aedee8b95b4d88de160e94ebf525772969f29a0f93fbeb1f5bf9027809f5b844

File: AuraToken.sol | Language:solidity | Size:9053 bytes | Date:2022-06-15T16:10:59.189Z

Critical	High	Medium	Low	Note
0	0	1	0	3

Issues

Severity	Issue	Analyzer	Code Lines
Medium	SWC-102	Achilles	1
Note	SWC-116	Achilles	156, 229
Note	SWC-118	Achilles	9 - 251

Code

1. SWC-102 / lines: 1 Medium Achilles

```
A security vulnerability has been detected.

1   pragma solidity 0.6.12;
2
```

In detail

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

2. SWC-116 / lines: 156 Note Achilles

```
A security vulnerability has been detected.

155     {
156         require(blockNumber < block.number, "AURA::getPriorVotes: not yet determined");
157
```

In detail

Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

3. SWC-116 / lines: 229 Note Achilles

```
A security vulnerability has been detected.

228     {
229         uint32 blockNumber = safe32(block.number, "AURA::_writeCheckpoint: block number exceeds 32 bits");
230
```

In detail

Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into

consideration, developers can't rely on the preciseness of the provided timestamp.

4. SWC-118 / lines: 9 - 251 Note Achilles

A security vulnerability has been detected.

```
8   // AuraToken with Governance.
9   contract AuraToken is ERC20("AuraSwap Token", "AURA"), Ownable {
10
11      /// @notice Total number of tokens
12      uint256 public constant maxSupply = 100_000_000e18; // 100_000_000 Aura
13
14      /// @notice Creates `_amount` token to `_to`. Must only be called by the owner (MasterChef).
15      function mint(address _to, uint256 _amount) public onlyOwner {
16          if(totalSupply().add(_amount) <= maxSupply){
17              _mint(_to, _amount);
18              _moveDelegates(address(0), _delegates[_to], _amount);
19          }
20      }
21
22      // Copied and modified from YAM code:
23      // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernanceStorage.sol
24      // https://github.com/yam-finance/yam-protocol/blob/master/contracts/token/YAMGovernance.sol
25      // Which is copied and modified from COMPOUND:
26      // https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/Comp.sol
27
28      /// @notice A record of each accounts delegate
29      mapping (address => address) internal _delegates;
30
31      /// @notice A checkpoint for marking number of votes from a given block
32      struct Checkpoint {
33          uint32 fromBlock;
34          uint256 votes;
35      }
36
37      /// @notice A record of votes checkpoints for each account, by index
38      mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;
39
40      /// @notice The number of checkpoints for each account
41      mapping (address => uint32) public numCheckpoints;
42
43      /// @notice The EIP-712 typehash for the contract's domain
44      bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,address verifyingContract)");
45
46      /// @notice The EIP-712 typehash for the delegation struct used by the contract
47      bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 nonce,uint256 expiry)");
48
49      /// @notice A record of states for signing / validating signatures
50      mapping (address => uint) public nonces;
51
52        /// @notice An event thats emitted when an account changes its delegate
53      event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);
54
55      /// @notice An event thats emitted when a delegate account's vote balance changes
56      event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);
57
58      /**
59       * @notice Delegate votes from `msg.sender` to `delegatee`
60       * @param delegator The address to get delegatee for
61       */
62      function delegates(address delegator)
63          external
64          view
65          returns (address)
66      {
67          return _delegates[delegator];
68      }
69
70     /**
71      * @notice Delegate votes from `msg.sender` to `delegatee`
72      * @param delegatee The address to delegate votes to
73      */
74      function delegate(address delegatee) external {
75          return _delegate(msg.sender, delegatee);
```

```
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "AURA::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "AURA::delegateBySig: invalid nonce");
        require(now <= expiry, "AURA::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account)
        external
        view
        returns (uint256)
    {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber)
```

```
        external
        view
        returns (uint256)
    {
        require(blockNumber < block.number, "AURA::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee)
        internal
    {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying AURAs (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(address srcRep, address dstRep, uint256 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(
        address delegatee,
        uint32 nCheckpoints,
        uint256 oldVotes,
        uint256 newVotes
    )
        internal
    {
```

```
        {
            uint32 blockNumber = safe32(block.number, "AURA::_writeCheckpoint: block number exceeds 32 bits");

            if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
                checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
            } else {
                checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
                numCheckpoints[delegatee] = nCheckpoints + 1;
            }

            emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
        }

        function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
            require(n < 2**32, errorMessage);
            return uint32(n);
        }

        function getChainId() internal pure returns (uint) {
            uint256 chainId;
            assembly { chainId := chainid() }
            return chainId;
        }
    }
```

In detail

Constructors are special functions that are called only once during the contract creation. They often perform critical, privileged actions such as setting the owner of the contract. Before Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract class containing it. A function meant to become a constructor becomes a normal, callable function if its name doesn't exactly match the contract name. This behavior sometimes leads to security issues, in particular when smart contract code is re-used with a different name but the name of the constructor function is not changed accordingly.