Week 11 Homework due on November 30, 18:59 Hour

Group 5

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

Exercise 1

Design and implement a simple key management protocol. The protocol should base on a KDC that shares keys with Alice and Bob. Alice, initiating communication, should establish (through the KDC) a shared key with Bob. Introduce the following three classes: KDC, Alice, Bob, and present the protocol as an interaction between three objects of these classes.

a) Are you aware of any limitations or security problems of your solution (consider replay attacks, confidentiality and authentication, overheads, etc.)?

b) Do you see any ways of improving them?

Answers:

A KDC shares a secret key with each user and in this case one with Alice and one with Bob. The protocol can be implemented using the following steps:

1) If Alice wants to initiate a communication with Bob, she would send a message to the KDC informing it of her intention.
2) The KDC returns a message containing several elements to Alice. They are:
    a. a shared secret key K_{AB};
    b. the same shared secret key K_{AB} but encrypted with Bob's secret key K_{Bob} which he shares with the KDC. This is also known as the "ticket".

   This entire message is encrypted with Alice's secret key K_{Alice} which she shares with the KDC.

3) Upon receiving this message, Alice decrypts the message to retrieve K_{AB} and sends the "ticket" which contains the secret key K_{AB} and is encrypted with Bob's secret key K_{Bob} to Bob. Alice and Bob can to communicate with each other using this secret key K_{AB}.

The protocol is shown in Figure 1 where A denotes Alice and B denotes Bob, K_{Alice} and K_{Bob} are Alice's and Bob's keys with the KDC and K_{AB} is the shared session key between them.

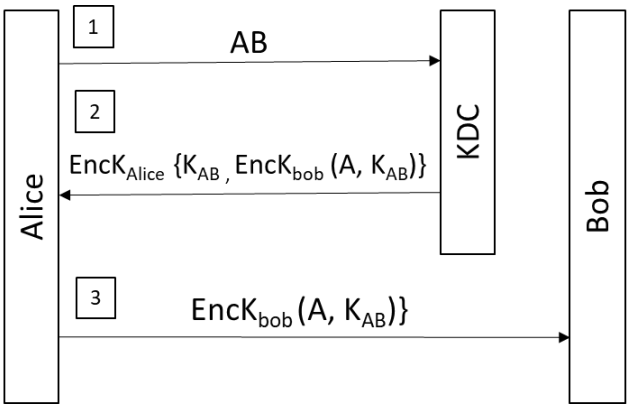

Figure 1

a) There is one limitation or security concern in the above protocol. The messages in step 3 can be replayed by an attacker. He can get hold of an old K_{AB} and replay message to Bob. Bob believing that Alice is on the other end, will start a communication with the attacker using the session key K_{AB}. The solution to this limitation is the introduction of nonce values and this is explained in part b) below.

Another limitation in the simple KDC design is the overhead. An adversary can forge many fake connections and apply for massive tickets, which will exceed the KDC's throughput. To solve this possible problem, the KDC needs to be well-designed to handle massive requests. For example, the KDC can use thread to respond to each request. In addition, the KDC can leverage the asynchronous and callback mechanism to handle requests (similar to Node.js). We can also add a monitoring mechanism in the KDC to detect malicious activities such as massive tickets demand in a short time impose a limit (or throttle) or block suspicious IP addresses.

Finally, the KDC can be hacked and causes the loss of confidentiality and authenticity. The ways to heck the KDC include: Pass-the-ticket: the process of forging a session key and presenting that forgery to the resource as credentials; Golden Ticket: A ticket that grants a user domain admin access; Silver Ticket: A forged ticket that grants access to a service; Credential stuffing/ Brute force: automated continued attempts to guess a password; Encryption downgrade with Skeleton Key Malware: A malware that can bypass Kerberos; and DCShadow attack: an attack where attackers gain enough access inside a network to set up their own DC to use in further infiltration[1]. We shall not provide solutions for the above as it is beyond our knowledge and scope. In the next para, we will provide a solution for the replay attack on the KDC.

[1] Kerberos Authentication Explained - https://www.varonis.com/blog/kerberos-authentication-explained/

b) One of the possible solution is to use nonces to prevent replay attacks. This is shown in Figure 2 where R_{A1}, R_{A2} and R_B are the nonces. The nonce is a random number that is used only once and its purpose is to uniquely relate two messages to each other.

R_{A1} is used by Alice to communicate with the KDC in message 1. On getting the appropriate reply (message 2 which include the ticket for Bob) from the KDC which contains R_{A1}, Alice knows that this is a response to message 1 and not a replay message.

Alice then starts communicating with Bob (in message 3 which includes the ticket) using another nonce number R_{A2}. Bob then decrypts the ticket to find the shared key and sends message 4 to Alice which include R_{A2}-1 and R_B which is Bob's nonce. By returning R_{A2}-1, this proves Bob decrypt the message and knows the shared key and authenticates Bob to Alice.

Alice then sends message 5 with Bob's nonce (-1) to authenticate Alice to Bob. These messages remove the possibility of replace attack.

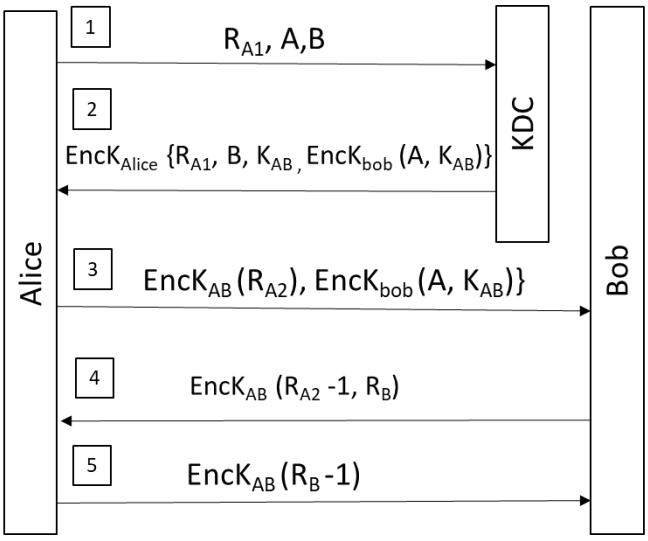

Figure 2

The program written in Python for the simple KDC protocol is listed below.

```
from Cryptodome.Cipher import AES
import hashlib

class KDC(object):
   def __init__(self):
      self.share_dic = {};

   def addSharekey(self, name, key):
      self.share_dic[name] = key;
      print self.share_dic

   def generateTickets(self, from_name, to_name):
      h = hashlib.sha256();
      h.update(self.share_dic[from_name]);
      h.update(self.share_dic[to_name]);
      KAB = h.hexdigest()
      print '\n====== generate KAB ============='
      print 'KAB:', KAB

      IV_bob = hashlib.sha256(to_name).hexdigest()[:16]
      key_bob = self.share_dic[to_name]
      encrypt = AES.new(key_bob, AES.MODE_GCM, IV_bob)
      encryptBob = encrypt.encrypt_and_digest(from_name + KAB)[0]
      print '====== KDC first generate EkB(Alice, KAB) ============='
      print 'EkB(Alice, KAB):', encryptBob.encode('hex');

      IV_alice = hashlib.sha256(from_name).hexdigest()[:16]
      key_alice = self.share_dic[from_name]
      encrypt = AES.new(key_alice, AES.MODE_GCM, IV_alice)
      encryptAlice = encrypt.encrypt_and_digest(KAB + encryptBob)[0]
      print '====== KDC then generate EkA(KAB, EkB(Alice, KAB)) ============='
      print 'EkA(KAB, EkB(Alice, KAB)):', encryptAlice.encode('hex')
      return encryptAlice

class Alice(object):
   def __init__(self, Ka):
      self.name = 'Alice';
```

```
        self.Ka = Ka;
        self.KAB ='';
        print 'kA:', self.Ka

    def addSharekey(self, KDC_object):
        KDC_object.addSharekey(self.name, self.Ka);

    def getTickets(self, name, KDC_object):
        return KDC_object.generateTickets(self.name, name);

    def sendToBob_getResponce(self, message, Bob_object):
        IV_alice = hashlib.sha256(self.name).hexdigest()[:16]
        key_alice = self.Ka
        decryptor = AES.new(key_alice, AES.MODE_GCM, IV_alice)
        plaintext = (decryptor.decrypt(message))
        print '====== Alice first decrypt EkA(KAB, EkB(Alice, KAB)) ============='
        print 'KAB||EkB(Alice, KAB):', plaintext
        print '====== Alice then extract KAB and EkB(Alice, KAB) ============='
        self.KAB = plaintext[0:64].decode('hex')
        tickets = plaintext[64:len(plaintext)]
        print 'KAB:', self.KAB.encode('hex')
        print 'EkB(Alice, KAB):', tickets.encode('hex')
        message = Bob_object.receiveFromAlice(tickets)
        self.parse(message);
        return message;

    def parse(self, message):
        IV = hashlib.sha256('Bob').hexdigest()[:16];
        key = self.KAB;
        decryptor = AES.new(key, AES.MODE_GCM, IV)
        plaintext = (decryptor.decrypt(message))
        print '====== Alice decrypt EKAB(Hi, it is Bob) ============='
        print 'Alice receive:', plaintext;

class Bob(object):
    def __init__(self, Kb):
        self.name = 'Bob';
        self.Kb = Kb;
        self.KAB ='';
```

```
        print 'kB:', self.Kb

    def addSharekey(self, KDC_object):
        KDC_object.addSharekey(self.name, self.Kb);

    def getTickets(self, name, KDC_object):
        return KDC_object.generateTickets(self.name, name);

    def receiveFromAlice(self, tickets):
        IV_bob = hashlib.sha256(self.name).hexdigest()[:16]
        key_bob = self.Kb
        decryptor = AES.new(key_bob, AES.MODE_GCM, IV_bob)
        plaintext = (decryptor.decrypt(tickets))
        print '====== Bob first decrypt EkB(Alice, KAB) ============='
        print 'Alice||KAB:', plaintext
        print '====== Bob then extract Alice and KAB ============='
        self.KAB = plaintext[5:len(plaintext)].decode('hex')
        print 'KAB:', self.KAB.encode('hex')
        encrypt = AES.new(self.KAB, AES.MODE_GCM, IV_bob)
        print '====== Bob then send EKAB(Hi, it is Bob) ============='
        encryptBob = encrypt.encrypt_and_digest('Hi, it is Bob')[0]
        print 'EKAB(Hi, it is Bob):', encryptBob
        return encryptBob

alice = Alice("so secret key-a!")
bob = Bob("so secret key-a!")
kdc = KDC()

alice.addSharekey(kdc)
bob.addSharekey(kdc)

message_from_KDC = alice.getTickets('Bob', kdc)

alice.sendToBob_getResponce(message_from_KDC, bob)
```

The outputs are:

```
kA: so secret key-a!
kB: so secret key-b!
{'Alice': 'so secret key-a!'}
{'Bob': 'so secret key-b!', 'Alice': 'so secret key-a!'}

====== generate KAB =============
KAB: 37bb5d8eb06063c9cb6cd104f5fb2e70bdc7c00760149177f302a94c621a91ad
====== KDC first generate EkB(Alice, KAB) =============
EkB(Alice, KAB):
6b98bdd350a906f572d6331c946358e3a12d3c9f59436a086a3186f68502df9c441bf27b4ad65b5413aebd743b2421dd79f4b5c9a
eabf57afc0d9ece0ea9167a71a5efd81d
====== KDC then generate EkA(KAB, EkB(Alice, KAB)) =============
EkA(KAB, EkB(Alice, KAB)):
d6b7b86af106a3a00c6f58725c257de7e9f36cdb986e53a17276258794beac0519e3dc326dae09e78394ac6a5bc6a6a880eb66f8b
08a89b37489ddb12813585978fc8a3ab05ae6224781d2c6de836faf418c0921dbb10bfbdb9b13686d4992891835564ce00ea7875a
3313605b1aca0e31e8560e62c5b55f4f3310ebcc4a01966beb46f85c
====== Alice first decrypt EkA(KAB, EkB(Alice, KAB)) =====
======= then extract KAB and EkB(Alice, KAB) =============
KAB: 37bb5d8eb06063c9cb6cd104f5fb2e70bdc7c00760149177f302a94c621a91ad
EkB(Alice, KAB):
6b98bdd350a906f572d6331c946358e3a12d3c9f59436a086a3186f68502df9c441bf27b4ad65b5413aebd743b2421dd79f4b5c9a
eabf57afc0d9ece0ea9167a71a5efd81d
====== Bob first decrypt EkB(Alice, KAB) =============
Alice||KAB: Alice37bb5d8eb06063c9cb6cd104f5fb2e70bdc7c00760149177f302a94c621a91ad
====== Bob then extract Alice and KAB =============
KAB: 37bb5d8eb06063c9cb6cd104f5fb2e70bdc7c00760149177f302a94c621a91ad
====== Bob then send EKAB(Hi, it is Bob) =============
EKAB(Hi, it is Bob): ?TXO57?,d?
====== Alice decrypt EKAB(Hi, it is Bob) =============
Alice receive: Hi, it is Bob
```

Exercise 2

Design (or find) a fair non-repudiation protocol to further reduce the TTP's involvement, for example, using an off-line TTP.

- On-line TTP is actively involved in every instance of a non-repudiation service (e.g. the protocol in Slide 38).

- Off-line TTP supports non-repudiation without being involved in each instance of a service.

Answers:

A fair non-repudiation protocol must support non-repudiation of origin and non-repudiation of receipt while neither the originator nor the recipient can gain an advantage by quitting prematurely or otherwise misbehaving during a transaction. It is important to have non-repudiation protocol that uses minimal workload of the trusted third party so that we can achieve efficiency. In this exercise, we will research two distinguished journal papers[2] and share a variant of the non-repudiation protocol that utilizes an off-line TPP to achieve this efficiency. In contrast, in-line and on-line TPP non-repudiation protocols would require the TPP to be actively involved thereby consuming larger amount of computing resources and may cause bottleneck problem.

We first describe the light-weighted on-line TPP non-repudiation protocol which is proposed in journal paper: "An Efficient Non-Repudiation Protocol" by Jianying Zhou and Dieter Gollmann.

From the paper, the main idea of this protocol is to split the definition of a message M into two parts, a commitment C and a key K. The commitment is sent from the originator A to the recipient B and then the key is lodged with the trusted third party TTP. Both A and B have to retrieve the confirmed key from the TTP as part of the nonrepudiation evidence required in the settlement of a dispute. The notation below is used in the protocol description.

- M: message being sent from A to B
- K: message key defined by A
- C = eK(M): commitment (ciphertext) for message M
- L = H(M,K): a unique label linking C and K.
- fi (i = 1, 2, ⋯): flags indicating the intended purpose of a signed message
- EOO_C = $sS_A(f_1$, B, L, C): evidence of origin of C
- EOR_C = $sS_B(f_2$, A, L, C): evidence of receipt of C.
- sub_K = $sS_A(f_5$, B, L, K): evidence of submission of K
- con_K = sS_{TTP} (f_6 , A, B, L, K): evidence of confirmation of K issued by the TTP.

The protocol is as follows.

[2] The papers are: An Efficient Non-Repudiation Protocol written by Jianying Zhou & Dieter Gollmann; and Evolution of Fair Non-repudiation with TTP written by Jianying Zhou, Robert Deng, and Feng Bao.

1. $A \rightarrow B : f_1$, B, L, C,EOO_C
2. $B \rightarrow A : f_2$, A, L,EOR_C
3. $A \rightarrow TTP : f_5$, B, L, K, sub_K
4. $B \leftrightarrow TTP : f_6$, A, B, L, K, con_K
5. $A \leftrightarrow TTP : f_6$, A, B, L, K, con_K

In this protocol, TTP needs to be involved in each protocol run. An improvement could be made to the above protocol to improve its efficiency. This can be achieved by invoking TPP only in the error-recovery phase initiated by originator A when he cannot get the evidence from B. In most normal cases, A and B will exchange messages and non-repudiation evidence directly. This is then the off-line TPP non-repudiation protocol.

In the off-line TPP non-repudiation protocol, the additional notations are used. When disputes relate to the time of message transfer, the originator and the recipient may need evidence about the time of sending and receiving a message besides evidence of origin and receipt. The TTP can time-stamp evidence con_K to identify when the message key, and thus the message, was made available.

- EOO_K = $sS_A(f_3$, B, L, K): evidence of origin of K.
- EOR_K = $sS_B(f_4$, A, L, K): evidence of receipt of K.

The protocol in the normal case is as follows and the TPP will not be involved.

- $A \rightarrow B : f_1$, B, L, C, EOO_C
- $B \rightarrow A : f_2$, A, L, EOR_C
- $A \rightarrow B : f_3$, B, L, K, EOO_K
- $B \rightarrow A : f_4$, A, L, EOR_K

If A does not send message 3, the protocol ends without disputes. If A cannot get message 4 from B after sending message 3 (either because B did not receive message 3 or because B does not want to acknowledge it), A may initiate the following recovery phase, which is the same as Steps 3 to 5 of the protocol described in the former paras on light-weighted on-line TPP non-repudiation protocol.

3′. $A \rightarrow TTP : f_5$, B, L, K, sub_K
4′. $B \leftrightarrow TTP : f_6$, A, B, L, K, con_K
5′. $A \leftrightarrow TTP : f_6$, A, B, L, K, con_K

If the protocol run is complete, the originator A will hold non-repudiation evidence EOR_C and EOR_K, and the recipient B will hold EOO_C and EOO_K. Otherwise, A needs to rectify the unfair situation by initiating the recovery phase so that non-repudiation evidence con_K will be available to both A and B. If disputes arise, A can use (EOR_C, EOR_K) or (EOR_C, con_K) as non-repudiation evidence to prove that B received M; B can use (EOO_C, EOO_K) or (EOO_C,

con_K) as non-repudiation evidence to prove that M originated from A. This protocol will be efficient in an environment where two parties usually play fair in a protocol run. Although the recipient B is temporarily in an advantageous position after Step 3, fairness can be retained by ensuring the success of the recovery phase, which relies on the assumption that the communication channels between the TTP and the participants A, B are resilient. In practice, however, a time limit for a protocol run may have to be set so that both parties can terminate an expired protocol run safely. We will not describe the choice of the time limit which is discussed in the journal paper.

In summary, an efficient and fair non-repudiation protocol can be derived from a light-weighted on-line TPP non-repudiation protocol by turning the TPP offline and only use it when the originator initiates an error-recovery phase. In most normal cases, the originator and the recipient will exchange messages and non-repudiation evidence directly.