ISA 562 Assignment 4   Mithilaesh Jayakumar (G01206238)

1. In mod n arithmetic why does x have a multiplicative inverse if and only if x is relatively prime to n?

The multiplicative inverse of x (written x^-1) is the number by which you'd multiply x to get 1.

Two numbers are said to be Relatively prime if they do not share any common factors other than 1.

Now if x and n are relatively prime then,

gcd(x,n)=1

Suppose we want to find a multiplicative inverse of x mod n. This means we want to find a number u such that ux = 1 mod n. Another way of saying this is that ux differs from 1 by a multiple of n, so there is an integer v such that ux + vn = 1 as gcd(x,n)=ux + vn by Extended Euclid Theorem

Let us consider an example.

Consider 9 and 10. Both are relatively prime.

gcd(9,10)=1

The multiplicative inverse of 9 mod 10 is 9 as 9x9 = 1 mod n. Therefore, 9x9 + (-8)x10 = 1

If x and n are not relatively prime then,

gcd(x,n)=1.k where k is a positive integer greater than 1.

Therefore, k=ux + vn which implies that there can never exist a multiplicative inverse for x.

2.In section §6.4.2 Defenses Against Man-in-the-Middle Attack, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

The problem about Man-in-the-Middle attack is that both sides are not confident about the other side's public key. If they were sure that they have the correct public key of their friend Man-in-the-Middle attack wouldn't be possible, because MITM attack is based on the forgery of public keys. Moreover, as we sign the DH elements with our private key after encrypting them even if the attacker tries to encrypt whatever message he wants using this public key he would not be able to authenticate himself as a valid sender of the message as he does not have the private key of the original sender.

3.Suppose we have the encoding that enables Carol to mount the cube root attack (see §6.3.5.2 The Cube Root Problem). If Carol sends a message to Bob, supposedly signed by you, will there be anything suspicious and noticeable about the signed message, so that with very little additional computation Bob can detect the forgery? Is there anything Carol can do to make her messages less suspicious?

When we use RSA to sign a message, we supply it a block input that contains a message digest. The PKCS1.5 standard formats that block as:

00h 01h ffh ffh ... ffh ffh 00h ASN.1 GOOP HASH

As intended, the ffh bytes in that block expand to fill the whole block, producing a "right-justified" hash (the last byte of the hash is the last byte of the message).

A common implementation flaw with RSA verifiers is that they verify signatures by "decrypting" them (cubing them modulo the public exponent) and then "parsing" them by looking for 00h 01h ... ffh 00h ASN.1 HASH.

which means the following operation is performed

D = EncryptedSignature ** e mod N

if e is equal to 3, it's quite possible that EncryptedSignature ** 3 ends up being smaller than N, therefore the modulo operation does not change the result. So, if Carol forges a block that satisfies only the conditions we know the system checks for, and also corresponds to a perfect cube, she can pass the cube root as a signature to such a verification system, and it will be accepted as valid.

If Bob doesn't check the padding, he leaves open the possibility that instead of hundreds of ffh bytes, he has only a few, which if he thinks about it means there could be squizzilions of possible numbers that could produce a valid-looking signature. To implement verification safely, instead of trying to parse the encoded signature, generate what you expect the signature to look like, complete of padding, ASN.1 and hash, and then simply compare the user/attacker's one. This can be done by finding a number that when cubed (a) doesn't wrap the modulus (thus bypassing the key entirely) and (b) produces a block that starts "00h 01h ffh ... 00h ASN.1 HASH".

4.In DSS, other than saving users the trouble of calculating their own p, q, and g, why is there an efficiency gain if the value of p, q, and g are constant, determined in the specification?

Since choosing a DSS <p, q, g> triple is computationally expensive, it is likely that many people will base their own key on parameters that have been published. With RSA, an attacker that breaks a key breaks only a single key. With DSS, an attacker that breaks a <p, q, g> triple breaks all keys upon which it is based which becomes even more difficult.

Also DSS is about a hundred times slower for signature verification than RSA with e = 3. In terms of the other operations, DSS is much faster for key generation. DSS has the advantage that some of the signature computation can be precomputed before seeing the message. Incase of applications like smart cards, this ability to precompute for signatures makes DSS superior, since a human will not need to wait as long when logging into a system.

5.Suppose Fred sees your RSA signature on m1 and on m2 (i.e. he sees m_1^d mod n and m_2^d mod n). How does he compute the signature on each of m_1^j mod n (for positive integer j), m_1^{-1} mod n, m1·m2 mod n, and in general $m_1^j \cdot m_2^k$ mod n (for arbitrary integers j and k)?

Let the RSA signature for m1 be m1^d mod n
Let the RSA signature for m2 be m2^d mod n
Same RSA signature on m1^j mod n for any positive integer j can be computed by :
= (m1^d)^j mod n
= (m1^j)^d mod n
Same RSA signature on m1^(-1) mod n for any positive integer j can be computed by :
= (m1^(-1))^d mod n
= (m1^(-d)) mod n
= (m1^(d)^(-1) mod n
As per Euclidean Algorithm, m1^(d)^(-1) mod n is multiplicative inverse of m1^d mod n

RSA Signature for m1.m2 mod n can be computed by:
= (m1*m2) ^ d mod n
= ((m1^d)mod n)*(m2 ^ d) mod n
RSA signature for m1^j.m2^k mod n can be calculated by:
= ((m1^j)*(m2 ^k)) ^ d mod n
According to product rule
= ((m1^j)*(m2 ^k)) ^ d mod n
= (m1^j)^d mod n * (m1^k)^d mod n
= (m1^d)^j mod n * (m2^d)^k mod n