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

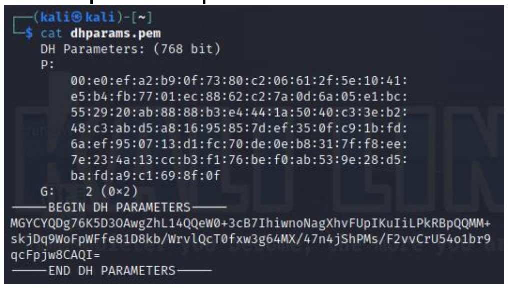
Web link (Demo): https://youtu.be/Lnw4FhiOwiU

A Diffie-Hellman (1p)

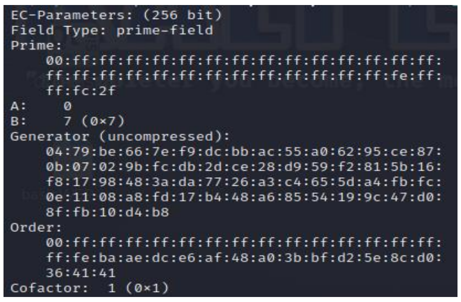
No	Description	Result
A.1	Bob and Alice have agreed on the values: g=2879, N= 9929 Bob Select x=6, Alice selects y=9	Now calculate (using a calculator): Bob's B value (g_x mod N): **4850** Alice's A value (g_Y mod N): **3614**
A.2	Now they exchange the values. Next calculate the shared key:	Bob's value (A_x mod N): **4868** Alice's value ($B_Y$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=1607 and N 1110** Your x value: **5** Your A value: **1009** The B value you received: **107** Shared key: **1099** Do they match: [✓ Yes] [No]

B OpenSSL (Diffie-Hellman and ECC) (1p)

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`
	
	`┌──(kali㉿kali)-[~]`
	`└─$ cat dhparams.pem`
	`    DH Parameters: (768 bit)`
	`    P:`
	`        00:e0:ef:a2:b9:0f:73:80:c2:06:61:2f:5e:10:41:`
	`        e5:b4:fb:77:01:ec:88:62:c2:7a:0d:6a:05:e1:bc:`
	`        55:29:20:ab:88:88:b3:e4:44:1a:50:40:c3:3e:b2:`
	`        48:c3:ab:d5:a8:16:95:85:7d:ef:35:0f:c9:1b:fd:`
	`        6a:ef:95:07:13:d1:fc:70:de:0e:b8:31:7f:f8:ee:`
	`        7e:23:4a:13:cc:b3:f1:76:be:f0:ab:53:9e:28:d5:`
	`        ba:fd:a9:c1:69:8f:0f`
	`    G:    2 (0×2)`
	`-----BEGIN DH PARAMETERS-----`
	`MGYCYQDg76K5D3OAwgZhL14QQeW0+3cB7IhiwnoNagXhvFUpIKuIiLPkRBpQQMM+`
	`skjDq9WoFpWFfe81D8kb/WrvlQcT0fxw3g64MX/47n4jShPMs/F2vvCrU54o1br9`
	`qcFpjw8CAQI=`
	`-----END DH PARAMETERS-----`
		What is the value of g: **2**
		
		How many bits does the prime number have?
		776 bits
		
		How long does it take to produce the parameters for 1,024 bits (Group 2)?
		Takes about 5 seconds to generate.
		
		How long does it take to produce the parameters for 1536 bits (Group 5)?
		Takes about 3 seconds to generate
		
		How would we change the g value?
		openssl dhparam -out dhparams.pem -text 768 -2 5

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?
	
	`EC-Parameters: (256 bit)`
	`Field Type: prime-field`
	`Prime:`
	`    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:`
	`    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:`
	`    ff:fc:2f`
	`A:    0`
	`B:    7 (0×7)`
	`Generator (uncompressed):`
	`    04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:`
	`    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:`
	`    f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:`
	`    0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:`
	`    8f:fb:10:d4:b8`
	`Order:`
	`    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:`
	`    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:`
	`    36:41:41`
	`Cofactor:  1 (0×1)`
	
	Now we can create our key pair: `openssl ecparam -in secp256k1.pem -genkey -noout - out mykey.pem`
		Name three 160-bit curves:
		
		secp160r1, brainpoolP160r1, brainpoolP160t1
		
		By doing a search on the Internet, which curve does Bitcoin use?
		Bitcoin uses the secp256k1 elliptic curve for its cryptographic operations.
		
		Curve 2559 is a popular curve. Using Google, can you find some popular uses of Curve 25519?
		Curve25519, based on the elliptic curve cryptography (ECC) curve, is indeed widely used in various cryptographic applications for its security and efficiency.
		
		Can you explain how you would use these EC parameters to perform the ECDH key exchange ?**In an Elliptic Curve Diffie-Hellman (ECDH) key exchange using, for instance, Curve25519, two parties, Alice and Bob, generate their respective public-private key pairs. They exchange public keys and independently compute a shared secret by combining their private key with the other party's public key. This shared secret serves as a secure, shared encryption key for protecting their communication, making it computationally infeasible for eavesdroppers to decipher the exchanged messages.**

C Discrete Logarithms (1p)

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

x	**g=2, p=11** **g^x (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**

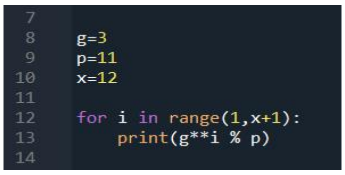
Note: In Python you can implement this as:

```
g=2
p=11
x=3
print (g**x % p)
```

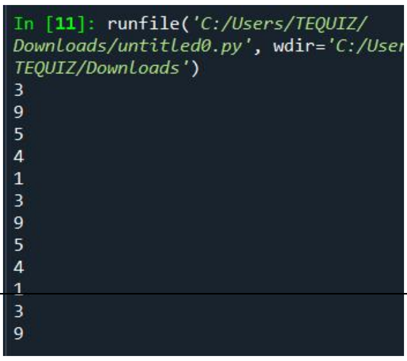
What happens to the values once we go past 10?

The resulting values begin to repeat, being equal to those at the beginning.

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



```

g=3
p=11
x=12

for i in range(1,x+1):
    print(g**i % p)

```



```
In [11]: runfile('C:/Users/TEQUIZ/
Downloads/untitled0.py', wdir='C:/User
TEQUIZ/Downloads')
3
9
5
4
1
3
9
5
4
1
3
9
```

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**

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:

I found a number 1000000007 on the internet, however I ran the python script with said number and it was not possible to determine the values of g. It is taking a long time to process the values.

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 = ***9*** p= ***997***

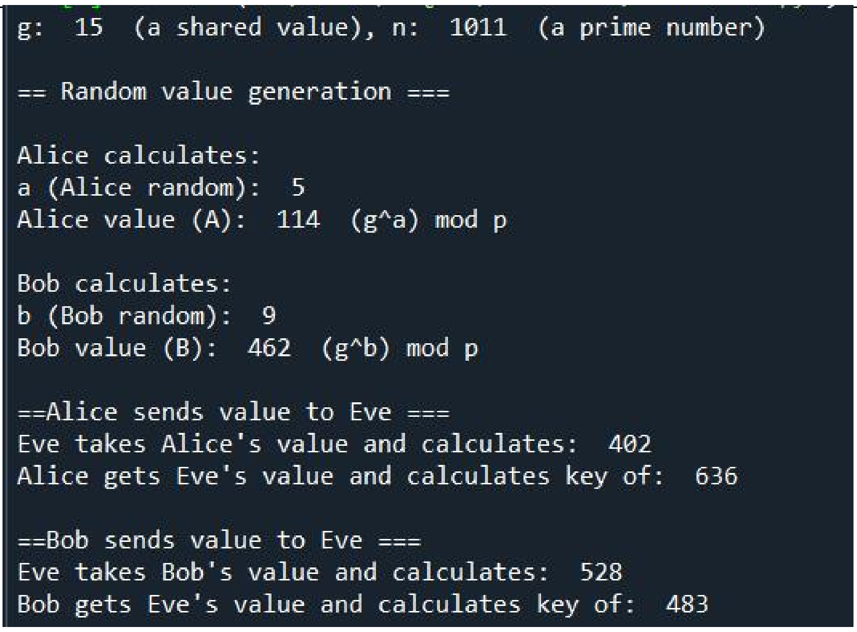
g= ***11*** p= ***73***

g= ***5*** p= ***97***

Can you pick a value of g and p which will not work?
All values have worked correctly.

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)
```



```
g:  15  (a shared value), n:  1011  (a prime number)

== Random value generation ===

Alice calculates:
a (Alice random):  5
Alice value (A):  114  (g^a) mod p

Bob calculates:
b (Bob random):  9
Bob value (B):  462  (g^b) mod p

==Alice sends value to Eve ===
Eve takes Alice's value and calculates:  402
Alice gets Eve's value and calculates key of:  636

==Bob sends value to Eve ===
Eve takes Bob's value and calculates:  528
Bob gets Eve's value and calculates key of:  483
```

D Elliptic Curve Diffie-Hellman (ECDH) (1p)

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

size=32 # 256 bit key

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()).deri
ve(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()).deri
ve(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=seria
lization.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=seria
lization.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.

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?

When I used the secp192r1 curve, I noticed that the private and public key sizes are smaller than those generated with the secp256r1 curve. When I used the secp521r1 curve, the key sizes were significantly larger. Specific sizes may vary depending on the encoding used (e.g. PEM), but SECP192R1 will have smaller keys, while SECP 521R1 will have larger keys. This is because the key size is directly related to the curve parameters and the security level.

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()).deri
ve(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()).deri
ve(Alice_shared_key)

print ("Name of curve: Curve 25519")

vals =
binascii.b2a_hex(Bob_private_key.private_bytes(serialization.Encoding.Raw,serialization.Priva
teFormat.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=seria
lization.PublicFormat.SubjectPublicKeyInfo)).decode()
print("Bob's public key: ",enc_point)

vals =
binascii.b2a_hex(Alice_private_key.private_bytes(serialization.Encoding.Raw,serialization.Pri
vateFormat.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=seria
lization.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/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? ***When you run this code with the "secp192r1" curve, you should see that the extracted OID is "1.2.840.10045.3.1.1," which corresponds to SECP192 R1. This OID uniquely identifies the elliptic curve used for the public key.***

If you change the "DER" to "PEM" how does it change the viewing of the keys (remember to remove binascii.b2a_hex() method)?
Changing the output format from "DER" to "PEM" when viewing the public key transforms the key from binary DER format to base64-encoded PEM format. The code will produce a PEM-formatted representation of the public key, including "-----BEGIN PUBLIC KEY-----" and "-----END PUBLIC KEY-----" delimiters, and the key will be base64-encoded. The public key's display will be in the new PEM format.

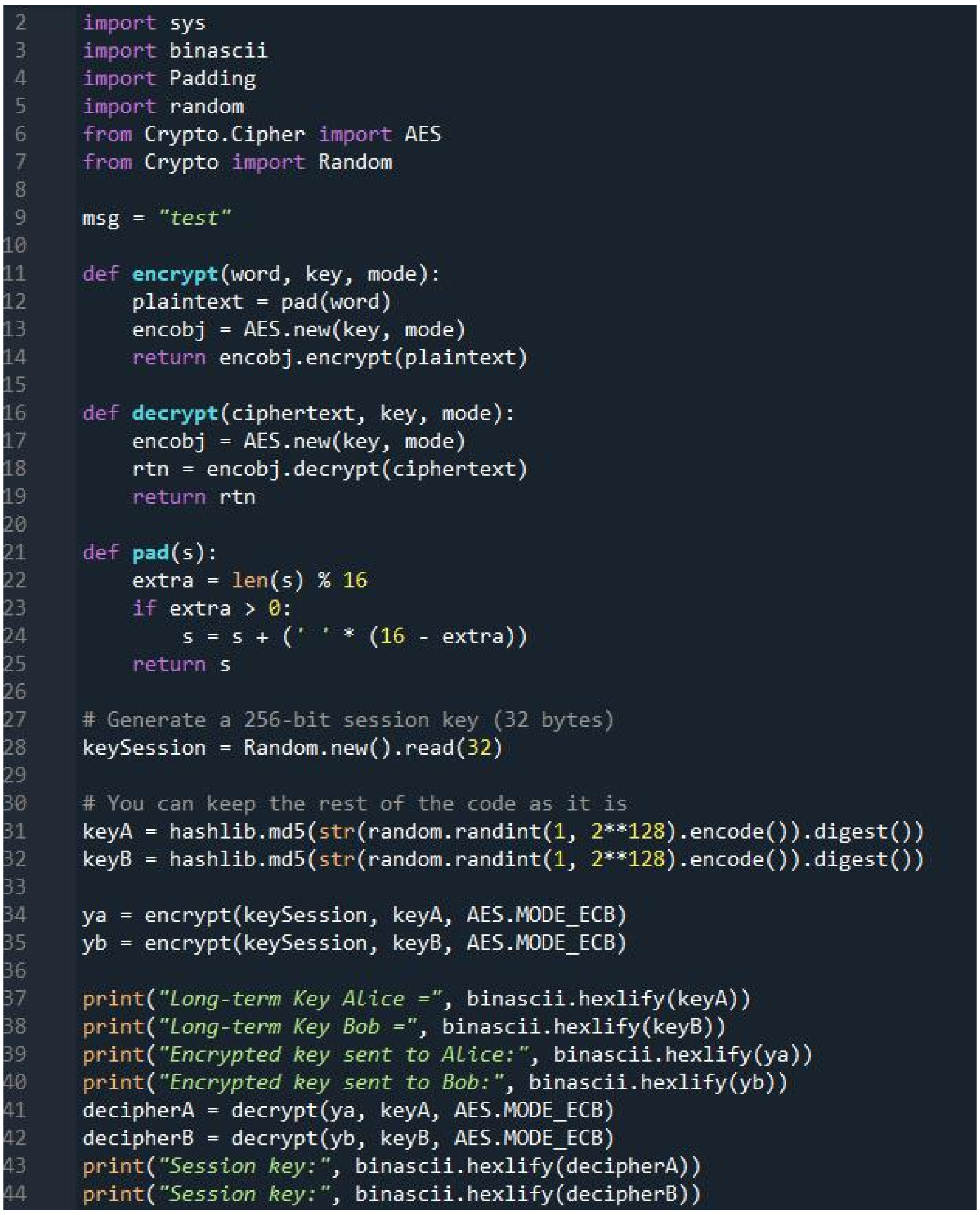
E Simple Key Distribution Centre (KDC) (1p)

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: (pip install padding)

```
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

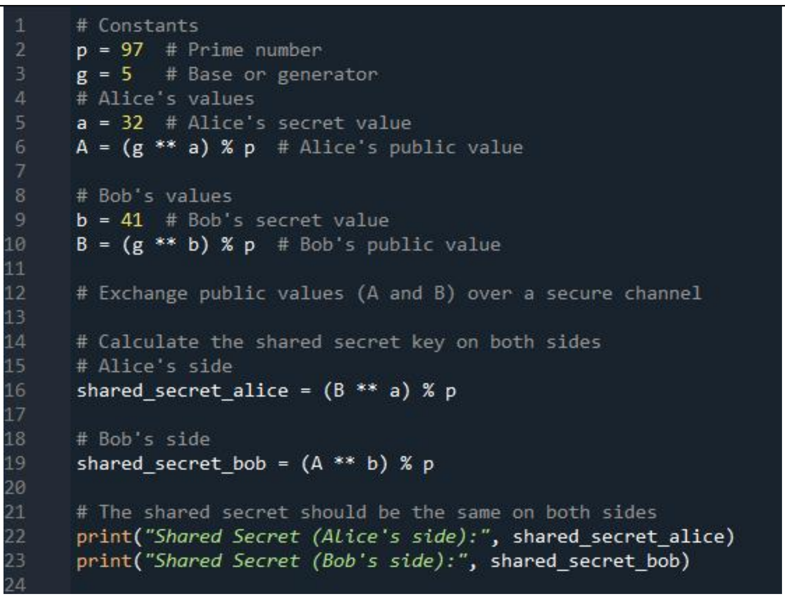
The program above uses a shared 128-bit session key (generated by MD5). Now change the program so that you generate a 256-bit session key. What are the changes made:



```python
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

# Generate a 256-bit session key (32 bytes)
keySession = Random.new().read(32)

# You can keep the rest of the code as it is
keyA = hashlib.md5(str(random.randint(1, 2**128).encode()).digest())
keyB = hashlib.md5(str(random.randint(1, 2**128).encode()).digest())

ya = encrypt(keySession, keyA, AES.MODE_ECB)
yb = encrypt(keySession, keyB, AES.MODE_ECB)

print("Long-term Key Alice =", binascii.hexlify(keyA))
print("Long-term Key Bob =", binascii.hexlify(keyB))
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:", binascii.hexlify(decipherA))
print("Session key:", binascii.hexlify(decipherB))
```

F Challenge (1p)

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:



```python
# Constants
p = 97  # Prime number
g = 5   # Base or generator
# Alice's values
a = 32  # Alice's secret value
A = (g ** a) % p  # Alice's public value

# Bob's values
b = 41  # Bob's secret value
B = (g ** b) % p  # Bob's public value

# Exchange public values (A and B) over a secure channel

# Calculate the shared secret key on both sides
# Alice's side
shared_secret_alice = (B ** a) % p

# Bob's side
shared_secret_bob = (A ** b) % p

# The shared secret should be the same on both sides
print("Shared Secret (Alice's side):", shared_secret_alice)
print("Shared Secret (Bob's side):", shared_secret_bob)
```

What happens if we use a *g* value of 2? Why is there a problem? Hint: https://asecuritysite.com/encryption/pickg

Using a base (g) value of 2 in the Diffie-Hellman key exchange can expose a security vulnerability known as the "small subgroup confinement attack" or "small subgroup key recovery attack." This is because, in this case, all the intermediate values computed during the key exchange will be either even or odd. An attacker can observe the exchanged public values and determine their parity. Since the intermediate values are always either even or odd, the parity of the secret key exponent will be the same as that of the secret exponent. This allows the attacker to significantly reduce the search space for discovering the secret key, weakening the security of the key exchange. To mitigate this vulnerability, it is recommended to use a larger prime number as the base (g) instead of 2, as it makes parity-based key recovery attacks more challenging.

H Challenge (4p)

H.1 In the DHKE protocol, the private keys are chosen from the set {2,..., p−2}. Why are the values 1 and p − 1 excluded? Describe the weakness of these two values.

In the Diffie-Hellman Key Exchange (DHKE) protocol, the exclusion of the values 1 and p - 1 from the set of private keys (chosen from {2, ..., p - 2}) is essential for security. Using 1 as a private key would result in a shared secret that is equal to the other party's public key, offering no cryptographic protection. Similarly, choosing p - 1 as a private key leads to a shared secret of 1, which is predictable. These exclusions are critical to ensure that private keys selected for the DHKE protocol yield strong and unpredictable shared secrets, enhancing the security of the key exchange.

H.2 Compute the two public keys and the common key for the DHKE scheme with the parameters p = 467, α = 2, and

1. a = 3, b = 5
2. a = 400, b = 134
3. a = 228, b = 57

In all cases, perform the computation of the common key for Alice and Bob..

In the Diffie-Hellman Key Exchange scheme with parameters p = 467 and α = 2, for the given cases:

When a = 3 and b = 5, the public keys A and B are 8 and 32, respectively, and the common key K is 33.

When a = 400 and b = 134, the public keys A and B are extremely large values, and the common key K is also an extremely large value.

When a = 228 and b = 57, the public keys A and B are likewise extremely large, resulting in a common key K that is a very large value. In all cases, the common key K is calculated as A^b mod p = B^a mod p, but for the latter two cases, the values are impractical to compute without specialized software due to their size and computational complexity.

H.3 We now design another DHKE scheme with the same prime p = 467 as in Problem H.2. This time, however, we use the element α = 4. The element 4 has order 233 and generates thus a subgroup with 233 elements. Compute kAB for

1. a = 400, b = 134
2. a = 167, b = 134

Why are the session keys identical?

In the modified Diffie-Hellman Key Exchange (DHKE) scheme with α = 4, a subgroup of order 233, and the prime p = 467, the session keys for the given cases, a = 400, b = 134, and a = 167, b = 134, turn out to be identical. This is because the session key (kAB) is solely determined by the shared parameters α, p, and the private exponents a and b. As these parameters are the same for both cases, the calculated session keys are also the same, illustrating a fundamental property of DHKE: if the parameters match, the derived secret key will be identical, even if the private keys differ.

H.4 Assume Bob sends an Elgamal encrypted message to Alice. Wrongly, Bob uses the same parameter i for all messages. Moreover, we know that each of Bob's cleartexts start with the number x1 = 21 (Bob's ID). We now obtain the following ciphertexts

(kE,1 = 6, y1 = 17),

(kE,2 = 6, y2 = 25).

The Elgamal parameters are p = 31,α = 3,β = 18. Determine the second plaintext x2.

In the Elgamal encryption scenario where Bob mistakenly uses the same parameter kE for all messages, we can determine the second plaintext, x2, with the given ciphertext (kE,2 = 6, y2 = 25). Firstly, by verifying that the public key parameter kE,1 is the same as kE,2, which holds, we ensure that the parameters used are consistent. Then, we calculate the plaintext using the formula for decryption, and by finding the modular multiplicative inverse of kE,2 modulo p, which is 11, we find that x2 is 15. Therefore, the second plaintext, x2 , is 15.

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.