N. Tippenhauer / G. Pellegrino

# Exercise Sheet 2 (Software Security)

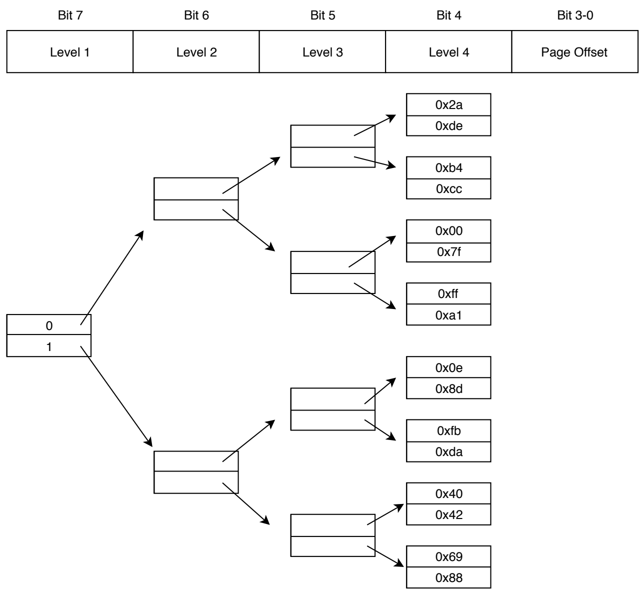
**Notes**

1. This is a **graded** exercise sheet. Please hand-in your solutions in the CMS until 01.12.2019 23:59.
2. You can submit in teams of up to two people. Make sure to write the **name** and the **matriculation number** of all team members on your submission. Your team may consist of two people that are not in the same tutorial.
3. The sheet contains 3 questions. You can reach 10 points on this sheet.
4. This sheet will be discussed in the tutorials in the week of 02.12.19-07.12.19.
5. The pratical tasks require some usage of gdb. You can find a Cheat Sheet which should contain all commands you need for them here
   If you find that some command is missing, you can contact us, and we will add it.
6. Parts 2 and 3 of this exercise require knowledge of assembly. This will be covered in the lecture in the very week that this sheet is released. It is possible to solve Exercise 3 and most of Exercise 2 without any knowledge of assembly, however.

## Theoretical Part

### E2.1 To Be or Not To Be (in the cache) (3 Points)

(a) (+1.5 points) Take a look at the following four level Page Table for 8 Bit virtual addresses, where the fourth level entries denote the physical page index, as well as the following directly mapped Cache for physical addresses:



| Cache Index (Bits 7-4) | Tag (Bit 3-0) | Cache Line |
|---|---|---|
| 0 | 0 | AAAAAAAA |
| 1 | 42 | bbbbbbbb |
| 2 | 1337 | cccccccc |
| 3 | 66 | 42424242 |
| 4 | 3 | 13371337 |
| 5 | 7777 | ffffffff |
| 6 | 1234 | dddddddd |
| 7 | 555 | 713857da |
| 8 | 98414 | dededede |
| 9 | 2 | lpcsgfzt |
| 10 | 68232 | zusdfiel |

Determine whether a process' try to access the virtual address **0xc2** will result in a Cache Hit or a Cache Miss. Please make sure to include all your calculations and give brief explanations of your steps.

(b) (+1.5 points) In the following, you will find three Multiple Choice Questions. **Exactly** one of the answers is correct, so please make sure you only pick one.
You get +0.5 for each correct answer, and +-0 for each incorrect one.
Please provide a short argument for your answer.

1. Copy-on-Write is mainly implemented to...
   (a) ...allow the kernel to be notified when a process accesses kernel pages.
   (b) ...allow for page sharing across processes.
   (c) ...identify processes that write out of their virtual memory space.
   (d) ...flush page changes on disk only if really required, i.e., if a page has been modified.
2. How does the OS solve the problem that a processes' virtual address space is much larger than the physical RAM available?
   (a) This is not a problem, as only one process can execute per CPU anyway, and caches can be used to augment the physical RAM.
   (b) Modern OSes use inverted page tables.
   (c) Only a subset of virtual pages are kept in memory, all others on disk.
3. What does *not* happen upon a context switch?
   (a) CR3 register is updated to point to new page table
   (b) Pages of the old process are swapped out to disk.
   (c) Registers of running process are stored in memory
   (d) TLB is flushed or invalidated (assume no PCID)

# Pratical Part

## E2.2 Reverse Engineering: Security by Obscurity (4 Points)

After BBC's attempt at securely communicating with RSA failed, they switched to something else: They encrypt all their messages with a secret algorithm before the sent them. The material for this exercise is provided via the CTF platform on `https://sec19.scy-phy.net/`.

Our networking experts managed to intercept three binaries: reverse, which they think contains the assembly code of the encryption algorithm, and secret and secret2. They also managed to bruteforce some messages sent by the BBC concerning this algorithm: Apparently, they are annoyed with how they have to "sync" every time for decryption of the ciphertexts, so they want to remove the "syncing". It seems that one of the secret files contains this change to be made to the algorithm, while the other contains the next plaintext to be encrypted.

Lastly, it seems that they used some kinds of encoding for the strings in the secret binaries to ensure that no one can read them even if they were somehow able to extract them from the binaries. The networking experts now came to you to help them understand how the algorithm works, and to compute the next ciphertext. Luckily, the version of the algorithm binary they intercepted still contains most debugging prints.

Your task is now to understand how the algorithm works, i.e. what its steps are, and to reconstruct the algorithm with the change stated in one of the secret binaries. You shall then use the reconstructed algorithm to encrypt the plaintext contained in the other secret binary to obtain the flag.

**Hints and Directions:**

Please make sure to include the following in your submission:

- The flag for this task, if you managed to solve it.
- The secret strings included in both the secret binaries, as well as a short description how you obtained them and what encoding was used on them.
- For each of the computations of the reverse binary a short explanation of what they are doing and how you figured out what they are doing.
- The reconstructed algorithm as source code. For this exercise, you may use C, Java or Python to reconstruct the program in.

Additional hints:

- You can solve most of this exercise without any knowledge of assembly, simply by observing the debugging outputs.
- Your reconstructed program must be equivalent to the original program. You will get small point deductions if your program behaves differently, even if it computes the flag correctly.
- You need not explain the behaviour of the bool_to_string function, as it is very obvious, and just a helper function. Your main focus should be on the functions which begin with "calculate".
- Your reverse engineered code does not need to print the debugging outputs. Neither does it need to do the input validation.
- Within all the actual computation functions, there are only two case distinctions. (Both of these relate to how the current result changes)
- The final output will be a number in the same number representation at the second to last debugging output. The final flag will be FLAG{x} where x is the final output of your reversed program.

- Do **NOT** use any external reverse engineering tools for this Exercise!
- Even if you can't solve this exercise completely, make sure to submit however far you got such that you can get partial points.

## E2.3 My very first Buffer Overflow (3 Points)

Your good friend Dieter Schlau is currently learning how to program C. His first project is a little service which takes some input from the user and simply echoes it back to them.
Dieter has heard of this horrible vulnerability called a **Buffer Overflow**, and made sure to include some protection against it in his code. He is convinced that surely no one would be able to cause a Buffer Overflow now. As a newly becoming expert for Software Exploits, you are not so convinced. Dieter hence gives you his source code and his binary and challenges you to exploit it and trigger the secret function on his server, and steal a flag.

The material for this exercise is provided via the CTF platform on `https://sec19.scy-phy.net/`. Analyze the source code provided in bufoverflow.c to find the vulnerability. State what it is and abuse it to get the flag from the server.

**Hints and Directions:**

Please make sure to include the following in your submission:

- A small description of the vulnerability.
- The exploitation script that you used. Comment it such that it is easily understandable what your steps were.
- The flag for this task, if you managed to solve it.

Additional hints:

- We highly advise to use Python with pwntools for this task.
- If you have not yet learned what a Buffer Overflow is, but want to start with the exercise anyway, you can find an introduction here:
  Introduction to Buffer Overflows
  Note, however, that the explanation is tailored to 32 bit binaries, as we don't want to give away the entire answer :)
- Pwntools offers a function to convert addresses given as integers to strings in little endian format.
- You don't have to disable ASLR, the binaries already has address randomization disabled for its .text segment.
- Try and exploit the program locally first. Once you have a local exploit running, you can easily do the same on the server. The binary running on the server is the exact same one that you have.
- Make sure to **NOT** recompile the binary, as this will cause your local address of the secret function to be different from the one on the server!
- Finally, if you can't solve the exercise, make sure you still submit what you have, such that we can give you partial points if at least some of your ideas were correct.