Homework 05

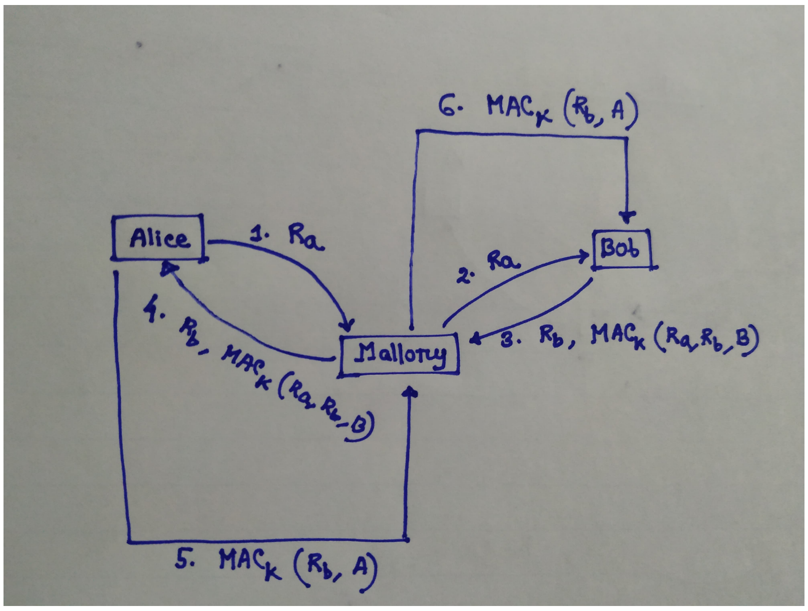
Md Moniruzzaman Monir, # 50291708

October 25, 2018

Problem 01

Solution:

a) In order to mount a **man-in-the-middle** attack Mallory will first intercept the initial message from Alice to Bob. Mallory will get the random number R$_a$ which was sent by Alice to Bob. Then Mallory will send this to Bob. Bob will send a random number R$_b$ and MAC$_k$(R$_a$,R$_b$,B). Mallory will intercept the message and forward it to Alice. Now Alice will authenticate Mallory as Bob. Later Alice will send MAC$_k$(R$_b$,A). Mallory will again intercept this message and forward it to Bob. So Bob will authenticate Mallory as Alice.



As a result of her attack Bob will authenticate Mallory as Alice. And same for Alice. So mutual authentication will not happen. Mallory will get all the messages sent from Alice to Bob and vice versa but Mallory can't decrypt the messages as the shared key K is not known by him.

Mallory is successful in the sense that he can break the mutual authentication protocol but also he can't break the confidentiality of the messages. So Mallory is unsuccessful in this sense that he can't alter the messages.

b) A parallel session attack is not possible by Mallory. Suppose Mallory intercepts the challenge R$_a$ from Alice and now open another session with Alice and sends the challenge R$_a$. Then Alice will send him the MAC$_k$(R$_a$,R$_b$,A) and R$_b$ as response in which MAC includes the name tag A for Alice. So if Mallory send this as a response in the first session then Alice will realize that it is her own messages because it will have the name tag A. So Mallory can't use any responses received from Alice against Alice because of this name tag inside MAC.

Problem 02

Solution:

Advantages of SSL over IPsec :

SSL Works on Transport layer (Layer 4). It's advantages over IPsec is given below:

- Client mobility (device and platform independent)
- Already included in all browsers and most web servers
- Web browser acts as client (clientless)
- Low maintenance costs
- High scalability

Advantages of IPsec over SSL :

IPsec Works on Internet layer (Layer 3). It's advantages over SSL is given below:

- Fully transparent network access
- Economical if running few clients as SSL sessions may need multiple handshakes, making a computationally heavy load for client and SSL devices
- It supports client authentication, pre-shared key mecahnism and compression.
- It works more closely to physcical layer and thus provide better security by encrypting all the packets.

Circumstances when one is preferred over the other :

Choosing IPSec or SSL depends on the security needs and costs. If a specific service is required, and is supported by SSL, then it is better to choose SSL. If overall services or Gateway-to-Gateway communications are needed then IPSec is a good choice. If there are many one time clients then SSL is better. Also if one wants to reduce costs then SSL is better as IPsec needs Installation and updates (drivers, applications) which is expensive if running many clients.

Problem 03

Solution:

a) SSL (Secure Sockets Layer) was originally deployed in Web browsers but now it is the de facto standard for secure communications over the internet. Nowadays we are all relying on software to protect our financial transactions and other critical network communications. So these Non-Browser softwares implement the SSL validation for secure communications. The paper presented different types of bugs in Non-Browser Softwares which validate SSL certificates. Their main finding is that SSL certificate validation is completely broken in many critical software applications and libraries. This is scary. In this paper they only exploit the logic errors in client-side SSL certificate validation. They showed that even when you start with a good protocol, the implementations may not follow it accurately. Or if they do, the resulting shared libraries may be used inappropriately because of incomplete documentation or confusing options. They found that the APIs (or Application Programming Interfaces) of the SSL/TLS implementations which application programmers rely upon have confusing settings and options. These include OpenSSL, GnuTLS, and JSSE, plus data-transport libraries such as cURL. This leads to weaknesses in applications built on these libraries.
   Amazons EC2 Java SDK is vulnerable, breaking the security of all cloud applications based on it. Some popular API calls dont even try to be secure. PHPs fsockopen and Pythons urllib, urllib2 and httplib are popular with programmers, even though they establish SSL connections with no attempt at verifying the servers identity.

   The lessons of the articles are given below:

   - Application software is not being rigorously tested. The state of adversarial testing is exceptionally poor even for critical software such as mobile banking apps and merchants SDKs.
   - Many SSL libraries are unsafe by default, requiring the higher-level software to correctly set several somewhat misleadingly named options and correctly interpret return values. Error reporting is often indirect and easily overlooked.
   - Even those SSL libraries that are safe by default tend to be misused by developers. For example cURL's wrapper around OpenSSL are misused by developers by misinterpreting the meaning of various options. So better documentation and more rigorous formalization of API semantics is required. APIs should present high-level abstraction to developers.

- SSL bugs tend to be buried in the middleware and hard to find.
- In many cases developers deliberately disable certificate validation while assuring both users and higher-level programs that SSL is being supported which is totally false.

b) Security bugs due to the misunderstanding of the SSL API is the most surprising fact to me. I never realize before how important it is to have a clear documentation of an API. The paper showed that the APIs (or Application Programming Interfaces) of the SSL/TLS implementations which application programmers rely upon have confusing settings and options. These include OpenSSL, GnuTLS, and JSSE, plus data-transport libraries such as cURL. This leads to weaknesses in applications built on these libraries. Amazons EC2 Java SDK is vulnerable, breaking the security of all cloud applications based on it. So are the software development kits which Amazon and PayPal on-line merchants use to transmit payment details. Integrated shopping cart applications relying on broken libraries include ZenCart, Ubercart, PrestaShop and osCommerce. Chase Banks mobile banking application and other Android apps are also vulnerable.
These poorly designed APIs confront programmers with confusing low-level interfaces. One example used in the paper is an Amazon PHP based payment library that attempts to secure a connection by setting cURLs CURLOPT_SSL_VERIFYHOST parameter to true. That certainly looks reasonable. However, it should be set to 2. If a programmer sets it to true, that is non-zero and so it ends up being interpreted as 1 and that disables the verification. It is very surprising to me that this simple stupid mistake actually leads to **shop-for-free** attack.

c) Bug in **Chase Mobile Banking Android App** :

In this app the validation checking is turned off while overriding the actual code. Chase is a major bank in USA but their SSL connections establishment by its mobile banking application on Android are insecure against a man-in-the-middle attack. Decompilation and analysis of this app's code show that it overrides the default **x509TrustManager**. The code snippet from reverse engineering is given below :

```
public final void checkServerTrusted(X509Certificate[] a,String paramString){
        if ( (a != null) && ( a.length == 1)) {
                a[0].checkValidity();
        while (true){
            return;
            this.a.checkServerTrusted(a, paramString);
        }
    }
}
```

Here, if we look from the technical point of view then we will see that the unreachable code (invocation of **checkServerTrusted**) is the main weakness. This call is never executed as there is a 'return' command before that line.