Tribe Turbo contest
Findings & Analysis Report

2022-04-19

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Tribe Turbo smart contract system written in Solidity. The audit contest took place between February 17—February 23 2022.

Wardens

30 Wardens contributed reports to the Tribe Turbo contest:

1. cmichel

2. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
3. gzeon
4. WatchPug (jtp and ming)
5. cccz
6. Picodes
7. Ruhum
8. 0xliumin
9. hyh
10. nascent (brock, 0xAndreas, and chris_nascent)
11. csanuragjain
12. IIIIIII
13. samruna
14. pauliax
15. catchup
16. Dravee
17. robee
18. kenta
19. defsec
20. asgeir
21. 0x1f8b
22. Tomio
23. 0v3rf10w

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 7 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 17 reports by wardens detailing issues that fall under the risk rating of LOW severity or non-critical. There were also 14 reports by wardens recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Tribe Turbo contest repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 1205 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] ERC4626 mint uses wrong `amount`

Submitted by cmichel, also found by 0xliumin, CertoraInc, Picodes, and Ruhum

> The docs/video say `ERC4626.sol` is in scope as its part of `TurboSafe`

The `ERC4626.mint` function mints `amount` instead of `shares`. This will lead to issues when the `asset <> shares` are not 1-to-1 as will be the case for most vaults over time. Usually, the asset amount is larger than the share amount as vaults receive asset yield. Therefore, when minting, `shares` should be less than `amount`. Users receive a larger share amount here which can be exploited to drain the vault assets.

```
function mint(uint256 shares, address to) public virtual returns
    amount = previewMint(shares); // No need to check for roundi

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), amount);
    _mint(to, amount);

    emit Deposit(msg.sender, to, amount, shares);

    afterDeposit(amount, shares);
}
```

Proof of Concept

Assume `vault.totalSupply() = 1000`, `totalAssets = 1500`

- call `mint(shares=1000)`. Only need to pay `1000` asset amount but receive `1000` shares => `vault.totalSupply() = 2000`, `totalAssets = 2500`.
- call `redeem(shares=1000)`. Receive `(1000 / 2000) * 2500 = 1250` amounts. Make a profit of `250` asset tokens.
- repeat until `shares <> assets` are 1-to-1

Recommended Mitigation Steps

In `deposit`:

```
function mint(uint256 shares, address to) public virtual returns
-     _mint(to, amount);
+     _mint(to, shares);
}
```

Alex the Entreprenerd (judge):

> The warden has identified what is most likely a small oversight, which would have drastic consequences in the internal accounting of the Vault. Because of impact, I agree with high severity.

> The sponsor has mitigated.

[H-02] TurboRouter: `deposit()`, `mint()`, `createSafeAndDeposit()` and `createSafeAndDepositAndBoost()` functions do not work

Submitted by cccz, also found by CertoraInc and WatchPug

The TurboRouter contract inherits from the ERC4626RouterBase contract. When the user calls the deposit, mint, createSafeAndDeposit and createSafeAndDepositAndBoost functions of the TurboRouter contract, the deposit and mint functions of the ERC4626RouterBase contract are called.

```
    function deposit(IERC4626 safe, address to, uint256 amount, uint
        public
        payable
        override
        authenticate(address(safe))
        returns (uint256)
    {
        return super.deposit(safe, to, amount, minSharesOut);
    }
    ...
    function deposit(
        IERC4626 vault,
        address to,
        uint256 amount,
        uint256 minSharesOut
    ) public payable virtual override returns (uint256 sharesOut) {
        if ((sharesOut = vault.deposit(amount, to)) < minSharesOut)
            revert MinAmountError();
        }
    }
```

The deposit and mint functions of the ERC4626RouterBase contract will call the deposit and mint functions of the TurboSafe contract. The TurboSafe contract inherits from the ERC4626 contract, that is, the deposit and mint functions of the ERC4626 contract will be called.

The deposit and mint functions of the ERC4626 contract will call the safeTransferFrom function. Since the caller is the TurboRouter contract, msg.sender will be the TurboRouter contract. And because the user calls the deposit, mint, createSafeAndDeposit, and createSafeAndDepositAndBoost functions of the TurboRouter contract without transferring tokens to the TurboRouter contract and approving the TurboSafe contract to use the tokens, the call will fail.

```
    function deposit(uint256 amount, address to) public virtual retu
        // Check for rounding error since we round down in previewDe
        require((shares = previewDeposit(amount)) != 0, "ZERO_SHARES

        // Need to transfer before minting or ERC777s could reenter.
        asset.safeTransferFrom(msg.sender, address(this), amount);

        _mint(to, shares);

        emit Deposit(msg.sender, to, amount, shares);

        afterDeposit(amount, shares);
    }
```

Proof of Concept

[TurboRouter.sol](#)

Recommended Mitigation Steps

In the deposit, mint, createSafeAndDeposit, and createSafeAndDepositAndBoost functions of the TurboRouter contract, add code for the user to transfer tokens and approve the use of tokens in the TurboSafe contract. For example:

TurboRouter.sol

```
+           IERC20(safe.asset).safeTransferFrom(msg.sender,address
+           IERC20(safe.asset).safeApprove(safe,amount);
```

```
        super.deposit(IERC4626(address(safe)), to, amount, minShares

    ...

    +         IERC20(safe.asset).safeTransferFrom(msg.sender,address(
    +         IERC20(safe.asset).safeApprove(safe,amount);
        super.mint(safe, to, shares, maxAmountIn);
```

[Joeysantoro (Tribe Turbo) disputed and commented]():

> Router uses Multicall and PeripheryPayments which can be combined to achieve the desired behaviors.

[Alex the Entreprenerd (judge) commented]():

> @Joeysantoro I don't quite understand your counter argument here for `createSafeAnd....` type functions.

> For Deposit and Mint, yes, you can create the safe, and then multicall the approvals, I agree with your counter, the functions don't need the extra approve calls.

> However for the functions that deploy a new safe, am not quite sure where the approval happens, see `createSafeAndDeposit` below:

```
    function createSafeAndDeposit(ERC20 underlying, address to,
        (TurboSafe safe, ) = master.createSafe(underlying);

        super.deposit(IERC4626(address(safe)), to, amount, minSh

        safe.setOwner(msg.sender);
    }
```

> I believe your counter argument could apply if you were deploying new vaults via Create2 so you could deterministically pre-approve the new safe, however in this case you are deploying a new safe, to an unpredictable address and then calling `deposit` on it. `deposit` will `safeTransferFrom` from the router to the vault and

> I can't quite see how this call won't fail since the router never gave allowance to the `safe`.

> Can you please clarify your counter argument for this specific function?

Joeysantoro (Tribe Turbo) commented:

> I was wrong, this issue is valid.

Alex the Entreprenerd (judge) increased severity to High and commented:

> Per the sponsor reply, I believe the finding to be valid. Impact is that the code doesn't work so I believe High Severity to be appropriate.

> Mitigation seems to be straightforward.

Please note: the following additional discussions took place approximately 3 weeks after judging and awarding were finalized. As such, this report will leave this finding in its originally assessed risk category as it simply reflects a snapshot in time.

Joeysantoro (Tribe Turbo) commented:

> imo this is not high risk because the router is a periphery contract. Its medium at best from a security perspective, but an important find within the context of the correctness of the code.

> To clarify, the issue only exists for createSafe... not deposit or mint for the reason I stated.

Alex the Entreprenerd (judge) commented:

> @Joeysantoro I think your perspective is valid and perhaps with more context, I would have indeed rated at a lower severity.

> My reasoning at the time was that because the code is broken, the severity should be high. On the other hand, we can also argue that the impact is minimal, as any call to those functions simply reverts, no safes with "wrong allowance" are set, and ultimately the impact is just some wasted gas.

> The bug doesn't cause a loss of funds nor bricks the protocol in any meaningful way (because this is just a periphery contract).

> I think you're right in your logic, at the time of judging I simply focused on how the code didn't work and thought that was reason to raise the severity

Medium Risk Findings (5)

[M-01] `ERC4626RouterBase.withdraw` should use a max shares out check

Submitted by cmichel, also found by CertoraInc and Picodes

> The docs/video say `ERC4626RouterBase.sol` is in scope as its part of `TurboRouter`

The `ERC4626RouterBase.withdraw` function withdraws the asset `amount` parameter by burning `shares`.

```
    function withdraw(
        IERC4626 vault,
        address to,
        uint256 amount,
        uint256 minSharesOut
    ) public payable virtual override returns (uint256 sharesOut) {
        // @audit-info from = msg.sender
        if ((sharesOut = vault.withdraw(amount, to, msg.sender)) < m
            revert MinAmountError();
        }
    }
```

It then checks that the burned shares `sharesOut` are not *less* than a `minSharesOut` amount. However, the user wants to be protected against burning *too many* shares for their specified `amount`, and therefore a `maxSharesBurned` amount parameter should be used.

The user can lose their entire shares due to the wrong check.

> this extends to `TurboRouter.withdraw`

Proof of Concept

User calls `Router.withdraw(amount=1100, minSharesOut=1000)` to protect against not burning more than `1000` shares for their `1100` asset amount. However, there's an exploit in the vault which makes the `sharesOut = 100_000`, the entire user's shares. The check then passes as it only reverts if `100_000 < 1000`.

Recommended Mitigation Steps

```
    function withdraw(
        IERC4626 vault,
        address to,
        uint256 amount,
-       uint256 minSharesOut
+       uint256 maxSharesIn
    ) public payable virtual override returns (uint256 sharesOut) {
-       if ((sharesOut = vault.withdraw(amount, to, msg.sender)) <
+       if ((sharesOut = vault.withdraw(amount, to, msg.sender)) >
            revert MinAmountError();
        }
    }
```

Also, rename the variable in `TurboRouter.withdraw`.

Joeysantoro (Tribe Turbo) disagreed with High severity and commented:

> This should be a medium severity issue. The function logic is correct, it's just not a useful check in its current state.

Joeysantoro (Tribe Turbo) resolved and commented:

> https://github.com/fei-protocol/ERC4626/pull/9

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I agree with both sides, ultimately the check doesn't provide protection against loss of value, as such medium severity is appropriate.

[M-02] Wrong implementation of `TurboSafe.sol#less()` may cause boosted record value in TurboMaster bigger than actual lead to `BoostCapForVault` and `BoostCapForCollateral` to be permanently occupied

Submitted by WatchPug, also found by cmichel and hyh

[TurboSafe.sol#L225-L236]

```
// Get out current amount of Fei debt in the Turbo Fuse Pool.
uint256 feiDebt = feiTurboCToken.borrowBalanceCurrent(address(th

// If our debt balance decreased, repay the minimum.
// The surplus Fei will accrue as fees and can be sweeped.
if (feiAmount > feiDebt) feiAmount = feiDebt;

// Repay Fei debt in the Turbo Fuse Pool, unless we would repay
if (feiAmount != 0) require(feiTurboCToken.repayBorrow(feiAmount

// Call the Master to allow it to update its accounting.
master.onSafeLess(asset, vault, feiAmount);
```

In the current implementation, when calling `less()` to withdraw Fei from the Vault and use it to repay debt, if the amount of Fei is bigger than the debt balance, the `onSafeLess` hook will use `feiDebt` as `The amount of Fei withdrawn from the Vault.`

As a result, `getTotalBoostedForVault[vault]` in TurboMaster will be larger than the actual total amount of Fei being used to boost the Vault.

Since the `Turbo Gibber` may impound some of the Safe's collateral and mint a certain amount of Fei and repay the Safe's Fei debt with the newly minted Fei. In that case, the Safe's debt balance can be less than the amount of Fei in Vault. Which

constitutes the precondition for the `less()` call to case the distortion of `getTotalBoostedForVault[vault]`.

Proof of Concept

Given:

- 1 WBTC = 100,000
- `collateralFactor` of WBTC = 0.6
- `getBoostCapForCollateral[WBTC]` = 300,000
- `getBoostCapForVault[vault0]` = 300,000
- Alice create Safe and deposit `10 WBTC` and Boost `300,000 Fei` to `vault0`
- Safe's debt = 300,000
- Safe's Fei in vault = 300,000

On master:

- getTotalBoostedForVault[vault0] = 300,000
- getTotalBoostedAgainstCollateral[WBTC] = 300,000

On safe:

- getTotalFeiBoostedForVault[vault0] = 300,000
- totalFeiBoosted = 300,000
- WBTC price drop to 50,000, `Turbo Gibber` impound `2 WBTC` and mint `100,000 Fei` to repay debt for Alice's Safe.
- Safe's debt = 200,000
- Safe's Fei in vault0 = 300,000
- Alice call `less()` withdraw `300,000 Fei` from Vault and repay `200,000` debt, in the hook: `master.onSafeLess(WBTC, vault0, 200,000)`
- Safe's debt = 0
- Safe's Fei in vault = 0

On master:

- getTotalBoostedForVault[vault0] = 100,000
- getTotalBoostedAgainstCollateral[WBTC] = 100,000

On Safe:

- getTotalFeiBoostedForVault[vault0] = 0
- totalFeiBoosted = 0
- Alice try deposit `20 WBTC` and Boost `300,000 Fei` will fail due to `BOOSTER_REJECTED`.

Recommended Mitigation Steps

Change to:

```
function less(ERC4626 vault, uint256 feiAmount) external nonReer
    // Update the total Fei deposited into the Vault proportiona
    getTotalFeiBoostedForVault[vault] -= feiAmount;

    unchecked {
        // Decrease the boost total proportionately.
        // Cannot underflow because the total cannot be less tha
        totalFeiBoosted -= feiAmount;
    }

    emit VaultLessened(msg.sender, vault, feiAmount);

    // Withdraw the specified amount of Fei from the Vault.
    vault.withdraw(feiAmount, address(this), address(this));

    // Call the Master to allow it to update its accounting.
    master.onSafeLess(asset, vault, feiAmount);

    // Get out current amount of Fei debt in the Turbo Fuse Pool
    uint256 feiDebt = feiTurboCToken.borrowBalanceCurrent(addres

    // If our debt balance decreased, repay the minimum.
    // The surplus Fei will accrue as fees and can be sweeped.
    if (feiAmount > feiDebt) feiAmount = feiDebt;

    // Repay Fei debt in the Turbo Fuse Pool, unless we would re
    if (feiAmount != 0) require(feiTurboCToken.repayBorrow(feiAm
}
```

transmissions11 (Tribe Turbo) commented:

> Good find

Alex the Entreprenerd (judge) commented:

> The warden identified a way to desynch the actual amounts of boostedFei for a vault and the system vs the amounts tracked in storage.
>
> Because of this discrepancy the availability of borrowable FEI in the system can be distorted, preventing new borrows.
>
> I do not believe this puts collateral at risk and also believe that the temporary "denial of borrowing" would be quickly fixed by raising caps.
>
> I want to commend the warden for finding a way to break the system invariants, while the system internal accounting has been broken, no meaningful leak of value, extended denial of service or funneling of funds happened.
>
> Liquidations can also still happen at the pool level.
>
> Because of these reasons, I agree with Medium Severity.

[M-03] Slurp can be frontrun with fee increase

Submitted by cmichel, also found by gzeon

The `TurboSafe.slurp` function fetches the current fee from the `clerk()`. This fee can be changed. The `slurp` transaction can be frontrun with a fee increase (specifically targeted for the vault or the asset) by the clerk and steal the vault yield that should go to the user.

Maybe the user would not want to `slurp` at the new fee rate and would rather wait as they expect the fees to decrease again in the future. Or they would rather create a new vault if the default fees are lower.

Recommended Mitigation Steps

Right now there's no good protection against this as the master can call `slurp` at any time. (They could even increase the fees to 100%, slurp, reset the fees.) This mechanic would need to be addressed first if mitigation and better user protection are desired.

Joeysantoro (Tribe Turbo) disputed and commented:

> Slurping is public, and fee increases will be behind governance timelocks. Users are sufficiently protected. Any more complete solution to this would dramatically increase the computational complexity of the architecture.

Alex the Entreprenerd (judge) commented:

> I have to agree with the Warden here that this type of Admin Privilege is present in the system and can be used to raise fees up to 100%.
>
> I believe protocol users could get stronger security guarantees by having a MAX_FEE hardcoded variable to ensure fees can never go above a certain threshold.
>
> I recommend the sponsor to publicly disclose this potential risk to protocol users, and given that I believe that the timelock will provide a good base security guarantee to which I'd recommend adding a MAX_FEE.
>
> Because the finding is contingent on malicious governance I believe medium severity to be appropriate.

[M-04] ERC4626 does not work with fee-on-transfer tokens

Submitted by cmichel

> The docs/video say `ERC4626.sol` is in scope as its part of `TurboSafe`

The `ERC4626.deposit/mint` functions do not work well with fee-on-transfer tokens as the `amount` variable is the pre-fee amount, including the fee, whereas the `totalAssets` do not include the fee anymore.

This can be abused to mint more shares than desired.

```
function deposit(uint256 amount, address to) public virtual retu
    // Check for rounding error since we round down in previewDe
    require((shares = previewDeposit(amount)) != 0, "ZERO_SHARES

    // Need to transfer before minting or ERC777s could reenter.
    asset.safeTransferFrom(msg.sender, address(this), amount);

    _mint(to, shares);

    emit Deposit(msg.sender, to, amount, shares);

    afterDeposit(amount, shares);
}
```

Proof of Concept

A `deposit(1000)` should result in the same shares as two deposits of `deposit(500)` but it does not because `amount` is the pre-fee amount. Assume a fee-on-transfer of `20%`. Assume current `totalAmount = 1000`, `totalShares = 1000` for simplicity.

- `deposit(1000) = 1000 / totalAmount * totalShares = 1000 shares`
- `deposit(500) = 500 / totalAmount * totalShares = 500 shares`. Now the `totalShares` increased by 500 but the `totalAssets` only increased by `(100% - 20%) * 500 = 400`. Therefore, the second `deposit(500) = 500 / (totalAmount + 400) * (newTotalShares) = 500 / (1400) * 1500 = 535.714285714 shares`.

In total, the two deposits lead to `35` more shares than a single deposit of the sum of the deposits.

Recommended Mitigation Steps

`amount` should be the amount excluding the fee, i.e., the amount the contract actually received. This can be done by subtracting the pre-contract balance from the post-contract balance. However, this would create another issue with ERC777 tokens.

Maybe `previewDeposit` should be overwritten by vaults supporting fee-on-transfer tokens to predict the post-fee `amount`. And do the shares computation on that, but then the `afterDeposit` is still called with the original `amount` and implementers need to be aware of this.

Joeysantoro (Tribe Turbo) disputed and commented:

> This is intended. Fee-on-transfer functions can be implemented in another base contract. This contract is only one implementation of the standard.

Alex the Entreprenerd (judge) commented:

> Given no context, I would side with the sponsor as the ERC4626 standard is meant to work with ERC20 Standard tokens.
>
> However, in bringing the ERC4626 mixin into scope, the sponsor's code has an explicit mention of ERC777 which does open up to the possibility of having fees on transfer.
>
> Because ultimately the warden didn't stretch the scope (as that happened because of the mixin), and the warden showed a way to provide leakage of value or denial of service exclusively if the mixin code is used in conjunction with a `feeOnTransfer` token.
>
> Given that the mixin code comments explicitly mention attempting to support non-standard tokens.
>
> I believe medium severity to be valid as any type of misbehavior is contingent on deploying an ERC4626 mixin with a `feeOnTransfer` Token.
>
> Given the circumstances, the best recommendation for developers is to use the mixin with ERC20 Standard Tokens.

[M-05] Gibber can take any amount from safes

Submitted by gzeon

Although Gibber is supposed to behind governance timelock, there are still significant “rug risk” when such privillaged user can remove all fund from a vault unconditionally.

Proof of Concept

[TurboSafe.sol#L335]

```
function gib(address to, uint256 assetAmount) external nonReentr
    emit SafeGibbed(msg.sender, to, assetAmount);

    // Withdraw the specified amount of assets from the Turbo Fu
    require(assetTurboCToken.redeemUnderlying(assetAmount) == 0,

    // Transfer the assets to the authorized caller.
    asset.safeTransfer(to, assetAmount);
}
```

Recommended Mitigation Steps

Limit gib to certain collateral ratio.

Joeysantoro (Tribe Turbo) disputed and commented:

> This is intended behavior. There will be an extended immutable timelock behind the gibber, so Turbo users will have notice to less and withdraw. The only scenario where they can’t is when the boosted strategy is insolvent, which is intended behavior.

Alex the Entreprenerd (judge) commented:

> While there’s something to be appreciated about a simple architecture, I believe that the flexibility of the Fuse Pool would allow to account for insolvency in a trustless way through the Fuse Liquidation code.

> Additionally, while the code may be put behind timelock, we can only review the code that is in scope and that we can prove behaves in a certain way.

> In this case, the warden has identified that the system admin can move funds at any time, without limitation.

> Because this type of exploit is contingent on a malicious admin, I believe Medium Severity to be appropriate.

> The sponsor intends on using an immutable timelock so for end users of the protocol I highly recommend to verify that statement.

Low Risk and Non-Critical Issues

For this contest, 17 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by warden **csanuragjain** received the top score from the judge.

The following wardens also submitted reports: **nascent**, **lllllll**, **hyh**, **samruna**, **pauliax**, **catchup**, **defsec**, **Ruhum**, **WatchPug**, **asgeir**, **cmichel**, **Dravee**, **robee**, **0x1f8b**, **kenta**, *and* **Picodes**.

[L-01] Missing checks on new Boost Cap

1. setBoostCapForVault function at TurboBooster.sol#L59 is missing checks to see if newBoostCap>getBoostCapForVault[vault]
2. This is required since vault might already be at old boost cap.
3. Setting lower boost cap would mean that boosting is already overflowed in this vault
4. In case if it is required to lower the boost cap then slurpAndLess function at TurboRouter.sol#L130 must be called to withdraw excess cap amount

Recommendation:

```
function setBoostCapForVault(ERC4626 vault, uint256 newBoostCap)
require(newBoostCap>getBoostCapForVault[vault], "Invalid boost")
    // Update the boost cap for the Vault.
    getBoostCapForVault[vault] = newBoostCap;

    emit BoostCapUpdatedForVault(msg.sender, vault, newBoostCap)
```

```
    }
```

[L-02] More funds extracted than required - Lose Interest

1. Debt can be paid using less function at TurboSafe.sol
2. But it was observed that User might call less function with higher feiAmount than required
3. In case if feiDebt<feiAmount, the difference feiAmount-feiDebt is kept as vault balance
4. This causes interest loss over these funds and even sweep can withdraw only after getTotalFeiBoostedForVault[vault]=0

Recommendation:

Calculate the feiDebt first and only withdraw the min(feiAmount,feiDebt) so that only required amount is withdrawn

[L-03] Missing zero address checks

1. setBooster, setClerk, setDefaultSafeAuthority at TurboMaster.sol are not checking whether the supplied address!=0

[N-01] Emit function called early

1. The emit function at multiple functions have been called before function end say emit VaultLessened at TurboSafe.sol#L220. This is incorrect since operation has not completed yet and function might revert based on condition ahead. Also this cause gas wastage

Joeysantoro (Tribe Turbo) acknowledged

Alex the Entreprenerd (judge) commented:

> Agree with the first finding, not super sure about mitigation as it would make the system more bloated.

> Second finding definitely worth investigating.

> Zero checks -> Agree by convention.

> Emit functions are being emitted early as a way to avoid reEntrancy, so I'm ambivalent on this.

> Overall the report was short and sweet, no particular formatting was needed and it looks good.

> Wish the warden put links to the findings to make it easier to check.

Alex the Entreprenerd (judge) commented:

> In judging, am also adding issue #9 ([L-04] Bypass Boosting cap set by Admin).

> 6/10

Alex the Entreprenerd (judge) commented:

> Bumping to 7 to make it the winner, 7/10.

Alex the Entreprenerd (judge) commented:

> After re-review I confirm 7/10. The simple formatting avoids confusion, and the Caps, Lose Interest, and #9 make the report unique.

Gas Optimizations

For this contest, 14 reports were submitted by wardens detailing gas optimizations. The report highlighted below by warden team **nascent** received the top score from the judge.

The following wardens also submitted reports: lllllll, WatchPug, CertoraInc, Dravee, gzeon, Picodes, catchup, csanuragjain, kenta, Tomio, 0v3rf10w, robee, *and* samruna.

[G-01] `slurp` SLOAD Gas Optimization

Severity: *Gas Optimization*
Likelihood: *High*
Status: {Not Submitted}
Scope: `slurp()`

There are two *sloads* of `getTotalFeiBoostedForVault[vault]` that can be gas golfed using an *mload* to reduce gas by `100 - 3`.

```
/// @notice Accrue any interest earned by the Safe in the Vault.
/// @param vault The Vault to accrue interest from, if any.
/// @dev Sends a portion of the interest to the Master, as determined by the Clerk.
function slurp(ERC4626 vault) external nonReentrant requiresLocalOrMasterAuth {
    // Ensure the Safe has Fei currently boosting the Vault.
    require(getTotalFeiBoostedForVault[vault] != 0, "NO_FEI_BOOSTED");

    // Compute the amount of Fei interest the Safe generated by boosting the Vault.
    uint256 interestEarned = vault.assetsOf(address(this)) - getTotalFeiBoostedForVault[vault];

    // Compute what percentage of the interest earned will go back to the Safe.
    uint256 protocolFeePercent = master.clerk().getFeePercentageForSafe(this, asset);

    // Compute the amount of Fei the protocol will retain as fees.
    uint256 protocolFeeAmount = interestEarned.mulWadDown(protocolFeePercent);

    // Compute the amount of Fei the Safe will retain as interest.
    uint256 safeInterestAmount = interestEarned - protocolFeeAmount;

    // Increase the boost total proportionately.
    totalFeiBoosted += safeInterestAmount;

    unchecked {
        // Update the total Fei held in the Vault proportionately.
        // Cannot overflow because the total cannot be less than a single Vault.
        getTotalFeiBoostedForVault[vault] += safeInterestAmount;
    }

    emit VaultSlurped(msg.sender, vault, protocolFeeAmount, safeInterestAmount);

    // If we have unaccrued fees, withdraw them from the Vault and transfer them to the Master.
    if (protocolFeeAmount != 0) vault.withdraw(protocolFeeAmount, address(master), address(this));

    // Call the Master to allow it to update its accounting.
    master.onSafeSlurp(asset, vault, safeInterestAmount);
}
```

[G-02] `save` SLOAD Gas Optimization

Severity: *Gas Optimization*
Likelihood: *Medium*
Status: {Not Submitted}
Scope: `save()`

There are two calls of `pool.oracle()` that can be gas golfed using an *mload* to reduce gas by `100 - 3`.

```
96      /// @notice Save a Safe (call less on owner's behalf to prevent liquidation).
97      /// @param safe The Safe to be saved.
98      /// @param vault The Vault to less from.
99      /// @param feiAmount The amount of Fei to less from the Safe.
100     function save(
101         TurboSafe safe,
102         ERC4626 vault,
103         uint256 feiAmount
104     ) external requiresAuth nonReentrant {
105         // Ensure the Safe is registered with the Master.
106         require(master.getSafeId(safe) != 0);
107
108         emit SafeSaved(msg.sender, safe, vault, feiAmount);
109
110         // Cache the Safe's collateral asset, saves a warm SLOAD below.
111         CERC20 assetTurboCToken = safe.assetTurboCToken();
112
113         // Get the Safe's asset's collateral factor in the Turbo Fuse Pool.
114         (, uint256 collateralFactor) = pool.markets(assetTurboCToken);
115
116         // Compute the value of the Safe's collateral. Rounded down to favor saving.
117         uint256 borrowLimit = assetTurboCToken
118             .balanceOf(address(safe))
119             .mulWadDown(assetTurboCToken.exchangeRateStored())
120             .mulWadDown(collateralFactor)
121             .mulWadDown(pool.oracle().getUnderlyingPrice(assetTurboCToken));
122
123         // Compute the value of the Safe's debt. Rounding up to favor saving them.
124         uint256 debtValue = feiTurboCToken.borrowBalanceCurrent(address(safe)).mulWadUp(
125             pool.oracle().getUnderlyingPrice(feiTurboCToken)
126         );
127
128         // Ensure the Safe's debt percentage is high enough to justify saving, otherwise revert.
129         require(
130             borrowLimit != 0 && debtValue.divWadUp(borrowLimit) >= minDebtPercentageForSaving,
131             "DEBT_PERCENT_TOO_LOW"
132         );
133
134         // Less the Fei from the Safe.
135         safe.less(vault, feiAmount);
136     }
```

[transmissions11 (Tribe Turbo) commented]:

> good finds, ty

[Alex the Entreprenerd (judge) commented]:

> G-01 Agree with finding, each time we're reading from memory we're saving 97 gas at the cost of 3 for the initial cache. -191

> G-02 Same idea -94

> 285 gas saved

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top