Problem 1 [30 points]: Broadcast Authentication.
Assume that a node wants to broadcast a real-time stream of data. Each receiver wants to verify
that the data is generated by the source and was not modified, or replayed (i.e., integrity/data origin protection).

$\boxed{\text{Packet } 1}$ $\boxed{\text{Packet } 2}$ $\dots$ $\boxed{\text{Packet } n}$

1. Why can't the source just use any message authentication code mechanism based on symmetric key crypto (e.g., HMAC) if we assuming that all the receivers share the same secret with the source (i.e., what is the threat)?

   For data origin protection, if every node shares the secret key then a proof of origin cannot be constructed because any recipient can construct the same proof as the source. The same holds for integrity protection: if the sender appends the HMAC of the data and the secret key, any receiver that wishes to inject its own bogus data can simply build the HMAC of such data and append it. Other receivers cannot distinguish between legitimate and illegitimate packets.

2. Now, let us assume that the sender uses asymmetric cryptography to sign each packet using his private key. Use openssl (hint: check the speed command) to estimate:
   a. How many packets can the transmitter send per second using an RSA key of size 512, 1024, 2048, and 4096 bits keys?
   b. How many packets can the receiver verify per second for RSA key size of 512, 1024, 2048, 4096 bits keys?

   The answer to these questions is summarized in the table openssl speeed rsa *outputs:*

```
                         sign        verify      sign/s   verify/s
rsa     512 bits    0.000121s   0.000011s   8272.0   91370.4
rsa     1024 bits   0.000587s   0.000031s   1704.2   31835.4
rsa     2048 bits   0.003674s   0.000109s   272.2    9196.4
rsa     4096 bits   0.025627s   0.000407s   39.0     2457.7
```

   c. Why is the signing time different from the verification time? Would the length of the packet matter very much in the signature/verification time?

   The public exponent e in RSA is often chosen to be a small prime number (like 3 or 65537) making encryption and verification to be efficient operations. The decryption exponent d, being the multiplicative inverse of e modulo a big number, guarantees the magnitude of the private exponent to be much larger than e. This means that the decryption and signing operations will take longer due to the magnitude of the exponents they use. As for the length of the packet affecting signature time, we need to look at how data is signed in practice: first a hash is applied to the message. Later, the result of the hash is used as the message to sign with the private key. For verification the hash is also calculated and then compared to the signature.

Obviously, larger messages will take longer to hash, but the bulk of the time is spent signing/verifying the constant-sized output of the hash (hash functions are about 3 orders of magnitude faster than exponentiation), thus signing large chunks of data does not affect signature nor verification time too much.

d. Indicate the specs of the machine you used to run openssl (e.g., lab VM, or Core Duo laptop 2.4GHz with 4GB RAM, etc.)

   Configuration of machine

3. If the sender node does not have the capability to sign all the packets **individually** using its private key, propose a technique to amortize the signatures and trades-off the computation with a delay in verification. Discuss potential DoS attacks on your approach.

   If message size is not a big contributor in signing time, you can delay signature generation by k packets. The sender only sends a signature packet after *k* datagrams have been broadcast. The signature packet has the form $[P_1, P_2, \ldots, P_k]_{\text{SENDER}}$. This way, the signing load of the sender is reduced by a factor of *k*. The tradeoff is on the time a receiver needs to wait to validate a set of packets. Also, a receiver needs all k packets in order to verify the signature. If an adversary is able to interfere with one packet or inject an incorrect packet the receiver won't be able to complete the signature verification for all the other valid *k*-1 packets.

Problem 2 [10 points]:

Assume that *A* and *B* share a secret K_{AB}. Consider the following authentication protocol:

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) $A \rightarrow B$: $K_{AB}\{B\}$, $K_{AB}\{A\}$, $K_{AB}\{R_2\}$

Is this authentication protocol susceptible to attacks? Explain.

Yes, it is susceptible to a reflection attack.

Session (I)

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

Session (II)

	1) $A \rightarrow B$:	I am A, R_2
	2) $B \rightarrow A$:	R_3, $K_{AB}\{A\}$, $K_{AB}\{B\}$, $K_{AB}\{R_2\}$
Session (I)		
	3) $A \rightarrow B$:	$K_{AB}\{B\}$, $K_{AB}\{A\}$, $K_{AB}\{R_2\}$

If it is susceptible to attacks, propose the simplest and most elegant fix you can think of.

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

Problem 3 [10 points]: Use of RSA and DH.
(RSA) Consider the two primes p = 23; q = 17; encryption exponent e = 3
• Compute the decryption component: d = …
• Encrypt M = 4
• Decrypt M = 2
(DH) In the Diffie-Hellman scheme, assume that p = 113, and g = 3. Assume A's secret is a = 5 and B's secret is b = 2. Compute the value of the shared secret established using DH.

Ans:

p = 23; q = 17; encryption exponent e = 3
n=391
$\phi(n)$ =(p-1) * (q-1) = 352

d should satisfy following condition
ed=1 mod $\phi(n)$

Decryption component: d = 235 (235 * 3 % 352)

Public Key (e , n) = (3, 391)
Private Key (d , n) = (235, 391)

Encrypt M = 4

4^3 % 51 = 64

Decrypt M = 2

2^11 % 51 =246

(DH) In the Diffie-Hellman scheme, assume that p = 11, and g = 2. Assume A's secret is a = 5 and B's secret is b = 2. Compute the value of the shared secret established using DH.

p = 113, and g = 3.
a=5 b=2

At A: g^a mod p= 3^5 mod 113 =17
At B: g^b mod p= 3^2 mod 113 =9
Shared secret g^ab mod p = 63

Problem 4 [15 points]
Consider the following two protocols for key establishment.
Protocol 1:
A -> B: { ga mod p }B
B -> A: { gb mod p }A
Share key K = gab mod p.
Protocol 2:
A -> B: { C1 }B
B -> A: { C2 }A
Share key K = C1 $\oplus$ C2.
Explain what additional service is provided by protocol 1 in comparison with protocol 2.
Explain the required conditions to be satisfied for this additional service to be trusted.
If this property is not needed, what would be the advantage of using protocol 2?

Ans:

Perfect Forward Secrecy

Easier to implement

Problem 5 [15 points]
Consider the following authentication protocol. *A* has a secret *W* (generated from a password), and
B only knows g_W mod *p*. The message exchange is as follows:
1) *A* -> *B*: "A", g_a mod *p*
2) *B* -> *A*: g_b mod *p*, c_1
The common key is: *K* = hash(g_{ab} mod *p*, g_{Wb} mod *p*)
3) A -> *B*: *K*{c_1}, c_2
4) *B* -> *A*: *K*{c_2}

Does this protocol have a vulnerability if the password *W* is weak?

Ans:

Attacker when eavesdrop, first thing he observes is Deffie Hellman key flowing unencrypted. Another observation is, plain text and cipher text pair as C1 and K(C1) & C2 and K(C2) is available on communication.

Key here is hash(g^ab mod p, g^Wb mod p)

Here g and p are public.

So first thing he will do is perform man in middle attack and take (g^a mod p) and raise it to z (own quantity) so he will have (g^ a z mod p). and send (g^z mod p) to B

Similarly will accept (g^b mod p) and raise it to z so he will have (g^b z mod p) and send (g^z mod p) to A.

So resultant output is

at A: (g^a z mod p) so key is hash(g^a z mod p, g^W z mod p)

at B: (g^a z mod p) so key is hash(g^b z mod p, g^W z mod p)

attacker knows (g^az mod p) and (g^b z mod p) so he has part of both keys.

so only part he needs to know is g^W z mod p.

In above protocol once DH key exchange is complete, A will send K(c1) and attacker already has c1 from A. So now attacker mounts offline attack on encrypted quantity by raising (g^z mod p) with probable guess say W' and by creating hash of (g^az mod p, g^W'z mod p). And then testing it on K(C1) to check whether he gets correct output as C1 or not. In this way he can find out W.

Problem 6 [10 points]:
Consider the following authentication protocol. Assume that *A* and *B* share a secret key *K* of length 64 bits. R_1 and R_2 are two 64-bits numbers. Does this protocol have any major vulnerability? If yes describe it in detail.

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

Assume that Hash is a cryptographically secure hashing function and that we are **only** interested in authenticating A to B (one-way authentication).

This protocol is vulnerable to key discovery by an attacker feeding specific R_1 and R_2 values to A and B, allowing her to discover one bit of K per 2 exchanges, as opposed to a brute force search of $O(2^{|K|})$. We define a phase as the process of intercepting a party's network address and sending the desired R value to the other party. To obtain the i^{th} key bit the attacker follows these steps:

Let $R_1 = R^1{}_{63}R^1{}_{62} \dots R_1{}^0$, and $R_2 = R_2{}^{63}R_2{}^{62} \dots R^2{}_0$

If $i = 63$:

> In Phase I, send to A: $R_1 = 0$, and in Phase II send to B: $R_2 = 0$.
> Call H_A the hash emitted by A, and H_B the hash emitted by B.
> Then $H_A = H_B$ is equivalent to bit $K_{63} = 0$.

If $i \neq 63$:

> In Phase I, set $R_1{}^{63} = K_{63}$, $R^1{}_{62\dots(i+1)} = \neg K_{62\dots(i+1)}$, $R_i{}^1 = 1$, $R^1{}_{(i-1)\dots0} = 0$;
> and in Phase II, set $R^2{}_{62\dots(i+1)} = \neg K_{62\dots(i+1)}$, $R_i{}^2 = 1$, $R^1{}_{(i-1)\dots0} = 0$.
> Then $H_A = H_B$ is equivalent to $K_i = 0$.

Problem 7 [10 points]:

Consider the following protocol for mutually authenticating a Client to a Server using an 8 digits PIN. If any step fails, the server sends back a NACK message. Does this protocol have a flaw?

```
Server -> Client: N1 || PKS
Server <- Client: N1 || N2 || PKC
Server -> Client: N2 || S-Hash1 || S-Hash2
Server <- Client: N1 || C-Hash1 || C-Hash2 || ENC_KeyWrapKey(R-S1)
// Here the server recovers R-S1 and verifies that it matches with C-Hash1
// If not return NACK
Server -> Client: N2 || ENC_KeyWrapKey(E-S1)
// Here the client recovers E-S1 and verifies that it matches with S-Hash1
// If not return NACK
Server <- Client: N1 || ENC_KeyWrapKey(R-S2)
// Here the server recovers R-S2 and verifies that it matches with C-Hash2
// If not return NACK
Server -> Client: N2 || ENC_KeyWrapKey(E-S2)
// Here the client recovers E-S2 and verifies that it matches with S-Hash2
// If not return NACK
Server <- Client: N1 || [ ENC_KeyWrapKey(ConfigData) ]
```

⍰ || this symbol means concatenation of parameters to form a message.
⍰ N1 is a 128-bit random number (nonce) specified by the Client.
⍰ N2 is a 128-bit random number (nonce) specified by the Server.
⍰ PKC and PKS are Diffie-Hellman public keys of the Client and Server, respectively
⍰ AuthKey is an authentication key derived from the Diffie-Hellman secret, the nonces N1 and N2, and the Client ID.
⍰ HMAC_AuthKey(…) This notation indicates an Authenticator attribute that contains a HMAC keyed hash over the values in parentheses and using the key AuthKey.
⍰ The keyed hash function is HMAC-SHA-256.
⍰ ENC_KeyWrapKey(…) This notation indicates symmetric encryption of the values in

parentheses using the key KeyWrapKey derived from DH. The encryption algorithm is AES-CBC.

 R-S1 and R-S2 are secret 128-bit nonces that, together with C-Hash1 and C-Hash2, can be used by the Server to confirm the Client’s knowledge of the first and second half of the Client’s password, respectively.

 E-S1 and E-S2 are secret 128-bit nonces that, together with S-Hash1 and S-Hash2, can be used by the Client to confirm the Server’s knowledge of the first and second half of the Enrollee’s device password, respectively.

 PSK1 = first 128 bits of HMAC_AuthKey(1st half of 8 digits PIN)

 PSK2 = first 128 bits of HMAC_AuthKey(2nd half of 8 digits PIN)

 S-Hash1 = HMAC_AuthKey(E-S1 || PSK1 || PKC || PKS)

 S-Hash2 = HMAC_AuthKey(E-S2 || PSK2 || PKC || PKS)

 C-Hash1 = HMAC_AuthKey(R-S1 || PSK1 || PKC || PKS)

 C-Hash2 = HMAC_AuthKey(R-S2 || PSK2 || PKC || PKS)

Ans:

Please go through for solution.

http://web.archive.org/web/20120426081839/http://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf