CSCI- 4230/6230: Cryptography & Network Security I

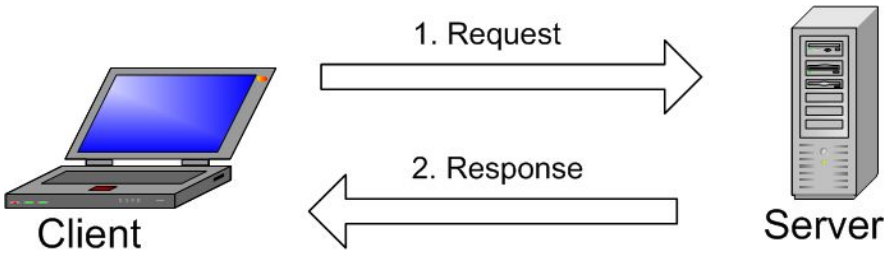
SSL/SSH Final Project

Due: December 3, 2018


## Team 9: Adam Rabinowitz, Jordan Charest, Colin Goldberg


Prefix: the code can be found at https://github.com/adamrab123/SSL-SSH

# *Part 0: Program Overview*



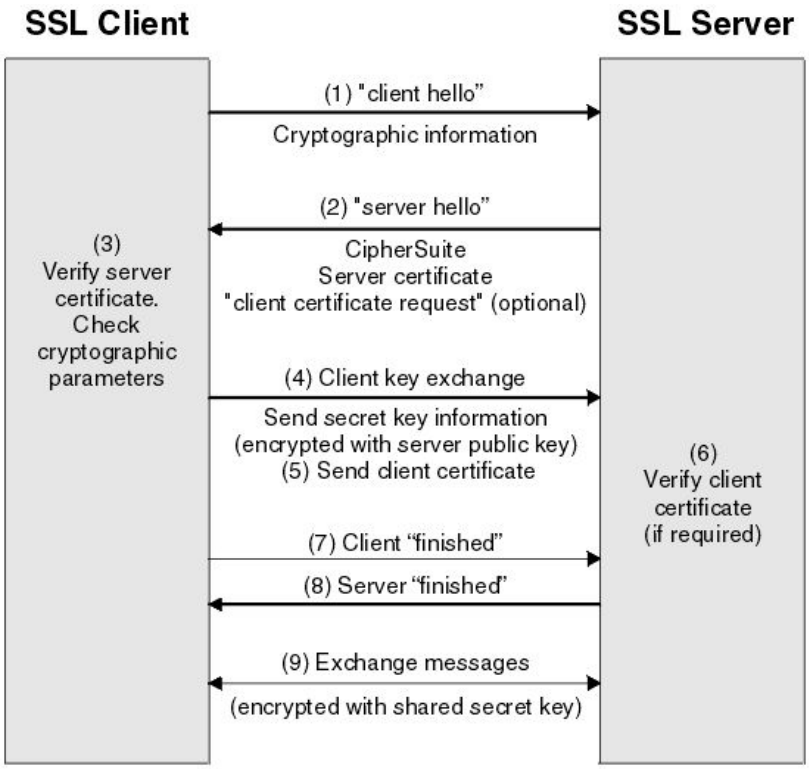

The purpose of this project is to provide secure connections, enabling two parties to communicate with privacy and data integrity. This project implements a simplified version of the Transport Layer Security (TLS) protocol, which evolved from that of the Secure Sockets Layer (SSL). The primary goals of both protocols are to provide authentication, confidentiality and data integrity. We have followed the TLS 1.3 protocol as closely as possible to maintain maximum security, with the only minor exception being our use of 3DES instead of AES.

The system operates as follows:

1. The server is initiated; it waits for a client/user to connect to it
2. A client can connect to the server by using the correct host and port
3. The client initiates the handshake protocol by sending their supported ciphersuite (key exchange protocols, encryption/decryption protocols, and message authentication protocols (in our case just HMAC-SHA1)).
4. The server picks randomly which key exchange protocol to use, which encryption/decryption standard to use, and HMAC-SHA1 for message authentication
    a. The reason for randomly choosing each level is to add security; an attacker can't predict which protocol we will use, and so it is harder to plan an attack.
5. Once the server has chosen each respective protocols, it initiates the key exchange with the client; the client clearly cooperates with the server
6. With the key exchange complete, both the server and the client should have an established shared master key to use for further encryption
7. With the key established, the client can now use the server-chosen encryption protocol with the created key to send an encrypted message to the server
8. The server now acts as a modified echo server: it decrypts the message, reverses it, encrypts it, and sends the encrypted reversed message back; the client now knows that the connection is secure

# *Part 1: Handshake protocol*

The SSL protocol for establishing which cipher suite capabilities to use looks like the following:




1. The client sends a "Client hello" message to the server with its supported cipher suites.
2. The server responds to the client by sending a "Server hello" message back
3. The server then sends its certificate to the client for authentication. The server sends the "Server hello done" message.
4. If the server has requested a certificate from the client, the client sends it.
5. The client creates a random Pre-Master Secret and encrypts it with the public key from the server's certificate, sending the encrypted Pre-Master Secret to the server.
6. The server receives the Pre-Master Secret. The server and client each generate the Master Secret and session keys based on the Pre-Master Secret.

7. The client sends "Change cipher spec" notification to the server to indicate that the client will snow tart using the new session keys for hashing and encrypting messages. The client also sends "Client finished" message.
8. The server receives "Change cipher spec" and switches its record layer security state to symmetric encryption using the session keys. The server sends "Server finished" message to the client.
9. Client and server can now exchange application data over the secured channel they have established. All messages sent from client to server and from server to client are encrypted using session key.

# *Part 2: Key Exchange*

## Computational Diffie-Hellman

The following steps are used for this procedure:

1. The server and the client agree to a modulus p = 13737952841431760672478349284808155572028307509187082819551913151516707086175459719982927438730232802701541906393603319156393007805258235602146755615434165972529705443791452641725617104569734022511865376231508075706196248812499437556738930092274740008066069466577356895098514540509512774261104430041947606048 3 and base = 17

2. The server chooses a secret integer a (in our case a random 1024-bit number). The server then computes $A = g^a \bmod p$ and sends it to the client
3. The client chooses a secret integer a (again a random 1024-bit number). The client then computes $B = g^b \bmod p$ and sends it to the server
4. The server now computes $S = B^a$ and the client computes $S = A^b$. This will be the shared key the server and the client use from now on. To see why the two are the same, refer to the explanation below):

   From properties of modulus, $B^a = A^b$. This is because $A^b \bmod p = g^{ab} \bmod p = g^{ba} \bmod p = B^a \bmod p$. In other words, $(g^a \bmod p)^b \bmod p = (g^b \bmod p)^a \bmod p$. It should also be noted that only a,b, and $g^{ab} \bmod p = g^{ba} \bmod p$ are kept secret. All the other values (p,g, $g^a \bmod p$, $g^b \bmod p$ ), are public. This is an example of the discrete logarithm problem.

## Elliptic-Curve Diffie–Hellman

To add to the security of our system and to keep as close as possible with the TLS 1.3 standard, we have also implemented Elliptic-curve Diffie–Hellman (ECDH) as another anonymous key agreement protocol. The overall structure of ECDH is similar to DH but has a few nuanced differences. It works as follows:

Context: Alice wants to establish a shared key with Bob, but they are afraid that a third party is eavesdropping on their only available channel. What do they do?

Before the procedure begins, domain parameters must be agreed upon. We used the prime case, and so our parameters are p,a,b,G,n,h (p is a prime that defines the elliptic curve; a and b are the parameters in the function of the elliptic curve $y^2 = x^3+ax^2+b$; G is the generator of the cyclic subgroup; n is the size of the subgroup; h is the cofactor).

Each party must also have a key pair suitable for ECC ( a private key d which is randomly selected in the interval (1,n-1) and a public key Q (Q = dG which means adding G to itself d times). Finally, each party must know the other party's public key prior to the execution of the protocol. We will let Alice's key pair be $(d_A,Q_A)$ and Bob's key pair be $(d_B,Q_B)$. The steps to excahnged keys are as follows:

1. Alice computes point $(x_k,y_k)=d_AQ_B$ and Bob computes point $(x_k,y_k)=d_BQ_A$
    a. The shared secret is $x_k$ (the x coordinate of the point).
2. Both parties can now compute the shared secret because $d_AQ_B=d_Ad_BG=d_Bd_AG=d_BQ_A$

Security Analysis:

The only information that Alice initially exposesabout her private key is her public key. That means that unless a 3rd party can solve the elliptic curve discrete logarithm problem, they can’t determine Alice's private key. The same can be said about Bob's private key. Further, no party other than Alice or Bob can compute the shared secret, unless that party can solve the elliptic curve Diffie–Hellman problem.

## Signing and verifying our messsages

The idea behind blind signing messages: Alice wants Bob to sign message m, but doesn't want Bob to know the contents of m. Alice blinds the message m with a random number blinding factor) b resulting in blind(m,b). Bob signs the message, resulting in sign(blind(m,b),d), where d is Bob's private key. Alice unblinds the message using b, resulting in unblind( sign( blind(m,b) ,d) ,b) which reduces to sign(m,d).

We chose to use RSA blind signatures. These work like the following:
While a traditional RSA signature is computed by raising the message m to the secret exponent d modulo the public modulus N, the blind version uses a random value r (note that r is relatively prime to N (i.e. gcd(r, N) = 1)).

We then raise r to the public exponent e modulo N, and the resulting value $r^e$ mod N is used as a blinding factor. The author (in our case the client) of the message computes the product of the message and blinding factor like this: $m' \equiv mr^e \bmod N$. They then send this value to the signing authority (in our case the server). It should be noted that since r is random, $r^e$ is also random. This means that m' does not leak any information about m.

Now the server calculates the blinded signature s' the following: $s' \equiv (m')^d \bmod N$. Now they send this back to the client. The client can remove the blinding factor and get s, the RSA signature of m ($s \equiv s' * r^{-1} \bmod N$).

The above property works because RSA satisdies the equation $r^{ed} = r \pmod N$ and thus we get the following: $s \equiv s' * r^{-1} \equiv (m')^d r^{-1} \equiv m^d r^{ed} r^{-1} \equiv m^d r r^{-1} \equiv m^d \pmod N$. Thus we know for sure that s is the signature of m.

## RSA Blinding Attack

On a holistic level, RSA is subject to the RSA blinding attack through which it is possible to be tricked into decrypting a message by blind signing another message. This happens because of the following: the signing process is equivalent to decrypting with the signer's secret key, which means that an attacker can provide a blinded version of a message m encrypted with the signer's public key m' for them to sign.

However, this isn't a problem in our cryptosystem! We only sign and verify our messages at the handshake protocol stage, which means that if an adversary was to obtain the original message m, it would be a hashed value.

Furthermore, we also prevent a man in the middle attack. This happens when we verify the public key. This means that signing it makes sure no one is faking just having the public key. This leads to even more security!

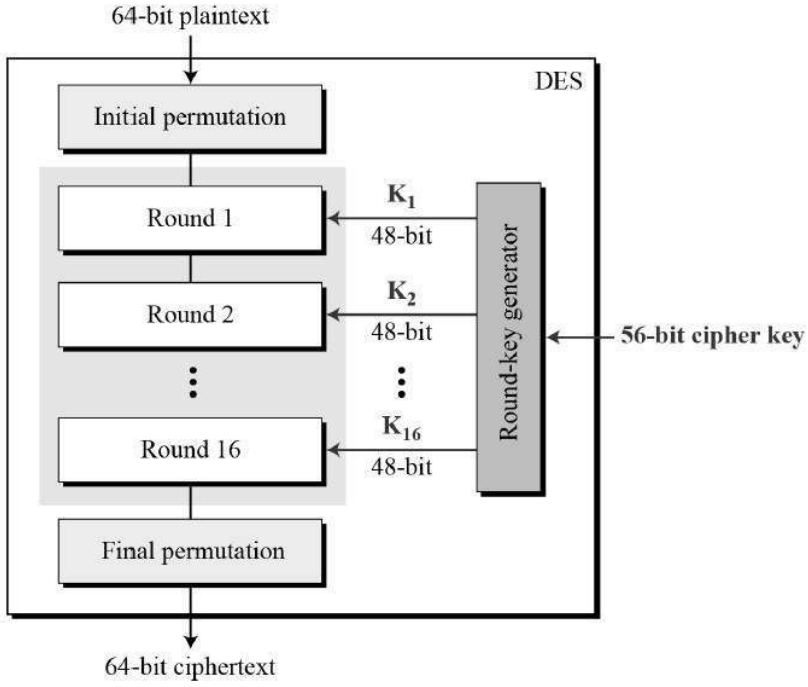
JORDAN PLEASE DO THIS

# Part 3: Message encryption/decryption

## Triples-DES(Triple Encryption Standard/3DES)

### DES(Data Encryption Standard)

DES is a product block cipher in which 16 or rounds of substitution and permutations are cascaded. The block size is 64 bits. The key size is 56 bits and 8 parity bits for a total of a block size of 64-bit.




Each round of DES encryption or decryption looks like the following:

1. At each individual stage of the encryption/decryption process, the cipher output from the preceding stage is partitioned into the 32 left-most bits($L_i$), and the 32 right-most bits($R_i$).
2. Ri is transposed to become the left-hand part of the next higher intermediate cipher, $L_{i+1}$. The right-hand half of the next cipher, $R_{i+1}$, however, is a complex function, $L_i + f(Ri, K_{i+1})$, of a subset of the key bits, $K_{i+1}$, and of the entire preceding intermediate cipher.

3. DES uses tabulated functions known as s boxes. These lead to the following property of DES: $f(A) + f(B) \neq f(A + B)$. This non-linear substitution is a big security pillar in DES
4. NOTE: steps 1-3 are repeated 16 times.

DES follows the structure of a Feistel cipher:

1. Each iteration involves the cipher output from the preceding step being divided in half
2. The halves are then transposed with a complex function controlled by the key being performed on the right half and the result combined with the left half using the "exclusive-or" from logic to form the new right half,

A key attribute about Feistel ciphers is that if the key subsets are used in reverse order, repeating the "encryption" decrypts a ciphertext to recover the plaintext.

## Turning it more secure! (3DES)

While DES cipher's key size of 56 bits was generally sufficient when that algorithm was designed, increasing computational power made brute-force attacks feasible. The easiest solution to increase security is to increase the key size, but that doesn't protect the user from a differential cryptography attack. Triple DES allows us to increase the "key size" to protect against such attacks, without the need to design a completely new block cipher algorithm.

Before diving into the nuances of 3DES, let's first discuss why 2DES wouldn't quite work: the idea behind 2DES would be the same as 3DES: use two keys (k1,k2) instead of one, and encrypt each block twice: $E_{k2}(E_{k1}(\text{plaintext}))$.

The issue with this implementation is that it is vulnerable to a meet-in-the-middle attack, which loses the perceived security of a key of length 2n: given a known plaintext pair (x, y), such

that $y = E_{k2}(E_{k1}(x))$, one can recover the key pair (k1,k2) in about $2^n$ steps, instead of the expected about $2^{2n}$ steps one would expect from algorithm with 2n bits of key.

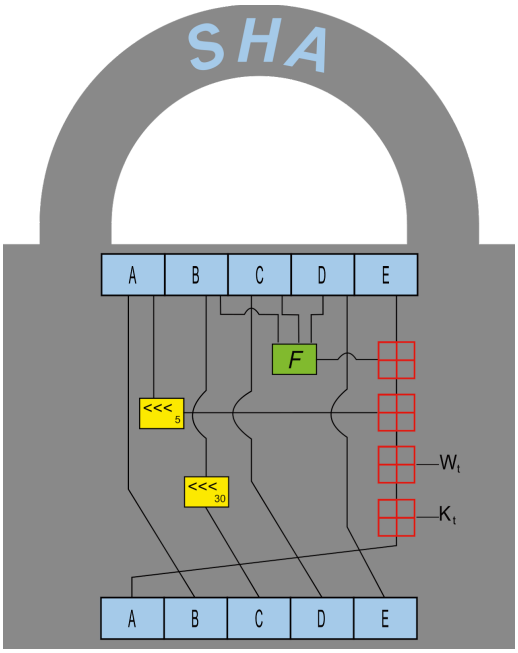
To counter this potential attack, 3DES uses an encrypt-decrypt-encrypt method with keys k1, k2, and k3. Encryption looks like the following: ciphertext = $E_{k3}(D_{k2}(E_{k1}(\text{plaintext})))$. As can be expected, decryption is the reverse: plaintext = $D_{k1}(E_{k2}(D_{k3}(\text{ciphertext})))$.

## AES(Advanced Encryption Standard)

The typical TLS 1.3 standard would require AES. We opted to write TripleDES to allow for code reuse from previous assignments in this class, and to save time. For future improvements, an upgrade to AES is very likely.

# *Part 4: Message authentication*

To authenticate messages, we use HMAC-SHA1 for all messages that are sent after establishing a private key.

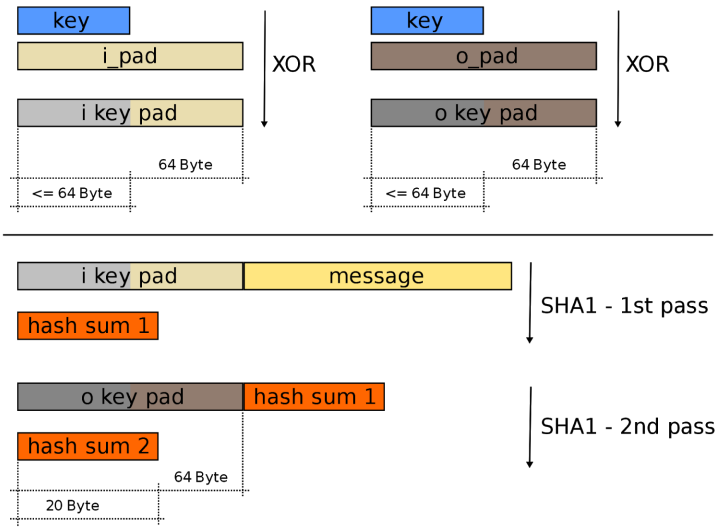



First, a quick overview of the SHA-1 hashing algorithm. This, like all cryptographically secure hashing algorithm, must have properties making it suitable for cryptographic purposes. Namely, this hashing function must be a suitable one-way function. This means that creating a hash from any given input is a fairly trivial process, and can be computed efficiently by any system. However, when attempting to go from a hash function to an original message, this must be computationally very difficult for a system. If this weren't the case, pieces of information could be recovered from the hash, making it insecure.

Creating a cryptographic hashing algorithm is incredibly important for this cryptosystem, as it gives a way to verify messages without providing the entire message. We use these hashes multiple times in our protocol: during the public key signing step, as well as when calculating HMACs.

HMAC is a MAC, or Message Authentication Code, based around a hashing algorithm. When sending any messages, we attach a HMAC to them to verify their integrity, as well as their owner. The HMAC is calculated by creating a hash based on the message to be encrypted, as well as the private key which both parties share. Because both parties can obtain the decrypted message, and both parties have the same key, the HMAC can be calculated the same way by both the client and the server. If both generate matching HMACs, the message has been verified, and thus can be trusted.

What problems does an appended HMAC fix? Firstly, it verifies integrity of any data being sent. For example, say a portion of an encrypted message was corrupted in communication, or tampered with by an adversary before being received. Because the MAC has hashed the decrypted message, any modification to this message will result in a different hash, and an unvalidated message. This guarantees that any data will be exactly what's sent. Secondly, it establishes the authentication of the user. Since it includes the private session key in the message, no adversary could properly create a MAC for any message without its corresponding key. This means that if an adversary were to send data, even if it could be properly decrypted, the recipient would still know that it wasn't real data because it wasn't validated using its key. This ensures that only users with knowledge of the specific session key are allowed to send messages, authenticating all users.




## *Part 5: References*


- Elliptic Curve Cryptography:
  - https://tools.ietf.org/html/rfc7748

- Elliptic Curve Diffie-Hellman:
    - https://en.m.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman?fbclid=IwAR3_BxnMekQZHdfkVZXb9iq_qkZM5H5iTmkxwrITWHl4sCeoBJ0wxjteiqs
    - List of elliptic curves: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
- Diffie Hellman additional numbers:
    - https://tools.ietf.org/html/rfc5114#section-2.1
- Handshake Protocol:
    - https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10660_.htm
- DES:
    - https://www.britannica.com/topic/Data-Encryption-Standard
    - http://page.math.tu-berlin.de/~kant/teaching/hess/krypto-ws2006/des.htm
    - https://en.wikipedia.org/wiki/DES_supplementary_material
- 3DES:
    - https://en.wikipedia.org/wiki/Triple_DES#Keying_options
- TLS:
    - https://tools.ietf.org/html/rfc5246
    - https://tools.ietf.org/html/rfc8446
    - https://en.wikipedia.org/wiki/Transport_Layer_Security
- HMAC:
    - https://en.wikipedia.org/wiki/HMAC
    - https://tools.ietf.org/html/rfc2104
- SHA-1
    - https://en.wikipedia.org/wiki/SHA-1