Devin Ross

4/14/2024

Cryptography and Network Security

Dr. Yener

Cryptographically Secure Bank: A Write-up

Encryption: The Asymmetric Scheme

When deciding on what type of cryptographic scheme to use for the encryption of messages sent across the socket, we weighed a few different approaches. First was the obvious scheme for consideration: RSA. It is simple to implement, understand, and modify for our specific needs. Though, it also comes with the drawbacks of being relatively slow and having large key size. There was also ElGamal, which was a similar candidate to RSA, but using the discrete logarithm problem instead of large integer factorization. This is important because it is going to change the positives and negatives of both schemes, with regard to speed, efficiency., and more. Considering that under experimentation “RSA… performs better as regards to energy usage, time complexity and space complexity,” and “ElGamal performs better in terms of time complexity during decryption process,” (“A Note on Time and Space Complexity of RSA and ElGamal Cryptographic Algorithms”, Emmanuel, Aderemi E, Marion O, Emmanuel) we thought RSA to be the better candidate for our project. There are many factors, but it boiled down to the simplicity and average speed gain that RSA showed to make it just that little bit better than ElGamal. It is also worth mentioning that it is very simple to make RSA semantically secure with a simple change to the algorithm. Though, this is not where public key cryptographic schemes end; we also had to consider elliptic curve cryptography. This scheme would allow us to have much smaller keys than RSA with a similar amount of security. However, this does not come without the

complexity that ECC adds to the Ultimately, RSA was the right choice because of its security, its simplicity, and its malleability (though not in regard to homomorphism).

We can now start with the implementation of the RSA scheme. We know the private keys – $p,\ q,\ d$ – and the public keys – $n,\ e$ – of a given person (or entity), so the simple RSA scheme is simple.

For encryption:

$ctxt = (r^e \bmod n,\ M \oplus H(r)) = (c_1, c_2)$ where r is a random binary string of length 1024 (for good security)

For decryption:

$ptxt = c_2 \oplus H(c_1^{\ d}) = M$

What this does, is first calculates what r is. This is done by the simple decryption process of RSA — raising the ciphertext to the modular inverse of the public key e all $mod\ n$. Then, because the encrypted part of the ciphertext actually is the random number r that we chose prior, this allows us to calculate the hash of this number, and xor it with c_2, which reverses the xor, thus yielding M.

Though, this scheme is not finished, as we need a way to verify that the message has not been tampered with. This is where a message authentication code is used. The details of the HMAC algorithm will be fleshed out later, but for now, it will act as a black-box hashing algorithm. Once we get the ciphertext from the encryption, and before we send it over the network, we will use an Encrypt-Then-MAC approach. This is where we will first encrypt the plaintext, obtaining the ciphertext, then we will use the hash of the ciphertext + the random number to get the MAC. We will then send this along with the rest of the ciphertext, and when received this MAC will be recalculated and checked, making sure that the message has

not been tampered with. The same process will be done for the symmetric scheme to continue this security after the initial key's are generated and distributed.

Encryption: Message Authentication Codes and Hashes

As mentioned before, we need a hashing algorithm for many steps of the process, including the MACs and in the SSH implementation. Choosing between the MD_ family and the SHA family, in my opinion, was pretty simple. First, was the choice of how secure does this application need to be. Because of the limited amount of time, people will have to brute force a collision, along with the requirement that this collision has to be viable. This creates the situation where I think using a less secure version of these families is permissible.

Now the conversation of SHA1 vs. MD5 begins. Although both are deprecated and broken, it is worth mentioning that SHA1 is "less" broken, and has been cracked much more recently. This makes it just *that* much more secure than MD5, which, for a very small speed and complexity difference between the two, is a good reason to select SHA1.

Then, the MAC algorithm we selected was HMAC. This is due to it's proven security, versatility, ability to work with SHA1, and it's ubiquity – which brought ample resources and guidance when implementing it. The HMAC algorithm I implemented creates two fixed values which are appended intermittently to the hash. We can see this right before the calls to the SHA1 algorithm. It starts by making sure the message size is viable for the key, then calculates both the *ipad* and *opad* which are the values that will be prepended to the message before it is put through the hash function. These values will incorporate the key to the hash function, obfuscate it because of the inner and outer layer they represent, and with the two calls to the SHA1 hash function, create a secure one-way function for the MAC.

It is worth mentioning that the strength of HMAC comes from it's underlying hash algorithm. So this means that because I chose to implement SHA1, we are bottle necked a little by it's security. Though, for the reasons illustrated prior, this seems to be a non-issue.

Encryption: The Symmetric Scheme

For the symmetric scheme, the choice was actually very straight forward. The options started with DES and AES, and comparing them meant first knowing how DES would function. Because DES is not secure enough without running it multiple times, making it 3-DES, the speed will drastically suffer. This, in comparison to AES, makes the choice simple: the only way to go is AES.

To start the implementation of AES, a path very similar to DES is layed out: make s_boxes, which will define the transformations of the plaintext, create functions that will shift rows, add the effect of the key, and change the columns. The beginning, finding the s_boxes, is nothing more than looking up the NIST standard, and copying it down for us to use (https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf). Once we have the boxes for the encryption, and its inverse for the decryption, we have simple functions that will use those boxes in conjunction with the current state of the text. The shifting rows and adding the key were similarly simple; the rows were once again a defined switch, and adding the key just involved xoring the part of the key that correspond with the same part of the current text. The challenge was the mixing of the columns. The way to do it involves solving for a set of linear equations against a known matrix. This involved implementing Galois multiplication for efficient multiplication within a finite field, in specific the $x^8 + x^4 + x^3 + x + 1$ field for our AES algorithm. The steps for this multiplication, a little simplified, are a bitwise multiplication followed by a reduction. This is what keeps the

multiplication fast: the constant ability to reduce the current value by the modulus of the field.

With all of these processes: adding the round key, mixing columns through solving a matrix equation with Galois fields, simply mixing the rows and using a substitution matrix to bring non-linearity to the process, creates a secure symmetric algorithm.

Next, is the same thing we did for RSA: making every encryption different and adding a MAC and a way to make sure the message was not tampered with. In this case, I used a random initialization vector to make a change to the plaintext, that made sure every encryption would be different. The random vector would be xor'd with the totality of the plaintext and then encrypted. To make sure the plaintext would be recoverable, I made sure to send the random vector as a parameter of the ciphertext, only giving information if you are able to decrypt the ciphertext. Then, to make sure that the message wasn't tampered with, I added a MAC; this consisted of the ciphertext in addition to the random vector being hashed together to form a number that would only be found with the specific encryption and initialization vector.

Encryption: The Whole Story

Our encryption uses a public key cryptosystem: RSA, a symmetric cryptosystem: AES, a hashing function: SHA1, and the MAC function: HMAC.

RSA is used to get the key's from Diffie-Hellman across an untrusted network to another party (the client) without the ability for an adversary to tamper with the message or be able to read it.

AES is then used as a faster encryption scheme once the key's have been dispersed. This allows for quick communication, with a similar, if not greater, amont of security.

The HMAC-SHA1 hashing algorithm and MAC creation is used to make sure that messages are what they say they are and come from the right place.

All of these systems work together to make a system that is efficient, secure, and ensures both parties that communications make it to where they need to go.