A Replication of an Online University Application Database System

By Aaryaman Sagar and Aditya Sule

XII-A, Sanskriti School, New Delhi

CONTENTS

Introduction

Online applications often have a plethora of activity occurring behind the scenes to make it easy to use and understand yet hard to break and hack. These Databases make use of the most advanced researched techniques in cryptography and computer security to enhance the security of their application system.

We have attempted to make a small scaled replica of an application system using the above mentioned methodology and created an application system on the basis of how we have imagined its functioning would be like.

Passwords

A 'Password' is often a secret word or string of characters that is used for authentication to prove identity, or for access approval to gain access to a resource (in this case the application database).

Storing passwords is quite a daunting task for computer programmers primarily because it has quite a lot of security issues that need to be dealt with before the password finds its way into storage on a server. But storing a password on the server is dangerous for the user as it exposes a flaw in security for the user. Most users often have the same password for many online database systems. So for example if the passwords are stored as follows

User Account	Password
aditya123@email.com	Password123
aaryaman@email.com	Passcode123
johndoe@email.com	Johnspassword
joebloggs@email.com	joespassword
…	…
…	…

This is insecure because if a hacker gains access to the server he can directly access the passwords for each respective account. Thus he or can then use it to login as the user in the respective database or in other databases used by the user if the passwords are the same.

This brings us to the topic of "Hash Functions" and how they will be used in this project.

Hash Functions

A hash function is any algorithm or subroutine that maps large data sets of variable length to smaller data sets of a fixed length. For example, a person's name, having a variable length, could be hashed to a single integer. The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes.

$$\#: A \rightarrow B$$

Where # is the hash function, x is an element of set 'A' and 'B' is the range of the hash function, which in most cases has much fewer elements than 'A'.

Hash Functions find a variety of applications, such as in storing passwords, digital signatures and Hash tables. For the purposes of our application we will be considering its uses only to verify a digital signature and to store passwords.

Cryptographic Hash Functions need to satisfy the following conditions

- It is easy to compute the hash value for any given message.
- It is infeasible to generate a message that has a given hash.
- It is infeasible to modify a message without changing the hash.
- It is infeasible to find two different messages with the same hash.

Thus in essence cryptographic functions can be treated as one way functions which are impossible to invert.

Using Hash Functions to store passwords

Using simple encryption to store passwords on the server often makes use of a key to encrypt the password which is stored on the server itself. So if the hacker gains access to the server, he or she will in tunrn gain access to the key and can use that key to decrypt the password. A better solution to store passwords in a database is to store the hash of the password instead of the password itself.

So instead of just storing the password along with the username the table in the server now becomes

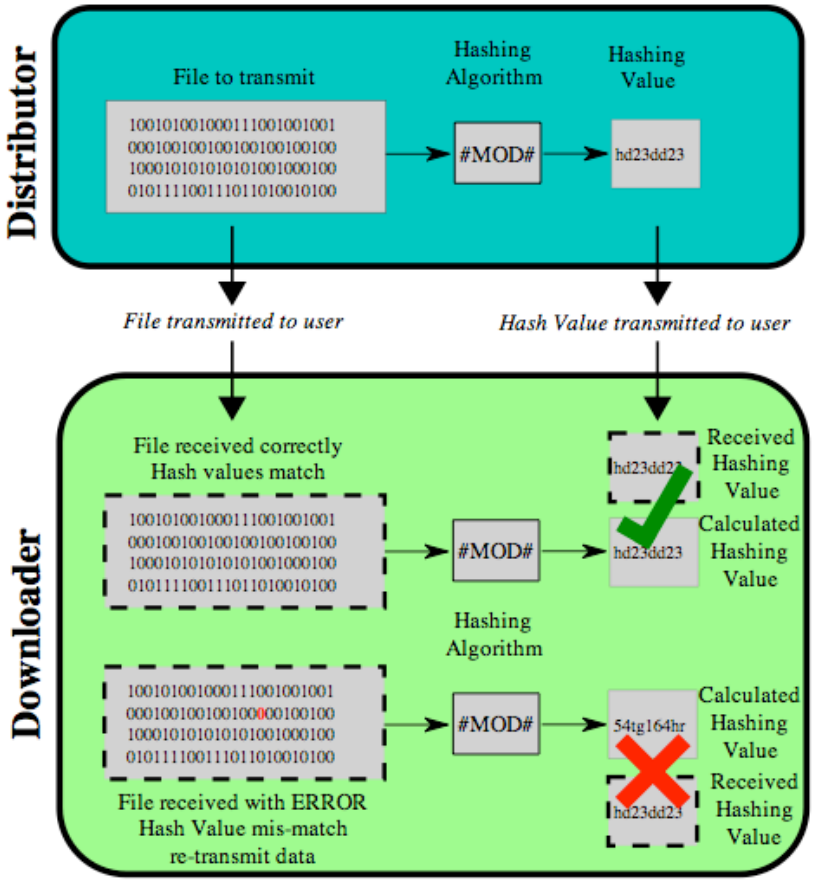
User Account	Password
aditya123@email.com	5baa6c9d3f55ab30
aaryaman@email.com	7ef6928fdb5ac7362
johndoe@email.com	de6284756919bc9a9
joebloggs@email.com	2764bc7a82ef90392
…	…
…	…

So, if a hacker gains access to the server he or she will only get access to the hash digest of the password and since a hash function is a one-way function, will not get the value of the password from the hash digest.

Using Hash functions as Digital Signatures

A digital signature or digital signature scheme is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that a known sender created the message such that they cannot deny sending it and that the message was not altered in transit. Digital signatures are commonly used for software distribution, financial transactions, and in other cases where it is

important to detect forgery or tampering. In our case we will be using it to verify that the document received was the one sent as the application.



Picture showing the flow of control in verifying a digital signature

NOTE The Avalanche effect
In cryptography, the avalanche effect refers to a desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions. The avalanche effect is evident if, when an input is changed slightly (for example, flipping a single bit) the output changes significantly. For example
#(000) = ab45683fec829
#(001) = 7389ea0c9b92

Before we indulge into the explanation of the hash function we have made for this project we need to explain certain concepts that have to be used to understand the working of our hash function, primarily that of a pseudorandom number generator (from page 12**********)

File Storage and Management

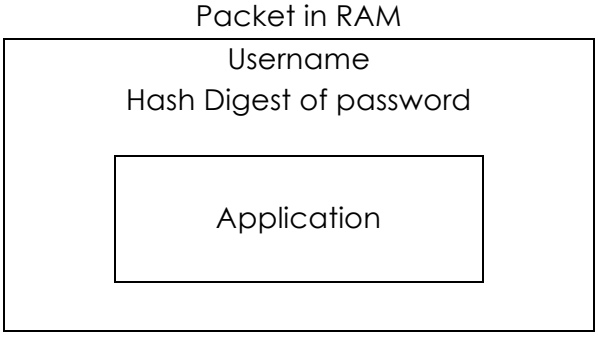
There are 3 files that have been used in the project – "saved.dat", "submitted.dat" and "managers.dat".

The file "saved.dat" stores the application packet (i.e. the username, the hash of the user password and the application) and is the one which is used to modify and create new accounts and new applications. This file is the incomplpete data.

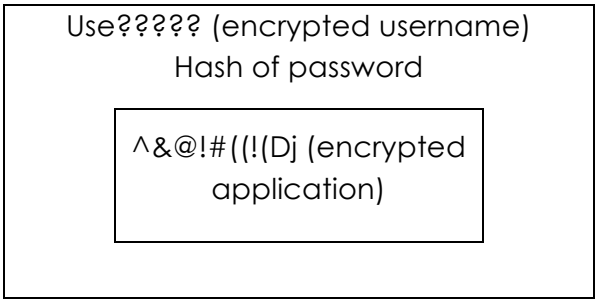
"managers.dat" contains manager packets (i.e. the manager username, the hash of the manager's password and the personal details of the manager). This file represents the server through which access is given to the university officials to check the application.

"submitted.dat" contains objects of the class 'SubmittedPacket' (which contains the application and the hash of the application that is used as a verification to check if the data in the application is different from what the hash shows, or if the data has been modified or corrupted). Managers gain access to view the applications present in this file. So, this file represents the stored and submitted applications.

saved.dat



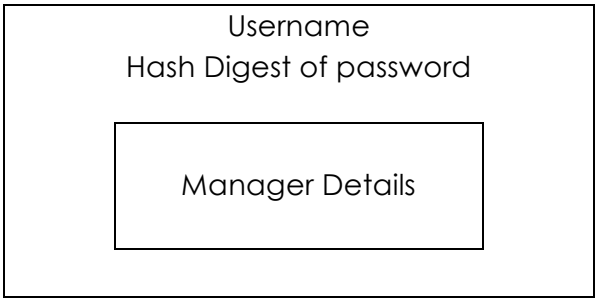
Packet in "saved.dat" after modification



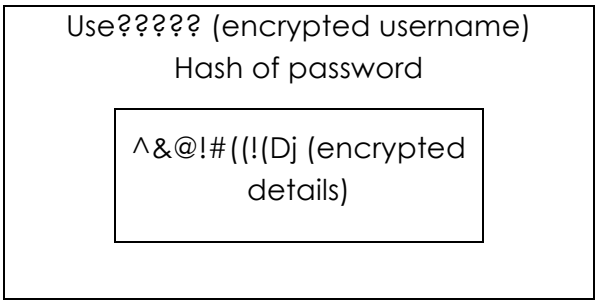
The packets that are to be modified are transferred to RAM by decrypting the username and application with the system password (qwerty) then matching it with the required username and then modifying the application. If the user saves and exits the reverse process is followed, i.e. the username and the application are encrypted and then stored. In case the user needs to change the password, the program replaces the Hash of the password in the file "saved.dat" with the hash digest of the new password.

managers.dat

Manager Packet in RAM

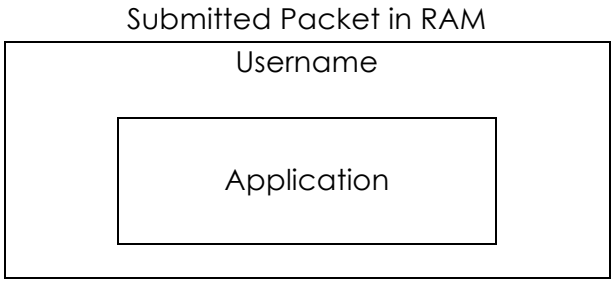


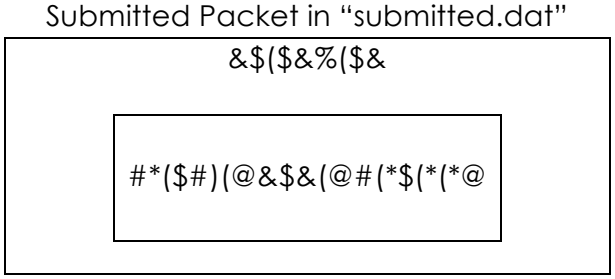
Packet in "managers.dat" after modification



The same procedure is followed for the manager packet, except the application in the applicant packet is replaced by the manager's details. No password changing capabilities are provided for the manager.

submitted.dat





The entire packet is encrypted before it is transferred to the “submitted.dat” file and the modifications take place in the same way (as mentioned for the saved packet in “saved.dat”) with the system password (qwerty).

Concepts (Number-Theory)

Modular Arithmetic

In mathematics, modular arithmetic (sometimes called clock arithmetic) is a system of arithmetic for integers, where numbers "wrap around" upon reaching a certain value—the modulus.

Let m ≥ 1 be an integer. We say that integers '*a*' and '*b*' are *congruent modulo* m if their difference a-b is divisible by m. We write

$a \equiv b$ (mod m)

to indicate that *a* and *b* are congruent modulo m. The number m is called the *modulus*.

Fields of Numbers

Definition The ring of a number is defined as the set of all the whole numbers less than it.

Definition The set (Z/mZ) is defined as all the integers in the ring of 'm'

(Z/mZ)* = {a ∈ Z/mZ : gcd(a,m) = 1} ;

where gcd(a,b) is the greatest common divisor of a and b

The set (Z/mZ)* is called the group of units modulo m.

Definition Euler's phi function (also sometimes known as Euler's totient function) is the function

Φ (m) = #(Z/mZ)* = #{$0 \leq a \leq m$: gcd(a,m) = 1}

Fields of Prime Numbers

Let a prime be *p*

Inference The set $(Z/pZ)^*$ is the group of units modulo p. This set therefore has all the whole numbers for which gcd(a,p) = 1. Therefore:

$(Z/pZ)^*$ = {1, 2, 3, 4, ……., p-1}

Definition The order of an element of (Z/pZ) is the number after which the powers modulo m start repeating their values.

Theorem (Fermat's Little Theorem) Let p be a prime number and 'a' be any integer. Then

$a^{p-1} \equiv 1 \pmod{p}$ $(a/p) \notin I$

$a^{p-1} \equiv 0 \pmod{p}$ $(a/p) \in I$

Theorem (Primitive Root Theorem) Let *p* be a prime number. Then there exists an element $g \in (Z/pZ)^*$, whose powers give every element of $(Z/pZ)^*$, i.e.,

$$(Z/pZ)^* = \{1, g, g^2, g^3, ..., g^{p-2}\}.$$

Elements with this property are called primitive roots of (Z/pZ) or generators of $(Z/pZ)^*$. They are elements of $(Z/pZ)^*$ having order p – 1.

Discrete log problem The Discrete log problem is defined as the problem of solving $a^x \equiv b \pmod{p}$ where 'a' and 'b' are known.

Although algorithms to solve the discrete log problem exist. They are very slow and therefore, the 'Discrete log problem' is considered one of the most difficult equations to solve in the field of mathematical cryptography and is used in various key exchanges.

Ψ The Pseudorandom number generator

Pseudorandom number generators are often used in cryptography to obtain a string of numbers or bits whose origin cannot be determined or which look random for someone who does not know the seed ('k' in this case).

The $\Psi(k)$ function

$$\Psi(k,\lambda,p) = \lambda^k \pmod p$$

$$x_n = \left(a^n \bmod p\right)^k \bmod p$$

OR

$$x_n = \Psi\left(k,\left(a^n \bmod p\right),p\right)$$

where 'a' is a primitive root of 'p', a preset prime and 'n' is the number of terms to be added to from the sequence.

$$Domain : Z \times \left((Z/pZ)*\right) \times Z,\ \ Range : (Z/pZ)*$$

Why the function works

The basic requisite for the $\Psi(k)$ is that the function required is very hard to inverse, i.e. finding $\Psi(k)$ becomes very difficult if k is not known.

$$\text{If } \Psi(k,\lambda,p) = \lambda^k \pmod p = \eta$$

i.e. 'η' is the pseudorandom stream generated using Ψ.

The inverse of the $\Psi(k,\lambda,p)$ function is

$$\Psi^{-1}(\eta,\lambda,p) = k = \log_\lambda \eta \pmod p$$

This function is another representation of the "Discrete log problem" as has been described in the theory above. Which is very difficult to solve.

Note The purpose of keeping the limits of the summation in the range $[1, p-1) \cap I$ follows from Fermat's little theorem, as $a^{p-1} \equiv 1 (\mathrm{mod}\, p)$.

Note We are using 'a' as the generator of the field of prime number 'p'. In other words 'a' is a primitive root of 'p'.

Ω The Hash Function

The Algorithm

Ψ is the Pseudorandom number generator

$\Psi : A \rightarrow A$ (where $A = [0,256) \bigcap Integers$)

λ = No of digits in the final hash digest

ptr = pointer = pointer to first byte of the object to be hashed

Φ = Size of the object in bytes

Set FinalHash = "00000…….0" (λ times)

Loop i = 1,2,3…….Φ

Convert the i^{th} byte into a binary string of length 4λ using Ψ

Convert the obtained binary string into hexadecimal (= HexString)

Set FinalHash = HexString FinalHash

RETURN FinalHash

The Code

```
char* HashDigest(char* arr, int lambda, int ClassSize = 0) {
char* ptr = arr;
    int phi;
    int fordifference = FastPower(lambda,2,PRIME);
     if (ClassSize == 0)
       phi = StringLength(arr);
     else
       phi = ClassSize;

     char* BinaryString = new char[4*lambda + 1];
     char* Hexed;
     char* FinalHash = new char[lambda + 1];

     for (int i = 0; i < lambda; i++) FinalHash[i] = '0';
     intoBinary(BinaryString, phi, 4*lambda);
     Hexed = Hex(BinaryString);
     Mod(FinalHash, Hexed, lambda);
     delete[] Hexed;

     for(int i = 0; i < phi; i++) {
       intoBinary(BinaryString, *ptr + i + fordifference,4*lambda);
       Hexed = Hex(BinaryString);
       Mod(FinalHash, Hexed, lambda);
       ptr++;
       delete[] Hexed;
     }

     FinalHash[lambda] = '\0';
     delete[] BinaryString;
     return FinalHash;
}
```

The **`intoBinary()`** function

```
void intoBinary(char arr[], int byte, int LENGTH) {

     char temp[5];
     for(int i = 0; i < LENGTH/4; i++) {
       byte = FastPower(FastPower(PRIM_ROOT, i+1, PRIME), byte, PRIME);
       Into4BitBinary(temp, byte);
       Concat(arr, 4*i, temp);
     }
     arr[LENGTH] = '\0';

}
```

The **`Hex()`** function

```
char* Hex(char x[]) {
// Returns precise hexadecimal equivalent string of binary string x

    int LengthofBinary = StringLength(x), lambda;

    if (LengthofBinary % 4 == 0) lambda = LengthofBinary/4;
     else lambda = LengthofBinary/4 + 1;

    int sum = 0, k;
     char* Hexed = new char[lambda+1];

     for(int i = LengthofBinary - 1, j = 1; i >= 0; i = i - 4, j++) {
       sum = 0;
       for(k = 0; k < 4; k++) {
              if ( (i-k) < 0 ) break;
              sum += pow(2,k) * (x[i-k] - 48);
       }
       Hexed[lambda-j] = HexForm2(sum);
     }

     Hexed[lambda] = '\0';
     return Hexed;
}
```

Explanation

The hash function takes a pointer to the first byte of the data structure that is to be hashed, and the size of the hash digest that is to be returned as **`lambda`** . The function converts the data into a hash by taking each byte at a time, then passing that to the psi function (pseudorandom generator described above) then converting each successive value of the psi sequence into its 4 bit binary equivalent by computing the XOR operation on the element of the psi series by going 4 bits at a time. For example of the value of the byte is 78, and the psi function returns 6,89,90,etc as its elements of the sequence the 4 bit equivalents will be 0110, 1100 (by computing XOR on 1001 and 0101) and 1111 (by computing XOR on 1010 and 0101),etc therefore the concatenated string will be "0110 1100 1111 etc" and its hexadecimal equivalent will thereby be "6 c f etc" the hexadecimal equialents obtained are then added to eaceh other (mod 16) to obtain the final hash value that is returned. Since the hash function used the psi (Ψ) function to create its sequences, the security of this hash function relies on the irreversibility of this function.

The hash function further implements the "Avalanche effect" because it mods each elementary hash value (i.e. the binary and hexadecimal sequences obtained) to the final hash thus each byte has equal weightage to the hash digest that is returned.

The hash function also gives weightage to the length of the object being hashed by treating the length of the object (int bytes) as the first byte and storing its elementary hash as the starting value for the FinalHash array.

Further, weightage is given to the length of the hash value required and the place of the byte in the data structure by adding a term to obtain the elementary hash value of each value, i.e. "`fordifference`" the iteration number 'i'.

Encryption

In cryptography, encryption is the process of encoding messages (or information) in such a way that eavesdroppers or hackers cannot read it, but that authorized parties can. In an encryption scheme, the message or information (referred to as plaintext) is encrypted using an encryption algorithm, turning it into an unreadable ciphertext. This is usually done with the use of an encryption key, which specifies how the message is to be encoded. Any adversary that can see the ciphertext (without knowing the key), should not be able to determine anything about the original message. An authorized party, however, is able to decode the ciphertext using a decryption algorithm, that usually requires a secret decryption key (in our case the same key).

For the purposes of this project we have utilized the "Viginere cipher" to encrypt and decrypt, according to which

$$c = m + k$$

$$m = c - k$$

where the + and – symobls dodnate addition modulo an appropriate number.

The Code

The **`Encrpyt()`** function

```
void Encrypt(char* ptr, int ClassSize, char Key[]) {
     for (int i = 0; i < ClassSize; i++) {
       *ptr = ((*ptr + Key[i%6]) % 256) - 128; // Because range of char is -
128 to 127
       ptr++;
     }
}
```

The **`Decrpyt()`** function

```
void Decrypt(char* ptr, int ClassSize, char Key[]) {
     for (int i = 0; i < ClassSize; i++) {
       *ptr = ((*ptr - Key[i%6]) % 256) - 128; // Because range of char is -
128 to 127
       ptr++;
     }
}
```