Safe Edges

Audit Report March, 2025

Audit Issue

[M-01] Incorrect totalPieces calcualtion in CliqueLock:calculateClaimableAmount can leads to unevenly vesting amount distribution

Description The current calculates the totalPieces like this:

```
uint256 totalPieces = vestingDuration / stream.pieceDuration + 1;
```

This works completely fine when the `vestingDuration % pieceDuration != 0`. Like in scenario where let's say vestingDuration is 13 months and pieceDuration is 3 months, then this would return totalPieces as 5. However when vestingDuration is 12 months and pieceDuration is 3months, then again this would create 5 totalPieces. Now the problem with such piece calculation is that it can create unevenly vesting amount, like in our case, it would allow to redeem 60% of vested amount for first 3 pieceDuration, and in the 4th piece (which is from 9 months -12 months), it will allow rest 40%. The way it is allowing the largest chunk in last piece, is because after 12 months, this condition will satisfy which will return the whole amount

```
        if (block.timestamp >= stream.endTime) {
            return stream.amount;
        }
```

Impact This leads to vestedAmount to be distributed unevenly potentially forcing the last piece getting the larger chunk than other pieces.

Recommended mitigation You can first check using modulus whether `1` should be added to totalPieces or not.

[M-02] Anyone can initiate claim of any stream by using just their `streamId` in `CliqueLock:claim` function

Description Claim takes only streamId as arguement, and then fetch the recipient of the exact claimable amount by calling `_resolveRecipient` function. However, it does not ensure that the caller is the rightful reciever. This create a scenario where one malicious user can initiate the claim of other streams by using just their streamId even if the receiver did not intended to do so.

Impact Maliciuos user can initiate the claim of other streams by using their streamId.

Recommended mitigation Consider implementing this check which will allows only the receiver to initiate the claim:

```
    function claim(uint256 streamId) external {
        Stream memory stream = streams[streamId];
```

```
        address recipient = _resolveRecipient(streamId, stream);
+        if (recipient != msg.sender) {
+            revert InvalidRecipient();
+        }
        if (stream.amount == 0) {
            revert StreamNotFound(streamId);
        }
        ...}
```

[M-03] `CliqueDistributorManager:createDistributor` should expilicit check for signature non-malleability

Description createDistributor function calls ECDSA.recover function to recover signer based on the hash, and signature. However, solady's ECDSA does not implement signature uniqueness check like Openzeppelin's ECDSA, which makes sure that the s field in the signature should be in the lower half order. Since during recovering s can potentially have two values, implementing such check prevent such scnearios.

Impact this issue can potentially create malleable signature

Recommended mitigation Consider either using Openzeppelin ECDSA library. Otherwise, integerating this check into the contract will ensure that signatures cannot be manipulated to create another valid signature from the original one.

```
    if (uint256(s) >
0x7fffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0) {
        return false;
    }
```

[L-01] Excessive sent ether would not be refunded to the user in `Distributor:claim` function

Description For being able to claim the airdrop tokens, the user must first sent the fee. Now the current logic only checks whether the amount is greater than fee, even if the amount is equals to the fee the call would get reverted. Moreover, the excess ether would not be refunded to the user. Only the owner can withdraw funds from this contracts. This create a interesting scenario, where the user must have to pay more than the fee in order to recieve the airdrop token, knowningly that their excess would be stuck there atleast for them.

```
function claim(
        bytes32[] calldata _proof,
        bytes calldata _signature,
        uint256 _amount,
        uint256 _index,
        address _onBehalfOf
    ) external payable whenActive {
```

```
@>>         if (msg.value < fee) {
             revert InsufficientFee();
        }
```

Impact Excess ether would not be refunded to the user leading to user's loss

Recommended mitigation Consider implementing these changes:

```
+  error IncorrectFee();

    function claim(
        bytes32[] calldata _proof,
        bytes calldata _signature,
        uint256 _amount,
        uint256 _index,
        address _onBehalfOf
        ) external payable whenActive {
+            if (msg.value != fee) {
+            revert IncorrectFee();
        }
```

[L-02] Missing fee validation checks in Distributor:setFee function

Description setFee can only be called by the deployer. However, it misses critical checks which puts restriction on the fee limit.

Recommended mitigation Consider adding minFee and maxFee restiction limit in the setFee function.

[L-03] Lack of multiple validations in `CliqueLock:_createStream` before creating stream leads to spam creations.

Description The `_createStream` function, creates a stream using the caller feed stream struct. However, during the creation lacks multiple validation checks related to the amount (>0) and current timestamp less than startTime, cliffTime and endTime. Lacks of such check can leads to creation of stream that just spam the system using its resources and does not actually be useful.

Recommended mitigation consider adding a the validation check in the function:

```
+   error InvalidAmount();

    function _createStream(StreamConfig calldata config) internal returns
(uint256) {
        uint256 streamId = nextStreamId++;

+        if (config.amount == 0) {
+            revert InvalidAmount();
```

```
+        }

+        if (config.startTime > config.cliffTime || block.timestamp >
config.startTime || config.cliffTime > config.endTime) {
            revert InvalidSchedule();
        }
```

[I-01] Twice imports of ECDSA library in CliqueDistributorManager.sol

Description Makes the code more redundant

Recommended mitigation Remove one imports out of two:

```
    import "solady/utils/MerkleProofLib.sol";
    import "solady/utils/ECDSA.sol";
    import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
    import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
    import "solady/utils/FixedPointMathLib.sol";
    import {UUPSUpgradeable} from "solady/utils/UUPSUpgradeable.sol";
    import {Initializable} from "solady/utils/Initializable.sol";
    import {OwnableRoles} from "solady/auth/OwnableRoles.sol";
    import {EIP712} from "solady/utils/EIP712.sol";
    import {Distributor} from "./sub/Distributor.sol";
-    import {ECDSA} from "solady/utils/ECDSA.sol";
```

[I-02] Missing check for signature usage in CliqueDistributorManager:createDistributor can leads to same signature being used multiple times

Description The current logic uses deadline as replay protection. However, it does not revert the call if the same signature is used multiple times to create Distributor within the signed deadline. This would create spam distributor creation. This can be avoided if you use nonce that keep updated once signature being used, making it more secure against replay attack scnearios.

```
bytes32 digest = _hashTypedData(
            keccak256(
                abi.encode(
                    keccak256(
@>                      "NewDistributor(address token, address vault, address
signer, uint256 fee, uint256 deadline)"
                    ),
                    _token,
                    _vault,
                    _signer,
                    _fee,
                    _deadline
```

```
                )
            )
        );
```

Impact User can replay a signature multiple times withing the signed deadline

Recommended mitigation Consider adding nonce in the signature hash.

[I-03] Consider using SafeERC20 for IERC20 to make the transfer more secure.

Description It is recommended to use OpenZeppelin's SafeERC20 library.

```
+   import {SafeERC20} from "openzeppelin-
contracts/contracts/token/ERC20/utils/SafeERC20.sol";

+   using SafeERC20 for IERC20;
```

[I-04] Missing checks for `address(0)` when assigning values to address state variables

Description Check for `address(0)` when assigning values to address state variables.

- Found in Distributor.sol

```
function setSigner(address _signer) external onlyRoles(PROJECT_ADMIN) {}
```

- Found in Distributor.sol

```
        vault = _vault;
```

- Found in CliqueDistributorManager.sol

```
function createDistributor(
    address _token,
    address _vault,
    address _signer,
    uint256 _fee,
    uint256 _deadline,
    bytes calldata _signature
) external returns (address) {}
```

[I-05] Remove Unused Custom Error to save more gas

Description it is recommended to remove the custom error when it remain unused

- Found in CliqueDistributorManager.sol

```
    error InvalidSignature();
```

- Found in Distributor.sol

```
    error InvalidMerkleRootIndex();
```

Audit Report
March, 2025

https://safeedges.in
info@safeedges.in

Safe Edges