Samad Farooqui

Cryptography White Hat Writeup

DES/3DES

When the connections are fully established, the messages themselves need to be secured with an established key that the people communicated agreed on, and the messages themselves need to be secured so an adversary listening in cannot make out what is being sent along. In order to encrypt the messages being sent along the network, the project uses a Triple Data Encryption Standard algorithm (3DES) that was constructed using a Data Encryption Standard algorithm (DES). The 3DES was constructed using the DES that was implemented from homework 1 after being improved upon some more. The project only required a toy DES be used for the passing along of encrypted messages, but the toy DES was improved upon in order to adhere to better security practices.

The toy DES that was originally implemented followed the same algorithm as a normal DES. A key *K* was broken up into two parts *K1* and *K2* after a multitude of steps, including an initial permutation, a split, left shifting of the different parts, a permutation on one part, and a permutation on the other part after some more left shifting when the key *K* had been broken up into bits. Using the constructed *K1* and *K2*, a plaintext is taken and permuted. The plaintext is broken up into two parts *L* and *R*, where the *L* part is not changed and the *R* part goes through a function involving *K1*. The function permutes the given part of the plaintext, XOR's the permuted string with the key, and then computes a result half the size of the given part after an S-Box is used to construct the result and the result is permuted more. The *L* part and modified *R* part are then XOR'd to create *L2*, while the modified *R* then is not changed. *L2* is then put through the function with *K2*. The result is XOR'd with the *R* value, and an XOR of the result from the last XOR and the last item to go through the function is computed. That result goes through a permutation that takes two arguments, being the result of the last XOR and the result from the first use of the function, and that is what the cipher is. This is repeated for every block of the original message. The decryption works in a same way, but the keys *K1* and *K2* switch place in every way.

The toy DES described had been originally implemented with the given message being split into blocks and each block being encrypted with the one key. However, for the sake of the project, this toy DES implementation was improved upon. The toy DES now takes 160-bit keys as opposed to the original 10-bits that was required from the homework. These 160-bits are used in 10-bit blocks, on different blocks of the plaintext so that brute force attacks would take much longer than

opposed to if just one key is used. This method of encryption was then used in order to create a 3DES which is ultimately used in the project. The algorithm follows a standard 3DES method; first, three keys are generated from the original 160-bit key, *K1, K2,* and *K3*. Then, the improved DES encrypts the plaintext with *K1* as the key used. This encrypted message is then decrypted with *K2*, to ultimately have that result be encrypted with *K3*. The decryption works in a similar fashion; the plaintext is constructed by first decrypting the given ciphertext with *K3*, then encrypting it with *K2,* to ultimately decrypt it with *K1*. With this 3DES, the project is more secure than it would have been with just 3DES. As the encryption is now 3DES, it is even harder for an adversary to perform a brute force attack as three keys need to be known. If the project were to be done again, an even greater key size would be used and the functions and S-boxes would be improved upon so 10-bits are not what is just encrypting each block. Another algorithm such as AES could also be used instead so that attacks that affect 3DES and DES would not be an issue. Ultimately, the encryption is used to pass messages back and forth, and is more secure than it would be if just toy DES was used.

HMAC

In order to keep data integrity of the messages that were signed, it is important that some kind of Message Authentication Code be added to the end of encrypted messages (MAC). Adding a MAC to the end of the message and having a receiver verify the MAC allows for the message to ensure that the original message is not tampered with - if the MAC produced is not the same, then the message has been tampered with. For this project, a MAC was added to the end of messages. The MAC was generated through the HMAC algorithm, utilizing SHA-1.

The HMAC had to be created after the SHA-1 was created and working. HMAC normally takes, as input, the key *K* and a message *m*. The implementation that was used also took the expected block size and the output size. Some implementations also have the function to be used for hashing as input. After these parameters are passed in, the parameter *K* is first checked to see if its length is larger than the block size passed in. If it is, the key is then hashed to math the block size length. Afterward, the key is compared to see if it less than the length of the block size. If this condition returns true, then the key *K* is then padded with zeros on the right to ensure that it is the correct length. After the key *K* is correctly modified, the constants used for the HMAC algorithm are then defined. Two constants are defined, being the inner pad (ipad) and outer pad (opad). The ipad and opad are constructed using *K*; the ipad is the result of an XOR operation between the key and the constant 0x36 (in hexadecimal) after 0x36 is multiplied by the block size. The opad is constructed the same way,

where a constant 0x5c (in hexadecimal) is XOR'd with the key *K* after 0x5c is multiplied by the block size, similar to the ipad constant.

With these constants defined, the actual HMAC can now be constructed. The ipad constant is first concatenated with the message. The result that comes from this operation is then hashed by the hash function to form an internal "envelope". Afterward, the opad constant is concatenated with this "envelope" and then hashed again to return what our HMAC will be. To verify that an HMAC actually works, the encrypted message is sent along with the HMAC concatenated at the end. A message receiver would then decrypt the message and use the HMAC algorithm to generate their own HMAC. If the HMAC generated does not match the HMAC that was sent along, then the user can assume that the original message has been modified and should disconnect with that user.

The project implements the described HMAC algorithm with SHA-1 as the hash function. Some design choices were made in order to make computation easier. In order to make computation easier, all the data types were converted to bits as Python can implement this easily and the XOR operations can also be computed seamlessly. Otherwise, the implementation is the same as the process described above. It is all wrapped up into one function, and that function is then used by the program when messages are being sent to each other. If this project were to be done again, then the key would be given a longer length to not be as susceptible to brute force attacks. Overall, in order to make the project secure, SHA-1 would not be used as it is a deprecated function, and HMACs are susceptible to birthday attacks and brute force attacks as much as their hashing algorithm makes them.