection. We utilize TCP as it provided us with extremely low risk of data loss and ensured that we would minimize the need to check for too many acknowledgements from client to server and vice versa.

Utilizing sockets allowed us to more intimately work with M2M communication protocols, and even design our own communication pattern using low level byte transmission for acknowledgement and signaling.

How tech stack relates to course module

Cryptography:

Utilizing OpenSSL directly relates back to our Cryptography section in class. A RSA private and public key pair was generated using openssl. RSA is an asymmetric encryption algorithm, asymmetric implying that it uses a key pair to encrypt and decrypt data. With RSA, either the private or public key can encrypt the data, while the other key decrypts it. This is one of the reasons RSA is the most used asymmetric encryption algorithm. RSA's encryption is based on the idea that it is easy to generate a number by multiplying two large numbers together, but factoring that number back into the original prime numbers is extremely difficult. The public and private keys are created with two numbers, one of which is a product of two large prime numbers. Both use the same two prime numbers to compute their value.

Apart from RSA the core concept of reconstructing keys was presented in the lecture and that is what this project tries to accomplish. Instead of one device having the key, splitting it into equal parts and distributing it to different clients who can't use them either but when enough clients come together the key can be recreated, this reflects on concepts of Cryptography and accessibility. Accessibility here means that all clients need not come back to recreate the key, even if a client has lost connection to the system the other clients can accomplish the key recreation. As learnt from our module, cryptography is the science and technique for secure communication in the presence of third party adversaries and this project directly relates to that.

M2M:

One aspect of this project is Cryptography, the other aspect is establishing communication between the clients as we have to transport the split keys from the server to the client. This system is not an IoT based system because there is no connection to a cloud service such as Amazon Web Services, instead we establish wireless device to device communication between our server and clients. Following this initial connection, there is a simple user input, but that is on the client side and only there to determine which pattern is executed between the two acting machines. Following the user input, the entire exchange of information is a fully automated, Machine to Machine interaction, with all relevant data being sent, received and parsed without any further interaction from a human user.

This communication happens on the TCP/IP model with the help of TCP. It is known as the TCP/IP model because the two most important protocols in it are TCP and IP. Socket connection on the TCP/IP based network environment provides the routines required for interprocess communication between applications. A TCP/IP based distributed network

application described by a socket is uniquely defined by Internet address, Communication Protocol and Port, the port in our case is 1016. When the command ‘G’ is given, implying generate key is given, this information is passed on to the server through the socket defined by the client.c code. The client issues the connect socket function to start the TCP handshake. The server issues the accept socket function to accept the connection request. The client and server issue the read() and write() socket functions to exchange data over the socket. Either the server or the client decides to close the socket. This causes the TCP closure sequence to occur. The server either closes the listener socket or repeats to accept another connection from a remote client. Once the server receives three ‘G’s’, it creates a RSA key and splits it and sends parts of it back to the clients using the same socket connection defined above. The same applies when the “Reconstruct” command is given in any of the 2 clients. M2M is an important criteria in this course and this project proved to be a direct application of it.

Project Teachings

This project was an extremely useful learning experience throughout, with many roadblocks. The most important process we went through was deciding on key generation and splitting methods. Initial research led me to believe that the best method of key generation was to try and seed a pseudorandom number generation function, such as rand(), with an unknown value as provided by an external sensor. However, with the limited range of values that could be provided by a sensor, the eventual output was full of repetitive patterns and was deemed not cryptographically sufficient to fulfill the needs of the project. That is when we decided to look into OpenSSL as a means of key generation. The library provided many options but the most

familiar and the eventual best solution was to go about implementing an RSA key pair generation algorithm. This was extremely useful in learning more about the intricacies of the OpenSSL libraries and more importantly the details of RSA key pair generation. We were able to take time to investigate the technology and the details of one-way encryption and decryption, with the public and private key respectively.

The problem of key splitting was also an extremely pertinent one during the entire lifecycle of the project, from conception to completion. Initially, we had not considered the idea of a cryptographic threshold system, we had instead a basic splitting method. After meetings with Professor, however, we began to work on methods by which the clients could reconstruct the key lacking the full participation of the initial clients. Initially we just decided to split the key string into equal parts and distribute them between the clients present. Then, when we were still using the external sensor for seeding a pseudorandom number generation function, we considered instead factoring the seed number generated and sending the factors to the clients. Once we moved onto the OpenSSL RSA key pair generation method, the process of recording and transmitting the system state, which is used to seed the key generation method, proved infeasible. Therefore we landed on our final implementation which splits the private key and then distributes multiple overlapping parts to all of the clients involved. Then, because of these overlapping private key parts, if there are N clients, then only N-1 clients are needed to reconstruct the generated private key. Though other methods were discussed and researched, all that was found were theoretical systems with limited examples from which inspiration could be gained.

Finally, the largest hurdle throughout the project, and the final one to be solved, was the machine to machine communication that was the foundation of the project. We considered

multiple different avenues, including MQTT, but eventually we believed that the low level control sockets offered were extremely useful in being able to control our project's process with the adequate level of granularity we wanted. Using sockets we were able to define our pattern to a bytewise level, but with that also came much difficulty. The project was mired with issues of garbage data being sent and received, overflown buffers, and dropped data important to communication leading to connection hangups. The problem, we realized, was our overcomplication of the communication process. We found that the best process was to acknowledge the two participants, client and server, and then allow the transmitting party to write directly into a buffer file, rather than a simple buffer character array. This allowed parsing of data after the fact and much easier debugging, as we could physically inspect what data was transmitted versus the data that was received by each client.

Overall this project has been extremely beneficial in both of our journeys as engineers and we hope to carry these skills over into our futures in the professional world.

Works Cited:

OpenSSL Foundation, Inc. “OpenSSL.” */Index.html*, https://www.openssl.org/.

Steve, et al. “How Mqtt Works -Beginners Guide.” *Steve's Internet Guide*, 12 Feb. 2021, http://www.steves-internet-guide.com/mqtt-works/.

“TCP (Transmission Control Protocol) – the Transmission Protocol Explained.” *IONOS Digitalguide*, https://www.ionos.com/digitalguide/server/know-how/introduction-to-tcp/.