Key Exchange

In order to ensure encrypted communication in our application, we followed the Diffie-Hellman algorithm to implement a method for secure key exchange between two parties. This algorithm 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 recipient, 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 achieve this we followed the same algorithm listed in RFC2361 to generate the keys, while using the prescribed values listed in RFC3526. For this assignment we used the 2048-bit MODP Group for both the prime number **p** and generator **g**. This MODP group should be suitable for our purposes as in section 8 of the document, they estimate the strength of using this particular modulus as between 110 and 160 bits. As we are aiming for a 128 bit strength encryption system, this should in theory make it suitable to balance between system strength and resource usage.

- 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 CA18217C 32905E46 2E36CE3B
        E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
        DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
        15728E5A 8AACAA68 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)
```

Confidentiality

We encrypt our data to hide the contents of our messages. For our system we use AES to encrypt our information. This encryption method was chosen due to its status as a Federal Information Processing Standard as endorsed by NIST. This means the underlying algorithm behind its implementation should be mathematically strong and that it has undergone rigorous scrutiny/testing.

The mode of operation that we chose is CBC (Cipher-Block Chaining) for this implementation. As we are running a botnet, it is possible that the connection from server to client may not be completely stable. CBC prevents errors from propgating beyond two blocks. In CBC, each block of plaintext is XOR'ed with the previous block of ciphertext (or the IV for the first block). This error propgation means that errors will only affect two blocks and can enhance reliability

Additionally, since the botnet will have a central server, the central server must decrypt many messages. While encryption with CBC cannot be done in parallel, decryption can.

This means that although the central server has more messages to decrypt, the time/resources required to decrypt is not as high if parallelisation was not allowed.

Using the shared secret we obtained during the Diffie Hellman key exchange, we create a symmetric key using the AES algorithim to encrypt and decrypt protected data. The code below highlights how we establish confidentiality using AES CBC to encrypt and decrypt data transmission.

```
# For encryption
# Create the cipher object
cipher = AES.new(self.shared_secret[0], AES.MODE_CBC)
# Encrypt the data.
data_to_send = cipher.encrypt(pad(data, AES.block_size))
```

```
# For decryption
# create new cipher object based on received IV.
cipher = AES.new(self.shared_secret[0], AES.MODE_CBC, iv)

# Decrypt the data and unpad our data
plain_text = unpad(cipher.decrypt(ct), AES.block_size)
```

Integrity

Since operation such as CBC only provide guarantees over the confidentiality of the message but not over its 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 allow 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 achieve this by mapping arbitary 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 weak 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, Reject Message")
else:
    print("Hashes are a match! Message integrity confirmed.")
```

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
# Create Nonce with 64 bits of randomness
nonce = get_random_bytes(16)

# Check if the nonce is already in the set, if it is, generate a new one.
while nonce in self.nonce_set:
    nonce = get_random_bytes(16)
    print("Nonce already in set. Generating new nonce.")
else:
    self.nonce_set.add(nonce)
    print(nonce, 'nonce not found in existing set. Adding to set.')

# 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 also create a nonce. We use a while loop to continually generate new 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.

Since the nonce is 64 bits we have 2^64 possibilities. So we could generate a million nonces a second for 100 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 extract the nonce from the dictionary and the receiver 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

Allowing peer-to-peer (P2P) file transfers between bots in a botnet can serve several purposes that may be desirable for certain types of scenarios.

- **Anonymity:** Implementing P2P communication between bots, can provide a level of anonymity for file transfers to occur. Unlike centralized communication through a server, having P2P transfers will make it more challenging for an adversary to trace back to the originator. This enhances the botnet's stealth capabilities.
- **Redundancy:** P2P file transfers between bots can also provide redundancy in file distribution. If one bot is unavailable or compromised, other bots can take its place and still distribute files among others. This enhances resilience in the network.
- **Updates and downloads:** By allowing for P2P file transfers, the bots can transfer large files to the central server or with each other. As we are running a botnet, the files would most likely have value, such as: key logs of the infected computer, stolen files, bitcoin data chunks and so on. Additionally allowing file transfers would enable patches/updates to bot behaviour.

Having a centralised web server to control the distribution of files in a botnet has its own advantages and disadvantages.

Advantages

- **Increased Security:** Having a centralized file distribution can be more secure than P2P transfers, since access controls and encryption can be more easily enforced on a central server. This reduces the risk of unauthorized access to sensitive files
- **Reliability:** A centralised server can offer higher reliability and availability compared to P2P file transfers. P2P can be subject to connectivity issues or bottlenecks among individual bots.
- **Control:** A central web server provides centralized control over file distribution. The botnet operator can easily manage and monitor the files being distributed, ensuring that only authorized content is distributed.
- **Logging and Auditing:** A Centralized server controlling file distribution can facilitate logging and auditing of file transfers. This providing a clear record of which files were distributed, when, and to whom. This can be valuable in situations where forensic analysis is needed.

Disadvantages

- **Single point of failure** Having a centralized file distribution also means that if it fails, the whole botnet is compromised. This introduces a major weakness to the botnet that attackers can focus their attention on. Instead of using resources to break encryption, attackers could try to find weaknesses in the central webserver instead.
- **Hardware Scalability:** In the lectures Conficker is used as an example of a highly 'succesfull' botnet. Conficker is said to have infected millions of computers. We can image the resources required to manage a million bots in terms of encrypting messages, decrypting messages, pushing updates, downloading files from the bots and so on. It is possible that without adequate and properly scaled hardware, the botnet would not operate at maximum efficiency and vulnerabilities could stem from this.
- **Traceability:** As mentioned above, with a million bots the traffic could not be hidden. It should be possible to find patterns in packet routing to deduct where the webserver is operating in. Again with the Conficker example, the botnet was too succesfull in spreading which lead to attention from security analysts. Obvious or large traffic patterns would bring attention to our illegal enterprise.

In the template given the major flaw that would easily allow for control to be taken over is the method of authentication before the Diffie-Hellman Key exchange. The way the template handles key exchange is to immediately share public keys to generate a

shared secret when the client a server. There is no form of authentication to ensure that the client is who they say they are. Once a malicious attacker pretending to be a client (or server) has a shared secret, it would be trivial for the botnet to be controlled.

The usual way that this issue is solved is through the use of a digital certificate system and a certificate authority. As the client is a bot, the certificate authority could use a challenge response system instead of a typical identification system. Challenge response could allow a client to prove their identity without revealing the secret. However new bots must be able to generate the exact same secret which would have its own challenges and vulnerabilities. Once a digital certificate is gained RSA can be used to sign the certificates using private keys with verification done with public keys. This method should allow the server some confidence that the client is friendly and not a malicious attacker.