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

Exercise #4 (27 Points)

Due due: July 19, 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 Secure Over The Air update (17 points)

Once again, *MetaCortex* needs your assistance. Unfortunately they did not pay attention to the IoT trend and now they are far behind with designing software for embedded systems (not speaking of security). They decided to make a compromise and ask you to design a secure over-the-air software update mechanism, so they can quickly sell the devices now and fix the bugs later.

For this exercise you will work with the ESP-32. The new image you should update to is located here[1], its signature here[2]. For creating the signature we hashed the file via *sha256* and encrypted the hash with padded RSA. Specifically, the used command is: *"openssl dgst -sha256 -sign privatekey.pem -out signed.bin -in firmware.bin"*. The content of **b64sign.sign** represent the result in base64 computed as *"base64 signed.bin > b64sign.sign"*. You may want to download the public PEM key[3] which you can use to verify the signature.

You can assume that until the deadline these files do not get compromised. After submitting you can expect us to run your software in an environment where an attacker has full control over the network and the server (for sure he will not have access to the private key which can be used to sign other image files). Note that we might change the image file **firmware.bin** and the signature **b64sign.sign**, so do not hardcode the files signature into your program.

[1] `https://emsec2019.pi/firmware.bin`.
[2] `https://emsec2019.pi/b64sign.sign`
[3] `https://emsec2019.pi/pubkey.pem`

You need to setup your IDE first. We recommend the ESP-IDF toolchain[4]. We cannot assure you that the task can be successfully completed with the Arduino IDE.

(2 points) (a) In order to keep the update process simple, *MetaCortex*'s first idea is to not verify the image, but just establish a secure connection over HTTPS. Briefly explain why it is important to check the integrity of a download with a signature.

(3 points) (b) The *esp_ota_ops* library includes tools which simplify the update process. Given the header file[5] and an example[6], briefly explain how the update process works. In order to do so, give an example where you outline *which* parts of the flash is used for *what* during the update process. Furthermore state one drawback of this mechanism.

(10 points) (c) Implement your approach. Your program should...

(1) ...connect to the WiFi **EmSec2019** with the password **EmSec2019Password** and obtain an IP via DHCP. **The network is provided by a raspberry PI in the student room 2.22.**

(3) ...connect to the Server given above through HTTPS. The root CA certificate used to sign the HTTPS certificate is located here.[7]

(5) ...download the flash file and its certificate, verify the image and boot from it if the signature is valid. It should fail downloading the image from here[8].

(1) ...blink the onboard LED if there was an error during the update process.

Hint: If you do not manage to perform a SSL handshake with *www.emsec2019.pi*, you maybe want to check if you can do one with another website.

(2 points) (d) *MetaCortex* now created the software for a media control system. Unfortunately, the image size is fairly huge (about 3MB) in comparison to the 4MB flash size. They propose to create a small custom second stage bootloader (a few hundred KB) that can automatically connect to a network, download the image from the server in small chunks, feed these chunks to a hash function (like SHA256) and if the resulting hash is successfully checked against the certificate, the server is trusted and the image file seen as not compromised. Then the image file is downloaded again by the bootloader, overwriting the old firmware. What could possibly go wrong?

Important note: You should submit both code *and* the compiled binary. Make sure to submit the image for the whole flash, not only the flash image starting at vector 0x10000. Your submission must be a zip archive that contains these elements:

1. A directory that contains all the files necessary to build the project.

[4] `https://github.com/espressif/esp-idf`

[5] `https://github.com/espressif/esp-idf/blob/master/components/app_update/include/esp_ota_ops.h`

[6] `https://github.com/espressif/esp-idf/tree/master/examples/system/ota`

[7] `https://emsec2019.pi/rootCA.crt`.

[8] `https://emsec2019.pi/evil.bin`

2. Your solution as a PDF.
3. The binary file.

2 No buffer overflow, we checked it! (10 points)

MetaCortex heard that you are leaving town soon, so they decided to bother you with their numerous questions on security related topics. A trainee just sent you a snippet of code[9] that allowed attackers to read memory the application was not supposed to read. He kindly asks you to explain him why and how this is possible.

(4 points) (a) Explain how *speculative execution* and *caching* works in CPUs and for both briefly explain why they are used in many modern architectures. Your solution should mention *pipelining*.

(6 points) (b) This is the code snippet you received from the trainee:

```
unsigned long readFrom = ...; /* attacker-controlled */
unsigned char uncachedData[1000];
unsigned char string[421];
unsigned long stringLength = 421;
/*
Part 1: Some code (I am sure you can figure out what happens here)
...
*/
//Always check index to not exceed the array's boundaries
if(readFrom < stringLength){
  unsigned char c = string[readFrom];
  unsigned long indexUncachedData = (c&1)*900;
  unsigned char unusedValue = uncachedData[indexUncachedData];
}
/*
Part 2: Super complex code that still has to be reverse-engineered
...
*/

```

In terms of *meltdown* and *spectre*, explain how memory can be read out from locations that the application is not supposed to read, if the program runs on a vulnerable machine. **As a guide**, your solution should explain...

- why it is important that `uncachedData` is actually not cached.
- why it is important that `stringLength` is not cached.
- why it is important that `readFrom` is cached.
- how many bits of memory leak when this code is executed only once (assuming there is no similar code in part 1 and 2).
- what happens in the code part 2 to actually leak data from other memory locations.

Hint: It might be helpful to recap the sidechannel-slides as well as to read this[10] post.

[9] This is a simplified version, you can assume that optimisations were turned off when the code was compiled

[10] `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`

3 Dolly - The unclonable sheep (0 points)

SRAM can be used as a physically unclonable function (PUF). The bits that are retrieved from non-initialised memory after the chip has been powered on are random when comparing it to other chips, but almost identical for every power cycle on the same chip. This similar pattern is unique for every chip due to manufacturing tolerances.

(+2 bonus) (a) While SRAM is present in modern computers in forms of cache, they are usually not usable because the state of the cache may be deterministic depending on the CPU.[11] However, there are other possibilities than SRAM for PUFs on these machines. Name one thing that can be used as PUF (no SRAM) that is part of basically every modern computer (GPUs are not necessarily contained in every computer) and briefly explain how it works.

(b) Now's the time! Metacortex last task for you is to create a way to let software run on specific hardware only. They want you to create a piece of software that only carries out its functionality if it is running on a specific computer. In this exercise, you create a program that only runs on *your* arduino.

(+4 bonus) i. Create a sketch that reads 256 bytes of uninitialised SRAM and writes that data to serial out after the boot is finished. Then execute this sketch several (at least 5) times. Make sure to remove the power plug between each execution for a few seconds. Briefly explain what you observe and determine a fingerprint for your arduino (This fingerrint is the 256 bytes of random memory but as it may not be the same all the time, you need to determine **one** fingerprint for later use).

(+4 bonus) ii. Extend your sketch from a) with an error detection function that accepts a 8% error rate (at most 164 biterrors) compared to a constant bitstring. This function should use the uninitialised memory read before boot and compare it against the previously determined fingerprint. It should be executed in the setup and output on serial "Check OK" if the uninitialised memory differs by at most 8% from the provided fingerprint and "Check FAILED" if it differs by more. Make sure that the output is "Check OK" for your arduino.

Make sure to provide the source code for subtask (ii). Furthermore, your code must include the serial number[12] both in the PDF **and** the source code. Also your Arduino should be tagged with a note when you hand the Arduino back. The bonus task can only be graded after we received your Arduino.

[11] `https://eprint.iacr.org/2015/760.pdf`

[12] You can obtain the serial number of your board by clicking on *Tools→Get Board Info* in the Arduino IDE