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

Part 1 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 x=6, Alice selects y=9 | Now calculate (using the Kali calculator):  Bob’s A value (gx mod N):    Alice’s B value (gY mod N): |
| **2** | Now they exchange the values. Next calculate the shared key: | Bob’s value (Bx mod N):    Alice’s value (AY mod N):  Do they match? [Yes] [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 generate a random number, and so should they. Do not tell them what your random number is. Next calculate your A value, and get them to do the same.  Next exchange values. | Numbers for g and N:  Your x value:  Your A value:  The B value you received:  Shared key:  Do they match: [Yes] [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:  openssl list -cipher-commands  openssl version | Outline five encryption methods that are supported:  Outline the version of OpenSSL: |
| **3** | Using openssl and the command in the form:  openssl prime –hex 1111 | 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  openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin  and enter your password. | Use following command to view the output file:  cat encrypted.bin  Is it easy to write out or transmit the output: [Yes][No] |
| **5** | Now repeat the previous command and add the –base64 option.  openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin –base64 | Use following command to view the output file:  cat encrypted.bin  Is it easy to write out or transmit the output: [Yes][No] |
| **6** | Now repeat the previous command and observe the encrypted output.  openssl enc -aes-256-cbc -in myfile.txt -out encrypted.bin –base64 | Has the output changed from the run in 4? [Yes][No]  Why has it changed? |
| **7** | Now let’s decrypt the encrypted file with the correct format:    openssl enc -d -aes-256-cbc -in encrypted.bin -pass pass:*napier* -base64 | Has the output been decrypted correctly?  What happens when you use the wrong password? |
| **8** | 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.  To receive a file, you listen on a given port (such as Port 1234)  nc -l -p 1234 > enc.bin  And then send to a given IP address with:  nc -w 3 [IP] 1234 < enc.bin | Did you manage to decrypt their message? [Yes][No] |

## 3 Public Key

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | 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? |
| **2** | Use following command to view the output file:  Cat private.pem | What can be observed at the start and end of the file: |
| **3** | Next we view the RSA key pair:  openssl rsa -in private.pem -text -noout  You should now see the public exponent (e), the private exponent (d), the two prime numbers (p and q), and the public modulus (N). | Which number format is used to display the information on the attributes:  Which are the elements of the key shown:  Which are the elements of the public key?  Which are the elements of the private key?  What does the –noout option do? |
| **4** | Let’s now secure the encrypted key with 3-DES:    openssl rsa -in private.pem -des3 -out key3des.pem  You should NEVER share your private key. |  |
| **5** | Next, we will export the public key:  openssl rsa -in private.pem -out public.pem -outform PEM -pubout | View the output public key.  What does the header and footer of the file identify?  Is the public key smaller in size than the private key? [Yes/No] |
| **6** | Now we will encrypt with our public key:  openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin |  |
| **7** | And then decrypt with our private key:  openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt | 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** | Next create the crt file with the following:  openssl req -new -key private.pem -out cert.csr | View the CRT file by double clicking on it from the File Explorer.  Using Google to search, what is PKCS#10, and which is it used for?  What is the type of public key method used:  What is the public key size (in bits): [512][1024][2048] |
|  | 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  Graphical user interface, text, application, chat or text message  Description automatically generated | 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>

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | Using:  <http://asecuritysite.com/encryption/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]? |
| **2** | Using:  <http://asecuritysite.com/encryption/md5>  Determine the number of hex characters in the following hash signatures. | MD5 hex chars:  SHA-1 hex chars:  SHA-256 hex chars:  How does the number of hex characters relate to the length of the hash signature: |
| **3** | On Kali, 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. | The passwords are password, napier, inkwell and Ankle123.  [Hint: openssl passwd -apr1 -salt ZaZS/8TF napier]  Bill’s password:  Mike’s password:  Fred’s password:  Ian’s password:  Jane’s password: |
| **4** | On Kali, download the following:  <http://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: |
| **5** | From Kali, download the following ZIP file:  <http://asecuritysite.com/letters.zip> | View the letters. Are they different?  Now determine the MD5 signature for them. What can you observe from the result? |

## 6 Hashing Cracking (MD5)

<http://youtu.be/Xvbk2nSzEPk>

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | On Kali, next create a words 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]? |
| **2** | 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):  FE01D67A002DFA0F3AC084298142ECCD  1F3870BE274F6C49B3E31A0C6728957F  72B302BF297A228A75730123EFEF7C41  8893DC16B1B2534BAB7B03727145A2BB  889560D93572D538078CE1578567B91A | FE01D:  1F387:  72B30:  8893D:  88956: |

## 7 Hashing Cracking (LM Hash/Windows)

All of the passwords in this section are in lowercase. <http://youtu.be/Xvbk2nSzEPk>

|  |  |  |
| --- | --- | --- |
| **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:500:E79E56A8E5C6F8FEAAD3B435B51404EE:5EBE7DFA074DA8EE8AEF1FAA2BBDE876:::  bert:501:10EAF413723CBB15AAD3B435B51404EE:CA8E025E9893E8CE3D2CBF847FC56814::: | Fred:  Bert: |
| **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: |

Repeat all 7.1, 7.2 and 7.3 using **Ophcrack**, and the rainbow table contained on the instance (rainbow\_tables\_xp\_free).

## 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” |  |

Now modify the code so that the user can enter the values from the keyboard, such as with:

cipher=input('Enter cipher:')

password=input('Enter password:')

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