

# Lab 5: Diffie-Hellman, Public Key, Private Key and Hashing

Demo: <http://youtu.be/3n2TMpHqE18>

## 1 Diffie-Hellman

No	Description	Result
1	Bob and Alice have agreed on the values:  $g=2,879$ , $N=9,929$ Bob Select $b=6$ , Alice selects $a=9$	Now calculate (using the Kali calculator):  Bob's B value ( $g^b \text{ mod } N$ ):  Alice's A value ( $g^a \text{ mod } N$ ):
2	Now they exchange the values. Next calculate the shared key:	Bob's value ( $A^b \text{ mod } N$ ):  Alice's value ( $B^a \text{ mod } N$ ):  Do they match? [ <u>Yes</u> ] [No]
3	If you are in the lab, select someone to share a value with. Next agree on two numbers ( $g$ and $N$ ).  You should pick a random number, and so should they. Do not tell them what your random number is. Next calculate your public value, and get them to do the same.  Next exchange values.	Numbers for $g$ and $N$ :  Your private value:  Your public value:  The public value you received:  Shared key:  Do they match: [ <u>Yes</u> ] [No]

## 2 Symmetric Key

No	Description	Result
1	Log into vSoC 2, and select your Kali host on the DMZ or public network.	What is your IP address?
2	Use:  <code>openssl list -cipher-commands</code>  <code>openssl version</code>	Outline five encryption methods that are supported:  Outline the version of OpenSSL:
3	Using openssl and the command in the form:  <code>openssl prime -hex 1111</code>	Check if the following are prime numbers:  42 [Yes][No] 1421 [Yes][No]
4	Now create a file named myfile.txt (either use Notepad or another editor).  Next encrypt with aes-256-cbc  <code>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin</code>  and enter your password.	Use following command to view the output file:  <code>cat encrypted.bin</code>  Is it easy to write out or transmit the output: [Yes][No]
5	Now repeat the previous command and add the -base64 option.  <code>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64</code>	Use following command to view the output file:  <code>cat encrypted.bin</code>  Is it easy to write out or transmit the output: [Yes][No]

6	<p>Now repeat the previous command and observe the encrypted output.</p> <pre>openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin -base64</pre>	<p>Has the output changed from the run in 4? [Yes][No]</p> <p>Why has it changed?</p>
7	<p>Now let's decrypt the encrypted file with the correct format:</p> <pre>openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:<i>napier</i> -base64</pre>	<p>Has the output been decrypted correctly?</p> <p>What happens when you use the wrong password?</p>
8	<p>If you are working in the lab, now give your secret passphrase to your neighbour, and get them to encrypt a secret message for you.</p> <p>To receive a file, you listen on a given port (such as Port 1234)</p> <pre>nc -l -p 1234 &gt; enc.bin</pre> <p>And then send to a given IP address with:</p> <pre>nc -w 3 [IP] 1234 &lt; enc.bin</pre>	<p>Did you manage to decrypt their message? [Yes][No]</p>
9	<p>With OpenSSL, we can define a fixed salt value that has been used in the cipher process. For example, in Linux:</p> <pre>echo -n "Hello"   openssl enc -aes-128-cbc -pass pass:"london" -e -base64 -S 241fa86763b85341 <b>U1q+o+vs5mvAc3GUiKt8hA==</b></pre> <pre>echo U1q+o+vs5mvAc3GUiKt8hA==   openssl enc -aes-128-cbc -pass pass:"london" -d -base64 -S 241fa86763b85341 <b>Hello</b></pre>	<p>[qwerty] [inkwell] [london] [paris] [cake]</p>

	For a cipher text for <b>256-bit AES CBC</b> and a message of “Hello” with a salt value of “241fa86763b85341”, try the following passwords, and determine the password used for a ciphertext of “U2FsdGVkX18kH6hnY7hTQT8aJwduPmvX91PyppQfvvg=”	
<b>10</b>	Now, use the decryption method to prove that you can decrypt the ciphertext.  echo U2FsdGVkX18kH6hnY7hTQT8aJwduPmvX91PyppQfvvg=   openssl enc -aes-256-cbc -pass pass:"password" -d -base64 -S 241fa86763b85341	Did you confirm the right password? [ <u>Yes</u> /No]
<b>11</b>	Investigate the following commands by running them several times:  echo -n "Hello"   openssl enc -aes-128-cbc -pass pass:"london" -e -base64 -S 241fa86763b85341  echo -n "Hello"   openssl enc -aes-128-cbc -pass pass:"london" -e -base64 -salt	What do you observe?  Why do you think causes this (ask your tutor if you want some detail)?

### 3 Public Key

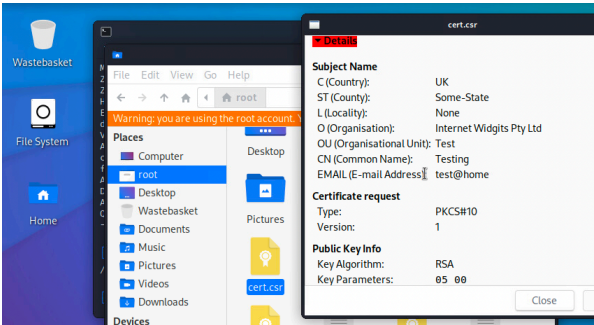
No	Description	Result
<b>1</b>	With RSA, we have a public modulus (and which is $N=p.q$ , and where p and q are prime numbers). To create this, we need to generate a key pair with:  openssl genrsa -out private.pem 1024  This file contains both the public and the private key.	What is the type of public key method used:  How long is the default key:  How long are the prime numbers that are used to generate the public key?
<b>2</b>	Use following command to view the output file:	What can be observed at the start and end of the file:

	<code>cat private.pem</code>	
<b>3</b>	<p>Next we view the RSA key pair:</p> <pre>openssl rsa -in private.pem -text -noout</pre> <p>You should now see the public exponent (e), the private exponent (d), the two prime numbers (p and q), and the public modulus (N).</p>	<p>Which number format is used to display the information on the attributes:</p> <p>Which are the elements of the key shown:</p> <p>Which are the elements of the public key?</p> <p>Which are the elements of the private key?</p> <p>What does the <code>-noout</code> option do?</p>
<b>4</b>	<p>Let's now secure the encrypted key with 3-DES:</p> <pre>openssl rsa -in private.pem -des3 -out key3des.pem</pre> <p>You should NEVER share your private key.</p>	
<b>5</b>	<p>Next, we will export the public key:</p> <pre>openssl rsa -in private.pem -out public.pem -outform PEM -pubout</pre>	<p>View the output public key.</p> <p>What does the header and footer of the file identify?</p> <p>Is the public key smaller in size than the private key? [<u>Yes</u>/No]</p>

6	Now we will encrypt with our public key:  <code>openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin</code>	
7	And then decrypt with our private key:  <code>openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt</code>	What are the contents of decrypted.txt:
8	If you are working in the lab, now give your public key to your neighbour, and get them to encrypt a secret message for you.	Did you manage to decrypt their message? [Yes][No]

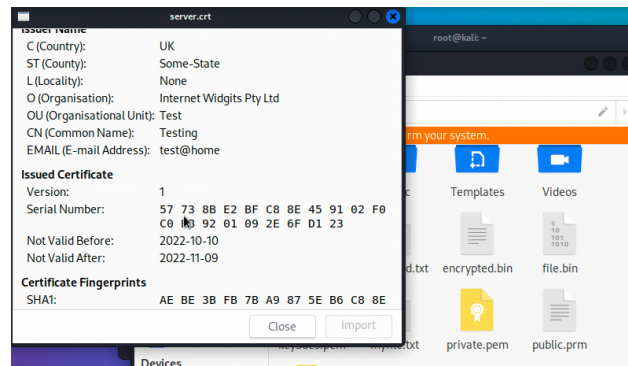
## 4 Storing keys

We have stored our keys on a key ring file (PEM). Normally we would use a digital certificate to distribute our public key. In this part of the tutorial we will create a crt digital certificate file.

No	Description	Result
1	<p>Next create the crt file with the following:</p> <pre>openssl req -new -key private.pem -out cert.csr</pre> 	<p>View the CRT file by double clicking on it from the File Explorer.</p> <p>Using Google to search, what is PKCS#10, and which is it used for?</p> <p>What is the type of public key method used:</p> <p>What is the public key size (in bits): [512][1024][2048]</p>

We can now take the code signing request, and create a certificate. For this we sign the certificate with a private key, in order to validate it:

```
openssl x509 -req -in cert.csr -signkey private.pem -out server.crt
```



From the File System, click on the newly created certificate file (server.crt) and determine:

The size of the public key (in bits):  
[512][1024][2048]

The public key encryption method:

Which is the hashing method that has been signed to sign the certificate: [MD5][SHA-1][SHA-256]

## 5 Hashing

<http://youtu.be/Xvbk2nSzEPk>

The current Hashcat version on Kali has problems with a lack of memory. To overcome this, install Hashcat 6.0.0. On Kali on your public network, first download Hashcat 6.0.0:

```
wget https://hashcat.net/files/hashcat-6.0.0.7z
```

Next unzip it into your home folder:

```
p7zip -d hashcat-6.0.0.7z
```

Then from your home folder, setup a link to Hashcat 6.0.0:

```
ln -s hashcat-6.0.0/hashcat.bin hashcat
```

```
# ./hashcat -version  
v6.0.0
```

No	Description	Result
1	<p>Using:</p> <p><a href="http://asecuritysite.com/encryption/md5">http://asecuritysite.com/encryption/md5</a></p> <p>Match the hash signatures with their words (“Falkirk”, “Edinburgh”, “Glasgow” and “Stirling”).</p> <p>03CF54D8CE19777B12732B8C50B3B66F D586293D554981ED611AB7B01316D2D5 48E935332AADEC763F2C82CDB4601A25 EE19033300A54DF2FA41DB9881B4B723</p>	<p>03CF5: Is it [Falkirk][Edinburgh][Glasgow][Stirling]?</p> <p>D5862: Is it [Falkirk][Edinburgh][Glasgow][Stirling]?</p> <p>48E93: Is it [Falkirk][Edinburgh][Glasgow][Stirling]?</p> <p>EE190: Is it [Falkirk][Edinburgh][Glasgow][Stirling]?</p>
2	<p>Using:</p> <p><a href="http://asecuritysite.com/encryption/md5">http://asecuritysite.com/encryption/md5</a></p> <p>Determine the number of hex characters in the following hash signatures.</p>	<p>MD5 hex chars:</p> <p>SHA-1 hex chars:</p> <p>SHA-256 hex chars:</p> <p>How does the number of hex characters relate to the length of the hash signature:</p>
3	<p>On Kali, for the following /etc/shadow file, determine the matching password:</p>	<p>The passwords are password, napier, inkwell and Ankle123. [Hint: openssl passwd -apr1 -salt ZaZS/8TF napier]</p>



	bill:\$apr1\$waZS/8Tm\$jDZmiZBct/c2hysERCZ3m1 mike:\$apr1\$mKfrJquI\$Kx0CL9krmqhCu0SHKqp5Q0 fred:\$apr1\$Jbe/hCIb\$/k3A4kjpJyC06BUUaPRKs0 ian:\$apr1\$0GyPhsLi\$jTTzw0HNS4Cl5ZEoyFLjB. jane: \$1\$rq0IRBBN\$R2pOQH9egTTVN1Nlst2U7.	Bill's password:  Mike's password:  Fred's password:  Ian's password:  Jane's password:
4	On Kali, download the following:  <a href="http://asecuritysite.com/files02.zip">http://asecuritysite.com/files02.zip</a>  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  Note: You can use <b>md5sum</b> to get the MD5 hash of the files.	Which file(s) have been modified:
5	From Kali, download the following ZIP file:  <a href="http://asecuritysite.com/letters.zip">http://asecuritysite.com/letters.zip</a>	View the letters. Are they different? Now determine the MD5 signature for them. What can you observe from the result?

## 6 Hashing Cracking (MD5)

No	Description	Result
1	<p>On Kali, next create a words file (<b>words</b>) with the words of “napier”, “password” “Ankle123” and “inkwell”</p> <p>Using hashcat crack the following MD5 signatures (hash1): 232DD5D7274E0D662F36C575A3BD634C 5F4DCC3B5AA765D61D8327DEB882CF99 6D5875265D1979BDAD1C8A8F383C5FF5 04013F78ACCFEC9B673005FC6F20698D</p> <p>Command used: <code>hashcat -m 0 hash1 words</code></p>	<p>232DD...634C Is it [napier][password][Ankle123][inkwell]?</p> <p>5F4DC...CF99 Is it [napier][password][Ankle123][inkwell]?</p> <p>6D587...5FF5 Is it [napier][password][Ankle123][inkwell]?</p> <p>04013...698D Is it [napier][password][Ankle123][inkwell]?</p>
2	<p>Using the method used in the first part of this tutorial, find crack the following for names of fruits (the fruits are all in lowercase):</p> <p>FE01D67A002DFA0F3AC084298142ECCD 1F3870BE274F6C49B3E31A0C6728957F 72B302BF297A228A75730123EFEF7C41 8893DC16B1B2534BAB7B03727145A2BB 889560D93572D538078CE1578567B91A</p>	<p>FE01D:</p> <p>1F387:</p> <p>72B30:</p> <p>8893D:</p> <p>88956:</p>

## 7 Hashing Cracking (LM Hash/Windows)

All of the passwords in this section are in lowercase.

No	Description	Result
1	On Kali, and using John the Ripper, and using a word list with the names of fruits, crack the following pwdump passwords:	Fred: Bert:

	fred:500:E79E56A8E5C6F8FEAAD3B435B51404EE:5EBE7DFA074DA8EE8AEF1FAA2BBDE876::: bert:501:10EAF413723CBB15AAD3B435B51404EE:CA8E025E9893E8CE3D2CBF847FC56814:::	
2	On Kali, 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: Fred: Bert:
3	On Kali, 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: Bert: Admin:

## 8 Python tutorial

In Python, we can use the Hazmat (Hazardous Materials) library to implement symmetric key encryption.

Web link (Cipher code): <http://asecuritysite.com/cipher01.zip>

The code should be:

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.backends import default_backend

import hashlib
import sys
import binascii

val='hello'
password='hello123'

plaintext=val
```

```

def encrypt(plaintext,key, mode):
    method=algorithms.AES(key)
    cipher = Cipher(method,mode, default_backend())
    encryptor = cipher.encryptor()
    ct = encryptor.update(plaintext) + encryptor.finalize()
    return(ct)

def decrypt(ciphertext,key, mode):
    method=algorithms.AES(key)
    cipher = Cipher(method, mode, default_backend())
    decryptor = cipher.decryptor()
    pl = decryptor.update(ciphertext) + decryptor.finalize()
    return(pl)

def pad(data,size=128):
    padder = padding.PKCS7(size).padder()
    padded_data = padder.update(data)
    padded_data += padder.finalize()
    return(padded_data)

def unpad(data,size=128):
    padder = padding.PKCS7(size).unpadder()
    unpadded_data = padder.update(data)
    unpadded_data += padder.finalize()
    return(unpadded_data)

key = hashlib.sha256(password.encode()).digest()
print("Before padding: ",plaintext)
plaintext=pad(plaintext.encode())
print("After padding (CMS): ",binascii.hexlify(bytearray(plaintext)))

ciphertext = encrypt(plaintext,key,modes.ECB())
print("Cipher (ECB): ",binascii.hexlify(bytearray(ciphertext)))

plaintext = decrypt(ciphertext,key,modes.ECB())
plaintext = unpad(plaintext)
print("  decrypt: ",plaintext.decode())

```

How is the encryption key generate?

Which is the size of the key used? [128-bit][256-bit]

Which is the encryption mode used? [ECB][CBC][OFB]

Now update the code so that you can enter a string and the program will show the cipher text. The format will be something like:

```
python cipher01.py hello mykey
```

where “hello” is the plain text, and “mykey” is the key. A possible integration is:

```
import sys
if (len(sys.argv)>1):
    val=sys.argv[1]
if (len(sys.argv)>2):
    password=sys.argv[2]
```

Now determine the cipher text for the following (the first example has already been completed):

Message	Key	CMS Cipher
“hello”	“hello123”	0a7ec77951291795bac6690c9e7f4c0d
“inkwell”	“orange”	
“security”	“qwerty”	
“Africa”	“changeme”	

Finally, change the program so that it does 256-bit AES with CBC mode.