

# Lab 1 : Hashing Implementation

---

This lab is adopted from the website: <http://asecuritysite.com>

In this lab we will explore implementations of hashing methods.

## A: LM Hash

---

LM Hash is used in many versions of Windows Pre NT to store user passwords that are fewer than 15 characters long. It is a weak security implementation can be easily broken using standard dictionary lookups.

Modern versions of Windows NT to current Windows 10 or Server 2016 use SYSKEY to encrypt passwords. The LM Hash is used in Microsoft Windows. For example, for LM Hash for the following words wil produce the following 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 is 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 simple Python script:

```
import passlib.hash;
string="hello"
print "LM Hash:"+passlib.hash.lmhash.encrypt(string)
print "NT Hash:"+passlib.hash.nthash.encrypt(string)
```

which should give:

LM Hash:	fda95fbeca288d44aad3b435b51404ee
NT Hash:	066ddfd4ef0e9cd7c256fe77191ef43c

**Web link:** <http://asecuritysite.com/encryption/lmhash>

No	Description	Result
1	Create a Python script to determine the LM hash and NTLM hash of the following words: Hashes for "Napier": LM Hash: 12b9c54f6fe0ec80aad3b435b51404ee NT Hash: 3ca6cef4b84985b6e3cd7b24843ea7d1 Hashes for "Foxtrot": LM Hash: 82121098b60f69f5aad3b435b51404ee NT Hash: 828f0524d3fffd8632ee97253183fef3	"Napier" "Foxtrot"  for source code see "A_LM_Hash.py"

## B: 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 htpasswd 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 create 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
----	-------------	--------

1	<p>Create a Python script to create the APR1 hash for the following:</p> <p>Prove them against on-line APR1 generator (or from the page given above).</p>	<p>“changeme”:</p> <p>Hashes for “changeme”: MD5 Hash: \$apr1\$Pkwj6gM4\$V2w1yci/N1HCLzcqo3jiZ/</p> <p>“123456”:</p> <p>Hashes for “123456”: MD5 Hash: \$apr1\$Pkwj6gM4\$OpHu7xKPBmSPWdV08vidC/</p> <p>“password”</p> <p>Hashes for “password”: MD5 Hash: \$apr1\$Pkwj6gM4\$0upRSchgsxe5lQj4.azPy.</p> <p>for source code see “B APR Hash.py”</p>
---	---	---

## C: SHA

While APR1 has a salted value, the SHA has for storing passwords does not have a salted value. It produces a 160-bit signature, thus can contain a larger set of hashed value, 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:

```
# httpasswd -nbs bill hello bill:{SHA}qVTGHdzF6KLavt4P00gs2a6pQ00=
```

We can also generate salted passwords, 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$8sFt66rZ$k1AZf7IPWRN1ACGNZIMxxuVaIKRj
SHA256:$5$rounds=535000$8sFt66rZ$.YYuHL27JtcOX8wpjwKf2VM876kLTGZHsHwCBbq9x
TD
SHA512:$6$rounds=656000$8sFt66rZ$aMTKQH160VXFjiDAsyNFxn4gRezZOZarxHaK.TcpV
YLpMw6MnX01yPQU06SSVmSdmF/VNbvPkkmpOEONvSd5Q1
```

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

1	Create a Python script to create the SHA hash for the following:	“changeme”:
	Prove them against on-line SHA generator (or from the page given above).	“123456”:  “password”

Hashes for "changeme":  
SHA1 Hash: \$sha1\$480000\$8sFt66rZ\$dNfLzeD4048TgFqDKd0zBYc4SJ5a

for source code see "C\_SHA\_Hash.py"

Hashes for "123456":  
SHA1 Hash: \$sha1\$480000\$8sFt66rZ\$RNdE8VtL.VnDBVLPgp7vKcVb0BaN

## D: PHPass

Hashes for "password":  
SHA1 Hash: \$sha1\$480000\$8sFt66rZ\$h0Q07GoRgcYjKiYsjpuFby/P7cf0

phpass is used as a hashing method by WordPress and Drupal. It is public domain software and used with PHP applications. The three main methods used are:

- CRYPT\_BLOWFISH. This is the most secure method is known as, and related to the bcrypt method. Another method (CRYPT\_EXT\_DES) uses the DES encryption method.
- CRYPT\_EXT\_DES. This method uses DES encryption.
- MD5. This is the least preferred method and simply uses an MD5 digest.

The output uses the following to identify the differing types:

- \$P\$. Standard.
- \$H\$. Phpbb.
- \$\$\$. Drupal. SHA-512-like digests.

A sample run with “password” and salt of “ZDzPE45C” for seven rounds gives:

```
$P$5ZDzPE45Ci.QxPaPz.03z6TYbakcSQ0
```

Where it can be seen that the salt value is placed after "\$P\$5" ("ZDzPE45C"), and after that there are 22 Base-64 characters (giving 128-bit hash signature).

We can check the output against the following Python code:

```
import passlib.hash; string = "password" salt="ZDzPE45C"
print passlib.hash.phpass.encrypt(string,
salt=salt, rounds=7)
```

which should give: \$P\$5ZDzPE45Ci.QxPaPz.03z6TYbakcSQ0

The code uses 7 rounds - to save processor time - but it can vary between 7 and 30. The number of iterations is  $2^{\text{rounds}}$ . In this case the hash is "i.QxPaPz.03z6TYbakcSQ0" which is a 128-bit hash signature.

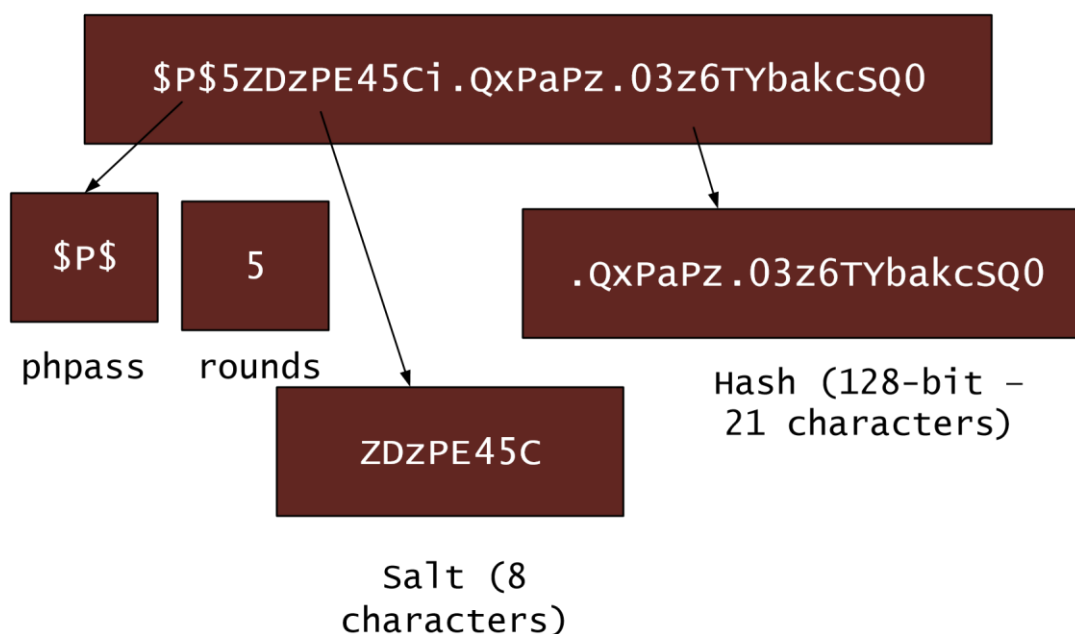



Figure 3.1 php pass hashing

 **Web link (phpass):** <http://asecuritysite.com/encryption/phpass>

No	Description	Result
1	<p>Create a Python script to create the PHPass hash for the following:</p> <p>Prove them against on-line PHPass generator (or from the page given above).</p> <p>Just note the first five characters of the hashed value.</p>	<p>“changeme”:</p> <pre>Hashes for "changeme": Salt Used: "ZDzPE45C" Hash \$P\$5ZDzPE45CgJFN2hHbmNiFh7yVqcGEA1</pre> <p>“123456”:</p> <pre>Hashes for "123456": Salt Used: "ZDzPE45C" Hash: \$P\$5ZDzPE45C5ATryn003x9x1fdCsPEk51</pre> <p>“password”</p> <pre>Hashes for "password": Salt Used: "ZDzPE45C" Hash: \$P\$5ZDzPE45Ci.QxPaPz.03z6TYbakcSQ0</pre>

for source code see "D\_PHPASS\_Hash.py"

## E: 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 (Figure 1). Its main focus is to produce 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 generalised format for PBKDF2 is:

$$DK = \text{PBKDF2}(\text{Password}, \text{Salt}, \text{MIterations}, \text{dkLen})$$

where Password is the pass phrase, Salt is the salt, MIterations 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:

$$\text{PSK} = \text{PBKDF2}(\text{PassPhrase}, \text{ssid}, \text{ssidLength}, 4096, 256)$$

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 (Miterations). The resulting hash value will have 32 hexadecimal characters (16 bytes).

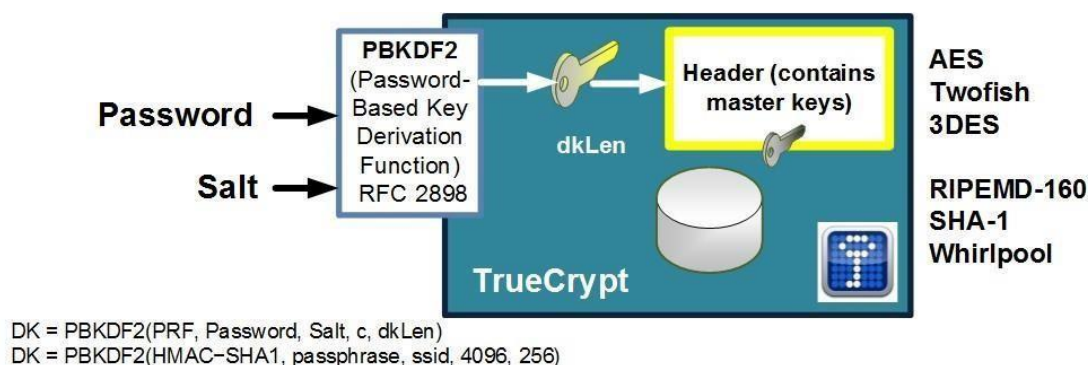


Figure 1 PBKDF2

**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)
```

for source code see "E\_PBKDF2\_Hash.py"

No	Description	Result
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": "123456": "password"

Hashes for "changeme":  
 PBKDF2 (SHA1): \$pbkdf2\$131000\$WkR6UEU0NUM\$qS7S53GV52Ha3Qq1SUna.X1rS1U

Hashes for "123456":  
 PBKDF2 (SHA1): \$pbkdf2\$131000\$WkR6UEU0NUM\$Ax363Np0kPa.8vfjSkepDqEMFYg

Hashes for "password":  
 PBKDF2 (SHA1): \$pbkdf2\$131000\$WkR6UEU0NUM\$.L1L.AVXTBSsc0FuHRQz4PNMVXc

## F: 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 nonrecurrent 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 2, 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 ( $2^{20}$ ).

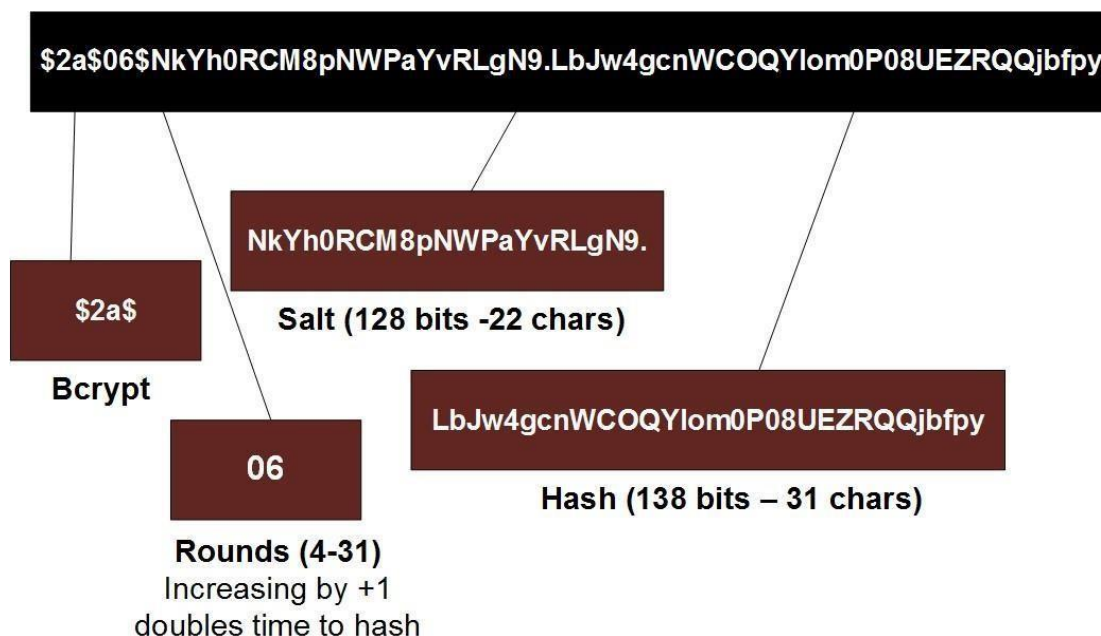


Figure 2 Bcrypt

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

```
import hashlib; import passlib.hash;

salt="ZDzPE45C" string="password" salt2="11111111111111111111"

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

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



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: \$1\$ZDzPE45C\$d0TT0LUnoqs6J7mNLdyse0 SHA1: \$sha1\$480000\$ZDzPE45C\$Lnx\$SENDwEXBWKtQ1fc9... SHA256: \$5\$rounds=535000\$ZDzPE45C\$TTN/Qd.elve1rH... SHA512: \$6\$rounds=656000\$ZDzPE45C\$6VW0iufRn... DES: ZDVX7N5Bz.8wk
	Also note the number hex characters that the hashed value uses:	Hex length for "hello": MD5: 34 Sun MD5: 50 SHA-1: 50 SHA-256: 69 SHA=512: 112

for source code see "F\_BCRYPT\_Hash.py"

## G: A more complete set of hashes

In this final exercise, we will attempt to 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="11111111111111111111111111111111"
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
"Bcrypt:"+passlib.hash.bcrypt.encrypt(string, salt=salt2[:22]) 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_salt_md5.encrypt(string, salt=salt)
#print "LDAP (SHA):"+passlib.hash.ldap_sha1.encrypt(string)
#print "LDAP (SHA1 Salted):"+passlib.hash.ldap_salt_md5.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_salt_md5.encrypt(string, salt=salt[:2]) print "Dyango
SHA1:"+passlib.hash.django_salt_md5.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)

```

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

1	<p>In the code, what does the modifier of “[:22]” do?</p> <p>In running the methods, which of them take the longest time to compute?</p> <p>Of the methods used, outline how you would identify some of the methods. For APR1 has an identifier of \$apr1\$.</p>	<p>(string, salt=salt2[:22]) will provide the first 22 characters of the variable 'salt2'</p> <p>Slowest on my device was bcrypt</p> <p>Many hash functions have delimiters that state the hash type. For ex: SHA1: \$sha1\$ SHA256: \$5\$ SHA512: \$6\$ PBKDF2 (SHA1): \$pbkdf2\$ PHPass: \$P\$</p> <p>Additionally, tools such as hashid, HashTag, or hash-identifier can be used to determine the algorithm used</p>
---	--	---

For the following identify the hash methods used:

- 5f4dcc3b5aa765d61d8327deb882cf99 MD5
- 5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d □  
8\$apr1\$ZDzPE45C\$y372GZYCbB1WYtOkbm4/u. APR1
- \$P\$HZDzPE45Ch4tvOeT9mhtu3i2G/JybR1 PHPass
- b109f3bbbc244eb82441917ed06d618b9008dd09b3befd1b5e07394c706a8bb980b1d7785e5  
976ec049b46df5f1326af5a2ea6d103fd07c95385ffab0cacbc86 SHA512
- \$1\$ZDzPE45C\$EEQHJaCXI6yInV3FnskMF1 MD5\_Crypt
- \$2a\$12\$11111111111111111111111111111111uAQxS9vJNRtBb6zeFDV6k7tyB0DZJF0a bcrypt

## H: Reflective statements

---

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

Having a password that was hashed multiple times would take longer to crack than a password that was hashed fewer times with a salt

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

BCrypt Phpass and PBFDK2 take longer as they are more secure than md5 and sha.

They are preferred for passwords because of the added security.

Additionally, because passwords aren't very long, the increased processing time is negligible.

**Deliverables:**

1. Answer the above questions in the designated boxes
2. Using the above code write an interactive python program, requesting a user's password and the hashing algorithm they wish to use. The output should be the actual password, the hash name, and the hashed password. Please include at least one extra hashing algorithm discussed in lecture or from your research of the topic see algs.py
3. Using the above code as a guide and pycrypto module write a python program using AES to encrypt and/or decrypt a file.  
For both programs Please add comments to your code explaining what each statement does and include a sample run for each of your hashes and the symmetric encryption see fileAES.py, testdoc.txt, estdoc.txt.dec, testdoc.txt.enc
4. Extra Credit: What question do you think I should have asked?

**Possible references:**

- Brief description of all encryption algorithms <https://asecuritysite.com/encryption>
- Crypto 101 a PDF book attached <https://github.com/crypto101/book>
- Public key cryptography - Diffie-Hellman Key Exchange video as promised <https://www.youtube.com/watch?v=YEBfamv-do>

