Effective Applications for Password Security

Ryan A. Plummer CSCI 373: Senior Research Seminar
St. John's University
Collegeville,MN
raplummer@csbsju.edu

***Abstract*—Today, most of the data that a singular person has access to is likely to be protected by a password. A password is the most common way to protect one's private information from being public. It usually involves a given string of letters, numbers and symbols arranged in such a way where no one but the user would be able to give. We have passwords to allow access to sites on the Internet, work computers, smart phones and even doors. However, attackers, who want your data, strive to use their cryptographic attacks to crack the databases that protect our passwords. Throughout the years, we have been developing ways to keep these attackers from getting our data. However, how do stop them?**

Contents

List of Figures

I. Introduction

Most of the data that individuals have access to will be protected by a password. This data is kept from individuals who may want the opportunity to use the data fraudulently or damage and alter said data for malicious reasons. Passwords

help protect data from individuals who should not have access. A password is the most common way to protect ones private information from being public. Passwords usually involve a given string of letters, numbers and symbols and the string is arranged in such a way, no one but the user would be able to present. We have passwords to allow access to sites on the Internet, work computers, smartphones and even doors. Passwords can protect such a large variety of things to allow us, as a society, to remain safe when sending and receiving confidential virtual items.

However, by themselves, passwords dont protect much of anything. In the past, researchers could get away with having an unaltered database of passwords because computers were expensive and were owned by few people. Today, however, malicious individuals or hackers have the technology to be able to easily acquire access to a password database. If one had a server holding every users passwords, they would not be safe by todays standards.

To protect these databases, they are encrypted through one-way hash functions. This altered the passwords in such a way that one would be unable to understand if they just copied the database for themselves. However, this did not deter hackers from employing other methods such as brute-force and dictionary attacks and rainbow tables. While some of these methods are merely algorithms trying to input every possible combination or inputting a set of commonly used passwords to obtain a valid password, some other attacks specifically exploit weakness within certain hash functions to obtain valid passwords. It seems like the hackers have the upper hand in this situation. However, with the help of proper password ideas and other hash altering algorithms, we can delay the hackers attempt to crack a password to the point where it would not be worth it to continue attacking.

II. Survey of Password Security

We have passwords to allow access to sites on the Internet, work computers, smartphones and even doors. However, passwords were not always the mainstream form of protection that it is today. Before the 1960s, no one had any need for passwords because computers were expensive and were reserved for large businesses and researchers. Furthermore, during those times, passwords were not necessary because computers were not designed to handle more than one user.

A. Compatible Time-Sharing System

However, in 1961, the Compatible Time-Sharing System (CTSS) created by Fernando J Corbato, a MIT (Massachusetts Institute of Technology) researcher, for the IBM 709 changed the future of computers. The time-sharing idea offered a system in which many users could be connected to the computer at the same time. Before CTSS, the computer that was used, an IBM 704, was only able to be used by one person at a time. Researchers had to sign-up to use it in advance. When the IBM 704 had a computational job, the researcher could wait hours or even days to receive the output or worse, an error from the job. For time-sharing to work, the computer had to halt its current progress, save it, load the process from another job and start that process from where it was halted. Corbato developed a process where the computer would save its progress on magnetic tape drives and that allow access for multiple CTSS users. [16] Now that every researcher had access to the computer, there was just the matter of discerning which person was using which data. Thus, each researcher received a password to provide access whatever data they were working on. They were each allowed a few hours of access time per week. Since passwords were new at the time, there was no knowledge of exploits or hacks to crack passwords. Furthermore, the MIT researchers that created the passwords did not seem to care about protecting the passwords. They were just held in a text file within the computer. [10] In 1962, a researcher named Scherr determined that those few hours a week was not enough time to run the simulations he developed. He effortlessly printed the entire database of password on the system. Suddenly, he had plenty of time to conduct his research. Also, in 1966, there was a software error that confused its welcome message and the password database. Whoever happened to log in was presented with the entire list of passwords for every researcher. [8]

III. Current State of the Field

As computers became more accessible and more affordable, more people began to adjust to a life with computers. On the other hand, since computers were more accessible to citizens and businesses alike, the opportunity to have private data stolen for malicious use was much higher. As a result, designs for the protection of the passwords needed to be established. Roger Needham was a British computer scientist known for pioneering the technique of protecting passwords using a one-way hash function in 1968. [6]

A. Cryptographic Hash Functions

Hashing is a term used to indicate the use of a cryptographic hash function, an algorithm that, along with a key, maps data of any size to data of a fixed size. It is known as ”one-way” because once an input is passed through, there is no way to change it back. This function has four main properties:

1) It is easy to computer the hash value for any given message.
2) It is infeasible to generate a message that has a given hash.
3) It is infeasible to modify a message without changing the hash.
4) It is infeasible to find two different messages with the same hash. [11]

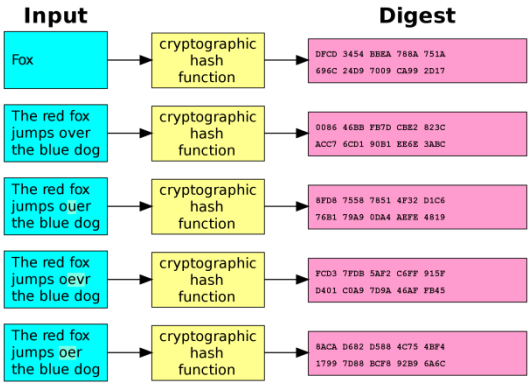

Fig. 1. The above picture provides an example of a hash function (SHA-1). The function takes an input and output a hash key that only the function can recognize.

B. Today's Hash Functions

The MD5 and SHA-1 were some of the most used hash functions during the 90s and even today. The MD5 was a message-digest algorithm that used a hash function outputting a 16 byte hash value expressed in a 32-digit hexadecimal number. It was regularly used to confirm data integrity. MD5 was developed by Ron Rivest in 1991 to fix inferior hash functions such as the MD2 (1989) and the MD4 (1990). [12] The MD2 was a non-one-way hash function created for 8-bit computers but it is still used in public key infrastructures in 2014. The MD4 was quicky updated to MD5 due to errors in coding. MD5 processes a variable-length message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit words); the message is salted so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits less than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, mod 264. [14]

SHA-1 was designed by the United States National Security Agency in 1995. Like MD5, SHA-1 used a hash function to produce a hash value. However, this value was a 20 bytes long and it was created as a 40-digit hexadecimal number. The MD5 and SHA-1 has had variants created, such as Tiger (1995) and Whirlpool (2005), that implemented the usage of both hash functions and executed them in similar ways. Both functions had issues with their security capabilities as they were compromised by collision attacks, an attempt to find the input to produce the hash value.

IV. Technical Analysis

A. Cryptographic Attacks

While we have ways to encrypt passwords, attacks have developed ways to decrypt or crack them using cryptographic attacks. Three commonly used attacks are Brute-Force Attacks, Dictionary Attacks and Rainbow Tables.

1) Brute-Force Attacks: A Brute-Force attack is an algorithm that goes through every possible string of characters. Because of this, a brute force attack is guaranteed to be successful. However, depending on the password it could take seconds to years to get a valid password.

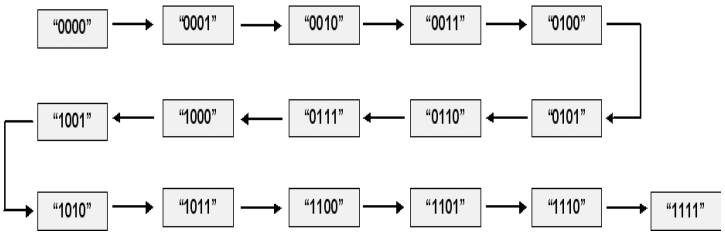

Fig. 2. Let's assume that in this set of binary numbers, our password is "0110". The Brute-Force Attack would start from the beginning at "0000" and slowly increase to "0001" to "0010" to "0011" and so on until it reaches our password "0110".

2) Dictionary Attacks: A Dictionary attack is an algorithm that searches through a set of words, such as a dictionary, to try to find a valid password. On average, this is much faster than a brute-force attack but, unlike it, a dictionary attack is not guaranteed to work.

3) Salting: To counteract the usage of some cryptographic attacks, a new technique called salting was created. Robert Morris explains that To this end, when a password is first entered, the password-program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the salt) and the 64-bit result of the encryption are entered into the pass word file. [8] In layman's terms, salt is random data used as an additional input to a hash function. Each password would have its own unique salt so that even if the hash key is discovered there would be no way to reverse engineer any passwords.

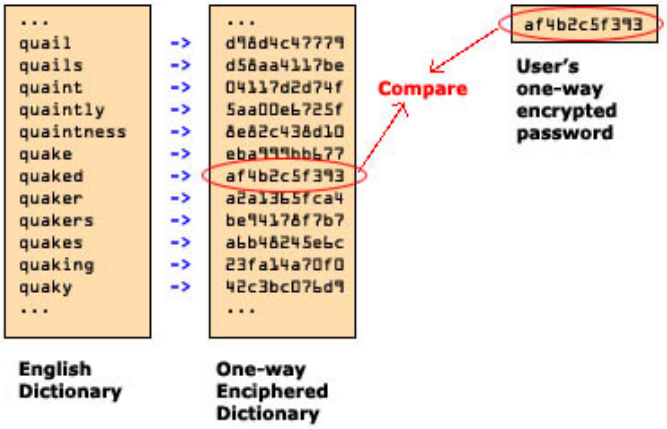

Fig. 3. We can see the algorithm go through the list of words. It hashes them and looks through the encrypted password database hoping for a matching hash.

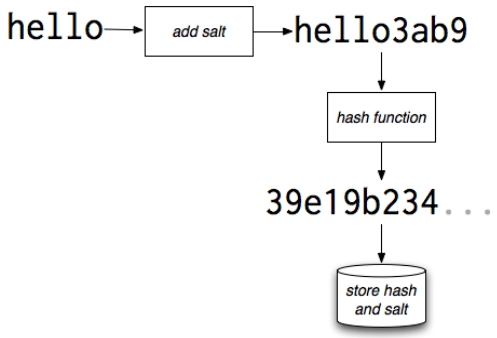

Fig. 4. This figure illustrates the use of salt. Every time an input is run through a hash function, the output will be the same. However, if salt is added to the end of the input and then hashed, the output will be different every time. This makes it harder for cryptographic attack to crack databases.

4) Rainbow Tables: Another attack called a Rainbow Table was also used to take advantage of hash collisions. It seems like a hybrid betweed and brute-force and dictionary attack. A rainbow table is a type of attack that uses a pre-computed table to attempt to reverse a cryptographic hash function. It is normally used for converting password hashes. It uses less processing time and more data than a brute-force attack but more processing time and less data than a dictionary attack. Since storing passwords in a normal plaintext file is dangerous, most owners that want to protect their database often use cryptographic hash function. Therefore, one cannot determine a valid password from looking at the hashed file. When a password is entered for authorization, its hash key is compared to the hash key in the database. If they match, the user is presented the data that the password was protecting. As we recall, hash functions map plaintext to a hash. [9]

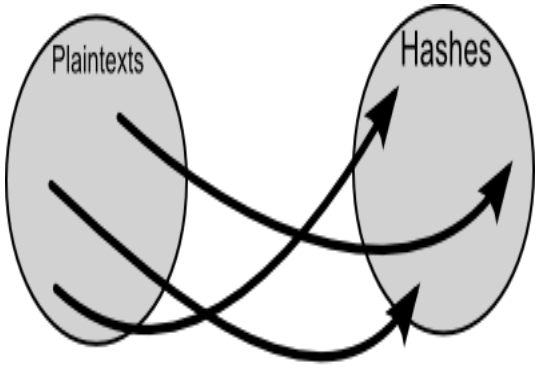

Fig. 5. This figure gives a visual representation of the algorithm chain that the rainbow table uses.

The objective of a rainbow table is to map the hash back to its original plaintext. To do that, rainbow tables use reduction functions. Let's be clear; reduction functions are not inverse hash functions. If you hash a plaintext then reduce the hash that is returned, you will not get the original plaintext file. The point of hash functions is that an inverse of a hash function cannot exist and are aptly named one-way. Rainbow tables work by taking a hash of a string and using a reduction function to reduce the hash to make a new string. Then, it hashes reduces the string again each time checking if a hash matches a database hash. It reduces for many iterations until a usable hash key is found. The chart below displays three different reduction functions that a rainbow table may use.

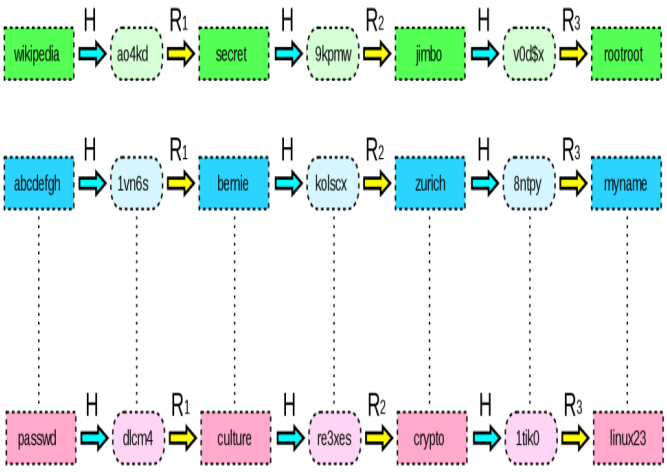

Fig. 6. To explain the chart above, in the first function, the rainbow table starts with the string wikipedia. Then, the string is placed through a one-way hash function producing the hash, ao4kd. Then, the rainbow table uses its reduction function to reduce the hash back into a string. This time it is secret. Then, secret is hashed again to get 9kpmw then to jimbo then to v0d$x then finally to rootroot. Every time a hash is returned, the rainbow table compares it to the password database hoping for the corresponding hash. This continues until the hash cannot be reduced anymore. The function then iterates through its generated hashes to find a valid hash

B. Collisions

If two different inputs end up generating the same hash value, a collision occurs. This is bad because if an attacker finds a valid hash value for a password and there are multiple collisions within the database, it would take the attacker less time to breach the entire database. Let's say that we pass two strings through a function and they both return a hash value of "P3A6H9". Since they both return the same hash value, if one were to pass the string through an authorizer for a website or something similar, it would be unable to tell the difference between both strings. Therefore, the attacker could use the passwords interchangeably for both things the passwords are protecting. Collisions are not preventable when you hash a very large set into a very short bit string. An example of this is the idea of the pigeonhole principle. That principle states that if n items are put into m containers and n is greater than m, then there must be at least one container that contains more than one item.

Fig. 7. This picture illustrates the pigeonhole principle. Since there are more pigeons than there are holes, there must be some holes that have more than one pigeon.

1) The Birthday Problem: The birthday problem is a variant of the pigeonhole principle. It asks, for a set of n randomly generated people in a room, what is the probability of two different people having the same birthday. There are 366 days in a year, including February 29 in the event of a leap year. If there are at least 367 people in that room, then there are at least two people that have the same birthday. However, the surprising fact is that in a group of 23 people, there is a 50% probability that there is a pair of people with a birthday on the same date. In a group of 70, that probability jumps to 99.9%. [17]

In order to calculate the probability, we can assume that P(B) is the probability of at least a pair of people having the same birthday. We can also assume that P(B') is the probability of no pair of people having the same birthday. In this case, B and B' are the only two outcomes so we can assume the equation P(B) = 1 - P(B'). Chapman illustrates a way to use this equation for the problem:

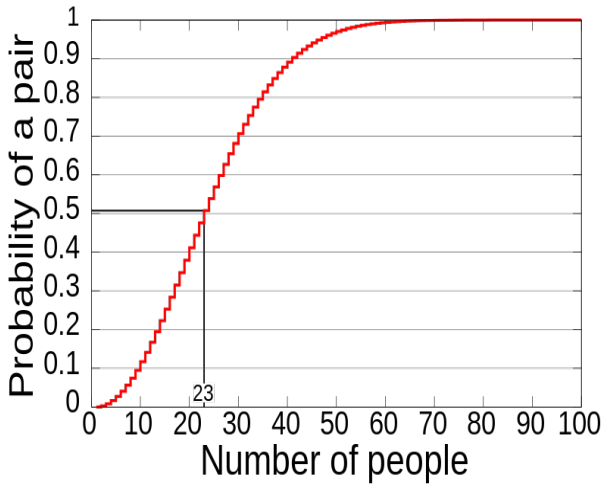

Fig. 8. The above graph illustrates the fact that as the number of people in the room increases, the probability that two people exists in the room with the same birthday also increases.

"When events are non-dependent of each other, the probability of all of the events occurring is equal to a product of the probabilities of each of the events occurring. Therefore, if P(A') can be described as 23 independent events, P(B') could be calculated as P(1)P(2)P(3)...P(23). The 23 independent events correspond to the 23 people, and can be defined in order. Each event can be defined as the corresponding person not sharing his/her birthday with any of the previously analyzed people. For Event 1, there are no previously analyzed people. Therefore, the probability,P(1), that person number 1 does not share his/her birthday with previously analyzed people is 1, or 100%. Ignoring leap years for this analysis, the probability of 1 can also be written as 365/365, for reasons that will become clear below. For Event 2, the only previously analyzed people is Person 1. Assuming that birthdays are equally likely to happen on each of the 365 days of the year, the probability,P(2), that Person 2 has a different birthday than Person 1 is 364/365. This is because, if Person 2 was born on any of the other 364 days of the year, Persons 1 and 2 will not share the same birthday. Similarly, if Person 3 is born on any of the 363 days of the year other than the birthdays of Persons 1 and 2, Person 3 will not share their birthday. This makes the probabilityP(3)=363/365. This analysis continues until Person 23 is reached, whose probability of not sharing his/her birthday with people analyzed before,P(23), is 343/365." [2]
P(B') is equal to the product of these individual probabilities:

1) P(B') = (365/365) *(364/365)* ...*(343/365)
2) P(B') = (1/365)23 *(365*364*363*...*343)
3) P(B') = 0.492703
4) P(B) = 1 - 0.492703 =0.507297 (50.7297%)

This probability algorithm can be applied to hash functions. The more passwords you try to hash raises the chances of two passwords created with the same hash key. In order to avoid hash collisions, a one-way hash function must be able to produce many hash iterations for many different passwords. Theoretically, since there are an infinite amount of numbers, there is no way to make a hash function collision immune. However, when making one collision resistant, it radically reduces the chances of collisions occurring.

V. Key-Stretching Algorithms

"Key Stretching" refers to the technique of increasing the time it takes to test each possible key. This works by feeding the initial key into an algorithm that outputs another key called an enhanced key. This key should be long enough to make it counter-productive for a brute-force or rainbow table to attempt to break into. This process should be secure enough to not allow any shortcuts through to calculate the enhanced key in less time. If a key is stretched, an attacker is left with two options. They can process the database with a brute-force attack or attempt to discover the enhanced key. This insures that the attackers with spend a longer time to find the keys. The speed of each attempt depends on the hardware of both the attackers and the protectors (the hardware running the key stretching algorithm). If they are the same, then it should take the attacker the same amount of time to process as the protector to stretch. However, it would still slow the attacker because they still need to compute the stretching function for every attack.

A. PBKDF2

PBKDF2 (Password Based Key Derivation Function 2) is password-strengthening algorithm that attempts to make it burdensome for a computer to check if a password is a database's derived or master key during a cryptanalytic attack. PBKDF2 implements SHA-1, another hashing algorithm in it's code. works by applying a function, such as a hash, to an input password along with a salt value for many iterations in order to produce a master key that can be used in future hashes. The number of iterations needed to produce a strong key was suggested to be around 1000. However, as CPU speeds improve, the number iterations can increase to create a more secure key. Since salt is added to each subsequent iteration, PBKDF2 effectively reduces the reliability of rainbow table because each password must be tested one at a time. The function takes five parameters:
Key = PBKDF2(PRF, Password, Salt, i, len)

- ***PRF*** is a pseudorandom function, such as HMAC-SHA-1 or MD5, that takes in two parameters, the password and a secret key only know to the user.
- ***Password*** is the phrase used to acquire the key from the pseudorandom function.
- ***Salt*** is a random cryptographic salt.
- ***I*** is the number of iterations desired to run through.
- ***Len*** is the desired length of the generated key. [3]

VI. Demonstration

For my project demonstration, I wanted to create a database of user names and passwords. I randomly generated the user-names using random letters, numbers and symbols with a random length from 3 to 7. For the password, I generated a random number from 1000 to 2000. While each user could have the same passwords, the database will not have two of the same user-names.

A. Creating the Database

For my test, I created 300 users for my database. Here is the source code for the creation of the database.

```
import java.util.*;
import java.io.*;

public class CreateRandomUserandPassword{

  private static String[] letters =
      {"a","b","c","d","e","f","g","h","i","j","k","l","m
  private static String[] numbers =
      {"0","1","2","3","4","5","6","7","8","9"};
  private static String[] sybmols =
      {"!","@","#","$","%","^","&","*","(",")","_","-","+
      "/", ",", "<", ".", ">", ";",","};
  private static HashMap<String,String>
      database = new
      HashMap<String,String>(500);

   public static String generateUserName(){
     Random ran = new Random();
     int length = ran.nextInt(3)+4;
     String userName = "";
     while(length > 0){
       int chance = ran.nextInt(4);
       if(chance == 1){
         userName +=
             letters[ran.nextInt(letters.length)];
       }else if(chance == 2){
```

```
      userName +=
          numbers[ran.nextInt(numbers.length)];
     }else if(chance == 3){
      userName +=
          sybmols[ran.nextInt(sybmols.length)];
     }
     length--;
   }
   return userName;
}

 public static String createUserName(){

  Scanner sc = new Scanner(System.in);
  String userName = sc.next();
  return userName;
 }

 public static String generatePassword(){
  Random ran = new Random();
  int num = ran.nextInt(1000)+1001;
  String pass = ""+num;
  return pass;
 }

 public static void addRandomUsers(int
     amount){
  for(int i = 0;i<amount;i++){
   String user = generateUserName();
   String pass = generatePassword();
   if(!database.containsKey(user)){
    database.put(user,pass);
    System.out.println(user + " : " +
        pass);
   }
   else{
    System.out.println("Username already
        exists");
   }
  }
 }

 public static void addCustomUser(){
  String user = createUserName();
  String pass = generatePassword();
  if(!database.containsKey(user)){
     database.put(user,pass);
     System.out.println(user + " : " +
         pass);
  }else{
     System.out.println("Username already
         exists");
   }
 }

 public static void writeDataBase() throws
     IOException{
  Scanner sc = new Scanner(System.in);
  System.out.println("Enter a name for the
      file.");
  String name = sc.next();
  PrintWriter writer = new
      PrintWriter(name+".txt");
  for(Iterator i =
      database.keySet().iterator();
      i.hasNext();){
   String s = (String) i.next();
   writer.println(s + " : " +
       database.get(s));
  }
  writer.close();
  System.out.println("Database written to
      file: "+name+".txt!");
 }

public static void main(String[] args){
  Scanner sc = new Scanner(System.in);
  int choice = 0;

  while(choice!=9){
   System.out.println("This is a random
       username and password generator!");
   System.out.println("Press 1 to generate
       a random amount of random users.");
   System.out.println("Press 2 to create a
       custom user.");
   System.out.println("Press 3 to show
       information about the current
       database.");
   System.out.println("Press 4 to write
       the database to a file.");
   System.out.println("Press 8 to clear
       the database.");
   System.out.println("Press 9 to exit the
       program.");
   try{
    choice = sc.nextInt();
    if(choice==9){
     System.out.println("Good-bye!");
     System.exit(0);
    }else if(choice==1){
     System.out.println("How many do you
         want to generate?");
     int num = sc.nextInt();
     addRandomUsers(num);
    }else if(choice==2){
     System.out.println("Enter a
         username!");
     addCustomUser();
    }else if(choice==3){
     System.out.println("There are
         "+database.size()+" unique users
         in this database");
    }else if(choice==4){
     writeDataBase();
    }else if(choice==8){
```

```
            database.clear();
            System.out.println("Database
                Cleared");
          }
        }catch(Exception e){
          System.out.println("Only numbers
              please!");
        }
      }
    }


  }
```

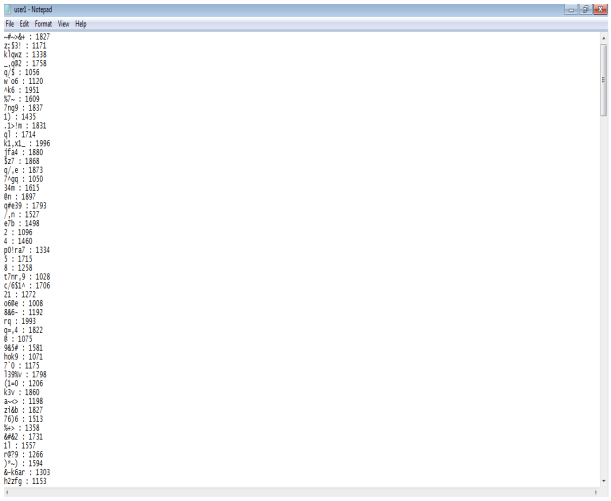

Fig. 9. The created password database.

B. Encrypting the Database

I used two encryption method in order to compare them to each other when I attempt to crack them. One method I used is MD5 without any salt as a placebo. The other method is to use PBKDF2 with the standard 128-bit salt and 256-bit output. Here are the source code for the MD5 hashing algorithm.

```
import java.io.*;
import java.security.*;
import java.util.*;



public class MD5Hasher{

  public static String mD5(String message){
    String digest = null;
    try{
      MessageDigest md =
          MessageDigest.getInstance("MD5");
      byte[] hash =
          md.digest(message.getBytes("UTF-8"));
      StringBuilder sb = new
          StringBuilder(2*hash.length);
      for(byte b : hash){
        sb.append(String.format("%02x",
            b&0xff));
      }
      digest = sb.toString();
    } catch (Exception ex){
      System.out.println("Unexpected Error!");
    }
    return digest;
  }

  public static void main(String[] args){
    try{
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the name of
            the file you want to encrypt!");
        String name = sc.next();
        Scanner infile = new Scanner(new
            File(name+".txt"));
        System.out.println("Enter a name for
            the encrypted file!");
        name = sc.next();
        PrintWriter writer = new
            PrintWriter(name+".txt");

        while(infile.hasNextLine()){
          String line = infile.nextLine();
          String user =
              line.substring(0,line.indexOf(":")-1);
          String password =
              line.substring(line.indexOf(":")+2);
          writer.println(user+ " : "
              +mD5(password));
        }
        writer.close();
      }
      catch(Exception e){
        System.out.println("Cannot Hash");
      }
    }
}
```

C. Cracking the Database

In order to crack my database, I decided to use a rainbow table. Since I know that every user has a password of a length of 4, I created a reduction function that look for the first 4 numbers of a hash value and uses that as the reduction key.

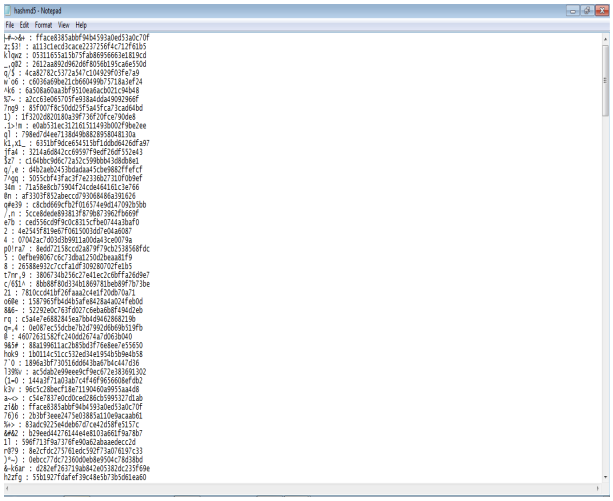

Fig. 10. The hashed database using MD5

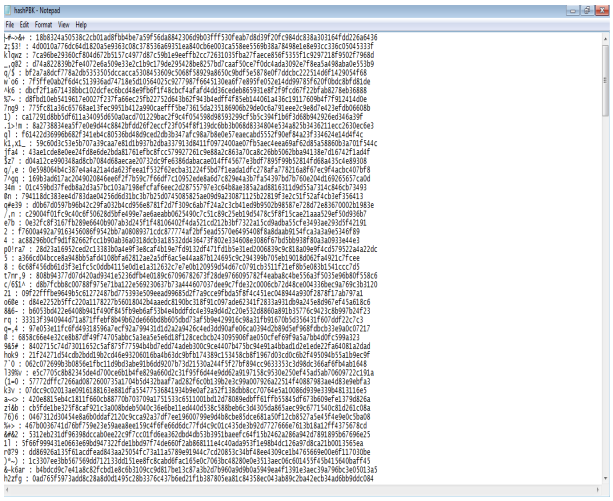

Fig. 11. The hashed database using PBKDF2

Then, it uses that key to create many chains of reduction keys and hash values. When it is done, it compares the chains to the encrypted password database to find a valid match. Here is the source code for my rainbow table for the MD5.

```
import java.io.*;
import java.security.*;
import java.util.*;



public class RainbowHashMD5{
  public static int itera = 10;
  public static int found = 0;
  public static final int MAX_UNITS = 2000;
  private static HashMap<String,String>
      rainbowtable = new
      HashMap<String,String>(MAX_UNITS);
  private static HashMap<String,String>
      answertable = new
      HashMap<String,String>(MAX_UNITS);

public static PrintWriter mwriter;

public static String mD5(String message){
  String digest = null;
  try{
    MessageDigest md =
        MessageDigest.getInstance("MD5");
    byte[] hash =
        md.digest(message.getBytes("UTF-8"));
    StringBuilder sb = new
        StringBuilder(2*hash.length);
    for(byte b : hash){
      sb.append(String.format("%02x",
          b&0xff));
    }
    digest = sb.toString();
  } catch (Exception ex){
    System.out.println("Unexpected Error!");
  }
  return digest;
}

public static String getReductionKey(String
    hash,int length){
  int l = length;
  String key = "";
  String hashk = hash;
  while(l > 0){
    String[] h = hashk.split("");
    for(int i = 0; i<h.length; i++){
      try{
        int n = Integer.parseInt(h[i]);
        key += h[i];
        hashk =
            hashk.substring(hashk.indexOf(h[i])+1);
        break;
      }
      catch(NumberFormatException e){
      }
    }
    l--;
  }
  return key;
}

public static void
    createRainbowTable(Scanner file, int i){
  int counter = i;
  int buffer = 500;
  String hashkey = "";
  String hash = "";

  //Getting the first reduction key
  if(file.hasNextLine()){
    String line = file.nextLine();
    String password =
```

```
      line.substring(line.indexOf(":")+2);
    hashkey = getReductionKey(password,4);

  }


  while(counter > 0){
    try{
      hash = mD5(hashkey);
      if(!rainbowtable.containsKey(hashkey)){
        System.out.println(hashkey);
        rainbowtable.put(hashkey ,hash);
        counter--;
        //System.out.println(hashkey + " -> "
            + hash);
      }
      else{
        buffer--;
        if(buffer <=0)
          break;
      }
      hashkey = getReductionKey(hash,4);
      //System.out.println(hashkey);
      //System.out.println("Steps until done:
          "+counter);
    }
    catch(Exception e){
      System.out.println("lol-----------------------------break;--------");
    }
  }

}

public static void writeRainbowTable(String
    name) throws IOException{
  PrintWriter writer = new
      PrintWriter(name+".txt");
  for(Iterator i =
      rainbowtable.keySet().iterator();
      i.hasNext();){
    String s = (String) i.next();
    writer.println(s + " : " +
        rainbowtable.get(s));


  }
  writer.close();
  //System.out.println("Rainbow Table
      written to file: "+name+".txt!");
}

public static void runHashSearch(String
    hashname,String tablename,int
    iterations) throws FileNotFoundException{
  Scanner hashedfile = new Scanner(new
      File(hashname+".txt"));
  PrintWriter writer = new
      PrintWriter("answerMD5-"+itera+".txt");
  Scanner tablefile;
  int similarhashesfound = 0;
  int iter = iterations;
  String hashtobefound = "",line="";
  while(iter > 0){
    while(hashedfile.hasNextLine()){
      line = hashedfile.nextLine();
      hashtobefound =
          line.substring(line.indexOf(":")+2);
      //System.out.println(hashtobefound);
      tablefile = new Scanner(new
          File(tablename+".txt"));

      while(tablefile.hasNextLine()){
        String sline = tablefile.nextLine();
        String rainhash =
            sline.substring(sline.indexOf(":")+2);
        if(hashtobefound.equals(rainhash)){
          if(!answertable.containsKey(sline.substring(0,s
          answertable.put(sline.substring(0,sline.indexOf
              , hashtobefound);
          System.out.println(hashtobefound
              +":"+rainhash);
          //System.out.println("Similar Hash
              Found!");
          similarhashesfound++;
          }
        }
      }
    }
    iter--;
    if(iter > 0){
      try{
        createRainbowTable(new Scanner(new
            File(tablename+".txt")),MAX_UNITS);
      }catch(IOException e){
        System.out.println("IO Error!");
      }
    }
  }

  if(similarhashesfound==0){
    System.out.println("No matches found...");
    writer.println("None");
  }
  else{
    System.out.println("Found "
        +similarhashesfound+ " potential
        hash(es)!");
    for(Iterator i =
        answertable.keySet().iterator();
        i.hasNext();){
      String s = (String) i.next();
      writer.println(s + " : " +
          answertable.get(s));
      mwriter.println(s + " : " +
          answertable.get(s));
```

```
    }
    writer.println("Found "
        +similarhashesfound+ " potential
        hash(es)!");
    found += similarhashesfound;
   }
   writer.close();

  }

  public static void main(String[] args){
   try{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the name of the
        encrypted file!");
    String name = sc.next();
    Scanner infile = new Scanner(new
        File(name+".txt"));
    System.out.println("Enter a name for the
        table.");
    String tname = sc.next();
    tname +="-";
    mwriter = new
        PrintWriter("MasterAnswerTable.txt");
    while(itera > 0){
    createRainbowTable(infile,MAX_UNITS);
    writeRainbowTable(tname+itera);
    runHashSearch(name,tname+itera,100);
    itera--;
    rainbowtable.clear();
    }
    mwriter.append("Found " +found+ "
        potential hash(es)!");
    mwriter.close();
   }
   catch(Exception e){
    System.out.println("Cannot Hash");
   }
  }
}
```

The rainbow table did not find any matches for the hashed PBKDF2 file. However, it found 8 unique matches for the hashed MD5 file. The conclusion that I can give is that the salt is the reason that the rainbow table could not find anything. For a hashed database, the rainbow table has to guess the correct hash. However, if a database has been salted, not only does the rainbow table have to guess the hash, it would also have to guess the salt. That is nearly impossible to do. Salt can increase security on its own.

VII. Ideal Passwords

Although we have many ways to encrypt and protect passwords, it all means nothing if an attacker simply guesses a user's password. Password strength is an important part

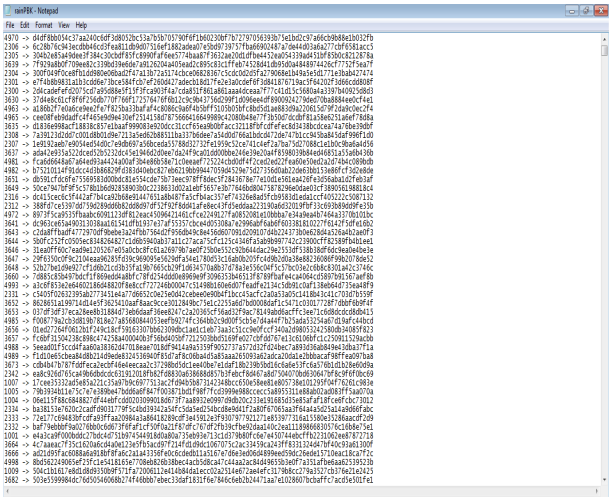

Fig. 12. Chain of hashes created by the rainbow table.

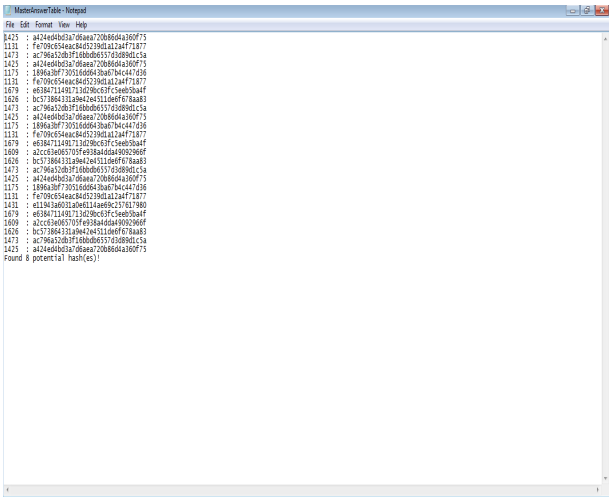

Fig. 13. The hash values my rainbow table found.

in creating a password because it determines how easily a password is guessed. Most modern websites allow passwords to contain numbers, letters and symbols and they critique your combination by giving you a "weak" or "strong" response to give an idea how detectable or easy to crack a password is. Passwords are bad when they are easy for humans to guess or hard for anyone to remember. Using a date or a name that is significant to you (such as a spouse's name or a birthday) would be a bad idea for a password. It is also unwise to use dictionary words or more commonly used words, like password, as passwords. Generally, the longer the password is, the longer it will take for someone to force their way to get it. If numbers, symbols and a mixture of uppercase and lower letters are used it can exponentially increase the amount of time it takes to discover it. The best password is an easy to remember phrase with a mixture of words, numbers, symbols, and abbreviations of words. [13] Using GRC's Interactive

Brute Force Password Search Space Calculator, basically a program that simulates the processes of a brute-force attack, one can simulate how long one such attack would take to crack a password. This calculator keeps track of the number of uppercase and lowercase letters, symbols and numbers in the string. If you use the string a in the calculator you would get this screen:

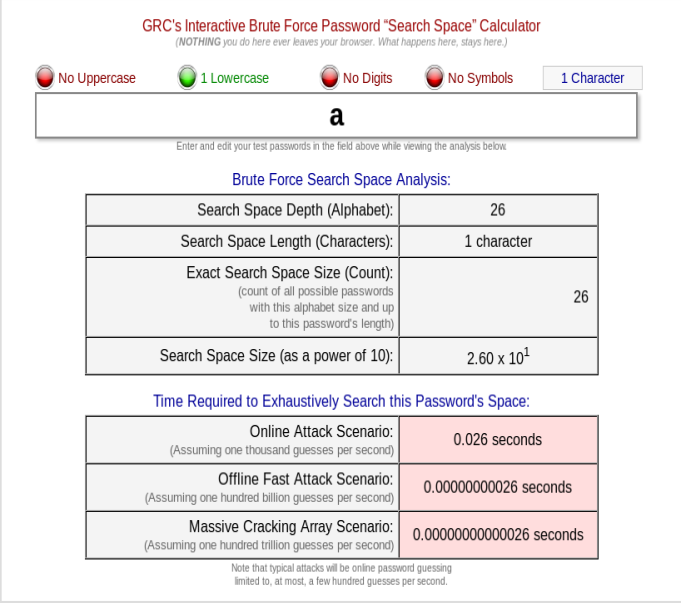

Fig. 14. The calculator simulates the total cracking time of string "a"

As you can see, with the string a, the calculator searches the exact depth of the password with respect to the alphabet (26 letters) , the number of digits allow on the keyboard (0-9 10 total) and the number of symbols on the keyboard (33 symbols including space). In an online attack scenario, assuming 1000 guesses per second, it would take 0.026 second to crack the password a. It would take an even shorter amount of time to guess it if there were more attempts per second such as an Offline (100 billion guesses) or a Massive Cracking (100 trillion guesses) attack. Most attacks use around 1000 guesses per second so it should be safe to assume that it would take the Online Attack Scenario's amount of time. However, let's add a little length to the password. With the string aa you get this:

Not only did the Exact Search Space Size, the count of all possible passwords with this alphabet size and up to this password's length, increase, the amount of time it would take for an Online Attack Scenario to crack aa has increased 27 times. Alone this proves that longer password are better passwords. However, if it is a long password that an attacker will check early, then it can take less time to crack it. According to the site, the #1 most commonly used password

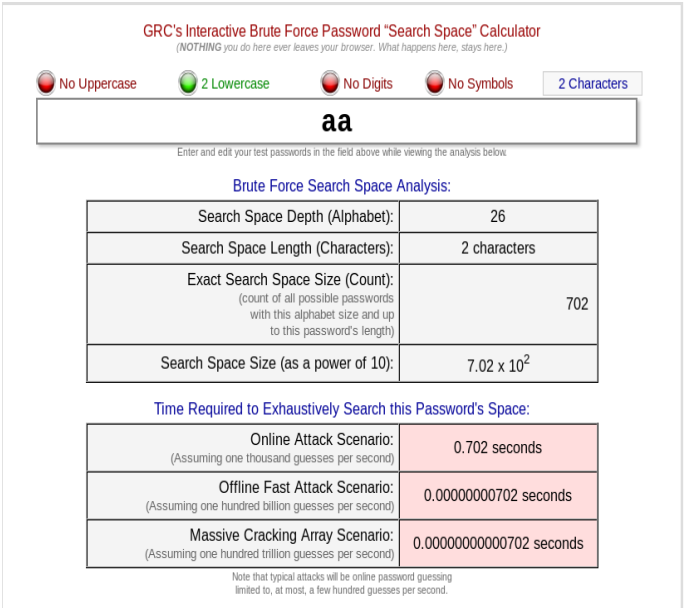

Fig. 15. The calculator simulates the total cracking time of string "aa"

is 123456, and the 4th most common is Password. So any password attacker and cracker would try those two passwords immediately. Yet the Search Space Calculator above shows the time to search for those two passwords online (assuming a very fast online rate of 1,000 guesses per second) as 18.52 minutes and 17.33 centuries respectively! If 123456 is the first password that's guessed, that wouldn't take 18.52 minutes. And no password cracker would wait 17.33 centuries before checking to see whether Password is the magic phrase. This calculator is designed to help users understand how many passwords can be created from different combinations of character sets (lowercase only, mixed case, with or without digits and special characters, etc.) and password lengths. The calculator then puts the resulting large numbers (with lots of digits or large powers of ten) into a real world context of the time that would be required (assuming differing search speeds) to exhaustively search every password up through that length, assuming the use of the chosen alphabet. [13] What this means is that not only must one have a sufficiently lengthy password to prevent brute-force attack, one must also make sure that the password is not common or easily guessed. Let's try the password "$pr1tE90". This is not a word that someone would think to guess because if someone were to just look at the word it would appear to be gibberish. When we input the password we get:

As you can see, since there are such a different combination characters, it would take the attack a long time to crack it. However, since it uses so many different characters, it would be hard to remember. Let's try an easier password,

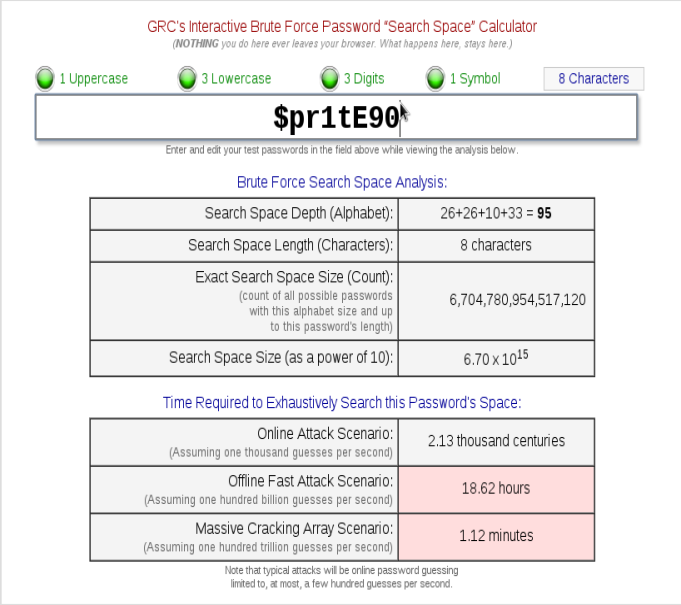

Fig. 16. The calculator simulates the total cracking time of string "$pr1tE90"

"WhatIsUpDude". This is such an arbitrary phrase that an attack would not guess early.

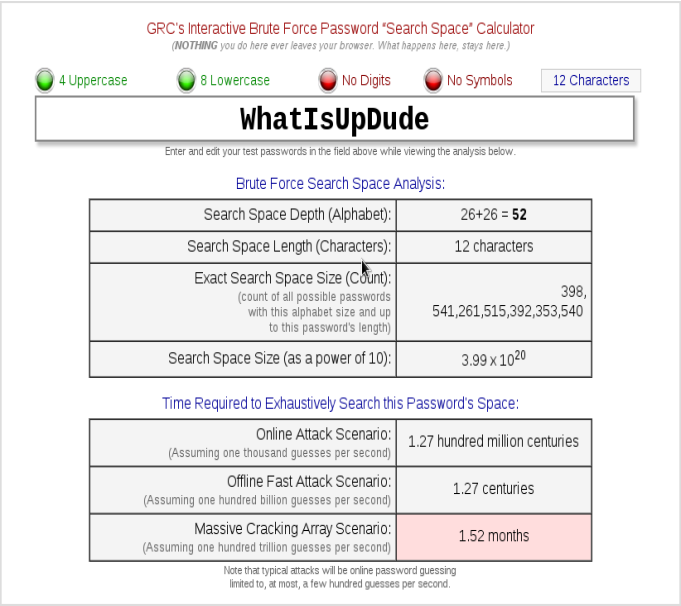

Fig. 17. The calculator simulates the total cracking time of string "WhatIsUpDude"

Compared to the previous password, it uses less symbols. But as you can see, it would take more time to crack this one. This proves that longer simpler words in phrase would be best used in a password. [15]

VIII. Future Trends

Password Security has come a long way since the creation of the Compatible Time-Sharing System in 1961. We have advanced from having passwords specifically made to designate separate users to protecting the specific passwords to keep them safe from malicious individuals that wish to obtain the information for their own personal gain. Since we have arrived here, we seem to have reached a point where computers are so fast and complex that it is becoming easier for attackers to develop ways to crack password databases. In my opinion, I feel that we have reached the peak of protection that we can give databases. Even though we have the ability to develop safer and more efficient ways to improve security as technology advances, attacker will also have access to these advances and therefore create faster and more efficient ways to crack password databases. However, I believe that in a few years we are poised to change the nature of password security for the better. We will change this security by turning to Biometric Authentication. [7]

A. Biometric Authentication

What is Biometric Authentication? It is the action of using human characteristics as a way of granting access to protected information. Imagine, instead of using text-based passwords, you could use your fingerprints, the retinas in your eyes or even your voice as a password. Because you use your own body parts as authorization, it would be even harder for attackers to gain access to your account using conventional means. Words or any combination of characters can be recreated or guess. As of today, there is no way to recreate a fingerprint.

B. Applications

Wesley Fenlon, an author for the tech site Tested, states that we are looking for a alternative to character-based passwords and worries if biometrics can fill the void that it would leave. He uses an example involving Apple's iPhone 5S, one of their newest version of iPhones. "Apple's new iPhone 5S aims to replace four-digit pins with fingerprint scans, which avoid most of the problems passwords currently face.

Fingerprints can't be guessed like common passwords, or brute-forced like short passwords. Apple also isn't giving third-party apps access to the fingerprint data or storing them in the cloud, which will make them much harder to steal in a security breach." [5] Since smartphones are typically used by one person, the scanner would be a perfect alternative for it. However, Fenlon went further, asking if there were any large-scale models that would include many people. One other example he provided was the *Nymi*. The *Nymi* is a wristband that analyzes your heartbeats to identify you. You can then pair it with another device that requires a heartbeat to provide authentication.

Fig. 18. Pictured is the Iphone 5. It displays an app that uses fingerprint recognition to provide access.

Fig. 19. Pictured is the "Nymi" wristband. It monitors heartbeats and heartrate to provide authorization to equipment that are compatible with it.

However, as Fenlon states "But the security of that system is worth little if no one can log into Google or Facebook with the *Nymi*, or use it to unlock their front door or start their car." It seems that we must wait for the rest of the world to catch up to adapt to the new ways technology advances.

C. Limitations

1) Speed: One limitation of biometrics is that sometimes they can be quite slow. The swiftest of typers can enter their text password and be granted access to their information in as little as a few seconds. However, if you were to use your fingerprints as a password, you would have to wait for the system to scan it, process it and determine if it is a valid match. Tim De Cant recalls his experience with biometric systems. "... Sarah Fischer, a programmer, had walked me through the process of scanning my fingerprints. She had me place my right fingers on the scanner, then my left, then both thumbs. The software would let her know if my prints were successfully captured. Going by what he said, I can only assume that only one fingerprint will not be enough to authorize anyone. "I failed at first, if you can fail at such a thing. Press harder, Sarah had said. That seemed to do the trick, but my second attempt still took longer than it should have 38 seconds to scan all ten fingerprints. According to guidelines issued by the federal government, the process should take 20 seconds or less. " Even though he took 38 seconds to scan due to a problem, it usually takes around 20 second which is longer than inputting a text password. [1]

2) Unexpected Changes: Another problem with using biometrics is determining what to do if something were to happen to your body that would alter or otherwise change the seemingly immutable parts that were used to authorize. Let's say that a person used their voice as a password. If they developed a cold or a stuffed nose, they would be unable to used their voice again until their aliment cleared up. What if we assume the worst and they lost their voice forever? Until we can overcome this hurdle, we would have a hard time improving security and biometrics will remain obscure. [4]

IX. Conclusion

Hashing and Salting have been very effective from the past to today. Because of their effectiveness, they are still relevant to this day. Outdated one-way hash functions updated and by PHP, Java, C# and Ruby source codes of one-way hash functions that many businesses of today use. These hashes protect the password in the event of a security breach of the password database. While this does not protect a user if a dictionary or brute-force attack is successful against their password, it will protect other passwords from being stolen. This is why developing a strong password is good. A good password will take cryptographic attacks out of the equation. As technology advances and we get faster computers, the need for better security will rise. Quicker processing speed will undeniably hasten cryptographic attacks. However, because of Biometrics, there may be no need to live in fear of attacker stealing information in the coming years. Biometrics has its pros and cons. In theory, it is a more secure way of providing authorization than a text-based password. No attacker has the means of accessing another persons body. However, it can be somewhat slower than text passwords and if something were to happen to the authorization process the owner could be permanently locked out. Even so,I believe that in a few years we can solve the problems that are keeping biometrics from being our main form of security.

References

[1] Tim De Chant. The boring and exciting world of biometrics. *NovaNext*, jun 2013.

[2] Ian M. Chapman, Sylvain P. Leblanc, and Andrew Partington. Taxonomy of cyber attacks and simulation of their effects. In *Proceedings of the 2011 Military Modeling & Simulation Symposium*, MMS '11, pages 73–80, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[3] Younsung Choi, Donghoon Lee, Woongryul Jeon, and Dongho Won. Password-based single-file encryption and secure data deletion for solid-state drive. In *Proceedings of the 8th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '14, pages 5:1–5:7, New York, NY, USA, 2014. ACM.

[4] Geoff Duncan. Why haven't biometrics replaced passwords yet? *Digital Trends*, March 2013.

[5] Wesley Fenlon. Are biometrics the future of passwords? *Tested*, Sept 2013.

[6] Li Gong. *Inside Java 2 Platform Security Architecture, API Design, and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] Shirley Li. Biometrics can be cool again. *The Atlantic*, oct 2014.

[8] Robert Mc Millan. The world's first computer password?

[9] Deepika Dutta Mishra, C. S. R. C. Murthy, Kislay Bhatt, A. K. Bhattacharjee, and R. S. Mundada. Development and performance analysis of hpc based framework for cryptanalytic attacks. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 789–794, New York, NY, USA, 2012. ACM.

[10] Robert Morris and Ken Thompson. Password security: A case history. *Commun. ACM*, 22(11):594–597, November 1979.

[11] C. Paar and J. Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, 2009.

[12] Bart Preneel. The first 30 years of cryptographic hash functions and the nist sha-3 competition. In *Proceedings of the 2010 International Conference on Topics in Cryptology*, CT-RSA'10, pages 1–14, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Gibson Research. Data recovery: Haystack. *Gibson Research Corporation*.

[14] Joseph D. Touch. Performance analysis of md5. *SIGCOMM Comput. Commun. Rev.*, 25(4):77–86, October 1995.

[15] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L. Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[16] Van T. Vleck. Fernando j. corbato.

[17] Kan Yasuda. Multilane hmac: Security beyond the birthday limit. In *Proceedings of the Cryptology 8th International Conference on Progress in Cryptology*, INDOCRYPT'07, pages 18–32, Berlin, Heidelberg, 2007. Springer-Verlag.

Appendix A
Reflection

When I first attended CSBSJU, I had no idea what I was going to do for my major. I just chose the two things that I thought I was good at: Math and Physics. At that time I had no idea that I would find a new found love for my current major Computer Science. Physics became a chore and I need a new major to get into so, on a whim, I jokingly took CS150. As I completed class after class, I felt more and more sure that COMSCI was my calling. I believe that, from taking classes from CS150 to CS373, I have acquired a great deal of knowledge of applications and history to be able to complete my capstone project. Java has been the turning point of my life.

I choose to do a presentation on Password Security because we live in an age that is completely dominated by computer technology. What would require human guidance in the past can now be somewhat controlled by a computer. Our doors, car and even some house have some sort of computer interaction with it. Of course, I can't forget the Internet, the biggest way our society uses computer technology. This has become so ingrained in our culture that we are putting personal information and other private data on the Internet. In order to keep that trust intact, we have to develop security to protect it. There will always be those types of people who would like nothing more than to steal information so they can benefit. I've always found it interesting that we have such powerful ways to protect password but the greatest protection of all relies on the user. Attackers are great guessers and if they even guess just one right password it could make it worthwhile to keep doing it which is why a strong password is so important.

In all, I felt like informing the masses that a strong password would go a long way due to the fact that most attacks prey on shorter, weaker, easily-guessed password. Coming from the age where attackers are breaking into bank accounts and hacking into others' personal data, I felt that it is important to educate others on how to protect themselves until biometrics makes its debut.