Jessup Barrueco, Samad Farooqui, Andrew Kiselik

Cryptography White Hat Writeup

SHA-1

SHA stands for Secure Hash Algorithm, specified in August 2015 by the National Institute of Standards and Technology (NIST). SHA-1 is able to claim *security* because it is computationally infeasible to

1) find a message that corresponds to a given message digest
2) find two different messages that produce the same message digest

SHA-1 has two main stages as described by NIST: preprocessing and hash computation. In preprocessing, the message is padded, parsed into *m*-bit blocks, and initialization values are set for the next part, hash computation. In hash computation, a *message schedule* is generated which is used along with functions ROTL(), Ch(), Parity() and Maj() to iteratively generate a series of hash values. SHA-1 has the following limitations:

Message Size (bits):	$< 2^{64}$
Block Size (bits):	512
Word Size (bits):	32
Message Digest Size (bits):	160

The functions integral to the implementation of SHA-1 are described as follows:
ROTL(x) or Rotate Left performs a circular left shift operation, where *x* is a *w*-bit word and *n* is an integer with $0 \le n \le w$

ROTL(x) = ((x << n) | (x >> (w-n)))

The logical functions Ch(), Parity() and Maj() all operate on three 32-bit words *x, y,* and *z* and produce a 32-bit word as output. SHA-1 iterates 80 times per message block. As *t* iterates from 0 to 79:

$Ch(x, y, z) = ((x \& y) \oplus (\neg x \& z))$	when $0 \le t \le 19$
$Parity(x, y, z) = (x \oplus y \oplus z)$	when $20 \le t \le 39$ *and* when $60 \le t \le 79$
$Maj(x, y, z) = ((x \& y) \oplus (x \& z) \oplus (y \& z))$	when $40 \le t \le 59$

SHA-1 also relies on a sequence of 80 constant 32-bit words, $K = \{ K_i : 0 \le i \le 79 \}$ denoted by the following hex values:

5a827999	$0 \le i \le 19$
6ed9eba1	$20 \le i \le 39$
8f1bbcdc	$40 \le i \le 59$
ca62c1d6	$60 \le i \le 79$

SHA-1 Preprocessing has 3 main steps:

1. Padding the Message: the padded message must be a multiple of 512 bits. For an *l*-bit message M, the bit “1” is appended to the end of M, followed by *k*

"0"-bits, where *k* is the smallest, non-negative solution to the equation *l + k +* 1 = 448 mod 512. Then, a 64-bit block equal to the number *l* expressed in binary.

2. Parsing the message into *N* 512-bit blocks, where the length of the padded message is equal to N*512.
3. Setting the Initial Hash Value H[] which will consist of the following five 32-bit words, in hex:
   H[0] = 67452301
   H[1] = efcdab89
   H[2] = 98badcfe
   H[3] = 1032546
   H[4] = c3d2elf0

The SHA-1 Hash Computation uses the functions and constants previously defined. Each message block is processed in order, and the final 160-bit message digest is the concatenation of the result of those operations on each element in H[].

The biggest challenge in our implementation of SHA-1 was computing the correct value of *k*, rather, the number of "0"-bits to append while padding the message in the Preprocessing stage. The line of code in Python3 which finally accomplished this problem for messages of any length is:

k = (((447 - (len(x)*x_bitlength)) % 512) + 512) % 512

Another challenge I experienced in SHA-1 implementation pertained to the technical differences between Java and Python3. All of my homework assignments up until this project were implemented in Java, and I was accustomed to explicit variable type declaration. I experienced difficulty at the beginning, incorrectly assigning the hex values as string variables and then encountering type disagreements in the logical functions described above. In the end, the issue boiled down to using extra print statements for debugging, but I found this challenge unique to my experience switching into a Python3 development environment from Java. However, one of Python3's clear advantages is its command line interface, which allowed me to test my functions quickly with toy variables and get to the root of the issue in a speedier and more efficient manner than I had experienced with Java.

RSA

In order to implement Textbook RSA in Python3, we first had to program Euclid's GCD algorithm and Euclid's Extended Algorithm for Multiplicative Inverses.

> gcd(a,b) takes two integers a, b and returns (g, x, y) such that
    a*x + b*y = gcd(a,b) = g

> multInv(a, b) takes two integers (a, b), uses gcd() to verify coprimality, and returns the multiplicative inverse of a modulo b.

The RSA key generation function keyGen(p, q) requires two primes p, q, so we also needed to implement a function to verify primality, called isPrime(). If both p, q pass this verification step, the public key variable n = p*q is calculated along with Euler's totient function phi(n) = (p-1)(q-1). Our public encryption key *e* is determined by a while loop which randomly generates an integer *e* between 1 and phi(n), checks if gcd(e, phi(n))=1, and if not, restarts the process. Euclid's Extended Algorithm to find Multiplicative Inverses is then used in order to find the decryption private key variable d = multInv(e, phi(n)). The keyGen(p, q) algorithm returns the public encryption key (e, n) and the private decryption key (d, p, q). The ciphertext C is generated and then decrypted with the following equations:

$$C = M^e \quad | \quad M = C^d$$

The biggest challenge in implementing RSA in Python3 was how to handle the incredibly large integers resulting from the encryption and decryption operations. The python3 operations which proved most efficient are as follows:

```
cipher = [pow(ord(char),e,n) for char in plaintext]
plain = [chr(pow(char, d, n)) for char in ciphertext]
```

The function we use in RSA to verify primality of p and q is potentially inefficient for very large inputs, as it performs a brute force check.

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.

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.