一、 保证信息安全的关键要素有哪些

安全的五大核心要素。

1. **认证(Authentication)。**认证是安全的最基本要素。信息系统的目的就是供使用者使用的，但只能给获得授权的使用者使用，因此，首先必须知道来访者的身份。使用者可以是人、设备和相关系统，无论是什么样的使用者,安全的第一要素就是对其进行认证。认证的结果无非是三种：可以授权使用的对象，不可以授权使用的对象和无法确认的对象。在信息化系统中，对每一个可能的入口都必须采取认证措施，对无法采取认证措施的入口必须完全堵死，从而防堵每一个安全漏洞。来访对象的身份得到认证之后，对不可授权的对象就必须拒绝访问，对可授权的对象则进到下一步安全流程，对无法确认的对象则视来访的目的采取相应的步骤。例如，公众 Web 的使用和邮件的接收等，虽然对来访对象无法完全认证，但也不能拒之门外。
2. **授权(Authorization)。**授权就是授予合法使用者对系统资源的使用权限并且对非法使用行为进行监测。授权可以是对具体的对象进行授权，例如，某一用户或某台设备可以使用所指定的资源。授权可以是对具体的对象进行授权，也可以是对某一组对象授权，也可以是根据对象所扮演的角色授权。授权除了授予某种权利之外，对于非法使用的发现和管理也是很重要的部分。尽管使用各种安全技术，非法使用并不是可以完全避免的，因此，及时发现非法使用并马上采取安全措施是非常重要的。例如，当病毒侵入信息系统后，如果不及时发现并采取安全措施，后果可以是非常严重的。
3. **保密(Confidentiality)。**认证和授权是信息安全的基础，但是光有这两项是不够的。保密是要确保信息在传送过程和存储时不被非法使用者“看”到。一个典型的例子是，合法使用者在使用信息时要通过网络，这时信息在传送的过程中可能被非法“截取”并导致泄密。一般来说，信息在存储时比较容易通过认证和授权的手段将非授权使用者“拒之门外”。但是，数据在传送过程中则无法或很难做到这一点，因此，加密技术就成为了信息保密的重要手段。
4. **完整性(Integrity)。**如果说信息的失密是一个严重的安全问题，那么信息在存储和传送过程中被修改则更严重了。例如，A 给 B 传送一个文件和指令。在其传送过程中，C 将信息截取并修改，并将修改后的信息传送给 B，使 B 认为被 C 修改了的内容就是 A 所传送的内容。在这种情况下，信息的完整性被破坏了。信息安全的一个重要方面就是保证信息的完整性，特别是信息在传送过程中的完整性。
5. **不可否认(Non-repudiation)。**无论是授权的使用还是非授权的使用，事后都应该是有据可查的。对于非授权的使用，必须是非授权的使用者无法否认或抵赖的，这应该是信息安全的最后一个重要环节。

二、 分析对称密码可否用于数字签名

不能。简单来说，如果使用的对称密钥，也就是说，签名和验证用的是同一个密钥 k，那么任一个持有 k 的人都给完成签名，这个文件到底是谁签

的就不知道了(**签名算法本身是完全公开的，所有人都知道，签名用的密钥才是验证“是谁签的”的唯一手段**)。

具体例子：假设你和银行都有可以用来签名的、相同的密钥 k，你在第一次转账一千块的时候给自己的电子支票用 k 签了个名。银行内部某人知道 k（很正常，因为 k 必须公开，否则其他人没法验证你的签名）,通过这次交易知道了你的卡号，然后他用你的信息和 k 弄了个一万块的电子支票转账给他自己。月底你查账单，发现亏了一万，起诉银行，结果败诉，原因是谁也说不清到底是你自己用 k 签的名还是别人用 k 签的名。

三、 分析 AES 算法中的雪崩效应

所谓理想的雪崩效应特性，是指密钥或明文变化一个比特时，对应的密文将有大致一半的比特发生变化。

给出了一个采用 128 比特密钥的 AES 算法加密实例，所选择的 明文、密钥及加密所得密文分别如下

输入明文：32 88 31 e0 43 5a 31 37 f6 30 98 07 a8 8d a2 34

输入密钥：2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

输出密文：39 02 dc 19 25 dc 11 6a 84 09 85 0b 1d fb 97 32

将该明文和密钥作为输入参数，运行我们编写的 AES 算法程序，可以得到完全相同的输出密文；对该密文进行解密，可以恢复出相同的明文。这说明，我们编写的 AES 算法加、解密程序是完全正确的。

为了测试 AES 算法的密钥雪崩效应特性,我们将上述的 128 比特明文、密钥和密文作为测试标准，在保持明文不变的前提下，按 1~128 比特编号的顺序依次改变密钥的 1 个比特然后进行加密，得到对应的密文和相对于标准密文的密文变化比特数如表所示。

密文变化比特序号	128 比特密钥	密文变化比特数
1	ab7e151628aed2a6abf7158809cf4f3c	59
2	6b7e151628aed2a6abf7158809cf4f3c	67
3	4b7e151628aed2a6abf7158809cf4f3c	66
4	3b7e151628aed2a6abf7158809cf4f3c	61
5	337e151628aed2a6abf7158809cf4f3c	69
6	2f7e151628aed2a6abf7158809cf4f3c	60
7	2d7e151628aed2a6abf7158809cf4f3c	70
8	2c7e151628aed2a6abf7158809cf4f3c	59
9	2bfe151628aed2a6abf7158809cf4f3c	61
10	2bbe151628aed2a6abf7158809cf4f3c	63
11	2b9e151628aed2a6abf7158809cf4f3c	58
12	2b8e151628aed2a6abf7158809cf4f3c	65
13	2b8615162 8aed2a6abf7158809cf4f3c	70
14	2b82151628aed2a6abf7158809cf4f3c	64
15	2b80151628aed2a6abf7158809cf4f3c	59
16	2b7f151628aed2a6abf7158809cf4f3c	57
17	2b7e951628aed2a6abf7158809cf4f3c	69
18	2b7e551628aed2a6abf7158809cf4f3c	61
19	2b7e351628aed2a6abf7158809cf4f3c	68
20	2b7e251628aed2a6abf7158809cf4f3c	56
21	2b7e1d1628aed2a6abf7158809cf4f3c	62
22	2b7e191628aed2a6abf7158809cf4f3c	61
23	2b7e171628aed2a6abf7158809cf4f3c	64
24	2b7e161628aed2a6abf7158809cf4f3c	72
25	2b7e159628aed2a6abf7158809cf4f3c	68
26	2b7e155628aed2a6abf7158809cf4f3c	58
27	2b7e153628aed2a6abf7158809cf4f3c	56
28	2b7e152628aed2a6abf7158809cf4f3c	62
29	2b7e151e28aed2a6abf7158809cf4f3c	62
30	2b7e151a28aed2a6abf7158809cf4f3c	57
31	2b7e151828aed2a6abf7158809cf4f3c	64
32	2b7e151728aed2a6abf7158809cf4f3c	57
33	2b7e1516a8aed2a6abf7158809cf4f3c	71
34	2b7e151668aed2a6abf7158809cf4f3c	54
35	2b7e151648aed2a6abf7158809cf4f3c	58
36	2b7e151638aed2a6abf7158809cf4f3c	64
37	2b7e151630aed2a6abf7158809cf4f3c	65
38	2b7e15162caed2a6abf7158809cf4f3c	57
39	2b7e15162aaed2a6abf7158809cf4f3c	54
40	2b7e151629aed2a6abf7158809cf4f3c	64
41	2b7e1516292ed2a6abf7158809cf4f3c	68

由表（表中 42 到 128 位的数据略）可见，在采用本测试标准参数的条件下，当改变密钥的第 77 比特时，对应的密文变化比特数最大，达到 82 比特；当改变密钥的第 34、39、53 和 90 比特时，对应的密文变化比特数最小，为 54 比特。**平均变化比特数为 64.27，非常接近 128 比特的半数——64 比特。测试结果表明，AES 算法的确具有非常好的密钥雪崩效应特性。**

四、 基于散列函数的特点分析数字摘要的防篡改性

散列函数（或散列算法，英语：Hash Function）是一种从任何一种数据中创建小的数字“指纹”的方法。散列函数把消息或数据压缩成摘要，使得数据量变小，将数据的格式固定下来。该函数将数据打乱混合，重新创建一个叫做散列值的指纹。

举个简单的例子，一个二进制的文件，只要有任一位的数据变动，就会导致得出整个 hash 值发生巨大的变化，因此，hash 值可以当作文件的身份证。

但实际上由于 hash 值是定长的，不同的文件得出相同的值也是有可能的（就是比你中彩票的几率还要低上个几亿倍），这种情况被称为哈希函数的碰撞。我国的王小云教授就因为攻破了 MD5 等而闻名。

基于哈希函数的防篡改技术实际上就是对文件提取下最初的 hash 值，以后只要 hash 值不对了就证明文件被更改过。

五、 分析四种保密通信协议的实际应用场景.

1. 古典密码
   算法简单，适用于家庭密码，普通大众日常密码。
2. 对称密码
   依赖于可信第三方，可扩展性差，但加密速度快，适用于间谍与中间人通信。
3. 公钥密码
   密钥管理简单，安全性依赖于公钥的可靠，适用于小文件传输，网络通讯运营商。
4. 混合密码。
   加密速度快，密钥管理简单，安全性依赖于公钥的可靠，适用性最高，各种情况都能较好适应。

六、 分析带 hash 的基于公钥密码的数字签名协议的签名性质

a) 签字是可以被确认的
b) 签字是无法被伪造的
c) 签字是无法重复使用的
d) 文件被签字以后是无法被篡改的
e) 签字具有无可否认性

举个例子

A 给 B 发送 MessageA 时，先 Hash 将 MessageA 生成一段摘要（Digest，Hash(MessageA) -> DigestA），然后将 DigestA 用私钥加密生成数字签名（Digital Signature, Encrypt(DigestA) -> SignatureA），将 SignatureA 附在 MessageA 之后（MessageA + SignatureA）发送给 B.

B 收到消息之后先用 A 的公钥解密数字签名（Decrypt(SignatureX) -> DigestX），如果能顺利解密，说明消息是由 A 发送的；B 再将 MessageX Hash 生成摘要（Hash(MessageX) -> DigestX2），如果 DigestX 和 DigestX2 相等，说明消息没有被修改过（MessageX == MessageA）。

七、 SKEY 协议中单向 hash 链的使用是反向的,如果采用正向使用是否可行,安全性如何

我认为安全性也是可以的，但没必要。

首先依据 $X_{i+1} = h(X_i)$ 得到 X_0, X_1 , X_2 , X_3 ,……, X_n 。

若用户第一次使用 X_0，（不加密），那么服务器会保存 $X_1 = h(X_0)$，因为哈希函数是公开的，则黑客攻击服务器获取 X_1 或不良服务器使用 X_1 进行交易，不用等到用户再一次交易，即可类比获得用户 X_2 到 X_n 的密钥。

若用户对 X_0 进行加密，每次加密密钥相同，这密钥被破解的概率大大增加，安全性减弱；若每次加密密钥不同，又会引入密钥管理问题，既然如此，为何不直接使用单向散列函数链呢？

八、 分析 Diffie-Hellman 协议的安全性

在选择了合适的 G 和 g 时，这个协议被认为是窃听安全的。偷听者可能必须通过求解 Diffie-Hellman 问题来得到 g^(a*b)。在当前，这被认为是一个困难问题。如果出现了一个高效的解决离散对数问题的算法，那么可以用它来简化 a 或者 b 的计算,那么也就可以用来解决 Diffie-Hellman 问题，使得包括本系统在内的很多公钥密码学系统变得不安全。

G 的阶应当是一个素数，或者它有一个足够大的素因子以防止使用 Pohlig－Hellman 算法来得到 a 或者 b。由于这个原因，一个 Sophie Germain 素数 q 可以用来计算素数 p=2q+1，这样的 p 称为安全素数，因为使用它之后 G 的阶只能被 2 和 q 整除。g 有时被选择成 G 的 q 阶子群的生成元，而不是 G 本身的生成元，这样 g^a 的勒让德符号将不会显示出 a 的低位。

秘密的整数 a 和 b 在会话结束后会被丢弃。因此，**Diffie-Hellman 密钥交换本身能够天然地达到完备的前向安全性，因为私钥不会存在一个过长的时间而增加泄密的危险。**

九、 分析 CBC 模式中的错误传播

加密时，明文中的微小改变会导致其后的全部密文块发生改变，而在解密时，从两个邻接的密文块中即可得到一个明文块。因此，解密过程可以被并行化，而**解密时，密文中一位的改变只会导致其对应的明文块完全改变和下一个明文块中对应位发生改变，不会影响到其它明文的内容。**

十、 编程实现 RC4 算法密钥流的生成

```
unsigned char rc4_crypto(unsigned char *sbox, unsigned char *dat
a, unsigned long len)
{
    int i = 0, j = 0;
    unsigned char tmp = 0;
```

```
    unsigned long k = 0;
    for(k = 0; k < len; k++)
    {
        i = (i + 1) % MAX;
        j = (j + sbox[i]) % MAX;
        tmp = sbox[i];
        sbox[i] = sbox[j];
        sbox[j] = tmp;
        int t = (sbox[i] + sbox[j]) % MAX;
        data[k] ^= sbox[t];
    }
}
```