Password Security

CSCE 499 Capstone

Daniel Case

Bachelors of Arts in Computer Science

Bachelors of Science in Mathematics

Faculty Advisor: Dr. George Hauser

May 15, 2014

Contents

Abstract

This research project serves as an analysis of two important concepts in password security: cryptographic hash functions and common password cracking techniques. Among the topics to be covered in this document are the pseudocode of the SHA-1, SHA-256, and MD5 algorithms, and the design of the Brute-Force Attack, Dictionary Attack, and Rainbow Table attack.

1 Introduction

Computer security is an integral part of software engineering. Entire systems can be compromised from the smallest errors or misuse of information. One of the most important and fundamental aspects of computer security is the concept of password security.

A *password* is a string of characters or a word that is used for authentication to prove identity or authorization to gain access to a resource that is required to be kept secret.[1] This infers that they are used as a means to control login attempts of various users or the switching of user privileges on a system. Passwords are, in short, used to safeguard sensitive information by restricting access.[2] The focus of this document is two-fold: one is research regarding common algorithms to securely obfuscate a password, and the other section of research is in regard to the process to how passwords are commonly attacked or compromised, informally called *password cracking*.[8]

2 Password Security Overview

Whether it is the case when a user is logging into an account or elevating privileges, the abstraction of how a password is used remains the same. A user will type in a string that is then compared to one that is stored somewhere on the system. If the strings match, the user is granted access or authorization to continue whatever task necessary.

2.1 Password Storage

Different operating systems have alternate ways of implementing the storage of passwords. Operating systems contain various system calls and software instructions that together are called a *password management system* or simply a *password manager*[3]. In order to be as secure as possible, password managers do not store what is referred to as the *plaintext* of a password. Plaintext in the context of password security is unaltered data that has need to be kept secret or concealed.[4] The password manager ensures that the password itself as plaintext is not stored in a file since a security breach is imminent if the password file were to be compromised. Instead a password is encoded via a *cryptographic hash function* and then the resulting hash value is stored. For example, in Windows the password is hashed and then stored within the registry using the Windows password management system SAM, short for Security-Accounts Manager. In recent Unix-like systems, the password is hashed and then a copy of the hash value is stored within a hidden directory called /etc/shadow/password. In OSX user passwords are kept hashed in /var/db/dslocal/nodes/Default/users.

2.2 Cryptographic Hash Functions

A *cryptographic hash function* is an algorithm that maps any input string of arbitrary length into an output bit-string of a fixed length.[5] Typically the input string that is chosen to be encoded is referred to as the *input* or *message* whereas the output string is called the *hash*, *hash value*, or *digest*.

A *cryptographic hash function* has three special properties that define it as being *cryptographically secure*:[6]

1. Given a hash h, it should be infeasible to find any message m such that h=*hash*(m). This is called *preimage resistance*.

2. Given an input m_1, it should be infeasible to find another input m_2, such that $m_1 \neq m_2$ and *hash*(m_1) = *hash*(m_2). This is referred to as *second-preimage resistance*.

3. It is infeasible to find two different messages, m_1 and m_2 such that *hash*(m_1) = *hash*(m_2). This is also referred to as *collision resistance*.[7]

   Corollary: *Collision resistance implies second-preimage resistance but does not guarantee preimage resistance.*[6]

What it means to be cryptographically secure is directly related to whether or not an operation is considered infeasible. *Feasibility* has to do with the computational complexity of the action being performed.[8][6]

In a cryptographic sense, an action that is infeasible means that it is almost certainly beyond the reach of any adversary who must be prevented from breaking the system.[6] It also means in a mathematical sense that the action cannot be fulfilled in asymptotic polynomial time. Meaning, for a cryptographic hash function with a digest length of n-bits, finding a message that corresponds to a given digest can always be done in at most 2^n evaluations.[9][11] Therefore it has a time complexity of $\mathcal{O}(2^n)$. Hence for hashes with relatively larger digest lengths it would be considered infeasible to generate all password combinations.

2.3 CSPRNG

A *cryptographic pseudo random number generator*, or as it is most commonly called, a *CSPRNG*, is an algorithm for generating a cryptographically secure sequence of numbers that approximates the properties of random numbers.[24] This sequence of numbers is not truly random since the sequence is determined by a small set of values called the *state* and by an initial value called a *seed*. Each number thereafter is calculated using the current state and seed.

The *seed* is a number chosen by combining some of the values in memory of keyboard typings, mouse movement, and disc activities. This value is then hashed via a cryptographic hash function to generate the next number in the CSPRNG. After a set amount of iterations, a new seed is generated and then hashed to continue the sequence.

A CSPRNG must satisfy two requirements in order to be considered cryptographically secure:[24]

1. A CSPRNG should satisfy the *next-bit test*. That is, given the first k bits of the sequence, it is infeasible to predict the $(k+1)^{\text{th}}$ bit with probability greater than 50 percent.

2. A CSPRNG should withstand *state compromise extensions.* That is, in the event that part or all of its state has been revealed, it is infeasible to reconstruct the sequence prior to revelation.

2.4 Salts

A *salt* is a random set of bytes of fixed length that is used as additional input for a cryptographic hash function.[25] Usually the salt is generated by choosing a number made by a CSPRNG, and then hashing the value. The salt and password are then concatenated into a new input string and processed by a cryptographic hash function. The resulting hash is then stored with the salt value.

Salts are used to increase the difficulty and cost of precomputed attacks, such as the use of Rainbow Tables, and to make Dictionary attacks infeasible if the salt is not known or marginally more computationally expensive if the salt is known. This topic is discussed in greater detail in the Password Cracking Limitations and Concluding Thoughts section.

2.5 Cryptographic Hash Function Weaknesses

The following two sections refer to mathematical principles that effect the security of cryptographic hash functions. The definition and theorems that are mentioned do not directly effect the design or the implementation of the chosen password cracking techniques, but serve instead as an insight as to why there exists an upper-bound on the number of computations necessary for a Brute-Force attack[7][10] or as to why some cryptographic hash functions are considered compromised.[7]

2.5.1 Collisions and the Pigeon Hole Principle

The properties of a cryptographic hash function imply that a malicious adversary cannot replace or modify the input string without changing the hash. Therefore if two strings have the same hash it can be ascertained with high probability that the input strings are identical, unless there is a collision. A *hash collision* is a situation where two distinct pieces of data have the same hash value.[7] This can happen whenever values of a large set are mapped to relatively shorter bit-strings. Property 3 does not mean that two different input strings cannot produce the same hash, merely that it is computationally hard to do so. However due to the *Pigeon Hole Principle*, for any cryptographic hash function there exists a collision after a certain number of computations.[10] An upper-bound on the number of computations required is outlined in the The Birthday Paradox section.

The *Pigeon Hole Principle* states that:

. *If n discrete objects are to be allocated to m containers, then at least one container must hold no fewer than* $\lceil n/m \rceil$ *objects.*

We also have the following *Pigeon Hole Corollary*:

. *If* $n > m$ *keys are hashed into m slots, then two keys will collide.*

Each cryptographic hash function will produce a fixed m bits of output from n bits of input, however $n>m$ since the input is the set of all words possible for a password and the resultant hash is a fixed-length bit-string. In terms of hash collisions this means that for a finite amount of input strings being encoded into another finite amount of hash values then there will always be at least one hash value that can be generated from two different input strings.[10]

2.5.2 The Birthday Paradox

The *Birthday Paradox* is a generalized probability concept based on the Pigeon Hole Principle. It concerns the probability that, in a set of n randomly chosen people, some pair of them will have the same birthday. In the context of hash functions, the formula in its generalized form creates an upper bound on the number of times a hash function can be computed before a collision is found.[10] The formula and general rule can be stated as:

. *If a hash function produces n-bits of output, then* $\sqrt{2^n}(2^{n/2})$ *hash operations will produce a collision.*

This general case of The Birthday Paradox leads to consequences in terms of cryptographic security. Another metric for whether or not a hash function is cryptographically secure is to test when a collision occurs after a number of hash operations. If a collision happens in a number of operations less than specified by The Birthday Paradox then the hash function is deemed to be insecure and cryptographically flawed.[7]

3 Design Methodology

The cryptographic hash functions, SHA-1, SHA-256, and MD5 all share similar properties, namely their use of a *compression function* and their construction known as *Merkle-Damgard Construction.*[22][23] The next few sections detail the properties and structure behind Merkle-Damgard Construction, Compression Functions, as well as the pseudocode used in their implementation.[17][18][19][20][21]

3.1 Compression Functions

A *compression function* or *one-way compression function* is a function that transforms two fixed-length input strings into one fixed-length output string.[23] More specifically, a compression function combines two fixed-length input strings and produces an output-string that is the same size in bits as one of the inputs.[23] The compression function uses a series of bit-wise logical operations within a loop in order to achieve this.[23]

3.2 Merkle-Damgard Construction

Merkle-Damgard Construction is a method of creating cryptographic hash functions using a technique called *length padding* and a compression function.[22]

The algorithm specifies several nested loops. A padding function (length padding), or pre-processing loop, pads the input message with the 0-bit until the input is a multiple of a fixed number of bits. The result is then broken into blocks of fixed size, and then processed through the compression function one at a time. Through each iteration, a new block of input is combined with the block of the output from the previous round.[22] The final block of output is then appended with bits representing the length of the entire message.

This creates what is called the *Avalanche effect.* Since each output block created between iterations is processed with each input block, by changing even one bit of the original message can drastically change the final output hash value.[22]

3.3 Cryptographic Hash Function Algorithms

All three cryptographic hash functions researched use a different combination of bit-wise logical operations.[19][20][21] The operations are the circular shifts (rotate no-carry): *left-rotate* and *right-rotate*, and logical shifts: *left-shift* and *right-shift* (since the integers are unsigned).

The following are the respective function definitions in pseudocode. **Note:** The << and >> symbols specifically refer to the bitwise operation, the logical shift, in C. That is, x << n is obtained by discarding the left-most n bits of the word x and then padding the result with n zeros on the right. Similarly, x >> n is obtained by discarding the right-most bits of the word n of the word x and padding the result with n zeros on the left.

//Left-rotate function definition:
leftrotate (x, c)
return (x << c) **or** (x >> (32 - c));

//Right-rotate function definition:
rightrotate (x, c)
return (x >> c) **or** (x << (32 - c));

//Left-shift function defintion:
leftshift (x, c)
return (x << c);

//Right-shift function definition:
rightshift (x, c)
return (x >> c);

3.3.1 SHA-1

SHA-1 produces a 160-bit digest. This section details the pseudocode used in its implementation.[17][19]

1. All variables are 32-bit unsigned integers, except for *ml*, the message length, which is appended as a 64-bit integer (bit-string), and the message digest, *hh*, which is 160-bits.
2. All variables and constants seen in the pseudocode are in big-endian. That is, the most significant byte is stored in the leftmost byte position.
3. The four constants, k, in the for loop is the number 2^{30} multiplied by $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{10}$ respectively.
4. The five initialized variables, $h0$ through $h4$, when converted into little-endian, are the following hexadecimal values:

   h0: 0x01234567
   h1: 0x89ABCDEF
   h2: 0xFEDCBA98
   h3: 0x76543210
   h4: 0xF0E1D2C3

Pseudocode:

//Initialize hash variables:
h0 = 0x67452301
h1 = 0xEFCDAB89
h2 = 0x98BADCFE
h3 = 0x10325476
h4 = 0xC3D2E1F0

```
ml = message length in bits

//Pre-processing: adding a single 1 bit:
append "1" bit to message

//Pre-processing: padding with zeros
append "0" bit until ml ≡ 448  mod 512

//Pre-processing: append ml as a 64-bit big-endian integer to message
append ml mod (2 pow 64) to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message do
   create an eighty-entry message schedule array, w[i], 0 ≤ i ≤ 79, of 32-bit big-endian words //words in
   array are initialized to 0

   copy chunk into first sixteen words of message schedule array, w[0...15]

   //Extend the sixteen words into the remaining sixty-four words, w[16...79], of the message schedule
   array:
   for i = 16 to 79 do
      w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1
   end for

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

   //Main Loop (Compression function):
   for i = 0 to 79 do
      if 0 ≤ i ≤ 19 then
         f = (b and c) or ((not b) and d)
         k = 0x5A827999
      else if 20 ≤ i ≤ 39 then
         f = b xor c xor d
         k = 0x6ED9EBA1
      else if 40 ≤ i ≤ 59 then
         f = (b and c) or (b and d) or (c and d)
         k = 0x8F1BBCDC
      else if 60 ≤ i ≤ 79 then
         f = b xor c xor d
         k = 0xCA62C1D6
      end if

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

end for

//Add the compressed chunk to the current hash value:
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
end for

//Produce the final hash value (big-endian):
hh = (h0 **leftshift** 128) **or** (h1 **leftshift** 96) **or** (h2 **leftshift** 64) **or** (h3 **leftshift** 32) **or** h4
□

3.3.2 SHA-256

SHA-256 produces a 256-bit digest. This section details the pseudocode used in its implementation.[17][20]

1. All variables are 32-bit unsigned integers, except the message length, *ml*, which is appended as a 64-bit integer (bit-string), and the message digest, *hh*, is 256-bits.
2. For each round of the loop, there is one round constant k[i], and one entry in the message schedule array w[i], where $0 \leq i \leq 63$
3. The main loop uses eight variables, a through h.
4. All variables and constants are in big-endian. The initialized constants, $h0$ through $h7$, are the first 32-bits of the fractional parts of the square roots of the first eight primes: 2...19. The round constants, $k[0...63]$, are the first 32-bits of the fractional parts of the cube roots of the first sixty-four primes: 2...311.

Pseudocode:

//Initialize hash variables:
h0 = 0x6A09E667
h1 = 0xBB67AE85
h2 = 0x3C6EF372
h3 = 0xA54FF53A
h4 = 0x510E527F
h5 = 0x9B05688C
h6 = 0x1F83D9AB
h7 = 0x5BE0CD19

ml = message length in bits

//Initialize array of round constants, k[i], $0 \leq i \leq 63$
k[0...63] =
0x428A2F98
0x71374491
0xB5C0fBCF
0xE9B5DBA5
0x3956C25B
0x59F111F1
0x923F82A4
0xAB1C5ED5
...

0xC67178F2

//Pre-processing: adding a single 1 bit
append "1" bit **to** message

//Pre-processing: padding with zeros
append "0" bit **until** ml $\equiv$ 448 mod 512

//Pre-processing: append ml as a 64-bit big-endian integer to message
append ml mod (2 **pow** 64) **to** message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message **do**
  create a 64-entry *message schedule array*, w[i], $0 \leq i \leq 63$, of 32-bit big-endian words *//words in the array are initialized to 0*

  copy chunk into first sixteen words, w[0...15], of the message schedule array

  //Extend the sixteen words into the remaining forty-eight words, w[16...63], *of the message schedule array:*
  for $i = 16$ **to** 63 **do**
    s0 = (w[i-15] **rightrotate 7**) **xor** (w[i-15] **rightrotate** 18) **xor** (w[i-15] **rightshift** 3)
    s1 = (w[i-2] **rightrotate** 17) **xor** (w[i-2] **rightrotate** 19) **xor** (w[i-2] **rightshift** 10)
    w[i] = w[i-16] + s0 + w[i-7] + s1
  end for

  //Initialize hash values for this chunk:
  a = h0
  b = h1
  c = h2
  d = h3
  e = h4
  f = h5
  g = h6
  h = h7

  //Main loop (*Compression function*)*:*
  for $i = 0$ **to** 63 **do**
    S1 = (e **rightrotate** 6) **xor** (e **rightrotate** 11) **xor** (e **rightrotate** 25)
    ch = (e **and** f) **xor** ((**not** e) **and** g)
    temp1 = h + S1 + ch + k[i] + w[i]
    S0 = (a **rightrotate** 2) **xor** (a **rightrotate** 13) **xor** (a **rightrotate** 22)
    maj = (a **and** b) **xor** (a **and** c) **xor** (b **and** c)
    temp2 = S0 + maj
    h = g
    g = f
    f = e
    e = d + temp1
    d = c
    c = b
    b = a
    a = temp1 + temp2
  end for

//Add the compressed chunk to the current hash value:
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
h5 = h5 + f
h6 = h6 + g
h7 = h7 + h
end for

//Produce the final hash value (big-endian):
hh = h0 **append** h1 **append** h2 **append** h3 **append** h4 **append** h5 **append** h6 **append** h7
□

3.3.3 MD5

MD5 produces a 128-bit digest. This section details the pseudocode used in its implementation.[18][21]

1. All variables and constants are in little-endian.
2. All variables and constants are 32-bit unsigned integers except for the message length, *ml*, which is appended as a 64-bit integer (bit-string), and the final hash value, *hh*, which is 128-bits.
3. The round constants, k[i], $0 \leq i \leq 63$, are binary representations of the integer part of the sines of integers (in radians).

Pseudocode:

//Initialize arrays for round constants, k, and round-shifts, s:
int[64] s, k

//Array s specifies the per-round shift amounts:
s[0...15] = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22]
s[16..31] = [5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20]
s[32..47] = [4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23]
s[48..63] = [6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

//Use binary integer part of the sines of integers:
for $i = 0$ **to** 63 **do**
  k[i] = $\lfloor |\sin(i+1)| (2$ **pow** $32) \rfloor$
end for

//Initialize hash variables:
a0 = 0x67452301
b0 = 0xEFCDAB89
c0 = 0x98ABCDFE
d0 = 0x10325476

//Pre-processing: adding a single 1 bit
append "1" bit **to** message

//Pre-processing: padding with zeros
append "0" bit **until** ml $\equiv 448 \mod 512$

//Pre-processing: append ml as a 64-bit big-endian integer to message

```
append ml mod (2 pow 64) to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of message do
  break chunk into sixteen 32-bit chunks, M[i], 0 ≤ i ≤ 15
  //Initialize hash value for this chunk:
  A = a0
  B = b0
  C = c0
  D = d0

  Main loop (Compression function:)
  for i = 0 to 63 do
    if 0 ≤ i ≤ 15 then
      F = (B and C) or ((not B) and D)
      g = i
    else if 16 ≤ i ≤ 31 then
      F = (D and B) or ((not D) and C)
      g = (5*i + 1) mod 16
    else if 32 ≤ i ≤ 47 then
      F = B xor C xor D
      g = (3*i + 5) mod 16
    else if 48 ≤ i ≤ 63 then
      F = C xor (B or (not D))
      g = (7*i) mod 16
    end if

    dTemp = D
    D = C
    C = B
    B = B + leftrotate((A + F + k[i] + M[g]), s[i])
    A = dTemp
  end for

  //Add this chunk's hash to the result:
  a0 = a0 + A
  b0 = bo + B
  c0 = c0 + C
  d0 = d0 + D
end for

//Produce the final hash value (little-endian):
hh = a0 append b0 append c0 append d0
```

□

3.4 Password Cracking Techniques

3.4.1 Introduction to Preimage Attacks

The password cracking techniques detailed in this document are all classified as *preimage attacks*. This is an attack where if given the hash function and the password hash, h, an attempt is made to compromise the password by generating a string, m, and then using the hash function to compare $hash(m)$ to h.[8][26]

3.4.2 Brute-Force Attack

A *Brute-Force Attack* attempts to obtain a password by generating a set of messages one character at a time, and then hashing the generated message and comparing it to the hash value of the stored password. The starting values for determining the first message and every generated message afterwards is given by an input *alphabet*. An alphabet in this context means the set of characters and the estimated length used for the original password. Once a message is generated whose hash matches the password hash, then the attack is deemed successful: the message generated was the stored password, and the attack halts and the message is returned.

The Brute-Force Attack has a time complexity of $\mathcal{O}(2^n)$, where n is the length of the digest produced by the cryptographic hash function.[11] Hence, evaluating a cryptographic hash function with an n-bit digest can take a maximum of 2^n evaluations to search through all password combinations.[9][10] This is because of the nature of generating subsets; it can be assumed in some cases that the entire character set will have to be searched. Typically this means that all alphabetical characters contained in the range of A through Z, all number combinations from 0 through 9, and all special characters will need to be considered. Generally the number of combinations to try is determined by $numCharset^{length}$[11], where *numCharset* is the number of characters in the character set and *length* is and estimated length of the original password. If any information about the length of the password or the characters used in the character set is known, then this number is bounded above by The Birthday Paradox.[7][9][11]

The algorithm of a Brute-Force Attack is as follows:[12]

Pseudocode:

Declared variables:
P *//An instance of our problem*
c *//A candidate for solving the problem instance* **P**

Methods:
first(P) *//Generates the first candidate for* **P**
next(P, c) *//Generates the next candidate*
valid(P, c) *//Checks whether a candidate c is a solution for problem instance* **P**
output(P, c) *//Outputs* ***P*** *and* ***c***

Function:
c = first(P)
while (**c** is in the search space) **do**
  if valid(P, c) then
    output(P, c)
  end if
  c = next(P, c)
end while

□

3.4.3 Dictionary Attack

A *Dictionary Attack* is a type of password cracking technique that shares some ties with the Brute-Force Attack; they both systematically check password candidates. However it is fundamentally different from the Brute-Force Attack in that it does not need to generate strings for the candidates. A large, pre-built table or list of words is used instead. The entries in this file are hashed using the given cryptographic hash function, and then the hashed values are compared to the password hash. If any of the hashes match then the plaintext listed in the table is the password; the attack is halted and that entry is returned.[13][14]

3.4.4 Rainbow Table Attack

A *Rainbow Table Attack* is a password cracking technique that is a Space/Time Trade-off algorithm based on using a precomputed table of hash values.[15][27] The Rainbow Table is first constructed by using a very large input file. This file is either similar to the word-list in a Dictionary Attack or is generated from the ground up from an alphabet, similar to the Brute-Force Attack. Like the other password cracking methods mentioned in this document, a Rainbow Table can only be made if the hash function of the hashed password is known.[15]

The core composition of a Rainbow Table is based on the use of *reduction functions* and the storage of *hash chains*.[15][16][27]

A *reduction function* is a function that maps a hashed value into a possible password candidate, sometimes refereed to as a new plaintext. The reduction function transforms the hash values into plaintext based on the alphabet of the input file. These new plaintext values are then hashed, and the resulting hash values are transformed using another reduction function. A *hash chain* starts with a column of plaintext from the input file, and then another column is made by the previous entries hashed row by row. These new hash values are then transformed by a reduction function, and the new plaintext is stored in another column.[27] The process then repeats itself. It is important to note that in order to gain more password candidates in a hash chain, a new reduction function is typically defined for each column of new plaintext.[27]

The final structure of a Rainbow Table, being a Space/Time Trade-off, is a smaller table where the only columns are the initial plaintext from the input file and the last hash value in the hash chain that is generated from each row.[16] Below is the pseudocode of the algorithm used in processing the hash chains in a Rainbow Table Attack:[16]

Pseudocode:

1. Search for the hashed value in the Rainbow Table. If it is found, goto step 4. If not:
2. Starting with the last reduction function, *reduce* the hashed value to get a new plaintext. (Every time Step 1 is repeated, use the next lowest reduction function.)
3. Hash the new plaintext and repeat Step 1 from the beginning with the new hash value.
4. Take the corresponding plaintext value in the Rainbow Table and hash it.
5. Compare the target hash.
6. If they match - the attack is successful. The plaintext that was hashed was the password plaintext.
7. If not, apply the reduction function to get a new plaintext value, and then goto step 2.

□

4 Password Cracking Limitations and Concluding Thoughts

As mentioned briefly in the Introduction to Preimage Attacks section, it is important to note that since the Brute-Force Attack, Dictionary Attack, and Rainbow Table Attacks are preimage attacks they all are bound to the following constraints:[26]

1. The list of hashed passwords has to be accessible.
2. The cryptographic hash function that hashed the passwords needs to be known.
3. The attacks must be made offline.

The reasoning behind these constraints stems from the nature of preimage attacks. In order to know when an attack has been completed successfully, a comparison between a hashed password candidate must be made versus the hashed value of the password. This implies that the list of password hashes has already been compromised; which is no longer in the scope of password cracking but more along the lines of social engineering or some other security exploit being put into place.

The comparison cannot be done if the cryptographic hash function is not known or if the incorrect hash function is used. Since different cryptographic hash functions output different digests, an incorrect choice in hash functions guarantees that the attack will fail. This implies that the attacker needs to have prepared different methods of password cracking suited for multiple hash functions.

All the aforementioned attacks must be made offline since modern web applications or databases use an account manager that will lock the user if a certain number of failed attempts are made at the login screen. However implementing an attack offline once again implies that a list of password hashes have been gained, further adding complication to the question of how they were compromised in the first place.

Lastly, there are some other disadvantages that preimage attacks suffer from that are not related to the listed constraints. Dictionary Attacks are not guaranteed to work because of their dependency on the input file. If the file does not contain enough candidate passwords, then the attack will fail. Also, if a Salt is used when hashing the password and the Salt is not known then the Dictionary Attack is impossible since each entry will yield an incorrect result. If the Salt is known then the Dictionary Attack may still work but the dependency on the word-list is still relevant.

Rainbow Tables can be configured to match almost any cryptographic hash function, and the size of the input file to generate them can be chosen to generate any number of password candidate hashes.[1][16] However, if a Salt is used when hashing the password the Rainbow Table Attack becomes extremely slow[28] since the Salt makes the final hash value less likely to show up in a hash chain. A new chain will have to have been made that contains the salt value during construction of the Rainbow Table. If the hash chain does not contain the hash with the salted value then the attack will fail.

The Brute-Force Attack is the only password cracking technique that is guaranteed to work, however the constraint given by it's exponential asymptotic running time[11] means that it is slow to obtain the password. Given a sufficiently large password hashed with an equally large Salt value can make the Brute-Force Attack infeasible simply because of the time it can take.

In summary, while there are weaknesses and strengths for both cryptographic hash functions and password cracking techniques alike, it would seem that preimage attacks are much harder to effectively use rather than implement.

5 Glossary

1. **Alphabet**: The set of characters and the estimated length used for the original password.
2. **Avalanche Effect**: An effect where changing any piece of an original message can change the final output hash value.
3. **Big-Endian**: Big-Endian is where the most significant byte of a number in computer memory is stored in the leftmost byte position.
4. **Compression Function/One-way Compression Function**: A function that transforms two fixed-length input strings into one fixed-length output string.
5. **Cryptographic Hash Function**: An algorithm that maps any input string of arbitrary length into an output bit-string of a fixed length. It has three unique properties that define it to be cryptographically secure.
6. **CSPRNG**: A cryptographic pseudo random number generator is an algorithm for generating a cryptographically secure sequence of numbers that approximates the properties of random numbers.
7. **Digest, hash, or hash value**: The output string from a cryptographic hash function.
8. **Feasibility**: The computational complexity of the action being performed.
9. **Length Padding**: Length Padding (also called a padding function) or pre-processing loop, is where the input message is padded with the 0-bit until the input is a multiple of a fixed number of bits.
10. **Little-Endian**: Little-Endian is where the least significant byte of a number in computer memory is stored in the leftmost byte position.

11. **Merkle-Damgard Construction**: A method of creating cryptographic hash functions using length padding and a compression function.

12. **Message or input**: The input string that is chosen to be encoded.

13. **Password**: A string of characters or a word that is used for authentication to prove identity or authorization to gain access to a resource that is required to be kept secret.

14. **Password Cracking**: A process or technique used to compromise or reveal passwords.

15. **Password Manager/Password Management System**: Various system calls and software instructions in operating systems that manage secure password storage.

16. **Plaintext**: Unaltered data that has need to be kept secret or concealed. In the context of password security, usually plaintext represents a password or a password candidate.

17. **Salt**: A random set of bytes of fixed length that is used as additional input for a cryptographic hash function.

18. **Seed**: An initial value in a CSPRNG chosen by combining some of the values in memory of keyboard typings, mouse movement, and disc activities.

19. **State**: The current values used along with seed in determining the next output of a CSPRNG.

6 References

1. "Password": wikipedia.org \wiki\password

2. "Password Memorability and Security": Yan J, Blackwell A, Anderson R, Grant A (2004) IEEE Security and Privacy Magazine Vol. 2

3. "Password Manager": wikipedia.org\wiki\Password_manager

4. "Plaintext": wikipedia.org\wiki\Plaintext

5. "Hash Functions": Dan Berstein (2003) cse.yorku.ca

6. "Cryptographic Hash Functions": wikipedia.org\wiki \Cryptographic_hash _functions

7. "Collision Resistance": wikipedia.org \wiki \Collision_resistance

8. "Password Cracking": wikipedia.org\wiki\Password_cracking

9. "Cryptography\Brute Force Attack": wikibooks.org \wiki\Cryptography\Brute_Force_Attack

10. "The Birthday Problem": Eric Weisstein: Wolfram Mathworld (2014)

11. "Cain and Abel: Brute-Force Password Cracker": Montero, Massimiliano (2009) Oxid.it

12. "A Simple Brute Force Search Algorithm Implementation in Java": Phil (2009) www.bigwhoop.co.za

13. "Dictionary Attack": wikipedia.org\wiki\Dictionary_attack

14. "Fast Dictionary Attack on Passwords using Time-Space Trade-off": Arvind Narayanan and Vitaly Shmatikov, University of Texas at Austin (2005)

15. "Rainbow Tables": wikipedia.org\wiki\Rainbow_table

16. "Rainbow Tables and Reduction Functions": stitchintime.wordpress.com: Paul Faulstich

17. "SHA-1/SHA-256 Standard": FIP-180 Publication, United States Government, csrc.nist.gov

18. "PolarSSL MD5 API Documentation": (2002) polarssl.org\api\md5_8h.html
19. "SHA-1 Pseudocode": wikipedia.org\wiki\SHA-1
20. "SHA-2 Pseudocode": wikipedia.org\wiki\SHA-2
21. "MD5 Pseudocode": wikipedia.org\wiki\MD5
22. "Merkle-Damgard Construction": wikipedia.org\wiki\Merkle-Damgard construction
23. "One-way Compression Function": wikipedia.org\wiki\One-way_compression_function
24. "CSPRNG": wikipedia.org\wiki\CSPRNG
25. "Salt (Cryptography)": wikipedia.org\wiki\Salt_(Cryptography)
26. "Preimage Attack": wikipedia.org\wiki\Preimage_attack
27. "How Rainbow Tables Work": kestas.kuliukas.com\RainbowTables: K Kuliukas (2006)
28. "How does password salt help against Rainbow Table Attack": stackoverflow.com\questions\420843