[C] - Precision loss allows attacker to take out loans without paying fees

Vulnerability Details

When a user withdraws from the vault the `withDrawn()` function will call `_chargeFees()` in the manager contract however it is possible for a malicious user to bypass paying the fees to the protocol when they withdraw if they pass a small enough value as the `_amount` parameter.

Consider the following code block.

```
    function _chargeFee(
        address _paymentToken,
        uint256 _amount
    ) internal returns (uint256 netAmount) {
        uint256 fee = config.fee();
        if (fee == 0) return _amount;
        //@audit - small token amounts cannot be withdrawn
        uint256 feeAmount = (_amount * config.fee()) / 1e4;
        netAmount = _amount - feeAmount;
        ISuperToken(_paymentToken).transfer(config.feeCollector(),
feeAmount);

        return netAmount;
    }
```

You can see when the fee amount is calculated it is divided by `10_000` in order to scale the BPS.

This causes us a problem because if the protocol chooses to go with basis points of 300 as stated by the devs. The minimum a person can withdraw so that the fee amount doesn't round down to zero is 34. If an attacker passes 33 as `_amount` precision loss would cause `feeAmount` to round down. Consider the following.

```
// 33 * 300 / 10000 = 9900 / 10000 = 0
```

Due to rounding the attacker would be able to withdraw all the tokens from the vault without paying a fee.

Remediation Steps

There are a few different ways to remediate this issue. You could easily enforce a minimum withdrawal amount. This way the attacker can never pass an amount that would lead to rounding errors.

Alternatively you can revert if `feeAmount` equals zero and give the user a useful error message that helps them know they need to withdraw more. Below is a code example showing this.

```
uint256 feeAmount = (_amount * config.fee()) / 1e4;
if(feeAmount == 0) revert WithdrawalTooLow();
```

[C] - Lender can drain borrowers account

Imagine this scenario, Bob is a creator and sells his services on the marketplace. He has paying customers which gives him a FLOW token worth $1000 in future streams. He then mints debt of $500 against this stream and sells it on a marketplace. This creates a debtNFT and sends it to the lender. The lender can now withdraw debt they're owed using the `claimDebt()` function and once they debt is paid off their debt NFT gets burnt.

This is where the problem is. The function is currently only checking if the amount that is being withdrawn is equal to the `debt.amount`, if it is the debtNFT is then burned since the lender would have claimed all they're owed and the loan is closed.

The problem is that the lenders previous withdrawals aren't being stored. A malicious lender can just continue to submit amounts that are less than the `debt.amount` bypassing the burning of the NFT. They can continue to do this until the vault is drained. Let's look at the following step by step of this attack.

1. Bob has 1000 tokens in his vault.
2. Bob has 500 in debt with a lender.
3. Bob calls withdrawn with `amount = 250`.
4. `amount` != `debt.amount` (250 != 500).
5. Bob keeps the NFT.
6. Bob then repeats this process:
    - Calling withdrawn with `amount = 250`.
    - `amount` != `debt.amount` (250 != 500).
    - Bob keeps the NFT.
7. Bob repeats the process to drain the vault.

```
        if (_amount == debt.amount) {
            NFTActions.transferAndBurn(
                debtNFTAddress,
                msg.sender,
                debt.tokenId
            );
        }
```

Remediation Steps

In order to fix this you should store the amount for every withdrawal request that a user submits. You can then check to see if the total amount withdrawn is greater than or equal to the debt owed. If it is then you burn the NFT.

```
        uint amountOut = debtWithdrawn[msg.sender] += _amount;
        if (amountOut > debt.amount) revert TooMuchWithdrawn();
        if (amountOut = debt.amount) {
            NFTActions.transferAndBurn(
                debtNFTAddress,
                msg.sender,
                debt.tokenId
            );
        }
```

[C] - Borrower can fully withdraw from vault even if they currently have debt

```
        checkSubscriptionActive(_subscription);
        //@audit - I should not be able to withdraw if theres a debt
        if (currentDebt.amount > _amount) {
            revert Errors.DebtPending();
        }
```

The following check in the `withdrawn` function only requires that the `_amount` a borrower is withdrawing from the vault is greater than the `currentDebt.amount`. Theoretically a borrower can withdraw from the vault multiple times and drain the vault so long as `_amount > currentDebt.amount`. This would allow the borrower to withdraw all the Supertokens in a vault while still having debt to be paid to the lender.

A malicious account could create multiple subscriptions and take out loans against all of them allowing them to effectively steal all of the lenders funds.

Remediation

The `withdrawn` function should keep track of every withdrawal that is made in a state variable. This value should then be compared to `balanceOf(vault) - currentDebt.amount` to make sure that the amount in the vault is always enough to cover the debt and all the excess is withdrawn. Guaranteeing that the owner can only withdraw the excess amount to their debt.

```
        mapping(address => uint) withdrawals;

        uint amountOut = withdrawals[msg.sender] += _amount;


    ISupertoken(subscription.paymentToken).balanceOf(subscription.vault);

        if (balance - currentDebt.amount > amountOut) {
            revert Errors.DebtPending();
        }
```

[H] - User can pass malicious contract for hub and take advantage of a vault

An attacker can front-run `initialize` vault and pass a malicious contract as the hub. The contract can have a function called `getManagerVault()` so that it will pass the modifier check when `onlyManger()` is called. This allows for a attacker to easily become the manager where they can now withdraw tokens from the vault.

The likelihood of this attack is low however the impact is high as all the tokens in the vault can be drained.

Consider the following code snippet.

```
    function initialize(
        ISuperToken _paymentToken,
        int96 _flowRate,
        address _hub
    ) external initializer {
        paymentToken = _paymentToken;
        flowRate = _flowRate;
        hub = LibreeHub(_hub);
    }

    /*------------------------------------------------------------*|
    |*  # MODIFIERS                                               *|
    |*------------------------------------------------------------*/

    modifier onlyManager() {
        require(
            msg.sender == hub.getManagerForVault(address(this)),
            "Only Manager can access"
        );
        _;
    }

    /*------------------------------------------------------------*|
    |*  # TRANSFER FUNCTIONS                                      *|
    |*------------------------------------------------------------*/

    /**
     * @notice Withdraw funds from the vault.
     * @param _amount Amount to withdraw.
     * @param _to Address to send the funds withdrawn.
     */
    function withdraw(
        uint256 _amount,
        address _to
    ) external onlyManager nonReentrant {
        if (paymentToken.balanceOf(address(this)) < _amount) {
            revert Errors.InsufficientBalance();
        }
```

```
        paymentToken.transfer(_to, _amount);
    }
```

An attacker can create a malicious contract that has a `getManagerForVault()` function. They can then front run the creation of this vault and pass this contract as the address for hub in the constructor. Now whenever `onlyManger()` is called it will always return true. If tokens are sent to this contract the tokens can then be withdrawn by calling `withdraw()`.

Remediation Steps

Consider adding an `onlyOwner` modifier to the `intialize` function that way only the deployer of the vault can call it to set the hub contract.

[M] - Initialize vault in vault factory is lacking an access modifier

This allows an attacker to initialize vaults since the `initVault()` function is missing an access modifier and allows anyone to initialize a new vault. Theoretically an attacker could continuously call this function and DOS the system. Though the likelihood is very low.

The following function is an external function.

```
    function initVault(
        ISuperToken _paymentToken,
        int96 _flowRate,
        address _manager
    ) external returns (address _newVault) {
        if (address(_paymentToken) == address(0))
            revert Errors.InvalidAddress();
        if (_manager == address(0)) revert Errors.InvalidAddress();
        if (_flowRate <= 0) revert Errors.InvalidFlowrate(_flowRate);

        _newVault = Clones.clone(vaultImplementation);

        Vault(_newVault).initialize(_paymentToken, _flowRate, _manager);

        emit VaultInitialized(_newVault, address(_paymentToken),
_flowRate, _manager);
    }
```

Remediation Steps

Considering the `FactoringManger` is the only one calling this function you could add a modifier that requires only the manager is allowed to call the vault. Since `FactoringManager` could be replaced in the future you could create a setter function that updates the manager in the modifier.

[L] - Counters library is deprecated in new version of OZ contracts

OpenZeppelin has removed the counters library in v5.0.0. Consider implementing your own counter for NFT mints as when you update it may cause issues. You can also lock you version of `@openzeppelin/contracts` in your `package.json`.

https://github.com/OpenZeppelin

[G] - Two MSTOREs saved by using inline calls

Here are two places where you can save an MSTORE if you want. Note I did not test the gas savings here but my guess is they're quiet small as these functions won't be called often.

```
  function updateMetadata(
        uint256 _subscription,
        string calldata _metadata
    ) external {

        //@audit - storing this just to use it once could save you a
mstore by calling inline
        Subscription storage subscription = subscriptions[_subscription];

        NFTActions.checkNFTOwner(
            config.getFlowNFTAddress(),
            msg.sender,
            subscription.tokenId,
            Constants.TOKEN_FLOW_TYPE
        );

        string memory oldMetadata = subscriptions[_subscription].metadata;
        subscriptions[_subscription].metadata = _metadata;
        emit MetadataUpdated(msg.sender, _subscription, oldMetadata,
_metadata);
    }
```

```
    function checkSubscriptionActive(uint256 _subscription) internal view
{
        //@audit - storing this once just to use it once could save you a
mstore
        Subscription memory subscription = subscriptions[_subscription];

        if (!subscription.active) {
            revert Errors.SubscriptionDisabled();
        }
    }
```

[G] - Struct can be packed

The subscription structs can be packed.

```
    struct Subscription {
        address user;
        address vault;
        ISuperToken paymentToken;
        uint256 tokenId;
        string metadata;
        int96 flowRate;
        bool active;
    }
```

Can turn into.

```
struct Subscription {
    address user;
    address vault;
    ISuperToken paymentToken;
    uint256 tokenId;
    int96 flowRate;
    bool active;
    string metadata;

}
```

With this layout:

1. The first slot will have `user`.
2. The second slot will have `vault`.
3. The third slot will have `paymentToken`.
4. The fourth slot will have `tokenId`.
5. The fifth slot will have `flowRate`(12 bytes) + `active` (1 byte)
6. The string metadata will point to a separate storage location, consuming its own set of slots based on its length.

[G] - TokenID starts at 1 so no need to check for >=0

This can just check if the tokenId is greater than zero since the mint tokens funtion increments before it mints starting token ids at 1.

```
            currentDebt.amount == 0 &&
            subscription.vault != address(0) &&
            //@audit- mints start at 1 not zero
```

```
            subscription.tokenId >= 0;
```