# CSC 321 – Module 3 Assignment
# Public Ciphers

Tobin Bohley
Matthew Suban

## Introduction

## Task 1 - Diffie-Hellman Key Exchange

In this task we implemented a simple demonstration of the Diffie-Hellman key exchange. First we used very small, insecure parameters then for part two we switched them out for real-world, 1024-bit parameters.

### Task 1 Code

```
alice_rand = random.randrange(1, q)     #each party gets a random private value
bob_rand = random.randrange(1, q)
print('Alice private key: ', alice_rand)
print('Bob private key: ', bob_rand)

alice_y = pow(a, alice_rand, q)     #each party computes their public value
bob_y = pow(a, bob_rand, q)
print('Alice sends: ', alice_y)
print('Bob sends: ', bob_y)

#Alice "sends" Bob  alice_y, and Bob sends bob_y

alice_s = pow(bob_y, alice_rand, q)     #computing s for both parties
bob_s = pow(alice_y, bob_rand, q)
print('Alice computes shared key s: ', alice_s)
print('Bob computes shared key s: ', bob_s)

alice_SHA = Crypto.Hash.SHA256.new()        #shared key computation
alice_SHA.update(str(alice_s).encode())

bob_SHA = Crypto.Hash.SHA256.new()
bob_SHA.update(str(bob_s).encode())         #shared key computation

print('Alice SHA:   ',alice_SHA.hexdigest())
print('BOB SHA:     ', bob_SHA.hexdigest())          #view that the two are identical

alice_key = bytes(alice_SHA.hexdigest()[:16].encode())
bob_key = bytes(bob_SHA.hexdigest()[:16].encode())      #shortened key for AES

msg = bytes('Hello, this is an encrypted message from Alice'.encode())
print('original message:', msg)

alice_cipher = AES.new(alice_key, AES.MODE_ECB)
```

```
bob_cipher = AES.new(bob_key, AES.MODE_ECB)     #both parties create ciphers to use on
their own

pad_len = 16 - (len(msg) % 16)
msg_padded = msg + bytes([pad_len] * pad_len)   #PCKS#7 padding

secret = alice_cipher.encrypt(msg_padded)
print('encrypted message (with Alice\'s cipher):    ',secret)

print('decrypted message(with Bob\'s cipher):       ', bob_cipher.decrypt(secret))
hexdigest=alice_SHA.hexdigest()
```

## Code output (small params):

```
Alice private key:  11
Bob private key:  26
Alice sends:  2
Bob sends:  21
Alice computes shared value s:  3
Bob computes shared value s:  3
Alice SHA:
4e07408562bedb8b60ce05c1decfe3ad16b72230967de01f640b7e4729b49fce
BOB SHA:
4e07408562bedb8b60ce05c1decfe3ad16b72230967de01f640b7e4729b49fce

original message: b'Hello, this is an encrypted message from
Alice'

encrypted message (with Alice's cipher):
b'\x92\x05e\x14=h\x81\x8d2D\xfdo\xc6\x92\xf0\xa6x\x9a\xa8*i\xce\
xc3\xae\xc1\xdd\x01(\xe0\xec\xc6\x0b\xbca\x7f=L\xc1\xff\xae!\x1d
\xa5\xaf\x19$\xb0\x8d'

decrypted message(with Bob's cipher):        b'Hello, this is an
encrypted message from Alice\x02\x02'
```

## Code output (real-world params):

```
Alice private key:
714621512466614033812443450326711811057786867675064524310271053075073717250473215536782663746867762629936868976332353330623188865727440160233797987597341458888763827634603208354293460803199021471206106035205655242088514068980016101…
Bob private key:
713766680354216037640962185173107300613685288353261156004009559962191827367685664405428070509198700761328848031698217381399650178779230865973773092403465119051648077961781706469567745636210619686413040316237575427962397737830488051…
Alice sends:
902807991692017689882244245765142285091071087835095790558377407807621016381345868391923102596483715464662798421614529072877160135292381996070714863934260998030455225345850749133400502970790197284007916082684025647751121434033436611…
Bob sends:
210159236296315025140615975521993961415109089373837358348542114827658902888115210676956152288152252215930987553695842149486748547269625525189152776853190397417968976075542520108127932079540717647339745972575450293781786020482663491…
Alice computes shared value s:
652278852132970598815171871906785690835544817118976655047418471…
Bob computes shared value  s:
652278852132970598815171871906785690835544817118976655047418471…
Alice SHA:
68eac374d4ffe90f316a6e958f911808b313ca2bb3df518945c23e7848bde55d
BOB SHA:
68eac374d4ffe90f316a6e958f911808b313ca2bb3df518945c23e7848bde55d
original message: b'Hello, this is an encrypted message from Alice'
encrypted message (with Alice's cipher):
b'\xdfH+\xdb-P\x8cErVL\x8d.\xdeO\xff\x14\xa1<\xa8\xa9\x1c\xa1;\xfd\xab\xacy\x8cy\x9bp\xdc\x871\xd1|\x8d\xd1\x12vo\x90\xe0\xc5\x02\x7fp'
decrypted message (with Bob's cipher):        b'Hello, this is an encrypted message from Alice\x02\x02'
```

**Questions**

*For task 1, how hard would it be for an adversary to solve the Diffie Hellman Problem (DHP) given these parameters? What strategy might the adversary take?*

Given small parameters q = 37, a = 5, it would be pretty easy and fast for a bad actor to solve the Diffie Hellman problem. There's not very many values to test, and by creating enough SHA-256 hashes they'd very easily be able to find one that works as a cipher key to decrypt any communication. For this case they could create hashes from values between 1 and 37, and as soon as they tried 3 (the shared key) they'd have a hash that could decrypt any of the messages sent between the two.

*For task 1, would the same strategy used for the tiny parameters work for the large values of q and a? Why or why not?*

No, probably not even with a ton of computing power behind the attack. The reason is the magnitude of possible shared keys they'd have to try. With 1024 bits of possibilities, it would take inordinate timespans of computation to guess the right shared key.

## Task 2 - MITM Key Fixing & Negotiated Groups

In task 2 we modified our task 1 code slightly to include Mallory, a theoretical bad actor. To do this she intercepts their exchange of $Y_a$ and $Y_b$ and replaces them with q. So, when computing the value *s*, Bob and Alice unknowingly use q instead of each other's public values which inevitably leads to a result of 0 since it is modded with itself during the operation.

### Task 2 Code

```
alice_rand = random.randrange(1, q)     #each party gets a random private value
bob_rand = random.randrange(1, q)

alice_y = pow(a, alice_rand, q)     #each party computes their public value
bob_y = pow(a, bob_rand, q)

#--------------------------Mallory intercepts, sending q instead of alice_y and bob_y
print('Mallory intercepted, sending q instead of Ya and Yb')

alice_s = pow(q, alice_rand, q)     #computing s for both parties
bob_s = pow(q, bob_rand, q)

print('Alice computes shared key s: ', alice_s)
print('Bob computes his shared key s: ', bob_s)
print('The key is 0 because q^n mod q = 0 for all n!\n')

alice_SHA = Crypto.Hash.SHA256.new()        #public key computation
alice_SHA.update(str(alice_s).encode())

bob_SHA = Crypto.Hash.SHA256.new()
bob_SHA.update(str(bob_s).encode())         #public key computation

mallory_SHA = Crypto.Hash.SHA256.new()      #Mallory can generate a SHA using 0
mallory_SHA.update('0'.encode())            #since she knows they will have 0 too

print('Alice SHA:   ',alice_SHA.hexdigest())
print('Bob SHA:     ',bob_SHA.hexdigest())          #view that the two are identical
print('Mallory SHA: ',mallory_SHA.hexdigest())

print('Mallory, knowing that the shared key is 0, can compute the same SHA\n')
alice_key = bytes(alice_SHA.hexdigest()[:16].encode())
bob_key = bytes(bob_SHA.hexdigest()[:16].encode())      #shortened key for AES
mallory_key = bytes(mallory_SHA.hexdigest()[:16].encode())
```

```
print('Now Alice is going to send a message to Bob')
msg = bytes('Hello, this is an encrypted message from Alice'.encode())
print('original message:', msg)

alice_cipher = AES.new(alice_key, AES.MODE_ECB)
bob_cipher = AES.new(bob_key, AES.MODE_ECB)     #both parties create ciphers to use on
their own
mallory_cipher = AES.new(mallory_key, AES.MODE_ECB)


pad_len = 16 - (len(msg) % 16)
msg_padded = msg + bytes([pad_len] * pad_len)   #PCKS#7 padding

secret = alice_cipher.encrypt(msg_padded)
print('encrypted message (with Alice\'s cipher):    ',secret)

print('decrypted message (with Mallory\'s cipher): ', mallory_cipher.decrypt(secret))
#print('decrypted message(with Bob\'s cipher):       ', bob_cipher.decrypt(secret))

hexdigest=alice_SHA.hexdigest()
print('\nMallory can successfully breach Alice and Bob\'s confidentiality')
```

## Code output:

```
Mallory intercepted, sending q instead of Ya and Yb
Alice computes shared key s:  0
Bob computes his shared key s:  0
The key is 0 because q^n mod q = 0 for all n!

Alice SHA:
5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9
Bob SHA:
5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9
Mallory SHA:
5feceb66ffc86f38d952786c6d696c79c2dbc239dd4e91b46729d73a27fb57e9
Mallory, knowing that the derived number is 0, can compute the
same SHA

Now Alice is going to send a message to Bob
```

```
original message: b'Hello, this is an encrypted message from
Alice'
encrypted message (with Alice's cipher):
b"\xb7\x11\xc7\xfb\xd9\xb9\xd4\xbe\xe3V\xe6g\xe3\x17\xe0\xd0>\x1
3\x07\x0e\x8b;\xc0\xc9\x7f\xf4M\xdeI\t6\t!\xbb\xb9
S\xd8'\r2\x026\xc1/\xc1q\xb3"
decrypted message (with Mallory's cipher):  b'Hello, this is an
encrypted message from Alice\x02\x02'
Mallory can successfully breach Alice and Bob's confidentiality.
```

**Task 2 Part 2**

The code is largely the same for this example, the changed part is what the generator is set to at the beginning and, based on that, the value Mallory uses to seed her hash in order to compute the shared key.

**Output (generator = q-1):**

```
Alice Xa:
740474350507481475214239004700358541226590384922278849995375725921734
531913834393374604931530064944497525200323383350110932987021450406066
954291280610742327014006463924814711774736285053699395682967771…
Bob Xb:
490818155467100882100204565821410627936884456541331340936707393599264
546691728667256127005937688865886701763402070107776018267781365469889
485801309750107863822877286183063345148931711879692752638350246…
Alice Ya:
835598388627951694911776052923598045419309364678383314755346342635727
781761754433795586680826909943289290633955425714988257700992742646854
442325945759772101722298239066823132407537580627330963854543985…
Bob Yb:  1
Alice computes shared value s:  1
Bob computes shared value s:  1

At this point Mallory knows the shared key based on how she changed
the generator, that is enough info to generate the same hash and
cipher and enough to decrypt their messages

Alice SHA:
6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b
Bob SHA:
6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b
Mallory SHA:
6b86b273ff34fce19d6b804eff5a3f5747ada4eaa22f1d49c01e52ddb7875b4b
Mallory, knowing that the shared value is 1, can compute the same SHA
```

The rest of the cases in task 2 part 2 work pretty much the same way. When the generator is set to 0, Bob and Alice will compute 0 as their s. When it is set to 1, they will compute 1 for their s as in the code output.

*For task 2, why were these attacks possible? What is necessary to prevent it?*
These attacks take advantage of how the modulus operator works and how it's used in the Diffie-Helman key exchange. Modulus by definition wraps around when it reaches a certain value, and so by changing the generator attackers can force it to become 0 or 1 by ensuring that any numbers that are mod'd are multiples (or near multiples) of the prime modulus parameter. In order to prevent the attack it is pretty easy, actually. Both parties just need to protect the integrity of their number exchanges, even if the confidentiality is breached. So signing each message would work, except of course... that requires secure communication to already exist, defeating the purpose. Regardless, if preliminary exchange is able to have its integrity protected, the eventual encryption/decryption will be secure.

## Task 3 - “Textbook” RSA & MITM Key Fixing via Malleability

In this task we implemented “textbook” RSA, using key generation, encryption, and decryption; we experimented with changing the length of the prime bits to see how it would affect the key size and runtime, as well as practicing encryption/decryption.

In Part 2, we tested to see how malleability can also affect signature schemes based on RSA. In our case we used two existing signatures in order to create a valid third signature.

- **Task 3 Key Implementation**
    - Key Functions (for RSA Implementation)
        - rsa_keygen(prime_bits=1024, e=65537):
            - Generates two primes, ensuring that the greatest common divider (gcd(e,phi(n)) == 1)
        - rsa_encrypt(m_int, n, e):
            - Encrypts m_int by raising m_int^e mod n
        - rsa_decrypt(c_int, n, d):
            - Decrypts message by finding c^d mod n
    - Output
        -
```
Message 1: b'Hello RSA'
m1 (int): 1335473908826942624577
c1: 136965246854975527169720585707939681618503287982301504
418170990886668239166246262086801183313137720908835322404 3
321683070309593262620907579883120801568116205273013341694 2
212675727095505363935404870933206086066603198255456952432 2
909993881502745295392802317766667922151250443641181721970 6
912456417336192584328308178880754487864947157234509200910 3
067649769132533637902652860436244609762808553924539745621 2
282016266894557264474049664488652299525670764491769451082 0
996224445176831428823543980559422756319179444163778561448 8
273489107438466771244783445771312011965184478147985415886 2
182519635148488020526382588372179440722 3
decrypted: b'Hello RSA'
```
        -

```
Message 2: b'Another message to encrypt'
m2 (int): 10514418692945943469632600226667387971609087018250595585919806
8
c2: 12522998204157776106248850113830616440367116121432922402370330428454
1639053075470467652624509004951440434647415917544200284315102897307587
4923210672909142215893596615345903555349559702391581274788098846746008
0713288672739508321550699008332553454608514607262870548442148906393762
3362928034032726025180256297311555257294427849461897847621623844678522
4651705315764632218669550949355126101946297591423456108458148997073336
6539647763381219736184068652987183320785535623149174764477505804924644
5951114513016724125616462214311746879331689806508847299382703986104953
0839922138338445509993188663136696075214048363262988123198 31
decrypted: b'Another message to encrypt'
```

- **Part 2 Implementation**
  - Output & Verification

```
Task 3 Part 2 (extra): RSA signature malleability
Signature scheme: Sign(m,d) = m^d mod n  (textbook / insecure)
m1: 219186979155226033980009691673433975836473234972216747769903941711328612934
m2: 301591352437421499819518417897130066416477538095363502241397849322964942426 64
m3 = (m1*m2) mod n: 498749540620895751918024902512644577488276606196122131272192 16328083009411002

Verify signatures by raising to e mod n:
sig1^e mod n == m1 ? True
sig2^e mod n == m2 ? True
forged sig3 = sig1*sig2 mod n
sig3^e mod n == m3 ? True
```

- **Questions**:
  - For task 3 part 1, while it's very common for many people to use the same value for e in their key (common values are 3, 7, $2^{16}+1$), it is very bad if two people use the same RSA modulus n. Briefly describe why this is, and what the ramifications are.
    - The RSA modulus is calculated as the product of two distinct (secret) prime numbers p & q, and determines the key length. However, since the security effectiveness of RSA relies on the computational infeasibility of factoring n to find both p & q, if two people use the same value for e in their key (eg. e = p, q), then it becomes much easier to factor back into their original counterparts and would therefore break the security of the cryptosystem.6