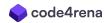


Stream Protocol

Smart Contract Security Assessment

Version 1.0

Audit dates: Feb 06 — Feb 10, 2025 Audited by: said windhustler



Contents

1. Introduction

- 1.1 About Zenith
- 1.2 Disclaimer
- 1.3 Risk Classification

2. Executive Summary

- 2.1 About Stream Protocol
- 2.2 Scope
- 2.3 Audit Timeline
- 2.4 Issues Found

3. Findings Summary

4. Findings

- 4.1 Critical Risk
- 4.2 Medium Risk
- 4.3 Low Risk
- 4.4 Informational



1. Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2. Executive Summary

2.1 About Stream Protocol

The average retail user does not have the time, knowledge, or access to properly deploy their funds across defi to optimally generate returns on their USDC/BTC/ETH. This market is massively inefficient, can be optimized significantly further than the current market allows for, and is an enormous opportunity that most people miss out on while being stuck in the

shitcoin trenches. Stream aims to solve this issue by making optimized yield farming easily accessible with the push of a button with zero fees to anyone with access to a phone.

2.2 Scope

Repository	StreamDefi/contracts
Commit Hash	21e5e7a566f1b89be2b03d5d5bfed2b5e4449b3c

2.3 Audit Timeline

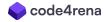
DATE	EVENT
Feb 06, 2025	Audit start
Feb 10, 2025	Audit end
Feb 14, 2025	Report published

2.4 Issues Found

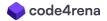
SEVERITY	COUNT
Critical Risk	1
High Risk	0
Medium Risk	3
Low Risk	12
Informational	1
Total Issues	17

3. Findings Summary

ID	DESCRIPTION	STATUS
C-1	Using tokens with 18 decimals leads to significant loss of funds when bridging across chains	Resolved



M-1	`instantUnstake` and `unstake` can be used to bypass `minimumSupply`	Resolved
M-2	The vault's `cap` can be bypassed, and the `minimumSupply` can be reached earlier than intended	Resolved
M-3	Providing yield on the first `rollToNextRound` causes unexpected behavior	Acknowledged
L-1	Short-term DoS in `processWithdrawals` function	Acknowledged
L-2	`getSharesFromReceipt` function should check if the round from stakeReceipt is greater than 1	Acknowledged
L-3	Setters for key state variables in the `StableWrapper` and `StreamVault` contracts should be considered for removal	Acknowledged
L-4	Missing check for non-zero creditor address in `StreamVault::depositAndStake` function	Resolved
L-5	Missing balance cap check in `StreamVault::rollToNextRound` function	Resolved
L-6	`rollToNextRound` could revert under certain conditions	Acknowledged
L-7	`rescueTokens` inside `StreamVault` could cause issues	Acknowledged
L-8	`transferAsset` inside `StableWrapper` could cause issues	Acknowledged
L-9	`processWithdrawals` has a risk due to the lack of slippage/amount control.	Acknowledged
L-10	`StableWrapper` may not work properly when using certain tokens as assets	Acknowledged
L-11	Lack of slippage control in `unstake` and `unstakeAndWithdraw`	Resolved
L-12	Dangerous usage of the `stableWrapper` balance inside `rollToNextRound`	Resolved
1-1	Lack of time interval restrictions on `rollToNextRound` and `processWithdrawals`	Acknowledged



4. Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] Using tokens with 18 decimals leads to significant loss of funds when bridging across chains

Severity: Critical Status: Resolved	Severity: Critical	Status: Resolved	
-------------------------------------	--------------------	------------------	--

Target

- <u>StreamVault.sol</u>
- <u>MyOFT.sol</u>
- StableWrapper.sol

Severity:

- Impact: High
- Likelihood: High

Description: All three contracts that extend the OFT - StreamVault, StableWrapper, and MyOFT -- override the decimals() from the underlying ERC20 contract and sharedDecimals() from the OFTCore contract.

This introduces an issue where if the token that is being wrapped has 18 decimals, such as USDT, sending across chains amounts bigger than type(uint64).max will result in a significant loss of funds.

The following POC demonstrates the issue:

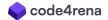
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
import "forge-std/Test.sol";
import {MyOFT} from "../src/OFT.sol";
import {IOFT, SendParam, MessagingFee} from "@layerzerolabs/oft-
evm/contracts/interfaces/IOFT.sol";
import {StableWrapper} from "../src/StableWrapper.sol";
import {TestHelperOz5} from "@layerzerolabs/test-devtools-evm-
foundry/contracts/TestHelperOz5.sol";
import {OptionsBuilder} from "@layerzerolabs/oapp-
evm/contracts/oapp/libs/OptionsBuilder.sol";
```



```
import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {StreamVault} from "../src/StreamVault.sol";
import {Vault} from "../src/lib/Vault.sol";
contract OFTMock is MyOFT {
    constructor(
        string memory _name,
        string memory _symbol,
        address _lzEndpoint,
        address _delegate,
        uint8 _underlyingDecimals
    ) MyOFT(_name, _symbol, _lzEndpoint, _delegate, _underlyingDecimals)
{}
    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
}
contract StableWrapperMock is StableWrapper {
    constructor(
        address _asset,
        string memory _name,
        string memory _symbol,
        uint8 _underlyingDecimals,
        address _keeper,
        address _lzEndpoint,
        address _delegate
    ) StableWrapper(_asset, _name, _symbol, _underlyingDecimals, _keeper,
_lzEndpoint, _delegate) {}
}
contract ERC20Mock is ERC20 {
    constructor(string memory _name, string memory _symbol) ERC20(_name,
_symbol) {}
    function mint(address _to, uint256 _amount) public {
        _mint(_to, _amount);
    }
}
contract OFTTest is Test, TestHelperOz5 {
    using OptionsBuilder for bytes;
    uint32 internal ethEid = 1;
    uint32 internal arbitrumEid = 2;
```



```
address admin = makeAddr("admin");
    address userB = makeAddr("userB");
    function setUp() public override {
        super.setUp();
       setUpEndpoints(1, LibraryType.UltraLightNode);
       setUpEndpoints(2, LibraryType.UltraLightNode);
    }
   // forge test --match-test testOFT --watch -vv
    function testOFT() public {
       vm.startPrank(admin);
       vm.deal(admin, 1000 ether);
       ERC20Mock usdt = new ERC20Mock("USDT", "USDT");
       uint8 decimals = 6;
       uint104 amountToDeposit = uint104(2**64); // ~ $18.
       StreamVault streamVault = new StreamVault(
            "StreamVault", "SV", address(0x123),
address(endpoints[ethEid]), admin, Vault.VaultParams(decimals, 10e10, 1e7
ether)
       );
       StableWrapper stableWrapper = new StableWrapper(
            address(usdt), "StableWrapper", "SW", decimals,
address(streamVault), address(endpoints[ethEid]), admin
        );
        streamVault.setStableWrapper(address(stableWrapper));
       OFTMock oftA = new OFTMock("OFTA", "OFTA",
address(endpoints[arbitrumEid]), admin, decimals);
       console.log("decimalConversionRate",
stableWrapper.decimalConversionRate());
       console.log("sharedDecimals", stableWrapper.sharedDecimals());
       console.log("decimals", stableWrapper.decimals());
       address[] memory ofts = new address[](2);
       ofts[0] = address(streamVault);
       ofts[1] = address(oftA);
       wireOApps(ofts);
       // mint USDT to admin
       usdt.mint(admin, amountToDeposit);
```



```
// depositAndStake
       usdt.approve(address(stableWrapper), amountToDeposit);
        streamVault.depositAndStake(amountToDeposit, admin);
       vm.warp(block.timestamp + 1 days);
       streamVault.rollToNextRound(0, false);
        console.log("roundPricePerShare",
streamVault.roundPricePerShare(1));
       console.log("omniTotalSupply", streamVault.omniTotalSupply());
       bytes memory options =
OptionsBuilder.newOptions().addExecutorLzReceiveOption(200000, 0);
       SendParam memory _sendParam = SendParam({
            dstEid: arbitrumEid,
            to: addressToBytes32(address(userB)),
            amountLD: amountToDeposit,
            minAmountLD: amountToDeposit,
            extraOptions: options,
            composeMsg: new bytes(0),
            oftCmd: new bytes(0)
       });
       MessagingFee memory _fee = streamVault.quoteSend(_sendParam,
false);
       printStakeReceipt(streamVault, admin);
       streamVault.bridgeWithRedeem{value: _fee.nativeFee}(_sendParam,
_fee, payable(admin));
       printStakeReceipt(streamVault, admin);
       console.log("userB balance before", oftA.balanceOf(userB));
       verifyPackets(arbitrumEid, addressToBytes32(address(oftA)));
       console.log("userB balance after", oftA.balanceOf(userB));
   }
    function printStakeReceipt(StreamVault streamVault, address user)
public view {
       (uint16 round, uint104 amount, uint128 unredeemedShares) =
streamVault.stakeReceipts(user);
       console.log("stakeReceipt for user", user);
       console.log("stakeReceipt.amount", amount);
       console.log("unredeemedShares", unredeemedShares);
```

}

Let's walk through the steps of what's happening.

- 1. StreamVault, MyOFT, and StableWrapper for USDT are deployed with decimals set to 6. For the vulnerability to exist it doesn't matter if the decimals are set to 6 or 18.
- With all these three contracts the shared decimals are equal to the ERC20 decimals. Based on this the decimalsConversionRate for the OFT is set to 1, meaning there is no conversion between the chains.
- 3. Since USDT has 18 decimals, staking an amount of <u>2**64</u> USDT (~\$18) will yield a similar amount of shares.
- 4. Once the user redeems and bridges these shares they are converted inside the OFTCore contract to shared decimals. The problem here is that this value is cast to uint64.

```
## OFTCore.sol
function _buildMsgAndOptions(
    SendParam calldata _sendParam,
    uint256 _amountLD
) internal view virtual returns (bytes memory message, bytes memory
options) {
   bool hasCompose;
   // @dev This generated message has the msg.sender encoded into the
payload so the remote knows who the caller is.
    (message, hasCompose) = OFTMsgCodec.encode(
        _sendParam.to,
        _toSD(_amountLD),
        // @dev Must be include a non empty bytes if you want to compose,
EVEN if you dont need it on the remote.
        // EVEN if you dont require an arbitrary payload to be sent...
eg. '0x01'
        _sendParam.composeMsg
    );
function _toSD(uint256 _amountLD) internal view virtual returns (uint64
amountSD) {
       return uint64(_amountLD / decimalConversionRate);
>>>
}
```

5. In solidity if you're casting a larger number to a smaller type it takes the least significant bits.



- 6. In the POC above inputting 2**64 USDT into _toSD yields 0 amount.
- 7. When this is received on the remote chain a conversion back to local decimals is tried but the damage was already done and it simply results in 0 tokens, while the whole amount is burned on the source chain.

Another observation is how the overriding of sharedDecimals and decimals skips important checks in the OFTCore contract.

It's the same for all three contracts but taking the example of MyOFT, the underlyingDecimals are set in the constructor of MyOFT so during the constructor call to the OFT and OFTCore the value for sharedDecimals and decimals is 0. In this case, it doesn't make a difference since they have the same value but otherwise checks in the child contract constructor might get skipped.

```
**OFTCore.sol**
    /**
     * @dev Constructor.
     * @param _localDecimals The decimals of the token on the local chain
(this chain).
     * @param _endpoint The address of the LayerZero endpoint.
     * @param _delegate The delegate capable of making OApp
configurations inside of the endpoint.
     */
    constructor(uint8 _localDecimals, address _endpoint, address
_delegate) OApp(_endpoint, _delegate) {
        if (_localDecimals < sharedDecimals()) revert</pre>
InvalidLocalDecimals();
        decimalConversionRate = 10 ** (_localDecimals -
sharedDecimals());
   }
```

Recommendation: Having the same value for sharedDecimals and decimals leads to the critical issue described in the POC.

Consider keeping the default 6 decimals for sharedDecimals, and the decimals should track the underlying token decimals.

Stream Protocol: Resolved with the following commit

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] `instantUnstake` and `unstake` can be used to bypass `minimumSupply`

Severity: Medium	Status: Resolved

Target

• StreamVault.sol#L292-L309

Severity:

- Impact: Medium
- Likelihood: Medium

Description: When instantUnstake is called, it allows the staker to withdraw their stake for the current round. However, this doesn't verify if the remaining stableWrapper inside the StreamVault is greater than the minimumSupply, allowing users to bypass the minimumSupply restriction.

This is also the case with unstake, when unstake is called, it is possible that the remaining stableWrapper is less than the configured minimumSupply.

Recommendation: Consider checking the minimumSupply restriction when users call instantUnstake/unstake and they not unstake all of their stakes.

Stream Protocol: Fixed with @438afe0410409... & @f1ce981a4d48...

Zenith: Verified.



[M-2] The vault's `cap` can be bypassed, and the `minimumSupply` can be reached earlier than intended

Severity: Medium

Status: Resolved

Target

• StreamVault.sol#L240-L246

Severity:

- Impact: Medium
- Likelihood: Low

Description: Due to the usage of stableWrapper's balanceOf inside _stakeInternal to get the totalWithStakedAmount value, users can bypass the minimumSupply by directly donating stableWrapper to StreamVault. They would then be able to stake an amount lower than the minimumSupply.

Also, due to same root cause, user can cause cap to reached earlier than intended by directly donating stableWrapper to StreamVault.

Recommendation: Consider tracking the totalWithStakedAmount instead of relying on stableWrapper's balanceOf.

Stream Protocol: Resolved with @13df0b775314... & @ce8278725cd4...

Zenith: Verified



[M-3] Providing yield on the first `rollToNextRound` causes unexpected behavior

Severity: Medium

Status: Acknowledged

Target

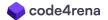
• StreamVault.sol#L431-L511

Severity:

- Impact: High
- Likelihood: Low

Description: When rollToNextRound is called for the first time, it is possible that a yield is provided with a non-zero value. However, since no shares have been minted previously, the yield will not be distributed to any user. But because the currentBalance is greater than (or lower than, in the case of negative yield) the balance, it will mint/burn the new stableWrapper so that the balance inside the StreamVault matches the currentBalance.

```
function rollToNextRound(
        uint256 yield,
        bool isYieldPositive
    ) external onlyOwner nonReentrant {
        uint256 balance = IERC20(stableWrapper).balanceOf(address(this));
>>>
        uint256 currentBalance;
        if (isYieldPositive) {
>>>
            currentBalance = balance + yield;
        } else {
>>>
            currentBalance = balance - yield;
        }
       // ...
        if (currentBalance > balance) {
>>>
            IStableWrapper(stableWrapper).permissionedMint(
                address(this),
                currentBalance - balance
            );
            emit RoundRolled(
                currentRound,
                newPricePerShare,
                mintShares.
                currentBalance - balance,
                0,
```



```
yield,
                isYieldPositive
            );
        } else if (currentBalance < balance) {</pre>
>>>
            IStableWrapper(stableWrapper).permissionedBurn(
                address(this),
                balance - currentBalance
            );
            emit RoundRolled(
                currentRound,
                newPricePerShare,
                mintShares,
                0,
                balance - currentBalance,
                yield,
                isYieldPositive
            );
        } else {
            emit RoundRolled(
                currentRound,
                newPricePerShare,
                mintShares,
                0,
                0,
                yield,
                isYieldPositive
            );
        }
    }
```

The increase/decrease in stableWrapper during the first rollToNextRound will impact users who stake in the first round when the next rollToNextRound is called to calculate the new roundPricePerShare. This means users can front-run first rollToNextRound operation to make a profit.

PoC:

```
function test_FrontRunAttack() public {
    vm.prank(depositor1);
    streamVault.depositAndStake(10e6, depositor1);
    // attacker see that first roll will include positive yield
    address attacker = address(0x1234);
    usdc.mint(attacker, 10e6);
    vm.startPrank(attacker);
```



```
usdc.approve(address(stableWrapper), 10e6);
streamVault.depositAndStake(10e6, attacker);
vm.stopPrank();
// first rollToNextRound
vm.prank(owner);
streamVault.rollToNextRound(2e6, true);
// redeem
vm.prank(depositor1);
streamVault.maxRedeem();
// attacker redeem
vm.prank(attacker);
streamVault.maxRedeem();
console.log("attacker share : ");
console.log(streamVault.balanceOf(address(attacker)));
vm.prank(owner);
streamVault.rollToNextRound(0, false);
console.log("attacker balance after second rollToNextRound : ");
console.log(streamVault.accountVaultBalance(attacker));
```

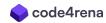
Output :

}

```
Logs:
attacker share :
10000000
attacker balance after second rollToNextRound :
11000000
```

Recommendation: Prevent non-zero yield when rollToNextRound is called for the first time.

Stream Protocol: Acknowledged. Decided its more gas effective to not have a check and make sure to correctly roll the first round with no yield at deployment.



4.3 Low Risk

A total of 12 low risk findings were identified.

[L-1] Short-term DoS in `processWithdrawals` function

0 11 1	
Severity: Low	Status: Acknowledged

Target

• StableWrapper.sol#L304

Severity:

- Impact: Low
- Likelihood: Low

Description: The StableWrapper::processWithdrawals function can be DoSed by frontrunning the call and initiating a withdrawal if the owner hasn't set enough allowance or doesn't have enough balance to cover the increased difference between withdrawalAmountForEpoch and depositAmountForEpoch.

Recommendation: While calling the function, ensure that the owner has set enough allowance and has enough balance to cover the difference between withdrawalAmountForEpoch and depositAmountForEpoch even if someone completes a withdrawal in that same block.

Stream Protocol: Acknowledged. I think we could leave it as is for two reasons: The owner can make sure to always have a buffer of funds as mentioned. Furthermore, there are no incentives to DDos in this way, and is costly due to gas



[L-2] `getSharesFromReceipt` function should check if the round from stakeReceipt is greater than 1

Severity: Low Status: Acknowledged

Target

• ShareMath.sol#L45

Severity:

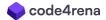
- Impact: Low
- Likelihood: Low

Description: The getSharesFromReceipt function calculates unredeemed shares for previous rounds. The if condition checks if the round from stakeReceipt is greater than 0, while the vaultState.round starts at 1 so this condition always holds true.

Recommendation: Change the condition to check if the round from stakeReceipt is greater than 1.

```
**StreamVault.sol**
function getSharesFromReceipt(Vault.StakeReceipt memory stakeReceipt,
uint256 currentRound, uint256 assetPerShare, uint256 decimals) internal
pure returns (uint256 unredeemedShares) {
    if (stakeReceipt.round > 0 && stakeReceipt.round < currentRound) {
        if (stakeReceipt.round > 1 && stakeReceipt.round < currentRound) {
            uint256 sharesFromRound = assetToShares(stakeReceipt.amount,
            assetPerShare, decimals);
            return uint256(stakeReceipt.unredeemedShares) + sharesFromRound;
        }
        return stakeReceipt.unredeemedShares;
}</pre>
```

Stream Protocol: Acknowledged



[L-3] Setters for key state variables in the `StableWrapper` and `StreamVault` contracts should be considered for removal

Target

- <u>StreamVault.sol</u>
- StableWrapper.sol

Severity:

- Impact: High
- Likelihood: Low

Description: There are setter functions that modify key state variables, such as asset and decimals, in the StableWrapper and StreamVault contracts. If these setters are not intended for use, they can be removed to prevent unnecessary modifications. Since the owner is trusted and it's assumed these values won't change, keeping these setters may be redundant.

Recommendation: If these setters are not meant to be used, consider removing them:

If decimals should remain unchanged, add a condition to enforce this. Additionally, the setCap function can be removed since cap can already be set through setVaultParams.

```
## StreamVault.sol
+ error DecimalsCannotBeModified();
- function setCap(uint256 newCap) external onlyOwner {
-     if (newCap == 0) revert CapMustBeGreaterThanZero();
-     ShareMath.assertUint104(newCap);
```



```
- emit CapSet(vaultParams.cap, newCap);
- vaultParams.cap = uint104(newCap);
- }
function setVaultParams(
    Vault.VaultParams memory newVaultParams)
) external onlyOwner {
    if (newVaultParams.cap == 0) revert CapMustBeGreaterThanZero();
+ if (newVaultParams.decimals != vaultParams.decimals) revert
DecimalsCannotBeModified();
    vaultParams = newVaultParams;
}
```

Stream Protocol: Partially-resolved. The reasoning for keeping the setAsset and setDecimal is given that the contracts are already max centralized - if there was an unforseen situation where changing decimals /asset would fix it (albeit unlikely) its convenient to have

Zenith: The <u>@7d06c98391453..</u> removes the setCap function.



[L-4] Missing check for non-zero creditor address in `StreamVault::depositAndStake` function

Severity: Low

Status: Resolved

Target

• <u>StreamVault.sol#L152</u>

Severity:

- Impact: Low
- Likelihood: Low

Description: The StreamVault::depositAndStake function doesn't check if the creditor address is address(0). If a user sets the creditor as the zero address, the tokens will still be staked in the contract, but since only the zero address can unstake or redeem them, they will be permanently stuck.

Recommendation: Add a validation check in StreamingNFT::depositAndStake to ensure the creditor address is non-zero:

```
function depositAndStake(
    uint104 amount,
    address creditor
) external nonReentrant {
+    if (creditor == address(0)) revert AddressMustBeNonZero();
    IStableWrapper(stableWrapper).depositToVault(msg.sender, amount);
    // Then stake the wrapped tokens
    _stakeInternal(amount, creditor);
}
```

Stream Protocol: Fixed with the following commit

Zenith: Verified.



[L-5] Missing balance cap check in `StreamVault::rollToNextRound` function

Sev	veritv	v:	Low

Status: Resolved

Target

• StreamVault.sol#L431

Severity:

- Impact: Low
- Likelihood: Low

Description: The StreamVault::rollToNextRound function does not check whether the total balance of the StableWrapper token, including yield, exceeds the vault's cap. This means the contract's StableWrapper token balance can go beyond the defined vault cap, which is meant to limit the balance.

Recommendation: Add a validation check in StreamVault::rollToNextRound to ensure the new balance does not exceed the vault's defined cap:

```
function rollToNextRound(
    uint256 yield,
    bool isYieldPositive
) external onlyOwner nonReentrant {
    uint256 balance = IERC20(stableWrapper).balanceOf(address(this));
    uint256 currentBalance;
    if (isYieldPositive) {
        currentBalance = balance + yield;
    } else {
        currentBalance = balance - yield;
    }
    Vault.VaultParams memory _vaultParams = vaultParams;
   if (currentBalance > uint256(_vaultParams.cap)) {
+
       revert CapExceeded();
   }
+
    if (currentBalance < uint256(_vaultParams.minimumSupply)) {</pre>
        revert MinimumSupplyNotMet();
    }
    . . .
}
```



Stream Protocol: Fixed with the following commit

Zenith: Verified



[L-6] `rollToNextRound` could revert under certain conditions

Severity: L	_ow
-------------	-----

Status: Acknowledged

Target

- StreamVault.sol#L431-L511
- ShareMath.sol#L59

Severity:

- Impact: Medium
- Likelihood: Low

Description: When rollToNextRound, there are certain scenarios which could cause rollToNextRound to revert unexpectedly.

1. When rollToNextRound is called, if the yield is negative and the yield provided is greater than balance - state.totalPending, it will cause currentBalance to be lower than state.totalPending. When calculating pricePerShare, the operation will revert due to underflow.

2. When rollToNextRound is called, if the yield is negative and the yield provided is equal to balance - state.totalPending, it will cause currentBalance to be equal to state.totalPending. When calculating pricePerShare, the price per share will be O, causing the assetToShares operation to revert when calculating mintShares.



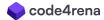
```
{
    // If this throws, it means that vault's
roundPricePerShare[currentRound] has not been set yet
    // which should never happen.
    // Has to be larger than 1 because `1` is used in
`initRoundPricePerShares` to prevent cold writes.
>>> require(assetPerShare > PLACEHOLDER_UINT, "Invalid
assetPerShare");
    return (assetAmount * (10 ** decimals)) / assetPerShare;
}
```

PoC:

```
function test_RollFail() public {
    vm.prank(depositor1);
    streamVault.depositAndStake(1e6, depositor1);
    // first rollToNextRound
    vm.prank(owner);
    streamVault.rollToNextRound(0, true);
    address user1 = address(0x1234);
    usdc.mint(user1, 1e6);
    vm.startPrank(user1);
    usdc.approve(address(stableWrapper), 1e6);
    streamVault.depositAndStake(le6, user1);
    vm.stopPrank();
    // first rollToNextRound
    vm.prank(owner);
    vm.expectRevert();
    streamVault.rollToNextRound(1e6+1, false);
}
```

Recommendation: Consider all of these scenarios and add more restrictions to prevent them, making the reverts more verbose.

Stream Protocol: Acknowledged



[L-7] `rescueTokens` inside `StreamVault` could cause issues

Severity: Low

Status: Acknowledged

Target

• StreamVault.sol#L672-L677

Severity:

- Impact: Low
- Likelihood: Low

Description: rescueTokens allows the owner to directly transfer any token from the StreamVault to themselves. If the transferred token is the stableWrapper, it could cause issues and break the balance assumptions of the stableWrapper inside StreamVault.

Recommendation: Consider restricting rescueTokens, if the provided _token is the stableWrapper, revert the operation.

Stream Protocol: Acknowledged.



[L-8] `transferAsset` inside `StableWrapper` could cause issues

Severity: Low

Status: Acknowledged

Target

• StableWrapper.sol#L322-L332

Severity:

- Impact: Medium
- Likelihood: Low

Description: transferAsset allows the owner to directly transfer any token from the StableWrapper to any address. If the transferred token is the asset used inside the wrapper, it could cause issues, since processWithdrawals, which relies on withdrawalAmountForEpoch and depositAmountForEpoch to settle withdrawals/deposits, depends on the correct balance of the asset inside the StableWrapper.

Recommendation: Consider restricting transferAsset, if the provided _token is the asset, revert the operation.

Stream Protocol: Acknowledged.



[L-9] `processWithdrawals` has a risk due to the lack of slippage/amount control.

Severity: Low

Status: Acknowledged

Target

• StableWrapper.sol#L301-L314

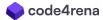
Severity:

- Impact: Medium
- Likelihood: Low

Description: When processWithdrawals is executed, depending on the amount of withdrawalAmountForEpoch and depositAmountForEpoch, asset could be transferred from the owner to the wrapper, or vice versa. This function could potentially result in unpredictable outcomes due to the lack of slippage/amount control. For instance, if, between the processWithdrawals request and execution, a large number of withdrawal requests suddenly occur, causing an unexpectedly large withdrawalAmountForEpoch, the system could suddenly lose a significant amount of asset.

Recommendation: Consider allowing the caller to provide the maximum asset they expect to send to the wrapper, or restricting withdrawalAmountForEpoch. When a user requests a withdrawal and the withdrawalAmountForEpoch exceeds a certain value per epoch, revert the withdrawal request.

Stream Protocol: Acknowledged.



[L-10] `StableWrapper` may not work properly when using certain tokens as assets

Severity: Low

Status: Acknowledged

Target

• StableWrapper.sol

Severity:

- Impact: Medium
- Likelihood: Low

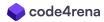
Description: Some ERC20 tokens might cause unexpected issues if used as an asset inside StableWrapper. For instance, if a token charges a fee on transfer, the actual token received inside StableWrapper will be lower than the provided amount, but the wrapper will mint the specified amount of tokens to the user. Some commonly used tokens, such as USDT and USDC, have the ability to enable this feature, but it is currently disabled.

Another case could involve using rebasing tokens as an asset, or using with tokens that have similar properties, where the amount provided is not equal to the received amount (e.g., cUSDCv3). In such cases, when the amount equals type(uint256).max in their transfer functions, only the user's balance is transferred.

Check this list for more potential ERC20 issues: https://github.com/d-xo/weird-erc20

Recommendation: If you plan to use a token that may implement fee-on-transfer, make sure to always check the balance before and after the transfer to get the correct amount. For other ERC20 cases, check the provided repo and adjust accordingly.

Stream Protocol: Acknowledged. If this was to be enabled we would redeploy.



[L-11] Lack of slippage control in `unstake` and `unstakeAndWithdraw`

Severity: Low

Status: Resolved

Target

- <u>StreamVault.sol#L164-L173</u>
- <u>StreamVault.sol#L315-L318</u>

Severity:

- Impact: Medium
- Likelihood: Low

Description: When users call unstake and unstakeAndWithdraw to withdraw assets, the last round's roundPricePerShare will be used to calculate the amount of assets they should receive.

```
function _unstake(
       uint256 numShares,
        address to
    ) internal returns (uint256) {
        if (numShares == 0) revert AmountMustBeGreaterThanZero();
        if (to == address(0)) revert AddressMustBeNonZero();
        // We do a max redeem before initiating a withdrawal
        // But we check if they must first have unredeemed shares
        {
            Vault.StakeReceipt memory stakeReceipt =
stakeReceipts[msg.sender];
            if (stakeReceipt.amount > 0 || stakeReceipt.unredeemedShares
> () {
                _redeem(0);
           }
        }
        // This caches the `round` variable used in shareBalances
        uint256 currentRound = vaultState.round;
        if (currentRound < MINIMUM_VALID_ROUND)</pre>
            revert RoundMustBeGreaterThanOne();
>>>
       uint256 withdrawAmount = ShareMath.sharesToAsset(
            numShares,
            roundPricePerShare[currentRound - 1],
            vaultParams.decimals
```



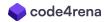
```
);
emit Unstake(msg.sender, withdrawAmount, currentRound);
_burn(msg.sender, numShares);
omniTotalSupply = omniTotalSupply - numShares;
IERC20(stableWrapper).safeTransfer(to, withdrawAmount);
return withdrawAmount;
}
```

However, users cannot specify slippage or the minimum asset they expect to receive from the operation. It is possible that between the unstake/unstakeAndWithdraw request and transaction execution, rollToNextRound is executed, causing the price per share to change, and the user might receive an unexpected amount of assets.

Recommendation: Allow users to specify the minimum asset they expect to receive from unstake/unstakeAndWithdraw.

Stream Protocol: Resolved with @c79c53ca71010...

Zenith: Verified



[L-12] Dangerous usage of the `stableWrapper` balance inside `rollToNextRound`

Severity: Low

Status: Resolved

Target

• StreamVault.sol#L435

Severity:

- Impact: Medium
- Likelihood: Medium

Description: When rollToNextRound is called, it will use stableWrapper's balanceOf to retrieve the balance inside StreamVault. The balance will impact the currentBalance value, which will be used to calculate the roundPricePerShare and mint new shares.

```
function rollToNextRound(
        uint256 yield,
        bool isYieldPositive
   ) external onlyOwner nonReentrant {
>>>
       uint256 balance = IERC20(stableWrapper).balanceOf(address(this));
        uint256 currentBalance;
        if (isYieldPositive) {
            currentBalance = balance + yield;
        } else {
            currentBalance = balance - yield;
        }
        Vault.VaultParams memory _vaultParams = vaultParams;
        if (currentBalance < uint256(_vaultParams.minimumSupply)) {</pre>
            revert MinimumSupplyNotMet();
        }
        Vault.VaultState memory state = vaultState;
        uint256 currentRound = state.round;
>>>
        uint256 newPricePerShare = ShareMath.pricePerShare(
            omniTotalSupply,
            currentBalance,
            state.totalPending,
            _vaultParams.decimals
        );
        roundPricePerShare[currentRound] = newPricePerShare;
```

```
vaultState.totalPending = 0;
vaultState.round = uint16(currentRound + 1);
vaultState.round = ShareMath.assetToShares(
state.totalPending,
newPricePerShare,
_vaultParams.decimals
);
_mint(address(this), mintShares);
omniTotalSupply = omniTotalSupply + mintShares;
// ...
}
```

This means StreamVault's roundPricePerShare can be manipulable, especially when allowIndependence is set to true.

PoC :

An attacker can manipulate the share price to steal assets from the next processed depositor.

```
function test_ManipulateSharePrice() public {
   vm.prank(owner);
    stableWrapper.setAllowIndependence(true);
   vm.prank(depositor1);
   streamVault.depositAndStake(1e6, depositor1);
   // first rollToNextRound
   vm.prank(owner);
   streamVault.rollToNextRound(0, true);
   address victim = address(0x1234);
   usdc.mint(victim, 1e6);
   vm.startPrank(victim);
   usdc.approve(address(stableWrapper), 1e6);
   streamVault.depositAndStake(1e6, victim);
   vm.stopPrank();
   vm.startPrank(depositor1);
   usdc.mint(depositor1, le12);
   usdc.approve(address(stableWrapper), 1e12);
   stableWrapper.deposit(depositor1, 1e12);
   stableWrapper.transfer(address(streamVault), 1e12);
```



```
vm.stopPrank();
    // first rollToNextRound
    vm.prank(owner);
    streamVault.rollToNextRound(0, true);
    // redeem
    vm.prank(depositor1);
    streamVault.maxRedeem();
    // victim redeem
    vm.prank(victim);
    streamVault.maxRedeem();
    console.log("victim share : ");
    console.log(streamVault.balanceOf(address(victim)));
    console.log("victim balance : ");
    console.log(streamVault.accountVaultBalance(victim));
    console.log("attacker share : ");
    console.log(streamVault.balanceOf(address(depositor1)));
    console.log("attacker balance : ");
    console.log(streamVault.accountVaultBalance(depositor1));
    console.log("balance inside vault : ");
    console.log(stableWrapper.balanceOf(address(streamVault)));
}
```

Output :

```
Logs:
victim share :
0
victim balance :
0
attacker share :
1000000
attacker balance :
1000001000000
balance inside vault :
1000002000000
```

Recommendation: Consider tracking the balance inside StreamVault rather than using stableWrapper's balanceOf

Stream Protocol: Resolved with @13df0b775314c... & @ce8278725cd40...

Zenith: Verified.

4.4 Informational

A total of 1 informational findings were identified.

[I-1] Lack of time interval restrictions on `rollToNextRound` and `processWithdrawals`

Severity: Informational Status: Acknowledged	
--	--

Target

- StableWrapper.sol#L301-L314
- StreamVault.sol#L431-L511

Severity:

- Impact: Low
- Likelihood: Low

Description: The documentation mentions that the yield distributed via rollToNextRound will be processed once a day. Additionally, processWithdrawals will be processed after one epoch (24 hours) has passed. Currently, there is no time interval restriction on either function, allowing them to be called at any time and at any interval, which could lead to issues and unpredictable behavior.

Recommendation: Consider adding a time interval restriction on both functions to align them with the documentation.

Stream Protocol: Acknowledged. This is done on purpose for additional flexibility by us.

