# Tirocinio Formativo Universitario

Breve elaborato sull’attività svolta durante il tirocinio curricolare interno.

## Obiettivi

Progettazione e realizzazione di un sistema di erogazione di bevande, automatizzato tramite l'utilizzo di tecnologie RFID e componenti comunemente impiegate nel campo dell'IoT.

Tale sistema, consentirà all'utente di effettuare in maniera autonoma una consumazione, una volta riconosciuto, usando la tecnologia dei moduli RFID; inoltre l'utente potrà consultare le proprie consumazioni registrate e saldare l'importo dovuto tramite un applicativo per smartphone.

Il sistema garantirà un livello di sicurezza tramite la cifratura dei dati utente sui moduli RFID e delle trasmissioni tra l'applicativo ed il server che ospita il database del progetto.

Sarà implementata una gestione dei canali di trasmissione tra i vari componenti del sistema adibiti al riconoscimento utente ed erogazione della bevanda, applicando le necessarie misure di QoS.

Verrà implementata la possibilità di avere una visione generale del sistema tramite software di monitoraggio.

## Sicurezza

Per garantire un maggiore livello di sicurezza durante l’uso dei tag/card NFC, bisogna impedire l’accesso a terzi alla loro memoria.

|  |  |
| --- | --- |
| Utilizzare il sistema senza un bicchiere-tag | ❌Le stazioni di erogazione non si attivano senza leggere un tag valido |
| Modificare l’ID del tag | ✅Bisognerebbe essere a conoscenza della struttura dati che ospita gli ID e anche di altri ID, ed avere accesso a strumenti |
| Clonare il tag da un bicchiere | ✅Bisognerebbe avere accesso a strumenti + bicchiere vittima |
| Modificare l’ID di una tessera | ✅ |
| Clonare una tessera |  |
| Intercettare i pacchetti di richiesta DB dall’app |  |
| Intercettare le richieste MQTT | https://techblogs.42gears.com/best-security-practices-in-mqtt-protocol/ |
| Generare richieste malevole MQTT per | Devo implementare la connessione con usr psw al broker |

Attualmente, tengo in considerazione 3 metodi diversi per prevenire ciò:

* Impedire l’accesso alla memoria con uno step auth
* Crittografare l’ID dentro al tag
* Riempire la memoria di byte randomici con lo scopo di rendere più difficile quale parte della memoria viene effettivamente utilizzata per salvare il vero ID utente

## NTAG213

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteConsultando la documentazione su xnp.com, è possibile trovare le specifiche sull’implementazione di uno step di autenticazione. Ci sono alcuni campi chiave circa la protezione con psw:



* La pagina CFG0 (0x29): L’ultimo byte (AUTH0) rappresenta lo stato della protezione con password. In particolare, il valore indica il numero della pagina da cui è richiesta l’autenticazione. Un valore 0xFF rappresenta che il tag non è protetto da psw, mentre un valore 0x00 sì, indicando che tutte le pagine sono soggette alla protezione. [pag 19 su 60 del pdf].
* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamenteLa pagina CFG1 (0x2A): PROT bit in ACCESS byte (primo byte nella pagina).  
  Il valore default per il byte ACCESS è 0x00 🡪 00000000b.  
  Il valore default per il bit PROT è 0b, che indica il tipo di protezione WRITE PROTEXTED by PSW; invece, 1b indica READ AND WRITE PROTECTED by PSW 🡪 10000000b 🡪 0x80



Gli altri byte sulla pagina 0x2A sono per ‘Reserved for future use – implemented’ (RFUI). Gli altri bit in ACCESS sono 0b per default.

* La pagina 0x2B: Contiene una password PWD a 4 byte (32-bit).
* La pagina 0x2C: Contiene un byte PACK che viene tornato dal chip (?) dopo un’autenticazione positiva. È una coppia di byte

Sequenza per crittografare:

1. Scrivi PACK buffer su 0x2C

uint8\_t PACKAddress = 0x2C;

byte PACKbuffer[4];

byte PACKSsize = sizeof(PACKbuffer);

PACKbuffer[0] = 0x80;

for (byte i = 1; i < 4; i++) {

  PACKbuffer[i] = 0x00;

}

status = (MFRC522::StatusCode)mfrc522.MIFARE\_Ultralight\_Write(PACKAddress, &PACKbuffer[0], 4);

1. Scrivi PSW buffer su 0x2B

uint8\_t PSWAddress = 0x2B;

byte PSWbuffer[6];

byte PSWsize = sizeof(PSWbuffer);

memcpy(PSWbuffer, "1234", 4);

status = (MFRC522::StatusCode)mfrc522.MIFARE\_Ultralight\_Write(PSWAddress, &PSWbuffer[0], 4);

1. Scrivi AUTH0 su 0x29

uint8\_t AUTH0Address = 0x29;

byte AUTH0buffer[4] = { 0x04, 0x00, 0x00, 0x00 }; //Per provare dalla pag 0x04

byte AUTH0size = sizeof(AUTH0buffer);

status = (MFRC522::StatusCode)mfrc522.MIFARE\_Ultralight\_Write(AUTH0Address, &AUTH0buffer[0], 4);

1. Scrivi ACCESS byte su 0x2A

uint8\_t ACCESSAddress = 0x2A;

byte ACCESSbuffer[4]={0x80,0x00,0x00,0x00};

byte ACCESSSsize = sizeof(ACCESSbuffer);

status = (MFRC522::StatusCode)mfrc522.MIFARE\_Ultralight\_Write(ACCESSAddress, &ACCESSbuffer[0], 4);

//Alla fine, 0x2A è uguale a 0x2C per errore mio:

// - 0x2C può essere qualsiasi 4 byte, ma ho usato lo stesso valore x sbaglio

// - 0x2A deve essere (0x80 0x00 0x00 0x00) in modo che chieda auth pure per

// eseguire una read

NB: Il bit PROT del byte ACCESS è rimasto al valore default 0b, quindi ora solo la scrittura è bloccata.  
NB2: Anche cambiando la riga 0x2A (la pagina che contiene ACCESS) con [0x80 0x00 0x00 0x00] funziona e blocco la lettura!!!!!!

Per accedere alla memoria e scrivere, è necessario prima autenticarsi

uint8\_t messageAddress = 0x0A;

byte message[4] = { 0xAA, 0xAA, 0xAA, 0xAA }; //Messaggio di prova

byte messagesize = sizeof(message);

byte PSWBuff[] = { 0x31, 0x32, 0x33, 0x34 }; //ogni byte è un char di ‘1234’ in hex

status = (MFRC522::StatusCode)mfrc522.PCD\_NTAG216\_AUTH(&PSWBuff[0], pACK);

if (status == 0) {status\_write = (MFRC522::StatusCode) {  
 mfrc522.MIFARE\_Ultralight\_Write(messageAddress, &message[0], 4);  
 }

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteNB: Scrivo solo se lo status dell’autenticazione è 0 perché corrisponde all’indice di uno stato positivo nell’enum definito dalla libreria



NB: Ora bisogna assumere che ogni tag NFC sia stato precedentemente protetto da psw con PSW XX XX XX XX e PACK YY YY

## Mifare 1K

Adesso rimane da crittografare la memoria della tessera nfc Mifare 1K. Teoricamente si può ovviare questo passaggio aggiungendo un controllo visivo da parte di un gestore umano che controlla i documenti dell’utente e verifica che sia effettivamente l’utente registrato sulla tessera.   
Alternativamente si può aggiungere una verifica di autenticazione tramite password tramite tastiera ma bisognerebbe rivedere la struttura della stazione di trasferimento ID in modo tale che supporti tale funzione.

La memoria del chip MIFARE 1K è strutturata in 16 settori da 4 blocchi ciascuno, ognuno dei quali è composto da 16 byte. All’interno di ogni settore, l’ultimo blocco (il numero 3) è detto *sector trailer*, e le informazioni salvate al suo interno dettano i permessi e le regole di accesso a tale settore.

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Al suo interno, infatti, i byte da 0-5 e 10-15 rappresentano rispettivamente le due chiavi KEY\_A e KEY\_B, usate per autenticare gli accessi al settore. I byte 6-7-8 invece contengono gli *access bits*, che dettano le regole di accesso alla memoria in base alla loro configurazione. Sono salvati sia in forma normale che in forma inversa per ridondanza all’interno dei bytes

Ogni configurazione dei 3 access bits C1, C2 e C3 corrisponde ad un insieme di condizioni di accesso per i 3 blocchi di dati + sector trailer. Un access bit è definito come

Immagine che contiene testo, elettronica, schermata, Dispositivo elettronico

Descrizione generata automaticamenteImmagine che contiene testo, schermata, diagramma, Parallelo

Descrizione generata automaticamenteLa configurazione iniziale dei bytes (6 7 8) è (FF 07 80)16 = (11111111 00000111 10000000)2

Da ciò si deduce che :

* Di default, gli access bits per accedere ad un data block {0,1,2} sono (0 0 0).   
  Dalla tabella delle condizioni di accesso si nota che è necessaria almeno la KEY\_A per ogni operazione che riguardi i data blocks:
* Immagine che contiene testo, schermata, Carattere, linea

  Descrizione generata automaticamenteDi default gli access bits per accedere ad un sector trailer {3} sono (0,0,1).  
  Dalla tabella delle condizioni di accesso al sector trailer si nota che è necessaria almeno la KEY A per ogni operazione, ma tale chiave non può mai essere letta: Siccome invece entrambe possono essere sovrascritte, bisogna fare attenzione perché un tentativo di sovrascrittura malpensato può portare alla corruzione del settore.

Consultando una tabella a parte si può vedere come indifferentemente dalla configurazione degli access bits, non è possibile leggere la chiave A, invece è possibile scrivere entrambe.

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

NB: Le chiavi KEY A e KEY B sono per default FF:FF:FF:FF:FF:FF.

Problema: voglio cambiare key A e B 🡪 devo scrivere nel sector trailer 🡪gli access bits me lo permettono? 🡪 0,0,1 SI 🡪 Scrivo l’intero blocco come (**new** KEY\_A | **old** ACCESS BITS | **new** KEY\_B)

SUCCESSO!

Procedura esperimento:

1. Di default la card ha sia KEY A che KEY B pari a (FF:FF:FF:FF:FF:FF)
2. Con il programma write\_card\_8266\_encrypted, pongo manualmente le chiavi a (FF…FF)
3. Accedo con successo e posso scrivere /leggere ogni campo per via degli access bits
4. Modifico un sector trailer con (FF:FF:FF:FF:FF:AA|FF:07:80|FF|FF:FF:FF:FF:FF:AA)
   1. NB il byte n=9 era 0x69 ma è usato come memoria utente quindi non importa se adesso è 0xFF

Authenticating again using key A...

Reading data from sector trailer block 7 ... (but you cannot see)

Data in sector trailer block 7:

<00 00 00 00 00 00 FF 07 80 69 FF FF FF FF FF FF

Authenticating again using key B...

Writing data into sector trailer 7 ...

>FF FF FF FF FF AA FF 07 80 FF FF FF FF FF FF AA

1. Se provo a leggere i dati del settore non posso accedere, ritorna

Card UID: 21 98 B8 1D

PICC type: MIFARE 1KB

Authenticating using key A...

PCD\_Authenticate() failed: Timeout in communication.

1. Ora scrivo a mano le chiavi come in punto 2, solo che l’ultimo byte sarà 0xAA
2. Accedo con successo, e riscrivo il sector trailer come era in orgine, ovvero solo FF:…:FF
3. Ora se provo a riaccedere con FF:FF:….:AA non va, perché ho scritto nuovamente il sector trailer con le chiavi di default, quindi FFFFF…
4. Dunque è possibile riscrivere le chiavi, impedire la manomissione e lettura dei valori dello UID utente e se c’è la necessità, riscrivere il campo

Quindi ora bisogna modificare tutte le card e mettere KEYA=KEYB=**FF:FF:FF:FF:FF:AA**

## MQTT

Immagine che contiene testo, schermata, software, Icona del computer

Descrizione generata automaticamenteImmagine che contiene testo, schermata, software, computer

Descrizione generata automaticamenteImmagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamentePotenzialmente, una persona con accesso alla rete può praticare il *packet sniffing* e leggere tutti i dati che vengono scambiati in chiaro, in questo caso per esempio, tramite il protocollo MQTT.

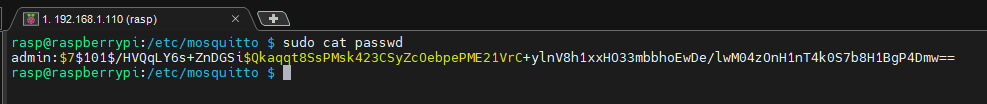


MQTT può usare TSL per criptare i messaggi

Bisogna comunque usare utente e psw per iscriversi al broker (è una cosa diversa ma serve comunque)

Qualcosa qualcosa su nomi dei broker

### USR e PSW MQTT

Per implementare l’autenticazione nel protocollo mqtt, devo creare un broker che accetta connessioni solo da client che forniscono utente e psw corretti.   
Con il comando mosquitto\_passwd -c nomefile nomeutente appare un prompt per una password. Una volta inserita, si crea automaticamente un file con le credenziali nel formato nomeutente:password, dove il campo password è però criptato. 

Nel mio caso ho usato le credenziali:

admin:password

Successivamente, si modifica il file mosquitto.conf

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

È importante mettere in commento la riga include\_dir se come ho io, ci sono altri file conf che cozzano.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamentesia publisher che subscriber si sono autenticati prima di connettersi

Mentre all’utente che non si autentica viene rifiutata la connessione

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

### Setting up certificates

Creazione dei certificati :

<https://openest.io/en/services/mqtts-how-to-use-mqtt-with-tls/>

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

Prima bisogna generare i certificati CA (cert. Authority):

cd certs

mkdir ca broker client

cd ca

openssl req -new -x509 -days 365 -extensions v3\_ca -keyout ca.key -out ca.crt

>> passphrase : abc123

>> CN : arzon

Ora bisogna creare i certificati per il broker MQTT

cd broker

openssl genrsa -out broker.key 2048

openssl req -out broker.csr -key broker.key -new

>> CN : 192.168.1.110 (RPi IP address)

openssl x509 -req -in broker.csr -CA ../ca/ca.crt -CAkey ../ca/ca.key -CAcreateserial -out broker.crt -days 365

>> passphrase : abc123

Ora bisogna creare i certificati per i client

mkdir client

openssl genrsa -out client.key 2048

openssl req -out client.csr -key client.key -new

>> CN : localhost (o altro penso)

openssl x509 -req -in client.csr -CA ../ca/ca.crt -CAkey ../ca/ca.key -CAcreateserial -out client.crt -days 365

>> passphrase : abc123

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamenteImmagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamente

Logicamente si deve creare generare certificati per ogni client, pc esp32,esp8266, ma avendo i file CA e la passphrase è possibile farlo da ogni terminale..

Adesso posso utilizzare i soliti comandi per comunicare:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Immagine che contiene testo, software, Software multimediale, Icona del computer

Descrizione generata automaticamente

Adesso bisogna provare a comunicare da esp32/8266 a broker mqtt usando tls:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteSiccome il broker vuole verificare anche i certificati del client, oltre ad estrarre il contenuto del certificato autorità ca.crt, avrò bisogno pure del certificato client e chiave client.

Copio la stringa dal terminale con i comandi:

sudo cat ca.crt

sudo cat client.crt

sudo cat client.key

E con la libreria WiFiClientSecure.h sullo scketch, creo le istanze

  WIFIclient.setCACert(ca\_cert);

  WIFIclient.setCertificate(client\_cert);

  WIFIclient.setPrivateKey(client\_key);

Che mi permettono di avviare la procedura di handshake TLS/SSL per la creazione di chiavi di sessione, che mi permettono di avere una trasmissione criptata.

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

Immagine che contiene testo, software, schermata, Software multimediale

Descrizione generata automaticamente

TODO:risolvere

CHATGPT: ESTABLISH A SECURE CONNECTION BETWEEN RASP MQTT BROKER AND MULT ESP32 USING TLS

To establish a secure connection between a Raspberry Pi functioning as an MQTT broker and multiple ESP32 boards, you'll need to follow these general steps:

1. **Setting Up TLS on the MQTT Broker (Raspberry Pi):**✅
   * Install and configure an MQTT broker on the Raspberry Pi. Popular options include Mosquitto and VerneMQ.
   * Enable TLS support on the MQTT broker. This involves generating or obtaining TLS certificates (public/private key pairs) for the broker.
   * Configure the MQTT broker to use the TLS certificates for secure communication.
2. **TLS Setup on ESP32:**
   * Implement TLS support in the ESP32 code using appropriate libraries like WiFiClientSecure or BearSSL.
   * Store the required TLS certificates (root CA certificate, server certificate, and private key if necessary) securely on the ESP32.
   * Set up the ESP32 MQTT client to establish a TLS-secured connection with the MQTT broker. Use the proper TLS settings and credentials (e.g., server address, port, client certificates if needed).
3. **Code Implementation (ESP32):**
   * Write code on the ESP32 to connect to the MQTT broker using TLS-secured MQTT protocol. Ensure that the ESP32 MQTT client is configured with TLS parameters.
4. **Certificate Management:**
   * Manage the TLS certificates securely. Ensure that the certificates are valid, not expired, and properly stored on both the MQTT broker and the ESP32 devices.
   * Consider using secure storage mechanisms on the ESP32 for storing sensitive information like private keys and certificates.
5. **Testing and Troubleshooting:**
   * Test the connection between the ESP32 boards and the MQTT broker to ensure that the TLS-secured communication is working correctly.
   * Debug any potential issues that may arise during the connection setup or data exchange.
6. **Security Considerations:**
   * Regularly update and renew TLS certificates to maintain security.
   * Implement additional security measures such as using unique client identifiers, implementing access control, and firewall configurations for better security.

Remember, configuring secure communication involves managing certificates, keys, and configuring the TLS stack correctly on both ends (MQTT broker and ESP32 devices). Follow best practices for secure communication and ensure proper handling of sensitive information to establish a robust and secure connection.

# Approfondimento funzionamento ISO14443somethingsomething

The [ISO/IEC 14443](http://web.archive.org/web/20120614185731/http:/www.waazaa.org/download/) standard is a four-part international standard for contact-less smart cards operating at 13.56 MHz in close proximity (~10cm) with a reader antenna. This ISO standard describes the modulation and transmission protocols between card and reader to create interoperability for contact-less smart card products. There are two main communication protocols supported by the ISO/IEC 14443 standard, they are addressed as Type A and Type B

The ISO 14443 consists in 4 parts:

1. ISO/IEC 14443-1 defines the physical size of ISO/IEC compliant PICC and antennas
2. 14443-2 defines the frequency, the modulation and coding. It is split up in type A /B
3. 14443-3 defines the start of the communication and how to select the PICC (initialization and anticollision seq). It is split into type A /B
4. 14443-4 defines the protocol for data exchange between PCD and PICC

Mifare 1k e NTAG213 sono type A

The PCD typically polls for PICCs in the field. This is done with the REQA. When a PICC is within the operating range of the PCD and receives the REQA, any MIFARE PICC returns the ATQA (It answers with ATQA also if an WUPA is issued, which I did in the code)

The library uses the ATQA to identify the chip type, although it is not recommended as when reading multiple chips at the same time could result in a collision and thus, reading incorrectly the data.

**Note**: In the case two or more MIFARE PICCs are in the operating field of the PCD at the same time, the received (combined) ATQA might contain “collisions” even if there is an anti-collision sequence.

Guarda su quaderno per schema e spiegazione sequenze del codice libreria