# Lab 5: Key Exchange

**Objective:** Key exchange allows us to pass a shared secret key between Bob and Alice. The main methods for doing this are either encrypting with the public key, the Diffie Hellman Method and the Elliptic Curve Diffie Hellman (ECDH) method. This lab investigates these methods.

🕮 **Web link (Weekly activities):**

https://github.com/billbuchanan/appliedcrypto/tree/master/unit05\_key\_exchange

🕮 **Web link (Demo):** <https://youtu.be/Lnw4FhiOwiU>

## A Diffie-Hellman

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **A.1** | Bob and Alice have agreed on the values:  g=2879, N= 9929  Bob Select x=6, Alice selects y=9 | Now calculate (using a calculator):  Bob’s B value (gx mod N): 4850    Alice’s A value (gY mod N): 3614 |
| **A.2** | Now they exchange the values. Next calculate the shared key: | Bob’s value (Ax mod N): 4868    Alice’s value (BY mod N): 4868  Do they match? [Yes] ~~[No]~~ |
| **A.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:  g: 321012345678901234567890123456789012345678901, N: 234512345678901234567890123456789012345678901  Your x value:  alices\_secret: 23456789012345678901234567890,  Your A value:  Alice's to send to Bob: 221011797940781211555778940349547081051880338,  The B value you received:  bobs\_secret: 32123456789  Bob's to send to Alice: 67863768260288046757322098643564897150447101  Shared key:  Bob's Key: 27884524121888161762800222170715726891890071, Alice's Key: 27884524121888161762800222170715726891890071  Do they match: [Yes] ~~[No]~~ |

## B OpenSSL (Diffie-Hellman and ECC)

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **B.1** | Generate 768-bit Diffie-Hellman parameters:  openssl dhparam -out dhparams.pem -text 768  View your key exchange parameters with:  cat dhparams.pem | What is the value of g:  2  How many bits does the prime number have?  (97 - 1 redundant) = 96 \* 8 = 768  How long does it take to produce the parameters for 1,024 bits (Group 2)?  2 secs  How long does it take to produce the parameters for 1536 bits (Group 5)?  Around 1 min  How would we change the g value?  Openssl dhparam takes flags -2, -3 or -5 to change the g value by changing the group |

|  |  |  |
| --- | --- | --- |
| **No** | **Description** | **Result** |
| **B.2** | Let’s look at the Elliptic curves we can create:  openssl ecparam -list\_curves  We can create our elliptic parameter file with:  openssl ecparam -name secp256k1 -out secp256k1.pem  Now view the details with:  openssl ecparam -in secp256k1.pem -text -param\_enc explicit -noout  What are the details of the key?  Now we can create our key pair:  openssl ecparam -in secp256k1.pem -genkey -noout -out mykey.pem  Can you explain how you would use these EC parameters to perform the ECDH key exchange? | Name three 160-bit curves:  secp160r1 : SECG curve over a 160 bit prime field  wap-wsg-idm-ecid-wtls7: SECG/WTLS curve over a 160 bit prime field  brainpoolP160r1: RFC 5639 curve over a 160 bit prime field  By doing a search on the Internet, which curve does Bitcoin use?  secp256k1  Curve 2559 is a popular curve. Using Google, can you find some popular uses of Curve 25519?  IoT devices |
|  | Bob picks a secret value b (aka sk) which becomes B or b added G times B = b.G  Alice picks a secret value a, which becomes A or a added G times A = a.G    Bob and Alice swap computed values, i.e. Bob sends B to Alice and Alice send A to Bob.    The shared secret then becomes:  For Bob:  b.A or b.a.G  For Alice:  a.B or a.b.G    The shared symmetric key will be created using HMAC Key Derivation Function HKDF and the shared secret. | |

## C Discrete Logarithms

**C.1** ElGamal and Diffie Hellman use discrete logarithms. This involves a generator value (*g*) and a prime number. A basic operation is gx (mod p). If p=11, and g=2, determine the results (the first two have already been completed):

|  |  |
| --- | --- |
| **x** | **g=2, p=11**  **gx (mod p)** |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 5 |
| 5 | 10 |
| 6 | 9 |
| 7 | 7 |
| 8 | 3 |
| 9 | 6 |
| 10 | 1 |
| 11 | 2 |
| 12 | 4 |

Note: In Python you can implement this as:

g=2

p=11

x=3

print (g\*\*x % p)

What happens to the values once we go past 10?

It starts to repeat at 11

What happens to this sequence if we use *g*=3?

It starts to repeat at x = 6

**C.2** We can determine the values of g which will work for a given prime number with the following:

import sys

p=11

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))

Code: [**https://asecuritysite.com/dh/pickg**](https://asecuritysite.com/dh/pickg)

Run the program and determine the possible g values for these prime numbers:

p=11: 2, 6, 7, 8

p=41: 6, 7, 11, 12, 13, 15, 17, 19, 22, 24, 26, 28, 29, 30, 34, 35

On the Internet, find a large prime number, and determine the values of g that are possible:

**C.3** We can write a Python program to implement this key exchange. Enter and run the following program:

import random

import hashlib

import sys

g=9

p=997

a=random.randint(5, 10)

b=random.randint(10,20)

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

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

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

print ('\nAlice calculates:')

print ('a (Alice random): ',a)

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

print ('\nBob calculates:')

print ('b (Bob random): ',b)

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

print ('\nAlice calculates:')

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

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

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

print ('\nBob calculates:')

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

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

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

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

g = 110 p=2699

Key: 2f57a2e309c463513b962750096f4389e229001dccc19afcb95ff98c70e14d09

g= 5013 p= 5641

Key: a0a5df5022dc9c2fc77a391c090e6025de8801366bc5d4c0a9a39df44fe0c7c5

g= 8732 p= 7853

Key: 5480ab857f30bc9abdc0d88179b66cb30b6a294029f8bed71e3b606a19941359

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

Not so far

The following program sets up a man-in-the-middle attack for Eve:

import random

import base64

import hashlib

import sys

g=15

p=1011

a= 5

b = 9

eve = 7

message=21

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

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

Eve1 = (A\*\*eve) % p

Eve2 = (B\*\*eve) % p

Key1= (Eve1\*\*a) % p

Key2= (Eve2\*\*b) % p

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

print ('\n== Random value generation ===')

print ('\nAlice calculates:')

print ('a (Alice random): ',a)

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

print ('\nBob calculates:')

print ('b (Bob random): ',b)

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

print ('\n==Alice sends value to Eve ===')

print ('Eve takes Alice\'s value and calculates: ',Eve1)

print ('Alice gets Eve\'s value and calculates key of: ',Key1)

print ('\n==Bob sends value to Eve ===')

print ('Eve takes Bob\'s value and calculates: ',Eve2)

print ('Bob gets Eve\'s value and calculates key of: ',Key2)

## D Elliptic Curve Diffie-Hellman (ECDH)

ECDH is now one of the most used key exchange methods, and uses the Diffie Hellman method, but adds in elliptic curve methods. With this Alice generates (a) and Bob generates (b). We select a point on a curve (G), and Alice generates aG, and Bob generates bG. They pass the values to each other, and then Alice received bG, and Bob receives aG. Alice multiplies by a, to get abG, and Bob will multiply by b, and also get abG. This will be their shared key.

**D.1** In the following we will implement ECDH using the secp256k1 curve (as used in Bitcoin). Confirm that Bob and Alice will have the same shared key.

🕮 **Web link (ECDH):** https://asecuritysite.com/hazmat/hashnew13

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import ec

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.backends import default\_backend

import binascii

import sys

Bob\_private\_key = ec.generate\_private\_key(ec.SECP256K1(),default\_backend())

Alice\_private\_key = ec.generate\_private\_key(ec.SECP256K1(),default\_backend())

Bob\_shared\_key = Bob\_private\_key.exchange(ec.ECDH(), Alice\_private\_key.public\_key())

Bob\_derived\_key = HKDF(algorithm=hashes.SHA256(),length=size,salt=None,info=b'',backend=default\_backend()).derive(Bob\_shared\_key)

Alice\_shared\_key = Alice\_private\_key.exchange(ec.ECDH(), Bob\_private\_key.public\_key())

Alice\_derived\_key = HKDF(algorithm=hashes.SHA256(),length=size,salt=None,info=b'',backend=default\_backend()).derive(Alice\_shared\_key)

print ("Name of curve: ",Bob\_private\_key.public\_key().curve.name)

print (f"Generated key size: {size} bytes ({size\*8} bits)")

vals = Bob\_private\_key.private\_numbers()

print (f"\nBob private key value: {vals.private\_value}")

vals=Bob\_private\_key.public\_key()

enc\_point=binascii.b2a\_hex(vals.public\_bytes(encoding=serialization.Encoding.PEM,format=serialization.PublicFormat.SubjectPublicKeyInfo)).decode()

print("Bob's public key: ",enc\_point)

vals = Alice\_private\_key.private\_numbers()

print (f"\nAlice private key value: {vals.private\_value}")

vals=Alice\_private\_key.public\_key()

enc\_point=binascii.b2a\_hex(vals.public\_bytes(encoding=serialization.Encoding.PEM,format=serialization.PublicFormat.SubjectPublicKeyInfo)).decode()

print("Alice's public key: ",enc\_point)

print ("\nBob's derived key: ",binascii.b2a\_hex(Bob\_derived\_key).decode())

print("Alice's derived key: ",binascii.b2a\_hex(Alice\_derived\_key).decode())())

Run the code and confirm that Bob and Alice will always get the same shared key.

Yup

Now modify the code to implement the SECP192R1 and also for the SECP521R1 curve. What do you notice about the sizes of the keys created between the different curve types?

The private keys differ in size between the curves but the derived keys are the same size.

**D.2** The code to implement Curve 25519 for key exchange (X25519) is:

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric.x25519 import X25519PrivateKey

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.backends import default\_backend

import binascii

import sys

Bob\_private\_key = X25519PrivateKey.generate()

Alice\_private\_key = X25519PrivateKey.generate()

size=32 # 256 bit key

Bob\_shared\_key = Bob\_private\_key.exchange(Alice\_private\_key.public\_key())

Bob\_derived\_key = HKDF(algorithm=hashes.SHA256(),length=size,salt=None,info=b'',backend=default\_backend()).derive(Bob\_shared\_key)

Alice\_shared\_key = Alice\_private\_key.exchange(Bob\_private\_key.public\_key())

Alice\_derived\_key = HKDF(algorithm=hashes.SHA256(),length=size,salt=None,info=b'',backend=default\_backend()).derive(Alice\_shared\_key)

print ("Name of curve: Curve 25519")

vals = binascii.b2a\_hex(Bob\_private\_key.private\_bytes(serialization.Encoding.Raw,serialization.PrivateFormat.Raw,serialization.NoEncryption()))

print (f"\nBob private key value: {vals}")

vals=Bob\_private\_key.public\_key()

enc\_point=binascii.b2a\_hex(vals.public\_bytes(encoding=serialization.Encoding.DER,format=serialization.PublicFormat.SubjectPublicKeyInfo)).decode()

print("Bob's public key: ",enc\_point)

vals = binascii.b2a\_hex(Alice\_private\_key.private\_bytes(serialization.Encoding.Raw,serialization.PrivateFormat.Raw,serialization.NoEncryption()))

print (f"\nAlice private key value: {vals}")

vals=Alice\_private\_key.public\_key()

enc\_point=binascii.b2a\_hex(vals.public\_bytes(encoding=serialization.Encoding.DER,format=serialization.PublicFormat.SubjectPublicKeyInfo)).decode()

print("Alice's public key: ",enc\_point)

print ("\nBob's derived key: ",binascii.b2a\_hex(Bob\_derived\_key).decode())

print("Alice's derived key: ",binascii.b2a\_hex(Alice\_derived\_key).decode())

Do Bob and Alice end up with the same key?

yes

In this case we have DER format for the public key. This normally starts with a "03". From the test run, copy the DER value and paste it [here](https://asecuritysite.com/encryption/sigs4):

Bob's public key: 302a300506032b656e032100387d723aafba7214a6d58b503d6e870128c2605e1d7fcec206ef854fac2f105a

Alice's public key: 302a300506032b656e032100fbf0fe799ced9077a20df60f85ec29b5673d1538adcd2e349f69532c9e610d16

<https://asecuritysite.com/digitalcert/sigs5>

Can you view the public key point?

With the DER form, you should find there is an OID of "1.3.101.110". From an Internet search, what does "1.3.101.110" represent?

<https://oid-rep.orange-labs.fr/get/1.3.101.110>

OID Repository

If you change the "DER" to "PEM" how does it change the viewing of the keys (remember to remove binascii.b2a\_hex() method)?

Public keys become much longer, other keys stay the same length

## E Simple Key Distribution Centre (KDC)

Rather than using key exchange, we can setup a KDC, and where Bob and Alice can have long-term keys. These can be used to generate a session key for them to use. Enter the following Python program, and prove its operation:

import hashlib

import sys

import binascii

import Padding

import random

from Crypto.Cipher import AES

from Crypto import Random

msg="test"

def encrypt(word,key, mode):

plaintext=pad(word)

encobj = AES.new(key,mode)

return(encobj.encrypt(plaintext))

def decrypt(ciphertext,key, mode):

encobj = AES.new(key,mode)

rtn = encobj.decrypt(ciphertext)

return(rtn)

def pad(s):

extra = len(s) % 16

if extra > 0:

s = s + (' ' \* (16 - extra))

return s

rnd = random.randint(1,2\*\*128)

keyA= hashlib.md5(str(rnd).encode()).digest()

rnd = random.randint(1,2\*\*128)

keyB= hashlib.md5(str(rnd).encode()).digest()

print('Long-term Key Alice=',binascii.hexlify(keyA))

print('Long-term Key Bob=',binascii.hexlify(keyB))

rnd = random.randint(1,2\*\*128)

keySession= hashlib.md5(str(rnd).encode()).hexdigest()

ya = encrypt(keySession.encode(),keyA,AES.MODE\_ECB)

yb = encrypt(keySession.encode(),keyB,AES.MODE\_ECB)

print("Encrypted key sent to Alice:",binascii.hexlify(ya))

print("Encrypted key sent to Bob:",binascii.hexlify(yb))

decipherA = decrypt(ya,keyA,AES.MODE\_ECB)

decipherB = decrypt(yb,keyB,AES.MODE\_ECB)

print("Session key:",decipherA)

print("Session key:",decipherB)

🕮 **Web link (Simple KDC):** <https://asecuritysite.com/encryption/kdc01>

The program above uses a shared 128-bit session key (generated by MD5). Now change the program so that you generate a 256-bit session key. What are the changes made:

Changes to the random int size

## F Challenge

**F.1** Bob and Alice agree on a *g* value of 5, and a prime number of 97. They then use the Diffie-Hellman key exchange method. Alice passes a value of 32, and Bob passes a value of 41. Can you determine the secret value that Bob and Alice have generated, and the resultant key value? Outline the code here:

g=5

p=97

def calculate(a):

    A = pow(g, a, p)

    if A == 32 or A == 41:

        print(f'g: {g}, p: {p}, a: {a}, A: {A}')

for i in range(0, 100):

    calculate(i)

g: 5, p: 97, a: 74, A: 32

g: 5, p: 97, a: 85, A: 41

What happens if we use a *g* value of 2? Why is there a problem?

Hint: https://asecuritysite.com/encryption/pickg

It never calculates a value for b, Bob’s secret

From asecuritysite.com

“…g=2, we get an output of 2, 4, 1, 2, 4 … for the sequence values of 1, 2, … This means that we do not get a unique output for the values from 1 to 6(where the maximum value will be six as we take the modulus of 7)…”

if g = 2

There are many repeating values in the range however the modulo value never become 41

Can you now write a generate DH key cracker for any value of g, p, A (passed by Alice), and B (passed by Bob) Outline code and run to evaluate the perform of our code with different ranges of the prime number (p):

## G What I should have learnt from this lab?

The key things learnt:

* The basics of the Diffie Hellman method.
* The basic method used with ECDH.