Homework 5 - Integrity, Authentication, Signatures and the End of the Adventures

Cryptography and Security 2016

- 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_{5a} and Q_{5b}. You can download your **personal** files on `http://lasec.epfl.ch/courses/cs16/hw5/index.php`
- The answers Q_2, Q_{5a}, Q_{5b} should be ASCII strings, Q_1 should be a point on an elliptic curve, i.e. a pair of integers, Q_3 should be a hexa string, and Q_4 should be a list of hexa strings. **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 Exercise 4, we recommend you to use **the php page we provide for you** on `http://lasec.epfl.ch/courses/cs16/hw5/query.php`. 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.

  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 **Friday the 23th of December** at 20h00 (Merry Christmas!).

Exercise 1 An Elliptic Curve Commitment

Our crypto-apprentice read about the Pedersen commitment and realized that its binding property comes from the hardness of discrete logarithm (DL) problem. Since DL is hard when you use a proper elliptic curve, he decides to use Pedersen commitment on an elliptic curve with some changes on the commit algorithm and the open algorithm. The elliptic curve commitment works as follows:

- Setup: generates a prime p_1 to work over a field $\mathbf{Z}_{p_1}$ and parameters a_1 and b_1 to define the curve $E_{a_1,b_1} : \{\mathcal{O}\} \cup \{(x, y))\ s.t.\ y^2 = x^3 + a_1x + b_1\}$ and picks two elements P_1 and G_1 from the curve, both of which have prime order n_1 and generate the same subgroup of the curve. The domain parameters are $(p_1, a_1, b_1, n_1, P_1, G_1)$.
- Commit: $(C, (m, R)) = \texttt{Commit}(m, r)$, where m is an ASCII string, $r \in \mathbb{Z}_{n_1}$, $C = H(m) \cdot P_1 + r \cdot G_1$, $R = r \cdot G_1$, and $H : \{0, 1\}^* \to \mathbb{Z}_{n_1}$.
- Open: $o = \texttt{Open}(C, (m, R))$ where $o = m$ if $H(m) \cdot P_1 + R = C$ and $o = \texttt{invalid}$ otherwise.

He is too obsessed about the secrecy of his committed message. Therefore, he uses as H a hash function SHA256 by reducing the output of SHA256 to modulo n_1 (i.e., $H(m) = \text{int}(\text{SHA256}(m)) \bmod n_1$) while committing a message m.

In this exercise, we ask you to break the binding property of this commitment scheme. In detail, given C_{11} from $(C_{11}, (m_{11}, R_{11})) = \texttt{Commit}(m_{11}, r_{11})$, you should provide a valid R_{12} such that $m_{12} = \texttt{Open}(C_{11}, (m_{12}, R_{12}))$ for a message m_{12}. You will find in your parameter file the domain parameters $(p_1, a_1, b_1, n_1, P_1, G_1)$ and the values C_{11}, m_{11} and m_{12}. Find R_{12} and write it under Q_1 (i.e. we expect a point on an elliptic curve with no compression).

Exercise 2 Merkle-Damgård Variant

Inspired by what he saw in the Integrity and Authentication chapter, while trying to learn from his mistakes, our apprentice decided to play with the Merkle-Damgård scheme. He has studied this scheme and saw that in order to use it he had to use a compression function C. He did not know too much about compression functions but he remembered that a blockcipher can be used to construct a good compression function with a single xor. The apprentice decided to use AES and constructed his compression function

$$F_{\mathsf{AES}}(X, H) = \mathsf{AES}_{0^{128}}(X \oplus H)$$

, where both the message input X and the chaining value H are strings of 128 bits, and AES is keyed with a string of 128 zeroes. He then decided to hash an English message with his own variant of Merkle-Damgård.

More precisely, our apprentice designed the following algorithm:

Algorithm MKV(M, IV)
  $M_0, M_1, \ldots, M_{m-1} \xleftarrow{128} M \| \mathsf{pad}(M)$
  $C_0 \leftarrow IV$
  for $i = 0$ **to** $m - 1$ **do**
    $C_{i+1} \leftarrow F_{\mathsf{AES}}(M_i, C_i)$
  end for
  $H \leftarrow C_m$
  return H
end Algorithm

In order to hash a message M we first pad it. The padding occupies from 8 to 129 bits and it's of the format $1\|0^{(120-|M|) \bmod 128}\|\mathsf{lengthbytes}_7(M)$ i.e. a bit 1 concatenated with a certain number of zeroes, and with $\mathsf{lengthbytes}_7(M)$, the 7-bit representation of the number of bytes of M. The number of zeroes we use is the smallest, such that bitlength of $M\|\mathsf{pad}(M)$ is a multiple of 128. For example for M = 11001100 11111111 11001100 11111110 we get that $\mathsf{pad}(M)$ = 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100 (as M has 4 bytes) and $M\|\mathsf{pad}(M)$ = 11001100 11111111 11001100 11111110 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100.

After the padding the whole message is split into blocks M_i of 128 bits. The initial value, IV, is a constant and has 128 bits. For each block M_i, we xor the chaining value C_i with M_i to form the input for AES. The key K is fixed to be 0^{128}, i.e. it contains no bit of 1. The output of the MKV algorithm is the encryption of the xor of the last message block and the corresponding chaining value.

To be able to process ASCII strings with AES (which works with binary strings), the crypto-apprentice encodes the ASCII value of every character as an 8-bit binary string, and then concatenates 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 crypto apprentice uses hexadecimal encoding when storing and sending binary strings, such as ciphertexts. For example, the binary string `01010010 01100101 01100100 00100000 01000110 01101111 01111000 00100001` would be encoded as `52 65 64 20 46 6F 78 21`.

The apprentice hashed a sentence Q_2 with IV_2 and obtained a hash H_2. You know that Q_2 starts with a prefix P_2, continues with a secret password pwd_2 and ends with a suffix S_2, so $Q_2 = P_2\|pwd_2\|S_2$. The password pwd_2 consists of 12 ASCII characters from the set

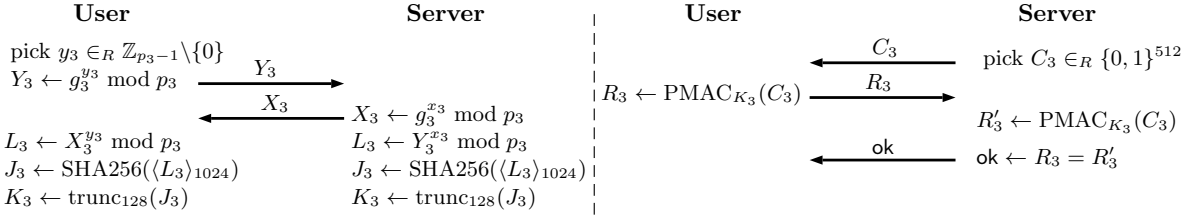

Figure 1: The challenge-response protocol for user authentication: user sign-up (left) and user login (right). We assume that the communication channel is authenticated. $\langle L \rangle_{1024}$ denotes encoding of an integer $L < 2^{1024}$ as a string of 1024 bits, $\mathrm{trunc}_{128}(J)$ denotes truncation of a string J to the first 128 bits, and g_3 is a generator of $\mathbb{Z}_{p_3}^*$.

{'0', '1', ..., '9', 'a', ..., 'z', 'A', ..., 'Z'} . In your parameter file you are given IV_2 and H_2 (in hexadecimal), and P_2 and S_2 (in plain ASCII). Invert the hash function MKV and recover the original phrase Q_2.

Exercise 3 Challenge-Response Login

A server that provides a certain service (in fact, it is streaming videos with cats) is using a PRF-based challenge-response protocol for user authentication. More precisely, when a new user signs up, he/she does a Diffie-Hellman key exchange over $\mathbb{Z}_{p_3}^*$ with the server to obtain a long-term secret K_3. The server uses the same secret value x_3 for the DH key exchange made with every new user. For every subsequent login, the user establishes an authenticated channel with the sever (in this exercise, we just assume this works well) and the server sends a 512-bit random challenge C to the user. The user then computes a 128-bit response $R = \mathrm{PRF}_{K_3}(C)$ and sends it to the server. The server then verifies the received response with a stored copy of the key K_3. The protocol is depicted in Figure 1.

The PRF used in this protocol is (a very slightly simplified version of) the blockcipher mode of operation "Parallelizable MAC" (PMAC) instantiated with the AES blockcipher. PMAC internally uses $\mathrm{GF}(2^{128})$ operations, with the reference polynomial used to instantiate the finite field being $P(\mathsf{x}) = \mathsf{x}^{128} + \mathsf{x}^7 + \mathsf{x}^2 + \mathsf{x} + 1$. To be able to use the $\mathrm{GF}(2^{128})$ operations with strings of 128 bits, we treat these strings as representations of polynomials with binary coefficients and degree smaller than 128. E.g. the string `100...0101` represents the polynomial $Q(\mathsf{x}) = \mathsf{x}^{127} + \mathsf{x}^2 + 1$.

We will need to multiply the 128-bit strings by the elements x and by x^{-1}. For multiplication of a binary string s by x, we notice that it corresponds to shifting the string s left by one position, and xoring the 128-bit string `0...010000111` to the result of the shift if the string s had it's first bit equal to 1. More precisely

$$s \cdot \mathsf{x} = \begin{cases} s \ll 1 & \text{if firstbit}(s) = 0 \\ (s \ll 1) \oplus 0^{120}\texttt{10000111} & \text{if firstbit}(s) = 1 \end{cases}$$

where 0^{120} denotes the binary string of 120 zeroes. When s is stored as a string of 16 ASCII characters (i.e. 8-bit integers), the shift should be implemented as

$$s'[i] \leftarrow ((s[i] \ll 1) \text{ AND } \texttt{0xff}) \text{ OR } ((s[i{+}1] \text{ AND } \texttt{0x80}) \gg 7) \text{ for } 0 \le i < 15,\ \ s'[15] \leftarrow s[15] \ll 1$$

where the binary operations are evaluated bitwise. Also, the function firstbit(s) returns $((s[0] \text{ AND } \texttt{0x80}) \gg 7)$. The multiplication of a string s by x^{-1} can be computed in a similar

Algorithm PRECOMPUTE(K)
  $L \leftarrow E_K(0^{128})$
  $L_{-1} \leftarrow L \cdot \mathsf{x}^{-1}$
  $L_0 \leftarrow L$
  for $i = 1$ **to** $\lceil \log_2(m_{\max}) \rceil$ **do**
    $L_i \leftarrow L_{i-1} \cdot \mathsf{x}$
  end for
end Algorithm

Algorithm PMAC(K, M)
  Split M into $M_1 M_2 \dots M_m$
  $\Delta \leftarrow 0^{128}$
  $\Sigma \leftarrow 0^{128}$
  for $i = 1$ **to** $m - 1$ **do**
    $\Delta \leftarrow \Delta \oplus L_{\mathrm{ntz}(i)}$
    $X_i \leftarrow M_i \oplus \Delta$
    $Y_i \leftarrow E_K(X_i)$
    $\Sigma \leftarrow \Sigma \oplus Y_i$
  end for
  $\Sigma \leftarrow \Sigma \oplus \mathrm{pad}(M_m)$
  if $|M_m| = 128$ **then**
    $X_m \leftarrow \Sigma \oplus L_{-1}$
  else
    $X_m \leftarrow \Sigma$
  end if
  return $E_K(X_m)$
end Algorithm

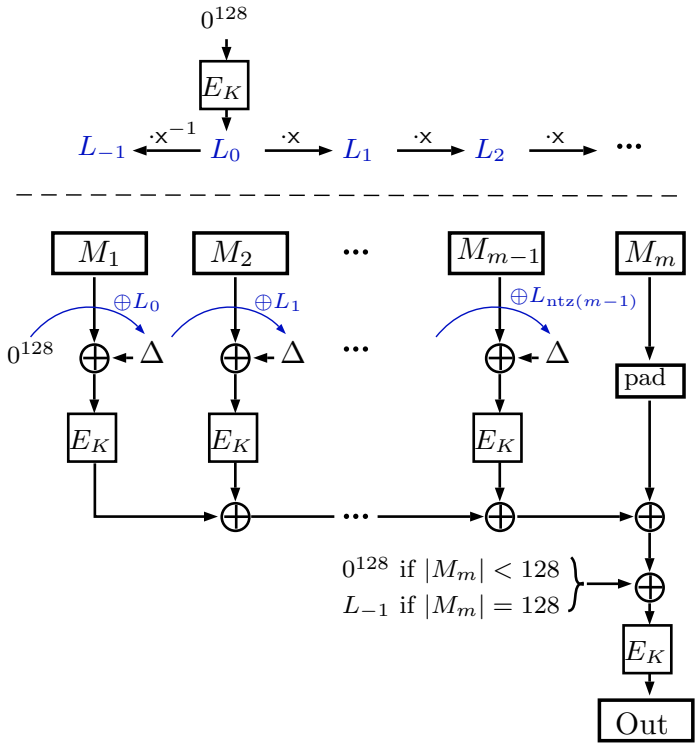

Figure 2: The PMAC mode. Here $E_K = \mathrm{AES}_K$, $|X|$ denotes the length of a binary string X, and $\mathrm{ntz}(i)$ denotes the number of trailing zeroes in the binary representation of an integer i (e.g. $\mathrm{ntz}(5) = 0$ and $\mathrm{ntz}(4) = 2$). "Split M into $M_1 M_2 \dots M_m$" denotes partitioning of a binary string M such that $M_1 \| M_2 \| \dots \| M_m = M$, $|M_i| = 128$ for $1 \le i < m$ and $1 \le |M_m| \le 128$, and $m_{\max}$ denotes the number of 128-bit blocks obtained by partitioning the longest message that can appear during the lifetime of the secret key. The padding $\mathrm{pad}(M_m)$ returns $M_m \| 1 \| 0^{128-|M_m|-1}$ if $|M_m| < 128$ and M_m if $|M_m| = 128$.

way as

$$s \cdot \mathsf{x}^{-1} = \begin{cases} s \gg 1 & \text{if lastbit}(s) = 0 \\ (s \gg 1) \oplus \mathtt{10^{120}1000011} & \text{if lastbit}(s) = 1. \end{cases}$$

If s is stored as a string of 16 ASCII characters, the function lastbit(s) returns (s[15] AND `0x01`). Algorithmic description and illustration of PMAC can be found in Figure 2. Test vectors for PMAC can be found at `http://web.cs.ucdavis.edu/~rogaway/ocb/pmac128.txt`.

The crypto-apprentice is using the cat-streaming service, and you would like to log into his account. Having planned this attack for some time, you have captured and saved the messages X_3, Y_3 of the DH key exchange that the apprentice did when he signed up. You also recorded a challenge C_{31} and apprentice's response R_{31} from some previous session. Recently, you managed to mount the Heartbleed attack on the cat-streaming server, and in the chunk of the contents of the server's RAM you managed to discover the static secret key x_3. Now you are finally logging into the apprentice's account and the server is sending you a challenge C_{32}.

In your parameter file, you will find the 1024-bit prime p_3, a generator $g_3 \in \mathbb{Z}^*_{p_3}$, DH messages X_3 and Y_3 (both integers), the static secret key x_3 (a big integer), the challenges C_{31}, C_{32} and the response R_{31} (all encoded in hexadecimal). Compute the correct response

$$X_4 = \begin{bmatrix} x_{a,1} & x_{a,2} & ... & x_{a,N} \\ x_{b,1} & ... & ... & x_{b,N} \\ : & : & : & : \\ x_{z,1} & ... & ... & x_{z,N} \\ x_{\_,1} & ... & ... & x_{\_,N} \\ x_{0,1} & ... & ... & x_{0,N} \\ : & : & : & : \\ x_{9,1} & ... & ... & x_{9,N} \end{bmatrix}$$

Table 1: Secret key

$$Y_4 = \begin{bmatrix} H'(x_{a,1}) & H'(x_{a,2}) & ... & H'(x_{a,N}) \\ H'(x_{b,1}) & ... & ... & H'(x_{b,N}) \\ : & : & : & : \\ H'(x_{z,1}) & ... & ... & H'(x_{z,N}) \\ H'(x_{\_,1}) & ... & ... & H'(x_{\_,N}) \\ H'(x_{0,1}) & ... & ... & H'(x_{0,N}) \\ : & : & : & : \\ H'(x_{9,1}) & ... & ... & H'(x_{9,N}) \end{bmatrix}$$

Table 2: Public key

to C_{32}, encode it in hexadecimal and place it under Q_3 in your answer file.

Exercise 4 Hash-based signature

Our crypto-apprentice had a bad experience with his bank. A malicious party ordered a transaction from his account without his permission. To prevent this, he decided to use a digital signature scheme in the conversations with his bank. He was afraid of doing one more mistake (as you know from his previous attempts) on the implementation of signature schemes that he learned from Cryptography and Security class. So, he searched for a different, simple and easily implementable scheme. Finally, he found hash-based signature which requires using only a hash function H.

The secret key and the public key are tables of size $37 \times N$ (See Table 1 and Table 2) in the hash based signature scheme. The secret key has as many columns as the size of the message that we sign; the signing is done character by character. Each column of the secret key represents the letters 'a','b',...'z' , space ␣and numbers $0, 1, 2, ..., 9$ in the corresponding order. This is the alphabet of the signed message. The values $x_{i,j}$ are random values of 256 bits. The public key Y_4 is the first 128 bits of the hash of each cell in the secret key X_4 ($Y_4[i][j] = H(x_{i,j})[:128] = H'(x_{i,j})$ where $[:128]$ means the first 128 bits).

Given the message $M = (m_1, m_2, ..., m_N)$ and the secret key X_4, the signing algorithm computes the signature $S = (s_1, s_2, ..., s_N) = (x_{m_1,1}, x_{m_2,2}, ...x_{m_N,N})$, where m_i is the i^{th} character of M. I.e. for a character char at position i, we look at the ith column in X_4 and see what value is on the char row.

The verification algorithm is using Y_4 and checks if $Y_4[m_i][i] = H'(s_i)$ given S and M.

In this exercise, we use SHA256 as H. We ask you to forge a signature of message M_4 which has 24 characters. You have an access to an oracle which will sign a message M of 24 characters that you submit with the key X_4, if M differs from M_4 on at least 12 positions. For example, the phrases "abab" and "babb" differ on 3 positions. You can access this oracle either through the PHP interface `http://lasec.epfl.ch/courses/cs16/hw4/query.php` where you have to provide the SCIPER and the message you want to sign. We recommend you to use this page for your queries.

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 message you want to sign. For instance

```
123456 my message
```

In your parameter file, you find the public key Y_4 and M_4. Find a valid signature of M_4 and write it next to Q_4.

Exercise 5 Quadruple encryption

The apprentice cryptographer studied 2-key 3DES design and the extra security it offers against key-recovery attacks. As usual, he saw some room for improvement. He designed his own scheme 2-key 4AES, which differs from 2-key 3DES in two aspects: (1) it uses AES instead of DES, and (2) it uses each of the two keys both directly and hashed. The exact definition of the scheme can be found in Figure 3.

1: **Algorithm** $4\text{AES}(K_1, K_2, P)$
2: $K_1' \leftarrow \text{trunc}_{128}(\text{SHA256}(K_1))$
3: $K_2' \leftarrow \text{trunc}_{128}(\text{SHA256}(K_2))$
4: $S \leftarrow \text{AES}(K_1', P)$
5: $S \leftarrow \text{AES}(K_1, S)$
6: $S \leftarrow \text{AES}(K_2, S)$
7: $C \leftarrow \text{AES}(K_2', S)$
8: **return** C
9: **end Algorithm**

Figure 3: The 2-key 4AES construction. The two keys K_1, K_2 have 128 bits each, and $\text{trunc}_{128}(X)$ denotes truncation of a binary string X to its first 128 bits.

The apprentice was so confident in his design that he decided to mock you with a challenge that he thought would be just too hard to be solved in the time you are given. He sampled two keys K_{51}, K_{52} independently with the same weak distribution. More precisely, for $i \in \{1, 2\}$ he constructed each of the keys as ASCII strings K_{5i} =“`aaaa bbbb cccc !`” where each of `aaaa, bbbb, cccc` is a 4-character word drawn independently randomly from the list of words KEY-WORDS.txt. With the keys constructed in this way, he encrypted 4 plaintexts $P_{51}, P_{52}, P_{53}, P_{54}$ and gave you both the plaintexts and the corresponding ciphertexts. The file KEY-WORDS.txt will be available on moodle.

In your parameter file, you will find the plaintexts $P_{51}, P_{52}, P_{53}, P_{54}$ (directly in ASCII!), the four ciphertexts $C_{51}, C_{52}, C_{53}, C_{54}$ (in hexadecimal). Recover the two keys K_{51} and K_{52}, and write them in you answer file under Q_{5a} (for K_{51}) and Q_{5b} (for K_{52}) **as ASCII strings** (we expect two strings of exactly 16 characters).