Eric Rothman
Practical Cryptographic Systems
Writing HW 2

Problem 1: Short Answer

1. Consider the following variant of the Encrypt-and-MAC construction:
C = Encrypt(k1, M)||MAC(k2, M||ctr)
Where ctr is a counter that increments after each message is encrypted, such that the same value is never used twice. Explain why this might not be insecure.

It isn't insecure because it no longer violates IND-CRA since the MAC function no longer has a deterministic encryption since the cipher text will changed depending on the ctr. So the message can't be decided in the test by just sending the exact plain text through earlier.

2. Describe the consequences (on the plaintext, following decryption) of flipping a single bit in a CFB mode ciphertext.

It will flip that equivalent bit in the plaintext of the block the cipher text bit was in and will change the value of the vector generated from that ciphertext, so it will change the entire next block that is decrypted to probably nonsense.

3. Explain why it might not be safe to use a small public RSA exponent (for example, e = 3) in some cases.

If a small public RSA exponent is used, and used for a lot of different accounts, the encryption may no longer be secure. If k>e transmitted ciphertexts of the same message but with different N can be obtained by the attack, then using the Hasted's Broadcast attack it is possible to recover the message. So if e = 3, only 3 ciphertexts of the same message need to be obtained. Also if e is small then using a Franklin-Reiter Related Message Attack it is possible to recover messages from ciphertexts if multiple messages if the messages are related in quadratic time in e. Coppersmith's short pad attack demonstarted that even if the messages are two different paddings of the same message, if e is small it takes a short amount of time to find the original M. Finally if e is small and at least a quarter of the bits of d is revealed, it is possible to calculate the rest of d. So in conclusion of the exponent is small the time it takes to break the RSA encryption is small also and certain attacks are only possible when e is small, such as 3.

4. Why is it unsafe to share the same prime between two different RSA public moduli (e.g., imagine I generate N1 = pq and N2 = p' q where $p \neq p'$ and publish N1, N2 as public keys)?

It is unsafe because if it is known that the moduli share the same prime than it is possible to calculate the value of q since q = gcd(N1,N2). Even if it is not known that they share the same prime, it is so cheap to calculate the gcd of two numbers, even if they are large, that it is possible to calculate the GCD's of all pairs of 11 million distinct public RSA moduli in less than 2 hours. Once q is figured out it is easy to calculate p and p'. Once those are known it is easy to figure out the values of d from the e's given. This will totally break the encryption.

5. The Imperfect Diffie-Hellman paper mentions that solving the discrete logarithm on 512 bit Diffie-Hellman keys can take a week of computation. What made it possible to solve discrete logarithms quickly enough that a solution could be found within the lifetime of a TLS or IKE handshake (typically minutes, at most)?

The first thing that made it possible was that it was possible to downgrade a TLS to 512 bit prime rather than using a 1024 bit prime. It also exploits the fact that most TLS use the same standard prime numbers given by Oakley so that if it some computations are done on those primes beforehand it becomes much faster to calculate all the logs for those primes. The final thing it exploits is the fact that in the NFS if more time is taken in one of the steps it becomes cheaper to do the other steps. Since a week was spent on the original prime numbers before hand, it became very fast to calculate the individual discrete logs based on the pre-computations.

Problem 2: Long Answer
1. (a) Assuming no fancy attacks on the encryption scheme, what prevents an evil user (C) from impersonating A to B?

C cannot impersonate A to B since B needs to verify by the end of the scheme that A has the shared key that B has. This is done by encrypting a nonce that only B has and making A return a modified version of it. For C to get the shared key though, it needs to either be able to decrypt the response from the server, which is encrypted using A's key, or decrypt the message it sends to B in step 3, which requires the ability to decrypt using B's key. Since C lacks both A and B's keys without doing a fancy attack, C cannot impersonate A sending a message to B.

(b) What are the nonces N_A, N_B needed for?

The Nonces are needed for two seperate purposes. N_a is necessary so that A knows the response they got from the server is still fresh and is the correct response to the request A sent to the server. N_b is necessary to verify that both sides have the key and that they can decrypt and encrypt sucessfully.

(c) Why does the final message encrypt $E_k(N_B - 1)$ rather than $E_k(N_B)$? What attack might be possible if the subtraction was removed, and the final message simply contained $E_k(N_B)$?

The subtraction is necessary in the final message because if it didn't exist then the message B expects back is the exact same one it sends. So all A needs to do is send back the exact value it receives from B. When the subtraction is necessary A must decrpyt the message it receives from B, modify it, and re-encrypt it for B to be able to validate that A has the key that B thinks it has. If A didn't need to validate that it had the same key that B has, it would be possible for C to modify the key that B uses to encrypt when it is sent and to have B trust it. How C modifies the key is dependent on the symmetrical encryption used.

(d) In the second (and third) message, why does the ciphertext $E_{K_{BT}}$ (k, A) encrypt the identity A?

The ciphertext encrypts the identity A so that when B decrypts the third message they can validate that the user who sent the message to them is the user whose shared key is being sent to it. It is another way to prevent C from impersonating A and making B think that its communicating to someone who it isn't. During this step, if the sender to B is not who the message says they are, then B knows that the sender is not who they claim they are.

(e) Imagine that the encryption scheme E is implemented using AES-CTR with no MAC. What attacks could you come up with against this protocol?

The first attack I can think of happens no matter the symmetric encryption used. Since B has no way of way of verifying how recent the message sent in question 3 is, it is possible for an attacker C to send an old version of message 3 that contains a compromised key k and was used in a previous interaction. In that case B will think that the current shared key is the same as the old one and think that it is communicating with A. C will be able to decypt and send the correct modified Nonce since they already know the key.

It is also susceptible to malleability attacks where the attacker generates a key from the server for communication between C and B and then modifies the identity in the encrypted packet it sends to B to say that A sent it.

(f) What happens to this protocol if A is ever compromised and all of its data (keys etc.) are stolen?

If A is ever compromised and all of its data is stolen then it becomes possible for the thief to disguise itself as A in any interaction since it becomes possible to ask the server for a shared key between a recipient and A and decrypt the response since the attacker already has the key.

In addition if A recorded old keys or interactions, or the attacker recorded previous interactions A has had, between it and different people, the attacker would be able to decrypt all those previous interactions.

So if A is every compromised the basis of authentication the scheme is built on disappears.

2) (a) List all of the subgroups of Z^*_{23} and provide one generator for each. See HAC §2.5 for definitions if this is helpful.

{1} generator 1
{1 2 3 4 6 8 9 12 13 16 18} generator 2
{1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22} generator 5
{1 22} generator 22

b) What is the order of the group Z^*_{23}. List the prime factors of the order of the group.

The order of Z^*_{23} is 22 since $|Z^*_{23}| = 22$, since it excludes 23 and 0.

The prime factors of the order of the group are 1, 2, 11, 22

(c) Describe a simple algorithm for solving the discrete logarithm of a value $h \in G$ base g, that is: finding an x such that $g^x \equiv h \bmod p$. Your algorithm can be inefficient.

The simple answer is to continuously raise g to higher and higher powers you arrive at x. The psudo-code is

```
x = 0
while ((g^x)%p != (h)%p) {
        x += 1
}
output x
```

(d) Let p be a prime and let g be a generator of the group Z^*_p , where (p, g) are used for Diffie-Hellman. Imagine that Bob re-uses the same secret key b for many Diffie-Hellman connections. How can Alice learn (at least) one bit of b? Hint: think about how you could do it in your toy p = 23 group above.

If Alice can figure out an a such that g^a is within a proper subgroup of Z^*_p called H, and Alice figures out the ordering of the group starting at g^a then Alice can figure out the last $\log_2(|H|)$ bits of the number. This is because the proper subgroup is cyclic so that it will always repeat in the same order and pattern. For example if Alice can find $g^a \equiv p\text{-}1 \bmod p$ then Alice can find out the last bit of b. That is because $(g^a)^b$ will always remain in the subgroup {1 p-1} so if $g^{ab} \equiv p\text{-}1 \bmod p$, b is odd so the last bit is 1. On

the other hand if $g^{ab} \equiv 1 \bmod p$ then b is even so the last bit is 0. This only works though if Alice can find an a such that g^a is within a proper subgroup of Z^*_p and she iterates through all values of the subgroup.

(e) A trusted central party wants to make and distribute many RSA keypairs that all share a single public modulus N = pq. For each of n users in the system she generates a different public/secret exponent pair $(e_1, d_1),(e_2, d_2), \ldots ,(e_n, d_n)$ that work with N, and sends each exponent pair to a party so that the i^{th} party's public key is (N, e_i). What is the risk of this system?

Well first of all if p and q are every figured out than every single pair can be obtained. Another problem is that since p and q stay the same, every pair must have a different e value so eventually a small e will need to be used, which as we saw had problems.

The other big risk of the system is that if an attacker gets one of the pairs (e_i, d_i), especially if one of the two values is small, then it is easier to calculate (p-1)(q-1) since the attacker knows that $e_i*d_i \bmod (p-1)(q-1) \equiv 1$. So they know that $(e_i*d_i - 1)$ is a multiple of (p-1)(q-1). So the attack knows that for all a in Z^*_p $a^{(e_i*d_i-1)} \equiv 1 \bmod p$. Let $e_i*d_i - 1 = 2^s t$ where t is an odd integer. It can be proven that for at least half of all a in Z^*_p, there is an i in [1, s] such that $a^{2^{(i-1)}*t}$ is not congruent to +-1 mod n but $a^{2^{(i)}*t}$ is congruent to 1. For any one of these a,i pairs, $\gcd(a^{2^{(i-1)}*t}, p)$ is a non-trivial factor of n. So if e_i and d_i are known to an attacker, they just need to guess values of a, which only averages about 2 tries, until they arrive at one that matches the conditions. So it is trivial to factor p and q here.