Diego Garay, Salvador Gutierrez

Bret Hartman

CPE 321-03

March 8th 2022

Public Ciphers

Questions

1. **For Task1, how hard would it be for an adversary to solve the Diffie Hellman Problem(DHP) given these parameters? What strategy might the adversary take?**
   It would take a bad actor a relatively short amount of time(computationally it would be expensive, but modern technology would make quick work of our small parameters) to solve the DHP given the parameters. However, given that the DHP gets exponentially computationally expensive, it is widely used. It was created based on the difficulty of computing discrete logarithms.
   In the DHP certains elements must be made publicly available. More specifically, a prime number p, and it's primitive root q would be known to a bad actor as it would be public. Using this information, a bad actor could theoretically brute-force guess the private key being used by a singular party by using $private\ key = q^{unknown\ integer} \% q$ and trying every integer between 0 and p for the unknown integer.

2. **For Task1, would the same strategy used for the tiny parameters work for the p and g? Why or why not.**
   Yes. The exact same strategy can and would be used. The larger the P and G however, the longer it would take to brute-force compute the private/secret keys. Given the

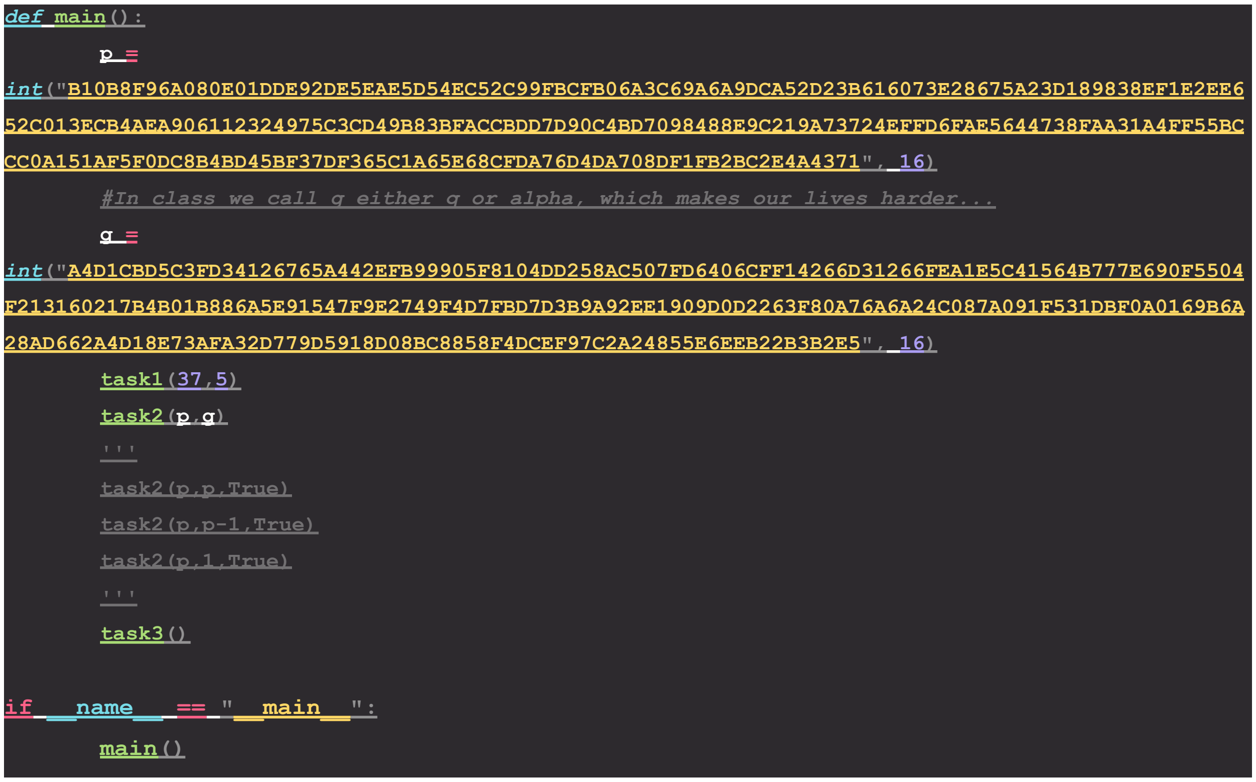
sharp increase in the amount of entropy that can be generated from the originally used P and G, the computation time it takes to solve the DHP would go from minutes to hundreds of years.

3. **For Task2, why were these attacks possible? What is necessary to prevent it?**
   These attacks are possible because the Diffie-Hellman exhanchange protocol has no user authentication. If a bad actor was motivated enough, they could target the messages between two users, and then recreate the private keys used by both parties. If successful, this bad actor could intercept the messages of both parties, and then impersonate both parties. So, if Alice and Bob are communicating with each other, Mallory could go about slowly but surely recreating both Alice and Bob's private keys. Once this is done, Mallory could pretend to be Bob to Alice, and then pretend to be Alice to Bob. If neither party is informed, then Mallory could simply pass the messages along without alerting either Alice or Bob that something is wrong, this is a passive man-in-the-middle attack. An active M.i.T.M. attack would be if Mallory would alter the contents of any message Alice/Bob would send to each other, before sending the message(s) forward.

4. **For Task3, while it's very common for many people to share an e(common values are 3,7,216+1), it is very bad if two people share an RSA modulus n. Briefly describe why this is, and what the ramifications are.**
   Sharing an e-value is common because e value is only used to derive a d-value, and even then the d-value is created using Omega, which is only known privately to an individual

user. However, if an RSA modulus n is shared, then theoretically anyone could reverse engineer someone else’s public key. A bad actor using their own e and d values could figure out the factors used for n, and then recreate omega, which can be used to recreate someone’s private key.

Code Main:



```
def main():
        p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D189838EF1E2EE6
52C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C219A73724EFFD6FAE5644738FAA31A4FF55BC
CC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708DF1FB2BC2E4A4371", 16)
        #In class we call g either g or alpha, which makes our lives harder...
        g =
int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564B777E690F5504
F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A
28AD662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24855E6EEB22B3B2E5", 16)
        task1(37,5)
        task2(p,g)
        '''
        task2(p,p,True)
        task2(p,p-1,True)
        task2(p,1,True)
        '''
        task3()

if __name__ == "__main__":
        main()
```

Task 1:

Description: For Task1, we were told to implement the Diffie-Hellman Key Exchange. Given that the Key-Exchange is meant to happen between two Users within a network, we implemented the task using Python Objects that represented Users. Within this User Class, we had methods used for creating secret/symmetric keys, and for encrypting/decrypting information using AES CBC. The process works by a User first choosing a public P/G value, alongside generating an initialization vector. Subsequently, when two users would like to communicate, they must share their public keys with each other and then create a new secret/symmetric key based on the combination of both user’s keys. Once this is done, the user's can now start to encrypt information using their individual symmetric keys, and then send these encrypted messages to the other user, who will decrypt the message using their individual symmetric key. There is a lot of fancy mathematics involved with this process, but it can be made a lot clearer if seen visually(https://www.youtube.com/watch?v=Yjrfm_oRO0w)

CODE:

```
def task1(p,g):
   IV = secrets.token_bytes(16)
   #usingo our good ole classic alice and bob
   usr1 = User(p,g,IV, "Alice")
   usr2 = User(p,g,IV, "Bob")

   #now that we've generated publive keys, we need to generate secret keys using the other's
public key
   usr1.genSecretKey(usr2.pub)
   usr2.genSecretKey(usr1.pub)

   #create a symmetric key using the secret key to share information
   usr1.genSymmetricKey()
   usr2.genSymmetricKey()

   #start sharing information using the symmetric key
   encryptedMsg1 =usr1.encrypt("I am a secret")
   encryptedMsg2 =usr2.encrypt("I am also a secret")
   #^^^ First half of Task1 ^^^#

   #vvv Second half of Task1 vvv#
   p =
int("B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D189838EF1E2EE6
52C013ECB4AEA906112324975C3CD49B83BFACCBDD7D90C4BD7098488E9C219A73724EFFD6FAE5644738FAA31A4FF55BC
CC0A151AF5F0DC8B4BD45BF37DF365C1A65E68CFDA76D4DA708DF1FB2BC2E4A4371", 16)
   g =
int("A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564B777E690F5504
F213160217B4B01B886A5E91547F9E2749F4D7FBD7D3B9A92EE1909D0D2263F80A76A6A24C087A091F531DBF0A0169B6A
28AD662A4D18E73AFA32D779D5918D08BC8858F4DCEF97C2A24855E6EEB22B3B2E5", 16)

   usr3 = User(p,g,IV, "Laura")
   usr4 = User(p,g,IV, "Frankie")

   usr1.genSecretKey(usr2.pub)
   usr2.genSecretKey(usr1.pub)

   usr1.genSymmetricKey()
   usr2.genSymmetricKey()

   msg1 = "Hahaha, don't tell John the secret"
   msg2 = "hahah, I told Laura's secret"

   encMsg1 = usr1.encrypt(msg1)
   encMsg2 = usr2.encrypt(msg2)
   print(f"I am {usr3.whoami()} this is my message{encMsg1}")
   print(f"I am {usr4.whoami()} this is my message{encMsg2}")

   decMsg1 = usr1.decrypt(encMsg2)
   decMsg2 = usr2.decrypt(encMsg1)

   print(f"{usr3.whoami()} found: {decMsg1}")
   print(f"{usr4.whoami()} found: {decMsg2}")
```

Task 2:

Description: Given that our implementation of the DHP was naive, there are attacks that our code is susceptible to, one of these attacks being Man-in-the-Middle attacks. To begin, we started off by simply showing that if Mallory could somehow intercept the Public keys of Alice/Bob and instead send both parties the same number(which is effectively making them both use the same "key"), then Mallory could then use this very same "key" she sent to decrypt messages being sent between Alice and Bob. This would be made possible because in effect, all 3 parties would be using the same secret key to encrypt/decrypt all messages being sent.

Subsequently, in the 2nd part of Task2, we were tasked with showing that if Mallory has done her research(as in she knows what initialization vector either Alice/Bob used) and bad P/G values are used, there is also a strong vulnerability present. In the case of using a simple G value, such as G being equal to 1 or being equal to P-1(Modulus math can lead to interesting similarities between numbers), then the public/secret keys will always end up being the number 1. In the same sense, if P=G, then Modulus math would dictate that the public/private keys would always be 0. So in any case, Mallory would in effect have access to the public/keys used by Alice and Bob, which would be identical, which would mean all 3 parties are in effect using the same information.

In either case, Mallory is able to intercept messages between 2 parties because she was able either to force the public/private keys to be identical, or because Modulus math made the public/private keys identical.

CODE:

```
def task2(p, g,mal=False):
   IV = secrets.token_bytes(16)
   usr1 = User(p,g,IV, "Alice")
   usr2 = User(p,g,IV, "Bob")
   #modification of the public keys by Mallory
   #usr1.pub = p
   #usr2.pub = p

   usr1.genSecretKey(usr2.pub)
   usr2.genSecretKey(usr1.pub)

   usr1.genSymmetricKey()
   usr2.genSymmetricKey()

   msg1 = "hahah, I am secret"
   msg2 = "ahahah, I am also a secret"

   encMsg1 = usr1.encrypt(msg1)
   encMsg2 = usr2.encrypt(msg2)
```

```
decMsg1 = usr1.decrypt(encMsg2)
decMsg2 = usr2.decrypt(encMsg1)

print(f"{usr1.whoami()} found: {decMsg1}")
print(f"{usr2.whoami()} found: {decMsg2}")

#case where we don't want to showcase Mallory's attacks!
if mal == False:
    return

usr3 = User(p,g,IV, "Mallory")
usr3.pub = 1
usr3.secKey = 1
usr3.genSymmetricKey()
recoveredMsg1 = usr3.decrypt(encMsg1)
recoveredMsg2 = usr3.decrypt(encMsg2)

if g == 1:
    print("g==1: Abused Modulus Math(keys will be 1)")
    print(f"I am {usr3.whoami()} and these are the messages I recoverd")
    print(f"Message1: {recoveredMsg1}Message2: {recoveredMsg2}\n")

elif g == p:
    print("g==p: Abused Modulus Math(keys will be 0)")
    usr3.pub = 0
    usr3.secKey = 0
    usr3.genSymmetricKey()
    recoveredMsg1 = usr3.decrypt(encMsg1)
    recoveredMsg2 = usr3.decrypt(encMsg2)
    print(f"I am {usr3.whoami()} and these are the messages I recoverd")
    print(f"Message1: {recoveredMsg1}Message2:{recoveredMsg2}\n")

elif g == p - 1:
    print("g==p-1: Abused Modulus Math(keys will be 1)")
    print(f"I am {usr3.whoami()} and these are the messages I recoverd")
    print(f"Message1: {recoveredMsg1}Message2: {recoveredMsg2}\n")
```

Task 3:

Description: For Task3, we were told to implement "Textbook" RSA and then M.I.T.M. Key fixing. Implementing Textbook RSA was rather straight-foward. To begin, the first part of the RSA is the key generation, which is done by finding two random prime numbers(P & Q) based on the bit-length provided to us by the user. Once these prime numbers are found, we create an “n” value which is the result of the P*Q. Simultaneously, we also calculate an Omega value, which is the result of (P-1)*(Q-1). With N & Omega, all we need is an “e” value(which was provided to us in the specs), and we can then start to calculate the public/private keys. As an intermediate step(we can forgo this step completely if wished), we create a “d” value, which is just the rearranging of the equation $1 = d * (e\%omega)$. With our D value calculated, we finally have our public/private keys, with the Public Key being {e,n} and the Private Key being {d,n}.

For the operation F(), we must assume that Mallory has done her research and is aware of the initialization vector used by Alice/Bob. Furthermore, we must assume that Mallory is capable of intercepting a message from Bob to Alice(Because Mallory only has Alice’s information). With this message, and the n value from Alice’s Public key, she should be able to decrypt messages being sent to Alice/Bob by doing some simple modulus math. Firstly, Mallory would have to figure out the prime numbers used to generate Alice’s n-value. With these numbers, and the e-value from Alice’s public key, Mallory should theoretically be able to recreate the D-value being used by Alice(which is used to create her signature). Combining all these elements, if Mallory has Bob’s message to Alice, and the D/N values being used by Alice, then Mallory could pretend to be Alice to Bob with the equation $signature = ciphertext^{d} \% n$.

Finally, we were tasked with figuring out how Mallory could recreate the valid signature for a new message, using two previous messages. As was discussed above, the equation for

generating a signature is $signature\ =\ ciphertext^{d}\ \%\ n$. Any signature being sent between two parties can be verified by using the equation $m = s^{e}\ \%\ n$. Since Mallory is already aware of e/n(she has Alice's public key), then mallory can create two valid signatures sent to her by Alice to recreate a third.
Assume that Mallory sends an intiatial message(m0) to Alice where they establish signature(s0). Now, further assume that Mallory sends a second message(m2), where the message is of length (length(m2)/m1) % n. Alice would then send back a 2nd signature(s1). Using s0 & s1, Mallory could create s2, a third valid signature, by using the equation $s2\ =\ (s1\ *\ s0)\ \%\ m2$

CODE:

```
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Random import get_random_bytes
from Crypto.Util.number import getPrime
from Crypto.Util.Padding import pad, unpad

byteOrder = "little"
blckLen = 16
def task3():
   msg = input("What is your message?: ")
   bitLen = int(input("Bit-length?(256, 1024, 2048): "))
   #vvv Textbook RSA
   hMsg = msg.encode().hex()
   iMsg = int(hMsg, blckLen)
   keyTup = generateKeys(bitLen)
   pubKey = keyTup[0]
   privKey = keyTup[1]
   ciphertext = encrypt(iMsg, pubKey)
   print(f"CipherText: {ciphertext}")
   plaintext = decrypt(ciphertext,privKey)
   print(f"Encoded-Plaintext: {plaintext}")
   ##^^^Textbook RSA
   IV = get_random_bytes(blckLen)
   encryptedMsg1 = keyFixing(pubKey, privKey, msg, IV)
   recoveredMsg = MalleabilitySignatures(encryptedMsg1, IV)
   print(f"Recovered Message: {recoveredMsg}")

#generates RSA public and private keys based off a provided string-length
# int -> int list * int list
```

```python
def generateKeys(bitLen):
   p1 = getPrime(bitLen)
   p2 = getPrime(bitLen)
   n = p1 * p2
   #what comes 1 after n?
   e = 65537
   omega = (p1 - 1) * (p2 - 1)
   d = pow(e, -1, omega)
   publicKey = [e, n]
   privateKey = [d, n]
   return (publicKey, privateKey)

#takes a string and private key{e,n} and encrypts the message"
# string * int list -> byte string
def encrypt(msg, pu):
   return pow(msg, pu[0], pu[1])

#inverse of encrypt
# byte_string ->
def decrypt(encMsg, pr):
   return pow(encMsg, pr[0], pr[1])

def keyFixing(pu, pr, msg, iv):
   cPrime = pu[1]
   #pr[0] = e, pr[1]= n
   secKey = pow(cPrime, pr[0], pr[1])
   secKey = secKey.to_bytes(128, byteOrder)
   hashThingy = SHA256.new()
   hashThingy.update(secKey)
   #convert our ASCII MSG into Hex
   digest = hashThingy.hexdigest()
   #convert our hex value into an integer
   intDigest = int(digest, blckLen)
   byteKey = intDigest.to_bytes(35, byteOrder)
   key = byteKey[:blckLen]

   enc = AES.new(key, AES.MODE_CBC, iv)
   bMsg = bytes(msg, "utf-8")
   cNought = enc.encrypt(pad(bMsg, blckLen))
   return cNought

#Mallory recreating valid signatures
def MalleabilitySignatures(cNought, iv):
   hashThingy2 = SHA256.new()
   secKey2 = 0
   secKey2 = secKey2.to_bytes(128, byteOrder)
   hashThingy2.update(secKey2)
   digest = hashThingy2.hexdigest()
```

```
    intDigest2 = int(digest, blckLen)
    byteKey2 = intDigest2.to_bytes(35, byteOrder)
    key2 = byteKey2[:blckLen]
    enc2 = AES.new(key2, AES.MODE_CBC, iv)

    pText = enc2.decrypt(cNought)
    pText = unpad(pText, blckLen)
    plaintext = pText.decode("utf-8")

    return plaintext
```

Resources used:

https://youtu.be/NmM9HA2MQGI

https://www.youtube.com/watch?v=Yjrfm_oRO0w

https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/

https://www.packetmania.net/en/2022/01/22/Python-Textbook-RSA/

https://www.section.io/engineering-education/rsa-encryption-and-decryption-in-python/