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

Part 1 Demo: <http://youtu.be/3n2TMpHqE18>

The hashcat version has a time-out, so enter the following command:

date -s "1 OCT 2015 18:00:00"

## 1 Diffie-Hellman

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | On Kali, login and get an IP address using:  sudo dhclient eth0 | What is your IP address? |
| **2** | Bob and Alice have agreed on the values:  G=2879, N= 9929  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): |
| **3** | 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] |
| **4** | 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 Private Key

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **1** | Use:  openssl list-cipher-commands  openssl version | Outline five encryption methods that are supported:  **AES256, Base64, DES, RC2, Seed**  Outline the version of OpenSSL:  **1.0.1e** |
| **2** | 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] |
| **3** | 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] |
| **4** | 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**] |
| **5** | 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? [**Yes**][No]  Why has it changed?  **Contents have been encoded in Base64 from the already encoded characters of Base64** |
| **6** | 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?  **No!**  What happens when you use the wrong password?  **Throws a bad decrypt error** |
| **7** | 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** | First 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:  **RSA**  How long is the default key:  **1024 bits**  How long did it take to generate a 1,024 bit key?  **Less than a second**  View the contents of the keys. |
| **2** | Use following command to view the output file:  Cat private.pem | What can be observed at the start and end of the file:  **Begin RSA Private Key**  **End RSA Private Key** |
| **3** | Next we view the RSA key pair:  openssl rsa -in private.pem -text -noout | Which are the attributes of the key shown:  **Modulus, PublicExponent, PrivateExponent, prime1, prime2, exponent1, exponent2, coefficient**  Which number format is used to display the information on the attributes:  **Hex**  What does the –noout option do?  **Outputs the key in readable attributes** |
| **4** | Let’s now secure the encrypted key with 3-DES:    openssl rsa -in private.pem -des3 -out key3des.pem |  |
| **5** | Next we will export the public key:  openssl rsa -in private.pem -out public.pem -outform PEM -pubout | View the output key. What does the header and footer of the file identify?  **The process type of the key, and the DEK information** |
| **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  **Nothing originally, put text hello in, encrypted, and received hello once decrypted.** |
| **8** | If you are working in the lab, now give your password 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  openssl x509 -req -in cert.csr -signkey private.pem -out server.crt | View the CRT file by double clicking on it from the File Explorer.  What is the type of public key method used:  **RSA**  View the certificate file and determine:  The size of the public key: **1024**  The encryption method: **AES-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: **32**  SHA-1 hex chars: **40**  SHA-256 hex chars: **64**  How does the number of hex characters relate to the length of the hash signature:  **Each char is 1 byte, each byte is 4 bits. Therefore 32\*4 for md5** |
| **3** | 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: **napier**  Mike’s password: **Ankle123**  Fred’s password: **inkwell**  Ian’s password: **password**  Jane’s password: **napier** |
| **5** | 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:  **MD5(2.txt)= 69faab6268350295550de7d587bc323d** |
| **6** | From your Kali, download the following ZIP file:  <http://asecuritysite.com/letters.zip> | View the letters. Are they different? **No**  Now determine the MD5 signature for them. What can you observe from the result?  **There is an MD5 collision** |

## 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 this lab we will encrypt a string with a public key, and the decrypt with the private key.

🕮 **Web link (Cipher code):** https://asecuritysite.com/encryption/rsa12

The code should be:

from Crypto.Util.number import \*  
from Crypto import Random  
import Crypto  
import gmpy2  
import sys  
  
bits=60  
msg="Hello"  
  
p = Crypto.Util.number.getPrime(bits, randfunc=Crypto.Random.get\_random\_bytes)  
q = Crypto.Util.number.getPrime(bits, randfunc=Crypto.Random.get\_random\_bytes)  
  
n = p\*q  
PHI=(p-1)\*(q-1)  
  
e=65537  
d=(gmpy2.invert(e, PHI))  
  
m= bytes\_to\_long(msg.encode('utf-8'))  
  
c=pow(m,e, n)  
res=pow(c,d ,n)  
  
print "Message=%s\np=%s\nq=%s\nN=%s\ncipher=%s\ndecipher=%s" % (msg,p,q,n,c,(long\_to\_bytes(res)))

Prove the operation of the code. Now, try with 128-bit prime numbers and 256-bit prime numbers. What can you observe from the increase in the prime number size?

Can you integrate a timer in your code, so that you can assess the time to encrypt and decrypt? Now complete the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Prime number size** | **Time to generate primes** | **Time to encrypt** | **Time to decrypt** |
| 60 |  |  |  |
| 128 |  |  |  |
| 256 |  |  |  |

We can write a Python program to implement this key exchange. Enter and run the following program:

import random

import base64

import hashlib

import sys

g=11

p=1001

x=random.randint(5, 10)

y=random.randint(10,20)

A=(g\*\*x) % p

B=(g\*\*y) % p

print 'g: ',g,' (a shared value), n: ',p, ' (a prime number)'

print '\nAlice calculates:'

print 'a (Alice random): ',x

print 'Alice value (A): ',A,' (g^a) mod p'

print '\nBob calculates:'

print 'b (Bob random): ',y

print 'Bob value (B): ',B,' (g^b) mod p'

print '\nAlice calculates:'

keyA=(B\*\*x) % p

print 'Key: ',keyA,' (B^a) mod p'

print 'Key: ',hashlib.sha256(str(keyA)).hexdigest()

print '\nBob calculates:'

keyB=(A\*\*y) % p

print 'Key: ',keyB,' (A^b) mod p'

print 'Key: ',hashlib.sha256(str(keyB)).hexdigest()

Pick three different values for g and p, and make sure that the Diffie Hellman key exchange works:

g= p=

g= p=

g= p=

Can you pick a value of g and p which will not work?

The code given below allows you to pick a value of g which will always work for a given value of p. Can you integrate the code and prove that it works?

<https://asecuritysite.com/encryption/pickg>

def getG(p):

for x in range (1,p):

rand = x

exp=1

next = rand % p

while (next <> 1 ):

next = (next\*rand) % p

exp = exp+1

if (exp==p-1):

print rand

print getG(p)

Using the prime number generates given in the RSA code, can you implement a Diffie-Hellman method which uses 256 bit prime numbers?