Technische Universität München
Embedded Systems and Security

Release date: 18.11.2017
Due date: 16:45 9.12.2017

Deliverable 2: Smashing the Stack for Fun and Profit

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.

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

Together with this problem description, you should have received multiple 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, e.g. using HTerm, it will be processed on the XMC board and the answer sent back to you, such that you can observe it. For that, your data has to be terminated by the ASCII line feed character known as `\n`, which is represented in HEX encoding as `0A`. Only after receiving this character, the program will process your data and return the response.

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.

2.1 Part A

The program for this part simply echoes back what was sent to it. 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[1]. Your exploit must be self contained, which means it must contain e.g. all necessary padding and the trailing line feed character.

[1] This is the default behaviour, anyway

To get full points, your exploit 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

The program for this part returns your text with inverted capitalization, i.e. all lower case letters will become upper case and vice-versa, non-letter characters are left unchanged. Instead of fixing the stack based buffer overflow vulnerability, the designer considered it sufficient to enable the MPU to harden the system against attacks.

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 application has to continue working as if nothing happened.

2.3 Part C

The program for this part will reverse your data before it is sent back, i.e. the last character before the line feed will become the first, and so on. Since the designer was still to lazy to fix his ugly code, he activated the stack protection feature in his compiler in addition to the MPU to make his system "The most secure one ever!!!".

Your task is to activate the blinking function again in order to convince the designer that he really *really* **really** should fix his ugly code instead of trying to hide it in more and more layers of security mechanisms. Again, to get full points, the application has to continue working as if nothing happened.

3 Hints

- 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.
- If your compiled code contains a line feed character, modify your assembler instructions slightly to get rid of it
- 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 the bytes
- Many Linux distributions contain programs that perform nasty things on data send to and received by tty devices. Use a terminal such as HTerm, screen, etc. to get around this.

- When you single step through your exploit, 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. This will not be the case on the grading machine, because it will not run a debug session.
- Watch out for convenient gadgets sprinkled into the FLASH

4 Submission

Remember that your submissions have to compile without errors or warnings. Any compile error will make your submission fail and there will be penalty points for every warning. Note that we will check the makefiles in your projects, so do not try to cheat by turning off warnings.

4.1 What to Submit

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

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