Problem Set 3:

Problem 1:

1. The source cannot simply use any message authentication code mechanism based on symmetric key crypto because of the key exchange problem and the trust problem.

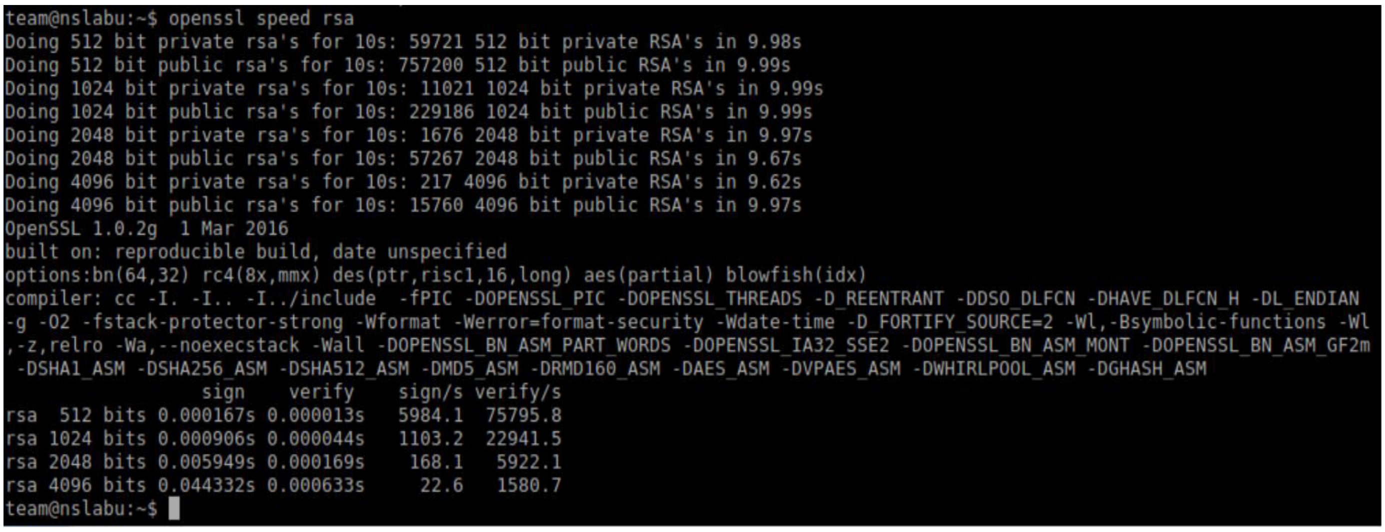
-The key exchange problem arises from the fact that the communicating parties should somehow exchange a secret key before any secret communication can be initiated. Both the parties must also ensure that the key should remain secret.

If an adversary is intercepting the communication then he/she will know the secret key when it is sent from one user to another and hence, will be able to decrypt all the data that was encrypted using that shared key.

-The trust problem arises because, even if we assume that somehow all the receivers and the sender share the same secret key. We will not be able to provide non-repudiation which is one of the basic principles of information security. As the key used in HMAC is known to both the sender and the user. The adversary or any other member from the group could tamper with the message as the same key which is used to compute the MAC is used to verify the MAC as well. So, no one can prove that the data received was the same data that the source had transmitted in the first place.

2. If the sender is now using asymmetric cryptography to sign each packet using his/her private key then,

By running the command -> openssl speed rsa we get:



```
team@nslabu:~$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 59721 512 bit private RSA's in 9.98s
Doing 512 bit public rsa's for 10s: 757200 512 bit public RSA's in 9.99s
Doing 1024 bit private rsa's for 10s: 11021 1024 bit private RSA's in 9.99s
Doing 1024 bit public rsa's for 10s: 229186 1024 bit public RSA's in 9.99s
Doing 2048 bit private rsa's for 10s: 1676 2048 bit private RSA's in 9.97s
Doing 2048 bit public rsa's for 10s: 57267 2048 bit public RSA's in 9.67s
Doing 4096 bit private rsa's for 10s: 217 4096 bit private RSA's in 9.62s
Doing 4096 bit public rsa's for 10s: 15760 4096 bit public RSA's in 9.97s
OpenSSL 1.0.2g  1 Mar 2016
built on: reproducible build, date unspecified
options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial) blowfish(idx)
compiler: cc -I. -I.. -I../include  -fPIC -DOPENSSL_PIC -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN
-g -O2 -fstack-protector-strong -Wformat -Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl
,-z,relro -Wa,--noexecstack -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m
 -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVPAES_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000167s 0.000013s   5984.1  75795.8
rsa 1024 bits 0.000906s 0.000044s   1103.2  22941.5
rsa 2048 bits 0.005949s 0.000169s    168.1   5922.1
rsa 4096 bits 0.044332s 0.000633s     22.6   1580.7
team@nslabu:~$ 
```

a-> The transmitter can sign 5984.1 packets per second by using an RSA key of size 512

The transmitter can sign 1103.2 packets per second by using an RSA key of size 1024

The transmitter can sign 168.1 packets per second by using an RSA key of size 2048

The transmitter can sign 22.6 packets per second by using an RSA key of size 4096

b-> The transmitter can verify 75795.8 packets per second by using an RSA key of size 512

The transmitter can verify 22941.5 packets per second by using an RSA key of size 1024

The transmitter can verify 5922.1 packets per second by using an RSA key of size 2048

The transmitter can verify 1580.7 packets per second by using an RSA key of size 4096

c-> The verification algorithm of RSA and signing algorithms are pretty much similar with the only difference that the verification algorithm depends on the public-key and the signing algorithm is dependent on the private-key.

The algorithm to raise a number to a power is iterative, so the time taken basically depends on the size of the exponent. When a pair of keys is generated, we usually get a large private-key (which is used for decryption and signing) and a small public key (which is used for encryption and verification). In real world scenarios the decryption keys ranges from just under twice as long to almost 17 times as long as the encryption key.

If we consider a small value for the modulus then, we can easily generate many key pairs. So, finding a pair where the decryption key may be smaller or almost like the encryption key is easy. But in a real RSA key it is not so easy and we accept the first pair we get. And as in that case the decryption key is much larger than the encryption key, all public-key operations (message encryption, signature verification) are much faster as compared to the private-key operations (message decryption, signature generation).

No, the length of the packet does not have much impact on the signature/verification time.

Specification of the machine used to run OpenSSL:

Lab VM

OS: Ubuntu(32-bit)

RAM: 1048 MB

Processor(s): 1

Execution Cap: 100%

3. If the sender node does not have the capability to sign all the packets individually using its private key, then we could do as follows:

Encryption-

1. Generate a random key at runtime.
2. Use that random key to encrypt the whole message using symmetric encryption
3. Encrypt the key generated at runtime using the public key of the receiver
4. Join the encrypted message and the encrypted key and sign the whole message

Decryption-

1. Verify the signature of the source
2. Separate the encrypted runtime key and the message
3. Decrypt the runtime key using own private key
4. Use the key retrieved to decrypt the data

A potential DoS attack on this system could be:

As the system uses TCP to communicate, an adversary can simply send a FIN packet to any of the parties involved and close the communication.

If the receiver can send some data to the source. Then the adversary may send many SYN packets from spoofed IP addresses and exhaust the server memory allocated for remembering communications

Problem 2:

GIVEN:

A and B have a secret shared key K_{AB}

1. $A \rightarrow B$: I am A, R_1
2. $B \rightarrow A$: R_2, $K_{AB}\{A\}$, $K_{AB}\{B\}$, $K_{AB}\{R_1\}$
3. $A \rightarrow B$: $K_{AB}\{B\}$, $K_{AB}\{A\}$, $K_{AB}\{R_2\}$

This attack is susceptible to a reflection attack.

Assume that the attacker is intercepting the communication between A and B. Once A sends the first message the attacker knows the message and the random number generated at runtime R1.

When B replies by sending: R_2, $K_{AB}\{A\}$, $K_{AB}\{B\}$, $K_{AB}\{R_1\}$ the adversary records this data and sends another connection request to B saying: I am A, R2. To this message B replies with: R_3, $K_{AB}\{A\}$, $K_{AB}\{B\}$, $K_{AB}\{R_2\}$. The adversary gets the value of $K_{AB}\{R_2\}$ from this message.

Now the adversary has all the information he/she needs to authenticate to the first connection which is: $K_{AB}\{B\}$, $K_{AB}\{A\}$, $K_{AB}\{R_2\}$

The adversary can successfully trick B into believing that it is talking to A.

Avoiding the reflection attack:

We can avoid the reflection attack by making one small change in the protocol. We will ask A to send I am A, $K_{AB}\{R_1\}$. The new protocol should look like:

1. $A \rightarrow B$: I am A, $K_{AB}\{R_1\}$
2. $B \rightarrow A$: R_2, $K_{AB}\{A\}$, $K_{AB}\{B\}$, R_1
3. $A \rightarrow B$: $K_{AB}\{B\}$, $K_{AB}\{A\}$, $K_{AB}\{R_2\}$

As the adversary does not know the key he/she will not be able to send $K_{AB}\{R_2\}$ and get the decrypted value of R_2 to launch a reflection attack.

Problem 3:

RSA decryption component:

GIVEN:

p = 23

q = 17

e = 3

NOW,

n = 23 X 17 = 391

LCM(22 , 16) = 176

We use the equation:

d x e mod 176 = 1

- ⇨ d x 3 mod 176 = 1
- ⇨ 177 mod 176 = 1
- ⇨ 59 x 3 mod 176 = 1

So, decryption component = d = 59

Encrypt M = 4

Encryption function = c(M) = M^e (mod n)

= 4^3 mod 391

= 64 mod 391 = 64

Decrypt M = 2

Decryption function = M(c) = c^d (mod n)

= 2^59 mod 391

Diffie-Hellman shared secret calculation:

GIVEN:

p = 113

g = 3

A’s secret = a = 5

B’s secret = b = 2

Value of shared secret = g^(a*b) (mod p)

= 3^(5*2)(mod 113)

= 3^10 (mod 113)

= 59049 (mod 113)

= 63

Problem 4:

We have been given the following protocols:

Protocol 1:

1. $A \rightarrow B$: $[g_a \text{ mod } p]_A$
2. $B \rightarrow A$: $[g_b \text{ mod } p]_B$

   Shared key $K = g_{ab} \text{ mod } p$.

Protocol 2:

1. $A \rightarrow B: \{C_1\}_B$
2. $B \rightarrow A: \{C_2\}_A$

   Shared key $K = C_1 \oplus C_2$.

Protocol 1, is using the Diffie-Hellman key exchange mechanism coupled with signing the data sent using the private key of each party, to achieve a safe public key exchange which also provides data integrity and **perfect forward secrecy**. This additional feature of **signing** the data before sending it will prevent any adversary who could have changed the keys in between and hijacked the communication. And **perfect forward secrecy** will be ensured because even if an adversary breaks into the system he will not be able to break Diffie-Hellman and compute the a or b values. As a result of which the adversary will not be able to decrypt past communication even if he successfully breaks into the system. Another advantage of protocol 1 over protocol 2 is that protocol 1 does not use asymmetric encryption and hence will use less resources and work faster.

Protocol 2, on the other hand is simply encrypting two numbers using the public key of the receiver. These numbers are then XORed to generate a shared key. No signing is used here so we cannot be sure that $\{C1\}_B$ came from A and not some adversary. If the adversary changes the value of C1 then A and B will compute two different values of K and their authentication will not be successful. Also, this system does not ensure perfect forward secrecy, if an adversary successfully breaks into one of the systems he/she can compute the shared key and decrypt all the past communication.

For perfect forward secrecy to be successfully implemented, the system should forget the values of ‘a’ and ‘b’ once the session is complete. If they remember the values or reuse those values then the adversary might get that data once he/she breaks into one of the systems and find the key to decrypt the past communication.

And for the signing service to be trusted both the parties need to have the public key of the other. Only then will they be able to check if the message was signed by the correct entity.

If these properties of perfect forward secrecy and verification are not required then the advantage of using protocol 2 is that we will not have to compute the Diffie-Hellman shared secret or sign the messages. This will save the time and resources consumed by the computation and the signing and verification process.

Problem 5:

GIVEN: A has a secret W (generated from a password), and B only knows $g^W \bmod p$.

1. $A \rightarrow B: A, g^a \bmod p$
2. $B \rightarrow A: g^b \bmod p, c_1$

   The common key is: $K = \text{Hash}(g^{ab} \bmod p, g^{Wb} \bmod p)$
3. $A \rightarrow B: K\{c_1\}, c_2$

4. $B \rightarrow A$: $K\{c_2\}$

If the given protocol is followed then an adversary can do a Man In The Middle attack and authenticate himself with both the sides by changing the content of the messages, but will not be able to calculate the value of K because he is not aware of W.

So, the security of this protocol greatly depends on the W which was generated from the password.

If the password was chosen from a reasonably small set then the attacker can easily brute force to find the value of W and then he will be able to calculate $g^{Wb} \bmod p$. Which will eventually enable the adversary to calculate the common key K.

Once the adversary has obtained the key he/she can hijack the communication by sending $K\{c_1\}$, c_2 to B and tricking B into thinking that he is A.

Problem 6:

GIVEN:

A and B share a secret key of length **64 bits**

R1 and R2 are two 64-bit numbers

Protocol:

1. $A \rightarrow B$: I am A
2. $B \rightarrow A$: R_1
3. $A \rightarrow B$: Hash($(K+R_1)$ mod 2^64), R_2
4. $B \rightarrow A$: Hash($(K+R_2)$ mod 2^63)

We are only interested in authenticating A to B (one-way authentication)

In the first message A initiates the connection and then B sends it one of the 64-bit random numbers. This number being in plain text can be seen by any adversary who is monitoring the communication.

Now, A needs to send a hash along with the second random number. **As the length of the key K is only 64-bits the attacker can easily brute force it and obtain the key**. Other information needed to recreate the hash is also available to the attacker. R1 is given and 2^64 is hardcoded into the protocol. So, the adversary can recreate the hash easily and authenticate with B impersonating to be A.

Problem 7:

GIVEN:

1. Server -> Client: N1 || PKS

2. Server <- Client: N1 || N2 || PKC
3. Server -> Client: N2 || S-Hash1 || S-Hash2
4. Server <- Client: N1 || C-Hash1 || C-Hash2 || ENC_KeyWrapKey(R-S1)
5. Server -> Client: N2 || ENC_KeyWrapKey(E-S1)
6. Server <- Client: N1 || ENC_KeyWrapKey(R-S2)
7. Server -> Client: N2 || ENC_KeyWrapKey(E-S2)
8. Server <- Client: N1 || [ENC_KeyWrapKey(ConfigData)]

The given protocol has some flaws. An adversary can do a simple Man in the Middle (MITM) attack. And can relay messages from the server to the client and from the client to the server. The way the protocol proceeds the attacker does not need to have any information about the secret 128-bit nonces E-S1, E-S2, R-S1, R-S2 as they come encrypted in the KeyWrapKey derived from the DH shared secret. The attacker can relay these messages and provide the server or the client the values they expected.

The MITM attack can also be used by the adversary to authenticate himself with both the client and the server as the first two components used to compute the Diffie-Hellman shared secret are public information and is not signed. Now the adversary can simply take the data that is sent by the client and relay it to the sever and act as a bridge between them.

There are a few other things the adversary can do to disrupt the communication system which is using this protocol:

The adversary can intercept communication and send a NACK to either client or server and push it out of the communication.

The PIN used for authentication is only 8 digits. Such a short PIN restricted to a set of (0, 9) can easily be brute forced. This can compromise the system.

PSK1 and PSK2 further split the hash of the PIN used into two parts and select the first 128-bits from it. This further weakens the hash and the keys derived.