Homework 5 - Commitment, Integrity, Authentication and Signatures

Cryptography and Security 2017

- You are free to use any programming language you want, although SAGE is recommended.
- Put all your answers **and only your answers** in the provided SCIPER-answers.txt file. This means you need to provide us with Q_1, Q_2, Q_3, Q_4, Q_5. You can download your **personal** files on `http://lasec.epfl.ch/courses/cs17/hw5/index.php`
- The answers Q_1,Q_4,Q_5 should be ASCII strings, and Q_2,Q_3 should be integers. **Please provide nothing else. This means, we don't want any comment and any strange character or any new line** in the answers.txt file.
- We also ask you to submit your **source code**. This file can of course be of any readable format and we encourage you to comment your code.
- The plaintexts of most of the exercises contain some random words. Don't be offended by them and Google them at your own risk. Note that they might be really strange.
- If you worked with some other people, please list all the names in your answer file. We remind you that you have to submit your own source code and solution.
- We might announce some typos in this homework on Moodle in the "news" forum. Everybody is subscribed to it and does receive an email as well. If you decided to ignore Moodle emails we recommend that you check the forum regularly.
- For Exercises 1, 4 and 5 you can either use **the php pages we provide for you**, or if you want to automatize your queries, the best option is then to use **the Sage code we provide you**. To connect to the servers, **you have to be inside the EPFL network** (for the php page you don't need to be inside the EPFL network). Use VPN if you are connecting from outside EPFL. **We recommend you to make the network queries directly from your script for exercises 4 and 5.** You will need to several queries and their format is not human-friendly.

  Alternatively, if you want to do it by hand, use *ncat* (you can download a Windows version on `http://nmap.org/download.html` along with nmap). To send a query QUERY to the server lasecpc28.epfl.ch under port PORT write to the command line

  ```
  echo QUERY | ncat lasecpc28.epfl.ch PORT
  ```

  under Windows from a cmd in the same directory as the ncat.exe program

```
echo QUERY | ncat.exe lasecpc28.epfl.ch PORT
```

under MAC OS and some linux distributions, ncat doesn't work as it should and you need to replace it by

```
echo QUERY | nc -i 4 lasecpc28.epfl.ch PORT
```

Note that under Windows, you will also have an additional output "close: no error" that you can ignore. The "|" is obtained (for a Swiss keyboard) using altGr + 7.

Under Windows, using Putty is also an option. Select "raw" for "connection type" and select "never" for "close windows on exit".

- Please contact us as soon as possible if a server is down.
- Denial of service attacks are **not** part of the homework and will be logged (and penalized).
- The homework is due on Moodle on **Thursday the 21st of December** at 22h00.

Exercise 1 A weak MAC

Intrigued by the subtleties of the symmetric-key message authentication, our apprentice decided to create his own version of CBCMAC. He saw that the plain CBCMAC, which is the simplest of the CBC-based MACs, is only secure if all processed messages are of the same length, which must also be a multiple of n, with n the block length of the used blockcipher. In the section about Merkle-Damgård he saw that a good padding scheme, which prevents collision of padded messages, could solve this problem. He put these two observations together and designed his own scheme. This scheme pads every message to the length of $2^{10} \cdot n$ bits, and then applies plain CBCMAC to the result, using AES as the blockcipher. The apprentice called his scheme "CBCMAC for Rapid Authentication with Padding" (CRAP).

The apprentice thought long and hard about the padding scheme for CRAP. He remembered that good cryptographic hash functions can produce constant-length fingerprints of arbitrary-length messages which do not easily collide. So he decided to append a message M with zeroes and with its hash under $\mathsf{SHA256}$. He did not like that the hash took 256 bits however, so he decided to shorten it to 12 bits (that must be enough, right?). The final version of the padding scheme works as follows:

$$\mathsf{pad}(M) = 0^{2^{17}-|M|-12} \| \mathsf{left}_{12}(\mathsf{SHA256}(M))$$

where $\mathsf{left}_{12}(X)$ denotes truncation of the string X to its 12 most significant bits. The apprentice decided that he will only work with messages whose length is a multiple of 8, and thus CRAP can authenticate messages of 8 to $(2^{17} - 16)$ bits. The computation of the tag is done as follows:

Algorithm CRAP(K, M)
  $\bar{M} \leftarrow M \| \mathsf{pad}(M)$
  return CBCMAC[AES]$(K, \bar{M})$
end Algorithm

As before, the crypto apprentice uses the following conversion between ASCII strings and binary strings: a binary representation of a string of characters is obtained by encoding the ASCII value of every character as an 8-bit binary string, and then concatenating these 8-bit strings. For example, "Red Fox!" would be encoded as `01010010 01100101 01100100 00100000 01000110 01101111 01111000 00100001` (we included spaces just for clarity!). As this representation is not very efficient, the apprentice uses the base64 encoding when storing and sending binary strings that may contain unprintable characters, such as ciphertexts. For example, the binary string `01010010 01100101 01100100 00100000 01000110 01101111 01111000 00100001` would be encoded as `UmVkIEZveCE=`.

Feeling that he did well, the apprentice challenged you to mount a chosen message forgery for a specific message M_1 he chose for you. For this purpose, the apprentice implemented his scheme, and ran a network oracle that listens for the base64-encoding of any message M, such that $M \neq M_1$, and replies with a $\mathsf{base64}(\mathrm{CRAP}(K, M))$ using a secret key K. As usual, you think you see the way to your victory.

You can access this oracle either through the PHP interface `http://lasec.epfl.ch/courses/cs17/hw5/query1.php` where you have to provide the SCIPER and the base64 encoding of the message you want to MAC.

Another alternative is to connect to the server lasecpc28.epfl.ch on port 5555 using Sage or ncat (see instructions). The query should have the following format: SCIPER followed by the base64 encoding of the message you want to MAC

```
123456 UmVkIEZveCE=\n
```

will return the base64-encoding of the MAC of the ASCII string "Red Fox!".

In your parameter file, you will find the message M_1 in ASCII. Find the correct MAC $T_1 = \mathrm{CRAP}(K, M_1)$ and write its **base64 encoding** $Q_1 = \mathsf{base64}(T_1)$ in your answer file.

Exercise 2 A backdoorless Pedersen commitment

Our crypto-apprentice read about the Pedersen commitment. He liked it, but found the way the element h is picked in the setup phase disturbing. Indeed, this may leave room for an evil security agency to tamper with standardised instances of Pedersen commitment. He saw a way to solve this. In his own version of Pedersen commitment, Pedersen2, the apprentice replaced h by a pseudo-random function of g^x.

The commitment scheme Pedersen2 works as follows:

Setup: Generate two primes p and q such that q has 160 bits, p has 1024 bits and $q \mid (p-1)$. Then find an element $g \in \mathbb{Z}_p^*$ of order q. The domain parameters are (p, q, g).

Commit: To compute $\mathsf{Commit}_{p,q,g}(x; r) = (c, k)$, we first set $G = g^x$. Then we compute $a = \mathsf{SHA256}(\langle G \rangle_{1024})$ and $h = g^{\mathsf{int}(a)}$. Finally we compute $c = G \cdot h^r$ and set $k = (x, r)$.

Open: To compute $\mathsf{Open}_{p,q,g}(c, (x, r))$, we reconstruct the computation of the commitment. I.e. we compute $a = \mathsf{SHA256}(\langle G \rangle_{1024})$ and $h = g^{\mathsf{int}(a)}$. If $c = G \cdot h^r$, we output x, otherwise we output $\perp$.

In these descriptions, $\langle G \rangle_n$ denotes interpreting an element of $\mathbb{Z}_p^*$ as an integer, and encoding this integer in 1024 bits with big endian (I.e. $\langle 7 \rangle_{16}$ = `00000000 00000111`), and $\mathsf{int}(X)$ denotes interpreting a binary string X as an integer.

To use Pedersen2 to commit to a short ASCII string M, the apprentice used the encoding $x = \mathsf{ascii2int}(M)$ that encodes a string $M = M_1 M_2 \ldots M_r$ of r ASCII characters as

$\mathsf{ascii2int}(M) = \sum_{i=0}^{r-1} \mathtt{ASCII}(M_{i+1}) \cdot 2^{8 \cdot i}$, where $\mathtt{ASCII}(c)$ is the ASCII value of a character c.[1] For example, the encoding of the string "Red Fox!" would be 2411799947438744914.

In this exercise, we ask you to break the binding property of this commitment scheme. In your parameter file, you will find integers p_2, q_2, g_2 that define an instance of Pedersen2, an ASCII string M_{21}, an element $r_{21} \in \mathbb{Z}_q^*$, the commitment $c_{21} = \mathsf{Commit}_{p_2,q_2,g_2}(\mathsf{ascii2int}(M_2), r_{21})$ and another ASCII string M_{22}. Your goal is to find the element $Q_2 \in \mathbb{Z}_q^*$, such that $\mathsf{Open}_{p_2,q_2,g_2}(c_{21}, (\mathsf{ascii2int}(M_{22}), Q_2)) = \mathsf{ascii2int}(M_{22})$.

Exercise 3 DSA with Bad Randomness

Our crypto-apprentice learned DSA signature and decided to use this because it has a lot of benefits comparing to other signature schemes seen in the class and it is provable secure.

He first started to generate the public parameters: He picked a 160-bit prime number q_3, and a large prime number $p_3 = a_3 q_3 + 1$. He then iteratively picked $h \in \mathbb{Z}_p^*$ and raised to the power a_3, $g_3 \leftarrow h_3^{a_3} \mod p_3$ until $g_3 \neq 1$. Then, he picked its secret keys $x_3 \in \mathbb{Z}_{q_3}$. In the end, he published his public key $y_3 = g_3^{x_3}$ with the public parameters (p_3, q_3, g_3).

He generates the signatures as follows: If he generates a signature for the first time, he picks $m_3, n_3, k_3 \in \mathbb{Z}_{q_3}^*$ and stores m_3, n_3, k_3. Otherwise, he computes $k_3 \leftarrow m_3 k_3 + n_3 \mod q_3$ and stores k_3. Then, he computes $r_3 \leftarrow (g_3^{k_3} \mod p_3) \mod q_3$ and $s_3 \leftarrow \frac{H(M_3) + x_3 r_3}{k_3} \mod q_3$. The signature is (r_3, s_3). Here, M_3 is the message to be signed and H is a function which hashes M_3 with $\mathsf{SHA256}$ and then encodes the output of hash with $\mathsf{ascii2int}$ (as defined in Exercise 2) (i.e., $H = \mathsf{ascii2int}(\mathsf{SHA256}(M_3))$).

Our crypto-apprentice thought that even if m_3 and n_3 are public, the signature scheme will be secure because each new k_3 is still random. Show that he is wrong. In your parameter file, you have the public parameters (p_3, q_3, g_3), the public key y_3, the randomness generation parameters m_3, n_3, two consecutively signed messages M_{31} and M_{32} and their corresponding signatures (r_{31}, s_{31}) and (r_{32}, s_{32}), respectively. Find the secret key x_3 and write it next to Q_3 in your answer file.

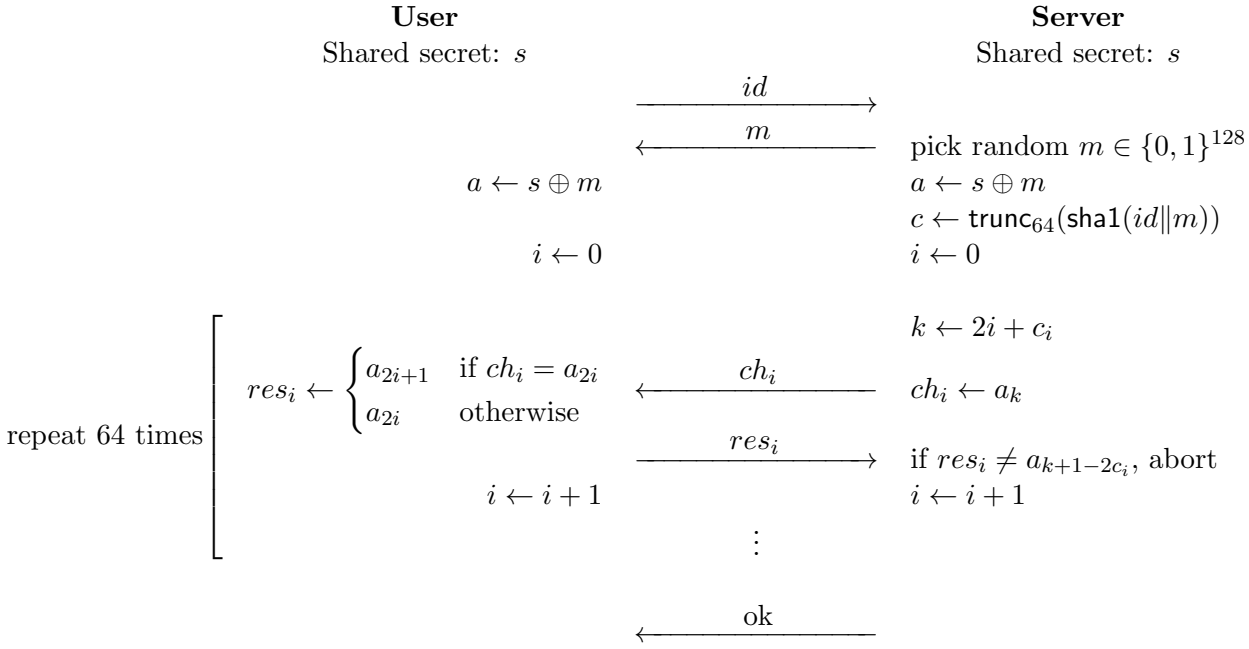
Exercise 4 Password-based Challenge-Response Login

A server that provides a secret service (in fact, it is streaming videos with meerkats) is using an interactive challenge-response protocol for user authentication. More precisely, when a new user signs up with a username id, the server and the user generate a shared secret $s = \mathsf{AES}(w \| 0^{128-|w|}, 0)$ from a password w which is chosen by the user upon enrolment. Here $\mathsf{AES}(K, m)$ is the encryption of a message m by using the AES with a key K, $|w|$ is the length of w, and 0^n is a string of n zero bits.

Once the user is registered, the user can authenticate itself to the server with its password w by following the protocol in Figure 1.

In the protocol, the user first computes $s = \mathsf{AES}(w \| 0^{128-|w|}, 0)$. Then, the user sends its username id to the server. The server picks a random string m of 128 bits, computes $a = s \oplus m$ and sends m to the user, who computes a as well. The challenge-response part starts after that, and consists of 64 rounds. The user needs 2 consecutive bits of a for each round. More precisely, at the i-th round (starting with round 0) for $i \in [0, 63]$, the user requires a_{2i} and a_{2i+1} where $a = a_0 \| a_1 \| \cdots \| a_{126} \| a_{127}$. The server deterministically selects a_{2i} or a_{2i+1} and sends it as a challenge and then the user needs to send the other bit of the two, which is

[1] Since ASCII characters can be represented by 8-bit binary strings, this encoding is injective and can be inverted.



*$\mathsf{trunc}_n(x)$ returns n most significant bits of x.

Figure 1: The challenge-response protocol for user authentication

not selected by the server, as a response. In other words, if the the challenge is a_{2i} (resp. a_{2i+1}), the the response should be a_{2i+1} (resp. a_{2i}). If the response is not correct, the server aborts the connection. If the connection is not aborted after 64 rounds, the server accepts the connection of the user.

The crypto-apprentice is one of clients of the secret meerkat-streaming service, and you would like to log into his account. You have access to a sequentialized version of login interface to the secret meerkat-streaming service from http://lasec.epfl.ch/courses/cs17/hw5/query2.php where you can execute the user authentication and see the messages from the server. In order to start the protocol, you should firstly send the username (the value in the response textbox will be ignored). Then, you will get the value m which is encoded in base64 and ch_0 from the server. When you receive the challenge ch_i from the server, you should respond with a response bit $res_i \in \{0,1\}$. If your response is correct, the server will send you ch_{i+1} (or ok when $i = 63$). If your response is incorrect, the server will send you an abort message and the protocol will be ended. You can also restart the protocol by clicking the reset button.

Alternately, you can connect to the server lasecpc28.epfl.ch on port 6666 using Sage or ncat to send queries directly. There are two different type of queries: one to initialize the protocol and one to send a response. **In the tutorial, we provide code for more efficient network communication with the server. We recommend using it to solve the exercise.**

In order to initialize the protocol, you need to send a a zero character followed by your username. This query should have the following format:

```
0 username\n
```

Then, you will get a reply of the following format:

`base64(m) a session\n`

where m is the random string m, $a \in \{0,1\}$ is the first challenge ch_0 and *session* is the session state that you should send with your response.

When you want to send a response to a server, you need to use a query of following format:

`1 username response session\n`

where *session* is the session state that you received with current challenge. Then, you will get a response of the following format:

`b session'\n`

where b is either new challenge, or an abort message, or an "ok" message, and $session'$ is the new session state for next response (If next response is needed). If $b \in \{0,1\}$, your response was correct and b is next challenge bit. When your response was wrong, you will only get $b = 2$ without any $session'$. When the protocol is correctly done, i.e. your response was res_{63} and it was correct, you will get $b = 3$ without any $session'$.

Here is an example of communication. We want to send a query with 123456 as username. We send

`0 123456\n`

and we receive back

`AAAAAAAAAAAAAAAAAAAAAA== 1 KNYcglnuIw3oNKCUang6\n`

where $m = 0^{128}$. Then, ch_0 is 1 and *session* is KNYcglnuIw3oNKCUang6. Now, we send $res_0 = 1$. Then, our message is

`1 123456 1 KNYcglnuIw3oNKCUang6\n`

and we receive back

`1 Ey9uf0k1Me9LGyLdfHEy\n`

So, the value of res_0 was correct and ch_1 is 1. Now, we send $res_1 = 0$. Then, our message is

`1 123456 0 Ey9uf0k1Me9LGyLdfHEy\n`

and we get back

`2\n`

So, the value of res_1 was incorrect and the protocol is failed.

You know that the username of the apprentice is first 6 characters of your sciper and that his password is somewhere in the `passwords.txt` file that can be found on Moodle.

Recover the password of the apprentice w, and write it under Q_4 in your answer file.
Hint: An exhaustive search can be useful at some point but not at the beginning.

Exercise 5 Integrity with Merkle-Hash Tree

After taking cryptography and security course, our crypto-apprentice started to write a very huge book called "A Collection of Bad Passwords". He is extremely cautious about his book. Therefore, he put his book in five different servers. He is also worrying about integrity protection of his book in the servers. He did his research about it and he learned that Merkle-hash tree is one of the best ways to check the integrity of large files. Since his book is very long, he decided to store also the Merkle-hash tree of the book in each server to check the integrity later.

In general, the Merkle hash tree is a binary tree in which leaf nodes are labelled with blocks of a padded message and all the non-leaf nodes are labelled with the hash of their children. For example, if we compute the hash of a padded file $f = f_0||f_1||...||f_7$ (see figure), we have that $hf_{10} = H(f_0||f_1)$ and $hf_{21} = H(hf_{12}||hf_{13})$, where H is the used hash function. For the sake of simplicity this exercise we are going to

- treat only messages whose length (in blocks) ℓ is a power of 2,
- assume all messages have the same length (and thus the same number of leaves on the same level),
- use no padding.

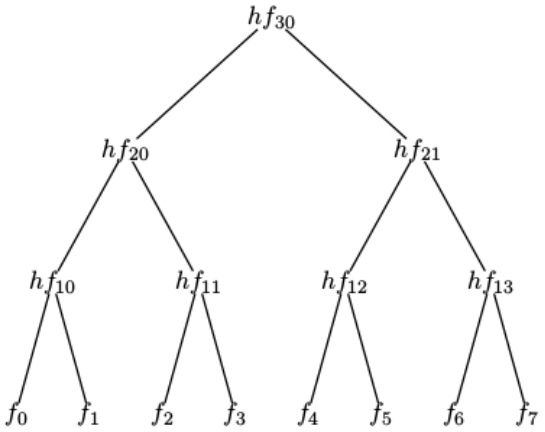


Figure 2:

We use the following convention: the leaves will be on the level 0 and the root, the final hash, will be on the level $\lceil \log_2(\ell) \rceil$. The internal nodes are indexed by their level and their position on their level. We count the elements on the same level from left to right starting from 0. For example hf_{12} is the third node on level 1.

As an example, the integrity check is done as follows for the tree in the figure: The apprentice knows the root hf_{30} and $f = f_0||f_1||...||f_7$. It sends a file index set where each index in the set is from $\{0, 1, ..., 7\}$ to the server. Then, the server replies with the smallest set of nodes which allow him to recompute the root of the tree. Assume that the index set is $\{2, 6, 7\}$. Then, the server replies with the leaves he asked f'_2, f'_6, f'_7 and $, hf'_{10}, hf'_{12}$ (the notation f' is used to emphasize what comes from the server). **The algorithm minimumTree that finds the minimal set of nodes is given in the tutorial.** The apprentice checks the integrity by first computing hf'_{11} by using f'_2, f_3 (he has to use f_3 from his copy of the

file) and computing hf_{13} by using f'_6 and f_7 (he has f_7). Second, he computes hf'_{20} by using hf'_{10}, hf'_{11} and computes hf'_{21} by using hf'_{12} and hf'_{13}. Finally, he computes the root hf'_{30} by using hf'_{20} and hf'_{21} and compares with his own value for tree's root hf_{30}. If they are not equal, it means that integrity is broken. Note that the efficiency comes from the fact that the apprentice can check integrity without checking whole file in the server and without receiving whole file from the server. He only receives a part of the file (e.g., f_2, f_6, f_7).

To hash it with a Merkle-hash tree, the apprentice divides his book into blocks of 32 bytes and uses each block f_i as a leaf. He uses SHA256 as a hash function. **The exact algorithm that he used to compute a hash tree is given in the tutorial.**

He recently heard about the attacks on servers in the campus. Therefore, he wants to regularly check if any change is done by an attacker to the copies of his book stored on the servers. The apprentice trusts you and so he gives this mission to you. However, you can query at most **ONE** file index to a server because the apprentice wants to keep the network communication overhead low. One of the blocks of his book (f_i with i unknown) has been changed in one of the servers (with unknown server index j) and you need to find both the block index i and the server index j of the changed block by making the integrity-check queries, sending **at most one file index** in each of your request. You can access the servers either on the address

`http://lasec.epfl.ch/courses/cs17/hw5/merkle.php`, where you will find a web interface that will take your SCIPER, the server index $j \in \{0, 1, 2, 3, 4\}$ and file index $[i]$ (you must enclose it in square brackets!). If you want to query with no file index, then you input [] in to file index box It will return a list of the nodes with the following format: [l-k is hf_{lk}, ..., m-n is hf_{mn}] where each hf is in base64.

Alternatively, you can connect to the server lasecpc28.epfl.ch on port 7777 using Sage or ncat to issue queries directly. The query should have the format `sciper server-index [file-index]`, e.g

```
12345 j [i]\n
```

or if you want to request nodes with no file index:

```
12345 j []\n
```

You don't have parameters for this exercise. You just need to use the file `password_DB.txt` provided in Moodle, which corresponds to the book of our apprentice.

Find f_i which has been changed in server j and write $Q_5 = j : i$ (server index:file index, including the colon) in your answer file.

Hint: The file is quite big and network slow. Doing a simple "exhaustive search" will not do the trick.