Google CTF 2018

MITM Writeup

shediyo @ PseudoRandom

In this challenge the setting is 2 parties, a client and a server, first performing handshake in in which a shared key is generated, and after that communicating safely (messages are encrypted with integrity checks).

In the safe communication stage the client can send a 'getflag' encrypted message to the server, on which the server responds with an encrypted flag message.

The goal here might be being able as an active attacker in the "middle" between the parties - affecting the shared key and the possibility of unsafe communication.

For this I checked the handshake function:

```
def Handshake(password, reader, writer):
  myPrivateKey = Private()
  myNonce = os.urandom(32)

  WriteBin(writer, myPrivateKey.get_public().serialize())
  WriteBin(writer, myNonce)

  theirPublicKey = ReadBin(reader)
  theirNonce = ReadBin(reader)

  if myNonce == theirNonce:
    return None
  if theirPublicKey in (b'\x00'*32, b'\x01' + (b'\x00' * 31)):
    return None

  theirPublicKey = Public(theirPublicKey)

  sharedKey = myPrivateKey.get_shared_key(theirPublicKey)
  myProof = ComputeProof(sharedKey, theirNonce + password)

  WriteBin(writer, myProof)
  theirProof = ReadBin(reader)

  if not VerifyProof(sharedKey, myNonce + password, theirProof):
    return None

  return sharedKey
```

The handshake for the server and the client "simultaneously", in steps:

1. Generate a private key and a random nonce.
2. Send the public key (inferred from private key) and the nonce.
3. Receive the other party's nonce and public key, check that their public key is neither 0 nor 1.
4. Generate the shared key using the private key and the other party's public key.

5. Compute an HMAC proof based on the other party's nonce and the shared password, using the shared key as a key for the proof computation.
6. Verify the other party's proof.

Without much reading the curve25119 library code used for the private/public/shared key generation, I just guessed it is a DH (Diffie-Hellma) key exchange, in which for a group G of order p and generator g:

1. The first party generates x as private key and $g^x \ mod \ p$ as public key sent over the network.
2. The second party generates y as private key and $g^y \ mod \ p$ s public key sent over the network.
3. The first party computes $(g^y)^x \ mod \ p$ and the second computes $(g^x)^y \ mod \ p$ and so they both get a shared key $g^{xy} \ mod \ p$.

As an attacker I can send an "evil" public key s and the parties will compute $s^x \ mod \ p, s^y \ mod \ p$ as their shared keys.

In the case $s = 0$ both parties agree on shared key 0, and in the case $s = 1$ both parties agree on shared key 1, that's the reason those values are checked in the handshake, but are there any other "evil" values?

I thought about $s = p - 1$ as a perfect candidate (where p is the order of the group, in our case $2^{255} - 19$), that is because for all values of i :

$$(p-1)^i \ mod \ p = \left(\sum_{j=1}^{i} p^j + (-1)^i\right) mod \ p = (-1)^i \ mod \ p.$$

The shared key will be $p - 1$ or 1 depending on the parity of the exponent, that is the shared key should be the same for the client and the server in 50% of the cases (when the parity of x is the same as the parity of y) and known to the attacker.

In the case I got the same shared key for both client and server, I can forward their nonces, and then the proofs they calculate are legitimate proofs with the agreed shared key.

Implementation:

```
ss = socket.socket()
cs = socket.socket()
ss.connect(('mitm.ctfcompetition.com', 1337))
cs.connect(('mitm.ctfcompetition.com', 1337))

WriteLine(ss, b's')
WriteLine(cs, b'c')
server_public_key = ReadBin(ss)
server_nonce = ReadBin(ss)
client_public_key = ReadBin(cs)
client_nonce = ReadBin(cs)
my_key = ((2 ** 255) - 19 - 1).to_bytes(255,
'little').rstrip(b'\x00')

WriteBin(ss, my_key)
WriteBin(ss, client_nonce)
WriteBin(cs, my_key)
```

```
WriteBin(cs, server_nonce)
server_proof = ReadBin(ss)
client_proof = ReadBin(cs)

WriteBin(cs, server_proof)
WriteBin(ss, client_proof)

auth_data = ReadBin(ss)
WriteBin(cs, auth_data)

myPrivateKey = Private()
theirPublicKey = Public(my_key)
sharedKey = myPrivateKey.get_shared_key(theirPublicKey)
mySecretBox = nacl.secret.SecretBox(sharedkey)
print(mySecretBox.decrypt(auth_data))

get_flag_msg = mySecretBox.encrypt(b'getflag')
WriteBin(ss, get_flag_msg)
flag_msg = ReadBin(ss)
print(mySecretBox.decrypt(flag_msg))

ss.close()
cs.close()
```

The flag:

CTF{kae3eebav8Ac7Mi0RKgh6eeLisuut9oP}