Mifare DESFire - An Introduction

David Coelho
Java Senior Software Engineer
Veröffentlicht: 19. Mai 2019

+ Folgen

This article will provide an introduction to MIFARE DESFire, using a blank DESFire EV1 card tracking the authentication steps from the theory to the code. Throughout the article the authentication process will be explained and the Java source code used on this article will be shared, then at the end you will be able to get a blank card and a card reader and test it on your own.

MIFARE is the NXP Semiconductors owned trademark of a series of chips used in contactless smart cards. MIFARE DESFire are cards that comply to parts 3 and 4 of ISO/IEC 14443-4 Type. The *DES* in the name refers to the use of a DES, two-key 3DES, three-key 3DES and AES encryption; while *Fire* is an acronym for *Fast, innovative, reliable, and enhanced*. Subtypes: MIFARE DESFire EV1, MIFARE DESFire EV2. These cards are so called “stored value” cards, you cannot install and execute

management, hospitality, event ticketing, loyalty, etc.

Card Features

The memory organization of DESFire supports up to 28 applications on the card and up to 32 files in each application. The file can be of five different types (Standard Data File, Backup Data File, Value File, Linear Record File and Cyclic Record File)

The native commands are wrapped inside ISO 7816 APDU as described below:

```
Request:
cls ins          p1 p2 lc [data] le
90  [native ins] 00 00 lc [data] 00
```



```
sw1 sw2
91  [native status code]
```

MIFARE DESFire Authentication Process

Authentication between PICC and PCD can be done employing either DES or AES keys. Three pass authentication guarantee that both parties (PCD and PICC) are owner of a common secret (DES/2K3DES/3K3DES/AES key). After the three pass authentication a session key will be generated and this session key will be used to protect the data transmission channel. The authentication steps are detailed below.

Hands On

Lets write some code to see how the authentication works!

Before authenticate on the card, we will execute a command that requires a previous authentication. Below it is the GetCardUID APDU and as there is no authentication provided it is expected a response error.

```
/**
    APDU explained:

    90  51 00 00 00
    cls |  p1 p2 le
        |
        GetCardUID command

    91  ae
    SW1 Authentication error

**/
```

Now lets start the authentication, requesting the challenge to the card.

```
>> Request APDU: 901a0000010000
<< Response APDU: 91af4e34c1314d385402


/**
    APDU explained:

    90  1a 00 00 01 00    00
    cls |  p1 p2 le keyNo le
        |
        Request Authentication Challenge command

    91  af 4e34c1314d385402
    SW1 |  authentication challenge (encrypted rndB)
        this status means that additional data is expected to be se

**/
```

```
// Received challenge on the last step
byte[] challenge = decodeHex("939d2c2ea16575d5");

// Of course the rndA shall be a random number,
// but we will use a constant number to make the example easier.
byte[] rndA = decodeHex("0001020304050607");

// This is the default key for a blank DESFire card.
// defaultKey = 8 byte array = [0x00, ..., 0x00]
byte[] defaultKey = decodeHex("0000000000000000");

// Decrypt the challenge with default key
// rndB: ea485013d80a0567
byte[] rndB = decrypt(challenge, defaultKey);

// Rotate left the rndB
// leftRotatedRndB: 485013d80a0567ea
byte[] leftRotatedRndB = rotateLeft(rndB);

// Concatenate the RndA and rotated RndB
// rndA_rndB = 0001020304050607 + 485013d80a0567ea
byte[] rndA_rndB = concatenate(rndA, leftRotatedRndB);

// Encrypt the bytes of the last step to get the challenge answer
// challengeAnswer = 69178b938c03edf186d3056bedc8d6cf
byte[] challengeAnswer = encrypt(rndA_rndB, defaultKey);

>> Request APDU: 90af00001069178b938c03edf186d3056bedc8d6cf00
<< Response APDU: 91000479ed6c4f74da4a

/**
    APDU explained:

    90  af 00 00 10 69178b938c03edf186d3056bedc8d6cf 00
    cls    p1 p2 le challenge answer                le
```

In the last step we answered the challenge and received the success status from the card, which means that the card opened and validated that the rndB (random number generated by the card) was properly decrypted. Alongside with the success status, the card also sent the rndA, now we have to decrypt it and check if the card also processed the rndA properly, if so we have the proof that both sides have the same key!

```
// encrypted rndA from Card, returned in the last step
byte[] encryptedRndAFromCard = decodeHex("0479ed6c4f74da4a");

// Decrypt the rnd received from the Card.
// rndAFromCard: 0102030405060700
byte[] rotatedRndAFromCard = decrypt(encryptedRndAFromCard, default

// As the card rotated left the rndA,
// we shall un-rotate (rotate right) the bytes in order to get comp
// rndAFromCard: 0001020304050607
byte[] rndAFromCard = rotateRight(rndAFromCard);

// Last step is just to verify that the Card sent the correct rndA
if (Arrays.equals(rndA, rndAFromCard)) {
    log("Authenticated!!!");
}
```

Putting all together

Below there is the complete Java 8 code used in this article. To make it simple, any external library was used and it is one single Java file code. In this way it is easier to you get the code and run in your own machine. For this article it is being used the

```
import javax.crypto.*;
import javax.crypto.spec.*;
import javax.smartcardio.*;
import java.security.*;
import java.util.Arrays;
import java.util.List;

public class Main {

    public static void main(String[] args) throws Exception {

        CardChannel channel = createCardChannel();
        try {

            ResponseAPDU response;

            /*
             * Sending the GetCardUID APDU: 9051000000
             * This command requires a previous authentication, as
             * the card shall return '0x91 0xae', which means Authe
             */
            response = channel.transmit(new CommandAPDU(new byte[]{
            log("Response from GetCardUID without authentication (9

            /*
             * Sending the GetChallenge APDU: 901a0000010000
             * This is the starting point of the authentication.
             */
            response = channel.transmit(new CommandAPDU(new byte[]{
            log("Authentication challenge (8 bytes challenge + 91af

            byte[] challenge = response.getData();

            // Of course rndA is expected to a random number. But f

            // DESFire Default DES key is zero byte array.byte[] de
            byte[] IV = new byte[8];
```

```
log("Left rotated rndB: " + toHexString(leftRotatedRndB

// Concatenate the RndA and rotated RndBbyte[] rndA_rnd
log("rndA and rndB: " + toHexString(rndA_rndB));

// Encrypt the bytes of the last step to get the challe
log("Challenge answer: " + toHexString(challengeAnswer)
IV = Arrays.copyOfRange(challengeAnswer, 8, 16);

/*
  Build and send APDU with the answer. Basically wrap t
  The total size of apdu (for this scenario) is 22 byte
  > 0x90 0xAF 0x00 0x00 0x10 [16 bytes challenge answer
*/byte[] challengeAnswerAPDU = new byte[22];
challengeAnswerAPDU[0] = (byte)0x90; // CLS
challengeAnswerAPDU[1] = (byte)0xAF; // INS
challengeAnswerAPDU[2] = (byte)0x00; // p1
challengeAnswerAPDU[3] = (byte)0x00; // p2
challengeAnswerAPDU[4] = (byte)0x10; // data length: 16
challengeAnswerAPDU[challengeAnswerAPDU.length - 1] = (
System.arraycopy(challengeAnswer, 0, challengeAnswerAPD
log("Challenge Answer APDU: " + toHexString(challengeAn

/*
 * Sending the APDU containing the challenge answer.
 * It is expected to be return 10 bytes [rndA from the
 */
response = channel.transmit(new CommandAPDU(challengeAn
log("Response for challenge answer (10 bytes expected):

/*
 * At this point, the challenge was processed by the ca
 * Now we need to check if the RndA sent by the Card is
 */// encrypted rndA from Card, returned in the last st

// Decrypt the rnd received from the Card.byte[] rotate
log("Rotated rndA from Card: " + toHexString(rotatedRnd
```

```
                System.err.println(" ### Authentication failed. ###
                log("rndA:" + toHexString(rndA) + ", rndA from Carc
            }

        } finally {
             channel.getCard().disconnect(false);
        }

    }

    private static CardChannel createCardChannel() throws CardExcep

        log("Opening channel...");

        TerminalFactory factory = TerminalFactory.getDefault();
        List<CardTerminal> terminals = factory.terminals().list();

        log("Terminals: " + terminals);

        CardTerminal terminal = terminals.get(0);

        Card card = terminal.connect("T=1");

        byte[] atr = card.getATR().getBytes();
        log("ATR: " + toHexString(atr));

        return card.getBasicChannel();
    }

    private static void log(String msg) {
        System.out.println(msg);
    }

    /***
     * Given a byte array, convert it to a hexadecimal representati
     *
     * @param data: Byte Array
     * @return String containing the hexadecimal representation
```

```
        }
        return hexString.toString();
    }

    private static byte[] decrypt(byte[] data, byte[] key, byte[] I
        Cipher cipher = getCipher(Cipher.DECRYPT_MODE, key, IV);
        return cipher.doFinal(data);
    }

    private static byte[] encrypt(byte[] data, byte[] key, byte[] I
        Cipher cipher = getCipher(Cipher.ENCRYPT_MODE, key, IV);
        return cipher.doFinal(data);
    }


    private static Cipher getCipher(int mode, byte[] key, byte[] IV
        Cipher cipher = Cipher.getInstance("DES/CBC/NoPadding");
        SecretKeySpec keySpec = new SecretKeySpec(key, "DES");
        IvParameterSpec algorithmParamSpec = new IvParameterSpec(IV

        cipher.init(mode, keySpec, algorithmParamSpec);

        return cipher;
    }

    private static byte[] rotateLeft(byte[] data) {
        byte[] rotated = new byte[data.length];

        rotated[data.length - 1] = data[0];

        for (int i = 0; i < data.length - 1; i++) {
            rotated[i] = data[i + 1];
        }
        return rotated;
    }

    private static byte[] rotateRight(byte[] data) {
        byte[] unrotated = new byte[data.length];
```

```
            unrotated[0] = data[data.length - 1];
            return unrotated;
        }

        private static byte[] concatenate(byte[] dataA, byte[] dataB) {
            byte[] concatenated = new byte[dataA.length + dataB.length];

            for (int i = 0; i < dataA.length; i++) {
                concatenated[i] = dataA[i];
            }

            for (int i = 0; i < dataB.length; i++) {
                concatenated[dataA.length + i] = dataB[i];
            }

            return concatenated;
        }

    }
```

See you soon!

Hope you enjoy the article. The authentication is maybe the most important part of the MIFARE DESFire, from now I will write another article covering more aspects of the card.

Gefällt mir · Antworten | 1 Reaktion

Maxie D. Schmidt 2 Jahre
Research Assistant at Georgia Institute of Technology

@David Cohelo
I think that the figure might duplicate steps 7.3 and 7.4? Do you mean to rotate left the received (right rotated) RndA back twice, or just once before you encrypt it and send it back from PICC to tag? I just had it once in my implementation.

Gefällt mir · Antworten

Mohan LALE 3 Jahre
Senior Software Engineer Technical Lead at ImproveID

Very helpful articles, where i can get datasheet for remaining commands David Coelho?

Gefällt mir · Antworten | 1 Reaktion

Renan Silva 3 Jahre
Backend Software Engineer at Tillster, Inc.

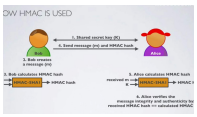
Very well structured and clear. Good job!

Gefällt mir · Antworten | 1 Reaktion

Weitere Kommentare anzeigen

Zum Anzeigen oder add a comment einloggen

Weitere Artikel von dieser Person



HMAC-Hashed Message Authentication Code

Nikhil Goyal · 2 Jahre

The Truth About Card Cloning

CDVI UK · 7 Monate