CSI4109 Assignment #4
Due: May 23th 1:59pm

Code Integrity

We studied in class the structure of an ELF-format executable file. In this homework assignment, you will learn how to *cryptographically* enforce a "Code Integrity" policy. In particular, you will enforce this policy, for a given ELF 64-bit executable file, by signing its executable sections (e.g., `.text` section). Once an executable file has been signed, the integrity of its code sections can be enforced by verifying the signature before we run it; if verification goes successful, we know that the signed sections have not been tampered with nor forged after signing.

To this end, you will write a program in the C/Rust programming language, called `signtool`, which is capable of (i) signing the executable sections of a given executable file, and (ii) verifying the signature. Throughout this assignment, you will learn what it means/takes to ensure the integrity of the code that natively runs on your CPU. As a side effect, you will get yourself much more used to the structure of ELF-format executable files.

Part A. Signing and Verifying Code Sections

RSA Key Generation. We will use a pair of RSA public and private keys for this assignment, which is *not* protected with a passphrase. So the RSA key pair, `private_key.pem` and `public_key.pem`, will be created as follows.

```
$ openssl genrsa -out private_key.pem 2048
$ openssl rsa -in private_key.pem -out public_key.pem -pubout
```

Signing Executable Sections. Your `signtool` must support the following command-line interface, which will be used to sign the executable sections (i.e., those that are included in an executable segment) of a given executable file.

```
$ ./signtool sign -e <path to executable> -k <path to private_key.pem>
```

- This command takes an executable file (specified by `-e`) and an RSA private key (specified by `-k`) as input, and produces as output a signed excutable file whose file path is the file path of the input executable appended with `-signed`.
- You can choose the signing algorithm used to generate a signature of the executable sections. We require, however, that the signature must be contained in a new section called `.signature`. The signed output executable file must contain this new section.

- You can assume that the input arguments are always refer to valid executable and private key files, and you must overwrite the output file if already present.
- When writing `signtool`, you can use the OpenSSL library sign the executable using the given RSA private key. For adding a new section to the binary, you can use the function `add_section` declared in `csi4109.h`, but you are welcome to implement your own or modify the function. Unfortunately, we do not provide such code in Rust; you would have to implement it yourself.
- When parsing a given ELF file, you can use open-source software, as long as you comply with their license (see **note on using open-source software** given below). You are also more than welcome to implement your own parsing logic (**Hint:** look at & use `elf.h`); this *could* be easier for you, because your program needs to parse only a portion of the ELF file.

Verifying Code Integrity. Your `signtool` must also support the following command-line interface, which can be used to verify the integrity of the executable sections found in the signed executable.

```
$ ./signtool verify -e <path to signed executable> -k <path to public_key.pem>
```

- This command will read the signed executable as well as the public key (which corresponds to the private key that was used to sign the executable), and perform the code integrity check (i.e., verify the signature).
- Three possible outcomes are: `OK`, `NOT_OK`, and `NOT_SIGNED`. Print on standard output (i) `OK` if the integrity check succeeded, (ii) `NOT_OK` if the check failed, or (iii) `NOT_SIGNED` if the given executable has not been signed by `signtool`.
- Each output (`OK`, `NOT_OK`, and `NOT_SIGNED`) on standard output must end with a newline character.

Part B. Bypassing Code Integrity

In this part, you are given an ELF executable called `licensechk-signed` that has its executable sections signed using `signtool`. This executable, when invoked, asks for a valid license number to be entered, and tells whether the entered number is `VALID` or `INVALID` license number. The following shows a **portion** of the source code it was compiled from.

```
#include <stdio.h>

void license_check_ok(void);

int do_license_check_v1(char *buf, size_t bufsize);

int do_license_check_v2(char *buf, size_t bufsize);

int (*check_license)(char *, size_t) = &do_license_check_v2;

int main(int argc, char **argv) {
    puts("Please enter license code (v2):");
    char buffer[32] = { 0 };
    scanf("%30s", buffer);

    int ret = check_license(buffer, sizeof(buffer));

    if (ret) {
        puts("INVALID");
    }
    else {
        puts("VALID");
        license_check_ok();
    }

    return 0;
}

int do_nothing(char *buf, size_t bufsize) {
    return 0;
}
```

In this part, you will edit the given executable `licensechk-signed` **at the binary level** such that the edited executable passes the license check *without entering a valid license code*, **while still passing the signature verification that checks the code integrity**. The expected output of running the given executable `licensechk-signed` and a successfully edited `bypass-licensechk-signed`, with an *invalid* license code as input to both, is as follows.

```
$ ./licensechk-signed
Please enter license code (v2):
invalid_license_code
INVALID

$ ./bypass-licensechk-signed
Please enter license code (v2):
invalid_license_code
VALID
```

Please note that the signature verification of `bypass-licensechk-signed` must succeed as well:

```
$ ./signtool verify -e ./bypass-licensechk-signed -k public_key.pem
OK
```

- Store the executable you edited as a new executable file called `bypass-licensechk-signed`, submit it.

Implementation

- **Your program must be written in C, Rust, or a combination thereof.** This means that your program could be wrriten entirely in C or entirely in Rust, and you can use Rust's foreign function interface to call into C functions in your Rust program.
- **Your program must work on Ubuntu 22.04 64-bit with the default packages installed.** In addition to the default packages, the following packages for developing in the C/Rust programming language as well as the OpenSSL & ELF libraries are also installed:
  - C (`gcc`)
  - Rust and Cargo (`rustc` and `cargo`). Use the following crates if you would like to use them:
    * `hex-literal = "0.4"`
    * `sha2 = "0.10"`
    * `openssl-sys = "0.9"`
    * `openssl = "0.10"`
    * `libelf = "0.1.0"`
  - OpenSSL library (`libssl-dev`)
  - ELF library (`libelf-dev`)

  You'll probably need to set up a virtual machine to do your development. VirtualBox is a free and open-source VM system. Or, if you are using Windows, you may want to use WSL (WSL version 2 is recommended.) (Ubuntu 22.04 on Microsoft Store).
- **Note on using open-source software:** Referencing and using publicly available source code licensed under an open-source license is fine, **provided that you fully comply with the licensing terms**, if any, enforced by the authors. Note, however, that you are still **not allowed to share your code with other students**. If you do use open-source software in any way in your HW submission, please make sure to list all of them them along with their licenses.

Submission Instructions

Submit on LearnUs (`ys.learnus.org`) your source code (including `csi4109.h` and `csi4109.c` if you used them), along with a `Makefile` and `README` . The `Makefile` must create your executable, called `signtool` , when the command `make` is run. Note that we may invoke `make` multiple times, and it needs to work every single time. Also make sure to include `bypass-licensechk-signed` in your submission. Your `README` file must be **plain text** and should contain your name, student ID, and a description of how your `signtool` works (mention the signing algorithm you used) and how you edited `licensechk-signed` .

Grading Rubric

- All files exist and `make` successfully creates `signtool` that is executable. (1 pt)
- A new `.signature` section exists in the signed executable. (1 pt)
- The signed executable works functionally the same as the original executable. (1 pt)

- Resistant to attacks that corrupt any executable section after signing. (2 pts)
- Resistant to attacks that corrupt the content of the `.signature` section after signing. (1 pts)
- Resistant to attacks that attempt to verify using an invalid public key. (1 pts)
- Error handling according to the spec. (1 pt)
- The edited signed executable (i) passes the code integrity (signature) check, (ii) runs functionally the same as before your edit, and (iii) bypasses the run-time license check. (2 pts)