Group 5's Project Report
COMP4203A

NFC Authentication protocols are an important part of modern payment systems and personal identification, confirming the security of NFC protocols will improve security for everyone.

Authors:

CHRISTIAN BELAIR – 101078744
KYLE KNOBLOCH – 101064056
ERIC NEWCOMBE – 100963532

Instructor:

PROF. AHMAD TRABOULSI
Carleton University

April 20th, 2021

Abstract

Our goal is to replicate and test the authentication protocol for NFC tags from the paper "Secure and Lightweight Authentication Protocol for NFC Tag Based Services". This authentication protocol relies on the client's device being able to read what was previously stored on the NFC Tag and replace it with new information from the server so that the tag can be authenticated again. A concern with this protocol is that the tag information can be overwritten by a third party as most NFC Tags are unable to authenticate before being written to. We plan on testing this by creating multiple tags that have been previously authenticated, then attempt to overwrite them by using an Android device running the NFCTools app publicly available in the Google Play store to attempt a Denial of Service (DoS) Attack. Once the tags have been overwritten, they can no longer be authenticated resulting in the inability to access the tag until an administrator re-adds the tag to the system.

Contents

1 Domain

Near Field Communication (NFC) technology has helped automate many functions in day-to-day life. It is an extension of the already present Radio Frequency Identification (RFID) which is used partly within the Internet-of-Things domain. NFC is used within a much closer proximity (within 10 cm) compared to its predecessor RFID which has an effective range of roughly 100 cm [1]. This shorter range means that certain functions can branch away from RFID and use NFC as its interface. Examples where the NFC interface is currently in use are in bus passes, credit cards, debit cards, ID badges, and even cellphones. The use of NFC in these devices are mainly used for payment systems or as a means to authorize user actions. While these devices are more convenient to use than the conventional methods, the current concern behind NFC technology is that it presents a vector for multiple types of attacks.

NFC Security is an important technology in payment systems, physical access security systems as well as identification. NFC payment attacks are the most common method for attackers as they are becoming easier to replicate by malicious players. A different approach is through a skimming attack which can be achieved by placing a "skimmer" on top of a registered payment system that collects information at the same time as the payment system. After skimming the information, it is stored until the malicious person can take the skimmer off and extract the data. Another kind of skimmer works by using a large antenna to grab NFC tags on physical credit cards. After receiving the data from a skimming device, malicious users can then perform a replay attack that emulates an NFC tag that contains real credit card information. The NFC Tag found in credit cards is one of the most common ways to steal credit card information as it is a passive device that is not able to easily identify when it is being read from.

2 Related Works

The two papers which we have chosen are different methodologies to implement cloud authentication using NFC tags and the devices scanning them. The first paper focuses on authenticating transactions using NFC payment terminals to initiate the authentication. A hash value is created from a timestamp and a random value from the terminal. The second paper uses an NFC tag instead of a terminal. In this paper, a hash is created from the tag ID and a random number which is stored on the tag itself and updated through the algorithm. We will be focusing on implementing our version of the second paper.

Our first paper is *A Cloud-Based Secure Authentication Protocol for Contactless-NFC Payment*, this paper proposes an authentication protocol for payment systems [3]. This paper proposes an alternative to the standard Europay Mastercard Visa (EMV) protocol currently used around the world to authenticate credit cards using NFC. The second papers we are examining are *Secure and Lightweight Authentication Protocol for NFC Tag-Based Services* which proposes an authentication tool that is lightweight and able to authenticate both tags and devices using a server [2]. The main goal of this protocol is to authenticate a device is at an NFC tag to do something such as open a door or confirm a device's location. Using this paper, we will attempt to duplicate their authentication process using our own NFC Tags and readers.

The two papers mentioned are using the same technology to authenticate the information transferred between NFC Tags and NFC Readers/Payment systems. Both protocols proposed are trying to be secure to protect people's data while also being efficient. This is an important security topic as many Asian, European countries, as well as Canada, have been using NFC payment protocols to authenticate credit and debit cards. As this becomes more commonplace, more effort will need to be put in place to secure these systems and make sure.

3 Development Technology

For the client-side hardware, we used a Raspberry Pi with the PN532 NFC/RFID controller breakout board. This acts as our NFC-enabled device which will be used to read and write to the tags. The tags we used are generic brand NFC tags available on Amazon.ca. We implemented the client-side functionality using Python and the NFCpy library to read and write information from NFC tags. It also allowed us to be able to cleanly parse and manipulate the data within our code. The Raspberry Pi is running the Raspberrian OS that is designed to be used on the Pi. After reading through documentation on the PN532 NFC/RFID board we were able to connect to the

board using UART and the Raspberry Pi's serial port. Once this was achieved, the NFCpy library was able to connect to the device using the serial port's tty path.

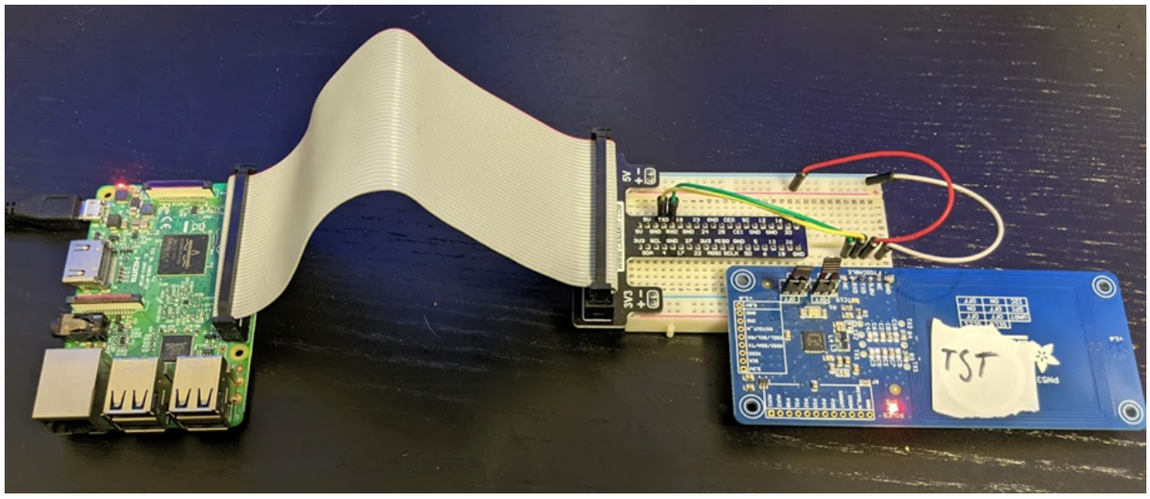

Figure 1: Client-side hardware setup

The NodeJS server is also running on the Raspberry Pi. The NodeJS server will act as the main authentication system that will receive data from the NFC-enabled client. The API functions replicate the algorithm that is outlined within the paper. Our API is able to check the hash values sent from the device to compare the values stored in the database then the API verifies the device, generates a random value to write to the tag, generates a response hash for the device, and generates a hash that contains the new tag value. Most of the computing power is done on the server-side as the hashes are computed and compared to be valid on the server.

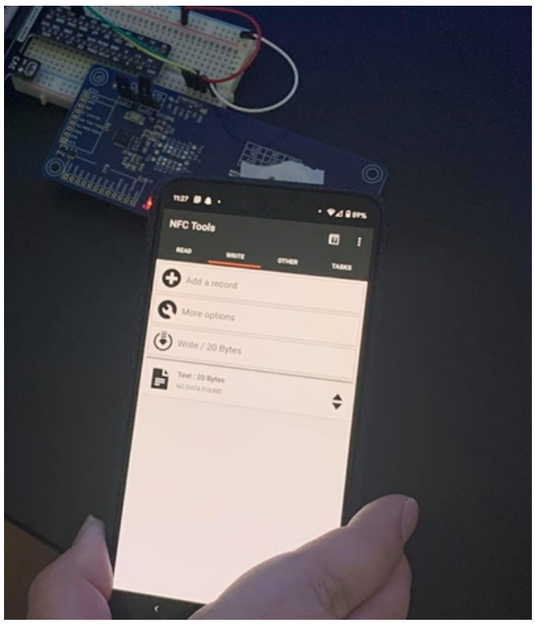

Figure 2: NFCTools editing tag information

For the database, all necessary information needed to be stored to authenticate these components use the SQLite3 database on the same Raspberry Pi as the server. We used a NodeJS module called sqlite3 which allows us to interact with the SQLite3 database on the server. This module allows us to be able to read, write, delete, and update entries within the database. The database is read and written to from the NodeJS server that sends information to the client device for authentication.

4 Implementation Details

4.1 Server-side Implementation

The server which we built for the project was built using NodeJS, using custom helper classes, modules from npm and built in NodeJS modules. Three helper classes were programmed for the server; binaryHelper.js, hashHelper.js and dbHelper.js with each dealing with the type of

functionality their names suggest. binaryHelper.js was written to deal with all of the binary requirements of the routes, hashHelper.js for dealing with calculating the hash and converting a hash back to its original data and dbHelper.js for dealing with all of database requests. These files were created to abstract away some of the functionality and make the code within the route files easier to read, as well as make adding new functionality to the program cleaner and easier. The npm and NodeJS modules which we used in the server are express.js and the http module.

The design of the server was to have HTTP routes available as an API which provided the functionality of the algorithm outlined in the paper. Other helper routes were added as well to allow us to more easily interact with the database and server as it is on a standalone Raspberry Pi and manually adding or changing data within the Raspberry Pi's database can be an involved process. In total five routes were exposed in the API we developed for our project and the functionality of each is listed below.

4.1.1 /auth/tag

Input : { `tReq: Integer, alpha: Integer` }
Output : { `tRes: Integer, beta: Integer` }

The purpose of this route is for the verification of a tag through the supplied integers `tReq` and `alpha`. `tReq` is a hashed integer containing the information of the `tagID` as well as the `rs_i` (random value stored on the tag). `Alpha` is the integer result of performing the operation `rs_i` XOR `r_t` (a random number generated by the client).

In order to authenticate the tag as valid, `tReq` is first run through a function to retrieve the unhashed integer representing the data. This integer is then converted into a binary string of length 14 with the left most 8 bits representing the `tagID` in binary and the remaining 6 bits representing the `rs_i` in binary. These two binary strings are converted to their respective integer values. The server checks the database to ensure that they correspond to the value stored in the database, returning an error if either the `tagID` doesn't exist, or the `rs_i` doesn't correspond to the `tagID` in the database.

Now that the tag is verified, the server generates a new random value (`rs_i1`) in order to be passed back to the client to be written to the tag. `r_t` is extracted from `alpha` by performing `rs_i` XOR `alpha`. Using this extracted `r_t`, `tRes` is calculated using `rs_i1` XOR `r_t`. `Beta` is calculated by appending the binary strings of `rs_i1`, `r_t` and `rs_i` together after ensuring each binary string was 6 digits long, appending zeros to the left side if needed. This 18 digit binary string is now converted to an integer, then run through the hash function. The database now updates the random number for `tagID` as `rs_i1` and then returns a json object containing `tRes` and `beta` as a response to the request.

4.1.2 /auth/client

Input : { `tReq : Integer, dReq : Integer, p : Integer` }
Output : { `tRes : Integer, dRes : Integer, alpha : Integer` }

The purpose of this route is for the verification of authenticity of both a client device and a tag using the supplied integers `tReq`, `dReq` and `p`. `tReq` is a hashed value representing the `tagID` and `rs_i` (tag random value) where `dReq` is a hashed value representing the `clientID` and `rd_i` (client random value). `p` is an integer such that $L/2 < p < L$ where L is the maximum binary string length of a `tagID` and `clientID`, in our case $L = 8$.

In order for the server to verify that both the tag and client are valid it first runs both `tReq` and `dReq` through a function to read the hashed data contained within each, then both are converted to 14 digit binary strings. `tagID` and `clientID` binary strings are extracted from `tReq` and `dReq` respectively by taking the first 8 bits of those binary strings, then converted from binary to integers. `rs_i` and `rd_i` are also extracted from `tReq` and `dReq` respectively using the last 6 bits of the binary string and converting them from binary to integers. The server now uses these values to check the database to ensure that the `clientID` corresponds to a valid client device, `tagID` corresponds to a valid tag, the `rd_i` is the stored random value for that device and that `rs_i` is the same stored random value for that tag. If any of these database checks results in finding mismatched information or a tag/device that doesn't exist, the process is stopped and an error is returned.

Now that the tag and device are both validated, new random values are generated for the tag (`rs_i1`) and client device (`rd_i1`). The server now has to calculate the partial `tagID` (`pid_t`) and partial `clientID` (pid_c). This is done by selecting the `p` least significant (rightmost) bits of the binary strings of `tagID` and `clientID`, then converting the `p` bit long strings back into integers. `tRes` is calculated as `pid_t` XOR `rs_i1` and `dRes` is calculated as pid_c XOR `rd_i1`. `Alpha` is calculated in a couple of steps. The binary strings of `rs_i1`, `pid_t` and `rs_i` are appended in that order. The resulting binary string is converted to an integer, then ran through the hash function. After all of this, the database is updated with the new random values `rs_i1` and `rd_i1` and then a JSON object containing `tRes`, `dRes` and `alpha` is sent as a response to the request.

4.1.3 /initialize-nfc/client

Input : none
Output : { `cid : Integer, crand : Integer` }

The purpose of this route was mostly for testing purposes. It was built only to create ease of use for initializing clients in the database and returning a JSON representation to be assigned to the program running on the client device. In a real world implementation this route would be protected or exist only as a script that administrators had access to.

4.1.4 /initialize-nfc/tag

Input : none
Output : { `tid : Integer, trand : Integer` }

The purpose of this route was mostly for testing purposes. It was built only to create ease of use for initializing tags in the database and returning a JSON representation to be written to the tag being initialized. In a real world implementation this route would be protected or exist only as a script that administrators had access to.

4.1.5 /view-entries

Input : none
Output : { `clients : [ { cid : Integer, crand : Integer } ],`
`tags : [ { tid : Integer, trand : Integer } ]` }

The purpose of this route was mostly for testing purposes. This is a helper route which allows us to easily see what is going on in the database in a returned JSON object containing all clients and tags in the database. This would be insecure in a real world implementation and would not exist publicly.

4.2 Client-side Implementation

The client was developed as a command based implementation. This implementation was decided on to be able to easily update or add features as necessary. Using the commands we are able to do a wide range of tests to confirm that the algorithm is working as expected as well as be able to test for potential attacks. The client is able to create new clients and tags, is able to authenticate an individual tag as well as a tag and client together. We also included functionality to be able perform malicious actions such as corrupt client or tag information for testing purposes. Functionality to display general information such as all tag and client entries in the database and read tag information was also included in the client.

A number of helper functions were written to be able to complete the client implementation. This includes functions to make a binary string a specific length and our custom hash function to complete tasks for the authentication algorithms. We also made functions to make HTTP GET and POST requests to our server that can send and receive data. Finally, two functions to access the NFC card reader were used, one to read and one to write. These functions were developed with the NFCpy library, the NFC card reader is accessed over the Raspberry Pi's serial port which uses Teletype (TTY) protocol to send and receive information. NFCpy is able to connect to the serial port and communicate with the PN532 NFC/RFID controller as the library has support for this device. Then we can sense the tag that is placed on the device and able to read and write to the tag until the connection is closed.

Storing information on the NFC tag itself was a design challenge. We needed something that could be quickly read from and be computer understandable. In the end, we decided to use the text data type and write JSON formatted text to the tag. This allowed us to read in the first text record on the tag and translate it into JSON without having to write additional helper functions since Python works well with JSON. The information stored on the tag is the tag ID and the tag random number. The JSON stored on the tag is `‘‘tid’’: #, ‘‘trand’’, #.`

4.2.1 Create New Client

Utilizing the /initalize-nfc/client server function, this command will create then save a new client authentication to use in the future.

4.2.2 Add a New Tag

This function utilizes /initialize-nfc/tag method on the server. This command will create a new tag on the server then write the information to the tag in JSON format. The resulting data stored on the NFC tag is a text data type with `‘‘tid’’: #, ‘‘trand’’: #`. Saving text that's formatted as JSON allowed for us to quickly retrieve all of the data stored on the tag and could be updated to quickly include new data in the future.

4.2.3 Authenticate Tag & Client

Utilizing the /auth/client server method, this reads the tag information then performs the binary functions to generate the desired information for the server. After receiving the data from the server, we decrypt it and update both the client and tag random numbers for future usage on the server.

4.2.4 Authenticate Tag Only

Using the /auth/tag server method, we first read the tag's data. Then the client performs the binary functions to generate the desired information for the server. After receiving the results from the server, the client performs the calculations to get the tag's new rand number and writes that information to the tag.

4.2.5 View All Entries (Client & Tag)

This command uses the /view-entries server method that returns the results of the database. This will display all of the clients and tags data.

4.2.6 Read Tag Data Only

This command displays the information on the tag by reading the first text record on the tag. We only ever use one data record on the tag.

4.2.7 BAD: Corrupt Tag

This command will write to the tag data that contains the wrong tag random number so that the tag cannot be authenticated.

4.2.8 BAD: Corrupt Client

This command will change the current client's random number so that the client cannot be authenticated.

4.2.9 Exit program

Safely close the program by closing the connection to the NFC Reader.

4.3 Database Design

The database was designed with the standard CRUD operations (create, read, update, delete) in mind. These operations are necessary for initializing, updating and removing the data for NFC tag and client device data. The database consists of two entity sets and one relationship set. There is an entity set for storing client device IDs and shared secrets named clients and the other entity set stores IDs and shared secrets of all tag entities. In each of these sets, the IDs for tags and client devices are used as the primary key as no two tags nor any two client devices should have the same ID. These IDs will never change for any existing tags or client devices, however the update operation is used to update a tag and clients shared secret as needed by the authentication protocol.

The relationship set, denoted as scanned, stores tag-client ID pairs which are used in the authentication process. This set has two main cases where data is retrieved. The first case is to identify client devices which have permission to overwrite the data of a scanned tag. The second case where tag-client ID pairs are needed is when identifying what tags a client device is able to scan. The former is used within the authentication process since when a client device scans a tag, the ID of both the tag and the client device is sent to the server. The latter allows the administrator of the system to retrieve all tags a client device is able to overwrite. As an added feature both tag IDs and client device IDs stored in this table are foreign keys meaning that these objects must exist within either entity set table for an ID pair to be formed and inserted into this table. This prevents unauthorized tag-client ID pairs from being formed. In addition to this feature, the database also makes use of triggers whenever a tag or client device is removed from their respective table. These triggers will remove entries in the scanned table which correspond with the deleted tag or client device object.

The database interacts with the server as follows:

1. Once the received data is processed by the server, each entity set is queried for its respective object.
   (a) If the object is found in its respective table then the data from the table row is used to create an instance of the object using the object's model.
   (b) If the object is not found, then the protocol is aborted.
2. Once both tag and client device objects have been returned, their ID pair is queried from the scanned relationship set
   (a) If the ID pair is found, the ID pair is returned and allows the protocol to continue.
   (b) Otherwise, the protocol is aborted.
3. The database is given update queries to update the scanned tag's shared secret and the client device's shared secret as described in section 4.1

5 Simulation and Results

The tests described below were performed in the sequence of the test number. The tests were also performed on the same server without resetting between usages. The goal of the following twenty tests is to be able to verify that our implementation works as expected, and is able to prevent unauthorized tags and clients from completing the protocol all while allowing authorized tags and clients to complete the protocol. These tests were successful in identifying these attacks and avoiding them.

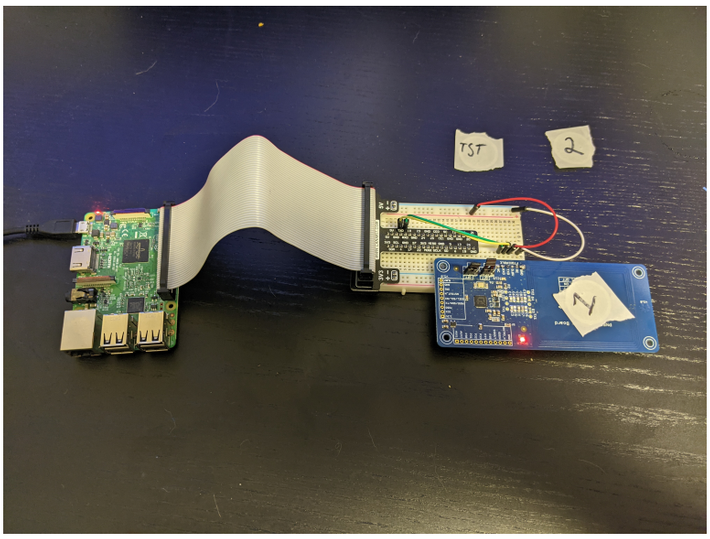

Figure 3: Hardware setup for testing tags

5.1 Verification Tests

Test Number	Test Tag ID	Authentication with Client ID	Results
Test 1	1	1	SUCCESS
Test 2	1	2	SUCCESS
Test 3	1	3	SUCCESS
Test 4	2	2	SUCCESS
Test 5	2	3	SUCCESS
Test 6	2	4	SUCCESS
Test 7	3	1	SUCCESS
Test 8	3	2	SUCCESS
Test 9	3	3	SUCCESS
Test 10	4	1	SUCCESS

The tests performed above were to test the functionality of the client and server as a whole. We used the /auth/client server method to test as it uses both the client and tag information and confirms both at the same time. Success was defined as being able to authenticate the tag and client with the server and correctly updating the random values of both respectfully. Tests one through nine were successful in testing the basic functionality of the developed software by authenticating the client and server at the same time. Test 10 was a confirmation that after test 7 was performed we could re-authenticate the tag and client after additional authentications.

5.2 Corrupted Tag Tests

Test Number	Tag ID	Original Tag Random Number	Corrupted Tag Random Number	Authentication with Client ID	Results
Test 11	1	4	63	4	SUCCESS
Test 12	1	28	63	4	SUCCESS
Test 13	1	63	63	5	SUCCESS
Test 14	1	63	63	5	SUCCESS
Test 15	1	56	63	5	SUCCESS

The goal of tests eleven through fifteen was to identify if a server could identify a tag that had been tampered with by a malicious actor. To do this test we would first confirm that the tag could be authenticated using the /auth/client server method, then we could corrupt the tag by setting the tag random number to sixty-three. After changing the tag's data we would then try to authenticate with a valid client, the expected result was failure. Success is defined as being unable to authenticate the client and tag after the tag's random number had been corrupted/tampered with. For tests 13 and 14, the tags had been previously corrupted in tests 11 and 12 respectively, as such they didn't have a change in the tag random numbers.

5.3 Corrupted Client Tests

Test Number	Tag ID	Authentication with Client ID	Original Client Random Number	Corrupted Client Random Number	Results
Test 16	7	6	33	255	SUCCESS
Test 17	8	6	33	255	SUCCESS
Test 18	4	7	4	255	SUCCESS
Test 19	7	7	4	255	SUCCESS
Test 20	8	7	4	255	SUCCESS

Finally, our last test set, tests sixteen to twenty is to confirm that a corrupted client could not be authenticated on the server even with a valid tag. These tests used the /auth/client method on the server to attempt to authenticate both the client and tag. The testing method was to first confirm that the client could authenticate with a valid tag, then we would corrupt the client's data by changing the client random number to two-hundred-fifty-five. Then we would try to authenticate the client with a valid tag, the expected result should be that the authentication fails.

6 Conclusion

In our project proposal we set out to evaluate the ability of the proposed algorithm to achieve security, and prevent certain types of NFC based attacks. Below we will outline our findings for each of the different attack vectors we had set out to find.

6.1 Protocol adequately protects the user from NFC Data Exchange Format (NDEF) value being overwritten by a malicious actor

Our conclusion is yes the protocol does protect the user from overwritten values on the NFC tags, however "zombie" tags which are not able to be used anymore would be created by the attacker by overwriting values.

Tags on their own being passive are not able to selectively allow themselves to be overwritten. They are either read only or writable, only powered active NFC devices would be able to make the distinction if a host is valid or not to overwrite it. Strictly looking at tags though, this algorithm does protect the user. If any data on the tag has been manipulated, the server will know it is no longer a valid tag as the values do not match the database that the server has access to. It will return an error to the client and the client will do no more computation with the information received from the tag.

6.2 Protocol adequately protects the user from spoofing

Our conclusion is yes the protocol can in fact protect the user from spoofing.

If there is a man in the middle attack being carried out with the proposed algorithm, the only way that they would be able to access the information is if they knew the hash function. In our project we kept it as a simple math function which would probably be easy to crack, but using modern advanced encryption methods such as having a public key for each device it would allow this to become more secure.

If any of the data is modified, there are able to be checks in place in both the client and server to ensure data integrity. For example the server can verify the random tag value using `tReq` and `alpha` as `rs_i` is expressed in both, and the client can verify the new `rs_i1` value sent back using the `r_t` value it initially used, as well as comparing it to the value extracted from beta.

6.3 Protocol can ensure that a Denial of Service (DoS) attack isn't possible.

Our conclusion is that some DoS attacks can be mitigated such as ones made against the server. However other attack vectors such as removing the tags themselves, overwriting values to create "zombie" tags which have to be reformatted by an admin and tag spoofing are all valid ways that a bad actor could attack the application.

Mitigations for a DoS attack on the server can be made in a couple of ways. By putting the server behind services such as Cloudflare or limiting requests made from the same IP within the node server itself it can prevent all of the resources from being used up and making it so that legitimate users are unable to communicate with the server. However one way that this system is susceptible to a DoS attack is by tags being overwritten. With tags being open to being written to by anyone, as soon as someone overwrites the data stored on it, the clients are no longer able to use them anymore.

Another way which a bad actor would be able to attempt a DoS attack would be by tag spoofing. They would be able to copy the data from one tag, to their own tag or NFC device pretending to be a tag. Whenever it was checked by a client, the attacker's tag would be updated with the new value rendering the original legitimate tag useless.

As more tags are overwritten by the bad actor, the user and server would be protected by the algorithm. This would come at the cost of the usability of whatever service is built on as the algorithm would be rendered useless.

To sum up our conclusions, the user and server are protected by the algorithm, allowing both to check if any information was tampered with in transit. It also protects the user and server from information overwritten on tags, however leaving the tags as writable allows a vector for bad actors to execute a denial of service attack by rending the tags, and eventually the service useless.

7 Additional Resources

7.1 GitHub Page

GitHub Page: https://github.com/EricNewcombe/lightweight-nfc-authentication

7.2 Project Demonstration

Project Demonstration: COMP 4203 - Group 5 Final Project Demo
Project Demonstration Presentation: Project FINAL REPORT Pres.pptx

7.3 Proposal Presentation

Proposal Presentation Slides: Group 5 Project Proposal
Proposal Presentation Video: 4203 Group 5 Project Proposal

8 Contributions

8.1 Christian Belair

Designed database, integrated with server-side components, report writing/editing, created the report using latex, and the proposal presentation slideshow.

8.2 Kyle Knobloch

Designed the client, maintained the server & device infrastructure, performed the testing & simulations, report writing/editing, and worked on the PowerPoint.

8.3 Eric Newcombe

Designed Server API, built server routes, implemented verification algorithms on the server, report writing/editing, video, PowerPoint.

References

[1] A. ALBATTAH, Y. ALGHOFAILI, and S. ELKHEDIRI. Nfc technology: Assessment effective of security towards protecting nfc devices services. In *2020 International Conference on Computing and Information Technology (ICCIT-1441)*, pages 1–5, 2020.

[2] J. Baek and H. Y. Youm. Secure and lightweight authentication protocol for nfc tag based services. In *2015 10th Asia Joint Conference on Information Security*, pages 63–68, 2015.

[3] N. El Madhoun, F. Guenane, and G. Pujolle. A cloud-based secure authentication protocol for contactless-nfc payment. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 328–330, 2015.