Technische Universität München
Embedded Systems and Security

Release date: 1.12.2022
Due date: 16:45 22.12.2022

Assignment 2: Smashing the Stack for Fun and Profit

1. Introduction

As an additional opportunity to consolidate your skills, we offer three 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, and will be checked for plagiarism.

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, but we will ask questions in the case of a conspicuous high number of submissions. So don't just adapt your code to the grading framework. In the end, your best submission will be used for grading. 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

Together with this problem description, you should have received two ELF files and parts of the corresponding C sources. All programs set up the XMC board to connect as a virtual COM port to the computer via connector X3 – the micro-USB plug on the opposite site of the debugger. If you send data to the board, it will be "encrypted" on the XMC board and the resulting ciphertext will be printed on your console in base64url encoded format. For your convenience, a script `sendExploit` is provided which takes a binary file as input, packs it and sends the packet to the board. The packing process encodes the file content in base64url and adds a base64url encoded data length and nonce as well as some control characters. An explanation of base64 encoding can be found in RFC 4648.

For Part A and Part B, do not change or recompile the given code, since this may change the exact location of buffers and functions and your exploit will not work when tested against the original ELF files.

For Part C, 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. Through 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. For Part A and Part B, the random number generator (RNG) is implemented with the C function `rand()`. This is a terrible choice because `rand()` is

not a cryptographically secure RNG. We recommend, although do not require, to replace it by a more secure implementation during Part C, e.g. the one that comes with libsodium[1]. 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

The program uses a homebrewn crypto algorithm which first decodes the base64url encoded input data, "encrypts" the content of the binary file, and finally encodes the ciphertext before it is sent back. However, the program is vulnerable to a stack based buffer overflow.

Your task is to find this vulnerability and design an exploit that turns both LEDs on by executing code that is part of the exploit and not previously on the system (code injection). The LEDs must be turned on directly upon completion of the vulnerable function[2]. Your exploit must be self contained, which means it must contain e.g. all necessary padding. It must be sufficient to send the file containing the exploit via the provided script to activate the LEDs.

To get full points, your exploit must be submitted as a binary file named `exploit` and must be designed such that the application continues working as if nothing happened. No reset, hang up, etc. may happen. Someone observing the USB port must not notice something went wrong except possibly a slight delay caused by executing the exploit.

2.2. Part B

Instead of fixing the stack based buffer overflow vulnerability, the designer considered it sufficient to enable the MPU to harden the system against attacks, and additionally activate the stack protection feature in his compiler to make his system "The most secure one ever!!!".

Your task is to convince the designer that this is insufficient by providing an exploit that activates the blinking function of the program. Again, to get full points, the exploit must be submitted as a binary file named `exploit` and the application has to continue working as if nothing happened.

2.3. Part C

After we showed that even the best security features cannot fix the vulnerabilities of an ugly written code, we do what we should have done right in the beginning: throw this homebrewn crypto away and replace it with something reasonable and tested, namely the real-world crypto library "libsodium". Use `crypto_secretbox_easy()` to replace the original `encrypt` function. The required 192 bit nonce is available from the packet parser. 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[3], but this is deemed sufficient for the data used here.

[1] The seed required for the libsodium RNG can for example be produced by hasing the start-up pattern of the SRAM via `crypt_generichash()`.

[2] This is the default behaviour, anyway

[3] 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.

Once you completed this task, use your solution to activate the marble run located in the corridor of our chair. The marble run control interface (MCI) can be reached at `http://10.152.249.7/` from within the LRZ network. Commands to the marble run must be properly encrypted, which can be done with the code you just implemented. The nonce provided by the marble run can be used as is with the `--nonce` argument of the provided `sendExploit` script. The commands for the marble run have to be put in a file like your exploit in the previous parts, and can be composed arbitrarily from the following list:

```
/* 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. Commands that would cause the loss of a marble or a hardware defect if executed at this point in time are also ignored. As an example, this sequence rolls a single marble:

```
      (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.

On success, the MCI will display a bonus token which is the only thing that has to be submitted for this part of the assignment. 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.

In case you have 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 MCI.

If you are interested, you may continue to try to exploit the packetizer, but this is not part of the assignment anymore. We recommend as a starting point the fact that the header contains a field for the length of the plaintext, cf. Heartbleed. Reports about successful exploits of the packetizer are accepted via email to florian.wilde@tum.de. Encrypted, of course.

3. Hints

3.1. Part A, Part B

- The provided `sendExploit` script is written in `python3`.
- Make use of the fact that the location of the stack is not randomized like with ASLR
- Use `info frame` in the debugger to obtain necessary addresses
- Think about how you could get your exploit code compiled. If necessary, use e.g. `xxd -r -p <infile> <outfile>` to convert from HEX to binary representation.
- There are some shells (e.g. the GNU tools) which support the `xor` functionality. Alternatively, you can write a simple C or Python function.
- Be aware of endianness
  - `12 34` and `1234` may reflect different data, compare e.g. `*.lst` file with debugger
  - Verify what your converter does
  - Write an array of `uint8_t` on disk using a few lines of C and open it with your converter in case you are unsure
  - You may also use a HEX editor, e.g. ghex, to manually alter individual bytes
- Unfortunately single stepping with the debugger alters the behaviour of the XMC4500 in some points, e.g.
  - A MemManagement fault will be delayed a few clock cycles, allowing to execute the first few instructions of your exploit although instruction fetch is prohibited in this region
  - Jumps to even addresses work, because the debugger silently prohibits the illegal attempt to change to ARM mode without generating a UsageFault

  During grading the XMC4500 is not running in debug mode, so do not rely on this altered behaviour.
- If you connect the debugger, it will reset the application processor. The reset will disconnect an already established serial connection, so always connect the debugger first, then the other USB cable for the serial connection.
- When you plug in the cable for the serial connection, the board must be able to answer the enumeration request of the PC within a few milliseconds. If the application processor is halted by the debugger, it cannot answer and thus the connection fails. So make sure you enter `continue` repeatedly until the debugger says `Continuing.` and you do not get a `(gdb)` prompt. Only then the application processor is running continuously and able to respond to the enumeration request in time. Afterwards you can halt it by pressing Ctrl - C. You can also set a breakpoint in the vulnerable function before plugging in the cable for the serial connection.
- Watch out for convenient gadgets sprinkled into the `text` section

- To keep Part B working after a successful exploit is a littly tricky and requires to find gadgets not explicitly added to the code. Skip it and continue with Part C if you feel stuck.

3.2. Part C

- A documentation about libsodium can be found here.
- The "unique chip ID" is relocated by the startup code and made available as a global variable. Search the `system_XMC4500.h` for it. The board ID is printed on the label on the backside next to the barcode.
- 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`)
  - Beware that `#include "foo"` is different from `#include <foo>`, cf. `https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html`.
- Do not forget to call `sodium_init()`
- 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.

4. Submission

4.1. What to Submit

- A ZIP archive containing exploits for PartA and PartB and a token for PartC
  - I.e. your ZIP must contain three folders in its root, named `PartA`, `PartB`, and `PartC`
  - The folders `PartA` and `PartB` must contain a file called `exploit` that contains your solution in binary format, i.e. no additional conversion necessary
  - The folder `PartC` must contain a file called `token` that contains the token as displayed by the MCI

4.2. 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`