Google CTF 2018 Quals - chq-matteo @ mHACKeroni

MITM

What can you do when you are stuck?

Overview

In the attachment there is only one python3 script.

In the script there are five interesting functions:

1. Challenge(password, flag, reader, writer)
2. Client(password, reader, writer)

3. Server(password, flag, reader, writer)
4. Handshake(password, reader, writer)
5. ComputeProof(key, data)

There are also several interesting modules

```
from curve25519 import Private, Public
import nacl.secret
import hmac
import hashlib
```

Functions

Challenge(password, flag, reader, writer)

```
def Challenge(password, flag, reader, writer):
  try:
    server_or_client = ReadLine(reader)
    is_server = server_or_client[0] in b'sS'
    is_client = server_or_client[0] in b'cC'

    if is_server:
      return Server(password, flag, reader, writer)
    elif is_client:
      return Client(password, reader, writer)
```

At each connection, we can choose whether we want to talk to a Server or a Client.

A password is passed to the Client and the Server, the flag is passed as an argument only to the Server function.

Client(password, reader, writer)

```
def Client(password, reader, writer):
  sharedKey = Handshake(password, reader, writer)
  if sharedKey is None:
    WriteLine(writer, b'Error: nope.')
    return 1

  mySecretBox = nacl.secret.SecretBox(sharedKey)
  line = mySecretBox.decrypt(ReadBin(reader))
  if line != b"AUTHENTICATED":
    WriteLine(writer, b'Error: nope.')
    return 1

  WriteBin(writer, mySecretBox.encrypt(b"whoami"))
  line = mySecretBox.decrypt(ReadBin(reader))

  if line != b'root':
    return 1

  WriteBin(writer, mySecretBox.encrypt(b"exit"))
  return 0
```

If we choose to communicate with the client, we have to:

1. complete a Handshake and compute a sharedKey
2. Initialize a nacl.secret.SecretBox with the sharedKey
3. send an encrypted 'AUTHENTICATED" message
4. decrypt a 'whoami' message
5. respond an encrypted 'root'
6. terminate the connection

Server(password, flag, reader, writer)

```
def Server(password, flag, reader, writer):
  sharedKey = Handshake(password, reader, writer)
  if sharedKey is None:
    WriteLine(writer, b'Error: nope.')
    return 1

  mySecretBox = nacl.secret.SecretBox(sharedKey)
  WriteBin(writer, mySecretBox.encrypt(b"AUTHENTICATED"))

  while 1:
    cmd = mySecretBox.decrypt(ReadBin(reader))
    if cmd == b'help':
      rsp = b'help|exit|whoami|getflag'
    elif cmd == b'exit':
      return 0
    elif cmd == b'whoami':
      rsp = b'root'
    elif cmd == b'getflag':
      rsp = flag
    else:
      return 1
    WriteBin(writer, mySecretBox.encrypt(rsp))
```

The Server shares a lot of logic with the client. After the Handshake, the Server listens for encrypted commands.

We need to send a 'getflag' command to retrieve the flag.

Handshake(password, reader, writer)

```
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 consists of the following steps:

1. Compute a Private key
2. get a random nonce
3. send the corresponding Public key
4. send the nonce
5. read the other party's public key and nonce
6. **refuse to continue if both nonce are equal or the other party's key is some special value**

7. compute a sharedKey
8. generate an authentication proof
9. send the proof
10. **verify the other party's proof**
11. return the sharedKey on success

ComputeProof(key, data)

```
def ComputeProof(key, data):
  return hmac.new(key, data, digestmod=hashlib.sha256).digest()
```

ComputeProof returns a message authentication code which ensures:

1. data integrity with the hash function
2. authentication thanks to a shared secret key

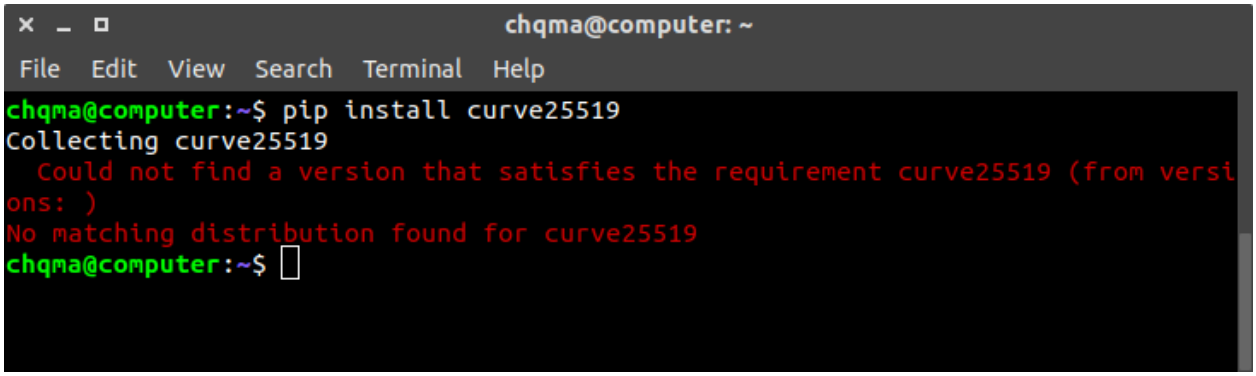
In this program ComputeProof is called with sharedKey as key and nonce + password as data.

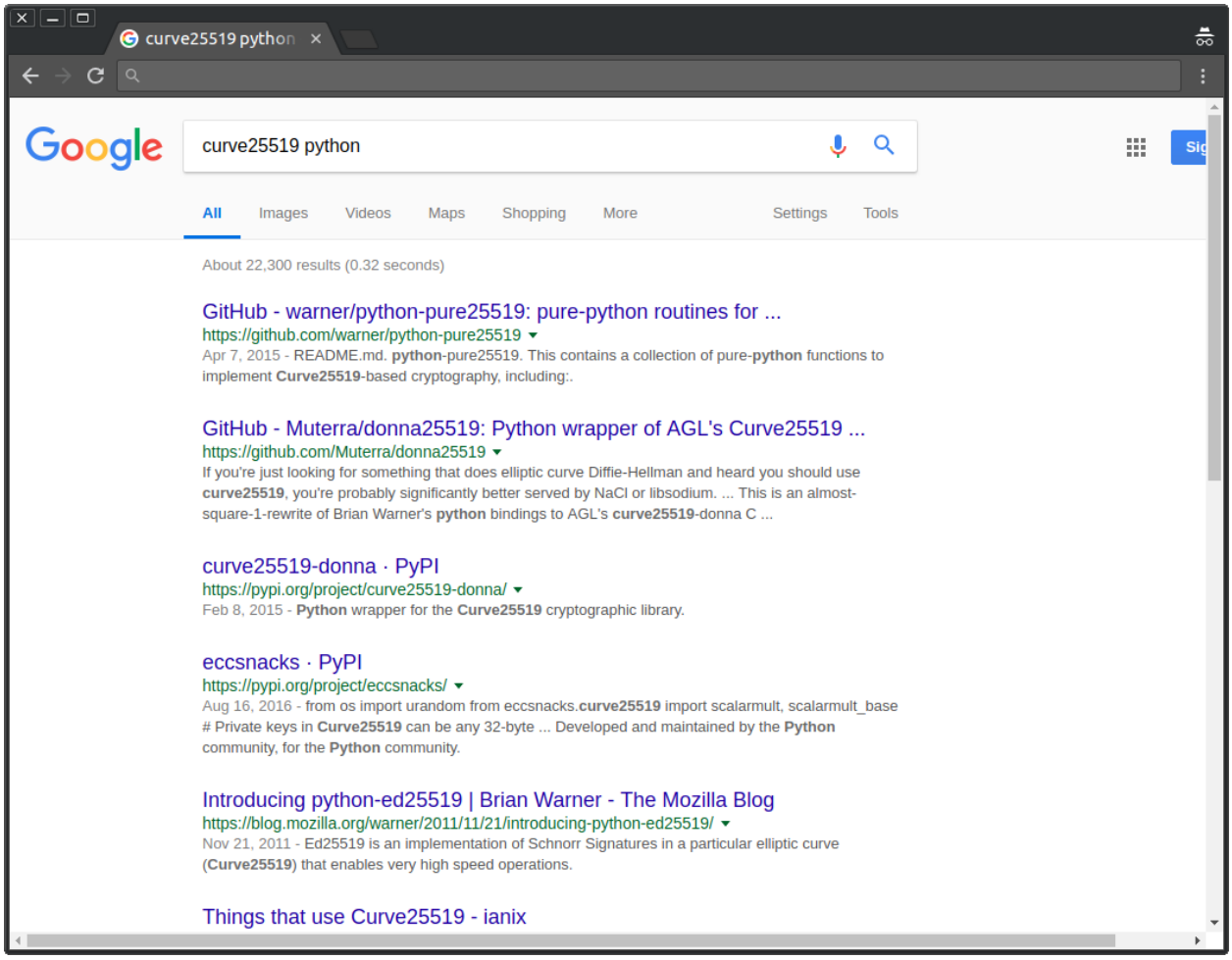
Without knowing the sharedKey or the password we cannot hope to compute a valid proof ourselves.

Modules

curve25519

There isn't any package named curve25519





Seems like the isn't a perfect match on the first results on Google.

nacl

pynacl[1] is a relatively well known library, it is a Python binding to libsodium.

Both nacl and libsodium pop up often in the CTF scene, odds are you already have it installed.

Solution

There are a couple of unanswered questions that we could not address looking at the program code:

1. does Private() generate a new key each time?
2. is Private.get_public() static?
3. what does Private.get_shared_key() do?
4. can we leak the password?
5. can we reuse an authentication proof?

Part 1: Exploration

To address some of the unanswered question we write a simple proxy which will relay messages back and forth for a Server and a Client without tampering with them.

This is what we gather:

- the public key changes every session (Question 2)
- the private key probably changes too (Q. 1)
- we probably cannot leak the password (Q. 4)

Part 2: Learn more about Curve25519

It's time to binge some articles about Curve25519

[1] pynacl github project page, https://github.com/pyca/pynacl

https://cr.yp.to/ecdh.html

https://cr.yp.to/ecdh/curve25519-20051115.pdf

https://cr.yp.to/ecdh/curve25519-20060209.pdf

The primitive for Curve25519 is the function Curve25519(scalar, point.X) -> point1.X where point.X is the X coordinate of the point, it is the multiplication of scalar * point.X on a particular elliptic curve.

A shared secret (Q. 3) can be computed by two parties A and B with the formula

sharedSecret = Curve25519(A.privateKey, B.publicKey) == Curve25519(B.privateKey, A.publicKey).

In our case we have control over both publicKeys

Part 3: Authentication

We need to complete the Handshake to advance with the challenge.

Recall that we can send to each Client or Server:

1. a public key
2. a nonce

Since we don't know the password, we have to get the authentication proof from either a Client or a Server. Let's call the **FlagServer** the Server from which we'll get the flag, **AuthClient** the Client that will compute an authentication proof for us.

To authenticate with FlagServer we need to ComputeProof with its nonce and password, that is **we must send its nonce to AuthClient**.

The publicKeys, on the other hand, have to be chosen so that both AuthClient and FlagServer **will compute the same sharedKey**. Unfortunately we don't know AuthClient and FlagServer privateKey, if we don't tamper with their publicKeys we won't be able to decrypt their conversation later on.

A failure condition in Handshake catches our eyes

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

The publicKey we send cannot be 0x0000..0000 nor 0x0100..0000. 0x0000..0000 is a special value because Curve25519(any scalar, 0x0000..0000) = 0x0000..0000, 0x0100..0000 is not special by any means.

Reading the article [Can we avoid tests for zero in fast elliptic-curve arithmetic?](https://cr.yp.to/ecdh/curvezero-20060726.pdf)[2] we find out that while 0x0100..0000 is just another publickey, 0x0000..0001 behaves like 0x0000..0000, suggesting that the publicKeys are stored in Little Endian byte order[3].

If we manage to set theirPublicKey to 0x0000..0000 or 0x0100..0000 we could predict sharedKey's value.

Luckily Elliptic Curve Cryptography is built on top of Elliptic Curves over Finite Fields, that is we can consider the coordinates modulo a certain prime number[4]. In Curve25519 such prime is $2^{255} - 19$.[5]

If we send littleEndian($2^{255} - 19$) as our publicKey **it will be reduced to 0x0000..0000 during the computation of the sharedSecret**, in the end the sharedSecret will be equal to 0x0000..0000 (similarly for littleEndian($2^{255} - 19 + 1$).

Message flow

1. Connect to AuthClient
2. Connect to FlagServer
3. Receive publicKeys and nonces

[2] Theorem 3.1, $X(2Q) = (X(Q)^2 - 1)^2 / (...) = (1^2 - 1)^2 / (...) = 0$ when $X(Q) = 1$, https://cr.yp.to/ecdh/curvezero-20060726.pdf
[3] 0x0100..0000 little endian is equal to 1 big endian, https://en.wikipedia.org/wiki/Endianness#Little-endian
[4] also prime powers, but usually not
[5] 2^**255** - **19** → **25519** clever right?

4. send littleEndian(2^{255} - 19) to AuthClient and FlagServer
5. send FlagServer.nonce to AuthClient and AuthClient.nonce to FlagServer
6. receive and forward each proof (Q. 5)
7. get encrypted 'AUTHENTICATED' message from FlagServer

Part 4: Decryption

We can now authenticate with FlagServer and complete the Handshake, the only thing left to do is initialize a nacl.secret.SecretBox with the sharedKey

```
mySecretBox = nacl.secret.SecretBox(sharedKey)
```

Let's try with sharedKey = b'\x00'*32 since that is the computed shared secret in Curve25519.

Well, no luck there.

Maybe Private.get_shared_key() makes use of a Key derivation function, usually a shared secret is hashed. Since sharedKey is a 32 byte (256 bits) string, let's try some hashing algorithm with 256 bit output.

sha256? No

blake2b (used in nacl documentation examples)? No

Complain on irc? No hints for the main challenges

Looks like we are stuck.

Part 5: Going deeper

There is one natural direction you can go when you are lost, it's not up because we cannot fly, it's down, so let's start digging.

The organizers didn't feel like including Private.get_shared_key() python module, that means that it has to be some kind of standard.

1. visit every single result on Google and Github up to the third page and beyond
2. search for "curve25519 key derivation function"
3. read the source code for libsodium
4. test whatever crosses your mind

Salsa20 core (found in some obscure paper with a ton of magic numbers)? No

ChaCha20 core (found in the source code of libsodium)? No

Salsa20 core (found in the source code of libsodium with different magic numbers)? No

Hmac with whatever parameters we can find? No

No solution yet, but we discover some pretty nice articles

- Why not validate Curve25519 public keys could be harmful https://vnhacker.blogspot.com/2015/09/why-not-validating-curve25519-public.html
- https://vnhacker.blogspot.com/2016/08/the-internet-of-broken-protocols.html

After 5 hours we can conclude that this is not the right approach.

Something is missing

Part 6: Backtracking

After trying every single specification we can find, we can conclude that curve25519 is a standard library, **there is no way we can solve this challenge other than finding its source code**.

Let's see if there is any line of code peculiar enough that could lead us to the source code

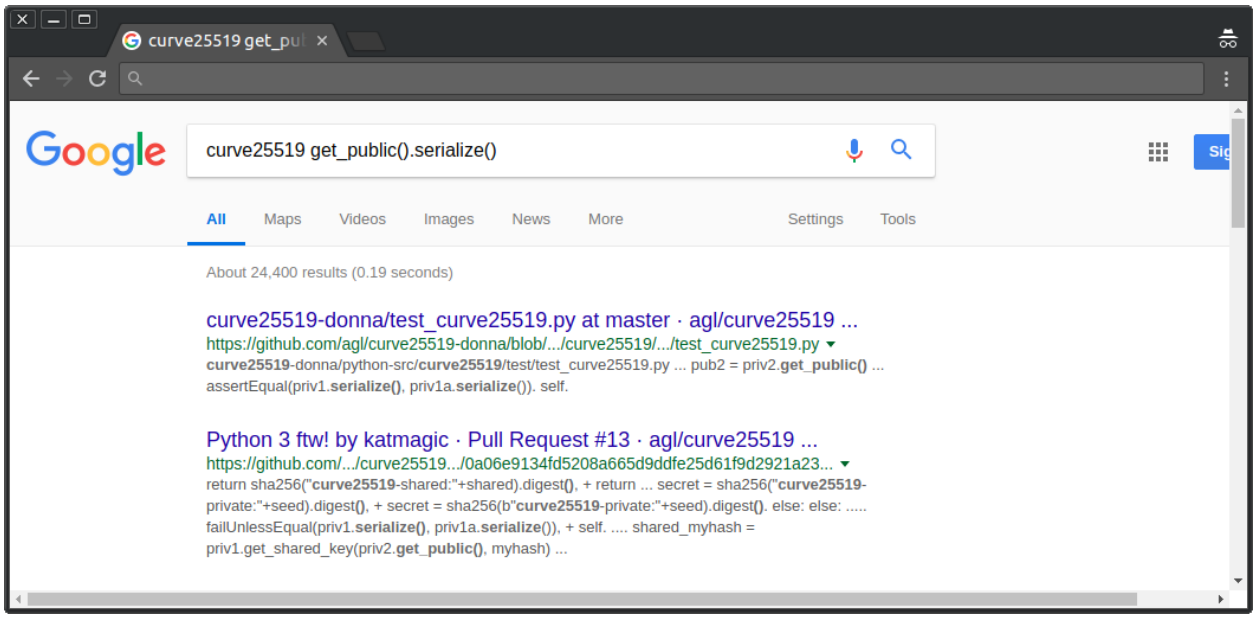
```
from curve25519 import Private, Public
Private()
WriteBin(writer, myPrivateKey.get_public().serialize())
theirPublicKey = Public(theirPublicKey)
sharedKey = myPrivateKey.get_shared_key(theirPublicKey)
```

There are five unique lines that involve curve25519 module

- Private() and Public() are too generic
- get_shared_key() is also too generic

get_public().serialize() is pretty unusual since its output is just a 32 byte string.

If we are lucky enough the same code should be in the module tests or documentation



Wow, first result and we even found that during the Overview. Tunnel vision sure is bad for your health.

In the end the key derivation function was super arbitrary in true cryptography fashion[6]

```
def _hash_shared(shared):
    return sha256(b"curve25519-shared:"+shared).digest()
```

We have it all now.

Part 7: getflag

In the end:

1. Connect to AuthClient
2. Connect to FlagServer
3. Receive publicKeys and nonces
4. send littleEndian(2^{255} - 19) to AuthClient and FlagServer
5. send FlagServer.nonce to AuthClient and AuthClient.nonce to FlagServer
6. receive and forward each proof (Q. 5)
7. get encrypted 'AUTHENTICATED' message from FlagServer
8. compute sharedKey = _hash_shared(b'\x00' * 32)
9. initialize the nacl.secret.SecretBox with sharedKey
10. send encrypted b'getflag'
11. receive and decrypt the flag
12. CTF{kae3eebav8Ac7Mi0RKgh6eeLisuut9oP}
13. submit and go to sleep

[6] https://github.com/agl/curve25519-donna/blob/master/python-src/curve25519/keys.py#L9

Appendix

Final solution script

```
#!/usr/bin/env python3
from binascii import hexlify
from binascii import unhexlify
import logging
import sys
import os
import nacl.secret
import hmac
import hashlib
import pwn
pwn.context.log_level = logging.DEBUG
from hashlib import sha256, sha512

def ReadLine(reader):
 data = b''
 while not data.endswith(b'\n'):
  cur = reader.recv(1)
  data += cur
  if cur == b'':
   return data
 return data[:-1]

def WriteLine(writer, msg):
 writer.send(msg + b'\n')

def ReadBin(reader):
 return unhexlify(ReadLine(reader))

def WriteBin(writer, data):
 WriteLine(writer, hexlify(data))

def main():
 pk =
long_to_bytes(57896044618658097711785492504343953926634992332820282019728792003956564819950
).rjust(32, '\0')[::-1]
 pk1 =
long_to_bytes(57896044618658097711785492504343953926634992332820282019728792003956564819949
).rjust(32, '\0')[::-1]
 with pwn.remote('mitm.ctfcompetition.com', 1337) as c:
  c.sendline('c')
  ckey = ReadBin(c)
  cnonce = ReadBin(c)
  WriteBin(c, pk1)
  with pwn.remote('mitm.ctfcompetition.com', 1337) as s:
   s.sendline('s')
```

```
    skey = ReadBin(s)
    snonce = ReadBin(s)
    WriteBin(s, pk)
    WriteBin(s, cnonce)
    WriteBin(c, snonce)
    cauth = ReadBin(c)
    # print('Recvd client verification')
    WriteBin(s, cauth)
    sauth = ReadBin(s)
    # print('Recvd server verification')
    WriteBin(c, sauth)
    authed = ReadBin(s)
    WriteBin(c, authed)
    WriteBin(s, ReadBin(c))
    WriteBin(c, ReadBin(s))
    k = ReadBin(c)
    mySecretBox = nacl.secret.SecretBox(sha256(b"curve25519-shared:"+'\0'*32).digest())
    WriteBin(s, mySecretBox.encrypt(b'getflag'))
    flag = ReadBin(s)
    print(mySecretBox.decrypt(flag))

if __name__ == '__main__':
 sys.exit(main())
```