# 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):<br><br>🕮 **Web link (Hashing):** `http://asecuritysite.com/encryption/md5`<br><br>Match the hash signatures with their words ("Falkirk", "Edinburgh", "Glasgow" and "Stirling").<br><br>`03CF54D8CE19777B12732B8C50B3B66F`<br>`D586293D554981ED611AB7B01316D2D5`<br>`48E935332AADEC763F2C82CDB4601A25`<br>`EE19033300A54DF2FA41DB9881B4B723` | `03CF5:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`D5862:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`48E93:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`EE190:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]? |
| **A.2** | Repeat Part 1, but now use openssl, such as:<br><br>echo -n 'Falkirk' \| openssl md5 | `03CF5:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`D5862:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`48E93:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]?<br><br>`EE190:` Is it [Falkirk][Edinburgh][Glasgow][Stirling]? |

| | | |
|---|---|---|
| **A.3** | Using:<br><br>🕮 **Web link (Hashing):**<br>`http://asecuritysite.com/encryption/md5`<br><br>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:<br><br>SHA-1 hex chars:<br><br>SHA-256 hex chars:<br><br>SHA-384 hex chars:<br><br>SHA-512 hex chars:<br><br>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:<br><br>`bill:$apr1$waZS/8Tm$jDZmiZBct/c2hysERcZ3m1`<br>`mike:$apr1$mKfrJquI$Kx0CL9krmqhCu0SHKqp5Q0`<br>`fred:$apr1$Jbe/hCIb$/k3A4kjpJyC06BUUaPRKs0`<br>`ian:$apr1$0GyPhsLi$jTTzW0HNS4Cl5ZEoyFLjB.`<br>`jane: $1$rqOIRBBN$R2pOQH9egTTVN1Nlst2U7.`<br><br>[Hint: openssl passwd -apr1 -salt *ZaZS/8TF napier*] | The passwords are **password**, **napier**, **inkwell** and **Ankle123**.<br><br>Bill's password:<br><br>Mike's password:<br><br>Fred's password:<br><br>Ian's password:<br><br>Jane's password: |
| **A.5** | From Ubuntu, download the following:<br><br>🕮 **Web link (Files):**<br>`http://asecuritysite.com/files02.zip`<br><br>(a quick way to download is `wget asecuritysite.com/files02.zip`) and the files should have the following MD5 signatures:<br><br>`MD5(1.txt)= 5d41402abc4b2a76b9719d911017c592`<br>`MD5(2.txt)= 69faab6268350295550de7d587bc323d`<br>`MD5(3.txt)= fea0f1f6fede90bd0a925b4194deac11`<br>`MD5(4.txt)= d89b56f81cd7b82856231e662429bcf2` | Which file(s) have been modified? |
| **A.6** | From Ubuntu, download the following ZIP file:<br><br>🕮 **Web link (PS Files):**<br>`http://asecuritysite.com/letters.zip`<br>(a quick way to download is `wget asecuritysite.com/letters.zip`)<br>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).<br><br>`cat letter_of_rec.ps \| openssl md5` | Do the files have different contents?<br><br><br>Now determine the MD5 signature for them. What can you observe from the result? |

## 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:<br>Hash rate for SHA-1:<br>Hash rate for SHA-256:<br>Hash rate for APR1: |
| **B.2** | On Ubuntu, next create a word file (**words**) with the words of "napier", "password" "Ankle123" and "inkwell"<br><br>Using hashcat crack the following MD5 signatures (hash1):<br><br>`232DD5D7274E0D662F36C575A3BD634C`<br>`5F4DCC3B5AA765D61D8327DEB882CF99`<br>`6D5875265D1979BDAD1C8A8F383C5FF5`<br>`04013F78ACCFEC9B673005FC6F20698D`<br><br>Command used: `hashcat -m 0 hash1 words` | `232DD...634C`<br>Is it [napier][password][Ankle123][inkwell]?<br><br>`5F4DC...CF99` Is it [napier][password][Ankle123][inkwell]?<br><br>`6D587...5FF5` Is it [napier][password][Ankle123][inkwell]?<br><br>`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):<br><br>`FE01D67A002DFA0F3AC084298142ECCD`<br>`1F3870BE274F6C49B3E31A0C6728957F`<br>`72B302BF297A228A75730123EFEF7C41`<br>`8893DC16B1B2534BAB7B03727145A2BB`<br>`889560D93572D538078CE1578567B91A` | `FE01D:` orange<br><br>`1F387:` apple<br><br>`72B30:` banana<br><br>`8893D:` pear<br><br>`88956:` peach |
| **B.4** | Put this SHA-256 value in a file named file.txt:<br><br>`106a5842fc5fce6f663176285ed1516dbb`<br>`1e3d15c05abab12fdca46d60b539b7`<br><br>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):<br><br>`hashcat -m 1400 file.txt words.txt` | |
| **B.5** | The following is an NTLM hash, for "help":<br><br>`0333c27eb4b9401d91fef02a9f74840e`<br><br>Prove that the following can crack the hash (where file.txt contains the hashed value):<br><br>`hashcat -m 1000 file.txt words.txt` | |

The cracked hashed are stored in:

```
~/.hashcat/hashcat.potfile
```

What do you observe when you use the command:

```
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
```

Football teams:

bef68 motherwell
6ac livingston
635 celtic
bc5f aberdeen

**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:
aa->zz:
aaa->zzz:
aaaa->zzzz:

What happens when you take the "--increment" flag away?

it can;t crack the hashes because it only tries passwords exactly 8 chars long

**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
```

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

| | |
|---|---|
| Words:<br><br>Number of tests for each: | db0 password5 10 tests<br>7a6 passwordW 26 tests |

## C Hashing Cracking (John The Ripper)

All of the passwords in this section are in lowercase.

| 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:<br><br>`fred:500:E79E56A8E5C6F8FEAAD3B435B51404EE:5EBE7DFA074DA8EE8AEF1FAA2BBDE876:::`<br>`bert:501:10EAF413723CBB15AAD3B435B51404EE:CA8E025E9893E8CE3D2CBF847FC56814:::` | Fred: apple<br><br>Bert: orange |
| **C.2** | On Ubuntu, and using John the Ripper, the following pwdump passwords (they are names of major Scottish cities/towns):<br><br>`Admin:500:629E2BA1C0338CE0AAD3B435B51404EE:9408CB400B20ABA3DFEC054D2B6EE5A1:::`<br>`fred:501:33E58ABB4D723E5EE72C57EF50F76A05:4DFC4E7AA65D71FD4E06D061871C05F2:::`<br>`bert:502:BC2B6A869601E4D9AAD3B435B51404EE:2D8947D98F0B09A88DC9FCD6E546A711:::` | Admin:<br><br>Fred:<br><br>Bert: |
| **C.3** | On Ubuntu, and using John the Ripper, crack the following pwdump passwords (they are the names of animals):<br><br>`fred:500:5A8BB08EFF0D416AAAD3B435B51404EE:85A2ED1CA59D0479B1E3406972AB1928:::`<br>`bert:501:C6E4266FEBEBD6A8AAD3B435B51404EE:0B9957E8BED733E0350C703AC1CDA822:::`<br>`admin:502:333CB006680FAF0A417EAF50CFAC29C3:D2EDBC29463C40E76297119421D2A707:::` | Fred: Aberdeen<br><br>Bert: Perth<br><br>Admin:<br>Dundee |

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.encrypt(string)
print "NT Hash:"+passlib.hash.nthash.encrypt(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: | LM Hash: 12b9c54f6fe0ec80aad3b435b51404ee<br>"Napier"<br>NT Hash: 3ca6cef4b84985b6e3cd7b24843ea7d<br><br>"Foxtrot"<br>LM Hash: 82121098b60f69f5aad3b435b51404ee<br>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.encrypt(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:<br><br>[just list first four characters of the hash] | "changeme":<br>$apr1$PkWj6gM4$V2w1yci/N1HCLzcqo3jiZ/<br>"123456":<br>$apr1$PkWj6gM4$opHu7xKPBmSPWdVO8vidC/<br><br>"password"<br>$apr1$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.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(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$8sFt66rZ$.YYuHL27JtcOX8WpjwKf2VM876kLTGZHsHwCBbq9x
TD
SHA512:$6$rounds=656000$8sFt66rZ$aMTKQHl60VXFjiDAsyNFxn4gRezZOZarxHaK.TcpV
YLpMw6MnX0lyPQU06SSVmSdmF/VNbvPkkMpOEONvSd5Q1
```

| No | Description | Result |
|---|---|---|



Text

| | | |
|---|---|---|
| **F.1** | Create a Python script to create the SHA Crypt hash for the following:<br><br>[just list first four characters of the hash] | "changeme":<br>$sha1$480000$RVVHquMy$IdG87Nk3Wl9WIhaNwR6rzcO9dcaP<br>"123456":<br>$sha1$480000$sOzwm1U4$dM9qXs/kgn01g7RFgwQ52G9IMTWe<br>"password"<br>$sha1$480000$qyGhiA9y$CUavI837kFjhfEu9FYrExtZNinMC |

## 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 hashlib;
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.encrypt(string, salt=salt)
print "PBKDF2 (SHA256):"+passlib.hash.pbkdf2_sha256.encrypt(string, salt=salt)
```

| 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":<br><br>"123456":<br><br>"password" |

# 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 ($2^{10}$) and from 6 to 26 increases by over 1 million ($2^{20}$).

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

```
import hashlib;
import passlib.hash;

salt="ZDzPE45C"
string="password"
salt2="1111111111111111111111"

print "General Hashes"
print "MD5:"+hashlib.md5(string).hexdigest()
print "SHA1:"+hashlib.sha1(string).hexdigest()
print "SHA256:"+hashlib.sha256(string).hexdigest()
print "SHA512:"+hashlib.sha512(string).hexdigest()

print "UNIX hashes (with salt)"
print "DES:"+passlib.hash.des_crypt.encrypt(string, salt=salt[:2])
print "MD5:"+passlib.hash.md5_crypt.encrypt(string, salt=salt)
print "Sun MD5:"+passlib.hash.sun_md5_crypt.encrypt(string, salt=salt)
print "SHA1:"+passlib.hash.sha1_crypt.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(string, salt=salt)
print "Bcrypt:"+passlib.hash.bcrypt.encrypt(string, salt=salt2[:22])
```

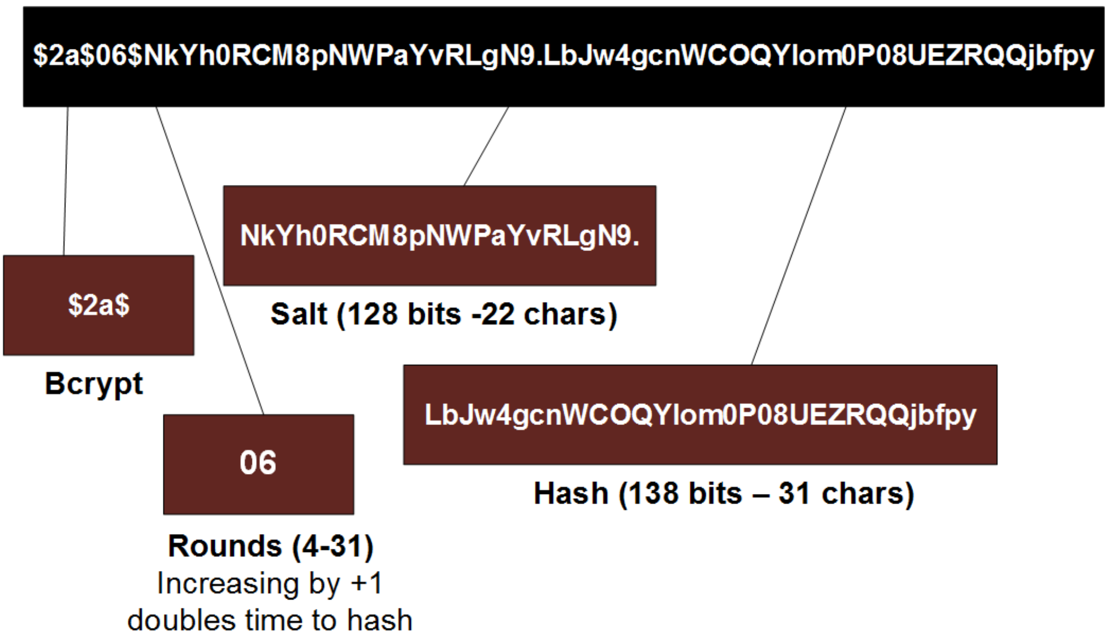


**Figure 1** Bcrypt

| 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): | MD5:<br>SHA1:<br>SHA256:<br>SHA512: |

| | | |
|---|---|---|
| | Also note the number hex characters that the hashed value uses: | DES:<br>MD5:<br>Sun MD5:<br>SHA-1:<br>SHA-256:<br>SHA-512: |

## I HMAC

Write a Python or Node.js program which will prove the following:

```
Type:       HMAC-MD5
Message:    Hello
Password:   qwerty123
Hex:        c3a2fa8f20dee654a32c30e666cec48e
Base64:     7376b67daf1fdb475e7bae786b7d9cdf47baeba71e738f1e
```

If you get this to work, can you expand to include other MAC methods. You can test against this page:

https://asecuritysite.com/encryption/js10

## 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?** |

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

## 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:

```
import hashlib;
import passlib.hash;
import sys;


salt="ZDzPE45C"
string="password"
salt2="1111111111111111111111"


if (len(sys.argv)>1):
	string=sys.argv[1]

if (len(sys.argv)>2):
	salt=sys.argv[2]

print "General Hashes"
print "MD5:"+hashlib.md5(string).hexdigest()
print "SHA1:"+hashlib.sha1(string).hexdigest()
print "SHA256:"+hashlib.sha256(string).hexdigest()
print "SHA512:"+hashlib.sha512(string).hexdigest()

print "UNIX hashes (with salt)"
print "DES:"+passlib.hash.des_crypt.encrypt(string, salt=salt[:2])
print "MD5:"+passlib.hash.md5_crypt.encrypt(string, salt=salt)
print "Sun MD5:"+passlib.hash.sun_md5_crypt.encrypt(string, salt=salt)
print "SHA1:"+passlib.hash.sha1_crypt.encrypt(string, salt=salt)
print "SHA256:"+passlib.hash.sha256_crypt.encrypt(string, salt=salt)
print "SHA512:"+passlib.hash.sha512_crypt.encrypt(string, salt=salt)

print "APR1:"+passlib.hash.apr_md5_crypt.encrypt(string, salt=salt)
print "PHPASS:"+passlib.hash.phpass.encrypt(string, salt=salt)
print "PBKDF2 (SHA1):"+passlib.hash.pbkdf2_sha1.encrypt(string, salt=salt)
print "PBKDF2 (SHA256):"+passlib.hash.pbkdf2_sha256.encrypt(string, salt=salt)
#print "PBKDF2 (SHA512):"+passlib.hash.pbkdf2_sha512.encrypt(string, salt=salt)
#print "CTA PBKDF2:"+passlib.hash.cta_pbkdf2_sha1.encrypt(string, salt=salt)
#print "DLITZ PBKDF2:"+passlib.hash.dlitz_pbkdf2_sha1.encrypt(string, salt=salt)

print "MS Windows Hashes"
print "LM Hash:"+passlib.hash.lmhash.encrypt(string)
print "NT Hash:"+passlib.hash.nthash.encrypt(string)
print "MS DCC:"+passlib.hash.msdcc.encrypt(string, salt)
print "MS DCC2:"+passlib.hash.msdcc2.encrypt(string, salt)

#print "LDAP Hashes"
#print "LDAP (MD5):"+passlib.hash.ldap_md5.encrypt(string)
#print "LDAP (MD5 Salted):"+passlib.hash.ldap_salted_md5.encrypt(string, salt=salt)
#print "LDAP (SHA):"+passlib.hash.ldap_sha1.encrypt(string)
#print "LDAP (SHA1 Salted):"+passlib.hash.ldap_salted_sha1.encrypt(string,
salt=salt)
#print "LDAP (DES Crypt):"+passlib.hash.ldap_des_crypt.encrypt(string)
#print "LDAP (BSDI Crypt):"+passlib.hash.ldap_bsdi_crypt.encrypt(string)
#print "LDAP (MD5 Crypt):"+passlib.hash.ldap_md5_crypt.encrypt(string)
#print "LDAP (Bcrypt):"+passlib.hash.ldap_bcrypt.encrypt(string)
#print "LDAP (SHA1):"+passlib.hash.ldap_sha1_crypt.encrypt(string)
#print "LDAP (SHA256):"+passlib.hash.ldap_sha256_crypt.encrypt(string)
#print "LDAP (SHA512):"+passlib.hash.ldap_sha512_crypt.encrypt(string)

print "LDAP (Hex MD5):"+passlib.hash.ldap_hex_md5.encrypt(string)
print "LDAP (Hex SHA1):"+passlib.hash.ldap_hex_sha1.encrypt(string)
print "LDAP (At Lass):"+passlib.hash.atlassian_pbkdf2_sha1.encrypt(string)
print "LDAP (FSHP):"+passlib.hash.fshp.encrypt(string)

print "Database Hashes"
print "MS SQL 2000:"+passlib.hash.mssql2000.encrypt(string)
print "MS SQL 2000:"+passlib.hash.mssql2005.encrypt(string)
print "MS SQL 2000:"+passlib.hash.mysql323.encrypt(string)
print "MySQL:"+passlib.hash.mysql41.encrypt(string)
print "Postgres (MD5):"+passlib.hash.postgres_md5.encrypt(string, user=salt)
print "Oracle 10:"+passlib.hash.oracle10.encrypt(string, user=salt)
```

```
print "Oracle 11:"+passlib.hash.oracle11.encrypt(string)

print "Other Known Hashes"
print "Cisco PIX:"+passlib.hash.cisco_pix.encrypt(string, user=salt)
print "Cisco Type 7:"+passlib.hash.cisco_type7.encrypt(string)
print "Dyango DES:"+passlib.hash.django_des_crypt.encrypt(string, salt=salt)
print "Dyango MD5:"+passlib.hash.django_salted_md5.encrypt(string, salt=salt[:2])
print "Dyango SHA1:"+passlib.hash.django_salted_sha1.encrypt(string, salt=salt)
print "Dyango Bcrypt:"+passlib.hash.django_bcrypt.encrypt(string, salt=salt2[:22])
print "Dyango PBKDF2 SHA1:"+passlib.hash.django_pbkdf2_sha1.encrypt(string,
salt=salt)
print "Dyango PBKDF2 SHA1:"+passlib.hash.django_pbkdf2_sha256.encrypt(string,
salt=salt)
print "Bcrypt:"+passlib.hash.bcrypt.encrypt(string, salt=salt2[:22])
```

| No | Description | Result |
|---|---|---|
| **L.1** | In the code, what does the modifier of "`[:22]`" do?<br><br>In running the methods, which of them take the longest time to compute?<br><br>Of the methods used, outline how you would identify some of the methods. For APR1 has an identifier of $apr1$. | |

For the following identify the hash methods used:

- `5f4dcc3b5aa765d61d8327deb882cf99`
- `5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8`
- `$apr1$ZDzPE45C$y372GZYCbB1WYtOkbm4/u.`
- `$P$HZDzPE45Ch4tvOeT9mhtu3i2G/JybR1`
- `b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7785e597 6ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86`
- `$1$ZDzPE45C$EEQHJaCXI6yInV3FnskmF1`
- `$2a$12$1111111111111111111111uAQxS9vJNRtBb6zeFDV6k7tyB0DZJF0a`

**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:

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
```

| First 10 passwords from bfield.hash: |
|---|
| |

Sample bfield: http://www.adeptus-mechanicus.com/codex/hashpass/bfield.hash.7z
Sample rockyou.txt: https://www.scrapmaker.com/data/wordlists/dictionaries/rockyou.txt