2013

Pairwise hierarchy key

Mehran Nasseri & Mahboobeh AbdalMahmoodAbadi

3/18/2013

Table of Contents

1. Introduction

In this assignment, we have implemented one of the most complicated key hierarchies, is called "Pairwise hierarchy key". We did pair programming and all parts of the project were done with participation of both of us. We worked alongside at one system and we were cooperating on the algorithm, program code, and test. One of us as driver typed at the computer and wrote down a code. The other one looked for the algorithms that were needed. We were both active in our position and communicate continually. Albeit we switched our roles between driver and navigator therefore, everyone knows what everyone is doing and everybody remains familiar with the whole system.

2. Pairwise hierarchy key

The IEEE 802.11i EAPOL-key exchange uses a number of keys and has a key hierarchy to divide initial key material into useful keys. The two key hierarchies are:

- **Pairwise key hierarchy**
- Group key hierarchy

The pairwise master key (PMK) is the top key in the whole security context is held by both the user and the server. This "supreme secret" key might be in a smart card, stored on a laptop disk, or remembered in a person's head. In this assignment, PMK is assumed to be remembered by user. (PMK=IK2002)

To generate the pairwise transient key (PTK), a pseudorandom function runs with PMK and other parameters consist of MAC address of both supplicant and authenticator, and nonce value of both parties.

In order to protect a link between two devices, PMK is used to generate set of keys as follow. In fact, the primary master key is not used directly.

- Data Encryption key (128 bits)
- Data Integrity key (128 bits)
- EAPOL-Key Encryption key (128 bits)
- EAPOL-Key Integrity key (128 bits)

The four mentioned keys are referred to as the temporal keys because they are regenerated every time a mobile device communicates to the access point. The collection of all four keys together is referred to as the pairwise transient key (PTK) (figure 1).

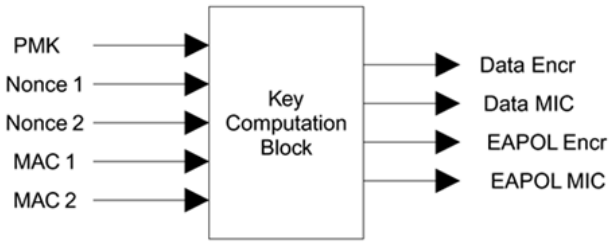

Figure1: Temporal Key Computation

3. Exchanging and Verifying Key Information

One of the main EAPOL-key exchanges defined in IEEE 802.11i is 4-way handshake. The 4-way handshake does several things:

- validates the PMK between the supplicant and authenticator
- computes the temporal keys to be used by the data-confidentiality protocol
- verifies the parameters that were negotiated
- Both devices have synchronized and turned on encryption of unicast packets.

It has called the 4-way handshake (figure 2) since four packets are passed between the supplicant and the authenticator.

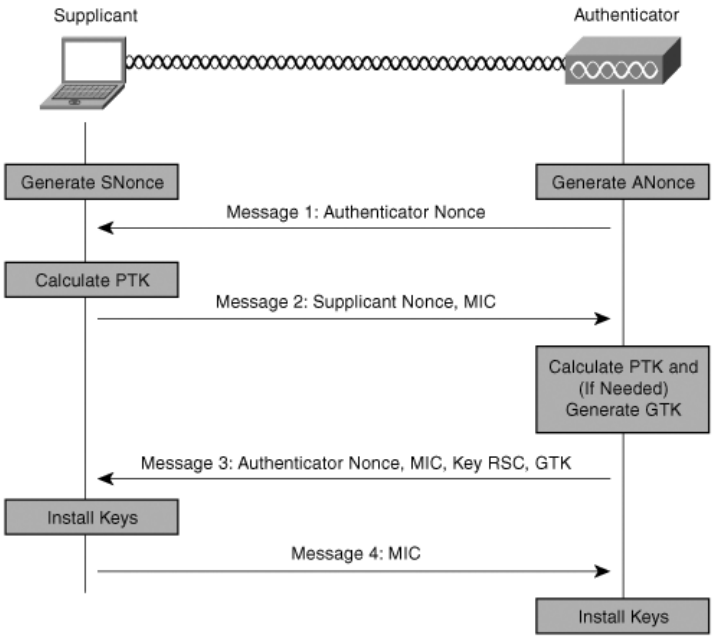

Figure2: 4-way handshake

In WPA, the key exchange is done using a special variant of the EAPOL-Key message, which is different from that defined in IEEE 802.1X (Figure 3).

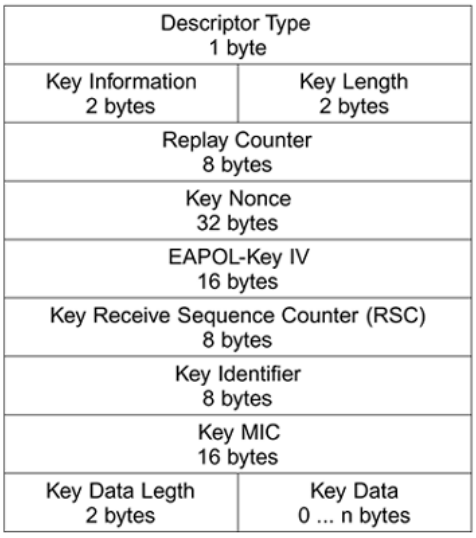

Figure 3: WPA Version of EAPOL-Key Descriptor

Here are the four packets included:

- **Message (A): Authenticator→ Supplicant**

In the first message, the authenticator sends its nonce value. This is referred to as the ANonce and calculated by using HmacSHA1 algorithm.

```
public static byte[] prf_n(int numberOfBits, byte[] keyOrRandomNo, String speceficText, byte[]
sequence) {

    ByteBuffer Buffer = ByteBuffer.allocate(numberOfBits / 8);

    Key secretKey = new SecretKeySpec(keyOrRandomNo, "HmacSHA1");
    Mac msgAuthenticationCode = null;
  try {
    msgAuthenticationCode = Mac.getInstance("HmacSHA1");
  } catch (NoSuchAlgorithmException ex) {
    Logger.getLogger(CryptoGraph.class.getName()).log(Level.SEVERE, null, ex);
  }
  try {
    msgAuthenticationCode.init(secretKey);
  } catch (InvalidKeyException ex) {
    Logger.getLogger(CryptoGraph.class.getName()).log(Level.SEVERE, null, ex);
  }

    ByteBuffer buffer = null;
  try {
    buffer = ByteBuffer.allocate(speceficText.getBytes("UTF8").length + 1 + sequence.length + 1);
```

```
        buffer.put(speceficText.getBytes("UTF8"));
    } catch (UnsupportedEncodingException ex) {
        Logger.getLogger(CryptoGraph.class.getName()).log(Level.SEVERE, null, ex);
    }

        buffer.put((byte) 0x00);
        buffer.put(sequence);
        byte B = 0;
        while (B * 160 < numberOfBits) {
            buffer.put(buffer.capacity() - 1, B++);
            byte[] macResult = msgAuthenticationCode.doFinal(buffer.array());
            Buffer.put(macResult, 0, (Buffer.remaining() > 20) ? 20 : Buffer.remaining());
        }

        return Buffer.array();
  }
```

Starting nonce = PRF-256(Random Number, "Init Counter", MAC || Time)

```
long ran=random.nextLong();
ByteBuffer kBuffer = ByteBuffer.allocate(8).putLong(ran);
authenticatorNonce = CryptoGraph.prf_n(256, kBuffer.array(), "Init Counter",
CryptoGraph.NonceByteSequence(authenticatorMac));
public static byte[] NonceByteSequence(byte[] mac) {
      ByteBuffer buffer = ByteBuffer.allocate(14);
      buffer.put(mac);
      buffer.putLong(System.currentTimeMillis());
      return buffer.array();
  }
```

MAC address retrieved as follow:

```
InetAddress add=clientSocket.getInetAddress();
        authenticatorMac =NetworkInterface.getByInetAddress(add).getHardwareAddress();
        if (authenticatorMac.length != 6) {
        throw new Exception();
      }
```

The nonce values are used only once with a given key. The important thing is that this should hold true even when the mobile device or access point is restarted or even if a Wi-Fi LAN adapter card is moved from one laptop to another.

- **Message (B): Supplicant → Authenticator**

The supplicant creates its nonce. This is referred to as the SNonce and calculated by using HmacSHA1 algorithm and CryptoGraph.prf_n(.) function as described in message(A) section.

The supplicant can now computes the PTK by using HmacSHA1 algorithm.

PRF-512(PMK, "Pairwise key expansion", MAC1||MAC2||Nonce1||Nonce2)

```
byte[] sequence=CryptoGraph.PTKBytesSequence(authenticatorMac, supplicantMac,
authenticatorNonce, supplicantNonce);
pairwiseTransientKey = CryptoGraph.prf_n(512, pairwiseMasterKey, "Pairwaise key expansion",
sequence);
```

In the message B, the supplicant sends the SNonce to the authenticator. Message B includes a message integrity code (MIC) to prevent tampering. This is the first use of the EAPOL-Key Integrity key, one of the four temporal keys. Computing the MIC over the whole of message (B) prevents anyone from modifying the message without detection.

More importantly, it lets the authenticator to confirm that the supplicant has the correct PMK. If both parties has the different PMK, then the MIC check will fail.

Computing MIC By using HmacMD5 algorithm:

```
byte[] EAPOL-Key Integrity key = Arrays.copyOfRange(pairwiseTransientKey, 48, 64);
byte[] message= Arrays.copyOfRange(EapolMsg.eapolMsgToByteArray(msgB), 0, 77);
clientMIC = CryptoGraph.generateMIC(EAPOL-Key Integrity key, message);

public static byte[] generateMIC(byte[] k, byte[] msg) {
        Key secretKey = new SecretKeySpec(k, "HmacMD5");
        Mac msgAuthenticationCode = null;
    try {
        msgAuthenticationCode = Mac.getInstance("HmacMD5");
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(CryptoGraph.class.getName()).log(Level.SEVERE, null, ex);
    }
    try {
        msgAuthenticationCode.init(secretKey);
    } catch (InvalidKeyException ex) {
        Logger.getLogger(CryptoGraph.class.getName()).log(Level.SEVERE, null, ex);
    }
        byte[] macResult = msgAuthenticationCode.doFinal(msg);

        return macResult;
  }
```

Checking MIC by Authenticator:

```
if (!Arrays.equals(serverMIC, clientMIC)) {
        System.out.println("ERROR: wrong MIC! suspect to attack! connection closed!");
        clientSocket.close();
        return;
      }
```

When the message B delivered to authenticator it starts to compute the PTK.

- **Message (C): Authenticator→ Supplicant**

This message is sent by the authenticator to tell the supplicant that it is ready to start using the new keys for encryption. It is important to synchronize this operation because, if either the access point or the mobile device turns on encryption before the other side is ready, the link will break. Message (C) includes a MIC computed by authenticator so the supplicant can confirm that the authenticator has a same PMK.

```
if (!Arrays.equals(clientMIC, serverMIC)) {
        System.out.println("ERROR: wrong MIC! suspect to attack! connection closed!");
        clientSocket.close();
        return;  }
```

It also includes the initial sequence number that will be used for the first encrypted frame to be sent using the key (normally, 0).

```
msgC.setReceiveSeqCounter(0L);
```

- **Message (D): Supplicant → Authenticator**

The supplicant replies with message D, include a MIC. This is the finishing point of the four-way handshake and specifies that the supplicant will now install the keys and start encryption. The message is sent unencrypted and then the supplicant installs its keys. When the authenticator receives this message, it checks the MIC and if it was correct it also installs the keys, so succeeding messages are all encrypted. For this assignment at this step we just simply printout all four keys.

```
ByteBuffer buffer = ByteBuffer.wrap(pairwiseTransientKey);
      byte [] temp=new byte[16];
      buffer.get(temp);
      System.out.println("Data Encryption key (128 bits): " + temp.toString());
      byte [] temp1=new byte[16];
      buffer.get(temp1);
      System.out.println("Data Integrity key (128 bits): " + temp1.toString());
      byte [] temp2=new byte[16];
      buffer.get(temp2);
      System.out.println("EAPOL-Key Encryption key (128 bits): " + temp2.toString());
      byte [] temp3=new byte[16];
```

```
buffer.get(temp3);
System.out.println("EAPOL-Key Integrity key (128 bits): " + temp3.toString());
```