# Lab 3: Hashing

**Objective:** The key objective of this lab is to understand the range of hashing methods used, analyse the strength of each of the methods, and in the usage of salting. Overall the most popular hashing methods are: MD5 (128-bit); SHA-1 (160-bit); SHA-256 (256-bit); SHA-3 (256-bit), bcrypt (192-bit) and PBKDF2 (256-bit). The methods of bcrypt, scrypt and PBKDF2 use a number of rounds, and which significantly reduce the hashing rate. This makes the hashing processes much slower, and thus makes the cracking of hashed passwords more difficult. We will also investigate the key hash cracking tools such as **hashcat** and **John The Ripper**.

🕮 **Web link:** https://github.com/billbuchanan/esecurity/tree/master/unit03\_hashing

Open up your **Ubuntu instance** within vsoc.napier.ac.uk and conduct this lab.

Demo: <https://youtu.be/rnTLr6iUbf0>

If required, you can check the hashing methods here: <https://asecuritysite.com/encryption/js10>

## A Hashing

In this section we will look at some fundamental hashing methods.

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **A.1** | Using (either on your Windows desktop or on Ubuntu):  🕮 **Web link (Hashing):**  [~~http://asecuritysite.com/encryption/md5~~](http://asecuritysite.com/encryption/md5)  https://asecuritysite.com/hash/md5  Match the hash signatures with their words (“Falkirk”, “Edinburgh”, “Glasgow” and “Stirling”).  03CF54D8CE19777B12732B8C50B3B66F  D586293D554981ED611AB7B01316D2D5  48E935332AADEC763F2C82CDB4601A25  EE19033300A54DF2FA41DB9881B4B723 | 03CF5: Is it [~~Falkirk~~][Edinburgh][~~Glasgow][Stirling]?~~  D5862: Is it [~~Falkirk][Edinburgh~~][Glasgow][~~Stirling~~]?  48E93: Is it [Falkirk][~~Edinburgh][Glasgow][Stirling]?~~  EE190: Is it [~~Falkirk][Edinburgh][Glasgow][~~Stirling]? |
| **A.2** | Repeat Part 1, but now use openssl, such as:  echo -n 'Falkirk' | openssl md5 | 03CF5: Is it [~~Falkirk~~][Edinburgh][~~Glasgow][Stirling~~]?  D5862: Is it ~~[Falkirk][Edinburgh][~~Glasgow][~~Stirling]~~?  48E93: Is it [Falkirk][~~Edinburgh][Glasgow][Stirling]?~~  EE190: Is it [~~Falkirk][Edinburgh][Glasgow~~][Stirling]? |
| **A.3** | Using:  🕮 **Web link (Hashing):**  <http://asecuritysite.com/encryption/md5>  Determine the number of hex characters for the hash signatures defined. Note: perhaps copy and paste your hash to an on-line character counter? | MD5 hex chars: 32 hex chars  128 bits  SHA-1 hex chars: 40 hex chars  160 bits  SHA-256 hex chars: 64 or 256 bits  SHA-384 hex chars: 96 or 384 bits  SHA-512 hex chars: 128 or 512 bits  How does the number of hex characters relate to the length of the hash signature: |
| **A.4** | For the following /etc/shadow file, determine the matching password:  bill:$apr1$waZS/8Tm$jDZmiZBct/c2hysERcZ3m1  mike:$apr1$mKfrJquI$Kx0CL9krmqhCu0SHKqp5Q0  fred:$apr1$Jbe/hCIb$/k3A4kjpJyC06BUUaPRKs0  ian:$apr1$0GyPhsLi$jTTzW0HNS4Cl5ZEoyFLjB. jane: $1$rqOIRBBN$R2pOQH9egTTVN1Nlst2U7.  [Hint: openssl passwd -apr1 -salt *ZaZS/8TF* *napier*]  openssl passwd -apr1 -salt waZS/8Tm napier  openssl passwd -apr1 -salt mKfrJquI Ankle123  openssl passwd -apr1 -salt Jbe/hCIb inkwell  openssl passwd -apr1 -salt 0GyPhsLi password  openssl passwd -1 -salt rqOIRBBN napier | The passwords are **password**, **napier**, **inkwell** and **Ankle123**.  Bill’s password: napier  Mike’s password: Ankle123  Fred’s password: inkwell  Ian’s password: password  Jane’s password: napier |
| **A.5** | From Ubuntu, download the following:  🕮 **Web link (Files):**  <http://asecuritysite.com/files02.zip>  (a quick way to download is wget asecuritysite.com/files02.zip) and the files should have the following MD5 signatures:  MD5(1.txt)= 5d41402abc4b2a76b9719d911017c592  MD5(2.txt)= 69faab6268350295550de7d587bc323d  MD5(3.txt)= fea0f1f6fede90bd0a925b4194deac11  MD5(4.txt)= d89b56f81cd7b82856231e662429bcf2 | Which file(s) have been modified?  2.txt  openssl dgst -md5 2.txt  e3fc91b12a36c2334ebb5b66caa2d75b |
| **A.6** | From Ubuntu, download the following ZIP file:  🕮 **Web link (PS Files):**  <http://asecuritysite.com/letters.zip>  (a quick way to download is wget asecuritysite.com/letters.zip)  On your Ubuntu instance, you should be able to view the files by double clicking on them in the file explorer (as you should have a PostScript viewer installed).  cat letter\_of\_rec.ps | openssl md5 | Do the files have different contents?  Yes  openssl dgst -md5 letter\_of\_rec.ps  a25f7f0b29ee0b3968c860738533a4b9  openssl dgst -md5 order.ps  a25f7f0b29ee0b3968c860738533a4b9  openssl dgst -sha256 order.ps  077046dd66015e05c3e03a43a6e4de129038e0701de5a4103fc7ed91c3782d06  openssl dgst -sha256 letter\_of\_rec.ps  SHA2-256 de4e4c6e2b94e95a3c5bd72a9a6af29bc5f83bf759325d9921943a6fc08ea245  Now determine the MD5 signature for them. What can you observe from the result?  MD5 Collision |

## B Hash Cracking (Hashcat)

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **B.1** | Run the hashcat benchmark (eg hashcat –b -m 0), and complete the following: | Hash rate for MD5: 22475.5 MH/s  Hash rate for SHA-1: 7063.9 MH/s  Hash rate for SHA-256: 2961.8 MH/s  Hash rate for APR1: ? |
| **B.2** | On Ubuntu, next create a word file (**words**) with the words of “napier”, “password” “Ankle123” and “inkwell”  Using hashcat crack the following MD5 signatures (hash1):  232DD5D7274E0D662F36C575A3BD634C  5F4DCC3B5AA765D61D8327DEB882CF99  6D5875265D1979BDAD1C8A8F383C5FF5  04013F78ACCFEC9B673005FC6F20698D  Command used: hashcat –m 0 hash1 words | 232DD...634C  Is it [napier~~][password][Ankle123][inkwell]?~~  5F4DC...CF99 Is it ~~[napier]~~[password]~~[Ankle123][inkwell]?~~  6D587...5FF5 Is it ~~[napier][password]~~[Ankle123]~~[inkwell]?~~  04013...698D Is it ~~[napier][password][Ankle123]~~[inkwell]? |
| **B.3** | Using the method used in the first part of this tutorial, find the following for names of fruits (the fruits are all in lowercase):  FE01D67A002DFA0F3AC084298142ECCD  1F3870BE274F6C49B3E31A0C6728957F  72B302BF297A228A75730123EFEF7C41  8893DC16B1B2534BAB7B03727145A2BB  889560D93572D538078CE1578567B91A | FE01D: orange  1F387: apple  72B30: banana  8893D: pear  88956: peach |
| **B.4** | Put this SHA-256 value in a file named  file.txt:  106a5842fc5fce6f663176285ed1516dbb 1e3d15c05abab12fdca46d60b539b7  By adding a word of “help” in a word file of words.txt, prove that the following cracks the hash (where file.txt contains the hashed value):  hashcat -m 1400 file.txt words.txt | 106a5842fc5fce6f663176285ed1516dbb1e3d15c05abab12fdca46d60b539b7:help  Session..........: hashcat  Status...........: Cracked  Hash.Mode........: 1400 (SHA2-256)  Hash.Target......: 106a5842fc5fce6f663176285ed1516dbb1e3d15c05abab12fd...b539b7  Time.Started.....: Sat Feb 11 16:05:47 2023 (0 secs)  Time.Estimated...: Sat Feb 11 16:05:47 2023 (0 secs) |
| **B.5** | The following is an NTLM hash, for “help”:  0333c27eb4b9401d91fef02a9f74840e  Prove that the following can crack the hash (where file.txt contains the hashed value):  hashcat -m 1000 file.txt words.txt | 0333c27eb4b9401d91fef02a9f74840e:help  Session..........: hashcat  Status...........: Cracked  Hash.Mode........: 1000 (NTLM)  Hash.Target......: 0333c27eb4b9401d91fef02a9f74840e  Time.Started.....: Sat Feb 11 16:09:35 2023 (0 secs)  Time.Estimated...: Sat Feb 11 16:09:35 2023 (0 secs) |

The cracked hashed are stored in:

~/.hashcat/hashcat.potfile

What do you observe when you use the command: It’s a list of the hashes and their cracks separated by a colon

cat ~/.hashcat/hashcat.potfile

Note, hashcat doesn’t show previously cracked values, so if you want it to recrack them, just use:

rm ~/.hashcat/hashcat.potfile

**B.6** Now crack the following Scottish football teams (all are single words):

635450503029fc2484f1d7eb80da8e25bdc1770e1dd14710c592c8929ba37ee9

BEF68628460A29657F55A2860407969E3AF183E889021B30091C815F6C6B248D

bc5fb9abe8d5e72eb49cf00b3dbd173cbf914835281fadd674d5a2b680e47d50

6ac16a68ac94ca8298c9c2329593a4a4130b6fed2472a98424b7b4019ef1d968

hashcat -m 1400 teams.txt .\rockyou.txt

Football teams:

635450503029fc2484f1d7eb80da8e25bdc1770e1dd14710c592c8929ba37ee9:celtic

bc5fb9abe8d5e72eb49cf00b3dbd173cbf914835281fadd674d5a2b680e47d50:aberdeen

bef68628460a29657f55a2860407969e3af183e889021b30091c815f6c6b248d:motherwell

6ac16a68ac94ca8298c9c2329593a4a4130b6fed2472a98424b7b4019ef1d968:livingston

**B.7** Rather than use a dictionary, we can use a brute force a hashed password using a lowercase character set:

hashcat -a 3 -m 1400 file.txt ?l?l?l?l?l?l?l?l --increment

Using this style of command (look at the hash type and perhaps this is a SHA-256 hash), crack the following words:

4dc2159bba05da394c3b94c6f54354db1f1f43b321ac4bbdfc2f658237858c70

0282d9b79f42c74c1550b20ff2dd16aafc3fe5d8ae9a00b2f66996d0ae882775

47c215b5f70eb9c9b4bcb2c027007d6cf38a899f40d1d1da6922e49308b15b69

Words:

Number of tests for each sequence tried:

a->z: 26

Guess.Mask.......: ?l [1]

Progress.........: 26/26 (100.00%)

aa->zz:

Guess.Mask.......: ?l?l [2]

Progress.........: 676/676 (100.00%)

aaa->zzz:

Guess.Mask.......: ?l?l?l [3]

Progress.........: 17576/17576 (100.00%)

aaaa->zzzz:

Guess.Mask.......: ?l?l?l?l [4]

Progress.........: 456976/456976 (100.00%)

What happens when you take the “--increment” flag away?

My machine over heats

Hashcat just reports the results.

**B.8** We can focus on given letters, such as where we add a letter or a digit at the end:

hashcat -a 3 -m 1000 file.txt password?l

hashcat -a 3 -m 1000 file.txt password?u

hashcat -a 3 -m 1000 file.txt password?d

Using these commands, crack the following:

7a6c8de8ad7f89b922cc29c9505f58c3

db0edd04aaac4506f7edab03ac855d56

hashcat -a 3 -m 1000 file.txt password?l

Progress.........: 26/26 (100.00%)

hashcat -a 3 -m 0 file.txt password?l

Progress.........: 26/26 (100.00%)

hashcat -a 3 -m 1000 file.txt password?u

Progress.........: 26/26 (100.00%)

hashcat -a 3 -m 0 file.txt password?u

Progress.........: 26/26 (100.00%)

hashcat -a 3 -m 1000 file.txt password?d

Progress.........: 10/10 (100.00%)

hashcat -a 3 -m 0 file.txt password?d

Progress.........: 10/10 (100.00%)

Note: Remember to try both MD5 (0) and NTLM hash (1000).

Words:

Number of tests for each:

## C Hashing Cracking (John The Ripper)

All of the passwords in this section are in lowercase.

wget https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt

john --wordlist=rockyou.txt pwdump

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **C.1** | On Ubuntu, and using John the Ripper, and using a word list with the names of fruits, crack the following pwdump passwords:  fred:500:E79E56A8E5C6F8FEAAD3B435B51404EE:5EBE7DFA074DA8EE8AEF1FAA2BBDE876:::  bert:501:10EAF413723CBB15AAD3B435B51404EE:CA8E025E9893E8CE3D2CBF847FC56814::: | Fred:APPLE  Bert:ORANGE |
| **C.2** | On Ubuntu, and using John the Ripper, the following pwdump passwords (they are names of major Scottish cities/towns):  Admin:500:629E2BA1C0338CE0AAD3B435B51404EE:9408CB400B20ABA3DFEC054D2B6EE5A1:::  fred:501:33E58ABB4D723E5EE72C57EF50F76A05:4DFC4E7AA65D71FD4E06D061871C05F2:::  bert:502:BC2B6A869601E4D9AAD3B435B51404EE:2D8947D98F0B09A88DC9FCD6E546A711::: | Admin: DUNDEE  Fred: ABERDEE  Bert: PERTH |
| **C.3** | On Ubuntu, and using John the Ripper, crack the following pwdump passwords (they are the names of animals):  fred:500:5A8BB08EFF0D416AAAD3B435B51404EE:85A2ED1CA59D0479B1E3406972AB1928:::  bert:501:C6E4266FEBEBD6A8AAD3B435B51404EE:0B9957E8BED733E0350C703AC1CDA822:::  admin:502:333CB006680FAF0A417EAF50CFAC29C3:D2EDBC29463C40E76297119421D2A707::: | Fred: SNAKE  Bert:  TIGER  Admin:  ELEPHAN |

Note:

Use rm -r ~/.john/ to remove the previously cracked hashes.

You can use john --wordlist=fruits pwdump to crack with a wordlist and pwdump.

## D LM Hash

The LM Hash is used in Microsoft Windows. For example, for LM Hash:

hashme gives: FA-91-C4-FD-28-A2-D2-57-**AA-D3-B4-35-B5-14-04-EE**

network gives: D7-5A-34-5D-5D-20-7A-00-**AA-D3-B4-35-B5-14-04-EE**

napier gives: 12-B9-C5-4F-6F-E0-EC-80-**AA-D3-B4-35-B5-14-04-EE**

Notice that the right-most element of the hash are always the same, if the password is less than eight characters. With more than eight characters we get:

networksims gives: D7-5A-34-5D-5D-20-7A-00-38-32-A0-DB-BA-51-68-07

napier123 gives: 67-82-2A-34-ED-C7-48-92-B7-5E-0C-8D-76-95-4A-50

For “hello” we get:

LM: FD-A9-5F-BE-CA-28-8D-44-AA-D3-B4-35-B5-14-04-EE

NTLM: 06-6D-DF-D4-EF-0E-9C-D7-C2-56-FE-77-19-1E-F4-3C

We can check these with a Python script:

import passlib.hash;

string="hello"

print ("LM Hash:"+passlib.hash.lmhash.hash(string))

print ("NT Hash:"+passlib.hash.nthash.hash(string))

which gives:

LM Hash:fda95fbeca288d44aad3b435b51404ee

NT Hash:066ddfd4ef0e9cd7c256fe77191ef43c

🕮 **Web link (LM Hash):** http://asecuritysite.com/encryption/lmhash

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **D.1** | Create a Python script to determine the LM hash and NTLM hash of the following words: | “Napier”  LM Hash:12b9c54f6fe0ec80aad3b435b51404ee  NT Hash:3ca6cef4b84985b6e3cd7b24843ea7d1  “Foxtrot”  LM Hash:82121098b60f69f5aad3b435b51404ee  NT Hash:828f0524d3fffd8632ee97253183fef3 |

## E APR1

The Apache-defined APR1 format addresses the problems of brute forcing an MD5 hash, and basically iterates over the hash value 1,000 times. This considerably slows an intruder as they try to crack the hashed value. The resulting hashed string contains “$apr1$” to identify it and uses a 32-bit salt value. We can use both htpassword and Openssl to compute the hashed string (where “bill” is the user and “hello” is the password):

# htpasswd -nbm bill hello

bill:$apr1$PkWj6gM4$XGWpADBVPyypjL/cL0XMc1

# openssl passwd -apr1 -salt PkWj6gM4 hello

$apr1$PkWj6gM4$XGWpADBVPyypjL/cL0XMc1

We can also create a simple Python program with the passlib library, and add the same salt as the example above:

import passlib.hash;

salt="PkWj6gM4"

string="hello"

print ("APR1:"+passlib.hash.apr\_md5\_crypt.hash(string, salt=salt))

We can created a simple Python program with the passlib library, and add the same salt as the example above:

APR1:$apr1$PkWj6gM4$XGWpADBVPyypjL/cL0XMc1

Refer to: http://asecuritysite.com/encryption/apr1

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **E.1** | Create a Python script to create the APR1 hash for the following:  [just list first four characters of the hash] | changeme:APR1:$apr1$PkWj6gM4$V2w1yci/N1HCLzcqo3jiZ/  123456:APR1:$apr1$PkWj6gM4$opHu7xKPBmSPWdVO8vidC/  password:APR1:$apr1$PkWj6gM4$OupRScHgsxe5lQj4.azPy. |

## F SHA

While APR1 has a salted value, the SHA-1 hash does not have a salted value. It produces a 160-bit signature, thus can contain a larger set of hashed value than MD5, but because there is no salt it can be cracked to rainbow tables, and also brute force. The format for the storage of the hashed password on Linux systems is:

# htpasswd -nbs bill hello

bill:{SHA}qvTGHdzF6KLavt4PO0gs2a6pQ00=

We can also generate salted passwords with crypt, and can use the Python script of:

import passlib.hash;

salt="8sFt66rZ"

string="hello"

print ("SHA1:"+passlib.hash.sha1\_crypt.hash(string, salt=salt))

print ("SHA256:"+passlib.hash.sha256\_crypt.hash(string, salt=salt))

print ("SHA512:"+passlib.hash.sha512\_crypt.hash(string, salt=salt))

SHA-512 salts start with $6$ and are up to 16 chars long.

SHA-256 salts start with $5$ and are up to 16 chars long

Which produces:

SHA1:$sha1$480000$8sFt66rZ$klAZf7IPWRN1ACGNZIMxxuVaIKRj

SHA256:$5$rounds=535000$8sFt66rZ$.YYuHL27JtcOX8WpjwKf2VM876kLTGZHsHwCBbq9xTD

SHA512:$6$rounds=656000$8sFt66rZ$aMTKQHl60VXFjiDAsyNFxn4gRezZOZarxHaK.TcpVYLpMw6MnX0lyPQU06SSVmSdmF/VNbvPkkMpOEONvSd5Q1

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **F.1** | Create a Python script to create the SHA Crypt hash for the following:  [just list first four characters of the hash] | changeme:SHA1:$sha1$480000$8sFt66rZ$dNfLzeD4O48TgFqDKd0zBYc4SJ5a  changeme:SHA256:$5$rounds=535000$8sFt66rZ$yNCVBp7NMi3UNzMEIoGoGnQZ.HMGaUETwiQNCBi/cl5  changeme:SHA512:$6$rounds=656000$8sFt66rZ$B/.Msj2UuS3qH.Qxsy.RL82oni6MV75LZ8olN6eCw6.LSHCCcJ4IGnzdX9Qv299whMbpz4rR9e7A9Ab0L3ZA0/  123456:SHA1:$sha1$480000$8sFt66rZ$RndE8VtL.VnDBVLPgp7vKcVb0BaN  123456:SHA256:$5$rounds=535000$8sFt66rZ$rAkO4NCQq4l0DjDAJFh2f6s9Ew.Y7qCIM7okpuHJR30  123456:SHA512:$6$rounds=656000$8sFt66rZ$cGaBfax5eeGRKwD.bFK0IFUvrk0jyyeWKIkIsmWX0H9xvJco6OwFPZ5QA4jh5mZnt1w9FlhO2pKlhkpPHICts0  password:SHA1:$sha1$480000$8sFt66rZ$h0Q07GoRgcYjKiYsjpufby/P7cf0  password:SHA256:$5$rounds=535000$8sFt66rZ$63AbYmdfWxNIp9x75xK4zBgdQxvzMGtNzpyI6DKhvb7  password:SHA512:$6$rounds=656000$8sFt66rZ$hiU32dGMdhmg9uxrHHBnVz1j5A35Ap193q.Naf9kUkP.e1klwhfPW3Lv/LIydTiS97Adp5zYGN.1RM8s6NcNy/ |

## G PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) is defined in RFC 2898 and generates a salted hash. Often this is used to create an encryption key from a defined password, and where it is not possible to reverse the password from the hashed value. It is used in TrueCrypt to generate the key required to read the header information of the encrypted drive, and which stores the encryption keys.

PBKDF2 is used in WPA-2 and TrueCrypt. Its main focus is to produced a hashed version of a password and includes a salt value to reduce the opportunity for a rainbow table attack. It generally uses over 1,000 iterations in order to slow down the creation of the hash, so that it can overcome brute force attacks. The generalise format for PBKDF2 is:

DK = PBKDF2(Password, Salt, MInterations, dkLen)

where Password is the pass phrase, Salt is the salt, MInterations is the number of iterations, and dklen is the length of the derived hash.

In WPA-2, the IEEE 802.11i standard defines that the pre-shared key is defined by:

PSK = PBKDF2(PassPhrase, ssid, ssidLength, 4096, 256)

In TrueCrypt we use PBKDF2 to generate the key (with salt) and which will decrypt the header, and reveal the keys which have been used to encrypt the disk (using AES, 3DES or Twofish). We use:

byte[] result = passwordDerive.GenerateDerivedKey(16,

ASCIIEncoding.UTF8.GetBytes(message), salt, 1000);

which has a key length of 16 bytes (128 bits - dklen), uses a salt byte array, and 1000 iterations of the hash (Minterations). The resulting hash value will have 32 hexadecimal characters (16 bytes).

🕮 **Web link (PBKDF2):** <http://www.asecuritysite.com/encryption/PBKDF2>

import passlib.hash;

import sys;

salt="ZDzPE45C"

string="password"

if (len(sys.argv)>1):

string=sys.argv[1]

if (len(sys.argv)>2):

salt=sys.argv[2]

print ("PBKDF2 (SHA1):",passlib.hash.pbkdf2\_sha1.hash(string, salt=salt.encode()))

print ("PBKDF2 (SHA256):",passlib.hash.pbkdf2\_sha256.hash(string,salt=salt.encode()))

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **G.1** | Create a Python script to create the PBKDF2 hash for the following (uses a salt value of “ZDzPE45C”). You just need to list the first six hex characters of the hashed value. | changeme: PBKDF2 (SHA1): $pbkdf2$131000$WkR6UEU0NUM$qS7S53GV52Ha3Qq1SUna.XlrS1U  changeme: PBKDF2 (SHA256): $pbkdf2-sha256$29000$WkR6UEU0NUM$gWsN0JM2s94YGo0W9On0Mz6yFvRMCFRE1Ms4dXIpCE4  123456: PBKDF2 (SHA1): $pbkdf2$131000$WkR6UEU0NUM$Ax363Np0kPa.8vfjSkepDqEMFYg  123456: PBKDF2 (SHA256): $pbkdf2-sha256$29000$WkR6UEU0NUM$GHyI8vXC/POt8bfLR35D.9uLvLtPCoBiUDa1O00Ef28  password: PBKDF2 (SHA1): $pbkdf2$131000$WkR6UEU0NUM$.L1L.AVXTBSsc0FuHRQz4PNMVXc  password: PBKDF2 (SHA256): $pbkdf2-sha256$29000$WkR6UEU0NUM$pd1VbFkOA/VwbhJZhJ.25kHPsKVXika2XsuKYoudcug |

## H Bcrypt

MD5 and SHA-1 produce a hash signature, but this can be attacked by rainbow tables. Bcrypt (Blowfish Crypt) is a more powerful hash generator for passwords and uses salt to create a non-recurrent hash. It was designed by Niels Provos and David Mazières, and is based on the Blowfish cipher. It is used as the default password hashing method for BSD and other systems.

Overall it uses a 128-bit salt value, which requires 22 Base-64 characters. It can use a number of iterations, which will slow down any brute-force cracking of the hashed value. For example, “Hello” with a salt value of “$2a$06$NkYh0RCM8pNWPaYvRLgN9.” gives:

$2a$**06**$**NkYh0RCM8pNWPaYvRLgN9.**LbJw4gcnWCOQYIom0P08UEZRQQjbfpy

As illustrated in Figure 1, the first part is "$2a$" (or "$2b$"), and then followed by the number of rounds used. In this case is it **6 rounds** which is 2^6 iterations (where each additional round doubles the hash time). The 128-bit (22 character) salt values comes after this, and then finally there is a 184-bit hash code (which is 31 characters).

The slowness of bcrypt is highlighted with an AWS EC2 server benchmark using hashcat:

* Hash type: MD5 Speed/sec: 380.02M words
* Hash type: SHA1 Speed/sec: 218.86M words
* Hash type: SHA256 Speed/sec: 110.37M words
* Hash type: bcrypt, Blowfish(OpenBSD) Speed/sec: 25.86k words
* Hash type: NTLM. Speed/sec: 370.22M words

You can see that Bcrypt is almost 15,000 times slower than MD5 (380,000,000 words/sec down to only 25,860 words/sec). With John The Ripper:

* md5crypt [MD5 32/64 X2] 318237 c/s real, 8881 c/s virtual
* bcrypt ("$2a$05", 32 iterations) 25488 c/s real, 708 c/s virtual
* LM [DES 128/128 SSE2-16] 88090K c/s real, 2462K c/s virtual

where you can see that BCrypt over 3,000 times slower than LM hashes. So, although the main hashing methods are fast and efficient, this speed has a down side, in that they can be cracked easier. With Bcrypt the speed of cracking is considerably slowed down, with each iteration doubling the amount of time it takes to crack the hash with brute force. If we add one onto the number of rounds, we double the time taken for the hashing process. So, to go from 6 to 16 increase by over 1,000 (210) and from 6 to 26 increases by over 1 million (220).

The following defines a Python script which calculates a whole range of hashes:

# https://asecuritysite.com/encryption/hash

import sys

from hashlib import md5

import passlib.hash;

import bcrypt

import hashlib;

num = 30

repeat\_n=1

salt="ZDzPE45C"

string="the boy stood on the burning deck"

salt2="1111111111111111111111"

print ("Word: ",string)print ("Salt: ",salt)

print("\nHashes")

print("SHA-1\t",hashlib.sha1(string.encode()).hexdigest())

print("SHA-256\t",hashlib.sha256(string.encode()).hexdigest())

print("SHA-512\t",hashlib.sha512(string.encode()).hexdigest())

print("MD-5:\t\t\t", md5(string.encode()).hexdigest())

print("DES:\t\t\t", passlib.hash.des\_crypt.hash(string.encode(), salt=salt[:2]))

print("Bcrypt:\t\t\t", bcrypt.kdf(string.encode(),salt=salt.encode(),desired\_key\_bytes=32,rounds=100 ).hex())

print("APR1:\t\t\t", passlib.hash.apr\_md5\_crypt.hash(string.encode(), salt=salt))

**print("PBKDF2 (SHA1):\t\t", passlib.hash.pbkdf2\_sha1.hash(string.encode(),rounds=5, salt=salt.encode()))**

**print("PBKDF2 (SHA-256):\t", passlib.hash.pbkdf2\_sha256.hash(string,rounds=5, salt=salt.encode()))**

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **H.1** | Create the hash for the word “hello” for the different methods (you only have to give the first six hex characters for the hash):  Also note the number hex characters that the hashed value uses: | MD5:  SHA1:  SHA256:  SHA512:  DES:  MD5:  Sun MD5:  SHA-1:  SHA-256:  SHA-512: |

Hashes

40 characters

SHA-1 aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

64 characters

SHA-256 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

128 characters

SHA-512 9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890caadae2dff72519673ca72323c3d99ba5c11d7c7acc6e14b8c5da0c4663475c2e5c3adef46f73bcdec043

32 characters

MD-5:

5d41402abc4b2a76b9719d911017c592

13 characters

DES:

ZDVX7N5Bz.8wk

64 characters

Bcrypt:

67e6b6a1446720cbcaa8cd0b30c6fae01aeb2361fff9bdacf033671fae050c92

22 characters not including the delineation or salt

APR1:

$apr1$ZDzPE45C$qn6wBl2beuA9CPlp4KFn9/

27 characters

PBKDF2 (SHA1):

$pbkdf2$5$WkR6UEU0NUM$HEZFFxE.Nw4kv9S9OAklO.I7Zdc

43 characters

PBKDF2 (SHA-256):

$pbkdf2-sha256$5$WkR6UEU0NUM$46kLMg7OEJgy5l3c5kyzJFySOpRG6ybxo//9pK0o48w

22 characters

md5:

$1$ZDzPE45C$dOTT0LUnoqs6J7mNLdyse0

32 characters

sun\_md5:

$md5,rounds=34000$ZDzPE45C$$fdZ8uoSiWj6RcJOoMiaKX1

28 characters

sha-1:

$sha1$1000$ZDzPE45C$PcIUj3WibdACcNHcpDtsor3k/y4c

43 characters

sha-256: $5$rounds=1000$ZDzPE45C$Srkz7w56pq4WC78AAeMdjlqoatpFkRTkAynTl0.8Tu0

86 characters

sha-512: $6$rounds=1000$ZDzPE45C$MTlLYYkZK/Dz/19ybYmAQg6LTOVwdWEqmPJf1WpSP3uubZllmxdrMrwB7osCXtnKXDifN8eBnZgiylcdBbdUm1



**Figure 1** Bcrypt

## I HMAC

Write a Python program which will prove the following:

Data: Hello

Hex: 48656c6c6f

Key: qwerty123

Hex: 717765727479313233

HMAC-MD5: c3a2fa8f20dee654a32c30e666cec48e w6L6jyDe5lSjLDDmZs7Ejg==

If you get this to work, can you expand to include other MAC methods (including HMAC-SHA1, HMAC-256, and so on). A starting point for your program is here:

## <https://asecuritysite.com/hash/hashnew2_hmacmd5>

## Using this online tool, check that the HMAC values are correct:

https://cryptii.com/pipes/hmac

## J Reflective statements

**1. Why might increasing the number of iterations be a better method of protecting a hashed password than using a salted version?**

Increasing the number of iterations will slow down the cracker programs, i.e. hashcat and john the ripper. A salt is always stored with the hash so an attacker knows what the hash is and so could compute rainbow tables for known salts

**2. Why might the methods bcrypt, Phpass and PBFDK2 be preferred for storing passwords than MD5, SHA?**

Because they are purposefully very slow hashing algorithms to make cracking more time consuming, to prevent an attacker parallelizing the hash cracking algorithms and to potentially cause buffer overflows in the cracking machine which would cause the cracking program to stop/fail.

## K What I should have learnt from this lab?

The key things learnt:

* The differing methods used to hash data.
* How hashcat and John The Ripper are used to crack hashed values.
* How salt is added to the hashing process.
* The core difference between the fast hashing methods (such as MD5 and SHA-1) and the slow ones (bcrypt and PBKDF2).

## L Additional

The following provides a hash most of the widely used hashing method. For this enter the code of:

# https://asecuritysite.com/encryption/hash

import sys

from hashlib import md5

import passlib.hash;

import bcrypt

import hashlib;

num = 30

repeat\_n=1

salt="ZDzPE45C"

string="the boy stood on the burning deck"

salt2="1111111111111111111111"

print ("Word: ",string)

print ("Salt: ",salt)

print("\nHashes")

print("SHA-1\t",hashlib.sha1(string.encode()).hexdigest())

print("SHA-256\t",hashlib.sha256(string.encode()).hexdigest())

print("SHA-512\t",hashlib.sha512(string.encode()).hexdigest())

print("MD-5:\t\t\t", md5(string.encode()).hexdigest())

print ("UNIX hashes (with salt)")

print("DES:\t\t\t", passlib.hash.des\_crypt.hash(string.encode(), salt=salt[:2]))

print("Bcrypt:\t\t\t", bcrypt.kdf(string.encode(),salt=salt.encode(),desired\_key\_bytes=32,rounds=100 ).hex())

print("APR1:\t\t\t", passlib.hash.apr\_md5\_crypt.hash(string.encode(), salt=salt))

print("PBKDF2 (SHA1):\t\t", passlib.hash.pbkdf2\_sha1.hash(string.encode(),rounds=5, salt=salt.encode()))

print("PBKDF2 (SHA-256):\t", passlib.hash.pbkdf2\_sha256.hash(string,rounds=5, salt=salt.encode()))

print ("MS Windows Hashes")

print("LM Hash:\t\t", passlib.hash.lmhash.hash(string.encode()))

print("NT Hash:\t\t", passlib.hash.nthash.hash(string.encode()))

print("MS DCC:\t\t\t", passlib.hash.msdcc.hash(string.encode(), salt))

print ("LDAP Hashes")

print ("LDAP (MD5):"+passlib.hash.ldap\_md5.hash(string))

print ("LDAP (MD5 Salted):"+passlib.hash.ldap\_salted\_md5.hash(string, salt=salt.encode()))

print ("LDAP (SHA):"+passlib.hash.ldap\_sha1.hash(string))

print ("LDAP (SHA1 Salted):"+passlib.hash.ldap\_salted\_sha1.hash(string, salt=salt.encode()))

print ("LDAP (DES Crypt):"+passlib.hash.ldap\_des\_crypt.hash(string))

print ("LDAP (BSDI Crypt):"+passlib.hash.ldap\_bsdi\_crypt.hash(string))

print ("LDAP (MD5 Crypt):"+passlib.hash.ldap\_md5\_crypt.hash(string))

print ("LDAP (Bcrypt):"+passlib.hash.ldap\_bcrypt.hash(string))

print ("LDAP (SHA1):"+passlib.hash.ldap\_sha1\_crypt.hash(string))

print ("LDAP (SHA256):"+passlib.hash.ldap\_sha256\_crypt.hash(string))

print ("LDAP (SHA512):"+passlib.hash.ldap\_sha512\_crypt.hash(string))

print ("LDAP (Hex MD5):"+passlib.hash.ldap\_hex\_md5.hash(string))

print ("LDAP (Hex SHA1):",passlib.hash.ldap\_hex\_sha1.hash(string))

print ("LDAP (At Lass):",passlib.hash.atlassian\_pbkdf2\_sha1.encrypt(string))

print ("LDAP (FSHP):"+passlib.hash.fshp.hash(string))

print ("Database Hashes")

print("MS SQL 2000:\t\t", passlib.hash.mssql2000.hash(string.encode()))

print("MySQL:\t\t\t", passlib.hash.mysql41.hash(string.encode()))

print("Oracle 10:\t\t", passlib.hash.oracle10.hash(string.encode(), user=salt))

print("Postgres (MD5):\t\t", passlib.hash.postgres\_md5.hash(string.encode(), user=salt))

print ("Other Known Hashes")

print("Cisco PIX:\t\t", passlib.hash.cisco\_pix.hash(string[:16].encode(), user=salt))

print("Cisco Type 7:\t\t", passlib.hash.cisco\_type7.hash(string.encode()))

print ("Dyango DES:"+passlib.hash.django\_des\_crypt.hash(string, salt=salt))

print ("Dyango MD5:"+passlib.hash.django\_salted\_md5.hash(string, salt=salt[:2]))

print ("Dyango SHA1:"+passlib.hash.django\_salted\_sha1.hash(string, salt=salt))

print ("Dyango Bcrypt:"+passlib.hash.django\_bcrypt.hash(string, salt=salt2[:22]))

print ("Dyango PBKDF2 SHA1:"+passlib.hash.django\_pbkdf2\_sha1.hash(string, salt=salt))

print ("Dyango PBKDF2 SHA1:"+passlib.hash.django\_pbkdf2\_sha256.hash(string, salt=salt))

print ("Bcrypt:"+passlib.hash.bcrypt.hash(string, salt=salt2[:22]))

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **L.1** | In the code, what does the modifier of “[:22]” do?  In running the methods, which of them take the longest time to compute?  Of the methods used, outline how you would identify some of the methods. For APR1 has an identifier of $apr1$. | It limits the string to 22 characters  SHA256 Crypt is noticeably slower than everything else |

For the following identify the hash methods used:

* 5f4dcc3b5aa765d61d8327deb882cf99
* 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
* $apr1$ZDzPE45C$y372GZYCbB1WYtOkbm4/u. **APR1**
* $P$HZDzPE45Ch4tvOeT9mhtu3i2G/JybR1
* b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7785e5976ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86 **SHA512**
* $1$ZDzPE45C$EEQHJaCXI6yInV3FnskmF1
* $2a$12$111111111111111111111uAQxS9vJNRtBb6zeFDV6k7tyB0DZJF0a

**L.2** It is known that a user has used a password of “passXord”, where X is an unknown character or number. Can crack the following hashes based on a filter:

5fa8051ada600a097bd0922d7a085b94734684c4e070b24a02cf43d24d6eedbe

a6f63a5fb10b3bba180a79f2fc565b1db2101040ce71ea80692d671857fe2117

Passwords used:

5fa8051ada600a097bd0922d7a085b94734684c4e070b24a02cf43d24d6eedbe

Token length exception hashcat doesn’t see this one as a hash value

a6f63a5fb10b3bba180a79f2fc565b1db2101040ce71ea80692d671857fe2117

PassWord

Number of tests:

?

**L.3** Download the bfield.hash password hash, and using the rockyou.txt list, determine the first 10 passwords in the hashed file. An example command might be:

hashcat -m 0 bfield.hash /usr/share/wordlists/rockyou.txt

hashcat -m 0 bfield.hash rockyou.txt

First 10 passwords from bfield.hash:

f01e0d7992a3b7748538d02291b0beae:samantha

1aef0a62ed84bb165989ab32f0ba56c2:barbie

91cb315a6405bfcc30e2c4571ccfb8ce:chelsea

8dae58e3f282b974328d53f96753f4c1:lovers

f21c0d3e564c7db5ccf73c095a0b9371:jasmine

fc275ac3498d6ab0f0b4389f8e94422c:brandon

f379eaf3c831b04de153469d1bec345e:666666

3bf1114a986ba87ed28fc1b5884fc2f8:shadow

ff5390bde5a4cf0aa2006cf2198efd29:melissa

26b637ed41273425be243e8d42e5b461:eminem

Sample bfield: <http://www.adeptus-mechanicus.com/codex/hashpass/bfield.hash.7z>

Sample rockyou.txt: https://www.scrapmaker.com/data/wordlists/dictionaries/rockyou.txt