Review Final — Solutions

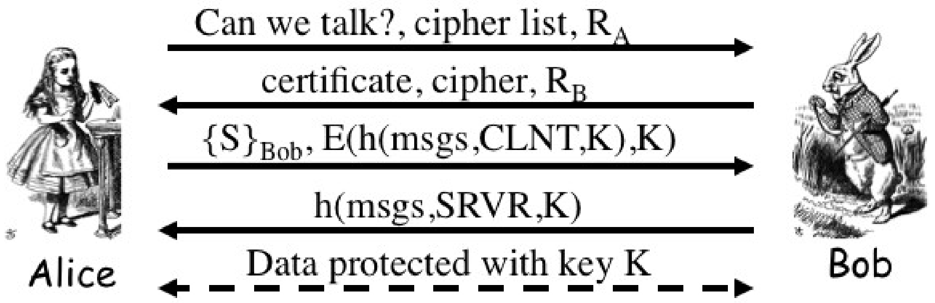
Name ______________________________________

Student ID number __________________________

Notation:	$\{X\}_{\text{Bob}}$	Apply Bob's public key to X
	$[Y]_{\text{Bob}}$	Apply Bob's private key to Y
	$E(P,K)$	Encrypt P with symmetric key K
	$D(C,K)$	Decrypt C with symmetric key K
	$h(x)$	Apply the cryptographic hash function h to x

Directions: Read each problem carefully and provide complete but concise answers. Note that when analyzing protocols, we assume that the cryptography is secure.

1. (10 points) The SSL protocol is illustrated in the figure below, where the session key is $K = h(S, R_A, R_B)$.


Can we talk?, cipher list, R_A
certificate, cipher, R_B
{S}_Bob, E(h(msgs,CLNT,K),K)
h(msgs,SRVR,K)
Data protected with key K
Alice
Bob

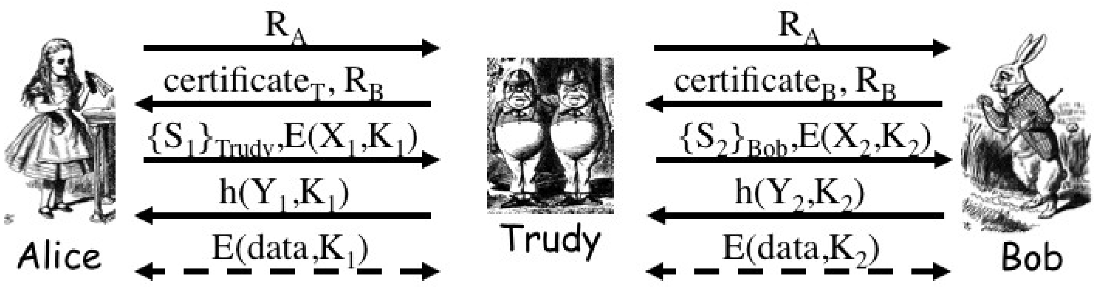
(a) Is Alice authenticated? If so, what authenticates Alice and how does Bob verify it? If not, why not?

Alice is not authenticated since only public key operations are required, which anyone can do.

(b) Is Bob authenticated? If so, what authenticates Bob and how does Alice verify it? If not, why not?

Yes, Bob is authenticated. To compute the hash in message 4 requires S, which is generated by Alice and encrypted with Bob's public key. By verifying the signature on the certificate, Alice is confident that only Bob can obtain S from $\{S\}_{\text{Bob}}$, and Alice can verify the 4th message is correct, since she knows S.

2. (10 points) A man-in-the-middle attack on SSL is illustrated in the figure below.



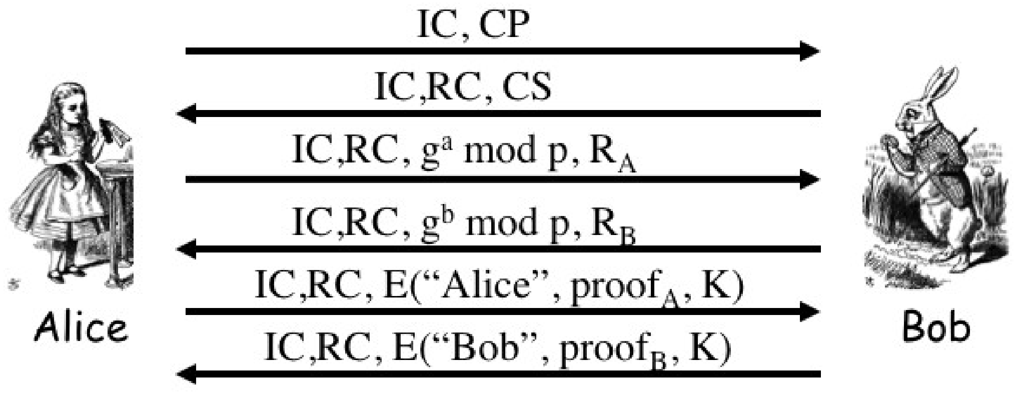
(a) Why does this attack fail to break the protocol? Be precise.

Alice cannot verify the signature on the certificate, so the authentication fails, as it should.

(b) In spite of the fact that this attack does not break the protocol, the attack would often succeed in practice. Explain this apparent contradiction.

Alice's browser will issue a warning that the certificate verification has failed, and give Alice the option to continue. If Alice chooses to continue, the attack will succeed.

3. (10 points) IPSEC, phase 1, main mode, symmetric key option is illustrated below, where $K = h(\text{IC}, \text{RC}, g^{ab} \bmod p, R_A, R_B, K_{AB})$, and K_{AB} is a symmetric key shared by Alice and Bob.



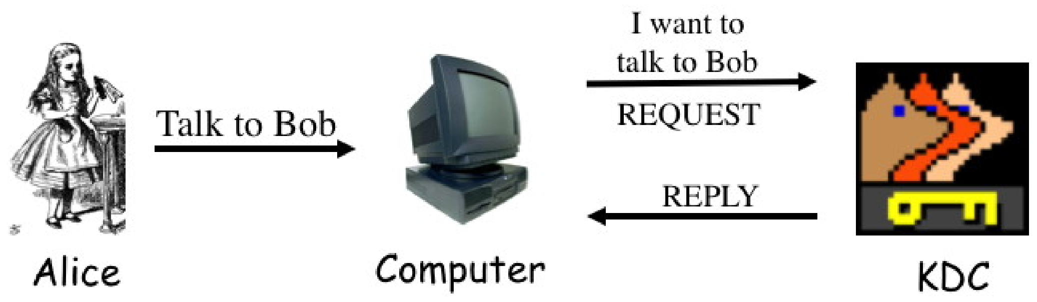
(a) As discussed in class and in the textbook, this protocol fails to provide meaningful anonymity. Since main modes are supposed to provide anonymity, this is a significant flaw. Clearly explain why this protocol fails to provide anonymity.

There is a Catch-22 situation—Bob must know that he is talking to Alice before he can decrypt message 5, which is the first place that Alice identifies herself. Consequently, Bob uses Alice's IP address to identify Alice. But this implies that Alice has a static IP address, which offers very limited (if any) anonymity.

(b) Is the corresponding aggressive mode also flawed, that is, does aggressive mode also fail to deliver its advertised level of security? Why or why not?

Aggressive mode is not flawed, since aggressive mode does not attempt to provide anonymity.

4. (10 points) This problem deals with the Kerberos interaction below, where REQUEST = (TGT, authenticator) and REPLY = $E(\text{"Bob"}, K_{AB}, \text{ticket to Bob}, S_A)$.



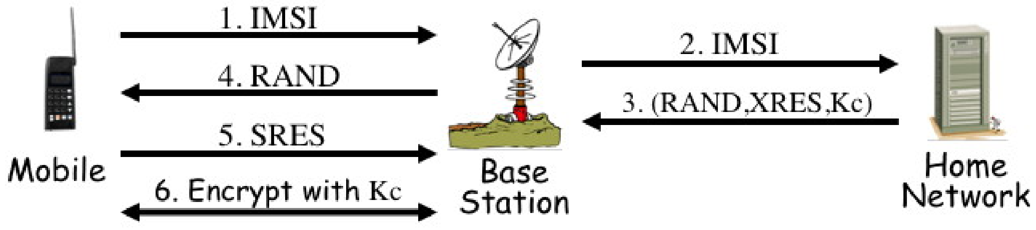
(a) Note that Alice never identifies herself. Why can Alice remain anonymous in this interaction? Explain.

Alice's TGT contains her identity, and the TGT is encrypted the a key that is known only to the KDC. Furthermore, all TGTs are encrypted with the same key, so the KDC does not need to know that it is talking to Alice before decrypting Alice's TGT.

(b) The REPLY contains a "ticket to Bob" which Alice must extract and then forward to Bob before they can communicate. Why is the "ticket to Bob" sent by the KDC to Alice—who must then forward it to Bob—instead of being sent directly from the KDC to Bob?

This allows Bob to remain stateless. If the "ticket to Bob" was sent directly to Bob, then Bob would have to maintain state while waiting for Alice to contact him.

5. (10 points) The GSM authentication protocol is illustrated below.



(a) Is the mobile authenticated? Is the base station authenticated? Justify your answers.

The mobile is authenticated, since Ki is used to compute SRES $= h(\text{RAND}, Ki)$, and the base station can verify that the mobile knows Ki, since SRES $=$ XRES. The base station is not authenticated.

(b) A "fake base station" attack relies on two weaknesses of the protocol. What are those two weaknesses and why do they enable the attack to succeed?

The two weaknesses are: 1) the base station is not authenticated and 2) the base station determines whether the call is encrypted or not.

6. (10 points) This problem deals with malware detection.

   (a) Briefly explain how anomaly-based detection works.

   Anomaly-based detection looks for unusual, abnormal, or “virus-like” behavior or characteristics.

   (b) Give one significant advantage of anomaly detection as compared to signature-based detection. Give one significant advantage of signature-based detection as compared to anomaly detection.

   The biggest advantage of anomaly-based detection is that you can potentially detect previously-unknown malware. There are many challenges with anomaly-based detection, such as the fact that it is difficult to define “normal” and the normal baseline must evolve over time.

7. (10 points) The code below illustrates a heap overflow.

```
int main()
{
     int diff, size = 8;
     char *buf1, *buf2;
     buf1 = (char *)malloc(size);
     buf2 = (char *)malloc(size);
     diff = buf2 - buf1;
     memset(buf2, ’2’, size);
     printf("BEFORE: buf2 = %s\n", buf2);
     memset(buf1, ’1’, diff + 3);
     printf("AFTER: buf2 = %s \n", buf2);
     return 0;
}
```

(a) What is printed when this code is executed?

```
BEFORE: buf2 = 22222222
AFTER: buf2 = 11122222
```

(b) How, in general, might Trudy exploit a heap overflow to attack a system?

A heap overflow can be used to overwrite data, and thereby change the behavior of a program. The specific example discussed in class involved overwriting a bit used to determined authentication.

8. (10 points) We discussed several defenses against stack-based buffer overflow attacks. For each of the following defenses, briefly explain how it works to prevent buffer overflow attacks. Also, mention the relative strengths and significant weaknesses (if any) of each defense.

   (a) Safe programming languages (such as Java)

   The language itself verifies that we do not write outside array bounds. This is a very strong defense, but there is a performance penalty.

   (b) NX bit

   We can set parts of memory (such as the stack) to "no execute." This does not prevent all buffer overflow attacks. For example, the so-called "return-to-libc" attack can still work (that is, an attack that does not inject code, but instead simply jumps to a specific location and executes the code that is already there).

   (c) Canary

   A special value, known as a canary, is pushed onto the stack after the return address. If the canary is corrupted, the return address is not to be trusted. This is an effective defense, but some implementations have been flawed.

   (d) Address space layout randomization (ASLR)

   Applications are loaded into random locations in memory. This is a strong defense, since code injection attacks rely on hardcoded addresses. The weaknesses include implementation flaws and "de-randomization" attacks.

9. (10 points) An excerpt from a program that checks a serial number appears below. Discuss in detail how you would patch this code (in the reverse engineering sense), so that any serial number will work.

```
.text:00401003          push  offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008          call  sub_4010AF
.text:0040100D          lea   eax, [esp+18h+var_14]
.text:00401011          push  eax
.text:00401012          push  offset aS        ; "%s"
.text:00401017          call  sub_401098
.text:0040101C          push  8
.text:0040101E          lea   ecx, [esp+24h+var_14]
.text:00401022          push  offset xyz
.text:00401027          push  ecx
.text:00401028          call  sub_401060
.text:0040102D          add   esp, 18h
.text:00401030          test  eax, eax
.text:00401032          jz    short loc_401045
.text:00401034          push  offset aErrorIncorrect ; "Error! Incorrect serial number.
.text:00401039          call  sub_4010AF
```

You can change the line at `00401030` to `xor eax, eax`. Note that there are many other possible attacks—you should be able to list several.

10. (10 points) Consider the following code, which is designed to verify a serial number before allowing access to the useful parts of the code.

```
if(length(inputString) == 8)// verify input is 8 characters
{
     for(i = 0; i < 8; ++i)// compare input to serial number
     {
          if(inputString[i] != serialNumber[i]) break;
     }
     if(i == 8)
     {
          printf("\nSerial number is correct!\n\n");
          goto run;// run the program
     }
     else
     {
          printf("\nIncorrect serial number\n\n");
          exit(0);
     }
}
else
{
     printf("\nSerial number must be 8 hex digits (0 thru F)\n\n");
     exit(0);
}
```

(a) What is a "linearization attack" and why is the program above susceptible to such an attack?

"Linearization" refers to an attack that picks apart a serial number (or key, or password, etc.), instead of attacking it all at once. In this particular program, careful timing would enable Trudy to determine the serial number one character at a time, since the more leading characters that are correct, the longer the program takes to execute.

(b) Modify the program so that for any incorrect serial numbers, the program takes exactly the same amount of time to execute. Note that this will make the program immune to a linearization attack. Hint: You cannot rely on predefined functions (such as `strcmp` or `strncmp`) or conditionals. Also, inserting a random delay is not a correct solution.

The following code satisfies the requirements:

```
if(length(inputString) == 8)// verify input is 8 characters
{
     count = 0;
     for(i = 0; i < 8; ++i)// compare input to serial number
     {
          count += (inputString[i] ^ serialNumber[i]);
     }
     if(count == 0)
     {
          printf("\nSerial number is correct!\n\n");
          goto run;// run the program
     }
     else
     {
          printf("\nIncorrect serial number\n\n");
          exit(0);
     }
}
else
{
     printf("\nSerial number must be 8 hex digits (0 thru F)\n\n");
     exit(0);
}
```