Diffie-Hellman key exchange implementation

In order to ensure encrypted communication in our application, we followed the Diffie-Hellman algorithim to implement a method for secure key exchange between two parties. This algorithim is the best method for two parties to be able to secretly share a symmetric key over a communication channel that isn't secure. In the end, this leads to each party possessing a simular shared secret without anyone else being able to determine what the shared secret is. This solves one of the biggest problems in cryptography i.e., the key exchange problem.

In our code we generate two shared secrets. One for encryption purposes and one for hashing purposes. This should improve security by reducing a centralised point of failure from having a single key. Should one of the keys be compromised in some way, only one of the functions (encryption or hashing) should fail. In this case, the failure/key leak should be detectable and precautions can be taken.

With Diffie-Hellman, the two parties must first agree on two parameters they are going to use. A value **g** called a generator, and value **p** which typically is a very large prime number. Each party will then select a secret value **a** and **b** that they keep private from anyone else. This secret value is used to calculate a public key which they can then exchange with each other openly.

By using the public key that was given from the other recipent, they can then combine with initially agreed values of **g** and **p** to calculate a third value, known only to the two parties. This third value is the Diffie-Hellman Shared Secret key.

Once the shared symmetric key is established, each party can then securely communicate over a public channel by exchanging encrypted data between each other that only the two parties sharing the same private key can decrypt. This essentially creates a secure channel for two parties to safely exchange secret information.

To acheive this we followed the same algorithim listed in RFC 2361 paper to generate the keys, while using the perscribed values listed in RFC 3526. For this assignment we used the 1536-bit MODP Group for both the prime number **p** and generator **g**

- Prime number *p*

```
# obtained from RFC 3526
raw_prime = """FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
670C354E 4ABC9804 F1746C08 CA237327 FFFFFFFF FFFFFFFF"""
```

```
# Convert from the value supplied in the RFC to an integer
prime = read_hex(raw_prime)
```

- Generator *g* or primitive root of the selected prime number *p*. This is the smallest multiple of the set primitive roots of *p*.

```
generator = 2 # As per RFC 3526
```

- This function in the __init__.py is first used to calculate the shared public key:

```
def create_dh_key() -> Tuple[int, int]:
    private_key = secrets.randbelow(prime - 2)          # Generate secret
using python secrets
    public_key = pow(generator, private_key, prime)     # Creates a Diffie-
Hellman key

    return (public_key, private_key)       # Returns (public, private)
```

Integrity

An attacker could try to modify a message in transit and hope the receiver still accepts it. We ensure integrity through the use of MACs or Message Authentication Codes. MACs allows us to verify knowledge without revealing details. Assuming that communications is captured, the MAC should not expose underlying information about our plain text data.

MACs achieves this by mapping abritary length strings into fixed length strings. The hashing algorithm should ensure that unique strings should generate unique codes.

Unlike hashes, MACs require a key. This is another reason that we favour MACs. As we are running a botnet, a highly illegal activity where being caught would lead to a jail sentence, we use a keyed hashing to provide better security and to also allow authentication in addition to data integrity. We do note, however this is more resource intensive than non-keyed hashing, but the extra security should be worthwhile.

The MAC framework we choose is HMAC, we choose this over secret suffix, secret prefix and envelope frameworks as HMAC has superior security and isn't weakn to birthday attacks or to length attacks and so on.

```
    # Create the hmac hash using key and plaintext data # Task 3
    hashcode = self.hmac_sha256(self.shared_secret[1], data)
     # Create the dictionary to send
```

```
    dict_to_send = {'nonce':nonce, 'ciphertext':data_to_send,
'hash':hashcode}
```

Within the "send" function of the "comms.py" file, we have the above line. This uses a hashing framework, HMAC, and a hashing algorithm SHA-256 to securely hash our message. The input to this is our second secret shared key and the plain text data. The output is our hash code which is packed into a dictionary and sent with our encrypted message.

```
    # Create the hashcode to check against the received hashcode # Task 3
    hashcode_check = self.hmac_sha256(self.shared_secret[1], original_msg)

    if hashcode_check != hashcode:
        raise ValueError("Hashes didn't match buddddy")
    else:
        print("Hashes are a match! Message wasn't tampered with. OR WAS
IT?")
```

On the receiver side within the "recv" function, we calculate the hashcode with the decrypted message and the shared key as inputs. If the hashes match, we can continue knowing the integrity of the message has not be compromised. Otherwise we raise a ValueError as it indicates the message has been tampered with.

Replay Prevention

Replay is when an attacker tries to resend/retransmit a message, hoping the system still accepts the message. We counter this with the use of a nonce. A nonce is short for number-used-once. It is a random number appended to the message.

```
            # Task 2
            cipher = AES.new(self.shared_secret[0], AES.MODE_CTR)

            # Create Nonce, in this case, automatically generated.
            nonce = cipher.nonce

            # Check if the nonce is already in the set, if it is, generate
a new one.
            while nonce in self.nonce_set:
                cipher = AES.new(self.shared_secret[0], AES.MODE_CTR)
                nonce = cipher.nonce
            else:
                self.nonce_set.add(nonce)

                print(nonce, 'nonce not found in existing set. Adding to
set.')
```

```
            # Encrypt the data with the successful nonce.
            data_to_send = cipher.encrypt(data)

            # Create the dictionary to send
            dict_to_send = {'nonce':nonce, 'ciphertext':data_to_send,
'hash':hashcode}
```

The code above is taken from "comms.py" within the "send" function where we implement a nonce creator/checker. In the creation of a AES cipher object, we automatically create a nonce. We use a while loop to continually generate new cipher objects and the corresponding nonces, checking for duplicates in our set until we generate a non duplicate. Once this occurs, we can encrypt our message with the cipher object and the associated nonce. If a duplicate is created, we simply generate another nonce and cipher object.

Since the nonce is 64 bits we have 2^64 possibilities. So we could generate a million nonces a second for 10 years and not run out of space.

```
            # Extract the nonce and the ciphertext
            nonce = b64['nonce']
            if nonce in self.nonce_set:
                print(nonce, 'nonce found in existing set. Discarding
message.')
                raise ValueError("Nonce was found in the set. Ending
connection.")
            else:
                print(nonce, 'nonce not found in existing set. Adding to
set. Message is not a replay attack.')
```

In the "recv" function we implement a nonce checker, this will be where we detect replay attacks. We exract the nonce from the dictionary and the reciever checks the nonce against a set of existing nonces. If a message has a nonce that already exists in the set, this indicates that the message may be a replay attack. However, if the nonce is unique, we add it to the set, indicating it is a "seen nonce"

Authentication

Why might we want to allow for peer-to-peer file transfers between bots? file transfer = keylogged data that's stolen financial files that's been stolen from user files

What are the advantages and disadvantages to using a central web server (pastebot.net in our case, similar to pastebin.com) to distribute files when controlling a botnet? advantages = patching bots = easier

disadvantage =

Although you did not work on communications between bots and a central server for this assignment, there is a major flaw in the template implementation of bot-server communications. Explain how your

botnet, if used in the real world, could be trivially controlled by other hackers or government agencies. How might one attempt to stop it?