Home Assignment 4

Advanced Web Security

2018

B-assignments

For grade 3, complete the three B-assignments below and solve them in groups of ≤ 2 students.

B-1 SAML assertions, which are formatted using XML, are digitally signed by the IdP. For this, and other similar purposes, there is a recommendation by IETF and W3C on how to combine signatures with XML. This is called XML Signatures. Read about and understand XML Signatures. Make sure that you get an overview of the `<signature>` tag and its contents, as well as the difference between enveloped, enveloping and detached signatures.
Assessment:

- There will be two Moodle-questions on XML signatures. One is general with a focus on understanding the different parts of the `<signature>` tag and one is focused on the difference between enveloped, enveloping and detached signatures. Though you are not supposed to know the details, it can be a good idea to have the specification accessible when you answer the questions. You need to correctly answer both questions in order to pass. Note that the second question can have more than one correct alternative. Both students must finish the quiz.

B-2 The security department at LTH is using a home made web page for giving students their grade in the best course at LTH – Advanced Web Security. The web page uses HTTP GET requests to handle the grading. Every message sent to the server needs to be signed, using the HMAC-SHA1 algorithm, in order to be accepted. The URL is of the form
"`https://eitn41.eit.lth.se:3119/ha4/addgrade.php?name=Kalle&grade=5&signature=6823...ba36`".
Due to the web page being written by a hungover Ph.D. student, it is not as secure as one would expect. The server side code is given on the next page. The signature is calculated as $s = \text{HMAC}(\text{name}||\text{grade}, k)$, where $||$ denotes string concatenation and k is the secret key. The signature is truncated to 10 bytes (20 hexadecimal digits). The function `chrcmp` is not optimized for the current hardware, thus taking a considerable amount of time. Exploit the weakness in the web page to be able to send any message you like with a valid signature, without knowledge of the secret key. Propose a good way to counter these kind of attacks.

Your program should take the name and grade as parameters and output a valid signature as a hexadecimal string.

Assessment:

- There will be one Moodle question following the problem statement above. Both students must finish the Moodle quiz. There will be a test quiz on Moodle, where you can try your implementation as many times as you like. The test quiz will not be graded.
- There will be an oral exam for this assignment which you must do individually. You have exactly **five** minutes to show me that you have done and understood the assignment, so be **well** prepared. You shall be able to answer questions regarding the type of attack, time complexities, and hash functions. Sign up at `https://doodle.com/poll/rhb9wq6xn2apu8qd`. **Make sure to be on time.** The time schedule is tight.

```php
<?php
function calc_signature($data) {
  // truncate signature to 10 bytes
  $sig = substr(hash_hmac('sha1', $data, $key), 0, 20);
  return $sig;
}

function check_signature($sig1, $sig2) {
  $n = max(strlen($sig1), strlen($sig2));
  for ($i = 0; $i <= $n; $i++) {
    // chrcmp compares chars at index i, return 1 if chars are equal, 0 otherwise
    if (!chrcmp($sig1, $sig2, $i)) {
      return '0';
    }
  }
  return '1';
}

// get the user supplied data
if (isset($_GET['name']) && isset($_GET['grade']) && isset($_GET['signature'])) {
  $name = $_GET['name'];
  $grade = $_GET['grade'];
  $sig = $_GET['signature'];

  // concatenate name and grade
  $data = $name . $grade;
  $sig2 = calc_signature($data);

  echo check_signature($sig, $sig2);
} else {
  echo 'Not enough arguments';
}
?>
```

B-3 Using textbook RSA for encrypting messages is insecure for several reasons, one being that it is deterministic. There is a padding scheme known as *Optimal Asymmetric Encryption Padding*, or OAEP, to solve this problem. The OAEP padding scheme turns RSA into a probabilistic encryption scheme and adds the property of *all-or-nothing*, meaning that you either get the full message or nothing at all. Standard RSA may leak some information without revealing the complete secret.

The latest version of OAEP is given in RFC 8017. In this problem you will implement OAEP in two steps directly from the RFC specification. The first step is to implement the MGF1 function. This function takes an input string of arbitrary length and outputs a string of (almost) arbitrary length. The specification of MGF1 can be found in Appendix B.2.1 of RFC 8017 and you will also need to implement the I2OSP function in Section 4.1. In this assignment you will use the SHA-1 hash function as the hash function in MGF1.
This program will take two inputs, one `mgfSeed` and one `maskLen`, and output a string of `maskLen` bytes. As an example, the input

`mgfSeed` = `0123456789abcdef` (hexadecimal)
`maskLen` = `30` (decimal)

will output the (hexadecimal) string

`18a65e36189833d99e55a68dedda1cce13a494c947817d25dc80d9b4586a`

Note again that both the `mgfSeed` and the output string are here written in their hexadecimal notation so the length of the mgfSeed is 8 bytes and the length of the output is 30 bytes.

With a working MGF1 function, the next step is to implement the rest of the OAEP scheme. The scheme is given in Section 7.1 of RFC 8017. Note that encryption and decryption is not a part of the assignment, but you should implement both the encoding and the decoding part of OAEP. The parameter L will be chosen as `null` in the scheme.

Your implementation of `OAEP_encode` will take two parameters, the message M to be encoded and the seed, and output the encoded message EM; `OAEP_encode`$(M) = EM$. Your implementation of `OAEP_decode` will take one parameter, the message EM to be decoded, and output the decoded message M; `OAEP_decode`$(EM) = M$. You should take hexadecimal strings as input and also output hexadecimal strings. The length of the encoded message should be 128 bytes, which would be preparing the message to be encrypted with 1024-bit RSA.

You can use the following hexadecimal test strings to check your implementation:

```
M    =  fd5507e917ecbe833878
seed =  1e652ec152d0bfcd65190ffc604c0933d0423381
EM   =  0000255975c743f5f11ab5e450825d93b52a160aeef9d3778a18b7aa067f90b2
        178406fa1e1bf77f03f86629dd5607d11b9961707736c2d16e7c668b367890bc
        6ef1745396404ba7832b1cdfb0388ef601947fc0aff1fd2dcd279dabde9b10bf
        c51f40e13fb29ed5101dbcb044e6232e6371935c8347286db25c9ee20351ee82
```

Having implemented OAEP directly from the specification and understanding how OAEP works, you now know how RSA is really used in practice!
Assessment:

- Upload your code to Urkund, jonathan.sonnerup.lu@analys.urkund.se. Only one student per group must do this.
- There will be three Moodle questions following the problem statement above. One for MGF1, one for encoding and one for decoding. Both students must finish the Moodle quiz. There will be a test quiz on Moodle, where you can try your implementation as many times as you like on an example different from the one given above. The test quiz will not be graded.

C-Assignments

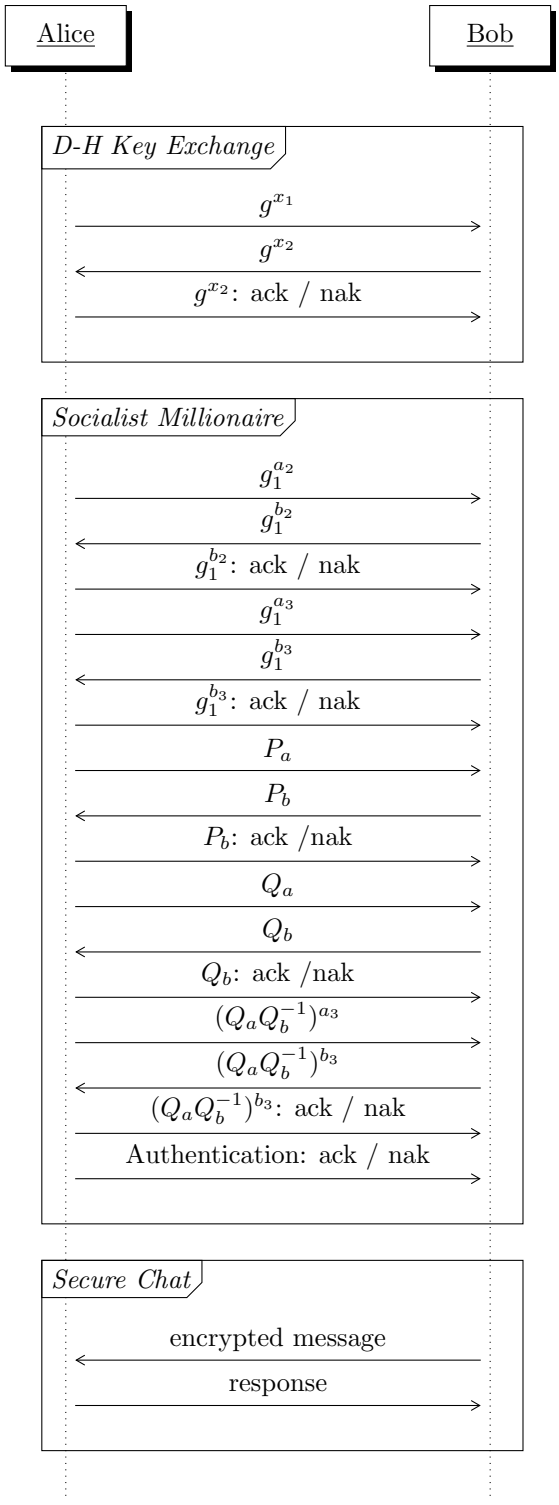
For grade 4, complete the C-assignment below and solve it in groups of ≤ 2 students.

C-1 Anytime during an OTR chat, one may run SMP for mutual authentication. Here, you will implement a simplified version of the OTR protocol, utilizing SMP, see protocol below. In the first part, you will need to negotiate a shared key using the Diffie-Hellman key exchange. Secondly, you will run the Socialist Millionaire Protocol, yielding mutual authentication. Lastly, you shall send an encrypted message to the server. The server then replies with a response for you to enter in Moodle.

All operations are to be calculated in the group $\mathbb{Z}_p^*$, i.e., mod p. The generators are $g = g_1 = 2$. The prime number p is defined (in the OTR documentation) as:

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

All integers sent to and from the server are encoded in hexadecimal string representation, e.g. if we want to send the decimal number 1337, we interpret it in hexadecimal, 0x539, and send the string "539". This can be achieved by the Python function `format(1337, 'x')` or the Java function `num.toString(16)`, where `num` is a `BigInteger` object with the value 1337. All randomly chosen numbers are assumed to be in $\mathbb{Z}_p^*$. The numbers sent from Bob will or will not be acknowledged by Alice. You will take the role as Bob, meaning that you have to read the ack/nak messages as well. In the last part of the protocol (Secure Chat), the encryption is a simple xor, using the shared D-H key as a One Time Pad (OTP), e.g., *enc_msg* = *msg* `xor` *D-H key*.


Alice
Bob
D-H Key Exchange
g^{x_1}
g^{x_2}
g^{x_2}: ack / nak
Socialist Millionaire
$g_1^{a_2}$
$g_1^{b_2}$
$g_1^{b_2}$: ack / nak
$g_1^{a_3}$
$g_1^{b_3}$
$g_1^{b_3}$: ack / nak
P_a
P_b
P_b: ack /nak
Q_a
Q_b
Q_b: ack /nak
$(Q_aQ_b^{-1})^{a_3}$
$(Q_aQ_b^{-1})^{b_3}$
$(Q_aQ_b^{-1})^{b_3}$: ack / nak
Authentication: ack / nak
Secure Chat
encrypted message
response

The server is located at `eitn41.eit.lth.se:1337`. You will be given a test client in Python3 and Java, so you don't have to figure out which socket readers to use. As an example, sending the message (in hexadecimal, un-encrypted)

```
1337
```

will yield the response

```
0b29099972244f2569bb5f6f4e34760cc7bf4645
```

The shared secret is calculated as $H(g^{xy} \parallel$ 'shared passphrase') where the shared passphrase is "`eitn41 <3`", and H is SHA-1. You shall interpret g^{xy} and the passphrase as a UTF-8 encoded byte array, e.g., if the number is 123123, it should be represented (in Python3) as `b'\x01\xe0\xf3'`. The function `x.to_bytes((x.bit_length() + 7) // 8, 'big')` may come in handy. A similar Java function is `num.toByteArray()`, where `num` is a `BigInteger`. Remember to remove leading bytes of `0x00` since Java represents the numbers in 2's complement.

Assessment:

- Upload your code to Urkund, jonathan.sonnerup.lu@analys.urkund.se. Only one student per group must do this.
- There will be one Moodle question following the problem statement above. Both students must finish the Moodle quiz. There will be a test quiz on Moodle, where you can try your implementation as many times as you like. The test quiz will not be graded.