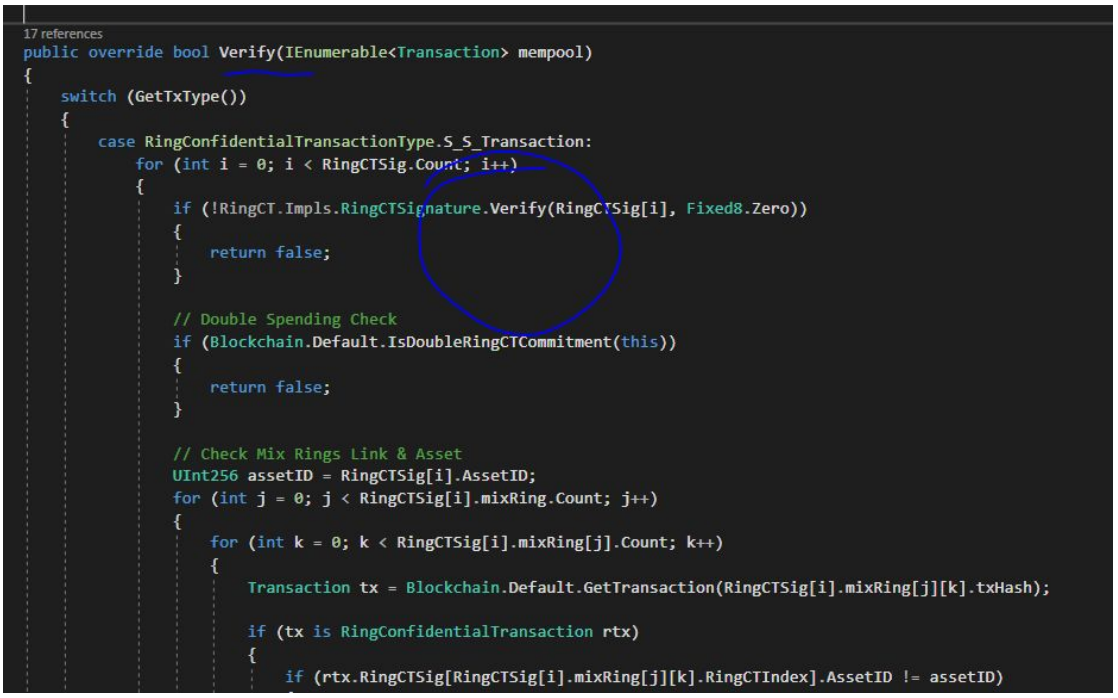
Update_1.0

RingCT

* (This update is regarding RingCT implementation itself, minor code issues won't be discussed in progress updates, we can list them as suggestions in final report)

Priority : Critical

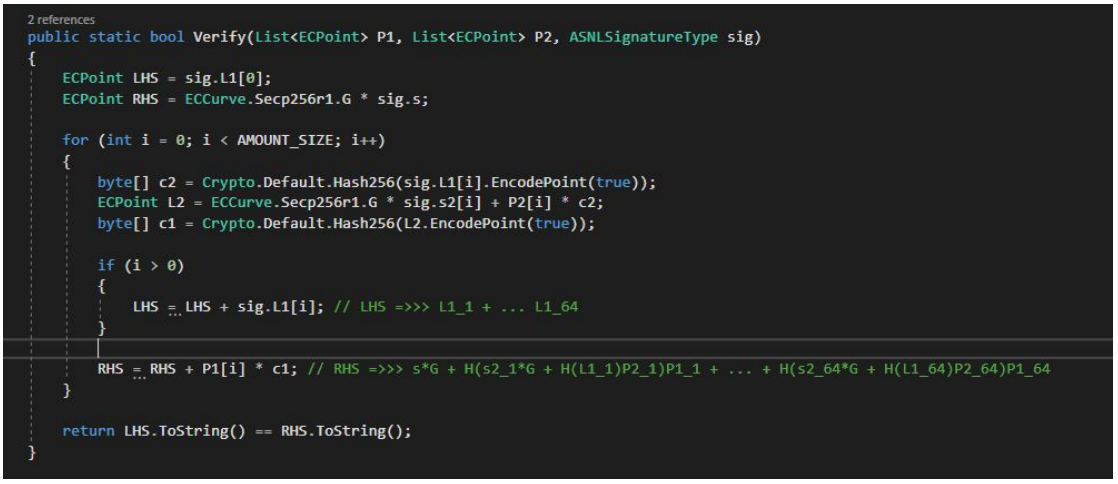
Issue : It is possible to pick a L1[i] in RingCt verify in ASNL ring signature such that it cancels out the sum on RHS where discrete logarithms of P1 & P2 are unknown. It can lead to an attacker forge proof of ASNL ring signature and create wrong transactions. (eg: Creating and destroying QURAS)



```
17 references
public override bool Verify(IEnumerable<Transaction> mempool)
{
    switch (GetTxType())
    {
        case RingConfidentialTransactionType.S_S_Transaction:
            for (int i = 0; i < RingCTSig.Count; i++)
            {
                if (!RingCT.Impls.RingCTSignature.Verify(RingCTSig[i], Fixed8.Zero))
                {
                    return false;
                }

                // Double Spending Check
                if (Blockchain.Default.IsDoubleRingCTCommitment(this))
                {
                    return false;
                }

                // Check Mix Rings Link & Asset
                UInt256 assetID = RingCTSig[i].AssetID;
                for (int j = 0; j < RingCTSig[i].mixRing.Count; j++)
                {
                    for (int k = 0; k < RingCTSig[i].mixRing[j].Count; k++)
                    {
                        Transaction tx = Blockchain.Default.GetTransaction(RingCTSig[i].mixRing[j][k].txHash);

                        if (tx is RingConfidentialTransaction rtx)
                        {
                            if (rtx.RingCTSig[RingCTSig[i].mixRing[j][k].RingCTIndex].AssetID != assetID)
```

The verify function in RingConfidentialTransaction calls Impls Ring verify function and finally calls ASNLRingSignature Verify function.

Now lets see the verify function in ASNL a little more in depth.



```
2 references
public static bool Verify(List<ECPoint> P1, List<ECPoint> P2, ASNLSignatureType sig)
{
    ECPoint LHS = sig.L1[0];
    ECPoint RHS = ECCurve.Secp256r1.G * sig.s;

    for (int i = 0; i < AMOUNT_SIZE; i++)
    {
        byte[] c2 = Crypto.Default.Hash256(sig.L1[i].EncodePoint(true));
        ECPoint L2 = ECCurve.Secp256r1.G * sig.s2[i] + P2[i] * c2;
        byte[] c1 = Crypto.Default.Hash256(L2.EncodePoint(true));

        if (i > 0)
        {
            LHS = LHS + sig.L1[i]; // LHS =>>> L1_1 + ... L1_64
        }

        RHS = RHS + P1[i] * c1; // RHS =>>> s*G + H(s2_1*G + H(L1_1)P2_1)P1_1 + ... + H(s2_64*G + H(L1_64)P2_64)P1_64
    }

    return LHS.ToString() == RHS.ToString();
}
```

The verification is done by ASNL range proof which works roughly as follows :

ASNL signature contains (P1_j, P2_j, L1_j, s2_j) for j = [1 to n] and s which is supposed to prove that the signer knows the discrete log of (P1_1 OR P2_1) & so on.

So LHS becomes :

LHS =>>> L1_1 + ... L1_64 and RHS becomes
RHS =>>> s*G + H(s2_1*G + H(L1_1)P2_1)P1_1 + ... + H(s2_64*G + H(L1_64)P2_64)P1_64

Where H is the hashing function used (in this case Secp256r1)
Now this is buggy as attackers can pick a L1[i] such that it cancels out the sum on RHS where discrete logarithms of P1 & P2 are unknown.

This leads to serious issues, basically breaking any transaction using RingCT.

A very similar issue was existing in Monero and they shifted away from ASNL to Booromean range proof.

Relevant discussion here from that project.
https://github.com/monero-project/research-lab/issues/4

Faulty Assumption used from RingCT paper : [Theorem 13]