Kunal Mukherjee
CS 6324
3/7/2021
Dr. Chung Hwan Kim

Homework #2

Problem 1:

a> The attack will be done in stages:

1> First for all the people on earth we will capture a sample of plaintext and ciphertext.

2> We will use a 2d array to store ciphertext for all the dictionary words and all the possible keys that can be used to encrypt them. Let, this data structure be called array_CT.

3> Initialization step will be to create this structure, so that we can query it later and get the keys. array_CT[dict_word][key_combination] = DES_Enc$_{key\_combination}$[dict_word]

4> To get the key for each person, we will have to find the CT for persons in array_CT, and the column value will give us the key.

```
// initialization function
InitializationStep():
    // initialize the array with 0s
    For words in dictonary:
        For key in AllCombinationKey:
            Array_CT[word][key] = 0

    // initialization step of finding all possible ways of encryption the word
    // using all possible keys
    For words in dictonary:
        For key in AllCombinationKey:
            Array_CT[word][key] = DES_Enc_key [word]

// recover keys function
recoverKeys(Array_CT , CT, PT):

    // Method 1
    // get the possible keys
    possibleKeys = []
    for j in Array_CT.col.size
        if Array_CT[PT][j] == CT
            possibleKeys.append(j)

    // query all the keys to find the actual key
    for key in possibleKeys:
```

```
                    if CT == Array_CT [PT][key]:
                            return key

            // Method 2
            row, col = Array_CT.find(CT)
            return column
```

b> It will not be very effective since the key and the block size both increases. They key size increase from 56 to 256 (in worst case), and that would make exhausting the key space computationally intractable. But, with huge amount of time and resource we will be able to employ the 1a attack.

c> The attack would be effective as well as fast. We will encrypt only one word and use it to get everyone's key. Let say, we chose a word e.g. "king" and get the cipher text from everyone. We will then populate array_CT[key] for all the possible keys that can be used to encrypt "king". Then, we will use this 1D array to find the CT of everyone in array_CT[key] and the index will be the key. We will decrease the storage and the search time by a factor of O(n).

d> It will not be effective. In CBC (cipher block chaining) the CT depends on the previous output, so we cannot miss any block for anyone. Another issue is that being able to exhaustively search the key is not enough, the attacker needs to find the correct IV and the encryption output of the previous block. The person needs to know the IV and the exact chain sequence. In the chosen plaintext attack the attacker cannot specify his own plaintext and encrypt it. Thus, if the attacker misses the initial block or any of the subsequent block, the attacker will not be able to recover the key.

Problem 2:

a> We can check 2^32 keys / sec.
Key len = 56, 64, 80.
Total possible keys = 2^56, 2^64, 2^80.
sec needed = total possible keys / check keys per sec
years needed = sec needed / 60 x 60 x 24 x 365

56: 0.53 years
64: 136.19 years
80: 8925512.96 years

b> 128 bits: 183380186319.94287 universe age
Look at problem2.py to see the code for answers.

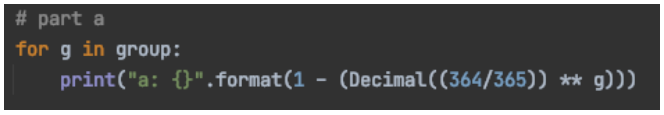
Key len = 128.
Total possible keys = 2^128.
sec needed = total possible keys / check keys per sec

years needed = sec needed / 60 x 60 x 24 x 365
universe needed = years needed / 13700000000

c> It can be broken by brute force if the password is non-random or follows a specific pattern such as words from a book or a deterministic algorithm. The attacker will therefore use side channel attacks as well as pattern matching to create the good guess password. The side channel attack is created in such a way, that if the password authentication mechanism leaks any information that can be used as a pointer to create a password that has strong probability of passing. Padding oracle attack or BEAST attack can be done even with CBC and random IV. In padding oracle attack, the attacker can change the last byte and find out if it is a valid padding or not by looking at the decrypted text. Using this leaked information, the attacker can figure out the last byte and doing it repetitively use it to get the full key. Guessing attack is as the name suggests the attackers randomly makes educated guess to crack the password. If it is a non-random password, or pattern-based password attacker can easily crack it by brute force. It was said in Lec6, "on average the attacker only has to try half the number of guesses before finding the correct one." Therefore, just the cipher having computation intractability does not mean it cannot be broken by brute force. It also has to be sufficiently random so that the attacker has to try the greatest number of possibilities before arriving at the correct answer.

Problem 3:

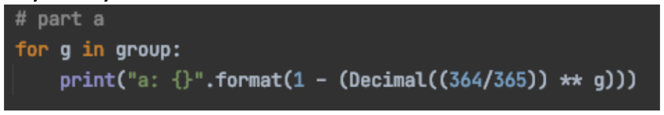
a> Assumption: 365 dates are possible equally likely. And we are finding that there is "at least" with DOB as Jan 1.
1 person has 364/365 probability of not having DOB as Jan 1.
Group A has 25 people. So, 25 person has (364 / 365)^25 probability of not having DOB as Jan 1.
Group A has 1 – (364 / 365)^25 = 0.0662879151841627872612219017 of having DOB Jan 1.
Group B has 1 – (364 / 365)^35 = 0.0915559772793593173930375215 of having DOB Jan 1.
Group C has 1 – (364 / 365)^180 = 0.3897139334802960098958858577 of having DOB Jan 1.



```
# part a
for g in group:
    print("a: {}".format(1 - (Decimal((364/365)) ** g)))
```

b> Assumption: 365 dates are possible equally likely. And we are finding that there is "at least" with DOB as my bday.
1 person has 364/365 probability of not having DOB as my bday.
Group A has 25 people. So, 25 person has (364 / 365)^25 probability of not having DOB as my bday.

Group A has 1 – (364 / 365)^25 = 0.0662879151841627872612219017 of having DOB as my bday.
Group B has 1 – (364 / 365)^35 = 0.0915559772793593173930375215 of having DOB as my bday.
Group C has 1 – (364 / 365)^180 = 0.3897139334802960098958858577of having DOB as my bday.



```
# part a
for g in group:
    print("a: {}".format(1 - (Decimal((364/365)) ** g)))
```

c> p(B) = probability that two person have same birthday
p(B) = probability that two person don't have same birthday
p(B) = 1 - p(B')

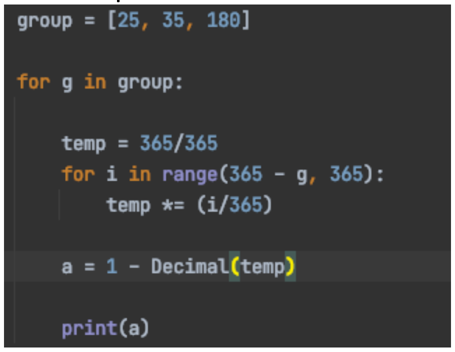
p(b1) = person 1 does not the same birthday as person 1
p(b2) = person 2 does not the same birthday as person 1
p(b3) = person 3 does not the same birthday as person 2, person1
p(b35) = person 25 does not the same birthday as person 24, person 23, person 22…person 1

p(B') = p(b1) x p(b2) x … x p(b25)
= 365/365 x 364/365 x … x 340/365 = (1/365)^25 x (365 x … x 340)
P(B) = 1 - (1/365)^25 x (365 x … x 340) = 0.5982408201359389332552041196

Similarly, for group B,
p(B') = p(b1) x p(b2) x … x p(b35)
= 365/365 x 364/365 x … x 330/365 = (1/365)^35 x (365 x … x 330)
P(B) = 1 - (1/365)^35 x (365 x … x 330) = 0.8321821063798796047628059114

Similarly, for group C,
p(B') = p(b1) x p(b2) x … x p(b180)
= 365/365 x 364/365 x … x 185/365 = (1/365)^180 x (365 x … x 185)
P(B) = 1 - (1/365)^180 x (365 x … x 185) = 0.9999999999999999999999981088

Code snipbit:



```
group = [25, 35, 180]

for g in group:

    temp = 365/365
    for i in range(365 - g, 365):
        temp *= (i/365)

    a = 1 - Decimal(temp)

    print(a)
```

Problem 4:

a> Existential Forgery under a chosen plaintext attack means that the attacker can choose any message, *m*, and can create a valid signature for the message, *sig,* even if they don't hold the private key. As long as the pair (*m*, *sig*), are valid, the attacker is able to execute an Existential Forgery. MAC needs to resist it meaning, that without the private key, k, the attacker should not be able to generate a valid signature for any message, even if it is reused, restructured or reformatted.

b> The construction of MAC is insecure because after the attacker sees a message and its corresponding authentication code pair, e.g. (M, T), it can combine any part of the original message and still be able to reuse it with the old tag T as a valid tag. Let's say, M = m1, m2, …, m*l*, and T = AES_k [m1] ⊕ … ⊕ AES_k [m*l*] . This issue arises because of the commutative property of XOR, a ⊕ b = b ⊕ a, or in our case, AES_k (m1) ⊕ AES_k (m2) = AES_k (m2) ⊕ AES_k (m1).

Let's say the attacker created M2 which is the reverse of M1 = m*l*,…,m1, and finds the tag, T' = AES_k [m*l*] ⊕ … ⊕ AES_k [m1]. Using, the commuative property, we can see, AES_k [m*l*] ⊕ … ⊕ AES_k [m1] = AES_k [m1] ⊕ … ⊕ AES_k [m*l*]. Thus, T' = T. So, the attacker can safely, verify the forged message, M2 with the tag T.

Also, the attack mentioned in part 4c can be utilized for this case also. This case the attacker sees the message and its corresponding authentication code pair as (m, A), where M = m1, m2, …, m*l*. Now, the attacker sees another message and its corresponding authentication code pair as (m', A'), where m' = m'1, m'2, …, m'*l*. The adversary can compute, ~M = (m1 ,…, m*l*, m'1⊕ A, m'2, …, m'*l*). The message, ~M and authentication code, A' even though it is a forgery, but it will pass the verification.

Proof:
~M = (m1, m2, …, m*l,* m'1⊕ A, m'2, … , m'*l*)
MAC_k(~M) = (AES_k [m1] ⊕ … ⊕ AES_k [m*l*] ⊕ AES_k [m'1] ⊕ A ⊕ AES_k [m'2] ⊕ … ⊕ AES_k [m'*l*])
Now, A can be expanded as (AES_k [m1] ⊕ … ⊕ AES_k [m*l*])
MAC_k(~M) = (AES_k [m1] ⊕ … ⊕ AES_k [m*l*] ⊕ AES_k [m'1] ⊕ AES_k [m1] ⊕ … ⊕ AES_k [m*l*]) ⊕ AES_k [m'2] ⊕ … ⊕ AES_k [m'*l*])
MAC_k(~M) = (AES_k [m'1] ⊕ AES_k [m'2] ⊕ … ⊕ AES_k [m'*l*]) = A'

Another attack we can employ is the BEAST attack. Here, the attacker instead of trying to modify the message will try to guess a PT and verify if it is correct or not. The attacker captures two continuous ciphertext block C0 and C1. Then, the attacker will try to guess P1, and accurately verify using CBC. Let's day, say the attacker chose P1= x. The attacker will carefully form P2 = x ⊕ C0 ⊕ C1. Using the MAC formulae, MAC_k(C2) = AES_k [C1 ⊕ P2] = AES_k [C1 ⊕ x ⊕ C0 ⊕ C1] = AES_k [x ⊕ C0]. Now, AES_k [x ⊕ C0] will be equal to C1 if x = P1.

c> The attacker sees the message with padding and its corresponding authentication code pair as (m, A), where M = Pad1||m1, Pad2||m2, …, Pad*l*||m*l*. Now, the attacker sees another message and its corresponding authentication code pair as (m', A'), where m' = Pad1||m'1, Pad2||m'2, …, Pad*l*||m'*l*. The adversary can compute, ~M = (Pad1||m1 ,…, Pad*l* ||m*l*, Pad1||m'1$\oplus$ A, Pad2||m'2, …, Pad *l*||m'*l*). The message, ~M and authentication code, A' is a forgery, but it will pass the verification.

Proof:
~M = (Pad1||m1, Pad2||m2, …, Pad*l*||m*l*, Pad1||m'1$\oplus$ A, Pad2||m'2, … ,Pad*l* ||m'*l*)
MAC_k(~M) = (AES_k [Pad1||m1] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m*l*] $\oplus$ AES_k [Pad1||m'1] $\oplus$ A $\oplus$ AES_k [Pad2||m'2] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m'*l*])
Now, A can be expanded as (AES_k [Pad1||m1] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m*l*])
MAC_k(~M) = (AES_k [Pad1||m1] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m*l*] $\oplus$ AES_k [m'1] $\oplus$ AES_k [Pad1||m1] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m*l*]) $\oplus$ AES_k [Pad2||m'2] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m'*l*])
MAC_k(~M) = (AES_k [Pad1||m'1] $\oplus$ AES_k [Pad2||m'2] $\oplus$ … $\oplus$ AES_k [Pad*l* ||m'*l*]) = A`

Problem 5:

a> CBC-MAC of M = for two block message CBC-MAC_k(M1M2)
$C_1 = IV \oplus E_k(M_1)$
$C_2 = C_1 \oplus E_k(M_2) = IV \oplus E_k(M_1) \oplus E_k(M_2)$

b> This is a chosen-cipher text attack, where that attacker can get a CBC-MAC tag T for a message M1, and tag T' for message M2. Then, we can create a new message M'' for which the CBC-MAC tag will be T'. M'' can be created by XORing the tag T with the first block of arbitrary message M2, and concatenating the message M. The tag T' is a valid tag for the combined FORGED message, M''.

Let, say (M, T), where M = m1, m2, …, m*l*. Now, the attacker sees another message and its corresponding authentication code pair as (M2, T'), where M2 = m'1, m'2, …, m'*l*. The adversary can compute MAC for ~M = (m1,…, m*l*, m'1$\oplus$ A, m'2, …,m'*l*). The message, ~M and authentication code, A' is a forgery, but it will pass the verification.

Proof:
~M = (m1 , … , m*l* , m'1 $\oplus$ T , m'2 , … , m'*l*)
CBC-MAC_k(~M) = (E_k[m1] $\oplus$ … $\oplus$ E_k[m*l*] $\oplus$ E_k[m'1] $\oplus$ T $\oplus$ E_k[m'2] $\oplus$ … $\oplus$ E_k[m'*l*])
Now, A can be expanded as (E_k[m1] $\oplus$ … $\oplus$ E_k[m*l*])
CBC-MAC_k(~M) = (E_k[m1] $\oplus$ … $\oplus$ E_k[m*l*] $\oplus$ E_k[m'1] $\oplus$ (E_k[m1] $\oplus$ … $\oplus$ E_k[m*l*]) $\oplus$ E_k[m'2] $\oplus$ … $\oplus$ E_k[m'*l*])
CBC-MAC_k(~M) = (E_k[m'1] $\oplus$ E_k[m'2] $\oplus$ … $\oplus$ E_k[m'*l*]) = T`

Problem 6:

a> P: 6073 Q: 9967
phi (60529591): 60513552.

Please look at problem6.py for the code to generate the result.

```
P, Q = None, None
for p in p_list:
    for q in q_list:
        if p * q == N:
            print("{} * {} = {}".format(p, q, N))
            P, Q = p, q

    if P is not None:
        break

# run it once 6073 * 9967 = 60529591
P, Q = 6073, 9967
print("P: {} Q: {}".format(P, Q))

phi_n = (P-1) * (Q-1)
print("Phi({}): {} ".format(N, phi_n))
```

b> d = 29280751. So, (31*29280751) mod 60513552 = 1.

```
# problem 6b
e = 31

d = 0
while True:
    temp = (e * d) % phi_n
    if temp == 1 and d < phi_n:
        print("d={}".format(d))
        break
    else:
        d += 1
```

c> I convinced myself.

d> compute M = C^d % n = (M^e % n) ^d % n = M^(e*d) % n

M=CT^d % n = 10000^29280751 % 60529591 = 34910012
M=CT^d % n = 20000^29280751 % 60529591 = 33207991
M=CT^d % n = 30000^29280751 % 60529591 = 54348463
M=CT^d % n = 40000^29280751 % 60529591 = 31535938
M=CT^d % n = 50000^29280751 % 60529591 = 5844860
M=CT^d % n = 60000^29280751 % 60529591 = 46207307

```
# compute M = C^d % n = (M^e % n)^d % n = M^(e*d) % n
# M=CT^d % n = 10000^29280751 % 60529591 = 34910012
# M=CT^d % n = 20000^29280751 % 60529591 = 33207991
# M=CT^d % n = 30000^29280751 % 60529591 = 54348463
# M=CT^d % n = 40000^29280751 % 60529591 = 31535938
# M=CT^d % n = 50000^29280751 % 60529591 = 5844860
# M=CT^d % n = 60000^29280751 % 60529591 = 46207307
for CT in CTs:
    print("M=CT^d % n = {}^{} % {} = {}".format(CT, d, N, (CT ** d) % N))
```

e> Since, we cannot factor N, we have to recover *e*. So, that we can find e^{-1}. *e* is special, as $d = e^{-1}$ mod phi(n). If we can get *e*, we can recover *m as:*
$M = (CT)^d \bmod n = (M^e)^d \bmod n = (M^{(ed)}) \bmod n = (M^{(e*e\wedge(-1))}) \bmod n$
$M = (M^1) \bmod n$.
If *e* is small, the adversary can do an exhaustive search and try all the possible values of e. But, if *e* is large then we will have to use a special algorithm that is Baby Step Giant Step algorithm, to try all the values of *e*.

Another attack, for small e, such that M^e is not greater than n. Then, the equation simplifies $C = M^e$. And, M is the eth root of C, $M = \sqrt[e]{C}$.

Problem 7:

a> Look at g = 3. Please look at problem7.py for the code to generate the result.

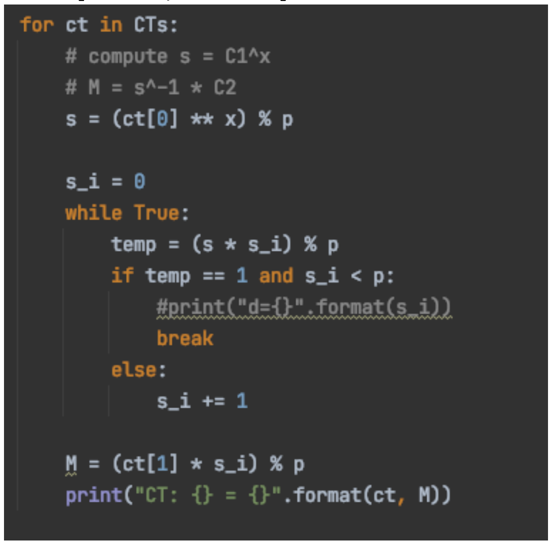
```
# find generator
g = 0
_g_list = list(sympy.sieve.primerange(0, p))

for _g in _g_list:
    _Z_p = []

    for i in range(0, len(Z_p)):
        _Z_p.append((_g ** i) % p)

    _Z_p.sort()

    if Z_p == _Z_p and _g != 1:
        g = _g
        break
    else:
        print("{}/{} g not found".format(_g, len(_g_list)))
```

b> x = 90620.

```
# b> find x = 90620
x = 0
for _x in range(0, len(Z_p) - 1):
    if (g ** _x) % p == 90307:
        x = _x
        print("x: {}".format(_x))
```

c> El Gamal decryption method Wiki
s = C1^x
M = s^-1 * C2

# CT: [5000, 5001] = 85101
# CT: [10000, 20000] = 4996
# CT: [30000, 40000] = 31532
# CT: [50000, 60000] = 28134
# CT: [70000, 80000] = 51546
# CT: [90000, 100000] = 46927



```
for ct in CTs:
    # compute s = C1^x
    # M = s^-1 * C2
    s = (ct[0] ** x) % p

    s_i = 0
    while True:
        temp = (s * s_i) % p
        if temp == 1 and s_i < p:
            #print("d={}".format(s_i))
            break
        else:
            s_i += 1

    M = (ct[1] * s_i) % p
    print("CT: {} = {}".format(ct, M))
```

d> No. For, El Gamal the public keys don't have a property that we can exploit to recover the message M. Computational Diffie Hellman (CDH) problem guarantees that M cannot be fully recovered unless you are able to factor N. That's why IND-CPA requires that Decision Diffie Hellman (DDH) problem because and we know DDH assumption is DDH is hard to solve and to solve DDH you have to solve CDH. And, we know CDH is hard to solve.

Problem 8:

a> Given two groups X and Y. Both contains m = 2^(n/2)
Number of messages in between group pair = m^2 = 2^[(n/2)*(2)] = 2^n messages.

No collision probability of 1 message = 1- ($1/2^n$)
No collision probability of m^2 message between groups = [1- ($1/2^n$)]^m^2 = [1- ($1/2^n$)]^2^n
At least one collision = 1- [1- ($1/2^n$)]^2^n

For n=128 = 1- [1- ($1/2^{128}$)]^2^{128} = 0E-27 ≈ 0

b> For systematically generate the candidate the candidates, first thing we will do is change the price to contract 1 to \$10 Million, let this be contract 2. We are trying to execute a birthday attack by attacking the collision resistance of the hash function. But, here we can change both the documents.
So, first we will calculate the hash of contract 2. We will change the paragraph structure, sentence structure and word choice(synonym words) for contract1, modified-contract1.

Then, we will calculate hash for each combination. If we get a hash of modified-contract1 be the same as contract2, we are done. As, the hash of contract1 and contract2 are same and the only thing differ between contract1 and contract2 is essentially the amount.
If by changing any of the modified contract1 did not work, we will change the contract2's sentence re-structure and check with all the combinations of contract1. Then, if that does not work, we will change paragraph structure and check. Finally, we will use synonym words of contract2 and check.
If none of them work, we will start by adding sentences and then paragraphs.

```
FindHashCollision(Contract1):

Contract2 = Contract1.changeAmount

For all combination ParagraphStructure (Contracts1):
    Modified-contract1 = contract1. changeParagraphStructure

    If Hash(Modified-contract1) == Hash(contract2)
        Return  Modified-contract1, contract2

    For all combination SentenceStructure(Modified-contracts1):

        Modified-contract1 = contract1. changeSentenceStructure

        If Hash(Modified-contract1) == Hash(contract2)
            Return  Modified-contract1, contract2

        For all combination synonymWords(Modified-contracts1):

            Modified-contract1 = contract1.getsynonymWord

            If Hash(Modified-contract1) == Hash(contract2)
                Return  Modified-contract1, contract2

            Else if not tried contract2.changeParagraphStructure
                Contract2 = contract2.changeParagraphStructure

            Else if not tried contract2.changeSentenceStructure
                Contract2 = contract2. changeSentenceStructure

            Else if not tried contract2.SynonymWords
                Contract2 = contract2.getSynonymWords

    While (True):
```

```
Contract1 = contract1.addContent

For all combination ParagraphStructure (Contracts1):

    Modified-contract1 = contract1. changeParagraphStructure

    If Hash(Modified-contract1) == Hash(contract2)
        Return  Modified-contract1, contract2

    For all combination SentenceStructure(Modified-contracts1):

        Modified-contract1 = contract1. changeSentenceStructure

        If Hash(Modified-contract1) == Hash(contract2)
            Return  Modified-contract1, contract2

            For all combination synonymWords(Modified-
            contracts1):
                Modified-contract1 =
            contract1.getsynonymWord

                If Hash(Modified-contract1) ==
                Hash(contract2)
                    Return  Modified-contract1,
                    contract2

                Else if not tried
            contract2.changeParagraphStructure
                    Contract2 =
                contract2.changeParagraphStructure

                Else if not tried
            contract2.changeSentenceStructure
                    Contract2 = contract2.
                changeSentenceStructure

                Else if not tried contract2.SynonymWords
                    Contract2 =
            contract2.getSynonymWords
Contract2 = contract2.addContent
```

Problem 9:

a> Given, that every character has 128 possible values, and the K is a site private key of length *l*. Therefore, we will have to try at most 128 x *l* values, so this attack can be in 128*l* time.

The attack will go as follows:

1> Username1 = "AAAAAAA" (7 char)
2> Get the T1.
3> Try all the 128 values for get the value1, such that hash (username1+value1) = T1
4> Username2 = "AAAAAA" +value1 get the T2.
5> Try all the 128 values for get the value2, such that hash (username2+value2) = T2
6> Username3 = "AAAAA" +value1+value2 Get the T3.
7> Try all the 128 values for get the value2, such that hash (username3+value3) = T3
8> Continue till username8…
9> Username8 = value1+ value2…+value7 Get the T8.
10> Try all the 128 values for get the value8, such that hash (username8 + value8) = T8
11> Thus, after at most 128*l* time and 8 user accounts, you have completely recovered the *K*.

Problem 10:

a> 1> It is an unrealistic expectation for users to remember past messages as well as paraphrase nor completely divulge past messages, so they don't divulge the plaintext corresponding to the cipher text. That's why the system needs to be secure against plain-text attack. The user should be able to use the system and forget about the past.
2> Security against chosen plaintext attack gives the users the confidence that the attacker cannot just plant messages in their system. Thus, it is providing integrity for the encryption system.
3> Chosen and known plaintext attack is a system identification problem, where the attacker can divulge issues about the system itself such as fault divergence and information theft by carefully crafting exploits, pushing them onto the system and then looking at how the system behaves or outputs.
4> Chosen plaintext attack can help distinguish friend from foe, by looking at the encrypted challenge sent by the party. So, being secure against chosen plaintext attack, the system is being secure against unauthorized authentication key divulgence.

b> 1> The issue about keys. It is "unrealistic" to expect that in the first initial meeting we will meet and exchange keys.
2> Also, it is unrealistic that for a huge group, the large number of (n^2 -n)/2 pairs of keys can be arranged in advance.
3> The cost and delay because of key distribution is huge hurdle for business communication
4> Large keys such as One-Time pad is impractical for most applications

c> 1> There is the issue of distinguishing legitimate access to a shared directory verses illegitimate access without knowing the user in advance.
2> The function for one-way is invertible from computation point-of-view not mathematical standpoint. Thus, it is not future proof, as stronger hardware with huge computation power can easily brute force them.
3> Symmetric authentication needs a fair amount of computation power for a valid login or authentication. For example, if an intermediate step, ‘S^t’ is incremented and stored, for this scheme proposed by Lamport to work for 1 month it would need ‘t’ to be 1.3 million. That means for a valid login, the computation cost increased by 1.3 million, and this can quickly go out of hand in most senarios.

d> 1> A public-key encryption can be used to generate digital signature
Alice wants to send M to Bob, she first decrypts it using her private key, so she sends M’ = Dec_k_alice_pri[M] to Bob. Bob can be certain of the authenticity by being able to encrypt it using Alice’s public key, Enc_k_alice_pub[M’] and get M back. Bob also saved, M’= Dec_k_alice_pri[M], as proof that the message came from Alice. Anyone can verify it, by doing this operation M = Enc_k_alice_pub[Dec_k_alice_pri[M]]. Since, only Alice can generate the message M’ = Dec_k_alice_pri[M] with such a property, it is authentication Alice is the originator of the message. Thus, public-key encryption can be used to generate digital signature.

2> public-key encryption can be used for public key distribution
Alice will choose a random key, RK, and encrypt a message with Bob’s public key, EK =Enc_k_bob_pub[RK]. Then, send EK to Bob. Bob will be able to recover the random key back RK by using this private key to decrypt. RK = Decr_k_bob_pri [Enc_k_bob_pub[RK]].

3> digital signature can be used for public key distribution
Digital signature can be used to verify the originator of the public key and if it can be trusted or not. It can be done by Alice will choose a random key, RK, and encrypt a message, M. Alice will encrypt it using the trap-door function of Bob and send both the plaintext and ciphertext to Bob. Bob, since he has the trap-door key, will use the plaintext and ciphertext to get the random key, RK.