**Embedded Security (Summer Term 2019)**
Dr. Stefan Nürnberger
CISPA Helmholtz Center for Information Security / Saarland University

# Exercise #3 (24 Points)

**Due due: June 12, 23:55**

## Submission Instructions

- The solutions must be submitted via CMS as a PDF.
- Put names and matriculation numbers of all students who authored the solutions **on the PDF itself**.
- Explain how you solved each task (otherwise 0 points).
- If not stated otherwise, code must be written in *C*, *C++* or *Python*.
- Additional files (e.g. source code) should be added to a .zip (with the PDF solution)
- Label your solutions corresponding to the exercises on the task sheet.
- Handwritten solutions are accepted, but they have to be readable; otherwise points might be deducted.
- If you have questions that might leak deduced information, write a mail to the tutors or visit the office hours instead of creating a post in the forum.

### 1 General info (0 points)

This time you will work with ESP32 on Bluetooth Low Energy. Also for this exercise, the hardware is located in the student room 2.17 at CISPA. This exercise is structured so that **multiple teams can work at the same time**. Moreover, we provide two pairs of server-client which implement the exact same code (hence you have double throughput).

## 2 Bluetooth security for noobs (20 points)

*MetaCortex* is in crisis. Until recently they had a monopoly on Bluetooth Low Energy (BLE) servers for flags assignment. However, a new company, *CryptoGuys*, had been funded with the claim they provide secure BLE flags exchanges.

*MetaCortex* thought this to be impossible as they were aware that the BLE library used by *CryptoGuys* didn't provide any sort of security and client/server authentication. They thought they could easily steal the secret flags stored in the server – and hence prove that *CryptoGuys* system was flawed.

Sadly, also the *CryptoGuys* were aware of the problem. To compensate, they implemented a custom protocol in which a client authenticates to the server (via BLE Characteristics) by sending a payload where it encodes $mat1$: a 6-char matriculation, $mat2$: a 6-char matriculation, *nonce*: a base64-encoded random nonce[1], and $sha256sum$ a base64 sha256 computed over

$$encode_{64}(sha256(mat1 \mid\mid mat2 \mid\mid nonce \mid\mid secret)) \tag{1}$$

The server receives the payload, computes the same hash using the known pre-shared *secret*, and compares it with the received one. If the comparison succeeds, the server returns (via BLE Characteristics) a 4 bytes encrypted flag by computing:

$$e_{flag} = (flag\ XOR\ secret) \tag{2}$$

If the comparison fails, the server returns a 4 bytes random number. A high-level overview of the process can be seen in Figure 1.

*MetaCortex* thought they where done for – *CryptoGuys* had won. However, after investing some brain-power, they understood that not all was lost! In fact, they found a big vulnerability in the protocol: if exploited correctly, they are confident in being able to retrieve the secret flag!

Some hints:

- $Mat1$ and $Mat2$ are passed to the hash function as strings. *nonce* as a base64 encoded string, and *secret* as a byte array.
- The text, Equations 1 and 2 provide all necessary information to infer the vulnerabilities.
- The client authenticates each team to the server in a loop. The secret and flag are different for each team: you should focus on the one where $mat1$ and $mat2$ corresponds to you matriculation number.
- There are two client-server pairs to work with. This is to allow multiple teams to work at the same time.

[1]The *nonce* generator is a secure RNG generator.

– For this exercise you are allowed to turn off the server or the client (e.g., you can remove the power) if you need it. Moreover, if something doesn't work, feel free to reset the ESPs via RST button.

For this exercise we won't provide any code. However, you can find everything you need in the arduino-esp32 repository `https://github.com/espressif/arduino-esp32`. For the sha256 another good reference library is `https://rweather.github.io/arduinolibs/crypto.html`. In platformIO, a project for the ESP32 requires a configuration of the sort *platform = espressif32, framework = arduino, board = X*, where X depends on the type of board you got.

(6 points) (a) The protocol is vulnerable due to two independent vulnerabilities. What are those vulnerabilities? How did you discover them? How can they be exploited? Also, briefly describe each step of the attack.

(2 points) (b) Is it possible to retrieve the correct flag by repeatedly authenticating the client while brute-forcing the key (i.e., for each key, compute the hash and wait for the server reply)? If yes, how long would it take (250ms per exchange)?

(6 points) (c) Provide the source code of your attack. You are not required to implement everything on the ESP: you can provide different files (python, c++, etc.) for each different step. However the code should work.

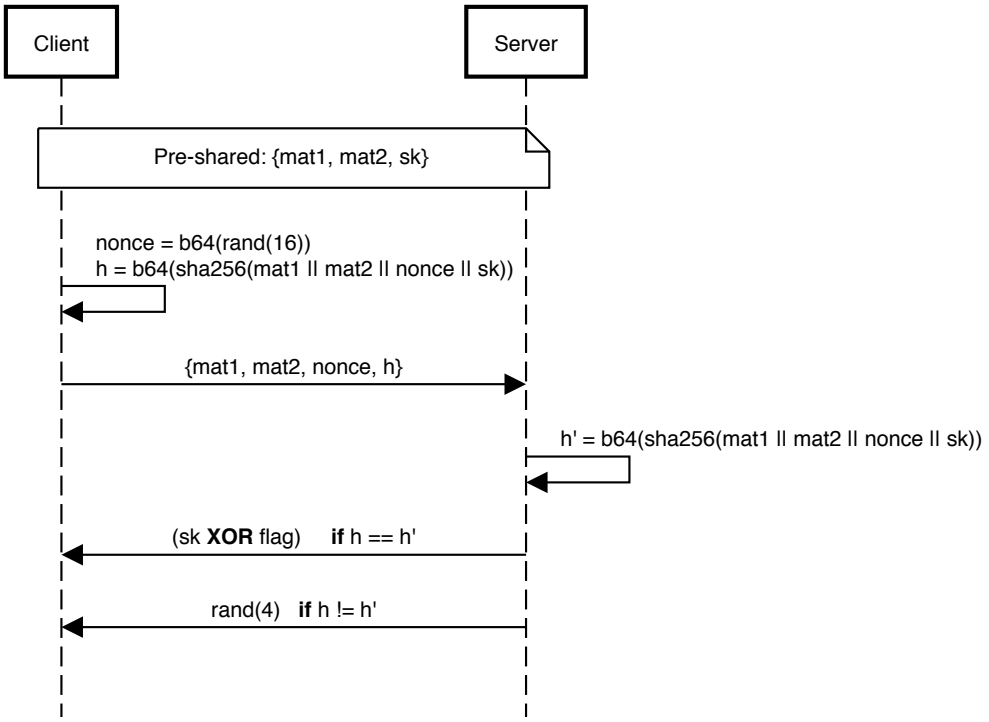
(2 points) (d) Attach to the PDF the client authentication message you used to attack the protocol. What's the secret you derived? How long did it take?

(2 points) (e) If you do everything correctly, and you send your message to the server, you should be able to receive and decrypt the "encrypted" flag. Attach it to this PDF. How can you be sure that the decrypted flag is the correct one?

(2 points) (f) Assuming you want to fix the vulnerability without changing the protocol: What would be the most obvious fix?

(+3 bonus) (g) Let us assume the attacker is able to query the server with each possible authentication message in finite time (independently to key length, hash, etc.), and that the server always reply (i.e., no delay). Can the protocol be fixed so that the attacker cannot decide wether he guessed the correct key – and hence got the correct flag?

Finally, there are multiple ways to attack this protocol: the questions are meant for a specific one, however if you think of a better or faster way, please verbalise it and let us know. If it's a reasonable attack, it will be considered as solution.




## 3 Protocol design questions (4 points)

(2 points) (a) What are the differences between a pseudo-random number generator, a cryptographically secure pseudo-random number generator and a true random number generator? Can you think of a use-case example for each of them?

(2 points) (b) Recall the challenge-response protocol used by MEGAMOS: the client sends a random challenge $r = RNG()$ and the server replies with a response $c = enc_k(r)$. The client then verifies $r == dec_k(c)$. An attacker can eavesdrop $c$ and $r$, do a partial key update and re-send $c$ as often as he wants. How can the attacker use $r$ to check if his partial key update was successful? How could this be fixed when re-designing this protocol?