Information Security
P2 System-Security Challenge v1.0

Winter term 2022/2023

Introduction

When developing software applications, most of the effort is usually spent fulfilling the basic functionality requirements. Often, security is left as an afterthought. However, many programming languages such as C are neither memory-safe nor type-safe, allowing mistakes to happen very quickly and frequently, while often being hard to detect during normal program flow. More and more data breaches or "hacking attacks" are reaching the mainstream news, with many of them being enabled by a quite trivial programming mistake.

In this challenge, you have to exploit some of the most common mistakes. The concrete scenarios for your exercises are constructed, but the underlying errors appeared countless times, not only in some hobbyist project but often in high-profile applications. Remember: **All these things still happen**, and there are countless bugs being discovered in software that is still used today. Furthermore, even if the software does not have any errors that could lead to exploitation, this might not be true for the hardware that is executing the program. Hardware manufacturers spent a lot of time and development efforts to protect against physical attacks, as these are often even more problematic because it is much harder to fix an oversight in a hardware product. Such hardware focused attacks include: (i) passive side-channel attacks, such as trying to gain information about the executing program by observing, for example, the electromagnetic waves emitted by the device, or its energy consumption; (ii) active fault attacks, trying to maliciously introduce errors into a program execution, for example, by getting it to skip execution of a specific instruction by cutting its power supply for a tiny fraction of time.

There are 2 main categories of tasks: Hacklets and Fault challenges. There are a total of 8 challenges in the hacklet category and 3 challenges in the fault category. The number of points awarded for each challenge is stated in this document. If you have any questions, please contact the responsible teaching assistants:
`m.schiffermueller@tugraz.at` (hacklets), `katharina.koschatko@student.tugraz.at` (faults), or ask a question in the IAIK Discord[1]: `#infosec`.

For local testing, we provide you with the secret files, so you can compare your solution. For testing, we will run your program with fresh challenges. On our test system, the maximum execution time is 10 seconds per challenge for the hacklet tasks, for the fault challenges you have 1 minute each.

[1] `https://discord.com/invite/bBbrVSJ`

1 Hacklets

The first part of this assignment is focused around common errors that happen when developing software applications. Many of these errors are specific to the C language, but their general concepts can be found in many other programming languages as well. In fact, many modern programming languages (e.g., Rust) are specifically designed to prevent (or at least make them as unlikely to occur as possible) many of these errors by design. Nevertheless, C and C++ still have a large market share, especially in the area of embedded systems and microcontrollers.

Your task is to analyze the following programs, find the errors and then take on the role of an attacker: Exploit the mistakes in the following short hacklets to gain more privileges than originally intended by the authors. Concretely, for all of the following hacklets the goal is to read the contents of a file called `flag.txt` in the same folder.

Framework

The challenges are written in standard C and have to be solved in Python (Python3). We recommend the following packages to be installed:

- pwntools

On a standard GNU/Linux system, the following should install pwntools for python3:

```
sudo apt-get update
sudo apt-get install python3 python3-pip python3-dev git libssl-dev libffi-dev build-essential
python3 -m pip install --upgrade --user pip
python3 -m pip install --upgrade --user pwntools
```

We provide a docker container for ease of development, which has an identical setup to the automated test system and also has all necessary packages and libraries pre-installed. A setup-script for this container is included in the upstream repository.

To use the provided container, you will need to install docker on your system. A detailed description on how to install docker can be found at `https://docs.docker.com/get-docker/`.

The script offers three different commands

- To get the latest Docker image, execute `./docker.sh update`
- To run Docker in the current directory, execute `./docker.sh run`
- To run Docker in a different directory named `hacklet` , execute `./docker.sh run hacklet`

If you do not want to install docker on your system, you can use the VM provided at `https://seafile.iaik.tugraz.at/f/fa780f1d5a48412aa69e/?dl=1`. This virtual machine comes with all dependencies that you will need to solve this assignment. The credentials for the VM are `infosec:InfoSec123!`

Each challenge is contained in a subdirectory, the concrete folder is stated for each challenge in this document. For the hacklet category, each challenge folder contains the following (except for the *Bonus* hacklet, which does not contain `main.c` and `Makefile`):

- C file (`main.c`)
- Binary ELF executable (`main.elf`), a pre-compiled version of the binary which is identical to the one used on the test system.
- `Makefile`, to show you the compilation options for the binary.
- `exploit`, a template to get you started for your exploit. You can choose to use `python3` or even a plain `bash` script, just change the shebang accordingly.
- `flag.txt` file. These files contain the information you need to read by exploiting the given binary.
- Optional: additional files (e.g., password files)

For all tasks, make sure your exploits work with the **precompiled, unmodified binary**, which is the one that you are given. You are free to modify and compile the source file yourself and try to attack this modified version if that helps you to arrive at a final solution, but your submitted solution has to work on the **precompiled, unmodified binary**.

1.1 Buffer Overflow (1P)

Folder: `hacklets/01_BOF`

Arrays are one of the most fundamental data structures that are used in programs. They allow the storage of a fixed number of values of a certain data type and usually cannot be resized once created. In C, an array can be created using `type array_name[ array_size ];`. If we, for example, write `int numbers[3];`, we declare an array of 3 integer variables that can subsequently be accessed using `int a = numbers[0], b = numbers[1], c = numbers[2];`. Notice the zero-based indices, which lead to a very common error for programming novices who instead try to access the array in the following way: `int a = numbers[1], b = numbers[2], c = numbers[3];`. While the variables `a` and `b` are well defined, but probably do not point to the intended value, what value does the variable `c` hold?

C, unlike many other modern programming languages, does not have runtime checks for bounds during array accesses. While this is obviously better for performance, since no time is spent doing comparisons for every array access, this can lead to very critical bugs, since the variable `c` just gets assigned whatever is at the position in the memory following the original numbers array. This can again lead to unintended information leaks. The situation gets even more dangerous when the user is allowed to write outside the intended bounds of the array, potentially overwriting the content of other, maybe security critical variables or divert the control flow of the program by overwriting return addresses or function pointers.

Challenge. You are given a half-finished password manager that reads a `username` and a `password` from `stdin`. The executable also contains a method which reads and prints the flag, which is the security-critical information that you want to have! Exploit a flaw in the program in order to hijack the control flow so that the flag function is called.

Hints. Take a look at the compiler flags and analyze the binary using `checksec`. What does `PIE` mean in this context? Furthermore, the `ELF` class of `pwntools` can help you find the location of functions in the binary. Also, make sure that you understand how function calls are handled and where return addresses are stored.

1.2 Format String Attacks (2P)

Folder: `hacklets/02_Format`

IO functionality, that is, the ability for a program to send output to a user or read input from the terminal or a file, is one of the most crucial parts of software development. Pretty much every program has a way to get some input from the user and to then react dynamically to that input, producing an output in some fashion. In C, the most well-known function for printing characters is `printf`, which **print**s **f**ormatted data to the standard output.

The definition of `printf` is as follows: `int printf(const char* format, ...);`. The first argument is a so-called *format string*, which can include *format specifiers* (e.g., `"%d"`). Additionally, `printf` is a *variadic function*, which can take a variable number of elements to enable the combination of multiple format specifiers in a single format string. Each format specifier should be paired with a corresponding argument to `printf`, e.g. `printf("%d", 5);` prints 5 as a signed integer.

Problems can occur if the number of format specifiers and additional arguments are mismatched. If there are too many format specifiers, the program will just try to get a value from the expected position (registers or stack) where a corresponding argument's value would normally reside. However, this can leak information about other variables that were never intended to be printed. The situation gets even worse when the format string itself is under the control of the user, since a malicious user can now force situations as described before on purpose.

Challenge. You are given a small program in which a chatbot challenges you to access the flag. You have 5 tries to impress the big-headed chatbot. It receives your trials from the standard input and prints

them back to the standard output - obviously to mock you. Abuse an error in this functionality to unveil the secret flag.

Hints. Have a look at the different format specifiers and their effects. Surprisingly, not all of them will write something to the standard output.

1.3 Use After Free (2P)

Folder: `hacklets/03_UAF`

Dynamic memory allocations are a standard feature in most programming languages, as it is often useful to allocate a dynamic amount of memory to store data based on the user's input. One example could be an image editor, where the program needs to allocate a buffer to store the content of the arbitrarily sized image.

Usually, such a dynamic memory allocation can be thought of in terms of resources. The program makes a request for a *resource*, this resource gets allocated, and a *resource handle* is returned, which can be thought of as a reference to the specific resource. Once the resource is no longer needed, it can be returned by invoking a function on the resource handle. In C, we can request memory using the `malloc` library function, which returns a pointer to the allocated memory. This pointer is our resource handle. We can return the memory by passing the resource handle to the `free` function.

Observe a problem with this previous workflow: The `free` function does return the resource (the allocated memory), but does not invalidate the resource handle (the pointer). Even zeroing the pointer does not always help, since it could have been copied or passed to another thread in the past. We are left with a so-called dangling pointer, a pointer that points to previously freed memory.

While this obviously is bad, since accessing this pointer after the underlying memory has been freed will most likely crash the program, it can get even worse. What if instead the memory location that was previously assigned to some request (and we have a pointer to), gets freed and then reused for some other request?

Challenge. You are given a small C implementation of a simple access system. Exploit a problem in the program to gain access to the red door, which reveals the secret flag.

Hints. Try to understand how `malloc` works and how free chunks are used on subsequent allocations.

1.4 Shellcode Injection (2P)

Folder: `hacklets/04_Inject`

A compilers job is to translate the source code written in a high-level programming language to a low-level target language, e.g., assembly or machine code. The generated machine code for the target architecture is then stored as part of the binary executable, where it will later be loaded and executed.

This machine code is just a combination of specific bit sequences that are then interpreted by the CPU, which then performs the desired operation. During normal execution, the memory of a program is segmented into different sections, so that the executable machine code is at a different location in memory than, for example, the content of a local variable. The special *instruction pointer* register points to the location in memory that holds the currently executed instruction and, during normal program flow, the instruction pointer should never point to memory locations not designated for code.

However, different bugs can lead to a violation of this fact and cause the instruction pointer to point to attacker-controlled memory.

Challenge. You are given a very minimal program that reads data into a fixed-size buffer. The program also contains a global variable called *random_nr* which is set to a random value. For this task, you **do not** have to retrieve the flag. You task is to calculate the `xor` of your group number and the

global variable *random_nr* (i.e. $xor\_value = group\_nr \oplus random\_nr$). Then write *odd* or *even* to stdout depending on whether *xor_value* is odd or even. Finally, exit the program with the calculated *xor_value*.

Hints. You need to write assembly code to solve this challenge. Remember the different assembly addressing modes and find out how system calls work in the target architecture. Finally, *pwntools* provides some useful tools to convert assembler code to machine code.

In general this type of exploit would not work on most modern systems due to a countermeasure called $W \oplus X$ or "write xor execute". This, in short, means that a section of memory is either writable or executable, but never both. This simple policy prevents simple attacks like modifying the binary code of the program or executing code that you wrote into a stack array. However, if you look at the `Makefile`, you may see some specific compiler options that are beneficial for an attacker.

1.5 Race Conditions (1P)

Folder: `hacklets/05_Race`

Accessing the same resource from multiple threads or processes can lead to unwanted side effects, depending on the order of execution. Such a situation is called a *race condition* and can lead to potential vulnerabilities. A special class of bugs caused by race conditions are TOCTOU bugs. Such bugs can occur if a program performs a check on a resource and, depending on the result of the check, performs actions on the resource somewhere later in the program. If the time between the check and the use of the resource is sufficiently long, an attacker can use this time to switch the resource and thus force the program to perform actions on data which would potentially not pass the check.

Challenge. You are given a program which derives a password from a seed and an iteration count. Only the correct parameters will cause the program to open and print a user-specified file. The program also tries to make sure you are not opening `flag.txt`.

Hints. Some filesystem operations may help you in writing your exploit. Also, the password derivation function uses a so-called SBOX as a nonlinear element. If you are stuck finding the correct seed and iteration count, try to visualize this SBOX on paper. This is not a cryptography challenge!

1.6 Timing (1P)

Folder: `hacklets/06_Timing`

Timing attacks utilize side-channel attacks to gather information by measuring how long it takes a process to complete a certain task. The time it takes a computer to carry out each operation is measured through timing attacks. Timing measurements can be noisy due to factors like network latency or CPU scheduling or because they contain operations that are irrelevant to side-channel analysis. Once the noise has been reduced to an acceptable level, it is possible to gain information about the analyzed program that can be used to exploit it. Security systems and cryptographic algorithms must be protected against timing attacks, which are powerful side-channels.

Challenge. You are given a binary that asks you to enter a secret password. The password is stored inside a file to which you do not have access. However, if you manage to provide the correct password, the program will print out the flag.

Hints. When performing timing side-channel attacks, the tool used to obtain the time measurements should be chosen to fit the amount of precision required. For this task, it suffices to simply use the Python *time* function to obtain the needed measurements.

1.7 Combination (2P)

Folder: `hacklets/07_Combination`

Obvious errors, like those in the previous examples, are very unlikely to occur in real software. Real-world exploitation might be a multi-level process in which you need to use several vulnerabilities to reach your desired goal. It is important to understand that vulnerabilities, which might not be that bad on their own, can compromise the security of a system when combined creatively.

Challenge. You are given a program that wants to test your luck. You have to guess at least 32 randomly generated numbers which change every time you start the program. If you can guess all of them correctly you are rewarded with a flag.

Hints. For this challenge you have to apply the knowledge you gained in the previous challenges and exploit not only one but multiple vulnerabilities. Make sure you are familiar with the concept of signed and unsigned data types. You should also make yourself familiar with how 32 bit binaries handle function parameters.

1.8 Reverse Engineering (Bonus) (1P)

Folder: `hacklets/08_Rev_Bonus`

In many practical scenarios, the source code is unknown to an attacker. The practice of *reverse engineering* binaries pursues the goal of recovering information about the inner workings of a compiled program. This knowledge can in turn be used to attack the application, e.g. by looking for bugs and undocumented features.

Challenge. You are given the binary of a program that checks if you know its secret encryption mechanism. In particular, you have to provide a valid plaintext-ciphertext pair. You need to reverse engineer the encryption mechanism and manually encrypt an appropriate plaintext.

Hints. Use a disassembler and / or a decompiler (e.g. `Ghidra` or `Cutter`) to scrutinize the binary and analyze how the password check is performed.

Useful Resources

Here is a short list of tools and resources you might find helpful.

- Pwntools documentation: `http://docs.pwntools.com/en/stable/`
- `gdb <executable>`, the GNU Project debugger
- Plugins for gdb (only activate one at a time, as they will probably clash with each other)
  - `peda`, a gdb plugin for exploit development
  - `pwndbg`, a gdb plugin for exploit development
  - `gef`, a gdb plugin for exploit development
- `valgrind --tool=memcheck <executable>`, a memory checker
- `readelf [-a] <executable>`, displays information about ELF files
- `objdump`, and its `-d` flag for disassembling a section of code
- `radare2`, an advanced disassembler and debugger
- `Ghidra`, a software reverse engineering framework with built-in disassembler and decompiler

2 Fault Attacks

"Welcome to the Faulty Caves, adventurer. Herein lie vast treasures waiting to be discovered, but also careless adventurers of long gone and not-so-long gone semesters. You can always return to me for advice, however once you are in there, I won't be there to guide you. Now go. Your own adventure is just about to begin."

The second part of this exercise is focused around fault attacks. In such an attack, the adversary somehow brings a device briefly outside of its specification and thereby **causes errors in the computation**. Faults can be introduced using, for instance, brief over- or undervolting, temporary overclocking (clock glitches), EM pulses, and lasers. When done carefully, these methods can cause, e.g., instruction skips and memory corruption of various sorts. These effects can be used to recover cryptographic keys and bypass many security checks, thereby gaining access to secure systems.

Fault Simulator

In this exercise, we do not cover fault-injection techniques, but rather focus on ways how certain faults can be exploited. To do this, we provide a **fault simulator**, which performs single stepping of a binary and allows manipulation of its execution (corrupting memory, changing the instruction pointer, etc.).

Injected faults are specified in a file. Faults are triggered either by a given value of the instruction pointer RIP (example: `@0x12ab`), or by counting the number of executed operations (example: `#3000`). The latter is a rough measure for the time since startup. Possible faults include manipulation of the instruction pointer (e.g., to skip instructions) and memory manipulation (`havoc` randomizes bytes, `zero` sets bytes to zero, `bitflip` flips a single bit in a byte). The attacked binaries specify which trigger types and which fault types are allowed for the attack. The simulator is called with two arguments, the first points to the file specifying the faults, the second argument is the targeted binary. For the challenges, a simulator with hardcoded arguments is provided.

For further information on the simulator, have a look at the `README` located in the `faults` folder. For demonstration purposes, we included demo attacks. You can run them using: `./simulator demos/victim<nr>.fault demos/victim<nr>`

Framework

You have to solve several challenges, each one requires injecting faults in the execution of a provided binary. Each challenge is contained in a subdirectory, the concrete folder is stated for each challenge in this document. Challenge folders contain the following:

- C sources of the attacked binary ((`<folder_name>.c` and sources of included libraries, if any)
- Binary ELF executable (`<folder_name>`), a pre-compiled version of the binary which is identical to the one used on the test system
- Fault simulator configuration script `<folder_name>.fault` specifying all faults you want to inject
- Fault simulator binary `simulator` performing fault injection on the target executable `<folder_name>` with the script `<folder_name>.fault`. Call as `./simulator <victim_args>`
- `exploit.py` file for challenges requiring post-processing of the faulty output. This script contains functions for calling the fault simulator and parsing the output.
- `exploit.sh` file for challenges that do not require any post-processing. Call this script as a shorthand for the fault simulator with the correct inputs
- `Makefile`, to show you the compilation options for the binary
- Optional: additional files (e.g., key files, auxiliary python scripts, etc.)

The attack is started by running `exploit.{py|sh}`. For solving the challenge, you are supposed to **modify the configuration file `<folder_name>.fault` and the python script `exploit.py`** (for some challenges). For all tasks, make sure that your exploits work with the **precompiled, unmodified binary**, which is the one that you are given. Also note: on the test system, access to all secret files (keys etc.) is locked.

2.1 The Safe (1P)

Folder: `faults/01_safe`

Wandering around, you stumble upon a sturdy-looking safe with sixteen dials on the front. Curious as you are, you want to know what the safe may contain. Upon closer inspection, you notice that, contrary to the safe's external appearance, the locking mechanism seems to be of mediocre quality at best. Not having the correct code to open it and lacking the time that would be needed to try all combinations, you decide to give the safe a healthy kick. The mechanism makes a peculiar noise, but the safe remains locked. For good measure, you kick it several more times, the mechanism clearly getting disturbed in various ways. Maybe, just maybe, if you kicked it in exactly the right location, it might open.

The security of any program can only be guaranteed as long as it is executed correctly. Even seemingly minor faults, such as skipping over a single instruction, can have catastrophic consequences. This introductory challenge demonstrates this.

Challenge. You are given a binary which, when given the correct password, prints a secret message. The correct password and the secret message are stored as files in the challenge folder, the entered password is given as first command-line argument: `./safe <combination>`

Your task is to **print the secret message without knowing the password**. You should do that by **injecting a single skipping fault** in the execution. That is, at a chosen point during execution, you should manipulate the instruction pointer (add or subtract some small number) and thus skip over one or multiple instructions.

This challenge should be solved by simply adding an appropriate line in `safe.fault`. No post-processing is required, so simply run `exploit.sh` for starting the simulator. Do not change `exploit.sh`, as we test using this script, and exploits may depend on the length of the specified password.

Hints. Have a look at the disassembly and try to determine an instruction that, when skipped, allows bypassing the security check. Then try to determine the correct instruction-pointer offset to skip over this instruction.

2.2 Fault Attacks on Deterministic Signature Schemes (2P)

Folder: `faults/02_eddsa`

That crusty old safe posed no obstacle to you. Venturing deeper into the vast system of natural caves and artificial tunnels, you come across a place that had apparently been a campsite very recently. It seems like something forced the former inhabitants to leave in a hasty manner. You search the site, but you do not find anything of particular interest. About to leave, you notice something strange in the rock wall. Just above your head, there is a small leather case neatly tucked into a deep gap in the rock. Inside the leather case, you find an expensive-looking device in a brass enclosure. On the front, there is a set of dials labeled "Message", and a display of numbers with the label "Signature". Engraved in the device's back cover is the text "Bernstein & Co. Precision Instruments Signature Generator". Strangely, the device feels very familiar to you. You discover that the delicate mechanism is rather easy to disrupt in its calculations, yielding different, seemingly random numbers as a result. Your gut feeling tells you that the application of force alone will not help you this time. You write down a few of the numbers that the device gave you. You carefully place the device back where you found it and leave the premises, knowing that the owner will likely come looking for it.

Secure and correct generation of random nonces has often been a problem in the past (see also `nonce_reuse_asym` in P1). For this reason, some more recent protocols don't use a random nonce, but instead use a **deterministic nonce generation**. This can be done by setting the nonce to the hash of the message m together with a secret value h. That way, a nonce reuse for different messages is equivalent to finding a hash collision, which should not be possible for a cryptographic hash function.

One example of such a protocol is the Edwards-curve Digital Signature Algorithm (EdDSA). All of its algorithms are given below.

Algorithm 1 EdDSA Key Generation

Input: Public parameters (q, B)
Output: Private key (a, h), public key A
1: Pick a random a mod q
2: Pick a random bit string h
3: Compute public key $A = a \times B$ ▷ Point-Scalar Multiplication
4: **return** sk $= (a, h)$, pk $= A$

Algorithm 2 EdDSA Signing Algorithm

Input: Message m, private key (a, h), public parameters (q, B)
Output: Signature (R, s)
1: $r = H(h, m)$ ▷ Deterministic derivation of nonce r
2: $R = r \times B$ ▷ Point-Scalar Multiplication
3: $s = (r + H(R, A, m) \cdot a)$ mod q
4: **return** (R, s)

Algorithm 3 EdDSA Verification Algorithm

Input: Message m, signature (R, s), public key A, public parameters (q, B)
1: Signature is valid if $s \times B = R + H(R, A, m) \times A$

Challenge. You are given a binary that runs the EdDSA signing algorithm. As input, it uses a message string given as command line argument: `./eddsa <tobesignedmessage>`. As output, the program prints the signed message (signature and message) as a hex-string. The key is read in from the file in the directory. Your goal is to **recover the signing key a with a fault attack**. You can use **memory corruption (bitflip, zero, havoc)**, but only with an **instruction-count trigger**.

Hints. Observe that in the signing algorithm, the message m is processed twice (line 1, line 3). What happens when you corrupt/change the message in between? Compare to the case of signing without faults. Think back to `nonce_reuse_asym` of P1.

The point-scalar multiplication (line 2) is by far the most time consuming operation in signing. Also note that m gets copied into the global buffer `signed_message` at the start of the algorithm and is then only read from there (see `ref10/sign.c`). A single fault injection should be sufficient for key recovery.

2.3 Differential Fault Attacks on AES (2+1P)

Folder: `faults/03_aes`

After pondering the numbers you obtained from the Signature Generator *for a while, you figure out the secret it unintentionally divulged. Already starting to feel a bit tired, you begin looking for a safe place to rest for the next few hours. In your search, you come across a fellow adventurer. They show you a device not unlike the one you had just seen. They seem to know that the device holds some value, even though they are not privy to its purpose. The device bears some rough lettering, reading "AES DEC". The other adventurer agrees to let you have a try at uncovering the secrets of this device.*

Symmetric cryptography can also be target of fault attacks. There, injected faults often need to be more precise (in terms of timing). Key recovery then often works in a *divide-and-conquer* fashion. Pieces of the key, such as its bytes, are recovered individually by trying all possible values and determining which one fits the injected fault.

Challenge. You are given a binary which performs AES decryption of a single 128-bit block. The ciphertext is given as the first command-line argument as a hex string, the output is written to stdout. The key is taken from a file. Your task is to **recover the key k with a fault attack**. You are only allowed to **use bit flips** in combination with an **instruction-pointer (RIP) trigger**. Since this challenge is a bit more involved and also requires you to have a deeper look at the internals of AES, solving it will not only give you the standard two point but will also reward you with an additional bonus point (in essence, this challenge is a 3P one...)!

Hints. You are supposed to perform a **differential fault attack**. That is, you run decryption twice, once with fault, once without fault. For each possible value of a key byte, use the output and compute back to the point where the fault was injected. Then, for each possible byte value, check if the difference between the true and the faulty value corresponds to the injected fault.

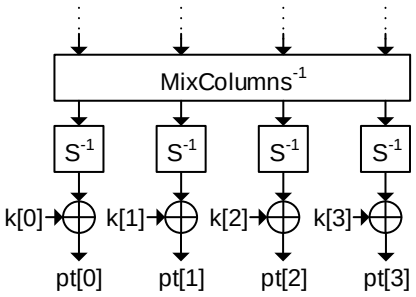

Figure 1: AES decryption, last operations for 4 of 16 output bytes

As a first step, determine where in the decryption process you want to inject your faults. You should pick an intermediate which, when computing backwards, only depends on a small number of key bits, i.e., a byte. To help you, the last couple of operations in the decryption process are shown in Figure 1.

As a hint, first look for fault locations such that a single bit flip causes only one output byte to be different. Such a fault might not allow an attack, but it is a good start. The solution will require more than a single fault per execution. First, however, focus on recovering a single key byte. The attack will require running decryption more than once (with different inputs).

Our AES implementation works in place. That is, the input array `ct` is directly used for storage of the AES state. Thus, in the end of the algorithm, the plaintext can be found in the same memory location.

Decryption vs. Encryption. Encryption can be attacked in exactly the same way (except for the reversed roles of plaintext and ciphertext). We chose to target decryption, since an attack can then directly recover the master key k. An attack on encryption would recover the last round key instead. As the key schedule of AES is reversible, the master key k can still be computed with an additional step.

More powerful attacks. In reality, attackers rarely have the ability to flip bits with such high precision. However, there do exist many more powerful attacks capable of exploiting, e.g., randomization of entire bytes.

3 Assignment interview

In order to help you prepare for the assignment interview, we have a small overview of possible questions. First off: Both team members should be able to answer, regardless of who actually solved the challenge.

3.1 Hacklets

You will need to explain how your exploit works in detail. More precisely, think about the following questions:

- Which security vulnerabilities did you find? How can they be exploited, in general?
- Explain your exploit in detail. In particular:
  - What is the general idea of your exploit? How can the security vulnerabilities be exploited such that the control flow of the program is altered in a favorable way.
  - What are the steps to reach your goal? Sketch the stack layout / heap layout / etc.
  - Explain the ingredients to your exploit. How did you find variables / addresses / offsets / etc. you used?
- Which security mechanisms would have prevented this exploit, besides not making programming errors. Have a look at the Makefile.

3.2 Fault attacks

You will need to explain how your exploit works in detail. More precisely, think about the following questions:

- Explain the theoretical idea of the fault attack. Which fault do you want to inject and how does the faulty program execution help you to reveal the secret information.
- Where do you inject a fault? In particular:
  - Explain the used fault and its arguments.
  - Use the tool of your choice to reproduce how you found the correct IP / instruction count / address.

4 Version History

v1.0

Initial release of P2 assignment sheet.