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

Solution of Exercise Sheet #3

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.

Solution

The aim of the exercise is to retrieve the decrypted flag correctly. This means you must know the key in order to decrypt the encrypted flag. Thankfully, there are two big "design flaws" in the whole client-authentication and flag retrieval process.

– The BLE link/network layer doesn't authenticate the communicating parties. This means it is quite easy to spoof the server identity (similar to GMS). Thanks do that, an attacker can spoof the real server in order to obtain a sample authentication message from a legitimate client.
– The k used to authenticated the client and encrypt the flag is only 4 bytes long. This means that if an attacker can get hold of a legitimate authentication message, then she can simply offline brute-force the key k in a couple of minutes.

There is no strict correct/wrong way to present the vulnerability – my wording was a bit unclear. In the end, if the code and the flag are returned correctly, and the answer seem reasonable, then we give full points.

(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)?

Solution
Yes, it is possible, since the server returns a sequence of random bits in case it gets a wrong authentication message. This means that an attacker is able to understand that she guessed the correct key by simply:

- Try every possible key by sending the authentication message twice (the second to verify the randomness). This would take, in average, $2 \cdot \frac{2^{32}}{2}$ tests. Assuming a test requires 250ms (no need to consider nsec delays due to hash computation etc.), this results in $2 \cdot \frac{2^{32}}{2} \cdot \frac{1}{4}$ seconds, which corresponds to ∼34 years.
- Obtain a legitimate authentication pair from the client, and use it to verify the reply from the server (in case of collision you can simply try trice). This would simply remove the 2 multiplier, resulting in $\frac{2^{32}}{2} \cdot \frac{1}{4}$ seconds ∼17 years.

Any of the above reply is sufficient. Also, is not an error to consider the number of possible tests as $\frac{2^{32}}{2}$. Any reasonable answer and result suffice.

(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.

Solution
There are three main steps in the attack.

1. Obtain the identify of the server you want to spoof. This requires you to SCAN all BLE decides in the area. You can use this script.[a]
2. Obtain the real authentication message from the client. Again, this implies you need to set-up a BLE server by spoofing the credentials obtained in (1). Then is just about listening the messages from the client and getting your own. Example code.[b]
3. Now that you have an authentication message, you simply have to offline-crack the key used to create it. You can do it however you want, one of the easiest way is via Linux `hashcat` command. Or a python script.
4. Finally, given the detected key, all that's remaining is to connect to the server, craft a legitimate authentication message and send it to server.[c] The server replies with the encrypted flag, which can be decrypted by XORing it with the guessed key.

As long as makes sense, and the secrets/results are correct, then it should be fine.

[a] `https://github.com/espressif/arduino-esp32/blob/master/libraries/BLE/examples/BLE_scan/BLE_scan.ino`

[b] `https://github.com/espressif/arduino-esp32/blob/master/libraries/BLE/examples/BLE_write/BLE_write.ino`

[c] https://github.com/espressif/arduino-esp32/blob/master/libraries/BLE/examples/BLE_client/BLE_client.ino

(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?

Solution
An example authentication message is:

- `Mat1: 1234567`
- `Mat2: 7654321`
- `Nonce: fIR9fiSyFAODHDgAvAJdKA==`
- `Sha256sum: SwDnK79ew1M38r0hHBKs9uwR9UxBWD6ro6Xj6S9KXyA=`

The resulting key would be something of the sort `0x00D1C2E3`. In average (given there where only 28 bits of entropy), this should have taken a couple of minutes on a single-core modern processor. Obviously, longer times won't make you lose points!

(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?

Solution
One possible way is by crafting two different authentication messages: if for both you receive the same flag, then you can be sure the key is correct (or at least, the probability of a collision to happen is negligible in this case). You can also just trust the sha256. Assuming everything is implemented correctly, the probability that sha256 returns a collision is also negligible.

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

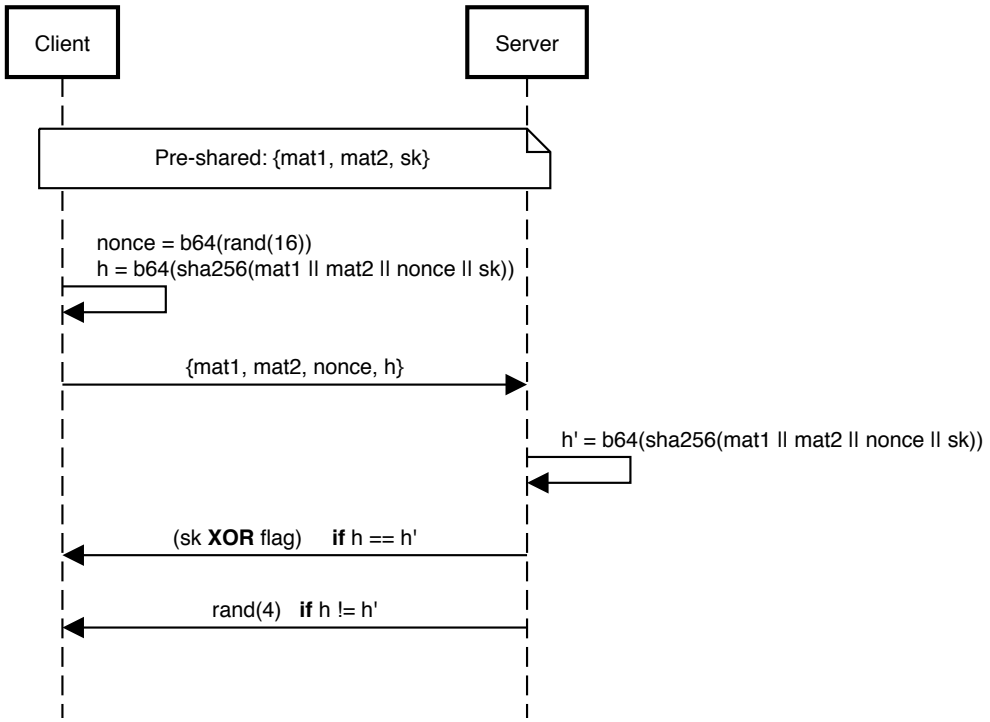
Solution
The easiest fix is to just increase the length of the *key* to 8+ bytes so that it can't be brute-forced. The flag and nonce can be padded to account for the longer *key*.

(+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 whether he guessed the correct key – and hence got the correct flag?

Solution
The idea is to exploit the fact the flags are random. In fact, it is impossible for an attacker to know whether she got the correct flag or not, independently of the key/computational power she posses. The easiest way to do this, is to have the server return a deterministic encrypted flag. This means that, given the same guessed key, the server *always* returns the same encrypted flag. There are multiple ways to achieve that, the easier being the server always returning the same encrypted 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?

Solution

PRNG	CSPRNG	TRNG
A pseudo random number generator takes a "seed" as input and expands it to a stream whose properties resemble a random sequence of bits.	A special type of PRNG that passes the next-bit test. Meaning given all the previous bits, the next bit in a stream cannot be determined in reasonable time.	A True random number generator produces randomness based on unpredictable physical phenomena. It generally requires special hardware and sources of entropy.
In video games (for procedural generation) and other applications that don't require security.	Generating keystreams for stream ciphers or secure and reproducible random sequences.	To generate symmetric keys, secure seeds for CSPRNGs, or random nonces.

(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?

Solution

The attacker can get hold of one of the challenge/response pairs, he now knows that the response c corresponds to challenge r when the key is k of length l.

By repeatedly updating one byte of the key to all possible values, and sending the same challenge r, if the attacker obtains the same response c, then he can infer the original value of th byte under test. Repeating this for each byte can brute-force the key k in a relatively small amount of time

($\frac{l}{8} * 256$ vs 2^l)
The easiest way is to prevent partial key updates by allowing only atomic key updates. Another way is by adding randomness to the response c. Other solutions are accepted.