Technische Universität München
Embedded Systems and Security

Release date: 07.12.2017
Due date: 16:45 21.12.2017

Deliverable 3: May I have some salt, please?

1. Introduction

As an additional opportunity to consolidate your skills, we offer four programming exercises. They make up 20% of your overall grade as outlined in the first exercise session. You are encouraged to discuss approaches and share experience with other students. But as it affects your grade, your submission must of course be your own original work. NB: This does not prohibit you from using free and open-source software *as part* of your submission.

The submissions are graded fully automatic once per hour after the grading machine is set up (usually a day or two after the release date above). Together with your achieved points you will be provided several logfiles which explain why your solution got this amount of points. You can upload a new solution as often as you like during the submission period. Should you find evidence that the grading does not follow the problem statement set out below, please contact us as soon as possible, so we can look into the problem and fix it before too many other students are affected.

2. Problem Description

In this deliverable, you will be using a real-world crypto library called "libsodium". It is a more portable version of the "NaCl" crypto library by Daniel J. Bernstein, Tanja Lange and Peter Schwabe. Though its portability, libsodium will not work out of the box with bare metal applications, because it requires either `/dev/urandom`, `getrandom()`, or `CryptGenRandom()` and neither is available without an operating system. As all entropy in this deliverable will be supplied from the outside, replace the random number generator (RNG) with a dummy implementation provided together with this document. Consult the libsodium documentation on how to do this.

For your convenience, a precompiled version of libsodium 1.0.15 ready to be linked into your project is provided together with this document. Place the library somewhere outside of your project folder. It should not be contained in your submission, otherwise your submission might be too large to upload. *Optional:* In case you want to compile it on your own, appendix A provides details on how to do that. However, in this case I cannot guarantee that your project compiles correctly on my grading framework, although it should.

2.1. Part A

Create a project named `PartA` and add code to send and receive data over the USB virtual COM port, you can also use `example_usb_project` as a starting point. In contrast to Deliverable 2, your submission should not just echo data back, but encrypt packets of data. What makes up

a packet is given in section 2.1.1. The base64url encoding used for some parts of the packet has to be stripped before further processing and applied again before returning the result.

For encryption of the data, you have to use `crypto_secretbox_easy()`. The nonce is supplied in the header of the plaintext packet. The key is made up of concatenating the 128 bit "unique chip ID" to itself, resulting in a 256 bit key – though with only 128 bit of entropy[1], but this is deemed sufficient for the data used here.

Your solution must have replied a valid packet containing the encrypted data within two seconds. To get full points, your solution has to be capable of processing packets containing up to 32 KiB (binary) respectively 43 692 B (base64url encoded) of text.

Test packets for verifying the behaviour of your program are supplied together with this document.

2.1.1. Transmission Protocol

Data is transmitted in packets starting with a "Start of Heading" control character, followed by a packet header, a "Start of Text" control character, the data to be transmitted and finally an "End of Text" control character.

`0x01`	`<base64url encoded>`	`0x02`	`<base64url encoded>`	`0x03`
Start of Heading	Header	Start of Text	Text	End of Text

The data in those parts marked as `<base64url encoded>` is encoded in base64url as specified in RFC 4648. The control characters of hexadecimal value 0x01, 0x02 and 0x03 together with the base64url alphabet are the only permissible ($\neq$ possible) values on the virtual COM port.

The header – before encoding to base64url – is comprised of:

Length of Text after base64url encoding (in Byte)	[Nonce]
24 bit	192 bit

Where the "Nonce" is only present if the packet contains plaintext data in its "Text" field, i.e. not if you return the encrypted data.

In case the receiver detects data not following the format given above, e.g. size of Text not matching what was specified in Header or use of impermissible characters, it has to discard the whole packet without further processing.

2.1.2. Hints

- The "unique chip ID" is relocated by the startup code and made available as a global variable. Search the `XMC4500.h` for it.
- When adding the sodium library, do not forget to
  - Add its `include` subfolder to the search path of the compiler (`-I`)
  - Add its `lib` subfolder to the search path of the linker (`-L`)
  - Add `sodium` to the list of libraries to link (`-l`)

[1] In fact it is far less, considering the way unique chip IDs are distributed among our boards. But since your submissions are graded only once per hour, the key space is deemed sufficiently large to avoid brute-forcing the key.

  - Beware that `#include "foo"` is different from `#include <foo>`, cf. `https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html`.

- Take care you do not throw the beginning of the next packet away if the previous one was not correctly finished
- Do not forget to replace the RNG and call `sodium_init()`
- When visually inspecting correctness of your results, beware of the quite awkward interpretation of endianness in RFC 4648 – cf. section 9
- Note that the reference implementation given in Section 11 of RFC 4648 does not use the url safe alphabet and rejects padding chars "=" as non-base64url although they are permissible parts of a packet

2.2. Part B

Use your solution from Part A to activate the marble run located in the corridor of our chair. On success, it will display a bonus token to be submitted for this part of the deliverable. As the token is publicly visible, it is of course cryptographically personalized. Note that this means you have to activate the marble run using the board handed out to you. If you submit a token created by someone else or someone else submitted your token, both of you will have to report to me in order to determine to whom the token belongs.

The marble run understands the following commands:

```
/* Commands are int, with the lowermost byte determining the command, upper
 * bytes might contain arguments as specified per command
 * */
enum MARBLE_CMD_e {MARBLE_CMD_WAIT, /* pauses execution of commands for p*10 »
  «millisecond, with p as uint16_t in bits 23:8 */
                   MARBLE_CMD_LOAD, /* loads one marble into separator, no »
                     «arguments */
                   MARBLE_CMD_RELEASE, /* releases the marble into the lift, »
                     «no arguments */
                   MARBLE_CMD_LIFT_UP, /* moves lift upwards, no arguments */
                   MARBLE_CMD_LIFT_DOWN, /* moves lift downwards, no arguments»
                     « */
                   MARBLE_CMD_SHORTCUT /* toggles shortcut in marble run, no »
                     «arguments */
                  };
```

In case you send an illegal command, it is ignored. You may compose arbitrary command sequences, but be warned that the marble run will also ignore any command that would cause the loss of a marble or a hardware defect if executed at this point in time. As an example, this sequence rolls a single marble:

```
int marble_commandRun(void) {
    if( (marbleAddCmd(MARBLE_CMD_LIFT_DOWN, 1) == 0) || \
        (marbleAddCmd(MARBLE_CMD_LOAD, 1)      == 0) || \
        (marbleAddCmd(MARBLE_CMD_WAIT | (800<<8), 1) == 0) || \
        (marbleAddCmd(MARBLE_CMD_RELEASE, 1)   == 0) || \
        (marbleAddCmd(MARBLE_CMD_LIFT_UP, 1)   == 0) ) {
        return CMDfailedError;
    }
    return DEMOsuccess;
}
```

Please also note that if you send very long command chains, your token may not fit into the response field and will be rejected upon grading. Use above sequence of commands if you do not want to find out the limit.

Because the request is sent via HTTP, the simple packet format above is not necessary in this case. Therefore the marble run expects only the encrypted and base64url encoded text without the control characters and header from Part A.

The board ID requested by the web form is the barcode on the backside of your XMC Relax Lite Kit. In case you have a board with ID above `0097` or use a board not borrowed from us, provide us with the unique chip ID of your board and we will add it to the key store of the marble run. You will receive a virtual board ID that you can enter in the web form.

The control interface for the marble run can be reached at `http://10.152.249.7/` from within the LRZ network.

2.2.1. Hints

- There is no denial-of-service (DoS) protection on the marble run control interface and its a microcontroller, not a server. We appeal to your honour and fair play.
- For Part B you only have to submit the bonus token, so in case you do not get the protocol parser correct, you can still set and read values using the debugger

3. Submission

3.1. What to Submit

- A ZIP archive containing your project including all subfolders and your token
  - I.e. your ZIP must contain two folders in its root, named `PartA` and `PartB`
  - Your token must be a file called `token` located in folder `PartB`
- Make sure the `*.elf` files in your project are up to date, since they are used without modification.
- If your submission does not contain `*.elf` files, your project will be rebuilt, which adds a point of failure.

3.2. How to Submit

1. From any of your project directories (e.g. directory `PartA`) run `make deliverable`
2. Make will create a `.zip`-file in `../`.
3. Upload this archive via moodle

A. Compiling libsodium on your own

The libsodium documentation provides an example on how to cross-compile it in its section about installation. We will mostly follow this example, but need to make some further adjustments.

1. Download the 1.0.15 tarball and extract it

2. Due to some strange compiler incompatibility, function pointers are sometimes incorrectly calculated. Therefore the *`pick_best_implementation` functions do not work, but instead yield pointers to no man's land. As the Cortex-M4 does not comprise any x86 instruction set extension anyway, there are no better implementations than the default ones. So we can get rid of these broken functions without loosing anything by commenting out or deleting lines 55 - 60 in `src/libsodium/sodium/core.c`

3. Now we can start with the first line of the cross-compiling example: In case the arm-none-eabi-gcc is not already in your `PATH`, use this line to add it temporarily:

```
1 export PATH=/path/to/gcc-arm-none-eabi/bin:$PATH
```

4. The second line can be used as is

```
2 export LDFLAGS='--specs=nosys.specs'
```

5. The third line needs to be extended, because the arm-none-eabi-gcc otherwise defaults to produce ARM code, which is not compatible with the Cortex-M microcontrollers. To keep the compiled code small, we use `-ffunction-sections`, so that every function gets its own section and can be omitted by the linker if it is never called. Finally, you probably also want to add some debugging symbols to make life easier:

```
3 export CFLAGS='-Os -mcpu=cortex-m4 -mfloat-abi=softfp -mfpu=fpv4-sp-d16 -»
    «mthumb -ffunction-sections -g3 -gdwarf-2'
```

6. To produce the various build scripts, run `./autogen.sh`. This essential step is unfortunately not mentioned in the online documentation.

```
4 ./autogen.sh
```

7. Now we can run the configuration, which location you choose for installing is not relevant. You can move the output folder freely, as long as you update the references in all projects linking them:

```
5 ./configure --host=arm-none-eabi --prefix=/install/path
```

8. The last command will run the actual compilation and archive the various `*.o` files into a single `*.a` library and put all the headers together in a single folder for easy referencing:

```
6 make install
```

   In case you want to install the library in a folder where only root can write to (e.g. `/opt/sodium`), you have to precede the above line with a `sudo`