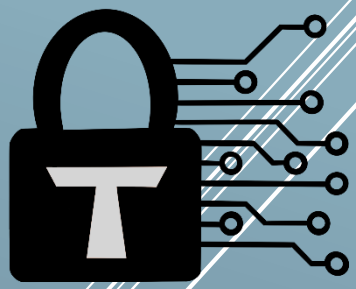
Trust Security



Smart Contract Audit

Reserve - DTF

24/12/2024

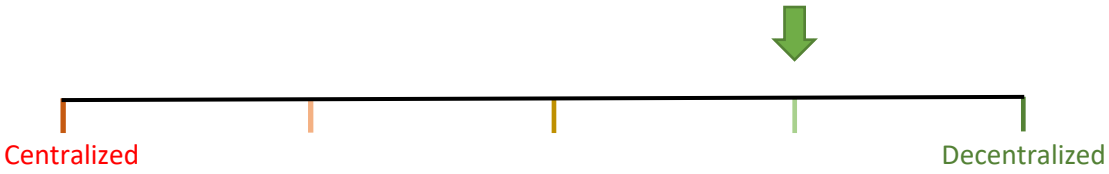
Executive summary

Category	Portfolio Management
Audited file count	15
Lines of Code	1216
Auditor	Trust HollaDieWaldfee cccz
Time period	16/12/2024- 24/12/2024

Findings

Severity	Total	Open	Fixed	Acknowledged
High	2	0	2	-
Medium	11	11	-	-
Low	8	8	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.12.24 10:10:35 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	24/12/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/Folio.sol
- contracts/staking/StakingVault.sol
- contracts/folio/FolioDeployer.sol
- contracts/governance/FolioGovernor.sol
- contracts/interfaces/IFolio.sol
- contracts/folio/FolioDAOFeeRegistry.sol
- contracts/folio/FolioVersionRegistry.sol
- contracts/staking/UnstakingManager.sol
- contracts/governance/GovernanceDeployer.sol
- contracts/folio/FolioProxy.sol
- contracts/interfaces/IFolioDAOFeeRegistry.sol
- contracts/interfaces/IFolioDeployer.sol
- contracts/interfaces/IFolioVersionRegistry.sol
- contracts/utils/Versioned.sol
- contracts/interfaces/IRoleRegistry.sol

Repository details

- **Repository URL:** https://github.com/reserve-protocol/dtfs
- **Commit hash:** a78f444831d0e47bce5b1b6b6fda4cecd56ffeb0

Note that some issues are already fixed in the commit hash above as auditing began informally on an earlier hash.

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Senior Watson at Sherlock and Lead Auditor for Trust Security and Renascence Labs.

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Excellent**	Project kept code as simple as possible, reducing attack risks
Documentation	**Moderate**	Project is still under active development and currently lacks documentation.
Best practices	**Good**	Project generally follows best practices.
Centralization risks	**Good**	Project does not introduce significant unnecessary centralization risks.

Findings

High severity findings

TRST-H-1 Attacker can freeze funds in *UnstakingManager*

- **Category:** Logical issues
- **Source:** UnstakingManager.sol
- **Status:** Fixed

Description

When a user withdraws tokens from *StakingVault*, the underlying tokens are sent to *UnstakingManager* for delayed unlocking and will create a Lock for the user. The user can call *claimLock()* to unlock the underlying tokens when it expires.

The problem is that *claimLock()* can be called even if the lock does not exist. It will set **lock.claimedAt = block.timestamp**.

```
function claimLock(uint256 lockId) external {
    Lock storage lock = locks[lockId];

    require(lock.unlockTime <= block.timestamp, UnstakingManager__NotUnlockedYet());
    require(lock.claimedAt == 0, UnstakingManager__AlreadyClaimed());

    lock.claimedAt = block.timestamp;
    SafeERC20.safeTransfer(targetToken, lock.user, lock.amount);

    emit LockClaimed(lockId);
}
```

Calling *createLock()* does not reset **lock.claimedAt**.

```
unction createLock(address user, uint256 amount, uint256 unlockTime) external {
    SafeERC20.safeTransferFrom(targetToken, msg.sender, address(this), amount);

    uint256 lockId = nextLockId++;
    Lock storage lock = locks[lockId];

    lock.user = user;
    lock.amount = amount;
    lock.unlockTime = unlockTime;

    emit LockCreated(lockId, user, amount, unlockTime);
}
```

So, an attacker can call *claimLock()* on any **lockId** that doesn't exist, so that when that **lockId** is created, since **claimedAt != 0**, the lock will be considered claimed. This results in funds being frozen in *UnstakingManager*.

Since the *safeTransfer()* in *claimLock()* attempts to transfer zero **lock.amount** to **address(0)**, this attack only works for underlying tokens that are allowed to transfer to **address(0)**.

The POC is as follows.

```
function _transfer(address from, address to, uint256 value) internal {
    if (from == address(0)) {
        revert ERC20InvalidSender(address(0));
    }
    // if (to == address(0)) {
    //     revert ERC20InvalidReceiver(address(0));
    // }
    _update(from, to, value);
}
…
function test_POC() public {
    StakingVault newVault = new StakingVault(
        "Staked Test Token",
        "sTEST",
        IERC20(address(token)),
        address(this),
        REWARD_HALF_LIFE,
        14 days
    );
    UnstakingManager manager = newVault.unstakingManager();
    token.mint(address(this), 1000e18);
    token.approve(address(newVault), 1000e18);
    newVault.deposit(1000e18, address(this));

    manager.claimLock(0); // claim lockId that doesn't exist

    newVault.redeem(1000e18, address(this), address(this));
    vm.warp(block.timestamp + 14 days);
    manager.cancelLock(0); // revert UnstakingManager__AlreadyClaimed()
}
```

Recommended mitigation

It is recommended to disallow claiming of non-existent Locks in *claimLock()*.

```
    function claimLock(uint256 lockId) external {
        Lock storage lock = locks[lockId];

-       require(lock.unlockTime <= block.timestamp,
UnstakingManager__NotUnlockedYet());
+       require(lock.unlockTime <= block.timestamp && lock.unlockTime != 0,
UnstakingManager__NotUnlockedYet());
        require(lock.claimedAt == 0, UnstakingManager__AlreadyClaimed());

        lock.claimedAt = block.timestamp;
        SafeERC20.safeTransfer(targetToken, lock.user, lock.amount);

        emit LockClaimed(lockId);
    }
```

Team response

Fixed in commit [38f6df4](38f6df4).

Mitigation review

Verified, the recommendation has been implemented.

TRST-H-2 Precision loss of deltaIndex causes reward loss

- **Category:** Rounding issues
- **Source:** StakingVault.sol

- **Status:** Fixed

Description

When tracking rewards in *StakingVault*, **deltaIndex** is calculated to represent the reward for each share, and the **deltaIndex** is scaled by the token decimals (18).

```
function _accrueRewards(address _rewardToken) internal {
    RewardInfo storage rewardInfo = rewardTrackers[_rewardToken];

    uint256 elapsed = block.timestamp - rewardInfo.payoutLastPaid;
    if (elapsed == 0) {
        return;
    }

    uint256 unaccountedBalance = rewardInfo.balanceLastKnown -
rewardInfo.balanceAccounted;
    uint256 handoutPercentage = 1e18 - UD60x18.wrap(1e18 -
rewardRatio).powu(elapsed).unwrap();

    // {reward} = {reward} * D18{1} / D18
    uint256 tokensToHandout = (unaccountedBalance * handoutPercentage) / 1e18;

    uint256 supplyTokens = totalSupply();

    if (supplyTokens != 0) {
        // D18{reward/share} = {reward} * D18 / {share}
        uint256 deltaIndex = (tokensToHandout * uint256(10 ** decimals())) /
supplyTokens;

        // D18{reward/share} += D18{reward/share}
        rewardInfo.rewardIndex += deltaIndex;
        rewardInfo.balanceAccounted += tokensToHandout;
    }
    // @todo Add a test case for when supplyTokens is 0 for a while, the reward are
paid out correctly.

    rewardInfo.payoutLastPaid = block.timestamp;
    rewardInfo.balanceLastKnown = IERC20(_rewardToken).balanceOf(address(this)) +
rewardInfo.totalClaimed;
}
```

However, this scaling is not enough. For example, assume the reward token is USDC with 6 decimals, **unaccountedBalance = 100e6 USDC**, **rewardHalfLife = 3.5 days**, **rewardRatio = 2292153374868**.

The malicious user calls *_accrueRewards()* every 12 seconds (one block on Mainnet), **handoutPercentage = 27505493739421**, **tokensToHandout = 2750**.

As long as **totalSupply** is greater than 2750e18 (considering that the price of stToken is 1 USD, the APR has reached 100 / 7 * 365 / 2750 = 190%, and the real **totalSupply** will be higher, making the APR much lower than 190%), **deltaIndex** will be rounded down to 0, and **payoutLastPaid** will be updated to **block.timestamp**, **balanceAccounted** will be added with **tokensToHandout**, which means that these rewards are lost.

The POC is as follows.

```
function test_POC() public {
    vault.setRewardRatio(3.5 days);
    reward.mint(address(vault), 100 * 1e6);
    vm.warp(block.timestamp + 1);
    vault.poke(); // set balanceLastKnown
```

```
    _mintAndDepositFor(address(this), 2750 * 1e18 + 1); // deposit 2750 * 1e18 will
claim 2750 wei reward (deltaIndex = 1)
    vm.warp(block.timestamp + 12);
    vault.poke();
    address[] memory rewardTokens = new address[](1);
    rewardTokens[0] = address(reward);
    vault.claimRewards(rewardTokens);
    console2.log("balance %6e", reward.balanceOf(address(this))); // 0
    vm.warp(block.timestamp + 12);
    vault.poke();
    vault.claimRewards(rewardTokens);
    console2.log("balance %6e", reward.balanceOf(address(this))); // 0
}
```

Recommended mitigation

It is recommended to expand the scaling when calculating the **deltaIndex**, e.g., multiply **deltaIndex** by **1e18**.

```
diff --git a/contracts/staking/StakingVault.sol b/contracts/staking/StakingVault.sol
index 64a4fc0..5865784 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -226,7 +226,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable
{

         if (supplyTokens != 0) {
             // D18{reward/share} = {reward} * D18 / {share}
-            uint256 deltaIndex = (tokensToHandout * uint256(10 ** decimals())) /
supplyTokens;
+            uint256 deltaIndex = (tokensToHandout * uint256(10 ** decimals())) * 1e18
/ supplyTokens;

             // D18{reward/share} += D18{reward/share}
             rewardInfo.rewardIndex += deltaIndex;
@@ -250,7 +250,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable
{

         // Accumulate rewards by multiplying user tokens by index and adding on
unclaimed
         // {reward} = {share} * D18{reward/share} / D18
-        uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals());
+        uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals()) / 1e18;

         // {reward} += {reward}
         userRewardTracker.accruedRewards += supplierDelta;
```

Furthermore, to prevent reward tokens getting lost to rounding, even if they are dust, **tokensToHandout** can be calculated from the rounded **deltaIndex** before **rewardInfo.balanceAccounted** is incremented. This additional line needs more research since it is not obvious that the amount of tokens paid out to users can't be bigger than the updated **tokensToHandout**.

```
+            tokensToHandout = deltaIndex * supplyTokens / uint256(10 ** decimals()) /
1e18;
```

Team response

Fixed in commit bd264f2.

Mitigation review

Verified, the scaling has been implemented correctly, which makes the reward calculation compatible with a broader range of tokens.

Medium severity findings

TRST-M-1 *Folio.mint()* and *Folio.redeem()* lack slippage control

- **Category:** Slippage control issues
- **Source:** Folio.sol
- **Status:** Open

Description

When users mint or redeem fToken, assets are deposited or withdrawn proportionally based on the composition of the basket.

Since the asset composition of the basket changes as trades are filled, the assets deposited or withdrawn by the user may be not as expected.

For example, if there are 30 ETH and 1 BTC in the contract, and the user expects to redeem 3 ETH and 0.1 BTC, but due to an ongoing trade, the assets change to 20 ETH, 1BTC and 50 BNB, the user will redeem 2 ETH, 0.1 BTC and 5 BNB.

Also, for *mint()*, an attacker could inflate the share value through donation and cause the user to make a much larger deposit into the contract than expected.

Recommended mitigation

It is recommended to allow users to control slippage for deposits and withdrawals.

Team response

TBD

TRST-M-2 The *_poke()* call in *Folio.initialize()* will fail in the future

- **Category:** Overflow issues
- **Source:** Folio.sol
- **Status:** Open

Description

When the user creates a Folio, in *Folio.initialize()*, the **folioFee** is set and then *_poke()* is called. Eventually, the **_pendingFeeShares** are calculated in *_getPendingFeeShares()*.

```
    _setFolioFee(_additionalDetails.folioFee);
    …
    _poke();
…
function _poke() internal {
    if (lastPoke == block.timestamp) {
        return;
    }

    pendingFeeShares = _getPendingFeeShares();
    lastPoke = block.timestamp;
}
...
function _getPendingFeeShares() internal view returns (uint256 _pendingFeeShares) {
    _pendingFeeShares = pendingFeeShares;
```

```
    uint256 supply = super.totalSupply() + _pendingFeeShares + daoPendingFeeShares +
feeRecipientsPendingFeeShares;
    uint256 elapsed = block.timestamp - lastPoke;

    // {share} += {share} * D18 / D18{1/s} ^ {s} - {share}
    _pendingFeeShares += (supply * SCALAR) / UD60x18.wrap(SCALAR -
folioFee).powu(elapsed).unwrap() - supply;
}
```

The problem here is that since the initial **lastPoke** is 0, the **elapsed** time will be very large, which makes *powu()* return 0 at a future timestamp, and lead to division by 0.

After testing, *powu()* returns 0 when the timestamp is greater than **1885685839**, which means that starting October 3rd 2029, Folio creation will fail.

Recommended mitigation

Instead of calling *_poke()* in *Folio.initialize()*, it is recommended to just set **lastPoke** to **block.timestamp**.

```
@@ -124,7 +126,7 @@ contract Folio is
             basket.add(address(_basicDetails.assets[i]));
         }

-        _poke();
+        lastPoke = block.timestamp;
         _mint(_creator, _basicDetails.initialShares);
         _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
     }
```

Team response

TBD

TRST-M-3 Calling *approve()* on missing return values tokens will revert

- **Category:** ERC20 compatibility issues
- **Source:** StakingVault.sol, UnstakingManager.sol
- **Status:** Open

Description

The protocol calls *IERC20.approve()* when it approves the contract to spend the tokens.

```
function approve(address spender, uint256 value) external returns (bool);
```

The *IERC20.approve()* call expects a bool-type value to be returned, but for tokens such as USDT, their *approve()* call does not return any value, which causes their *approve()* call to revert.

For *StakingVault* contracts that support such tokens, the user can only withdraw tokens when the **unstakingDelay** is 0, and for *UnstakingManager* contracts, the user cannot cancel the Lock.

Recommended mitigation

It is recommended to use *SafeERC20.forceApprove()*.

```
diff --git a/contracts/staking/StakingVault.sol b/contracts/staking/StakingVault.sol
index 64a4fc0..b037ba0 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -91,7 +91,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable {
             // Burn the shares first.
             _burn(_owner, _shares);

-            IERC20(asset()).approve(address(unstakingManager), _assets);
+            SafeERC20.forceApprove(IERC20(asset()), address(unstakingManager),
_assets);
             unstakingManager.createLock(_receiver, _assets, block.timestamp +
unstakingDelay);

             emit Withdraw(_caller, _receiver, _owner, _assets, _shares);
diff --git a/contracts/staking/UnstakingManager.sol
b/contracts/staking/UnstakingManager.sol
index a5d8ee3..c249542 100644
--- a/contracts/staking/UnstakingManager.sol
+++ b/contracts/staking/UnstakingManager.sol
@@ -52,7 +52,7 @@ contract UnstakingManager {
         require(lock.user == msg.sender, UnstakingManager__Unauthorized());
         require(lock.claimedAt == 0, UnstakingManager__AlreadyClaimed());

-        targetToken.approve(address(vault), lock.amount);
+        SafeERC20.forceApprove(targetToken, address(vault), lock.amount);
         vault.deposit(lock.amount, lock.user);

         emit LockCancelled(lockId);
```

Team response

TBD

TRST-M-4 Removing reward tokens may cause users to lose rewards

- **Category:** Logical issues
- **Source:** StakingVault.sol
- **Status:** Open

Description

When a reward token is removed with *StakingVault.removeRewardToken()*, users can no longer accrue the rewards. They can only claim rewards that have already been accrued with *claimRewards()*.

```
    function removeRewardToken(address _rewardToken) external onlyOwner {
        disallowedRewardTokens[_rewardToken] = true;

        if (!rewardTokens.remove(_rewardToken)) {
            revert Vault__RewardNotRegistered();
        }

        delete rewardTrackers[_rewardToken];

        emit RewardTokenRemoved(_rewardToken);
    }
```

Accrual only happens in *_accrueRewards()* which is not reachable for a removed reward token. A consequence of this is that reward tokens are stuck once the token is removed. Even if a reward token has been accrued globally, an inactive user will not be able to earn these rewards if the reward token is removed.

Recommended mitigation

Due to the need to remove malicious reward tokens, it is recommended not to make any changes, but just notify users there is a final opportunity to cash out in the next X amount of time.

Team response

TBD

TRST-M-5 Price with 18 decimals precision is not compatible with all tokens

- **Category:** Integration issues
- **Source:** Folio.sol
- **Status:** Open

Description

The protocol plans to support tokens with up to 27 decimals, but prices with 18 decimals precision are not compatible with these plans.

Consider that the protocol intends to sell 27-decimal weirdBTC tokens to buy 8-decimal WBTC tokens. In terms of value, 1e8 WBTC == 1e27 weirdBTC, and 1 wei BTC is worth 1e19 weirdBTC, but in the protocol, since the price is 18 decimals precision, the price will be 0.1, which cannot be represented.

In fact, as the 27 decimal token decreases in value, 1e8 WBTC corresponds to more than 1e27 of the 27 decimal token, this difference can be orders of magnitude.

Recommended mitigation

It is recommended to increase the decimal precision of prices. However, even with increased precision, if there are no bounds around the USD value of the supported tokens, it is still possible to construct the same scenario.

Team response

TBD

TRST-M-6 Read-only reentrancy in *Folio.toAssets()*

- **Category:** Reentrancy issues
- **Source:** Folio.sol
- **Status:** Open

Description

In *Folio.bid()*, the *bidCallback()* is executed when the protocol has sent out sell tokens, but not yet received buy tokens.

Calling the read-only function *Folio.toAssets()* in *bidCallback()* would return lower *Folio* assets composition due to asset reduction.

If there are external protocols that rely on the results of *Folio.toAssets()*, the read-only reentrancy can be used in an attack.

Recommended mitigation

It is recommended that the external protocol first calls a *Folio* function that reverts on reentrancy before relying on the result of *Folio.toAssets()*, e.g. *Folio.distributeFees()*. The *Folio* itself does not need to be changed.

Team response

TBD

TRST-M-7 Folio is not compatible with Fee-on-Transfer tokens

- **Category:** ERC20 compatibility issues
- **Source:** Folio.sol, FolioDeployer.sol
- **Status:** Open

Description

Folio aims to support Fee-on-Transfer tokens, however there are three instances of transfers in *Folio* and *FolioDeployer* that introduce issues.

Folio.mint() transfers tokens from the user into the *Folio*.

```
for (uint256 i; i < assetLength; i++) {
    if (_amounts[i] != 0) {
        SafeERC20.safeTransferFrom(IERC20(_assets[i]), msg.sender, address(this),
_amounts[i]);
    }
}
```

If the token charges a fee, the amount that the *Folio* receives is less than expected, allowing the user to mint *fToken* without providing their fair share of underlying tokens.

In *Folio.bid()*, the same issue occurs, where the amount of **buy** tokens received can be less than intended.

```
} else {
    trade.buy.safeTransferFrom(msg.sender, address(this), boughtAmt);
}
```

Finally, when a new *Folio* is created, the amount of tokens that is transferred to the *Folio* in *FolioDeployer.deployFolio()* can be less than expected.

```
for (uint256 i; i < basicDetails.assets.length; i++) {
    IERC20(basicDetails.assets[i]).safeTransferFrom(msg.sender, address(folio),
basicDetails.amounts[i]);
```

```
}
```

Unlike the first two instances, the token transfer in *FolioDeployer.deployFolio()* does not pose a security risk and can be acknowledged.

Recommended mitigation

The code needs to be refactored such that only the received tokens are considered, not the amount that is transferred. This is not straightforward, since currently the amount to transfer is calculated first, and then it is transferred. It is not possible to calculate the amount to transfer from the intended received amount without token internals.

Team response

TBD

TRST-M-8 Donation attack allows to buy assets at decayed trade prices

- **Category:** Logical issues
- **Source:** Folio.sol
- **Status:** Open

Description

The protocol can open Dutch trades where the price of the sold assets decays over time. When only dust amounts can be traded, it may not be profitable to execute the trade, and so the price continues to decay, which could cause the following attack scenario:

1. **TRADE_PROPOSER** approves and starts trade for 10 WBTC, price is allowed to decay from 200,000 USDC to 30,000 USDC.

2. Depositors pull out, there remains only 0.1 WBTC in the protocol (which is owned by the attacker).

3. Attacker trades 0.0999 WBTC, only 0.0001 remains, no one completes the trade for a while, price drops to 30,000 USDC.

4. Victim wants to make a deposit.

5. Attacker sandwiches with:

   a. donate 4 WBTC.

   b. victim calls *mint()* and sends 4 WBTC (or any other non-dust amount).

   c. buy 8 WBTC with 240,000 USDC

The attacker ended up buying 4 WBTC for 120,000 USDC, which is below market price.

Recommended mitigation

One possible direction is to close trades when there remains only dust in the contract or when the trade's remaining **sellAmount** is only dust. This can be part of the logic in *bid()* and also a policy for the **PRICE_CURATOR** to follow.

The solution for this issue must be in sync with the other mitigations that are applied to the trading mechanism.

Team response

TBD

TRST-M-9 Assets added at trading may be sold at decayed trade price in another trade

- **Category:** Logical issues
- **Source:** Folio.sol
- **Status:** Open

Description

The protocol can open Dutch trades where the price of the sold assets decays over time. When the asset balance in the protocol reaches 0, the asset is removed from the basket, however the trade is not closed and the price still decays.

If there are parallel trades in progress, and a trade that buys that asset is active, the asset can be re-added to the basket, where an attacker can buy the asset at the decayed price. An example scenario follows:

1. **TRADE_PROPOSER** approves and starts trade A for 10 WBTC, price is allowed to decay from 200,000 USDC to 30,000 USDC

2. Depositors pull out, there remains only 5 WBTC in the protocol.

3. User trades the 5 WBTC, and the trade will not be closed, price drops to 30,000 USDC.

4. **TRADE_PROPOSER** approves and starts trade B for 200000 USDC, user trades it with 2 WBTC.

5. The attacker trades trade A, using 60000 USDC to trade 2 WETH.

Recommended mitigation

One possible direction is to close trades when the asset balance is 0. This can be part of the logic in *bid()* and also a policy for the **PRICE_CURATOR** to follow.

The solution for this issue must be in sync with the other mitigations that are applied to the trading mechanism.

Team response

TBD

TRST-M-10 Fixed trade amounts can lead to unintended Folio composition

- **Category:** Logical issues
- **Source:** Folio.sol
- **Status:** Open

Description

When approving trades, the **sellAmount** can be specified. As users mint and redeem, the percentage of **sellAmount / balance** changes, which results in an unintended composition of the *Folio* when the trade is completed.

Consider that the current *Folio* total supply is 10,000, and the protocol's asset composition is 100 BTC, 1,000 ETH, and 100,000 USDC. The protocol is intended to trade 50,000 USDC for 100 BNB, so that 1 Folio share will be composed of 0.01 BTC, 0.1 ETH, 5 USDC, and 0.01 BNB.

However, the users redeem 2000 fToken, leaving 80 BTC, 800 ETH, and 80,000 USDC in the contract, and when 50,000 USDC is traded for 100 BNB, 1 Folio will be composed of 0.01 BTC, 0.1 ETH, 3 USDC, and 0.0125 BNB, which is unintended.

Recommended mitigation

It is an important property of a token portfolio that the governance has control of the ratio of its constituents. There is no straightforward solution that extends or slightly modifies the current functionality and that can be recommended within the scope of this audit. Instead, it is necessary to evaluate the trading mechanism from the ground up, and to sync the solution with all findings related to trading (TRST-M-8 and TRST-M-9). Eventually, a possible solution could include a trading proposal to change the portfolio allocations, and additional rebalancing functionality could be introduced to allow desired long-term exposure.

Team response

TBD

TRST-M-11 Folio may suffer from inflated shares when initial shares are redeemed

- **Category:** Logical issues
- **Source:** Folio.sol
- **Status:** Open

Description

When the deployer creates the *Folio*, they provide funds and mint the initial shares. Once these initial shares are redeemed, the *Folio* may suffer from inflated share value.

1. Consider that the deployer provides funds and mints initial shares, and Alice also deposits funds into the Folio and mints shares, after which the deployer redeems the initial shares.
2. Alice can redeem all shares except 1 wei and donate funds to increase the value of 1 wei shares.
3. As a consequence, all rounding operations involving shares (minting fees, demurrage fees) will have large errors. In fact, fee recipients may not receive any fees at all due to rounding down. Also, depositors may lose all the deposited funds to the **mintingFee** due to rounding up the fee shares.

Recommended mitigation

It is recommended to mint a small amount of dead shares in the initial minting. The current mechanism can be extended by requiring a minimum amount of **initialShares** and minting some of them to **address(0)**.

Team response

TBD

Low severity findings

TRST-L-1 *setFeeRecipient()* and *setDefaultFeeNumerator()* do not distribute pending fees

- **Category:** Logical issues
- **Source:** FolioDAOFeeRegistry.sol
- **Status:** Open

Description

FolioDAOFeeRegistry._setTokenFee() calls *Folio.distributeFees()* to distribute the pending fees before changing the DAO fee for a specific fToken.

However, *setFeeRecipient()* and *setDefaultFeeNumerator()* do not calls *Folio.distributeFees()* before changing **feeRecipient** and **defaultFeeNumerator**. Specifically for **defaultFeeNumerator** this is an issue because pending fees will accumulate at the new fee rate.

Recommended mitigation

Since it is not possible to iterate over all Folios to distribute fees, one option is to apply new fee logic:

1. When creating a Folio, set **fTokenFeeNumerator** to **defaultFeeNumerator** for that Folio in *_setTokenFee()*.
2. *getFeeDetails()* always returns **fTokenFeeNumerator[fToken]**.
3. Changing the **defaultFeeNumerator** will only be applied to newly created Folios.

Team response

TBD

TRST-L-2 StakingVault does not register new rewards when rewards have been accrued in the same block

- **Category:** Logical issues
- **Source:** StakingVault.sol
- **Status:** Open

Description

When rewards have already been accrued in the same block, *StakingVault._accrueRewards()* returns early, and this means **rewardInfo.balanceLastKnown** is not updated.

It should be expected that when rewards are sent to *StakingVault*, and the global reward tracker is updated, they start to accrue. However, with the current implementation, the global reward tracker needs to be updated at a future timestamp.

Recommended mitigation

New rewards should be registered even when no time has passed since the last reward accrual and **elapsed = 0**.

```
diff --git a/contracts/staking/StakingVault.sol b/contracts/staking/StakingVault.sol
index 64a4fc0..bc9c8c5 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -210,13 +210,15 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes,
Ownable {

     function _accrueRewards(address _rewardToken) internal {
         RewardInfo storage rewardInfo = rewardTrackers[_rewardToken];
+        uint256 balanceLastKnown = rewardInfo.balanceLastKnown;
+        rewardInfo.balanceLastKnown = IERC20(_rewardToken).balanceOf(address(this)) +
rewardInfo.totalClaimed;

         uint256 elapsed = block.timestamp - rewardInfo.payoutLastPaid;
         if (elapsed == 0) {
             return;
         }

-        uint256 unaccountedBalance = rewardInfo.balanceLastKnown -
rewardInfo.balanceAccounted;
+        uint256 unaccountedBalance = balanceLastKnown - rewardInfo.balanceAccounted;
         uint256 handoutPercentage = 1e18 - UD60x18.wrap(1e18 -
rewardRatio).powu(elapsed).unwrap();

         // {reward} = {reward} * D18{1} / D18
@@ -235,7 +237,6 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable
{
         // @todo Add a test case for when supplyTokens is 0 for a while, the reward
are paid out correctly.

         rewardInfo.payoutLastPaid = block.timestamp;
-        rewardInfo.balanceLastKnown = IERC20(_rewardToken).balanceOf(address(this)) +
rewardInfo.totalClaimed;
     }
```

Team response

TBD

TRST-L-3 Rebasing tokens can introduce rounding edge cases

- **Category:** Integration issues
- **Source:** Folio.sol
- **Status:** Open

Description

It is a known issue with rebasing tokens that the **amount** that is passed to *transfer()* / *transferFrom()* can deviate a few wei from the received amount.

In the case of *Folio.mint()*, the *Folio* can receive a few wei less than expected, effectively rounding in favor of the user. The same can happen in *Folio.redeem()* where the user can receive a few wei more than expected.

Depending on how the rebasing token is implemented, there could also be an issue in *Folio.bid()* if the *Folio* tries to transfer all of its balance but leaves a few wei due to rounding. Since trades can only remove tokens from the basket if the token balance can reach zero, a rebasing token that rounds down the transfer amount and that makes it impossible to reach a zero balance will break this functionality.

```
if (trade.sell.balanceOf(address(this)) == 0) {
    basket.remove(address(trade.sell));
}
```

Recommended mitigation

Due to the lack of a standard for rebasing tokens, a variety of edge cases are possible. The notion that *balanceOf()* and transferred amounts can be off due to rounding breaks the fundamental assumptions of the *Folio* and requires a refactoring.

It is possible to restrict compatibility to a subset of rebasing tokens by defining invariants in the protocol documentation, but this would require internal knowledge of rebasing tokens that most *Folio* users might not have. Another option is to restrict compatibility to specific rebasing tokens.

Team response

TBD

TRST-L-4 *Folio.distributeFees()* can lead to decreased totalSupply and effectively make *mint()* round in favor of the user

- **Category:** Rounding issues
- **Source:** Folio.sol
- **Status:** Open

Description

Due to rounding down the shares that are minted to each fee recipient, the total shares minted to the fee recipients can be less than **feeRecipientsPendingFeeShares**. As a consequence, the call to *distributeFees()* can make **totalSupply** decrease.

A user that has minted **fToken** in the same block before the fee distribution has paid for the **shares** based on an ERC20 amount per share that is too low. Once fees are distributed, the **totalSupply** drops, and so shares become more valuable. In short, the downward rounding for the fee recipient shares allows users to mint shares, distribute fees, and immediately observe an increase in their share value.

Recommended mitigation

Shares for the fee recipients that are lost due to rounding should be minted to the Reserve DAO.

```
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..fc90d38 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -306,14 +306,10 @@ contract Folio is
         _poke();
         // pendingFeeShares is up-to-date

-        // DAO
-        (address recipient, , ) = daoFeeRegistry.getFeeDetails(address(this));
-        _mint(recipient, daoPendingFeeShares);
-        daoPendingFeeShares = 0;
```

```
-
         // Fee recipients
         uint256 _feeRecipientsPendingFeeShares = feeRecipientsPendingFeeShares;
         feeRecipientsPendingFeeShares = 0;
+        uint256 feeRecipientsTotal;

         uint256 len = feeRecipients.length;
         for (uint256 i; i < len; i++) {
@@ -321,7 +317,13 @@ contract Folio is
             uint256 shares = (_feeRecipientsPendingFeeShares *
feeRecipients[i].portion) / SCALAR;

             _mint(feeRecipients[i].recipient, shares);
+            feeRecipientsTotal += shares;
         }
+
+        // DAO
+        (address recipient, , ) = daoFeeRegistry.getFeeDetails(address(this));
+        _mint(recipient, daoPendingFeeShares + _feeRecipientsPendingFeeShares -
feeRecipientsTotal);
+        daoPendingFeeShares = 0;
     }
```

Team response

TBD

TRST-L-5 Folio becomes unusable when totalSupply drops to zero

- **Category:** Logical issues
- **Source:** Folio.sol
- **Status:** Open

Description

Folio.toAssets() does not consider the case when **totalSupply = 0**, which means once the last user has redeemed their shares, *toAssets()* reverts due to division by zero and the *Folio* can no longer be used.

```
uint256 assetBal = IERC20(_assets[i]).balanceOf(address(this));


// {tok} = {share} * {tok} / {share}
_amounts[i] = Math.mulDiv(shares, assetBal, _totalSupply, rounding);
```

Recommended mitigation

It is possible to define a *Folio* such that when the **totalSupply** is zero, constituents can be provided at an equal ratio to mint new shares.

To prevent share inflation attacks (TRST-M-11), it is also recommended to mint dead shares and this will eliminate the **totalSupply = 0** case. If the approach with dead shares is implemented, no further change is necessary.

Team response

TBD

TRST-L-6 Successive governance actions can deflate shares value and cause overflow

- **Category:** Overflow issues
- **Source:** Folio.sol
- **Status:** Open

Description

Repeated governance intervention can cause the shares supply to grow relative to the underlying value of the *Folio* constituents. There is no limit to the deflation in share value and it can be achieved by a small number of actions taken by the governance:

1. Folio contains 1e18 asset A and 1 wei asset B, **totalSupply = 1e18**
2. Governance removes asset A
3. User provides 1e18 asset B and receives **1e18 * 1e18 = 1e36 shares**
4. Governance adds asset C, 1 wei asset C is donated, Governance removes asset B
5. Folio contains 1 wei asset C and 1e36 shares

Eventually, the calculations in *Folio* can overflow leading to a DOS of *mint()* and *redeem()*.

```
        _pendingFeeShares += (supply * 1e18) / UD60x18.wrap(1e18 -
folioFee).powu(elapsed).unwrap() - supply;
```

Recommended mitigation

The issue can be eliminated by responsible governance action. Still, governance must be aware of the risk, and over time, governance intervention can compound. A possible code-level fix would be doing a share rebasing process when amount of circulating shares is getting close to overflow levels. For example, Governance would define 1e36 old shares to be worth 1 wei of new shares.

Team response

TBD

TRST-L-7 Donations during queued proposal to remove basket token can DOS minting

- **Category:** Griefing attacks
- **Source:** Folio.sol
- **Status:** Open

Description

Folio.removeFromBasket() can only be called by governance and is supposed to be used in emergencies, or to get rid of dust tokens that could not be fully traded.

Since the token will be removed, it can be expected that users won't mint new fToken if they need to provide a significant amount of a token whose value will be lost.

An attacker can abuse this situation and make a donation to deliberately increase the dust balance to a significant amount that discourages users from minting fToken during the time it takes for the proposal to be executed.

Recommended mitigation

To prevent the DOS vector, governance needs to provide a trade path to swap the token that is going to be removed for another token such that donations will be swiped and don't remain in the *Folio*. However, this requires foresight by the governance and may not be possible if the decision to remove a token needs to be made quickly. Overall, the risk can be considered acceptable and largely mitigated by a competent governance.

Team response

TBD

TRST-L-8 User rewards can be griefed by frequently accruing them

- **Category:** Rounding issues
- **Source:** StakingVault.sol
- **Status:** Open

Description

Since there currently exists no documentation for the maximum value of 1 wei of a reward token in *StakingVault*, it is theoretically possible that an attacker can frequently accrue rewards for a user and thereby cause a non-negligible loss of funds.

Suppose the reward token is WBTC with 8 decimals, and the price of 1 WBTC is $100,000. Then the value of 1 wei WBTC is $1e-3. By accruing rewards a thousand times, the user can lose up to $1 (and assuming uniform distribution, loses $0.5). For other tokens, which Folio aims to support, it could cause a more noticeable loss.

Recommended mitigation

It is recommended to provide guidelines for the maximum value of 1 wei for tokens that can safely be used as reward tokens in *StakingVault* and with the protocol more generally. Then, it would be possible to document the maximum loss formula based on stated assumptions.

Team response

TBD

Additional recommendations

TRST-R-1 price changes unevenly due to rounding of *exp()*

When **startPrice** is large relative to **endPrice**, the price changes unevenly due to rounding of *exp()*.

For example, if **startPrice** is 1e36, **endPrice** is 1e18, and **auctionLength** is 7 days, the price will stay at **endPrice** for the last 4.5 hours (when *exp()* returns 1 and 0), and due to rounding of *exp()*, the price will go directly from **startPrice * 2 / 1e18** to **startPrice * 1 / 1e18**, which makes it inapplicable to some auctions that start at very high prices.

It is recommended to limit **startPrice** to **endPrice** multiples.

TRST-R-2 Governance functions in Folio allow for reentrancy

Folio supports ERC777 tokens and therefore needs to protect against reentrancy. For user-facing functions, reentrancy guards are implemented. However, users can also access functions that are limited to the **owner** by waiting until a governance proposal becomes executable and executing that governance proposal in a callback.

The affected functions are *addToBaskset()*, *removeFromBasket()*, *setTradeDelay()* and *setAuctionLength()* and no exploitable scenario has been uncovered yet.

Given that the contracts are still in active development, it is important to present this attack vector and to rule out any such reentrancy issues in the future (an example of a critical attack involving reentrancy from approved proposal can be found in H-1 of the report for Optimism Bedrock). The four mentioned functions should also be protected with a reentrancy guard.

TRST-R-3 Reward ratio is calculated incorrectly for small half-lives

StakingVault.setRewardRatio() takes a **rewardHalfLife** parameter and calculates **rewardRatio = ln(2)/rewardHalfLife**. This **rewardRatio** is then applied via discrete compounding in *_accrueRewards()*. However, **rewardRatio** needs to be applied via continuous compounding which leads to a small error that increases for smaller half-lives.

The maximum error is for a half-life of 1 second where the percentage of rewards paid out after 1 second with discrete compounding is 69% in comparison to the intended 50%.

Possible solutions are to either implement a minimum half-life or to apply **rewardRatio** via continuous compounding.

TRST-R-4 Check that fee recipients do not contain duplicates

As an additional sanity check, it is recommended to ensure that **_feeRecipients** contains unique addresses. Note that this new check also covers the **feeRecipient == address(0)** case.

```
@@ -541,8 +543,10 @@ contract Folio is
             revert Folio__TooManyFeeRecipients();
         }

+        address previousRecipient;
+
         for (uint256 i; i < len; i++) {
-            if (_feeRecipients[i].recipient == address(0)) {
+            if (_feeRecipients[i].recipient <= previousRecipient) {
                 revert Folio__FeeRecipientInvalidAddress();
             }

@@ -551,6 +555,7 @@ contract Folio is
             }

             total += _feeRecipients[i].portion;
+            previousRecipient = _feeRecipients[i].recipient;
             feeRecipients.push(_feeRecipients[i]);
             emit FeeRecipientSet(_feeRecipients[i].recipient,
_feeRecipients[i].portion);
         }
```

TRST-R-5 Only governor should be configured as a Timelock executor

To prevent potential issues with proposals that call the **governor** itself, it is recommended that only the **governor** is granted the **EXECUTOR_ROLE**.

```
diff --git a/contracts/folio/FolioDeployer.sol b/contracts/folio/FolioDeployer.sol
index cecff4f..7db1e54 100644
--- a/contracts/folio/FolioDeployer.sol
+++ b/contracts/folio/FolioDeployer.sol
@@ -144,7 +144,7 @@ contract FolioDeployer is IFolioDeployer, Versioned {
         address[] memory proposers = new address[](1);
         proposers[0] = governor;
         address[] memory executors = new address[](1);
-        // executors[0] = address(0);
+        executors[0] = governor;

         TimelockControllerUpgradeable timelockController =
TimelockControllerUpgradeable(payable(timelock));
diff --git a/contracts/governance/GovernanceDeployer.sol
b/contracts/governance/GovernanceDeployer.sol
index d93a1d7..f221651 100644
--- a/contracts/governance/GovernanceDeployer.sol
+++ b/contracts/governance/GovernanceDeployer.sol
@@ -53,7 +53,7 @@ contract GovernanceDeployer is Versioned {
         address[] memory proposers = new address[](1);
         proposers[0] = governor;
         address[] memory executors = new address[](1);
-        // executors[0] = address(0);
+        executors[0] = governor;

         TimelockControllerUpgradeable timelockController =
TimelockControllerUpgradeable(payable(timelock));
```

TRST-R-6 Only allow vault to call *createLock()*

UnstakingManager.createLock() can be called by anyone, even though the intended use is that it is called by the **vault**. This makes it possible to emit arbitrary **LockCreated** events that need to be filtered off-chain. Furthermore, there is no validation in *createLock()* that **unlockTime > 0** and **unlockTime == 0** prevents the lock from being claimed. *StakingVault* calls *createLock()* with **unlockTime > block.timestamp** and thus this is only an issue if *createLock()* is incorrectly called by a third-party.

Both concerns can be mitigated by restricting *createLock()* to the **vault**.

TRST-R-7 TotalClaimed contains dirty value after reward token removal

When a reward token is removed in *StakingVault.removeRewardToken()*, the global reward tracker struct for the token is deleted with **delete rewardTrackers[_rewardToken]**. However, the **totalClaimed** field is still written to *claimRewards()*. After the reward token has been removed, **totalClaimed** therefore starts from zero, which can break any integration that relies on this value.

One solution is to only increment **totalClaimed** if the reward token has not been removed yet.

```
+  if(!disallowedRewardTokens[_rewardToken]) rewardInfo.totalClaimed +=
claimableRewards;
-           rewardInfo.totalClaimed += claimableRewards;
```

Another solution is to not delete the global reward tracker at all. This allows integrations to read all reward tracker fields even after removal.

```
            revert Vault__RewardNotRegistered();
        }

-       delete rewardTrackers[_rewardToken];
-
        emit RewardTokenRemoved(_rewardToken);
    }
```

TRST-R-8 *FolioVersionRegistry.getLatestVersion()* reverts before the first folioDeployer is registered

When *FolioVersionRegistry.registerVersion()* has not been called yet, and **latestVersion** does not point to a valid deployer, calling *getLatestVersion()* reverts due to calling *version()* on **address(0)**. It is recommended to handle this case more gracefully and to revert with a descriptive error.

TRST-R-9 *Folio.getBidAmount()* does not consider available sell token amount

Folio.getBidAmount() converts an **amount** of sell tokens to buy tokens: However, there is no validation to ensure that this **sellAmount** is actually available (it is constrained by

trade.sellAmount and sell token balance). It should be assessed if the current behavior is intended or if **sellAmount** should be validated.

TRST-R-10 Handout percentage is rounded up

The calculation of **handoutPercentage** in *StakingVault._accrueRewards()* rounds up since *powu()* rounds down, and so the overall calculation rounds up.

```
uint256 handoutPercentage = 1e18 - UD60x18.wrap(1e18 -
rewardRatio).powu(elapsed).unwrap();
```

As a general rule, reward payouts should round against the user, however there is no uncovered impact to this wrong rounding direction.

TRST-R-11 Folio upgrades are inconvenient due to not allowing for data to be passed

FolioProxyAdmin.upgradeToVersion() does not allow to pass data, which means that an upgrade that requires initialization needs to execute two transactions in a batch. First, the upgrade, then the initialization. It is more convenient if data can be passed directly with the upgrade.

```
diff --git a/contracts/folio/FolioProxy.sol b/contracts/folio/FolioProxy.sol
index 735524f..7711870 100644
--- a/contracts/folio/FolioProxy.sol
+++ b/contracts/folio/FolioProxy.sol
@@ -20,7 +20,7 @@ contract FolioProxyAdmin is Ownable {
         versionRegistry = _versionRegistry;
     }

-    function upgradeToVersion(address proxyTarget, bytes32 versionHash) external
onlyOwner {
+    function upgradeToVersion(address proxyTarget, bytes32 versionHash, bytes memory
data) external onlyOwner {
         IFolioVersionRegistry folioRegistry = IFolioVersionRegistry(versionRegistry);

         if (folioRegistry.isDeprecated(versionHash)) {
@@ -29,7 +29,7 @@ contract FolioProxyAdmin is Ownable {

         address folioImpl = folioRegistry.getImplementationForVersion(versionHash);

-        ITransparentUpgradeableProxy(proxyTarget).upgradeToAndCall(folioImpl, "");
+        ITransparentUpgradeableProxy(proxyTarget).upgradeToAndCall(folioImpl, data);
     }
 }
```

TRST-R-12 Remove unused errors

```
diff --git a/contracts/interfaces/IFolioDAOFeeRegistry.sol
b/contracts/interfaces/IFolioDAOFeeRegistry.sol
index 9575fe3..311ea8e 100644
--- a/contracts/interfaces/IFolioDAOFeeRegistry.sol
+++ b/contracts/interfaces/IFolioDAOFeeRegistry.sol
```

```
@@ -7,7 +7,6 @@ interface IFolioDAOFeeRegistry {
     error FolioDAOFeeRegistry__InvalidFeeNumerator();
     error FolioDAOFeeRegistry__InvalidRoleRegistry();
     error FolioDAOFeeRegistry__InvalidCaller();
-    error FolioDAOFeeRegistry__RTokenAlreadySet();

     event FeeRecipientSet(address indexed feeRecipient);
     event DefaultFeeNumeratorSet(uint256 defaultFeeNumerator);
diff --git a/contracts/interfaces/IFolio.sol b/contracts/interfaces/IFolio.sol
index 116af71..83c151e 100644
--- a/contracts/interfaces/IFolio.sol
+++ b/contracts/interfaces/IFolio.sol
@@ -26,7 +26,6 @@ interface IFolio {

     // === Errors ===

-    error Folio__BasketAlreadyInitialized();
     error Folio__EmptyAssets();

     error Folio__FeeRecipientInvalidAddress();
```

TRST-R-13 Avoid storage reads

```
diff --git a/contracts/folio/FolioDAOFeeRegistry.sol
b/contracts/folio/FolioDAOFeeRegistry.sol
index f15d98d..3e92800 100644
--- a/contracts/folio/FolioDAOFeeRegistry.sol
+++ b/contracts/folio/FolioDAOFeeRegistry.sol
@@ -57,7 +57,7 @@ contract FolioDAOFeeRegistry is IFolioDAOFeeRegistry {
         }

         defaultFeeNumerator = feeNumerator_;
-        emit DefaultFeeNumeratorSet(defaultFeeNumerator);
+        emit DefaultFeeNumeratorSet(feeNumerator_);
     }

     function setTokenFeeNumerator(address fToken, uint256 feeNumerator_) external
onlyOwner {
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..7122ff3 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -593,7 +595,7 @@ contract Folio is
         }

         folioFee = _newFee;
-        emit FolioFeeSet(folioFee);
+        emit FolioFeeSet(_newFee);
     }

     function _setMintingFee(uint256 _newFee) internal {
@@ -602,7 +604,7 @@ contract Folio is
         }

         mintingFee = _newFee;
-        emit MintingFeeSet(mintingFee);
+        emit MintingFeeSet(_newFee);
     }

     function _setFeeRecipients(FeeRecipient[] memory _feeRecipients) internal {
@@ -643,7 +645,7 @@ contract Folio is
             revert Folio__InvalidTradeDelay();
         }
         tradeDelay = _newDelay;
-        emit TradeDelaySet(tradeDelay);
+        emit TradeDelaySet(_newDelay);
```

```
    }

    function _setAuctionLength(uint256 _newLength) internal {
@@ -652,7 +654,7 @@ contract Folio is
         }

         auctionLength = _newLength;
-        emit AuctionLengthSet(auctionLength);
+        emit AuctionLengthSet(_newLength);
     }
```

TRST-R-14 Shortcut rewards accrual if there are no rewards to accrue

```
diff --git a/contracts/staking/StakingVault.sol b/contracts/staking/StakingVault.sol
index 8f7f1d2..9ebfbfd 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -280,14 +280,15 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes,
Ownable {

         // D18{reward}
         uint256 deltaIndex = rewardInfo.rewardIndex -
userRewardTracker.lastRewardIndex;
+        if (deltaIndex > 0) {
+            // Accumulate rewards by multiplying user tokens by index and adding on
unclaimed
+            // {reward} = {share} * D18{reward} / {share} / D18
+            uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals()) / SCALAR;

-        // Accumulate rewards by multiplying user tokens by index and adding on
unclaimed
-        // {reward} = {share} * D18{reward} / {share} / D18
-        uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals()) / SCALAR;
-
-        // {reward} += {reward}
-        userRewardTracker.accruedRewards += supplierDelta;
-        userRewardTracker.lastRewardIndex = rewardInfo.rewardIndex;
+            // {reward} += {reward}
+            userRewardTracker.accruedRewards += supplierDelta;
+            userRewardTracker.lastRewardIndex = rewardInfo.rewardIndex;
+        }
     }
```

TRST-R-15 FolioDeployer and GovernanceDeployer should refer to contracts via interfaces

FolioDeployer and *GovernanceDeployer* make use of *Folio*, *FolioGovernor* and *TimelockControllerUpgradeable* to cast the proxy contracts. This is bad practice since it loads the respective contracts into the bytecode of the deployers. Instead, the contracts should be referred to via their interfaces.

TRST-R-16 Check that basket additions and removals are successful

It is best practice to verify that basket token additions and removals are successful. If a proposal gets stuck because a token is already added to the basket or already removed from the basket, the proposal can be cancelled by the guardian.

```
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..02fef82 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -143,12 +143,12 @@ contract Folio is
     // ==== Governance ====

     function addToBasket(IERC20 token) external onlyRole(DEFAULT_ADMIN_ROLE) {
-        basket.add(address(token));
+        require(basket.add(address(token)), "basket already contains token");
         emit BasketTokenAdded(address(token));
     }

     function removeFromBasket(IERC20 token) external onlyRole(DEFAULT_ADMIN_ROLE) {
-        basket.remove(address(token));
+        require(basket.remove(address(token)), "basket does not contain token");
         emit BasketTokenRemoved(address(token));
     }
```

TRST-R-17 Check that versionHash exists and handle error gracefully

If *FolioProxyAdmin.upgradeToVersion()* is called with a **versionHash** that does not have a deployer registered, the call to **deployments[versionHash].folioImplementation()** reverts due to calling the zero address.

It is recommended to check for this case and revert with a custom error.

```
diff --git a/contracts/folio/FolioProxy.sol b/contracts/folio/FolioProxy.sol
index 735524f..976ad0a 100644
--- a/contracts/folio/FolioProxy.sol
+++ b/contracts/folio/FolioProxy.sol
@@ -15,6 +15,7 @@ contract FolioProxyAdmin is Ownable {
     address public immutable versionRegistry; // @todo sync with version/upgrade
manager

     error VersionDeprecated();
+    error InvalidVersion();

     constructor(address initialOwner, address _versionRegistry) Ownable(initialOwner)
{
         versionRegistry = _versionRegistry;
@@ -26,6 +27,9 @@ contract FolioProxyAdmin is Ownable {
         if (folioRegistry.isDeprecated(versionHash)) {
             revert VersionDeprecated();
         }
+        if (address(folioRegistry.deployments(versionHash)) == address(0)) {
+            revert InvalidVersion();
+        }

         address folioImpl = folioRegistry.getImplementationForVersion(versionHash);

diff --git a/contracts/interfaces/IFolioVersionRegistry.sol
b/contracts/interfaces/IFolioVersionRegistry.sol
index 5949aea..3cc465a 100644
--- a/contracts/interfaces/IFolioVersionRegistry.sol
+++ b/contracts/interfaces/IFolioVersionRegistry.sol
@@ -15,4 +15,5 @@ interface IFolioVersionRegistry {
     function getImplementationForVersion(bytes32 versionHash) external view returns
(address folio);
```

```
     function isDeprecated(bytes32 versionHash) external view returns (bool);
+    function deployments(bytes32 versionHash) external view returns (IFolioDeployer);
 }
```

TRST-R-18 Only emit RewardsClaimed event if claimableRewards are greater than zero

StakingVault.claimRewards() allows to emit the **RewardsClaimed** event even when **claimableRewards = 0**, including the case when the reward token is not an old or current reward token. These events need to be filtered off-chain, and it is recommended to only emit the event when **claimableRewards > 0** which ensures that the reward token is either an old or current reward token.

```
diff --git a/contracts/staking/StakingVault.sol b/contracts/staking/StakingVault.sol
index 8f7f1d2..f29cc2d 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -182,15 +182,15 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes,
Ownable {

             uint256 claimableRewards = userRewardTracker.accruedRewards;

-            // {reward} += {reward}
-            rewardInfo.totalClaimed += claimableRewards;
-            userRewardTracker.accruedRewards = 0;
+            if (claimableRewards > 0) {
+                // {reward} += {reward}
+                rewardInfo.totalClaimed += claimableRewards;
+                userRewardTracker.accruedRewards = 0;

-            if (claimableRewards != 0) {
                 SafeERC20.safeTransfer(IERC20(_rewardToken), msg.sender,
claimableRewards);
-            }

-            emit RewardsClaimed(msg.sender, _rewardToken, claimableRewards);
+                emit RewardsClaimed(msg.sender, _rewardToken, claimableRewards);
+            }
         }
     }
```

TRST-R-19 Check that buy and sell token in a trade are different

It is possible to approve trades where **buy == sell**. This case can break the logic in *bid()* due to removing the token from the basket without adding it back. It is recommended to check that **buy != sell**, as equality is not a valid use case.

```
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..553d2f1 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -364,7 +364,7 @@ contract Folio is
             revert Folio__InvalidTradeId();
         }

-        if (address(sell) == address(0) || address(buy) == address(0)) {
+        if (address(sell) == address(0) || address(buy) == address(0) || address(buy)
== address(sell)) {
             revert Folio__InvalidTradeTokens();
```

```
        }
```

TRST-R-20 Check that trade has not already ended when killing it

In *Folio.killTrade()*, it is recommended to revert when **block.timestamp > trade.end && trade.end != 0** since in this case the trade has already been terminated or has ended.

```
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..1272b1a 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -508,8 +508,10 @@ contract Folio is
         if (!hasRole(TRADE_PROPOSER, msg.sender) && !hasRole(PRICE_CURATOR,
msg.sender)) {
             revert Folio__Unauthorized();
         }
+        if (trades[tradeId].end < block.timestamp && trades[tradeId].end != 0) {
+            revert Folio__TradeCannotBeKilled();
+        }

-        /// do not revert, to prevent griefing
         trades[tradeId].end = 1;
         emit TradeKilled(tradeId);
     }
diff --git a/contracts/interfaces/IFolio.sol b/contracts/interfaces/IFolio.sol
index 8854d89..cebf755 100644
--- a/contracts/interfaces/IFolio.sol
+++ b/contracts/interfaces/IFolio.sol
@@ -45,6 +45,7 @@ interface IFolio {
     error Folio__InvalidTradeId();
     error Folio__InvalidSellAmount();
     error Folio__TradeCannotBeOpened();
+    error Folio__TradeCannotBeKilled();
     error Folio__TradeCannotBeOpenedPermissionlesslyYet();
     error Folio__TradeNotOngoing();
     error Folio__InvalidPrices();
```

There is a comment indicating that *killTrade()* should not revert to prevent griefing. However, there is no known griefing vector as *killTrade()* is privileged and there is no impact to **TRADE_PROPOSER** and **PRICE_CURATOR** front-running each other.

TRST-R-21 Documentation improvements

```
diff --git a/contracts/Folio.sol b/contracts/Folio.sol
index 989345a..848acf6 100644
--- a/contracts/Folio.sol
+++ b/contracts/Folio.sol
@@ -304,7 +304,7 @@ contract Folio is

     function distributeFees() public nonReentrant {
         _poke();
-        // pendingFeeShares is up-to-date
+        // daoPendingFeeShares and feeRecipientsPendingFeeShares are up-to-date

         // DAO
         (address recipient, , ) = daoFeeRegistry.getFeeDetails(address(this));
@@ -349,7 +349,7 @@ contract Folio is
```

```
     /// @param startPrice D18{buyTok/sellTok} Provide 0 to defer pricing to price
curator
     /// @param endPrice D18{buyTok/sellTok} Provide 0 to defer pricing to price
curator
     /// @param ttl {s} How long a trade can exist in an APPROVED state until it can
no longer be OPENED
-    ///     (once opened, it always finishes). Accepts type(uint256).max .
+    ///     (once opened, it always finishes).
     ///     Must be longer than tradeDelay if intended to be permissionlessly
available.
     function approveTrade(
         uint256 tradeId,
@@ -655,7 +655,7 @@ contract Folio is
         emit AuctionLengthSet(auctionLength);
     }

-    /// @dev After: pendingFeeShares is up-to-date
+    /// @dev After: daoPendingFeeShares and feeRecipientsPendingFeeShares are up-to-
date
     function _poke() internal {
         if (lastPoke == block.timestamp) {
             return;diff --git a/contracts/staking/StakingVault.sol
b/contracts/staking/StakingVault.sol
index 8f7f1d2..e616bfa 100644
--- a/contracts/staking/StakingVault.sol
+++ b/contracts/staking/StakingVault.sol
@@ -41,7 +41,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable {

     struct RewardInfo {
         uint256 payoutLastPaid; // {s}
-        uint256 rewardIndex; // D18{reward}
+        uint256 rewardIndex; // D18+decimals{reward/share}
         //
         uint256 balanceAccounted; // {reward}
         uint256 balanceLastKnown; // {reward}
@@ -49,7 +49,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable {
     }

     struct UserRewardInfo {
-        uint256 lastRewardIndex; // D18{reward}
+        uint256 lastRewardIndex; // D18+decimals{reward/share}
         uint256 accruedRewards; // {reward}
     }

@@ -232,6 +232,7 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes, Ownable
{

             // If a deposit/withdraw operation gets called for another user we should
             // accrue for both of them to avoid potential issues
+            // This is important for accruing for "from" and "to" in a transfer.
             if (_receiver != _caller) {
                 _accrueUser(_caller, rewardToken);
             }
@@ -256,10 +257,10 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes,
Ownable {
         uint256 supplyTokens = totalSupply();

         if (supplyTokens != 0) {
-            // D18{reward} = D18 * {reward} * {share} / {share}
+            // D18+decimals{reward/share} = D18 * {reward} * decimals / {share}
             uint256 deltaIndex = (SCALAR * tokensToHandout * uint256(10 **
decimals())) / supplyTokens;

-            // D18{reward} += D18{reward}
+            // D18+decimals{reward/share} += D18+decimals{reward/share}
             rewardInfo.rewardIndex += deltaIndex;
             rewardInfo.balanceAccounted += tokensToHandout;
         }
```

```
@@ -278,11 +279,11 @@ contract StakingVault is ERC4626, ERC20Permit, ERC20Votes,
Ownable {
         RewardInfo memory rewardInfo = rewardTrackers[_rewardToken];
         UserRewardInfo storage userRewardTracker =
userRewardTrackers[_rewardToken][_user];

-        // D18{reward}
+        // D18+decimals{reward/share}
         uint256 deltaIndex = rewardInfo.rewardIndex -
userRewardTracker.lastRewardIndex;

         // Accumulate rewards by multiplying user tokens by index and adding on
unclaimed
-        // {reward} = {share} * D18{reward} / {share} / D18
+        // {reward} = {share} * D18+decimals{reward/share} / decimals / D18
         uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals()) / SCALAR;

         // {reward} += {reward}
```

Centralization risks

TRST-CR-1 Folio admin is fully trusted

The *Folio* admin is the owner of the *FolioProxyAdmin* contract as well as the role admin in the *Folio* implementation itself. It can remove and add tokens to the basket, approve trades (by granting itself the **TRADE_PROPOSER** role), configure the *Folio* settings and upgrade the *Folio* to whitelisted implementations. *Folio* admin is therefore fully trusted and by default is held by owner governance.

TRST-CR-2 Trade proposer is fully trusted

The **TRADE_PROPOSER** role is granted by the *Folio* admin and can call *Folio.approveTrade()* and *Folio.killTrade()*.

Since **TRADE_PROPOSER** can approve arbitrary trades, the role is fully trusted, and by default is held by the trading governance.

TRST-CR-3 Price curator

The **PRICE_CURATOR** role is granted by the *Folio* admin and is allowed to call *Folio.openTrade()* and *Folio.killTrade()*.

In *Folio.openTrade()*, **PRICE_CURATOR** can increase the trade start price by 100x and increase the trade end price arbitrarily (must not be higher than the trade start price). In case that **TRADE_PROPOSER** has set the start price to zero, **PRICE_CURATOR** can freely choose the start and end price (again with the caveat that end price must not be higher than start price).

Overall, **PRICE_CURATOR** has significant power to direct trade execution, but can't cause a loss of funds If properly restricted by **TRADE_PROPOSER**.

TRST-CR-4 Reserve DAO

The Reserve DAO is the entity that controls *FolioDAOFeeRegistry* and *FolioVersionRegistry*. Therefore, Reserve DAO is allowed to determine the *Folio* implementations that *Folio* governance can upgrade to. No upgrades can be performed by Reserve DAO itself.

Furthermore, Reserve DAO can set the DAO fee to up to 50% of the **mintingFee** and **folioFee**. It is important to note that Reserve DAO always receives at least 0.05% of all minted shares even if no **mintingFee** is configured.

TRST-CR-5 StakingVault owner is fully trusted

StakingVault inherits from *ERC20Votes* and it is used as the voting token in all governors associated with a given *Folio*. A *Folio* can be deployed with any *ERC20Votes* token, so these risks only apply when a *StakingVault* is deployed which is then used as the voting token in the *Folio* owner governor, *Folio* trading governor and *StakingVault* governor.

The owner of *StakingVault* can set the **unstakingDelay** up to a limit of 4 weeks, add reward tokens, remove reward tokens and set the reward ratio (also within safe bounds). By adding a malicious reward token, the call to *balanceOf()* in *_accrueRewards()* can be reverted such that users can't claim rewards and can't transfer the *StakingVault* token. This leads to a loss of funds in the underlying vault token and the reward tokens. It also interferes with the fact that the *ERC20Votes* token should be used in governance since it can no longer be transferred. However, votes can still be used and can be moved from one user to another with delegations.

Systemic risks

TRST-SR-1 External token risk

Since *Folio* is essentially a wrapper around a basket of external tokens, any risks in the external tokens propagate to *Folio* and the security of a *Folio* depends on the external tokens that it holds.

Folio has the ability to remove malicious or broken tokens that block the functionality, and it also protects against reentrancies. However, if there is an open trade, it is possible to siphon out another token given that one token becomes malicious or just loses its value.

Therefore, in general, one token can lead to a complete loss of all funds in the *Folio*.

Furthermore, *Folio* aims to be compatible with any token as long as it complies with the ERC20 standard, including rebasing tokens, tokens with transfer fees, and other uncommon properties. Since there are still limits to the tokens that are supported, the documentation needs to be checked for whether a certain token behavior is supported. Additionally, it is recommended for the project to prepare a whitelist of supported functionalities, and only calling tokens that implement a subset of those functionalities as supported.

TRST-SR-2 Governance can become inactive

The whole DTF protocol can deploy up to three different Governors: one Governor for the *Folio* owner, one Governor for *StakingVault*, and one Governor for the *Folio* trade proposer. Since the Governors have full control over the protocol, users need to be actively engaged in governance, or otherwise there is a risk for malicious proposals to pass or proposals not to reach the quorum percentage.