



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

DATABASE SECURITY- TO DETECT AND PREVENT DICTIONARY ATTACKS

A PROJECT REPORT

Submitted by

Sudamshu M Dushyanth
18BCE2197

Course Title:

Information Security Analysis and Audit

Course Code:

CSE3501

Under the guidance of

Prof. Umadevi

VIT University, Vellore.

ABSTRACT

In today's day and age, databases are of prime importance, so their security and protection is also the foremost agenda for the companies managing and maintaining them. Sadly, a variety of attacks exist which can hamper the normal functionality of the relational databases. In this project we take into consideration the 'DICTIONARY ATTACKS'. Dictionary Attacks are a method of using a program to try a list of words on the interface or program that is protecting the area that you want to gain access to. A weakness of dictionary attacks is that it obviously relies on words supplied by a user, typically real words, to function. Examples of programs that use dictionary attacks: John the Ripper, L0phtCrack, and Cain And Abel.

Specifically considering 'JOHN THE RIPPER' attack it can be prevented by using a method called 'SALTING'. A salt is random data that is used as an additional input to a one-way function that "hashes" a password or passphrase. Salts are closely related to the concept of the nonce. The primary function of salts is to defend against dictionary attacks or against its hashed equivalent, a pre-computed rainbow table attack.

For detecting this attack, we can see the average time is taken by the attack to crack a password, depending on the result we can find whether the 'John The Ripper' uses 'Dictionary Attacks' or the general brute force method.

In this project, programs will be created for detecting the above-mentioned attack and preventing it from using 'SALTING'.

KEYWORDS:

- Dictionary Attack
- John the ripper
- Hash Suit
- MD5
- bcrypt
- Salting
- Blowfish

INTRODUCTION

To understand how to protect yourself from a password attack, you should become familiar with the most commonly used types of attacks. With that information, you can use password cracking tools and techniques to regularly audit your own organization's passwords and determine whether your defences need bolstering. The most common type of attack is password guessing. Attackers can guess passwords locally or remotely using either a manual or automated approach. Password guessing isn't always as difficult as you'd expect. Most networks aren't configured to require long and complex passwords, and an attacker needs to find only one weak password to gain access to a network.

A dictionary attack is a technique or method used to breach the computer security of a password-protected machine or server. A dictionary attack attempts to defeat an authentication mechanism by systematically entering each word in a dictionary as a password or trying to determine the decryption key of an encrypted message or document. Dictionary attacks are often successful because many users and businesses use ordinary words as passwords.

The most common method of authenticating a user in a computer system is through a password. This method may continue for several more decades because it is the most convenient and practical way of authenticating users. However, this is also the weakest form of authentication, because users frequently use ordinary words as passwords. Antagonistic users such as hackers and spammers take advantage of this weakness by using a dictionary attack. Hackers and spammers attempt to log in to a computer system by trying all possible passwords until the correct one is found. Two countermeasures against dictionary attacks include:

- **Delayed Response**: A slightly delayed response from the server prevents a hacker or spammer from checking multiple passwords within a short period of time.
- **Account Locking**: Locking an account after several unsuccessful attempts (for example, automatic locking after three or five unsuccessful attempts) prevents a hacker or spammer from checking multiple passwords to log in.

The scope of this project includes an explanation of dictionary attacks, how to prevent them using slow hashing algorithms like bcrypt, scrypt, PBKDF2. Dictionary attacks are not effective against systems that make use of multiple-word passwords and also fail against systems that use random permutations of lowercase and uppercase letters combined with numerals.

LITERATURE REVIEW

Passwords have been the most popular means of authentication for many decades. An enterprise employee uses multiple passwords every day in order to use all applications and systems provided by his employer. Businesses spend a considerable amount of resources in order to deploy authentication mechanisms and policies, which are then compromised due to password leaks or misuse. Passwords by nature, introduce many security problems into otherwise sufficiently secure architectures. Since humans select passwords easy to remember, it therefore easy to guess passwords as well. Additionally [1], attackers have sophisticated automated software that can guess average passwords with great success. On the other hand, when users select random password, they once again compromise security by selecting short random passwords, which they can remember. Attackers can once again guess the password by trying all possible combinations, which are not many for a short password.

Security professionals implement techniques that attempt to stop attackers from using the offline dictionary or brute-force attacks on passwords. These techniques are not always effective. While an enormous random password might seem more secure, users will simply write it down somewhere for everyone to see. By introducing password policies, which force users to change their passwords frequently and include additional complexity such as numbers and special characters, security professionals often force users into making up predictable password creation strategies. The way that users interact with and choose passwords has been studied for many years. Thus given an attacker and

a big collection of user-selected passwords, we are in a position to say with certainty that many (if not most) passwords will be recovered. Furthermore, passwords are very attractive as a means of compromising a system because, given the effort needed to realize other techniques such as penetration testing and social engineering, password cracking is the easiest. Passwords [4] are usually encrypted using a one-way algorithm that produces a hash of a set length regardless of the size of password, such that recovering the password from the hash is considered impossible. Hashes can only be cracked by trying to hash possible passwords and comparing the output with the hash to be cracked [3]. Password cracking is not the same as breaking the underlying cryptographic primitives. It is fair to assume that all public cryptographic algorithms are unbreakable for a common attacker. Password cracking is all about analyzing human behavior and finding automated ways to effectively and efficiently “reverse” encrypted passwords back to their plaintext form. The purpose of this paper is to identify existing password cracking techniques and propose new ones. By doing so, it aims to help better understand the weaknesses in password selection techniques, why they no longer provide adequate security and the reason why organizations should start using better means of authentication.

Dictionaries [5] are raw text files consisting of one word or phrase per line. Each line is a candidate match where each hash is computed and compared to the hashes to be recovered. The difference between a Dictionary and a brute-force attack is that a Dictionary contains a list of probable matches rather than all possible string combinations [2]. A Dictionary needs to be well optimized otherwise if it includes any string combinations it risks becoming a brute-force attack and loses its efficiency. Therefore, Dictionaries often include known popular passwords, words from the English and other languages, ID numbers, phone numbers, sentences from books, etc.

(*Reference mentioned in the last page of the document).

Contribution

John the Ripper:

John the Ripper is a free password cracking software tool. Initially developed for the Unix operating system, it now runs on fifteen different platforms (eleven of which are architecture-specific versions of Unix, DOS, Win32, BeOS, and OpenVMS). It is one of the most popular password testing and breaking programs as it combines a number of password crackers into one package, autodetects password hash types, and includes a customizable cracker. It can be run against various encrypted password formats including several crypt password hash types most commonly found on various Unix versions (based on DES, MD5, or Blowfish), Kerberos AFS, and Windows NT/2000/XP/2003 LM hash. Additional modules have extended its ability to include MD4-based password hashes and passwords stored in LDAP, MySQL, and others.

One of the modes John can use is the dictionary attack. It takes text string samples (usually from a file, called a wordlist, containing words found in a dictionary or real passwords cracked before), encrypting it in the same format as the password being examined (including both the encryption

algorithm and key), and comparing the output to the encrypted string. It can also perform a variety of alterations to the dictionary words and try these. Many of these alterations are also used in John's single attack mode, which modifies an associated plaintext (such as a username with an encrypted password) and checks the variations against the hashes.

John the Ripper is often used in the enterprise to detect weak passwords that could put network security at risk, as well as other administrative purposes. The software can run a wide variety of password-cracking techniques against the various user accounts on each operating system and can be scripted to run locally or remotely.

bcrypt:

bcrypt is a password hashing function designed by Niels Provos and David Mazieres, based on the Blowfish cipher, and presented at USENIX in 1999. bcrypt is a hashing algorithm that is scalable with hardware (via a configurable number of rounds). Its slowness and multiple rounds ensure that an attacker must deploy massive funds and hardware to be able to crack your passwords. Add to that per-password salts (bcrypt REQUIRES salts) and you can be sure that an attack is virtually unfeasible without either ludicrous amount of funds or hardware.

Bcrypt is a cross-platform file encryption utility. Encrypted files are portable across all supported operating systems and processors. Passphrases must be between 8 and 56 characters and are hashed internally to a 448-bit key. However, all characters supplied are significant. The stronger your passphrase, the more secure your data.

Bcrypt uses the Eksblowfish algorithm to hash passwords. While the encryption phase of Eksblowfish and Blowfish are exactly the same, the key schedule phase of Eksblowfish ensures that any subsequent state depends on both salt and key (user password), and no state can be precomputed without the knowledge of both. Because of this key difference, bcrypt is a one-way hashing algorithm. You cannot retrieve the plain text password without already knowing the salt, rounds, and key (password).

In addition to encrypting your data, bcrypt will by default overwrite the original input file with random garbage three times before deleting it in order to thwart data recovery attempts by persons who may gain access to your computer. If you're not quite ready for this level of paranoia yet, see the installation instructions below for how to disable this feature. If you don't think this is paranoid enough see below.

Bcrypt Algorithm:

```
bcrypt (cost, salt, pwd)
  state ~ EksblowfishSetup (cost, salt, key)
  ctext ~ //Enter text to hashed//
  repeat (64)
    ctext ~ EncryptECB (state, ctext)
  return Concatenate (cost, salt, ctext)
```

Eksblowfish Algorithm:

```
EksBlowfishSetup (cost, salt, key)
```

```

    State ~ InitState()
    State ~ ExpandKey (state, salt, key)
repeat (2^cost)
    state ~ ExpandKey (state, 0, salt)
    state ~ ExpandKey (state, 0, key)
return state
//

```

EksblowfishSetup has three input parameters: a cost, a salt, and the encryption key. It returns a set of subkeys and S-boxes, also known as a *key schedule*. The cost parameter controls how expensive the key schedule is to compute. The salt is a 128-bit value that modifies the key schedule so that the same key need not always produce the same result. Finally, the key argument is a secret encryption key, which can be a user-chosen password of up to 56 bytes (including a terminating zero byte when the key is an ASCII string).

- EksBlowfishSetup begins by calling InitState, a function that copies the digits of the number first into the subkeys, then into the S-boxes.
 - EksBlowfishSetup begins by calling InitState, a function that copies the digits of the number first into the subkeys, then into the S-boxes. Subsequently, ExpandKey blowfish-encrypts the first 64 bits of its salt argument using the current state of the key schedule. The resulting ciphertext replaces subkeys P_1 and P_2 . That same ciphertext is also XORed with the second 64-bits of salt, and the result encrypted with the new state of the key schedule. The process continues, alternating between the first and second 64 bits salt. When ExpandKey finishes replacing entries in the P-Array, it continues on replacing S-box entries two at a time. After replacing the last two entries of the last S-box, $S_4[254]$ and $S_4[255]$, ExpandKey returns the new key schedule.
 - After calling InitState to fill a new key schedule with the digits of, EksBlowfishSetup calls Expand Key with the salt and key. This ensures that all subsequent state depends on both, and that no part of the algorithm can be precomputed without both salt and key. Thereafter, ExpandKey is alternately called with the salt and then key for iterations. For all but the first invocation of ExpandKey, the second argument is a block of 128 0 bits.
- //

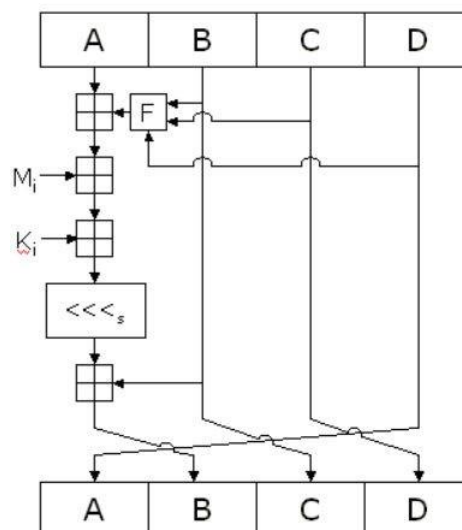
MD5 Algorithm:

- The MD5 algorithm first divides the input in **blocks** of 512 bits each. 64 Bits are inserted at the end of the last block. These 64 bits are used to record the length of the original input. If the last block is less than 512 bits, some extra bits are 'padded' to the end.
- Next, each **block** is divided into 16 **words** of 32 bits each. These are denoted as $M_0 \dots M_{15}$.
- MD5 uses a buffer that is made up of four words that are each 32 bits long. These words are called A, B, C and D. They are initialized as:
word A: 01 23 45 67
word B: 89 ab cd ef
word C: fe dc ba 98
word D: 76 54 32 10
- MD5 further uses a table K that has 64 elements. Element number i is indicated as K_i . The table is computed beforehand to speed up the computations. The elements are computed using the mathematical sin function:
$$K_i = \text{abs}(\sin(i + 1)) * 2^{32}$$
- In addition MD5 uses four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word. They apply the logical operators and, or, not and xor to the input bits.
$$F(X,Y,Z) = (X \text{ and } Y) \text{ or } (\text{not}(X) \text{ and } Z)$$

$$G(X,Y,Z) = (X \text{ and } Z) \text{ or } (Y \text{ and } \text{not}(Z))$$

$$H(X,Y,Z) = X \text{ xor } Y \text{ xor } Z$$

$$I(X,Y,Z) = Y \text{ xor } (X \text{ or } \text{not}(Z))$$
- The contents of the four buffers (A, B, C and D) are now mixed with the words of the input, using the four auxiliary functions (F, G, H and I). There are four *rounds*, each involves 16 basic *operations*. One operation is illustrated in the figure below.



The figure shows how the auxiliary function F is applied to the four buffers (A, B, C and D), using message word M_i and constant K_i . The item "<<<s" denotes a binary left shift by s bits.

Implementation

▣ Description of Modules/Programs – Hash suite:

Storing user passwords in the plain text naturally results in an instant compromise of all passwords if the password file is compromised. To reduce this danger, Windows applies a cryptographic hash function, which transforms each password into a hash, and stores this hash. This hash function is one-way in the sense that it is infeasible to infer a password back from its hash, except via the trial and error approach described below. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.

Hash Suite, like all other password hash crackers, does not try to "invert" the hash to obtain the password (which might be impossible). It follows the same procedure used by authentication: it generates different candidate passwords (keys), hashes them and compares the computed hashes with the stored hashes. This approach works because

users generally select passwords that are easy to remember, and as a side-effect, these passwords are typically easy to crack. Another reason why this approach is so very effective is that Windows uses password hash functions that are very fast to compute, especially in an attack (for each given candidate password).

□ Source Code

MD5:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef union uwb {
    unsigned w;
    unsigned char b[4];
} MD5union;

typedef unsigned DigestArray[4];
unsigned func0( unsigned abcd[] ){
    return ( abcd[1] & abcd[2]) | (~abcd[1] & abcd[3]);}
unsigned func1( unsigned abcd[] ){
    return ( abcd[3] & abcd[1]) | (~abcd[3] & abcd[2]);}
unsigned func2( unsigned abcd[] ){
    return abcd[1] ^ abcd[2] ^ abcd[3];}
unsigned func3( unsigned abcd[] ){
    return abcd[2] ^ (abcd[1] |~ abcd[3]);}
typedef unsigned (*DgstFctn)(unsigned a[]);

unsigned *calctable( unsigned *k)
{
    double s, pwr;
    int i;
    pwr = pow( 2, 32);
    for (i=0; i<64; i++) {
        s = fabs(sin(1+i));
        k[i] = (unsigned)( s * pwr );
    }
}
```

```
        return k;
    }
    unsigned rol( unsigned r, short N )

    {

        unsigned mask1 = (1<<N) -1;
        return ((r>>(32-N)) & mask1) | ((r<<N) & ~mask1);
    }
    unsigned *md5( const char *msg, int mlen)
    {
        static DigestArray h0 = { 0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476 };
        static DgstFctn ff[] = { &func0, &func1, &func2, &func3 }; static short M[] = { 1, 5,
        3, 7 }
```

```

static short O[] = { 0, 1, 5, 0 };
static short rot0[] = { 7,12,17,22};
static short rot1[] = { 5, 9,14,20};
static short rot2[] = { 4,11,16,23};
static short rot3[] = { 6,10,15,21};
static short *rots[] = {rot0, rot1, rot2, rot3 };
static unsigned kspace[64];
static unsigned *k;
static DigestArray h;
DigestArray abcd;

DgstFctn fctn; short
m, o, g; unsigned f;
short *rotn;

union {
    unsigned w[16];
    char b[64];
}mm;
int os = 0;
int grp, grps, q, p;
unsigned char *msg2;
if (k==NULL) k= calctable(kspace);
for (q=0; q<4; q++) h[q] = h0[q]; // initialize
{
    grps = 1 + (mlen+8)/64;
    msg2 = malloc( 64*grps);
    memcpy( msg2, msg, mlen);
    msg2[mlen] = (unsigned char)0x80;
    q = mlen + 1;
    while (q < 64*grps){ msg2[q] = 0; q++ ; }

    {
        MD5union u;
        u.w = 8*mlen;
        q -= 8;
        memcpy(msg2+q, &u.w, 4 );
    }
}

```

```

for (grp=0; grp<grps; grp++)

```

```

{
    memcpy( mm.b, msg2+os, 64);
    for(q=0;q<4;q++) abcd[q] = h[q]; for
    (p = 0; p<4; p++) {
        fctn = ff[p];
        rotn = rots[p];
        m = M[p]; o= O[p];
        for (q=0; q<16; q++) { g
            = (m*q + o) % 16;

            f = abcd[1] + rol( abcd[0]+ fctn(abcd) + k[q+16*p] + mm.w[g], rotn[q%4]);
abcd[0] = abcd[3];
abcd[3] = abcd[2];
abcd[2] = abcd[1];
abcd[1] = f;
        }
    }
    for (p=0; p<4; p++)
h[p] += abcd[p];
    os += 64;
}
return h;
}
int main( int argc, char *argv[] )
{
    FILE* file_ptr = fopen("md5.txt", "a+");
    int j,k;
    const char *msg = "This is a Network Security Project...";
    printf("-----\n");
    printf("-----Made by C codechamp-----\n");
    printf("-----\n\n");
    printf("\t MD5 ENCRYPTION ALGORITHM IN C
\n\n"); int n1;
    char inp[200];
    printf("Enter the value of inputs you want to hash or encrypt\n"); scanf("%d",&n1);
    int i1; for(int
    i1=0;i1<n1;i1++){
        scanf("%s",inp);

```

```

printf("Input String to be Encrypted using bcrypt :
\n\t%s",inp); unsigned *d = md5(inp, strlen(inp)); MD5union
u;

printf("\n\nThe bcrypt code for input string is :
\n"); printf("\t= 0x");
fprintf(file_ptr, "%s", "0x");
//fclose(file_ptr);

for (j=0;j<4; j++){
    u.w = d[j];
    for (k=0;k<4;k++){
fprintf(file_ptr, "%02x",u.b[k]);
        printf("%02x",u.b[k]);
    }
}
fprintf(file_ptr, "\n");
}
fclose(file_ptr);
printf("\n");
printf("\n\t bcrypt Encyption Successfully Completed!!!\n\n");
getch();
system("pause");
return 0;
}

```

PYTHON CODE:

bcrypt:

```

import bcrypt
def hash():
    plaintext = str(input("Enter the string to be hashed:"))
    b = bytes(plaintext, 'utf-8')
    b = plaintext.encode('utf-8')
    my_hashed_password = bcrypt.hashpw(b, bcrypt.gensalt())

```

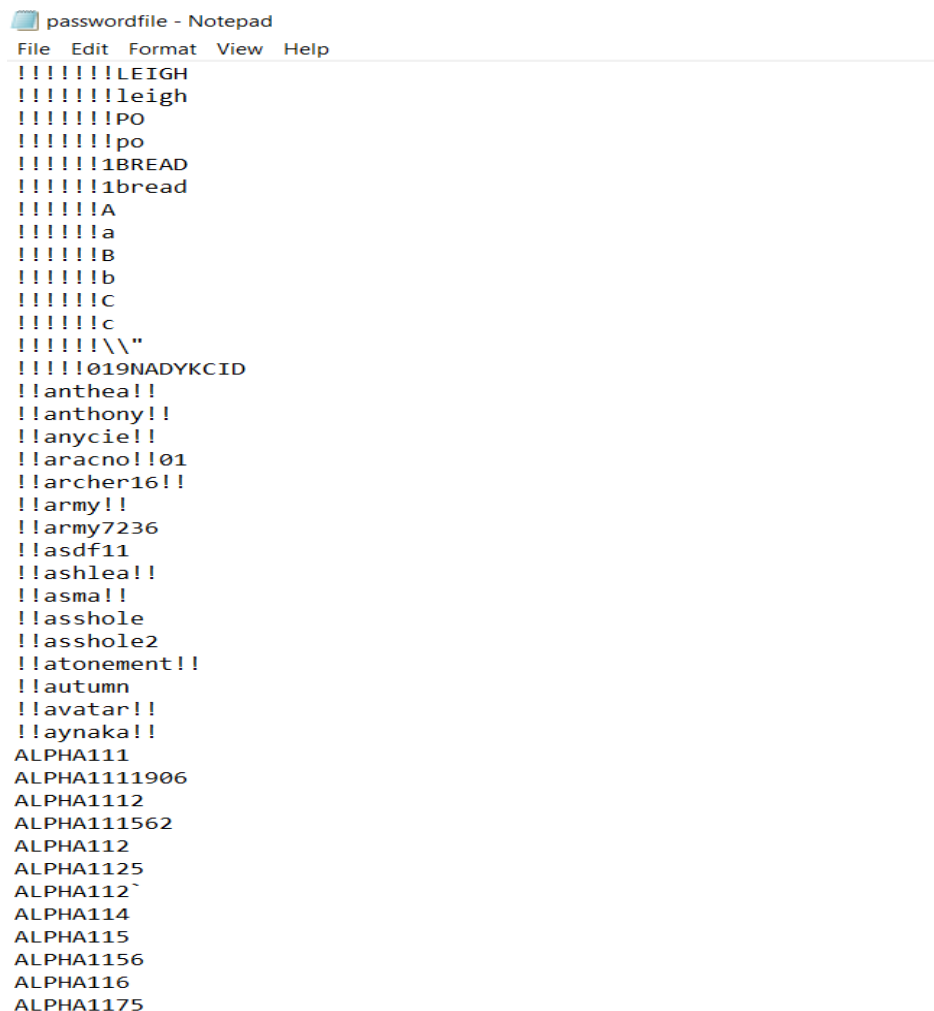
```
print(my_hashed_password)
n=int(input("Enter the number of strings to be hashed:"))
for i in range (0,n):
    hash()
```

4.3 Test Case

- BRITANIA123
- Monkey2011

4.1 Execution Snapshot

John The Ripper:



```
passwordfile - Notepad
File Edit Format View Help
!!!!!!!!LEIGH
!!!!!!!!leigh
!!!!!!!!PO
!!!!!!!!po
!!!!!!!!1BREAD
!!!!!!!!1bread
!!!!!!A
!!!!!!a
!!!!!!B
!!!!!!b
!!!!!!C
!!!!!!c
!!!!!!\\"
!!!!!!019NADYKCID
!!anthea!!
!!anthony!!
!!anyone!!
!!aracno!!01
!!archer16!!
!!army!!
!!army7236
!!asdf11
!!ashlea!!
!!asma!!
!!asshole
!!asshole2
!!atonement!!
!!autumn
!!avatar!!
!!aynaka!!
ALPHA111
ALPHA1111906
ALPHA1112
ALPHA111562
ALPHA112
ALPHA1125
ALPHA112`
ALPHA114
ALPHA115
ALPHA1156
ALPHA116
ALPHA1175
```

```

C:\john\run>john -help
John the Ripper 1.9.0-jumbo-1 OMP [cygwin 64-bit x86_64 AVX2 AC]
Copyright (c) 1996-2019 by Solar Designer and others
Homepage: http://www.openwall.com/john/

Usage: john [OPTIONS] [PASSWORD-FILES]
--single[=SECTION[,...]]    "single crack" mode, using default or named rules
--single=:rule[,...]        same, using "immediate" rule(s)
--wordlist[=FILE] --stdin   wordlist mode, read words from FILE or stdin
                           --pipe   like --stdin, but bulk reads, and allows rules
--loopback[=FILE]           like --wordlist, but extract words from a .pot file
--dupe-suppression          suppress all dupes in wordlist (and force preload)
--prince[=FILE]             PRINCE mode, read words from FILE
--encoding=NAME             input encoding (eg. UTF-8, ISO-8859-1). See also
                           doc/ENCODINGS and --list=hidden-options.
--rules[=SECTION[,...]]     enable word mangling rules (for wordlist or PRINCE
                           modes), using default or named rules
--rules=:rule[;...]         same, using "immediate" rule(s)
--rules-stack=SECTION[,...] stacked rules, applied after regular rules or to
                           modes that otherwise don't support rules
--rules-stack=:rule[;...]   same, using "immediate" rule(s)
--incremental[=MODE]        "incremental" mode [using section MODE]
--mask[=MASK]               mask mode using MASK (or default from john.conf)
--markov[=OPTIONS]          "Markov" mode (see doc/MARKOV)
--external=MODE             external mode or word filter
--subsets[=CHARSET]         "subsets" mode (see doc/SUBSETS)
--stdout[=LENGTH]           just output candidate passwords [cut at LENGTH]
--restore[=NAME]            restore an interrupted session [called NAME]
--session=NAME              give a new session the NAME
--status[=NAME]             print status of a session [called NAME]
--make-charset=FILE         make a charset file. It will be overwritten
--show[=left]               show cracked passwords [if =left, then uncracked]
--test[=TIME]               run tests and benchmarks for TIME seconds each
--users=[-]LOGIN|UID[,...]  [do not] load this (these) user(s) only
--groups=[-]GID[,...]       load users [not] of this (these) group(s) only
--shells=[-]SHELL[,...]     load users with[out] this (these) shell(s) only
--salts=[-]COUNT[:MAX]    load salts with[out] COUNT [to MAX] hashes
--costs=[-]C[:M][,...]      load salts with[out] cost value Cn [to Mn]. For
                           tunable cost parameters, see doc/OPTIONS
--save-memory=LEVEL         enable memory saving, at LEVEL 1..3
--node=MIN[-MAX]/TOTAL      this node's number range out of TOTAL count
--fork=N                    fork N processes
--pot=NAME                  pot file to use
--list=WHAT                 list capabilities, see --list=help or doc/OPTIONS
--devices=N[,...]           set OpenCL device(s) (see --list=opencl-devices)
--format=NAME               force hash of type NAME. The supported formats can
                           be seen with --list=formats and --list=subformats

```

```
C:\john\run>john passwordfile.txt
Warning: detected hash type "descrypt", but the string is also recognized as "descrypt-opencl"
Use the "--format=descrypt-opencl" option to force loading these as that type instead
Warning: only loading hashes of type "descrypt", but also saw type "tripcode"
Use the "--format=tripcode" option to force loading hashes of that type instead
Warning: only loading hashes of type "descrypt", but also saw type "pix-md5"
Use the "--format=pix-md5" option to force loading hashes of that type instead
Using default input encoding: UTF-8
Loaded 4 password hashes with 3 different salts (descrypt, traditional crypt(3) [DES 256/256 AVX2])
Will run 8 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:password.lst, rules:Wordlist
Proceeding with incremental:ASCII
Warning: MaxLen = 13 is too large for the current hash type, reduced to 8
0g 0:00:01:51 3/3 0g/s 3103Kp/s 9310Kc/s 12414Kc/s p1n350f..p0p1lmd
0g 0:00:01:56 3/3 0g/s 3150Kp/s 9452Kc/s 12603Kc/s gizacke..gizng8s
0g 0:00:02:06 3/3 0g/s 3297Kp/s 9893Kc/s 13192Kc/s cyby1256..cychmc23
Session aborted
```

```
C:\john\run>john -format=RAW-MD5 passwordfile.txt
Using default input encoding: UTF-8
Loaded 2 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])
Warning: no OpenMP support for this hash type, consider --fork=8
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Warning: Only 23 candidates buffered for the current salt, minimum 24 needed for performance.
Warning: Only 22 candidates buffered for the current salt, minimum 24 needed for performance.
Almost done: Processing the remaining buffered candidate passwords, if any.
Warning: Only 2 candidates buffered for the current salt, minimum 24 needed for performance.
Proceeding with wordlist:password.lst, rules:Wordlist
monkey (user1)
Proceeding with incremental:ASCII
1g 0:00:00:32 3/3 0.03082g/s 1036Kp/s 1036Kc/s 1037Kc/s pamonn7..pamo188
monkey2011 (user2)
2g 0:00:01:05 DONE 3/3 (2020-10-19 17:29) 0.03056g/s 4873Kp/s 4873Kc/s 4873Kc/s monked13..monkentord
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed
```

```
C:\john\run>john --show --format=Raw-MD5 passwordfile.txt
```

```
C:\john\run>john -show -format=Raw-MD5 passwordfile.txt
user1:monkey
user2:monkey2011

2 password hashes cracked, 0 left
```

bcrypt:

```
Enter the number of strings to be hashed:2
Enter the string to be hashed:Britania123
0.6416952610015869
b'$2b$12$odkNwxfrQ.6w0DU/bRMtE.'
b'$2b$12$aFRK/BZsSButk2Ju.eLK9u9ihDrFqVMdYM4lYWPD1ZhyFGLPVpuZO'
Enter the string to be hashed:monkey2011
0.6445834636688232
b'$2b$12$X.Oo1iZLLG6gHSpfTU3s2e'
b'$2b$12$sFUJJNlCtkQyxM0P65e2feokh2pN1VZMD3gxVEEeS2SYM/Y2o10Ra'
```

Inputs followed by Outputs:

▣ **BRITANIA123:**

b'\$2b\$12\$aFRK/BZsSButk2Ju.eLK9u9ihDrFqVMdYM4lYWPD1ZhyFGLPVpuZO'

▣ **monkey2011:**

b'\$2b\$12\$sfUJJNlCtkQyxM0P65e2feokh2pN1VZMD3gxVEEeS2SYM/Y2o1ORa '

MD5:

```
C:\Users\Dell\Desktop\md5.exe
MD5 ENCRYPTION ALGORITHM IN C

Enter the value of inputs you want to hash or encrypt
2
Britania123
Input String to be Encrypted using bcrypt :
    Britania123

The bcrypt code for input string is :
    = 0x17e9e3fdb7f23d670ff3bfaa4a4c76d
monkaey2011
Input String to be Encrypted using bcrypt :
    monkaey2011

The bcrypt code for input string is :
    = 0xa05fc47550b510988db761d94b1ba13d

    bcrypt Encyption Successfully Completed!!!
```

INPUTS Followed by Outputs:

- BRITANIA123 – 0x17e9e3fdb7f23d670ff3bfaa4a4c76d
- Monkey2011 – 0xa05fc47550b510988db761d94b1ba13d

Conclusion

So with a one-way hash password, a server does not store plain text passwords to authenticate a user. A password has a hashing algorithm applied to it to make it more secure. When it comes to encryption and hashing faster is never better. bcrypt can expand what is called its Key Factor to compensate for increasingly more-powerful computers and effectively “slow down” its hashing speed.

Salting will increase the time needed to find other user's passwords. Because the attacker would need to create a Rainbow Table for every salt used because salts change the output in unpredictable ways. Cracking the hash for one user wouldn't be much harder but cracking the hash for all users would be exponentially harder. Strong salts should be used in conjunction with strong passwords.

In Comparison to MD5, bcrypt hashes the password ‘yaaa’ in about 0.3 seconds on the other hand MD5 takes less than a microsecond. So as slower the better bcrypt is preferable MD5. It is not vulnerable to rainbow tables (since creating them is too expensive) and not even vulnerable to brute force attacks. However, 11 years later, many are **still** using SHA2x with salt for storing password hashes and **bcrypt** is not widely adopted.

References:

1. Wiemer, Friedrich & Zimmermann, Ralf. (2015). High-speed implementation of bcrypt password search using special-purpose hardware. 2014 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2014. 10.1109/ReConFig.2014.7032529.
2. Bošnjak, Leon & Sres, J. & Brumen, B.. (2018). Brute-force and dictionary attack on hashed real-world passwords. 1161-1166. 10.23919/MIPRO.2018.8400211.
3. Yiannis, Chrysanthou , Modern Password Cracking: A hands-on approach to creating an optimised and versatile attack.
4. Raza, Mudassar & Iqbal, Muhammad & Sharif, Muhammad & Haider, Waqas. (2012). A Survey of Password Attacks and Comparative Analysis on Methods for Secure Authentication. World Applied Sciences Journal. 19. 439-444. 10.5829/idosi.wasj.2012.19.04.1837.
5. Madiraju, Tarun, "Dictionary Attacks and Password Selection" (2014). Thesis. Rochester Institute of Technology.
6. Yiannis, Chrysanthou , Modern Password Cracking: A hands-on approach to creating an optimised and versatile attack.
7. Techopedia – home/dictionary/tags/Security/Dictionary Attacks.
8. theEconomicTimes/definition/cryptography
9. learcryptography.com(attack-vectors/dictionary-attack)
10. learcryptography.com(attack-vectors/dictionary-attack)
11. cs.cmu.edu(lectures/Hashing)
12. https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/provos/provos_html/node5.html
13. https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/provos/provos_html/node4.html
14. <http://www.faqs.org/rfcs/rfc1321.html>
15. <https://www.iusmentis.com/technology/hashfunctions/md5/>