interlock Writeup (justCTF teaser 2024)

> by FluxFingers

The challenge scenario is man-in-the-middle of a cryptographic protocol. Both sides implement a state machine with timing based transitions:

1. Alice commits her nonce and her public key inside of a hash
2. Alice starts waiting for 4 seconds
3. Bob receives alice's commitment
4. Bob starts waiting for 4 seconds
5. Alice is done waiting and sends her public key and nonce
6. Alice then starts waiting 4 seconds for a message including bob's public key (if she doesn't receive the message, the program exits and the attacker loses)
7. Bob is done waiting and receives alice's data and sends back his nonce (but encrypted)

The classic MITM-approach would be to pick a new nonce and create a new commitment with our own public key which would then be sent to bob. However, this doesn't work as the challenge verifies that the nonces are equal.

Alice verifies her 4 second timeout using a separate binary, called "timer". Reversing the binary shows us that it uses C++'s std::chrono library which respects leap seconds. The timer also always initialized right before a leap second.

The solution is then to wait until right before the leap second, start the protocol with alice, ignore her commitment, wait for her real data, swap her public key with our own, do the entire protocol with Bob (which will take 4 point something seconds), and resume the protocol with alice. Alice will compare the two timestamps from when she started waiting to when we resumed the protocol with her. Since pythons library doesnt respect leap seconds, it will appear as though only 3 point something seconds have passed, which is okay. This way, alice doesn't exit and the nonces are known. We can then show the nonces to the server as proof that we succeeded and get the flag.

Here's our solve script:

```
#!/usr/bin/env python

from pwn import *
from datetime import datetime

from time import sleep

import json

import hpke

from binascii import hexlify, unhexlify

from cryptography.hazmat.primitives.serialization import PublicFormat, Encoding

from cryptography.hazmat.primitives.asymmetric import ec

from cryptography.hazmat.primitives import hashes


suite = hpke.Suite__DHKEM_P256_HKDF_SHA256__HKDF_SHA256__ChaCha20Poly1305

ske = suite.KEM.generate_private_key()
```

```python
pke = ske.public_key().public_bytes(
    encoding=Encoding.X962, format=PublicFormat.UncompressedPoint
)

def fmt(data):
    return hexlify(data).decode()


def ufmt(data):
    return unhexlify(data.encode())

def send(conn, t, msg):
    conn.sendline(json.dumps({"type": "write", "target": t, "msg": msg}).encode())


def send_alice(conn, msg):
    send(conn, "alice", msg)


def send_bob(conn, msg):
    send(conn, "bob", msg)


def recv(conn, t):
    conn.sendline(json.dumps({"type": "read", "target": t}).encode())
    msg = conn.recvline(keepends=False)
    if msg == b"none":
        return None
    return msg


def recv_blocking(conn, t):
    msg = None
    while msg is None:
        msg = recv(conn, t)
    return msg


def recv_alice(conn):
    return recv_blocking(conn, "alice")


def recv_bob(conn):
    return recv_blocking(conn, "bob")


def main():
    if args.REMOTE:
        conn = remote('interlock.nc.jctf.pro', 7331)
    # conn = remote("localhost", 7331)


    welcome = conn.recvline(keepends=False).decode()
    print(welcome)
```

```
    t = welcome[11:-29]
    t = datetime.strptime(t, "%Y-%m-%d %H:%M:%S.%f")


    before = datetime.fromtimestamp(662687995)


    # t = welcome[]


    send_bob(conn, "start")

    sleep((before - t).total_seconds())

    send_alice(conn, "start")


    recv(conn, "alice")

    print("c1")

    # tweak
    sleep(4)

    m1_sig = json.loads(recv(conn, "alice").decode())

    m1a = m1_sig["m1"]
    m1 = json.loads(m1a)

    print("m1_sig")

    pka = ufmt(m1["pka"])

    m1["pka"] = fmt(pke)

    x1 = m1["x1"]

    m1 = json.dumps(m1)

    c1_d = hashes.Hash(hashes.SHA3_256())
    c1_d.update(m1.encode())
    c1 = c1_d.finalize()

    send_bob(conn, fmt(c1))

    m1_sig["s1"] = fmt(ske.sign(m1.encode(), ec.ECDSA(hashes.SHA3_256())))
    m1_sig["m1"] = m1
    m1_sig = json.dumps(m1_sig)

    print("sending m1_sig to bob: ", m1_sig)
    send_bob(conn, m1_sig)

    # tweak
```

```python
    sleep(3)

    m2_enc = json.loads(recv(conn, "bob").decode())
    print("received m2_enc: ", m2_enc)

    encap, ct, pkb = ufmt(m2_enc["encap"]), ufmt(m2_enc["ct"]), ufmt(m2_enc["pkb"])
    pkb_k = ec.EllipticCurvePublicKey.from_encoded_point(suite.KEM.CURVE, pkb)

    m2 = suite.open_auth(
        encap,
        ske,
        pkb_k,
        info=b"interlock",
        aad=pkb,
        ciphertext=ct,
    )
    m2 = json.loads(m2)

    x2 = m2["x2"]

    m2["pka"] = fmt(pka)
    m2["m1"] = m1a

    m2 = json.dumps(m2)

    pka_k = ec.EllipticCurvePublicKey.from_encoded_point(suite.KEM.CURVE, pka)

    encap, ct = suite.seal_auth(
        pka_k, ske, info=b"interlock", aad=pke, message=m2.encode()
    )

    m2_enc = json.dumps({"encap": fmt(encap), "ct": fmt(ct), "pkb": fmt(pke)})

    send_alice(conn, m2_enc)
    print("sending m2_enc to alice")

    conn.sendline(json.dumps({"type": "quit"}).encode())

    print(conn.recvline_startswith(b"Communication"))

    conn.recvuntil(b"Give me x1: ")
    conn.sendline(x1)

    conn.recvuntil(b"Give me x2: ")
    conn.sendline(x2)

    err = conn.recvline().strip()
    print(err)
    conn.close()
    assert err == b"NOPE", err


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