Solution to Exercise #3

|  |  |
| --- | --- |
| Authors | |
| **Name** | **Matriculation Number** |
| Beshoy Saad | 2572741 |
| Vitalij Minor | 2565427 |

# Bluetooth Security for Noobs

1. The vulnerabilities are:
   1. There is no mechanism in the protocol to verify the identity of the server or the client, making it easy for an attacker to impersonate either (especially the server).
   2. The information over which the hash is calculated (except the secret key) is sent from client to server over the air (a non-secure medium) **in the clear** (without encryption). Among this information, revealing the **nonce** was the biggest vulnerability (*mat1* and *mat2* are already known) as it reduced the brute-force effort to discover the secret by calculating all possible hashes from the length of the nonce + secret (16 + 4) × 8 = 160 bits to only the length of the secret 32 bits. This, along with the fact that the client-server pair does not share an internal state (e.g. a counter) that is also used in calculating the hash, enables the hash to be **replayed** by an attacker.

We discovered the vulnerabilities by reading the protocol description. The vulnerabilities can be exploited by:

1. Masquerading as a client or a server to recover information.
2. Using the recovered information to launch a replay then a brute-force attack to discover the key.

More specifically, the attack goes as follows:

1. We use an off-the-shelf BLE app (e.g. nRF Connect) to scan for BLE servers, find the target server, and learn its Service and Characteristic UUIDs.
2. We develop our own BLE server to masquerade as the legitimate server. We turn off the legitimate server and replace it with our own.
3. Once the legitimate client discovers our server, it begins sending it payloads of the form {mat1, mat2, nonce, h}. We store all the payloads.
4. We go through the stored payloads and recover one that has our matriculation numbers in mat1 and mat2. Now we have a valid hash that we can replay to the server along with its associated nonce.
5. We turn off our server, power up the legitimate server, turn off the legitimate client, and replace it with our own client.
6. We use our client to connect to the legitimate server and send it our stored payload {mat1, mat2, nonce, h}. Since this is a valid payload, the server replies with r = f XOR sk, where f is our flag and sk is the secret key.
7. Now to recover the flag we need the secret key sk. We can discover sk by running a script that calculates h’ = sha256(mat1 || mat2 || nonce || sk’), trying all possible values of sk’, and comparing h’ to our stored hash h. Once we have h’ == h, the value of sk’ that was used to generate h’ is the secret key sk.
8. We calculate our flag as f = r XOR sk’.
9. It is possible to retrieve the key using the described method; however, the attacker will need to perform two exchanges for each value of sk in order to differentiate between a random reply (which changes between the first and second tries, indicating that sk is incorrect) and a valid flag XOR sk reply (which wouldn’t change between tries). Mounting this attack would take 250 ms × 2 × 232 = 2 147 483 648 seconds = approx. 68 years to go through all values of sk, so on average the time needed would be around 34 years, which is beyond the due date of this exercise.
10. We would like to point out that the code we used is based on the examples provided with the Arduino ESP32 library.

## BLE Server

#include <Arduino.h>

#include <BLEDevice.h>

#include <BLEUtils.h>

#include <BLEServer.h>

#include <BLE2902.h>

#define SERVICE\_UUID "f7826da6-4fa2-4e98-8024-bc5b71e0893e"

#define CHARACTERISTIC\_UUID "b9407f30-f5f8-466e-aff9-25556b57fe6d"

bool written = false;

BLECharacteristic \*pCharacteristic;

uint8\_t test\_value[] = {0xA, 0xB, 0xC, 0xD};

class MyCallbacks : public BLECharacteristicCallbacks

{

void onWrite(BLECharacteristic \*pCharacteristic)

{

std::string value = pCharacteristic->getValue();

Serial.print("Received ");

Serial.print(value.length());

Serial.println(" bytes");

Serial.println(value.c\_str());

written = true;

}

void onRead(BLECharacteristic \*pCharacteristic)

{

Serial.println("Read!");

}

};

class MyServerCallbacks : public BLEServerCallbacks

{

void onConnect(BLEServer \*pServer)

{

Serial.println("Connected!");

}

};

void setup()

{

Serial.begin(115200);

BLEDevice::init("Emsec2019-Server1");

BLEServer \*pServer = BLEDevice::createServer();

BLEService \*pService = pServer->createService(SERVICE\_UUID);

pCharacteristic = pService->createCharacteristic(CHARACTERISTIC\_UUID,

BLECharacteristic::PROPERTY\_READ

| BLECharacteristic::PROPERTY\_WRITE

| BLECharacteristic::PROPERTY\_NOTIFY);

pCharacteristic->setCallbacks(new MyCallbacks());

pCharacteristic->addDescriptor(new BLE2902());

pCharacteristic->setValue(test\_value, 4);

pServer->setCallbacks(new MyServerCallbacks());

pService->start();

BLEAdvertising \*pAdvertising = pServer->getAdvertising();

pAdvertising->addServiceUUID(SERVICE\_UUID);

pAdvertising->start();

}

void loop()

{

if (written == true)

{

written = false;

test\_value[0]++;

pCharacteristic->setValue(test\_value, 4);

pCharacteristic->notify();

}

delay(10);

}

## BLE Client

#include <Arduino.h>

#include "BLEDevice.h"

#include <base64.h>

#include "mbedtls/md.h"

// The remote service we wish to connect to.

static BLEUUID serviceUUID("f7826da6-4fa2-4e98-8024-bc5b71e0893e");

// The characteristic of the remote service we are interested in.

static BLEUUID charUUID("b9407f30-f5f8-466e-aff9-25556b57fe6d");

static bool doConnect = false;

static bool connected = false;

static bool doScan = false;

static BLERemoteCharacteristic \*pRemoteCharacteristic;

static BLEAdvertisedDevice \*myDevice;

static void notifyCallback(

BLERemoteCharacteristic \*pBLERemoteCharacteristic,

uint8\_t \*pData,

size\_t length,

bool isNotify)

{

Serial.print("Notify callback for characteristic ");

Serial.print(pBLERemoteCharacteristic->getUUID().toString().c\_str());

Serial.print(" of data length ");

Serial.println(length);

Serial.print("data: ");

for (size\_t i = 0; i < length; i++)

{

Serial.print(pData[i], HEX);

}

Serial.println();

// The secret we recovered using the brute-force calculation

uint8\_t secret[] = {0x39, 0x64, 0x4F, 0x07};

Serial.print("Flag: ");

for (size\_t i = 0; i < 4; i++)

{

Serial.print(pData[i] ^ secret[i], HEX);

}

Serial.println();

}

class MyClientCallback : public BLEClientCallbacks

{

void onConnect(BLEClient \*pclient)

{

}

void onDisconnect(BLEClient \*pclient)

{

connected = false;

Serial.println("onDisconnect");

}

};

bool connectToServer()

{

Serial.print("Forming a connection to ");

Serial.println(myDevice->getAddress().toString().c\_str());

BLEClient \*pClient = BLEDevice::createClient();

Serial.println(" - Created client");

pClient->setClientCallbacks(new MyClientCallback());

// Connect to the remove BLE Server.

pClient->connect(myDevice); // if you pass BLEAdvertisedDevice instead of address, it will be recognized type of peer device address (public or private)

Serial.println(" - Connected to server");

// Obtain a reference to the service we are after in the remote BLE server.

BLERemoteService \*pRemoteService = pClient->getService(serviceUUID);

if (pRemoteService == nullptr)

{

Serial.print("Failed to find our service UUID: ");

Serial.println(serviceUUID.toString().c\_str());

pClient->disconnect();

return false;

}

Serial.println(" - Found our service");

// Obtain a reference to the characteristic in the service of the remote BLE server.

pRemoteCharacteristic = pRemoteService->getCharacteristic(charUUID);

if (pRemoteCharacteristic == nullptr)

{

Serial.print("Failed to find our characteristic UUID: ");

Serial.println(charUUID.toString().c\_str());

pClient->disconnect();

return false;

}

Serial.println(" - Found our characteristic");

// Read the value of the characteristic.

if (pRemoteCharacteristic->canRead())

{

std::string value = pRemoteCharacteristic->readValue();

Serial.print("The characteristic value was: ");

for (int i = 0; i < 4; i++)

{

Serial.print(value[i], HEX);

}

Serial.println();

}

if (pRemoteCharacteristic->canNotify())

pRemoteCharacteristic->registerForNotify(notifyCallback);

connected = true;

}

/\*\*

\* Scan for BLE servers and find the first one that advertises the service we are looking for.

\*/

class MyAdvertisedDeviceCallbacks : public BLEAdvertisedDeviceCallbacks

{

/\*\*

\* Called for each advertising BLE server.

\*/

void onResult(BLEAdvertisedDevice advertisedDevice)

{

Serial.print("BLE Advertised Device found: ");

Serial.println(advertisedDevice.toString().c\_str());

// We have found a device, let us now see if it contains the service we are looking for.

if (advertisedDevice.haveServiceUUID() && advertisedDevice.isAdvertisingService(serviceUUID))

{

BLEDevice::getScan()->stop();

myDevice = new BLEAdvertisedDevice(advertisedDevice);

doConnect = true;

doScan = true;

} // Found our server

} // onResult

}; // MyAdvertisedDeviceCallbacks

void setup()

{

Serial.begin(115200);

Serial.println("Starting Arduino BLE Client application...");

BLEDevice::init("");

// Retrieve a Scanner and set the callback we want to use to be informed when we

// have detected a new device. Specify that we want active scanning and start the

// scan to run for 5 seconds.

BLEScan \*pBLEScan = BLEDevice::getScan();

pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());

pBLEScan->setInterval(1349);

pBLEScan->setWindow(449);

pBLEScan->setActiveScan(true);

pBLEScan->start(5, false);

// Use unconnected analog pin as source of randomness for seed

randomSeed(analogRead(A0));

} // End of setup.

// This is the Arduino main loop function.

void loop()

{

// If the flag "doConnect" is true then we have scanned for and found the desired

// BLE Server with which we wish to connect. Now we connect to it. Once we are

// connected we set the connected flag to be true.

if (doConnect == true)

{

if (connectToServer())

{

Serial.println("We are now connected to the BLE Server.");

}

else

{

Serial.println("We have failed to connect to the server; there is nothin more we will do.");

}

doConnect = false;

}

// If we are connected to a peer BLE Server, update the characteristic each time we are reached

// with the current time since boot.

if (connected)

{

/\*

// Replay the stored mat1, mat2, nonce, h to the server. This is done to retrieve our encrypted flag.

String newValue = "{mat1:2565427,mat2:2572741,nonce:xmZ5adIxEJCBFp/GBywx6Q==,sha256sum:lcQ3AGW++w7zuiOtC2t2JpCs6XIcsy6+28PUIf/zxo8=}";

\*/

// Construct a new payload to send to the server. This is done to verify that the secret we discovered is correct.

String newValue = "{mat1:2565427,mat2:2572741,nonce:";

// Generate random nonce

uint8\_t nonce[16];

for (uint8\_t i = 0; i < 16; i++)

{

nonce[i] = (uint8\_t)random(0, 255);

}

// Encode nonce as a base64 string

String encoded\_nonce = base64::encode(nonce, 16);

// Append nonce to payload

newValue.concat(encoded\_nonce);

newValue.concat(",sha256sum:");

// Construct payload over which sha256 will be calculated. First: matriculation numbers

String sha\_payload = "25654272572741";

// Second: base64-encoded nonce

sha\_payload.concat(encoded\_nonce);

// Third: secret. We appended any four printable characters then replaced each one with a byte of the secret

sha\_payload.concat("aaaa");

sha\_payload.setCharAt(sha\_payload.length() - 4, 0x39);

sha\_payload.setCharAt(sha\_payload.length() - 3, 0x64);

sha\_payload.setCharAt(sha\_payload.length() - 2, 0x4F);

sha\_payload.setCharAt(sha\_payload.length() - 1, 0x07);

// Calculate the hash

mbedtls\_md\_context\_t ctx;

mbedtls\_md\_type\_t md\_type = MBEDTLS\_MD\_SHA256;

mbedtls\_md\_init(&ctx);

mbedtls\_md\_setup(&ctx, mbedtls\_md\_info\_from\_type(md\_type), 0);

mbedtls\_md\_starts(&ctx);

mbedtls\_md\_update(&ctx, (const unsigned char \*)sha\_payload.c\_str(), sha\_payload.length());

byte shaResult[32];

mbedtls\_md\_finish(&ctx, shaResult);

mbedtls\_md\_free(&ctx);

// Encode the hash as a base64 string

String encoded\_hash = base64::encode(shaResult, 32);

// Finally, append the encoded hash to the payload

newValue.concat(encoded\_hash);

newValue.concat("}");

Serial.println("Setting new characteristic value to \"" + newValue + "\"");

// Set the characteristic's value to be the array of bytes that is actually a string.

pRemoteCharacteristic->writeValue(newValue.c\_str(), newValue.length());

}

else if (doScan)

{

BLEDevice::getScan()->start(0); // this is just example to start scan after disconnect, most likely there is better way to do it in arduino

}

delay(1000); // Delay a second between loops.

} // End of loop

## Python Script to Brute-force Hashes

**import** base64  
**import** sys  
**import** hashlib  
**import** binascii  
  
mat1 = **"2565427"**mat2 = **"2572741"***# Stored nonce*nonce = **"xmZ5adIxEJCBFp/GBywx6Q=="***# Stored hash. We decode it to be able to compare directly to our generated hash*correct\_hash = base64.b64decode(**"lcQ3AGW++w7zuiOtC2t2JpCs6XIcsy6+28PUIf/zxo8="**)  
  
*# In order to optimize the brute-forcing performance, we parallelize the script  
# execution into a number of processes, where each process goes through a partition  
# of the whole 32-bit range. The number of partitions (and thus processes) is  
# passed to the script as a command-line argument, alongside this instance's index  
# in the partition. We determine the number of partitions argument according to  
# the number of available processor cores on the machine, so that each core runs  
# one process, achieving maximum CPU utilization*partition = int(sys.argv[1])  
index = int(sys.argv[2])

*# the number of sk values each instance is going to go through*  
step = 2 \*\* 32 // partition

*# start sk value* start = index \* step

*# end sk value*end = (index + 1) \* step print(**f"Starting hash search from {**start**} to {**end**}"**)  
**for** secret **in** range(start, end):  
 *# Show progress to get a feeling for the feasibility of this method* print(**f"Progress: {**(secret - start) \* 100 // (end - start)**}%"**, end=**'\r'**)  
 *# Construct the hash input* data = bytes(mat1 + mat2 + nonce, **'utf-8'**) + secret.to\_bytes(4, byteorder=**'big'**)  
 *# Calculate the hash* h = hashlib.sha256(data).digest()  
 **if** h == correct\_hash: *# We found the secret* print(**"\nSuccess!"**)  
 print(**"SHA256: "** + str(binascii.hexlify(h)))  
 print(**"Secret: "** + str(secret))  
 **break**

## Windows Batch Job to Launch 8 Instances of the Python Script

@echo off

for /l %%x in (0, 1, 7) do (

start cmd /k python hashtest.py 8 %%x

)

1. We sent the following message from our client to the server in order to retrieve the encrypted flag:

{mat1:2565427,mat2:2572741,nonce:xmZ5adIxEJCBFp/GBywx6Q==,sha256sum:lcQ3AGW++w7zuiOtC2t2JpCs6XIcsy6+28PUIf/zxo8=}

This message is a replay of another message that we were able to capture from the legitimate client using our masquerading server. We used the python script in part (c) to brute-force the 4-byte secret number, taking advantage of an 8-core i7 processor to parallelize the brute-force attack into 8 concurrent processes, each one calculating all the hashes in a separate 1/8th of the full 32-bit range. After **about 9 hours**, we got a matching hash. The secret used to generate that hash was

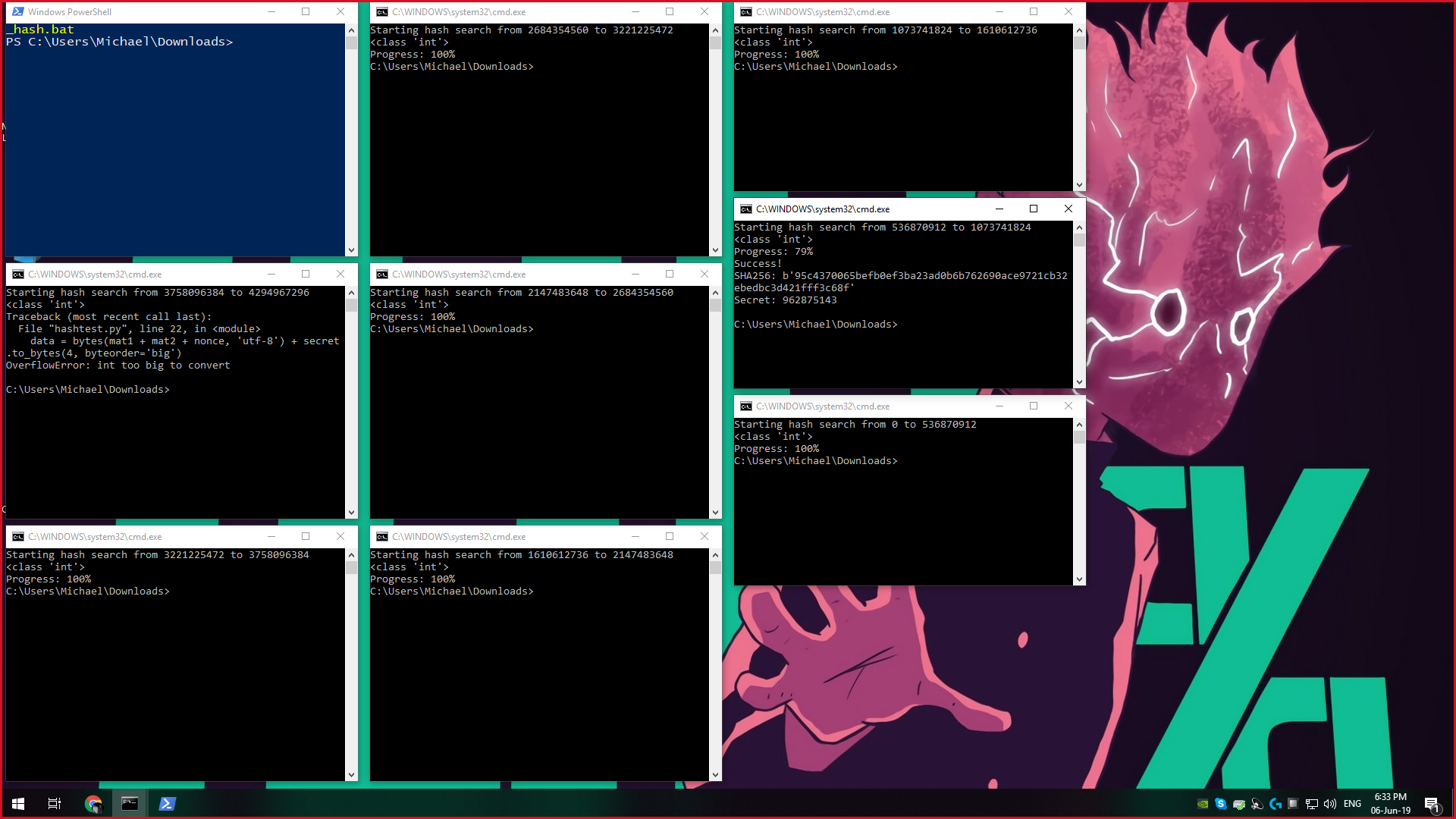


Figure 1 - Brute-forcing the secret by calculating hashes

1. The encrypted flag we received was **[0x23, 0xB8, 0x28, 0x20]**. XORing with the recovered secret we obtain the plain-text flag **[0x1A, 0xDC, 0x67, 0x27]**. We verified that our results (secret and flag) were correct by using our client to repeatedly authenticate with the legitimate server by generating a random nonce, calculating the hash using the recovered secret, sending a message to the server containing our matriculation numbers, the generated nonce, and the calculated hash, and checking that the server’s response is (i) the same every time, and (ii) decrypts to the same flag when it’s XORed with the recovered secret.
2. The most obvious fix would be for the client to first authenticate the server so as to prevent an attacker from masquerading as one. This can be done by first sending a challenge from the client to the server to encrypt and send back to the client, which compares it to its own version of encrypted challenge and if there’s a match, then the server is legitimate.
3. The attacker is able to guess whether the server’s response is random in the current version because the random response changes from one query to the next while a response containing the encrypted flag does not. This can be fixed by having the client and server use the pre-shared secret as a seed to generate a 4-byte pseudorandom number for each transaction. The server then, upon receiving a valid request, replies with the flag XORed with the current transaction’s pseudorandom number. The client is able to decrypt the encrypted flag because it has generated the same pseudorandom number (as the output of a PRNG is deterministic given the seed) while an attacker cannot distinguish between a random reply and an encrypted flag.

# Protocol Design Questions



|  |  |  |  |
| --- | --- | --- | --- |
|  | Pseudo-Random Number Generator | Cryptographically-Secure Pseudo-Random Number Generator | True Random Number Generator |
|  | A mathematical function that takes a seed as input, and produces a sequence of numbers that *looks* random from a statistical point of view, but is actually deterministic given the seed. If this PRNG is not cryptographically-secure, then an intelligent attacker (that doesn’t necessarily know the seed) can distinguish its output from true randomness in polynomial time | A PRNG whose output cannot be distinguished from true randomness by an intelligent attacker (that doesn’t know the seed) in polynomial time | A number generator that uses a source of randomness from the environment to generate its output, and thus its output is not deterministic |
| Use Case | Can be used to provide a source of random input for computer simulations (e.g. physics or chemistry simulations) where there’s no need to protect against attackers | Used in cryptographic protocols to provide random numbers (e.g. in key generation) | Can be used as a source of seeds for (CS)PRNGs, or can be used like a CSPRNG in cryptography |

1. The attacker can send to the client, receive the encrypted feedback , and check if , where is the updated key; If the check is true then the key update was successful. This can be fixed by sharing a random number between the server and the client that is incremented with every authentication, and concatenating that number to before encrypting the combination and sending it back. This way, if the attacker doesn’t know the shared number then they won’t be able to determine if the key update succeeded, because sending the same again doesn’t result in the same even if the key is unchanged.