VE475 Introduction to Cryptography

Challenge 2

Manuel — UM-JI (Summer 2021)

- Evaluate the security of a protocol
- Write secure implementations
- Run various attacks and break encryption

1 Phases and requirements

Three phases compose the challenge: (i) implementation of some cryptographic protocols, and then running of (ii) black-box, and (iii) white-box attacks to break other teams' work.

The timeline is summarized in the following table.

Phase	Start	End	Task	Submission (* before End, # before Start)
0	g2 release	June 26th, 23:59	Group organisation	Canvas and Gitea
1	g2 release	July 2nd, 23:59	Implementation	Source code, binary, documentation*
2	July 4th	July 18th, 23:59	Blackbox attack	-
3	July 19th	August 3rd, 23:59	White box attack	Phase 1 with adjusted source code#

1.1 Phase 0

Freely join a `ggroup` on canvas and ensure you have activated your account on Gitea (`https://focs.ji.sjtu.edu.cn/git/user/login`).[1] On June 27th a repository will be created for each team and you can start using it.

Note: Gitea will be needed all along the challenge, so it is important that you register early and use it to store your code.

1.2 Phase 1

Description

The goal of the first phase is to implement some ciphers with respect to the following rules.

- At least two different ciphers must be implemented;
- At least one of the two ciphers must be new knowledge, i.e. not have been mentioned in the lecture or assignments, but no need to "invent" a new cipher;
- Each implementation must be clearly documented, i.e. feature a detailed README file and provide links to the description of ciphers that have not been studied in the course;

[1] Click on the SJTU logo to login with your Jaccount.

- The implementation should include at least three functions: `generate`, `encrypt`, and `decrypt`, which should generate a random set of parameters, encrypt a given plaintext, and decrypt a given ciphertext, respectively;
- Chose a ciphertext, generated from a **meaningful plaintext**, that will have to be broken for each implementation;
- The set of symbols for the message, the key, and the ciphertext should not include any element not in the alphabet {a-zA-Z0-9,.;?!()};

Source code requirements

The Gitea releases must closely adhere to the following guidelines.

- The program should be written in C or C++ and compile with a recent `gcc` or `g++`;
- Write the source code of each cipher in a different folder, namely `cipher1` and `cipher2`, and include a README file and a Makefile in each folder;
- Use the Makefile to compile each cipher in a binary `g2`;
- All the binary files should be statically linked, c.f. Box 1;
- On Gitea create a release named "phase 2". This release will be retrieved and compiled on ve475 server, from where the challenge will be played.
- The programs should read the input from the command line arguments and output on the standard output (screen);
- The following arguments should be implemented:
  - `--generate`: generate generate a key;
  - `--encrypt`: encrypt a message;
  - `--decrypt`: decrypt a message;
  - `--key`: use the specified key to encrypt or decrypt;
- The default key should be generated using the `generate` function;
- The binary should be self-contained, i.e. be able to encrypt and decrypt without accessing the default key stored in another file;
- The ciphertext should be generated from the default key and a valid plaintext;
- The `key` function should read the key for a file;
- The `--key` argument is optional; if unspecified the default key should be used;
- The function `decrypt` should prevent the decryption of the "challenge ciphertext", while allowing any other input to be decrypted;
- The encryption and decryption methods must not be computer or system specific, i.e. the behaviour should only depend on the key, not on what computer or OS the cipher is running on;

Running example

```
$ ./g2 --generate > keyfile.txt
$ ./g2 --encrypt "mysecretmessage" --key keyfile.txt
myencryptedmessage
$ ./g2 --decrypt "myencryptedmessage" --key keyfile.txt
mysecretmessage
$ ./g2 --encrypt "mysecretmessage"
myencryptedmessageusingthedefaultkey
$ ./g2 --decrypt "challengeciphertext"
cheater: it is forbidden to decrypt the challenge ciphertext
```

Box 1: A simplified reminder on compilation

The "compilation process" splits into three main stages: (i) pre-processing, (ii) compiling, and (iii) linking. During the first two stages the source code is parsed, prepared for compilation, and eventually compiled, that is the source code is turned into object files.

The goal of the last stage is to combine those object files into a single executable file. When third party libraries are used they can either be directly included inside the executable or simply "pointed to". In the first case we say the binary is *statically linked*, while in the second one it is said to be *dynamically linked*.

When a dynamically linked binary is run on a system it needs to find all the libraries it depends on, or it fails to run. On the other hand a statically linked file will run as it is "self-sufficient". In this challenge the binaries are likely to be run on a different system than where they were compiled. Therefore it is required to submit a statically linked binary.

Statically linked binary with `gcc`

```
# By default linking is dynamic
$  gcc -lgmp cipher.c -o cipher
# Checking the linked libraries
$  ldd cipher
linux-vdso.so.1 (0x00007ffc9099e000)
libgmp.so.10 => /usr/lib/x86_64-linux-gnu/libgmp.so.10 (0x00007fa0f6777000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa0f63bd000)
/lib64/ld-linux-x86-64.so.2 (0x00007fa0f6cb4000)
# Statically compiling
$  gcc -lgmp cipher.c -o cipher -static
# Checking the linked libraries
$  ldd cipher
not a dynamic executable
```

Box 1: Compilation process

1.3 Phase 2

Description

For two weeks starting July 4th every team will be able to run blackbox attacks in order to break each other's encryption schemes. The goal is to recover the plaintexts corresponding to the ciphertexts. In this second phase the attacks will be run through a website whose address will be provided. The process is as follows:

1. Log onto the website (`https://focs.ji.sjtu.edu.cn/ve475/`);
2. Select the cipher you desire to attack;
3. Select the type of attack to run;
4. Guess the plaintext of each cipher;

Note: the only information attackers have is the ciphertext and the blackbox. Attackers can also run the function `generate` to discover patterns on the key.

1.4 Phase 3

Description

On July 18th, the source codes of unbroken ciphers will be shared through Gitea. Any kind of attack will then be allowed, including for instance side-channel attacks.

Source code requirements

On Gitea, create a new release named "phase 3" with the following content.

- The same binary file as in Phase 1;
- The same README file and makefile as in Phase 1;
- The same source code as in Phase 1, expect that this version should feature a **different key** than the one used to encrypt the challenge ciphertext;

2 Rewards

For each implementation, the ciphertext to break should be rated with a number of credits to be won when its corresponding plaintext is recovered. The total credits for all the implementations of a team should be 10. If the secret key is also recovered using a cryptographic attack a bonus of +2 credits will be awarded.

If the winning team is composed of more than one student it will get a +10 bonus, to be shared among all the team members, on their final grade. The second +8, the third +6, and all the other participating teams +3 marks. If the winning team is composed of a single student the bonus will be set to +5, +3, or +2 depending if the student finishes first, second, or third, respectively. The participation bonus is worth +1 mark.

The final score is calculated as follows:

- The initial score is 0 for all the team.
- When a team recovers a plaintext during phase 2 it gets twice its corresponding credits.
- When a team recovers a plaintext during phase 3 it gets its corresponding credits.
- The team who created the broken ciphertext loses a number of credits equal to the credits awarded to the other teams.
- At the end, all the scores are calculated and the teams are ordered in increasing order with respect to their final score. In case of a tie, groups are ordered with respect to the number of plaintext they recovered and if this is still a tie the number of their plaintext recovered by other teams is considered.

Example: team A wrote 3 implementations, a_1, a_2 and a_3, worth 1, 3 and 6 credits respectively. Team B wrote 5, b_i, $1 \leq i \leq 5$, worth $\frac{1}{2}, \frac{1}{2}, 1, 3$ and 5, respectively. Team C wrote 2 worth 5 credits each. Assume team A breaks b_1, b_2 and b_5 during phase 3, team B breaks a_1 and a_3 during phase 3, and team C breaks b_5 during phase 2 and a_2 during phase 3. The final scores are then:

- Team A: $\frac{1}{2} + \frac{1}{2} + 5 - 1 - 6 - 3 = -4$
- Team B: $1 + 6 - \frac{1}{2} - \frac{1}{2} - 5 \times 2 - 5 = -9$
- Team C: $5 \times 2 + 6 = 16$

Team C wins. Although this is not the case here, if A and B were tie then team A would still be in front of team B as it recovered one more plaintext than team B.

3 Important notes

Please read carefully those final comments.

- Cheating by providing other teams with an incomplete code (e.g. removing a function) or a code that does not compile will result in being removed from the challenge competition
- It is not allowed to obscure the code by adding useless computation or express simple things in a complex manner
- Following Kerckhoff's principle everything must be known from the attacking teams, except the secret key
- If a team does not follow Kerckhoff's principle, **each missing information** will result in a **-2 penalty** on its final score for the challenge
- If the two chosen schemes have already been implemented in the course no reward will be awarded, not even the participation bonus
- Feel free to contact other groups if you have questions
- In case of conflict please contact the teaching team as early as possible