[H] - `createSubscription` should use the superToken interface to avoid funds getting stuck

Summary

A user can create a subscription and use a regular ERC20 token as the `paymentToken`. However in `unwrapAndTransfer` the function attempts to downgrade using the superToken interface before transferring. Since this function is used in the `withdraw` function a user would be unable to remove their funds as it would revert if a regular ERC20 token was used.

```
    /// @inheritdoc Manager
    function createSubscription(
        address _paymentToken,
        int96 _flowRate,
        string calldata _metadata,
        bytes calldata /* data */
    ) external override nonReentrant whenNotPaused {
        uint256 nftTokenId = NFTActions.mint(
            config.getFlowNFTAddress(),
            msg.sender,
            _metadata
        );
```

```
function _unwrapAndTransfer(address _superToken, uint256 _amount) private
{
        ISuperToken(_superToken).downgrade(_amount);
        address underlying =
ISuperToken(_superToken).getUnderlyingToken();
        uint256 underlyingBalance =
IERC20(underlying).balanceOf(address(this));
        IERC20(underlying).safeTransfer(msg.sender, underlyingBalance);
    }
```

Remediation Steps

Considering checking in `createSubscription` if the `paymentToken` is a superToken. Alternatively you can remove superTokens from the rest of the contract in order to make it compatible with any ERC20 though this may introduce other issues such as ERC777 re-entrancy and rebasing tokens getting stuck in the contract.

[M] - Use method `safeTransfer` instead of `transfer`

Vulnerability

Using `transfer` method instead of `safeTransfer` may cause some transactions to fail.

Summary

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification: Use the SafeERC20 library implementation from OpenZeppelin and call safeTransfer or safeTransferFrom when transferring ERC20 tokens.

Remediation Steps

Consider using OpenZeppelin's SafeERC20 library that handles checking these return values for you.

[L] - Use `forceApprove` instead of `approve`

Summary

Since we're now importing the SafeERC20 library, instead of approving to zero first for an ERC20 allowance you can just use OZs `forceApprove` function. This will help save gas instead of needing to call `approve` twice.As it will only `approve` to zero first if the specific token requires it.

There are two instances of this in the codebase.

`UserEscrow.sol` 96 `UserEscrow.sol` 126

```
    IERC20(underlying).approve(_superTokenAddress, 0);
    IERC20(underlying).approve(_superTokenAddress, netAmount);
```

[L] - `setVaultImplementation` should check to see if the address passed is a contract or EOA

Summary

Similar to how OZ checks for `code.length` in their Beacon proxy implementation instead of doing a zero address check in `setVaultImplementation` you can check instead that the address being passed is a contract address. This will do two things, it will check that the address is not a zero address, and check that it is not an EOA. This is a more thorough check then just checking to see if the address is a zero address.

```
    function setVaultImplementation(
        address _newVaultImplementation
    ) external onlyOwner {
```

```
        if (_newVaultImplementation == address(0))
            revert Errors.InvalidAddress();

        address oldImplementation = vaultImplementation;
        vaultImplementation = _newVaultImplementation;

        emit VaultImplementationChanged(
            _newVaultImplementation,
            oldImplementation
        );
    }
```

Remediation Steps

Consider checking for `code.length` instead

```
    function setVaultImplementation(
        address _newVaultImplementation
    ) external onlyOwner {
     if (_newImplementation.code.length == 0) {
        revert InvalidImplementation(newImplementation);

        address oldImplementation = vaultImplementation;
        vaultImplementation = _newVaultImplementation;

        emit VaultImplementationChanged(
            _newVaultImplementation,
            oldImplementation
        );
    }
```

[G] - Redundant modifiers can be consolidated into just one modifier

Summary

In `HubAccessControl` there are two modifiers that are performing the same check. You could consolidate these into one modifier that reverts with an error message such as "Not Authorized". This would help save on deployment gas.

NOTE - The gas savings on this are very small. Considering you're also deploying to Polygon it may not even be worth changing. However I wanted to note it either way.

```
    modifier onlyActiveManager() {
        require(
            hub.getManagerStatus(msg.sender),
            "Only active manager can access"
        );
        _;
    }

    /**
     * @dev Modifier to restrict minting to authorized managers.
     * @notice This modifier ensures that only authorized managers
     */
    modifier onlyMinter() {
        require(hub.getManagerStatus(msg.sender), "Only managers can
mint");
        _;
```

Areas of Concern

There should be a lot more testing for unhappy paths and overall more test cases. Below is an example of testing if a contract can be reinitalized by calling `initializer`.

```
    //@audit - creating tests that fail after initialization are also
helpful
    it("should revert when trying it initialize", async function () {
        const depositAmount = 100
        await dai.connect(alice).approve(userEscrowFactory.address,
depositAmount)

        expect(
            await
userEscrowFactory.connect(alice).createEscrowAndDeposit(daix.address,
depositAmount)
        ).to.emit(userEscrowFactory, 'Deposit').withArgs(alice.address, 0,
depositAmount, dai.address);

        const aliceEscrowAfter = await
userEscrowFactory.getUserEscrow(alice.address)

        const escrow = await ethers.getContractAt('UserEscrow',
aliceEscrowAfter)
        await expect(
            escrow.connect(bob).initialize(bob.address)
        ).to.be.revertedWith('Initializable: contract is already
initialized'
        )
    });
```

Ideas for tests

- Check implementation can't be set to zero address
- Check that upgradeTo cannot be called by a random address
- Should test that fee on transfer tokens work with the escrow
- Should test that contracts don't work when they're paused
- Should test that unpausing works
- Should test for ownable two step
- Should test that ownable two step reverts if called by a random address
- Check that if a user who isn't onlyUser of an escrow call subscribe it will revert
- Check that if a user who isn't onlyUser of an escrow calls unsubscribe it will revet

Eventually writing fuzz/invariant tests may be a good idea also but starting with testing for unhappy paths would help a lot without needing to write much code considering you can just reuse the tests you already have with minor tweaks as shown above.