Andrew Kiselik

Cryptography White Hat Writeup

Handshake

The handshake follows the following process, which is similar to that of SSH/SSL and the newer TLS protocol:

1. The client sends “Client Hello”, client’s random session value, & supported cipher suites to server
2. Server responds with “Server Hello”, chosen protocol, server’s random session value, certificate. The server sends “Server Hello Done.”
3. Client p, q to Server (encrypted with server’s public key from the certificate).
4. The key is now shared between both parties after completion of RSA.
5. The client sends “Change Cipher Spec” notification to the server to begin use of session key for hashing/encrypting messages. Ends with a finished message that contains an HMAC over the previous message.
6. The server sends “Change Cipher Spec” and switches to chosen encryption protocol using session key. Ends with encrypted “Server Finished” message to the client.
7. Secure connection established. Messages are encrypted using the session key.

To get into more specifics, the protocol is determined randomly and then the agreement is made off of the random choice between Blum-Goldwasser or 3DES. In order to allow for the Blum-Goldwasser encryption, both p and q are required, therefore both are generated randomly as Gaussian primes (primes that are congruent to 3 mod 4) and can be multiplied to form n for Blum-Goldwasser or the general key for 3DES. RSA was selected to share the key because it allows for the guarantee that we know the key is a Gaussian prime, rather than computational Diffie-Hellman where there isn’t even a guarantee that the number will be prime. After the exchanging of keys, the client sends a message to the server appended with a HMAC. Because of the implementation of SHA1, we know that the HMAC will always be 40 characters long, therefore to verify the HMAC, you take everything except the last 40 characters and then compute the HMAC and ensure that the value is equivalent to the last 40 characters of the message. The server completes this verification and then sends back a message which has an appended HMAC and then is encrypted using the appropriate algorithm. After the client decrypts the server’s message and verifies the HMAC using the same method as described previously, the secure connection has been established and the client and server and now communicating using the encrypted algorithm chosen with HMACs appended to the

messages.

Blum-Goldwasser

Blum-Goldwasser operates as a modification from our homework 3 implementations. The Blum-Goldwasser algorithm is semantically secure based on the Blum Blum Shub portion of the algorithm makes the hardness of guessing the bits along with the key sufficient to provide semantic security. A message is given and then it is encrypted using n, which is used to calculate k, h, and X0, all of which are used in the encryption process.

Encrypt follows the following form:

- *n* is the public key calculated from *p*q*
- $k = \lfloor \log_2(n) \rfloor$ and $h = \lfloor \log_2(k) \rfloor$
- *m* is the message broken up into *t* blocks of binary, each of length *h*
    - In our implementation, to correct for an undefined behavior, we append 0s to the message to ensure that the last block (*t*) is of size h. Otherwise, when decrypting it provides an undefined behavior, which means that the output of the algorithm may not be consistent with the originally encrypted message.
- X0 is generated from random seed *r* which is between 1 and *n*
- Blum Blum Shub is completed and then x_{t+1} is calculated and appended to produce the final ciphertext.

A message is decrypted using *p, q, a, b,* or *c* where *a* and *b* are calculated using the greatest common divisor.

- Like in the encrypt, c is the ciphertext in *t* blocks or length *h* bits
    - The last block (*t*) has the extra 0s which were appended to them in the encrypt function, but it doesn't change the output of the message, rather just allows the algorithm to operate in the correct manner.
- $d1 = ((p+1)/4)t+1 \bmod (p-1)$
- $d2 = ((q+1)/4)t+1 \bmod (q-1)$
- $u = xd1^{t+1} \bmod p$
- $V = xd2^{t+1} \bmod q$
- X0 = vap + ubq mod n
- Do the Blum Blum Shub again, producing m as the plaintext

A problem that was encountered during the implementation of the Blum-Goldwasser algorithm was when verifying the HMAC. Because BG doesn't require that the number of bits divides exactly into h, the algorithm had an undefined behavior that filled bits in automatically, thus producing a slightly different output on the HMAC because the message got cut off when decrypting. In order to solve this

problem, 0s were appended to the end of the text to ensure that the undefined behavior was treated as no additional information, creating no changes. After padding the end with 0s, the undefined behavior was fixed and allowed the HMACs to verify, thus allowing the application to work as intended.

Socket Programming

For the assignment, socket programming was necessary as SSH and SSL are both networking protocols. We used the python socket package, which allows us to easily transfer data over the network, in our case the localhost. We created a basic chat server application, which after completing the handshake allows users to communicate back and forth between the server and the client. The chat server encrypts on each end and then decrypts and verifies the HMAC using out algorithms, but uses socket programming to send the message in an otherwise unencrypted nature. The important functions that are used to implement this application are recv and sendall, which respectively allow information that is being transferred using socket programming to be received and sent. Because recv takes in a size, in bits, the number that is used had to be increased to allow for long messages, however, the general maximum message length is 15360 bits, which provides for sufficiently long messages to be received. Conversely, using sendall allows for messages of any length to be sent, however in the application, they may not always be received if the recv function isn't used to handle a message of the length being sent. In general, the application uses localhost or ip 127.0.0.1 on port 9898 to operate.