Secure Sports Betting

MSCS 630

Anthony Connolly

May 15 2022

Abstract:

I explain the need for encyption of betting data and hashing of passwords for databases in mobile betting applications. I present SHA-1 and AES as viable candidates for doing each respectively. I explain how each algorithm works. I explain how I implemented AES encryption and part of the functions required for AES decryption and utalized the REACT crypto-es package for AES decryption and SHA-1 hashing. I explain the backend solution of my application and the current limitations it causes.

Introduction: Sports betting services using mobile applications or websites provides new opportunities for prospective sport bettors but also may expose users to new risks. A person may not wish access to previous bets, so those must also be made private. We will explore how existing mobile betting applications keep a user's account and profile secure so that we may begin to build a similar, secure application that emulates sports betting. We will also explore how we can encrypt and decrypt a users betting data and store it locally.

For an implementation of this project as a consumer product, I would avoid storing data locally and use something like Firebase, which has it's own encryption system that hashes and salts passwords. For this reason, I will be storing account and bet data locally and exploring effective ways to encrypt both types of data.

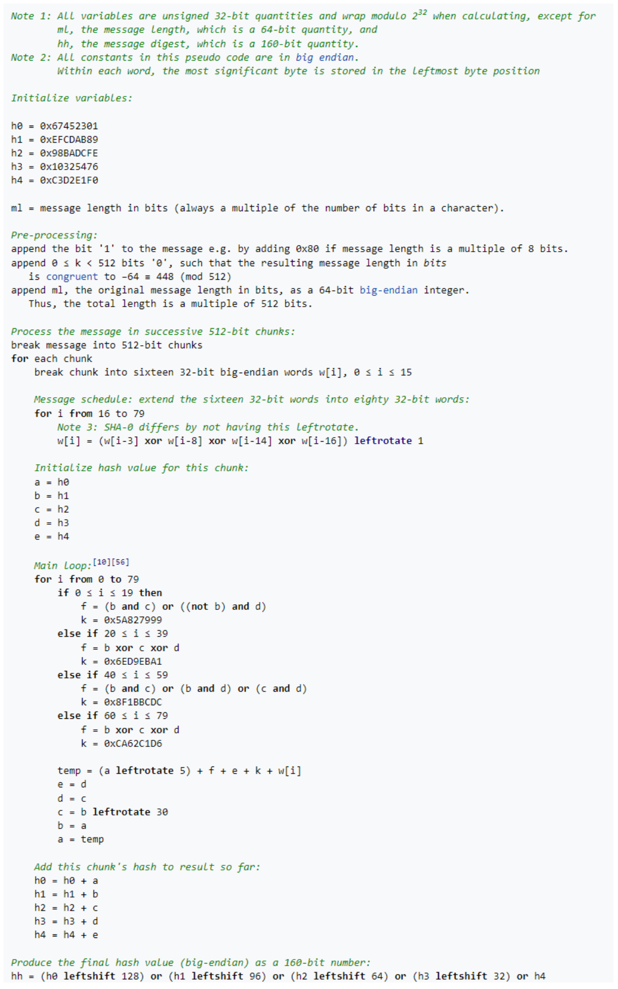
Background:

Two types of data must be encrypted/hashed for this application, the password for storage in the database and the betting data. Typically, passwords stored in a local database should be hashed and salted. Hashing and salting passwords correctly prevent two users from having the same hashed password stored in a database, thus also helping to prevent hash-table attacks. (Arias) Some examples of hashing algorithms used for hashing and salting passwords are SHA 1, SHA 2, bcypt, etc…. For my application it would be sufficient to implement SHA 1 hashing and to salt the password for storage.

The first step of SHA 1 is to add leading zeros until we reach a length of 32 bytes and ensuring that the message to be encoded is a multiple of 512 bits using other preprocessing. We also have word variables h0 through h4. We then perform the following on all of these 512 bit chunks:

The algorithm will split our message into 16 words. These 16 words are extended in 80 words by iterating over the words, performing xors with other words in the list and left shifting, and adding the resultant into the list such that it is used in further expansion.

For each of the 80 words, depending on its index we perform some combination of binary logic using b, c, and d to set a value for the variable f. K is set to a set value depending on the word number. The values of a, b, c, d, e, and f are then all reset where a, b,d, and e are set to the previous letter and c is set to b after a left rotation of 30. We add a - e to h0 - h4 respectively. Finally, we perform a left shift on each of h0-h4 and or them together for our final hased result.



```
Note 1: All variables are unsigned 32-bit quantities and wrap modulo 2^32 when calculating, except for
        ml, the message length, which is a 64-bit quantity, and
        hh, the message digest, which is a 160-bit quantity.
Note 2: All constants in this pseudo code are in big endian.
        Within each word, the most significant byte is stored in the leftmost byte position

Initialize variables:

h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

ml = message length in bits (always a multiple of the number of bits in a character).

Pre-processing:
append the bit '1' to the message e.g. by adding 0x80 if message length is a multiple of 8 bits.
append 0 ≤ k < 512 bits '0', such that the resulting message length in bits
   is congruent to −64 ≡ 448 (mod 512)
append ml, the original message length in bits, as a 64-bit big-endian integer.
   Thus, the total length is a multiple of 512 bits.

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i], 0 ≤ i ≤ 15

    Message schedule: extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        Note 3: SHA-0 differs by not having this leftrotate.
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:[10][56]
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian) as a 160-bit number:
hh = (h0 leftshift 128) or (h1 leftshift 96) or (h2 leftshift 64) or (h3 leftshift 32) or h4
```

Fig 1. SHA -1 Pseudocode (SHA-1)

Salting a password simply requires we add some random word to our original message.

For betting data we must encrypt plaintext since all the necessary data for the bet can be converted to and from plaintext. Some examples of such algorithms are AES, Blowfish, and RSA. (Simplilearn) AES at 128 bits will be used for encrypting our betting data. In an implementation of such a betting application for created for a large group of consumers, we would much more likely seek to deploy this application in a client server architecture and store data server side.

Since AES is a block encryption algorithm our data must be essentially modified into a byte array. We first use a first use a randomly generated or selected key to generate a key schedule from on key schedule. We then use each generated key for the AES rounds which consist of the steps, substitute bytes (where we used a predefined substitution matrix), shift rows (index moved to the right row number times), mix columns (multipling matrix over a galois field) and finally add the corresponding round key. The final round skips the mix columns step.

For AES decryption, we essentially leverage the inverse operations in a different order. This order becomes, inverse row shift (index moved to the left row number times), inverse subbytes (using matrix where the solution bytes become the key in a sense and vice versa) , add corresponding round key (reverse order of encryption), and inverse mix columns (where we multiply over a different gallois field. Round 10 has us inverse shift rows, inverse sub bytes and add the first key. (Daemen)

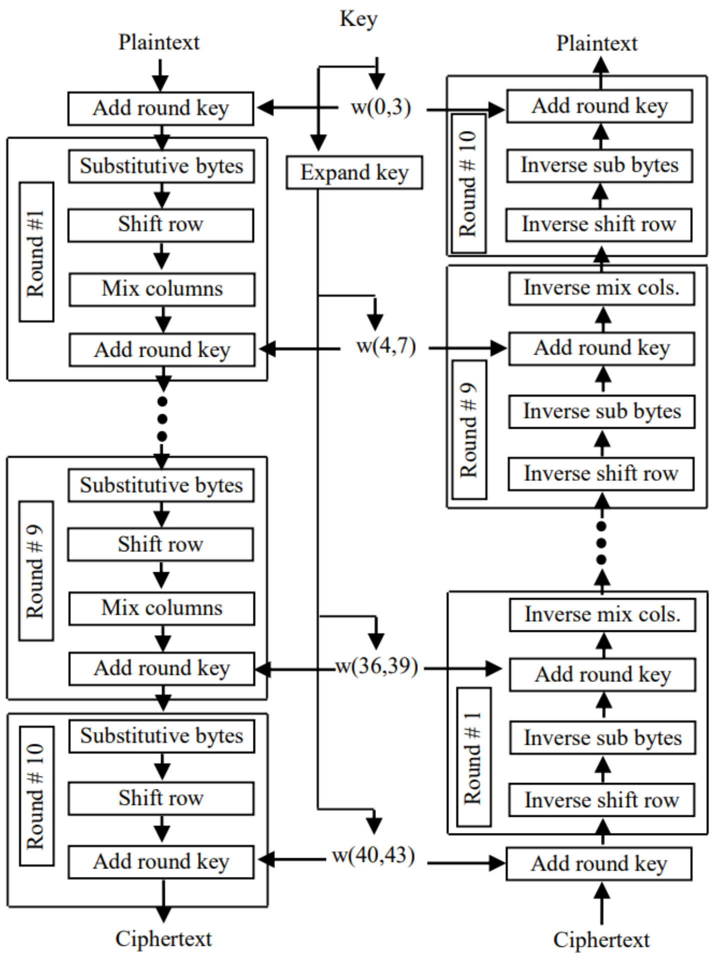


Fig 2. AES encryption and decryption steps (Wadday)

Methodology:

Previously for the milestone, I was able to completely create the front end of the application in React and planned to implement, SHA 1 for password hashing and the AES encryption algorithm for user data. The first step I looked towards in this process was parsing the user data for account creation, login, and betting. This was implemented for all using the useState() react hooks. Next, I had to implement a backend to store the data. The original plan was to use sqlite for this. An issue I ran into was that I built the application using React with Expo which does not work with sqlite. For this reason, I decided to use Asyncronous storages for this implementation. We lose all semblance of a structured local database in favor of a simple key value backend. Data is still preserved using Asynconous storage.

One limitation with this current implementation of that we may only store one bet. For this example, I wanted to provide a proof of concept so one I only programmed one game a user can bet on. If we were to expand the selection of bets, we would need some logic structure to store bets asynchronously and name the keys differently in some way. Right now, we may only save one bet with the key “bet”.

Next, I aimed to tackle AES encyption for the bets. The first step of this was to standardize the data into a string that we can encrypt and also parse data from easily when decrypting. I decided to do so by making it a string of bet amount followed by team name. I needed to pad the length to 32 with zeros. This is an easy format to parse data from upon decryption since we will have integers (bet) followed by characters (team) followed by zeros(padding).

As for the AES algorithm, we split this into encryption and decryption. The easiest way to tackle both is to create functions to manage the key schedule, and functions to manage the algorithm. We then uses these functions to put it all together for an encyption or decryption function. For encryption, we build the aesRoundKeys to generate a key schedule from a starting key, we use an AES stateXor function, a nibble substitution function, a shift rows function, and a mix rows function.

Similarly for decryption, since AES decryption functions as essentially as a reverse of encryption, we most have functions that perform matrix operations that are the inverse of AES encryption functions such as inverse sub byte, inverse shift rows and so on. I was able to implement most of these function successfully up until the inversion mix columns function. For this reason, I began to leverage the CryptoEs package of React to complete a working model of the vision of my application. I used the built in AES decryption function to convert the encrypted data into usable data to present to the user.

Some limitations with this implementation include the following. Currently, I am not randomly generating a key for the AES algorithm and have one hard coded for this example. In a full realise this would be essential for security. Another possible vulnerability is that the key used is stored using Async storage, similar to the data. Ideally, we would want to have this key stored in some other place than the data for security.

Finally, we need to implement hashing and salting with SHA-1. Similarly to AES decrypt, I leveraged Crypto-ES’s built in hashing functions to do this. I also used the package to randomly generate our salt. I simply generated the salt and added the data string and salt string together before hashing. When the user attempts to login, we compare the attempted password + stored salt hashed using SHA-1 to the real password + stored stal hashed with SHA-1.

Conclusion:

The application build along side this paper has successfully found utalization of AES for encrypting bets and SHA-1 for hashing and salting passwords. AES and SHA-1 are certainly steps towards building a more secure betting application. In my research, I have found that SHA-1 is essentially broken and it’s usage is strongly recommended against. For this reason, future iterations of this or a similar app would probably best seek to implement a more advanced or secure hashing algorithm.

References:

Arias, Dan. “Adding Salt to Hashing: A Better Way to Store Passwords.” *Auth0 - Blog*, 25 Feb. 2021,
auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords.

Daemen, Joan, and Vincent Rijmen. “The Block Cipher Rijndael.” Lecture Notes in Computer Science, 2000, pp. 277–284., https://doi.org/10.1007/10721064_26.

Simplilearn. “What Is Data Encryption: Algorithms, Methods and Techniques [2022 Edition]: Simplilearn.” *Simplilearn.com*, Simplilearn, 4 Mar. 2022, https://www.simplilearn.com/data-encryption-methods-article.

“SHA-1.” *Wikipedia*, Wikimedia Foundation, 29 Apr. 2022,

https://en.wikipedia.org/wiki/SHA-1.

Wadday, Ahmed ‘Study of WiMAX Based Communication Channel Effects on the Ciphered Image Using MAES Algorithm’. International Journal of Applied Engineering Research 13 (04 2018): n. pag. Print.