Tapioca DAO
Findings & Analysis Report

2023-11-16

Table of contents

- [H-06] BalancerStrategy `_withdraw` uses `BPT_IN_FOR_EXACT_TOKENS_OUT` which can be attack to cause loss to all depositors
- [H-07] Usage of `BalancerStrategy.updateCache` will cause single sided Loss, discount to Depositor and to OverBorrow from Singularity
- [H-08] `LidoEthStrategy._currentBalance` is subject to price manipulation, allows overborrowing and liquidations
- [H-09] `TricryptoLPStrategy.compoundAmount` always returns 0 because it's using staticall vs call
- [H-10] Liquidated USDO from BigBang not being burned after liquidation inflates USDO supply and can threaten peg permanently
- [H-11] TOFT `exerciseOption` can be used to steal all underlying erc20 tokens
- [H-12] TOFT `removeCollateral` can be used to steal all the balance
- [H-13] TOFT `triggerSendFrom` can be used to steal all the balance
- [H-14] All assets of (m)TapiocaOFT can be stealed by depositing to strategy cross chain call with 1 amount but maximum shares possible
- [H-15] Attacker can specify any `receiver` in `USD0.flashLoan()` to drain `receiver` balance
- [H-16] Attacker can block LayerZero channel due to variable gas cost of saving payload
- [H-17] Attacker can block LayerZero channel due to missing check of minimum gas passed
- [H-18] `multiHopSellCollateral()` will fail due to call on an invalid market address causing bridged collateral to be locked up
- [H-19] `twTAP.participate()` can be permanently frozen due to lack of access control on host-chain-only operations
- [H-20] `_liquidateUser()` should not re-use the same minimum swap amount out for multiple liquidation
- [H-21] Incorrect liquidation reward computation causes excess liquidator rewards to be given

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 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 outlined in this document, C4 conducted an analysis of the Tapioca DAO smart contract system written in Solidity. The audit took place between July 5—August 4 2023.

Wardens

134 Wardens contributed reports to the Tapioca DAO:

1. GalloDaSballo

2. peakbolt
3. KIntern_NA (duc and TrungOre)
4. windhustler
5. carrotsmuggler
6. 0x73696d616f
7. zzzitron
8. kaden
9. cergyk
10. ItsNio
11. rvierdiiev
12. Ack (plotchy, popular00, and igorline)
13. Vagner
14. dirk_y
15. xuwinnie
16. 0xStalin
17. Koolex
18. bin2chen
19. ladboy233
20. SaeedAlipoor01988
21. 0xRobocop
22. mojito_auditor
23. HE1M
24. Sathish9098
25. Madalad
26. 0xSmartContract
27. zzebra83
28. 0xWaitress
29. chaduke
30. unsafesol (Angry_Mustache_Man and devblixt)

31. n1punp
32. 0xnev
33. 0x007
34. c7e7eff
35. hunter_w3b
36. K42
37. JCK
38. minhtrng
39. cryptonue
40. ayeslick
41. 7e1e
42. erebus
43. LosPollosHermanos (jc1 and scaraven)
44. 0xTheC0der
45. BPZ (Bitcoinfever244, PrasadLak, and zinc42)
46. adeolu
47. glcanvas
48. zhaojie
49. Rolezn
50. naman1778
51. ReyAdmirado
52. dharma09
53. 0xfuje
54. Ruhum
55. jasonxiale
56. plainshift (thank_you and surya)
57. 0xrugpull_detector
58. jaraxxus
59. Udsen

60. wahedtalash77
61. mgf15
62. kodyvim
63. RedOneN
64. Kaysoft
65. kutugu
66. Nyx
67. ltyu
68. rokinot
69. 0xG0P1
70. andy
71. hassan-truscova
72. nadin
73. gizzy
74. Breeje
75. DelerRH
76. hendrik
77. Raihan
78. paweenp
79. Brenzee
80. vagrant
81. ak1
82. clash
83. marcKn
84. tsvetanovv
85. Limbooo
86. SY_S
87. petrichor
88. ybansal2403

89. 0xhex
90. 0xta
91. flutter_developer
92. SAQ
93. xfu
94. LeoS
95. lllllll
96. wangxx2026
97. dontonka
98. hack3r-0m
99. pks_
100. offside0011
101. CrypticShepherd
102. ACai
103. 0xSky
104. ck
105. iglyx
106. John_Femi
107. Walter
108. Deekshith99
109. SPYBOY
110. JP_Courses
111. Z3R0
112. audityourcontracts
113. l3r0ux
114. Tatakae (SaharDevep and mahdikarimi)
115. kaveyjoe
116. hals
117. SooYa

118. CryptoYulia
119. ABA
120. catwhiskeys
121. rumen
122. Oxadrii
123. Topmark
124. Oxsadeeq
125. TiesStevelink

This audit was judged by LSDan.

Final report assembled by PaperParachute.

Summary

The C4 analysis yielded an aggregated total of 159 unique vulnerabilities. Of these vulnerabilities, 60 received a risk rating in the category of HIGH severity and 99 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 79 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 19 reports 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 Tapioca DAO repository, and is composed of 68 smart contracts written in the Solidity programming language and includes 13291 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **IIIIIII-bot** from warden IIIIIII, generated the Automated Findings report and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (60)

[H-01] TOFT in (m)TapiocaOft contracts can be stolen by calling removeCollateral() with a malicious removeParams.market

Submitted by 0x73696d616f

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L190

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L516

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTMarketModule.sol#L230-L231

The `TOFT` available in the `TapiocaOFT` contract can be stolen when calling `removeCollateral()` with a malicious market.

Proof of Concept

`(m)TapiocaOFT` inherit `BaseTOFT`, which has a function `removeCollateral()` that accepts a market address as an argument. This function calls `_lzSend()` internally on the source chain, which then is forwarded to the destination chain by the relayer and calls `lzReceive()`.

`lzReceive()` reaches `_nonBlockingLzReceive()` in `BaseTOFT` and delegate calls to the `BaseTOFTMarketModule` on function `remove()`. This function approves `TOFT` to the `removeParams.market` and then calls function `removeCollateral()` of the provided market. There is no validation whatsoever in this address, such that a malicious market can be provided that steals all funds, as can be seen below:

```
    function remove(bytes memory _payload) public {
        ...
        approve(removeParams.market, removeParams.share); // no vali
        IMarket(removeParams.market).removeCollateral(
            to,
            to,
            removeParams.share
        );
        ...
    }
```

The following POC in Foundry demonstrates this vulnerability, the attacker is able to steal all `TOFT` in `mTapiocaOFT`:

▶ Details

Tools Used

Vscode, Foundry

Recommended Mitigation Steps

Whitelist the `removeParams.market` address to prevent users from providing malicious markets.

0xRektora (Tapioca) confirmed

[H-02] `exitPosition` in `TapiocaOptionBroker` may incorrectly inflate position weights

Submitted by ItsNio, also found by KIntern_NA

Users who `participate()` and place stakes with large magnitudes may have their weight removed prematurely from `pool.cumulative`, hence causing the weight logic of participation to be wrong. `pool.cumulative` will have an incomplete image of the actual pool hence allowing future users to have divergent power when they should not. In particular, this occurs during the `exitPosition()` function.

Proof of Concept

This vulnerability stems from `exitPosition()` using the current `pool.AverageMagnitude` instead of the respective magnitudes of the user's weights to update `pool.cumulative` on line 316. Hence, when users call `exitPosition()`, the amount that `pool.cumulative` is updated but may not be representative of the weight of the user's input.

Imagine if we have three users, Alice, Bob, and Charlie who all decide to call `participate()`. Alice calls `participate()` with a smaller amount and a smaller time, hence having a weight of 10. Bob calls `participate()` with a larger amount and a larger time, hence having a weight of 50. Charlie calls `participate()` with a weight of 20.

Scenario

- Alice calls `participate()` first at time 0 with the aforementioned amount and time. The `pool.cumulative` is now 10 and the `pool.AverageMagnitude` is 10 as well. Alice's position will expire at time 10.

- Bob calls `participate()` at time 5. The `pool.cumulative` is now 10 + 50 = 60 and the `pool.AverageMagnitude` is 50.
- Alice calls `exitPosition()` at time 10. `pool.cumulative` is 60, but `pool.AverageMagnitude` is still 50. Hence, `pool.cumulative` will be decreased by 50, even though the weight of Alice's input is 10.
- Charlie calls `participate` with weight 20. Charlie will have divergent power in the pool with both Bob and Charlie, since 20 > `pool.cumulative` (10).

If Alice does not participate at all, Charlie will not have divergent power in a pool with Bob and Charlie, since the `pool.cumulative` = Bob's weight = 50 > Charlie's weight (20).

We have provided a test to demonstrate the `pool.cumulative` inflation. Copy the following code into `tap-token-audit/test/oTAP/tOB.test.ts` as one of the tests.

▶ Details

This test runs the aforementioned scenario.

Expected Output:

```
BigNumber { value: "30000000000" }
A Duration:  10  B Duration:  100
Just A participation
[B4] Just A Cumulative:  BigNumber { value: "10" }
[B4] Just A Average:  BigNumber { value: "10" }
Exit A position
[A4] Just A Cumulative:  BigNumber { value: "0" }
[A4] Just A Average:  BigNumber { value: "10" }

Run both participation---
Time:  2023-08-03T21:40:52.700Z
[IN] Initial Cumulative:  BigNumber { value: "0" }
Participate A (smaller weight)
[ID] A Token ID:  BigNumber { value: "1" }
[B4] Both A Cumulative:  BigNumber { value: "10" }
[B4] Both A Average:  BigNumber { value: "10" }

Participate B (larger weight), Time(+5):  2023-08-03T21:40:52.8(
[ID] B Token ID:  BigNumber { value: "1" }
```

```
    [B4] Both AB Cumulative:  BigNumber { value: "60" }
    [B4] Both B Average:  BigNumber { value: "50" }

    Exit A (Dispraportionate Weight, Time(+6 Expire A):  2023-08-03T
    [!X!] Just B Cumulative:  BigNumber { value: "10" }
    [A4] Just B Average:  BigNumber { value: "50" }
    Exit B, Time(+100 Expire B):  2023-08-03T21:40:53.029Z
    [A4] END Cumulative:  BigNumber { value: "0" }
      ✓ POC (1077ms)
```

The POC is split into two parts:

The first part starting with `Just A Participation` is when just A enters and exits. This is correct, with the `pool.cumulative` increasing by 10 (the weight of A) and then being decreased by 10 when A exits.

The second part starting with `Run both participation---` describes the scenario mentioned by the bullet points. In particular, the `pool.cumulative` starts as 0 (`[IN] Initial Cumulative`).

Then, A enters the pool, and the `pool.cumulative` is increased to 10 (`[B4] Both A Cumulative`) similar to the first part.

Then, B enters the pool, before A exits. B has a weight of 50, thus the `pool.cumulative` increases to 60 (`[B4] Both AB Cumulative`).

The bug can be seen after the line beginning with `[!X!]`. The `pool.cumulative` labeled by “Just B Cumulative” is decreased by 60 - 10 = 50 when A exits, although the weight of A is only 10.

Recommended Mitigation Steps

There may be a need to store weights at the time of adding a weight instead of subtracting the last computed weight in `exitPosition()`. For example, when Alice calls `participate()`, the weight at that time is stored and removed when `exitPosition()` is called.

[0xRektora (Tapioca) confirmed](#)

[H-03] The amount of debt removed during `liquidation` may be worth more than the account's collateral

Submitted by ItsNio

The contract decreases user's debts but may not take the full worth in collateral from the user, leading to the contract losing potential funds from the missing collateral.

Proof of concept

During the `liquidate()` function call, the function `_updateBorrowAndCollateralShare()` is eventually invoked. This function liquidates a user's debt and collateral based on the value of the **collateral** they own.

In particular, the equivalent amount of debt, `availableBorrowPart` is calculated from the user's collateral on line 225 through the `computeClosingFactor()` function call.

Then, an additional fee through the `liquidationBonusAmount` is applied to the debt, which is then compared to the user's debt on line 240. The minimum of the two is assigned `borrowPart`, which intuitively means the maximum amount of debt that can be removed from the user's debt.

`borrowPart` is then increased by a bonus through `liquidationMultiplier`, and then converted to generate `collateralShare`, which represents the amount of collateral equivalent in value to `borrowPart` (plus some fees and bonus).

This new `collateralShare` may be more than the collateral that the user owns. In that case, the `collateralShare` is simply decreased to the user's collateral.

`collateralShare` is then removed from the user's collateral.

The problem lies in that although the `collateralShare` is equivalent to the `borrowPart`, or the debt removed from the user's account, it could be worth more than the collateral that the user owns in the first place. Hence, the contract loses out on funds, as debt is removed for less than it is actually worth.

To demonstrate, we provide a runnable POC.

Preconditions

```
...
if (collateralShare > userCollateralShare[user]) {
        require(false, "collateralShare and borrowPart not worth
        collateralShare = userCollateralShare[user];
    }
    userCollateralShare[user] -= collateralShare;
...
```

Add the `require` statement to line 261. This require statement essentially reverts the contract when the `if` condition satisfies. The `if` condition holds true when the `collateralShare` is greater that the user's collateral, which is the target bug.

Once the changes have been made, add the following test into the `singularity.test.ts` test in `tapioca-bar-audit/test`

Code

▶ Details

Expected Result

```
Collateral amt:  BigNumber { value: "100000000000000000000" }
WBTC borrow val:  BigNumber { value: "74000000" }
[$] Original price:  BigNumber { value: "10000" }
Price Drop: 120%
Running liquidation...
[*] Reverted with reason: collateralShare and borrowPart not wor
      ✓ POC (2289ms)
```

As demonstrated, the function call reverts due to the `require` statement added in the preconditions.

Recommended Mitigation

One potential mitigation for this issue would be to calculate the `borrowPart` depending on the existing users' collateral factoring in the fees and bonuses. The `collateralShare` with the fees and bonuses should not exceed the user's collateral.

cryptotechmaker (Tapioca) confirmed

[H-04] Incorrect solvency check because it multiplies collateralizationRate by share not amount when calculating liquidation threshold

Submitted by **Koolex**

When a Collateralized Debt Position (CDP) reaches that liquidation threshold, it becomes eligible for liquidation and anyone can repay a position in exchange for a portion of the collateral. `Market._isSolvent` is used to check if the user is solvent. if not, then it can be liquidated. Here is the method body:

```
    function _isSolvent(
            address user,
            uint256 _exchangeRate
    ) internal view returns (bool) {
            // accrue must have already been called!
            uint256 borrowPart = userBorrowPart[user];
            if (borrowPart == 0) return true;
            uint256 collateralShare = userCollateralShare[user];
            if (collateralShare == 0) return false;

            Rebase memory _totalBorrow = totalBorrow;

            return
                    yieldBox.toAmount(
                            collateralId,
                            collateralShare *
                                    (EXCHANGE_RATE_PRECISION / FEE_P
                                    collateralizationRate,
                            false
                    ) >=
                    // Moved exchangeRate here instead of dividing t
                    (borrowPart * _totalBorrow.elastic * _exchangeRa
```

```
                                _totalBorrow.base;
    }
```

[Code link]

The issue is that the collateralizationRate is multiplied by collateralShare (with precision constants) then converted to amount. This is incorrect, **the collateralizationRate sholud be used with amounts and not shares. Otherwise, we get wrong results.**

```
    yieldBox.toAmount(
                            collateralId,
                            collateralShare *
                                    (EXCHANGE_RATE_PRECISION / FEE_P
                                    collateralizationRate,
                            false
                    )
```

Please note that when using shares **it is not in favour of the protocol**, so amounts should be used instead. The only case where this is ok, is when the share/amount ratio is 1:1 which can not be, because totalAmount always get +1 and totalShares +1e8 to prevent 1:1 ratio type of attack.

```
    function _toAmount(
        uint256 share,
        uint256 totalShares_,
        uint256 totalAmount,
        bool roundUp
    ) internal pure returns (uint256 amount) {
        // To prevent reseting the ratio due to withdrawal of al
        // 1 amount/1e8 shares already burned. This also starts
        // functions like 8 decimal fixed point math. This preve
        // due to 'gifting' or rebasing tokens. (Up to a certair
        totalAmount++;
        totalShares_ += 1e8;
```

[Code link]

Moreover, in the method `_computeMaxAndMinLTVInAsset` which is supposed to returns the min and max LTV for user in asset price. Amount is used and not share. Here is the code:

```
function _computeMaxAndMinLTVInAsset(
        uint256 collateralShare,
        uint256 _exchangeRate
) internal view returns (uint256 min, uint256 max) {
        uint256 collateralAmount = yieldBox.toAmount(
                collateralId,
                collateralShare,
                false
        );

        max = (collateralAmount * EXCHANGE_RATE_PRECISIC
        min = (max * collateralizationRate) / FEE_PRECIS
}
```

[Code Link]

I've set this to high severity because solvency check is a crucial part of the protocol. In short, we have :

1. Inconsistency across the protocol
2. Inaccuracy of calculating the liquidation threshold
3. Not in favour of the protocol

Note: this is also applicable for ohter methods. For example, `Market._computeMaxBorrowableAmount`.

Proof of Concept

- When you run the PoC below, you will get the following results:

```
[PASS] test_borrow_repay() (gas: 118001)
Logs:
  ===BORROW===
  UserBorrowPart: 745372500000000000000
  Total Borrow Base: 745372500000000000000
```

```
    Total Borrow Elastic: 745372500000000000000
    ===356 days passed===
    Total Borrow Elastic: 749089151896269477984
    ===Solvency#1 => multiply by share===
    A: 749999999999925000007500074999999924999
    B: 749089151896269477984000000000000000000
    ===Solvency#2 => multiply by amount===
    A: 749999999999925000007500000000000000000
    B: 749089151896269477984000000000000000000
    ===Result===
    Solvency#1.A != Solvency#2.A

Test result: ok. 1 passed; 0 failed; finished in 16.69ms
```

As you can see, numbers are not equal, and when using shares it is not in favour of the protocol, so amount should be used instead.

- Code: Please note some lines in borrow method were commented out for simplicity. It is irrelevant anyway.
- `_toAmount` copied from YieldBoxRebase

▶ Details

Recommended Mitigation Steps

Use amount for calculation instead of shares. Check the PoC as it demonstrates such an example.

0xRektora (sponsor) confirmed

[H-05] Ability to steal user funds and increase collateral share infinitely in BigBang and Singularity

Submitted by Ack, *also found by Koolex (*1, 2*),* RedOneN, plainshift, ladboy233, bin2chen, zzzitron, ayeslick, KIntern_NA, kaden, xuwinnie, Oxsadeeq, 0xStalin, 0xG0P1, ltyu, cergyk, TiesStevelink, rvierdiiev, *and* 0xRobocop

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/

bigBang/BigBang.sol#L288

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/singularity/SGLCollateral.sol#L27

The `addCollateral` methods in both BigBang and Singularity contracts allow the share parameter to be passed as `0`. When `share` is `0`, the equivalent amount of shares is calculated using the YieldBox `toShare` method. However, there is a modifier named `allowedBorrow` that is intended to check the allowed borrowing amount for each implementation of the `addCollateral` methods. Unfortunately, the modifier is called with the `share` value passed to `addCollateral`, and in the case of `0`, it will always pass.

▶ Details

This leads to various critical scenarios in BigBang and Singularity markets where user assets can be stolen, and collateral share can be increased infinitely which in turn leads to infinite USDO borrow/mint and borrowing max assets from Singularity market.

Refer to **Proof of Concept** for attack examples

Impact

High - allows stealing of arbitrary user yieldbox shares in BigBang contract and Singularity. In the case of BigBang this leads to infinite minting of USDO. Effectively draining all markets and LPs where USDO has value. In the case of Singularity this leads to infinite borrowing, allowing an attacker to obtain possession of all other users' collateral in Singularity.

Proof of concept

1. Malicious actor can add any user shares that were approved to BigBang or Singularity contracts deployed. This way adversary is stealing user shares that he can unwrap to get underlying collateral provided.

▶ Details

2. For Singularity contract this allows to increase collateralShare by the amount of assets provided as collateral infinitely leading to `x / x + 1` share of the

collateral for the caller with no shares in the pool, where x is the number of times the `addColateral` is called, effectively allowing for infinite borrowing. As a consequence, the attacker can continuously increase their share of the collateral without limits, leading to potentially excessive borrowing of assets from the Singularity market.

▶ Details

3. In the BigBang contract, this vulnerability allows a user to infinitely increase their collateral shares by providing collateral repeatedly. As a result, the user can artificially inflate their collateral shares provided, potentially leading to an excessive borrowing capacity. By continuously adding collateral without limitations, the user can effectively borrow against any collateral amount they desire, which poses a significant risk to USDO market.

▶ Details

Recommended Mitigation Steps

- Check allowed to borrow shares amount after evaluating equivalent them

0xRektora (Tapioca) confirmed via duplicate issue 55

[H-06] BalancerStrategy `_withdraw` uses `BPT_IN_FOR_EXACT_TOKENS_OUT` which can be attack to cause loss to all depositors

Submitted by GalloDaSballo

Withdrawals can be manipulated to cause complete loss of all tokens.

The BalancerStrategy accounts for user deposits in terms of the BPT shares they contributed, however, for withdrawals, it estimates the amount of BPT to burn based on the amount of ETH to withdraw, which can be manipulated to cause a total loss to the Strategy.

Deposits of weth are done via userData.joinKind set to `1`, which is extracted here in the generic Pool Logic:
https://etherscan.io/address/0x5c6ee304399dbdb9c8ef030ab642b10820db8f56#code#F24#L49

The interpretation (by convention is shown here):
https://etherscan.io/address/0x5c6ee304399dbdb9c8ef030ab642b10820db8f56#code#F24#L49

```
enum JoinKind { INIT, EXACT_TOKENS_IN_FOR_BPT_OUT, TOKEN_IN_
```

Which means that the deposit is using `EXACT_TOKENS_IN_FOR_BPT_OUT` which is safe in most circumstances (Pool Properly Balanced, with minimum liquidity).

`BPT_IN_FOR_EXACT_TOKENS_OUT` is vulnerable to manipulation

`_vaultWithdraw` uses the following logic to determine how many BPT to burn:

https://github.com/Tapioca-DAO/tapioca-yieldbox-strategies-audit/blob/05ba7108a83c66dada98bc5bc75cf18004f2a49b/contracts/balancer/BalancerStrategy.sol#L224-L242

```
    uint256[] memory minAmountsOut = new uint256[](poolTokens.length
            for (uint256 i = 0; i < poolTokens.length; i++) {
                if (poolTokens[i] == address(wrappedNative)) {
                    minAmountsOut[i] = amount;
                    index = int256(i);
                } else {
                    minAmountsOut[i] = 0;
                }
            }

            IBalancerVault.ExitPoolRequest memory exitRequest;
            exitRequest.assets = poolTokens;
            exitRequest.minAmountsOut = minAmountsOut;
            exitRequest.toInternalBalance = false;
            exitRequest.userData = abi.encode(
                2,
                exitRequest.minAmountsOut,
                pool.balanceOf(address(this))
            );
```

This query logic is using `2`, which Maps out to `BPT_IN_FOR_EXACT_TOKENS_OUT` which means Exact Out, with any (all) BPT IN, this means that the swapper is willing

to burn all tokens:
https://etherscan.io/address/0x5c6ee304399dbdb9c8ef030ab642b10820db8f56#code#F24#L51

```
enum ExitKind { EXACT_BPT_IN_FOR_ONE_TOKEN_OUT, EXACT_BI
```

This meets the 2 prerequisite for stealing value from the vault by socializing loss due to single sided exposure:

- 1. The request is for at least `amount` `WETH`
- 2. The request is using `BPT_IN_FOR_EXACT_TOKENS_OUT`

Which means the strategy will accept any slippage, in this case 100%, causing it to take a total loss for the goal of allowing a withdrawal, at the advantage of the attacker and the detriment of all other depositors.

POC

The requirement to trigger the loss are as follows:

- Deposit to have some amount of BPTs deposited into the strategy
- Imbalance the Pool to cause pro-rata amount of single token to require burning a lot more BPTs
- Withdraw from the strategy, the strategy will burn all of the BPTs it owns (more than the shares)
- Rebalance the pool with the excess value burned from the strategy

Further Details

Specifically, in withdrawing one Depositor Shares, the request would end up burning EVERYONEs shares, causing massive loss to everyone.

This has already been exploited and explained in Yearns Disclosure:

https://github.com/yearn/yearn-security/blob/master/disclosures/2022-01-30.md

More specifically this finding can cause a total loss, while trying to withdraw tokens for a single user, meaning that an attacker can setup the pool to cause a complete loss to all other stakers.

Mitigation Step

Use `EXACT_BPT_IN_FOR_TOKENS_OUT` and denominate the Strategy in LP tokens to avoid being attacked via single sided exposure.

cryptotechmaker (Tapioca) confirmed

[H-07] Usage of `BalancerStrategy.updateCache` will cause single sided Loss, discount to Depositor and to OverBorrow from Singularity

Submitted by **GalloDaSballo**, *also found by* **carrotsmuggler**, **kaden**, *and* **cergyk**

The BalancerStrategy uses a cached value to determine it's balance in pool for which it takes Single Sided Exposure.

This means that the Strategy has some BPT tokens, but to price them, it's calling `vault.queryExit` which simulates withdrawing the LP in a single sided manner.

Due to the single sided exposure, it's trivial to perform a Swap, that will change the internal balances of the pool, as a way to cause the Strategy to discount it's tokens.

By the same process, we can send more ETH as a way to inflate the value of the Strategy, which will then be cached.

Since `_currentBalance` is a view-function, the YieldBox will accept these inflated values without a way to dispute them

https://github.com/Tapioca-DAO/tapioca-yieldbox-strategies-audit/blob/05ba7108a83c66dada98bc5bc75cf18004f2a49b/contracts/balancer/BalancerStrategy.sol#L138-L147

```
function _deposited(uint256 amount) internal override nonRe
```

```
            uint256 queued = wrappedNative.balanceOf(address(this));
            if (queued > depositThreshold) {
                _vaultDeposit(queued);
                emit AmountDeposited(queued);
            }
            emit AmountQueued(amount);

            updateCache(); /// @audit this is updated too late (TODO
        }
```

POC

- Imbalance the pool (Sandwich A)
- Update `updateCache`
- Deposit into YieldBox, YieldBox is using a `view` function, meaning it will use the manipulated strategy `_currentBalance`
- `_deposited` trigger an `updateCache`
- Rebalance the Pool (Sandwich B)
- Call `updateCache` again to bring back the rate to a higher value
- Withdraw at a gain

Result

Imbalance Up -> Allows OverBorrowing and causes insolvency to the protocol
Imbalance Down -> Liquidate Borrowers unfairly at a profit to the liquidator
Sandwhiching the Imbalance can be used to extract value from the strategy and steal user deposits as well

Mitigation

Use fair reserve math, avoid single sided exposure (use the LP token as underlying, not one side of it)

[cryptotechmaker (Tapioca) confirmed]

[H-08] `LidoEthStrategy._currentBalance` is subject to price manipulation, allows overborrowing and liquidations

Submitted by GalloDaSballo, *also found by* ladboy233, carrotsmuggler, kaden, cergyk, *and* rvierdiiev

The strategy is pricing stETH as ETH by asking the pool for it's return value

This is easily manipulatable by performing a swap big enough

https://github.com/Tapioca-DAO/tapioca-yieldbox-strategies-audit/blob/05ba7108a83c66dada98bc5bc75cf18004f2a49b/contracts/lido/LidoEthStrategy.sol#L118-L125

```
function _currentBalance() internal view override returns (u
    uint256 stEthBalance = stEth.balanceOf(address(this));
    uint256 calcEth = stEthBalance > 0
        ? curveStEthPool.get_dy(1, 0, stEthBalance) // TODO:
        : 0;
    uint256 queued = wrappedNative.balanceOf(address(this));
    return calcEth + queued;
}

/// @dev deposits to Lido or queues tokens if the 'depositTh
function _deposited(uint256 amount) internal override nonRee
    uint256 queued = wrappedNative.balanceOf(address(this));
    if (queued > depositThreshold) {
        require(!stEth.isStakingPaused(), "LidoStrategy: sta
        INative(address(wrappedNative)).withdraw(queued);
        stEth.submit{value: queued}(address(0)); //1:1 betwe
        emit AmountDeposited(queued);
        return;
    }
    emit AmountQueued(amount);
}
```

POC

- Imbalance the Pool to overvalue the stETH
- Overborrow and Make the Singularity Insolvent
- Imbalance the Pool to undervalue the stETH

- Liquidate all Depositors (at optimal premium since attacker can control the price change)

Coded POC

Logs

```
[PASS] testSwapStEth() (gas: 372360)
  Initial Price 544366353773257141792O
  Changed Price 21870716512849779079 21
  Initial Price 21870716512849779079 21
  Changed Price 1073148438886623970
```

```
[PASS] testSwapETH() (gas: 300192)
Logs:
  value 10000000000000000000000
  Initial Price 544366353773257141792O
  Changed Price 9755041616702274912586
  value 70000000000000000000000
  Initial Price 9755041616702274912586
  Changed Price 680711874102963551173181
```

Considering that swap fees are 1BPS, the attack is profitable at very low TVL

▶ Details

Mitigation

Use the Chainlink stETH / ETH Price Feed or Ideally do not expose the strategy to any conversion, simply deposit and withdraw stETH directly to avoid any risk or attack in conversions

https://data.chain.link/arbitrum/mainnet/crypto-eth/steth-eth

https://data.chain.link/ethereum/mainnet/crypto-eth/steth-eth

0xRektora (Tapioca) confirmed via duplicate issue 828

[H-09] `TricryptoLPStrategy.compoundAmount` always returns 0 because it's using staticall vs call

Submitted by GalloDaSballo

`compoundAmount` will always try to sell 0 tokens because the `staticall` will revert since the function changes storage in `checkpoint`

This causes the `compoundAmount` to always return 0, which means that the Strategy is underpriced at all times allowing to Steal all Rewards via:

- Deposit to own a high % of ownerhsip in the strategy (shares are underpriced)
- Compound (shares socialize the yield to new total supply, we get the majority of that)
- Withdraw (lock in immediate profits without contributing to the Yield)

POC

This Test is done on the Arbitrum Tricrypto Gauge with Foundry

1 is the flag value for a revert 0 is the expected value

We get 1 when we use staticcall since the call reverts internally We get 0 when we use call since the call doesn't

The comment in the Gauge Code is meant for usage off-chain, onChain you must accrue (or you could use a Accrue Then Revert Pattern, similar to UniV3 Quoter)

NOTE: The code for Mainnet is the same, so it will result in the same impact https://etherscan.io/address/0xDeFd8FdD20e0f34115C7018CCfb655796F6B2168#code#L375

Foundry POC

```
forge test --match-test test_callWorks --rpc-url https://arb-mai
```

Which will revert since `checkpoint` is a non-view function and staticall reverts if any state is changed

https://arbiscan.io/address/0x555766f3da968ecbefa690ffd49a2ac02f47aa5f#code#L168

▶ Details

Mitigation Step

You should use a non-view function like in `compound`

cryptotechmaker (Tapioca) confirmed

[H-10] Liquidated USDO from BigBang not being burned after liquidation inflates USDO supply and can threaten peg permanently

Submitted by unsafesol, *also found by* peakbolt, 0xnev, rvierdiiev, *and* 0xRobocop

Absence of proper USDO burn after liquidation in the BigBang market results in a redundant amount of USDO being minted without any collateral or backing. Thus, the overcollaterization of USDO achieved through BigBang will be eventually lost and the value of USDO in supply (1USDO = 1$) will exceed the amount of collateral locked in BigBang. This has multiple repercussions- the USDO peg will be threatened and yieldBox will have USDO which has virtually no value, resulting in all the BigBang strategies failing.

Proof of Concept

According to the Tapioca documentation, the BigBang market mints USDO when a user deposits sufficient collateral and borrows tokens. When a user repays the borrowed USDO, the market burns the borrowed USDO and unlocks the appropriate amount of collateral. This is essential to the peg of USDO, since USDO tokens need a valid collateral backing.

While liquidating a user as well, the same procedure should be followed- after swapping the user's collateral for USDO, the repaid USDO (with liquidation) must be

burned so as to sustain the USDO peg. However, this is not being done. As we can see here: https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L618-L637, the collateral is swapped for USDO, and fee is extracted and transferred to the appropriate parties, but nothing is done for the remaining USDO which was repaid. At the same time, this was done correctly done in BigBang#_repay for repayment here: https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L734-L736.

This has the following effects:

1. The BigBang market now has redundant yieldBox USDO shares which have no backing.
2. The redundant USDO is now performing in yieldBox strategies of tapioca.
3. The USDO eventually becomes overinflated and exceeds the value of underlying collateral.
4. The strategies start not performing since they have unbacked USDO, and the USDO peg is lost as well since there is no appropriate amount of underlying collateral.

Recommended Mitigation Steps

Burn the USDO acquired through liquidation after extracting fees for appropriate parties.

0xRektora (Tapioca) confirmed

[H-11] TOFT `exerciseOption` can be used to steal all underlying erc20 tokens

Submitted by **windhustler**, *also found by* **Ack**

Unvalidated input data for the `exerciseOption` function can be used to steal all the erc20 tokens from the contract.

Proof of Concept

Each BaseTOFT is a wrapper around an `erc20` token and extends the `OFTV2` contract to enable smooth cross-chain transfers through LayerZero. Depending on the erc20 token which is used usually the erc20 tokens will be held on one chain and then only the shares of `OFTV2` get transferred around (burnt on one chain, minted on another chain). Subject to this attack is `TapiocaOFTs` or `mTapiocaOFTs` which store as an underlying token an erc20 token(not native). In order to mint `TOFT` shares you need to deposit the underlying erc20 tokens into the contract, and you get `TOFT` shares.

The attack flow is the following:

1. The attack starts from the `exerciseOption`. Nothing is validated here and the only cost of the attack is the `optionsData.paymentTokenAmount` which is burned from the attacker. This can be some small amount.
2. When the message is received on the remote chain inside the `exercise` function it is important that nothing reverts for the attacker.
3. For the attacker to go through the attacker needs to pass the following data:

```
function exerciseInternal(
        address from,
        uint256 oTAPTokenID,
        address paymentToken,
        uint256 tapAmount,
        address target,
        ITapiocaOptionsBrokerCrossChain.IExerciseLZSendTapData
            memory tapSendData,
        ICommonData.IApproval[] memory approvals
    ) public {
        // pass zero approval so this is skipped
        if (approvals.length > 0) {
            _callApproval(approvals);
        }

        // target is the address which does nothing, but has the
        ITapiocaOptionsBroker(target).exerciseOption(
            oTAPTokenID,
            paymentToken,
            tapAmount
```

```
            );
            // tapSendData.withdrawOnAnotherChain = false so we ente
            if (tapSendData.withdrawOnAnotherChain) {
                ISendFrom(tapSendData.tapOftAddress).sendFrom(
                    address(this),
                    tapSendData.lzDstChainId,
                    LzLib.addressToBytes32(from),
                    tapAmount,
                    ISendFrom.LzCallParams({
                        refundAddress: payable(from),
                        zroPaymentAddress: tapSendData.zroPaymentAdc
                        adapterParams: LzLib.buildDefaultAdapterPara
                            tapSendData.extraGas
                        )
                    })
                );
            } else {
                // tapSendData.tapOftAddress is the address of the u
                // from is the address of the attacker
                // tapAmount is the balance of erc20 tokens of this
                IERC20(tapSendData.tapOftAddress).safeTransfer(from,
            }
        }
```

4. So the attack is just simply transferring all the underlying erc20 tokens to the attacker.

The underlying `ERC20` token for each `TOFT` can be queried through `erc20()` function, and the `tapAmount` to pass is `ERC20` balance of the `TOFT`.

This attack is possible because the `msg.sender` inside the `exerciseInternal` is the address of the `TOFT` which is the owner of all the ERC20 tokens that get stolen.

Recommended Mitigation Steps

Validate that `tapSendData.tapOftAddress` is the address of `TapOFT` token either while sending the message or during the reception of the message on the remote chain.

0xRektora (Tapioca) confirmed

[H-12] TOFT `removeCollateral` can be used to steal all the balance

Submitted by **windhustler**, *also found by* **0x73696d616f**

`removeCollateral` -> `remove` message pathway can be used to steal all the balance of the `TapiocaOFT` and `mTapiocaOFT` tokens in case when their underlying tokens is native. TOFTs that hold native tokens are deployed with **erc20 address** set to address zero, so while **minting you need to transfer value**.

Proof of Concept

The attack needs to be executed by invoking the `removeCollateral` function from any chain to chain on which the underlying balance resides, e.g. host chain of the TOFT. When the message is **received on the remote chain**, I have placed in the comments below what are the params that need to be passed to execute the attack.

▶ Details

Neither `removeParams.marketHelper` or `withdrawParams.withdrawLzFeeAmount` are validated on the sending side so the former can be the address of a malicious contract and the latter can be the TOFT's balance of gas token.

This type of attack is possible because the `msg.sender` in `IMagnetar(removeParams.marketHelper).withdrawToChain` is the address of the TOFT contract which holds all the balances.

This is because:

1. Relayer submits the message to `lzReceive` so he is the `msg.sender`.
2. Inside the `_blockingLzReceive` there is a call into its own public function so the `msg.sender` is the **address of the contract**.
3. Inside the `_nonBlockingLzReceive` there is **delegatecall** into a corresponding module which preserves the `msg.sender` which is the address of the TOFT.
4. Inside the module there is a call to **withdrawToChain** and here the `msg.sender` is the address of the TOFT contract, so we can maliciously transfer all the balance of the TOFT.

Tools Used

Foundry

Recommended Mitigation Steps

It's hard to recommend a simple fix since as I pointed out in my other issues the airdropping logic has many flaws. One of the ways of tackling this issue is during the `removeCollateral` to:

- Do not allow `adapterParams` params to be passed as bytes but rather as `gasLimit` and `airdroppedAmount`, from which you would encode either `adapterParamsV1` or `adapterParamsV2`.
- And then on the receiving side check and send with value only the amount the user has airdropped.

0xRektora (Tapioca) confirmed and commented:

> Related to https://github.com/code-423n4/2023-07-tapioca-findings/issues/1290

[H-13] TOFT `triggerSendFrom` can be used to steal all the balance

Submitted by **windhustler**

`triggerSendFrom` -> `sendFromDestination` message pathway can be used to steal all the balance of the `TapiocaOFT` and `mTapiocaOFT` tokens in case when their underlying tokens is native gas token. TOFTs that hold native tokens are deployed with erc20 address set to address zero, so while minting you need to transfer value.

Proof of Concept

The attack flow is the following:

1. Attacker calls `triggerSendFrom` with `airdropAdapterParams` of type airdropAdapterParamsV1 which don't airdrop any value on the remote chain

but just deliver the message.

2. On the other hand lzCallParams are of type `adapterParamsV2` which are used to airdrop the balance from the destination chain to another chain to the attacker.

```
struct LzCallParams {
    address payable refundAddress; // => address of the attacker
    address zroPaymentAddress; // => doesn't matter
    bytes adapterParams; //=> airdropAdapterParamsV2
}
```

3. Whereby the `sendFromData.adapterParams` would be encoded in the following way:

```
function encodeAdapterParamsV2() public {
    // https://layerzero.gitbook.io/docs/evm-guides/advanced/rel
    uint256 gasLimit = 250_000; // something enough to deliver t
    uint256 airdroppedAmount = max airdrop cap defined at https:
    address attacker = makeAddr("attacker"); // => address of th
    bytes memory adapterParams = abi.encodePacked(uint16(2), gas
}
```

4. When this is received on the remote inside the `sendFromDestination` `ISendFrom(address(this)).sendFrom{value: address(this).balance}` is instructed by the malicious `ISendFrom.LzCallParams memory callParams` to actually airdrop the max amount allowed by LayerZero to the attacker on the `lzDstChainId`.
5. Since there is a cap on the maximum airdrop amount this type of attack would need to be executed multiple times to drain the balance of the TOFT.

The core issue at play here is that `BaseTOFT` delegatecalls into the `BaseTOFTOptionsModule` and thus the BaseTOFT is the `msg.sender` for `sendFrom` function.

There is also another simpler attack flow possible:

1. Since `sendFromDestination` passes as value whole balance of the TapiocaOFT it is enough to specify the refundAddress in callParams as the address of the attacker.
2. This way the whole balance will be transferred to the _lzSend and any excess will be refunded to the `_refundAddress`.
3. This is how layer zero works.

Tools Used

Foundry

Recommended Mitigation Steps

One of the ways of tackling this issue is during the `triggerSendFrom` to:

- Not allowing `airdropAdapterParams` and `sendFromData.adapterParams` params to be passed as bytes but rather as `gasLimit` and `airdroppedAmount`, from which you would encode either `adapterParamsV1` or `adapterParamsV2`.
- And then on the receiving side check and send with value only the amount the user has airdropped.

```
// Only allow the airdropped amount to be used for another messa
ISendFrom(address(this)).sendFrom{value: aidroppedAmount}(
    from,
    lzDstChainId,
    LzLib.addressToBytes32(from),
    amount,
    callParams
);
```

0xRektora (Tapioca) confirmed

[H-14] All assets of (m)TapiocaOFT can be stealed by depositing to strategy cross chain call with 1 amount but maximum shares possible

Submitted by 0x73696d616f

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L224

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L450

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L47

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L58

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L154

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L181-L185

https://github.com/Tapioca-DAO/YieldBox/blob/f5ad271b2dcab8b643b7cf622c2d6a128e109999/contracts/YieldBox.sol#L118

Attacker can be debited only the least possible amount (`1`) but send the `share` argument as the maximum possible value corresponding to the `erc` balance of `(m)TapiocaOFT` . This would enable the attacker to steal all the `erc` balance of the `(m)TapiocaOFT` contract.

Proof of Concept

In `BaseTOFT` , `SendToStrategy()` , has no validation and just delegate calls to `sendToStrategy()` function of the `BaseTOFTStrategyModule` .

In the mentioned module, the quantity debited from the user is the `amount` argument, having no validation in the corresponding `share` amount:

```
    function sendToStrategy(
        address _from,
        address _to,
        uint256 amount,
        uint256 share,
        uint256 assetId,
        uint16 lzDstChainId,
        ICommonData.ISendOptions calldata options
    ) external payable {
        require(amount > 0, "TOFT_0");
        bytes32 toAddress = LzLib.addressToBytes32(_to);
        _debitFrom(_from, lzEndpoint.getChainId(), toAddress, amount
        ...
```

Then, a payload is sent to the destination chain in `_lzSend()` of type `PT_YB_SEND_STRAT`.

Again, in `BaseTOFT`, the function `_nonBlockingLzReceive()` handles the received message and delegate calls to the `BaseTOFTStrategyModule`, function `strategyDeposit()`. In this, function, among other things, it delegate calls to `depositToYieldbox()`, of the same module:

```
    function depositToYieldbox(
        uint256 _assetId,
        uint256 _amount,
        uint256 _share,
        IERC20 _erc20,
        address _from,
        address _to
    ) public {
        _amount = _share > 0
            ? yieldBox.toAmount(_assetId, _share, false)
            : _amount;
        _erc20.approve(address(yieldBox), _amount);
        yieldBox.depositAsset(_assetId, _from, _to, _amount, _share)
    }
```

The `_share` argument is the one the user initially provided in the source chain; however, the `_amount`, is computed from the `yieldBox` ratio, effectively overriding the specified `amount` in the source chain of `1`. This will credit funds to the attacker from other users that bridged assets through `(m)TapiocaOFT`.

The following POC in Foundry demonstrates how an attacker can be debited on the source chain an amount of `1` but call `depositAsset()` on the destination chain with an amount of `2e18`, the available in the `TapiocaOFT` contract.

▶ Details

Tools Used

Vscode, Foundry

Recommended Mitigation Steps

Given that it's impossible to fetch the `YieldBox` ratio in the source chain, it's best to stick with the amount only and remove the `share` argument in the cross chain `sendToStrategy()` function call.

0xRektora (Tapioca) confirmed

[H-15] Attacker can specify any `receiver` in `USD0.flashLoan()` to drain `receiver` balance

Submitted by **mojito_auditor**, *also found by* **n1punp**

The flash loan feature in USD0's `flashLoan()` function allows the caller to specify the `receiver` address. USD0 is then minted to this address and burnt from this address plus a fee after the callback. Since there is a fee in each flash loan, an attacker can abuse this to drain the balance of the `receiver` because the `receiver` can be specified by the caller without validation.

Proof of Concept

The allowance checked that `receiver` approved to `address(this)` but not check if `receiver` approved to `msg.sender`

```
    uint256 _allowance = allowance(address(receiver), address(this))
    require(_allowance >= (amount + fee), "USDO: repay not approved'
    // @audit can specify receiver, drain receiver's balance
    _approve(address(receiver), address(this), _allowance - (amount
    _burn(address(receiver), amount + fee);
    return true;
```

Recommended Mitigation Steps

Consider changing the "allowance check" to be the allowance that the receiver gave to the caller instead of `address(this)`.

0xRektora (Tapioca) confirmed

[H-16] Attacker can block LayerZero channel due to variable gas cost of saving payload

Submitted by **windhustler**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/master/contracts/usd0/BaseUSDO.sol#L399

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/master/contracts/tOFT/BaseTOFT.sol#L442

https://github.com/Tapioca-DAO/tap-token-audit/blob/main/contracts/tokens/BaseTapOFT.sol#L52

This is an issue that affects `BaseUSDO`, `BaseTOFT`, and `BaseTapOFT` or all the contracts which are sending and receiving LayerZero messages. The consequence of this is that anyone can with low cost and high frequency keep on blocking the pathway between any two chains, making the whole system unusable.

Proof of Concept

I will illustrate the concept of blocking the pathway on the example of sending a message through `BaseTOFT's` `sendToYAndBorrow`. This function allows the user to mint/borrow `USDO` with some collateral that is wrapped in a `TOFT` and gives the option of transferring minted `USDO` to another chain.

The attack starts by invoking `sendToYBAndBorrow` which delegate calls into `BaseTOFTMarketModule`. If we look at the implementation inside the `BaseTOFTMarketModule` nothing is validated there except for the `lzPayload` which has the packetType of `PT_YB_SEND_SGL_BORROW`.

The only validation of the message happens inside the `LzApp` with the configuration which was set. What is restrained within this configuration is the `payload size`, which if not configured defaults to **10k bytes**.

The application architecture was set up in a way that all the messages regardless of their packetType go through the same `_lzSend` implementation. I'm mentioning that because it means that if the project decides to change the default payload size to something smaller(or bigger) it will be dictated by the message with the biggest possible payload size.

I've mentioned the **minimum gas enforcement in my other issue** but even if that is fixed and a high min gas is enforced this is another type of issue.

To execute the attack we need to pass the following parameters to the function mentioned above:

```
function executeAttack() public {
    address tapiocaOFT = makeAddr("TapiocaOFT-AVAX");
    tapiocaOFT.sendToYBAndBorrow{value: enough_gas_to_go_thr
        address from => // malicious user address
        address to => // malicious user address
        lzDstChainId => // any chain lzChainId
        bytes calldata airdropAdapterParams => // encode in
        ITapiocaOFT.IBorrowParams calldata borrowParams, //
        ICommonData.IWithdrawParams calldata withdrawParams,
        ICommonData.ISendOptions calldata options, // can be
        ICommonData.IApproval[] calldata approvals // Elabor
    )
```

```
}
```

`ICommonData.IApproval[] calldata approvals` are going to be fake data so max payload size limit is reached(10k). The `target` of the 1st approval in the array will be the `GasDrainingContract` deployed on the receiving chain and the `permitBorrow = true`.

```
contract GasDrainingContract {
    mapping(uint256 => uint256) public storageVariables;

    function permitBorrow(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external {
        for (uint256 i = 0; i < 100000; i++) {
            storageVariables[i] = i;
        }
    }
}
```

Let's take an example of an attacker sending a transaction on the home chain which specifies a 1 million gasLimit for the destination transaction.

1. Transaction is successfully received inside the `lzReceive` after which it reaches _blockingLzReceive.
2. This is the first external call and according to `EIP-150` out of 1 million gas:
    - 63/64 or ~985k would be forwarded to the external call.
    - 1/64 or ~15k will be left for the rest of the execution.
3. The cost of saving a big payload into the `failedMessages` and emitting events is higher than 15k.

When it comes to 10k bytes it is around 130k gas but even with smaller payloads, it is still significant. It can be tested with the following code:

▶ Details

- If the payload is 9999 bytes the cost of saving it and emitting the event is 131k gas.
- Even with a smaller payload of 500 bytes the cost is 32k gas.
- If we can drain the 985k gas in the rest of the execution since storing `failedMessages` would fail the pathway would be blocked because this will fail at the level of LayerZero and result in `StoredPayload`.
- Let's continue the execution flow just to illustrate how this would occur, inside the implementation for `_nonblockingLzReceive` the `_executeOnDestination` is invoked for the right packet type and there we have another **external call** which delegatecalls into the right module.

Since it is also an external call only 63/64 gas is forwarded which is roughly:

- 970k would be forwarded to the module
- 15k reserved for the rest of the function
- This 970k gas is used for `borrow`, and it would be totally drained inside our **malicious GasDraining contract from above**, and then the execution would continue inside the `executeOnDestination` which also fails due to 15k gas not being enough, and finally, it fails inside the `_blockingLzReceive` due to out of gas, resulting in blocked pathway.

Tools Used

Foundry

Recommended Mitigation Steps

`_executeOnDestination` **storing logic** is just code duplication and serves no purpose. Instead of that you should override the `_blockingLzReceive`.

Create a new storage variable called `gasAllocation` which can be set only by the owner and change the implementation to:

```
(bool success, bytes memory reason) = address(this).excessivelyS
```

While ensuring that `gasleft() > gasAllocation` in each and every case. This should be enforced on the sending side.

Now this is tricky because as I have shown the gas cost of storing payload varies with payload size meaning the `gasAllocation` needs to be big enough to cover storing max payload size.

Other occurrences

This exploit is possible with all the packet types which allow arbitrary execution of some code on the receiving side with something like I showed with the `GasDrainingContract`. Since almost all packets allow this it is a common issue throughout the codebase, but anyway listing below where it can occur in various places:

▶ Details

Since inside the `twTap.claimAndSendRewards(tokenID, rewardTokens)` there are no reverts in case the `rewardToken` is invalid we can execute the gas draining attack inside the `sendFrom` whereby `rewardTokens[i]` is our malicious contract.

0xRektora (Tapioca) confirmed

[H-17] Attacker can block LayerZero channel due to missing check of minimum gas passed

Submitted by **windhustler**, *also found by* **0x73696d616f**

This is an issue that affects all the contracts that inherit from `NonBlockingLzApp` due to incorrect overriding of the `lzSend` function and lack of input validation and the ability to specify whatever `adapterParams` you want. The consequence of this is that anyone can with a low cost and high frequency keep on blocking the pathway between any two chains, making the whole system unusable.

Proof of Concept

Layer Zero minimum gas showcase

While sending messages through LayerZero, the sender can specify how much gas he is willing to give to the Relayer to deliver the payload to the destination chain. This configuration is specified in relayer adapter params. All the invocations of `lzSend` inside the TapiocaDao contracts naively assume that it is not possible to specify less than 200k gas on the destination, but in reality, you can pass whatever you want. As a showcase, I have set up a simple contract that implements the `NonBlockingLzApp` and sends only 30k gas which reverts on the destination chain resulting in `StoredPayload` and blocking of the message pathway between the two lzApps. The transaction below proves that if no minimum gas is enforced, an application that has the intention of using the `NonBlockingApp` can end up in a situation where there is a `StoredPayload` and the pathway is blocked.

Transaction Hashes for the example mentioned above:

- LayerZero Scan: https://layerzeroscan.com/106/address/0xe6772d0b85756d1af98ddfc61c5339e10d1b6eff/message/109/address/0x5285413ea82ac98a220dd65405c91d735f4133d8/nonce/1
- Tenderly stack trace of the sending transaction hash: https://dashboard.tenderly.co/tx/avalanche-mainnet/0xe54894bd4d19c6b12f30280082fc5eb693d445bed15bb7ae84dfaa049ab5374d/debugger?trace=0.0.1
- Tenderly stack trace of the receiving transaction hash: https://dashboard.tenderly.co/tx/polygon/0x87573c24725c938c776c98d4c12eb15f6bacc2f9818e17063f1bfb25a00ecd0c/debugger?trace=0.2.1.3.0.0.0.0

Attack scenario

The attacker calls `triggerSendFrom` and specifies a small amount of gas in the airdropAdapterParams(~50k gas). The Relayer delivers the transaction with the specified gas at the destination.

The transaction is first validated through the LayerZero contracts before it reaches the `lzReceive` function. The Relayer will give exactly the gas which was specified through the `airdropAdapterParams`. The line where it happens inside the LayerZero contract is **here**, and `{gas: _gasLimit}` is the gas the sender has paid for. The objective is that due to this small gas passed the transaction reverts somewhere inside the `lzReceive` function and the message pathway is blocked, resulting in `StoredPayload`.

The objective of the attack is that the execution doesn't reach the `NonblockingLzApp` since then the behavior of the `NonBlockingLzApp` would be as expected and the pathway wouldn't be blocked, but rather the message would be stored inside the `failedMessages`

Tools Used

Foundry, Tenderly, LayerZeroScan

Recommended Mitigation Steps

The minimum gas enforced to send for each and every `_lzSend` in the app should be enough to cover the worst-case scenario for the transaction to reach the first try/catch which is **here**.

I would advise the team to do extensive testing so this min gas is enforced.

Immediate fixes:

1. This is most easily fixed by overriding the `_lzSend` and extracting the gas passed from adapterParams with `_getGasLimit` and validating that it is above some minimum threshold.
2. Another option is specifying the minimum gas for each and every packetType and enforcing it as such.

I would default to the first option because the issue is twofold since there is the minimum gas that is common for all the packets, but there is also the minimum gas per packet since each packet has a different payload size and data structure, and it is being differently decoded and handled.

Note: This also applies to the transaction which when received on the destination chain is supposed to send another message, this callback message should also be validated.

When it comes to the default implementations inside the `OFTCoreV2` there are two packet types `PT_SEND` and `PT_SEND_AND_CALL` and there is the available configuration of `useCustomAdapterParams` which can enforce the minimum gas passed. This should all be configured properly.

Other occurrences

There are many occurrences of this issue in the TapiocaDao contracts, but applying option 1 I mentioned in the mitigation steps should solve the issue for all of them:

▶ Details

0xRektora (Tapioca) confirmed via duplicate issue 841

[H-18] `multiHopSellCollateral()` will fail due to call on an invalid market address causing bridged collateral to be locked up

Submitted by peakbolt

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/master/contracts/markets/singularity/Singularity.sol#L409-L427

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/master/contracts/markets/singularity/SGLLeverage.sol#L81

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/master/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L79-L108

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/master/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L227

`multiHopSellCollateral()` allows users to leverage down by selling the `TOFT` collateral on another chain and then send it to host chain (Arbitrum) for repayment of USDO loan.

However, it will fail as it tries to obtain the `repayableAmount` on the destination chain by calling `IMagnetar.getBorrowPartForAmount()` on a non-existing market. That is because Singularity/BigBang markets are only deployed on the host chain.

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/master/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L205-L227

```
function leverageDownInternal(
    uint256 amount,
    IUSDOBase.ILeverageSwapData memory swapData,
    IUSDOBase.ILeverageExternalContractsData memory external
    IUSDOBase.ILeverageLZData memory lzData,
    address leverageFor
) public payable {
    _unwrap(address(this), amount);

    //swap to USDO
    IERC20(erc20).approve(externalData.swapper, amount);
    ISwapper.SwapData memory _swapperData = ISwapper(externa
        .buildSwapData(erc20, swapData.tokenOut, amount, 0,
    (uint256 amountOut, ) = ISwapper(externalData.swapper).s
        _swapperData,
        swapData.amountOutMin,
        address(this),
        swapData.data
    );

    //@audit this call will fail as there is no market in de
    //repay
    uint256 repayableAmount = IMagnetar(externalData.magneta
        .getBorrowPartForAmount(externalData.srcMarket, amou
```

Impact

The issue will prevent users from using `multiHopSellCollateral()` to leverage down.

Furthermore the failure of the cross-chain transaction will cause the bridged collateral to be locked in the TOFT contract on a non-host chain as the refund mechanism will also revert and `retryMessage()` will continue to fail as this is a permanent error.

Proof of Concept

Consider the following scenario where a user leverage down by selling the collateral on Ethereum (a non-host chain).

1. User first triggers `Singularity.multiHopSellCollateral()` on host chain Arbitrum.
2. That will call `SGLLeverage.multiHopSellCollateral()`, which will conduct a cross chain message via `ITapiocaOFT(address(collateral)).sendForLeverage()` to bridge over and sell the collateral on Ethereum mainnet.
3. The collateral TOFT contract on Ethereum mainnet will receive the bridged collateral and cross-chain message via `_nonBlockingLzReceive()` and then `BaseTOFTLeverageModule.leverageDown()`.
4. The execution continues with `BaseTOFTLeverageModule.leverageDownInternal()`, but it will revert as it attempt to call `getBorrowPartForAmount()` for a non-existing market in Ethereum.
5. The bridgex collateral will be locked in the TOFT contract on Ethereum mainnet as the refund mechanism will also revert and `retryMessage()` will continue to fail as this is a permanent error.

Recommended Mitigation Steps

Obtain the repayable amount on the Arbitrum (host chain) where the BigBang/Singularity markets are deployed.

0xRektora (Tapioca) confirmed

[H-19] `twTAP.participate()` can be permanently frozen due to lack of access control on host-chain-only operations

Submitted by peakbolt

`twTAP` is a omnichain NFT (ONFT721) that will be deployed on all supported chains.

However, there are no access control for operations meant for execution on the host chain only, such as `participate()`, which mints `twTAP`.

The implication of not restricting `participate()` to host chain is that an attacker can lock `TAP` and participate on other chain to mint `twTAP` with a tokenId that does not exist on the host chain yet. The attacker can then send that `twTAP` to the host chain using the inherited `sendFrom()`, to permanently freeze the `twTAP` contract as `participate()` will fail when attempting to mint an existing `tokenId`.

It is important to restrict minting to the host chain so that `mintedTWTap` (which keeps track of last minted tokenId) is only incremented at one chain, to prevent duplicate tokenId. That is because the `twTAP` contracts on each chain have their own `mintedTWTap` variable and there is no mechanism to sync them.

Detailed Explanation

In `TwTAP`, there are no modifiers or checks to ensure `participte()` can only be called on the host chain. So we can use it to mint a `twTAP` on a non-host chain.
https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/governance/twTAP.sol#L252-L256

```
    function participate(
        address _participant,
        uint256 _amount,
        uint256 _duration
    ) external returns (uint256 tokenId) {
        require(_duration >= EPOCH_DURATION, "twTAP: Lock not a
```

The `tokenId` to be minted is determined by `mintedTWTap`, which is not synchronized across the chains.

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/governance/twTAP.sol#L309-L310

```
        function participate(
            ...
            //@audit tokenId to mint is obtained from `mintedTWTap`
            tokenId = ++mintedTWTap;
            _safeMint(_participant, tokenId);
```

Suppose on host chain, the last minted `tokenId` is `N`. From a non-host chain, we can use `sendFrom()` to send over a `twTAP` with `tokenId N+1` and mint a new `twTAP` with the same `tokenId` (see `_creditTo()` below). This will not increment `mintedTWTap` on the host chain, causing a de-sync.

```
24 <br>https:
25
26     function _creditTo(uint16, address _toAddress, uint _toke
27         require(!_exists(_tokenId) || (_exists(_tokenId) && E
28         if (!_exists(_tokenId)) {
29             //@audit transfering token N+1 will mint it as it
30             _safeMint(_toAddress, _tokenId);
31         } else {
32             _transfer(address(this), _toAddress, _tokenId);
33         }
34     }
```

On the host chain, `participate()` will always revert when it tries to mint the next `twTAP` with `tokenId N+1`, as it now exists on the host chain due to `sendFrom()`.

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/governance/twTAP.sol#L309-L310

```
function participate(
    ...
    tokenId = ++mintedTWTap;
    //@audit this will always revert when tokenId already ex
    _safeMint(_participant, tokenId);
```

Impact

An attacker will be able to permanent freeze the `twTAP.participate()`. This will prevent `TAP` holders from participating in the governance and from claiming rewards, causing loss of rewards to users.

Proof of Concept

Consider the following scenario,

1. Suppose we start with `twTAP.mintedTwTap == 0` on all the chains, so next tokenId will be `1`.
2. Attacker `participate()` with 1 TAP and mint `twTAP` on a non-host chain with `tokenId 1`.
3. Attacker sends the minted `twTAP` across to host chain using `twTAP.sendFrom()` to permanently freeze the `twTAP` contract.
4. On the host chain, the `twTAP` contract receives the cross chain message and mint a `twTAP` with `tokenId 1` to attacker as it does not exist on host chain yet. (Note this cross-chain transfer is part of Layer Zero ONFT71 mechanism)
5. Now on the host chain, we have a `twTAP` with `tokenId 1` but `mintedTwTap` is still `0`. That means when users try to `participate()` on the host chain, it will try to mint a `twTAP` with `tokenId 1`, and that will fail as it now exists on the host chain. At this point `participate()` will be permanently DoS, affecting governance and causing loss of rewards.
6. Note that the attacker can then transfer the `twTAP` back to the source chain and exit position to retrieve the locked `TAP` token. However, the host chain still remain frozen as the owner of `tokenId 1` will now be `twTAP` contract itself after the cross chain transfer.

Note that the attack is still possible even when `mintedTwTap > 0` on host chain as attacker just have to repeatly mint on the non-host chain till it obtain the required `tokenId`.

Recommended Mitigation Steps

Add in access control to prevent host-chain-only operations such as `participate()` from being executed on other chains .

0xRektora (Tapioca) confirmed

[H-20] `_liquidateUser()` should not re-use the same minimum swap amount out for multiple liquidation

Submitted by **peakbolt**, *also found by* **carrotsmuggler**, **Nyx**, **n1punp**, **Ack**, *and* **rvierdiiev**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/master/contracts/markets/singularity/SGLLiquidation.sol#L337-L340

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L603-L606

Vulnerability details

In Singularity and BigBang, the `minAssetAmount` in `_liquidateUser()` is provided by the liquidator as a slippage protection to ensure that the swap provides the specified `amountOut`. However, the same value is utilized even when `liquidate()` is used to liquidate multiple borrowers.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/master/contracts/markets/singularity/SGLLiquidation.sol#L337-L351

```
        function _liquidateUser(
            ...
            uint256 minAssetAmount = 0;
            if (dexData.length > 0) {
```

```
            //@audit the same minAssetAmount is incorrectly appl
            minAssetAmount = abi.decode(dexData, (uint256));
        }

        ISwapper.SwapData memory swapData = swapper.buildSwapDat
            collateralId,
            assetId,
            0,
            collateralShare,
            true,
            true
        );
        swapper.swap(swapData, minAssetAmount, address(this), "'
```

Impact

Using the same `minAssetAmount` (minimum amountOut for swap) for the liquidation of multiple borrowers will result in inaccurate slippage protection and transaction failure.

If `minAssetAmount` is too low, there will be insufficient slippage protection and the the liquidator and protocol could be short changed with a worse than expected swap.

If `minAssetAmount` is too high, the liquidation will fail as the swap will not be successful.

Proof of Concept

First scenario

1. Liquidator liquidates two loans X & Y using `liquidate()` , and set the `minAssetAmount` to be 1000 USDO.
2. Loan X liquidated collateral is worth 1000 USDO and the swap is completely successful with zero slippage.
3. However, Loan Y liquidated collateral is worth 5000 USDO, but due to low liquidity in the swap pool, it was swapped at 1000 USDO (`minAssetAmount`).

The result is that the liquidator will receive a fraction of the expected reward and the protocol gets repaid at 1/5 of the price, suffering a loss from the swap.

Second scenario

1. Liquidator liquidates two loans X & Y using `liquidate()`, and set the `minAssetAmount` to be 1000 USDO.
2. Loan X liquidated collateral is worth 1000 USDO and the swap is completely successful with zero slippage.
3. we suppose Loan Y's liquidated collateral is worth 300 USDO.

Now the `minAssetAmount` of 1000 USDO will be higher than the collateral, which is unlikely to be completed as it is higher than market price. That will revert the entire `liquidate()`, causing the liquidation of Loan X to fail as well.

Recommended Mitigation Steps

Update `liquidate()` to allow liquidator to pass in an array of `minAssetAmount` values that corresponding to the liquidated borrower.

An alternative, is to pass in the minimum expected price of the collateral and use that to compute the `minAssetAmount`.

0xRektora (Tapioca) confirmed via duplicate issue 122

[H-21] Incorrect liquidation reward computation causes excess liquidator rewards to be given

Submitted by peakbolt, also found by minhtrng, bin2chen, carrotsmuggler, 0x007, and 0xRobocop (1, 2)

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L577

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/singularity/SGLLiquidation.sol#L310-L314

In `_liquidateUser()` for BigBang and Singularity, the liquidator reward is derived by `_getCallerReward()`. However, it is incorrectly computed using `userBorrowPart[user]`, which is the portion of borrowed amount that does not include the accumulated fees (interests).

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L576-L580

```
uint256 callerReward = _getCallerReward(
    //@audit - userBorrowPart[user] is incorrect as it c
    userBorrowPart[user],
    startTVLInAsset,
    maxTVLInAsset
);
```

Using only `userBorrowPart[user]` is inconsistent with liquidation calculation in Market.sol#L423-L424, which is based on borrowed amount including accumulated fees.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L423-L424

```
function _isSolvent(
    address user,
    uint256 _exchangeRate
) internal view returns (bool) {
    ...
    return
        yieldBox.toAmount(
            collateralId,
            collateralShare *
                (EXCHANGE_RATE_PRECISION / FEE_PRECISION) *
                collateralizationRate,
            false
        ) >=
        //@audit - note that the collateralizion calculatior
        // Moved exchangeRate here instead of dividing the c
        (borrowPart * _totalBorrow.elastic * _exchangeRate)
```

```
                _totalBorrow.base;
        }
```

As the protocol uses a dynamic liquidation incentives mechanism (see below), the liquidator will be given more rewards than required if the liquidator reward is derived by borrowed amount without accumulated fees. That is because the dynamic liquidation incentives mechanism decreases the rewards as it reaches 100% LTV. So computing the liquidator rewards using a lower value (without fees) actually gives liquidator a higher portion of the rewards.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L442-L462

```
    function _getCallerReward(
        uint256 borrowed,
        uint256 startTVLInAsset,
        uint256 maxTVLInAsset
    ) internal view returns (uint256) {
        if (borrowed == 0) return 0;
        if (startTVLInAsset == 0) return 0;

        if (borrowed < startTVLInAsset) return 0;
        if (borrowed >= maxTVLInAsset) return minLiquidatorRewar

        uint256 rewardPercentage = ((borrowed - startTVLInAsset)
            FEE_PRECISION) / (maxTVLInAsset - startTVLInAsset);

        int256 diff = int256(minLiquidatorReward) - int256(maxLi
        int256 reward = (diff * int256(rewardPercentage)) /
            int256(FEE_PRECISION) +
            int256(maxLiquidatorReward);

        return uint256(reward);
    }
```

Impact

The protocol is shortchanged as it gives liquidator more rewards than required.

Proof of Concept

1. Add the following console.log to BigBang.sol#L581`

```
console.log("    callerReward (without fees) = \t %d (ac

callerReward= _getCallerReward(
    //userBorrowPart[user],
    //@audit borrowed amount with fees
    (userBorrowPart[user] * totalBorrow.elastic) / total
    startTVLInAsset,
    maxTVLInAsset
);
console.log("    callerReward (with fees)  = \t %d (expe
```

2. Add and run the following test in `bigBang.test.ts`. The console.log will show that the expected liquidator reward is lower when computed using borrowed amount with fees.

▶ Details

Recommended Mitigation Steps

Change BigBang.sol#L576-L580, SGLLiquidation.sol#L310-L314, Market.sol#L364 from

```
uint256 callerReward = _getCallerReward(
    userBorrowPart[user],
    startTVLInAsset,
    maxTVLInAsset
);
```

to

```
uint256 callerReward = _getCallerReward(
    (userBorrowPart[user] * totalBorrow.elastic) / total
    startTVLInAsset,
    maxTVLInAsset
);
```

0xRektora (Tapioca) confirmed via duplicate issue 89

[H-22] Lack of safety buffer between liquidation threshold and LTV ratio for borrowers to prevent unfair liquidations

Submitted by peakbolt

In BigBang and Singularity, there is no safety buffer between liquidation threshold and LTV ratio, to protects borrowers from being immediately liquidated due to minor market movement when the loan is taked out at max LTV.

The safety buffer also ensure that the loans can be returned to a healthy state after the first liquidation. Otherwise, the loan can be liquidated repeatly as it will remain undercollateralized after the first liquidation.

Detailed Explanation

The `collateralizationRate` determines the LTV ratio for the max amount of assets that can be borrowed with the specific collateral. This check is implemented in `_isSolvent()` as shown below.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L402-L425

```
    function _isSolvent(
        address user,
        uint256 _exchangeRate
    ) internal view returns (bool) {
        // accrue must have already been called!
        uint256 borrowPart = userBorrowPart[user];
        if (borrowPart == 0) return true;
        uint256 collateralShare = userCollateralShare[user];
        if (collateralShare == 0) return false;

        Rebase memory _totalBorrow = totalBorrow;

        return
            yieldBox.toAmount(
```

```
                collateralId,
                collateralShare *
                    (EXCHANGE_RATE_PRECISION / FEE_PRECISION) *
                    collateralizationRate,
                false
            ) >=
            // Moved exchangeRate here instead of dividing the c
            (borrowPart * _totalBorrow.elastic * _exchangeRate)
                _totalBorrow.base;
    }
```

However, the liquidation start threshold, which is supposed to be higher (e.g. 80%) than LTV ratio (e.g. 75%), is actually using the same `collateralizationRate` value. We can see that `computeClosingFactor()` allow liquidation to start when the loan is at max LTV.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L283-L284

```
        uint256 liquidationStartsAt = (collateralPartInAssetScal
            collateralizationRate) / (10 ** ratesPrecision);
```

Impact

Borrowers can be unfairly liquidated and penalized due to minor market movement when taking loan at max LTV. Also loan can be repeatedly liquidated regardless of closing factor as it does not return to healthy state after the first liquidation.

Proof of Concept

Consider the following scenario,

1. Borrower take out loan at max LTV (75%).
2. Immediately after the loan is taken out, the collateral value dropped slightly due to minor market movement and the loan is now at 75.000001% LTV.
3. However, as the liquidation start threshold begins to at 75% LTV, bots start to liquidate the loan, before the borrower could react and repay the loan.

4. The liquidation will cause the loan to remain undercollateralized despite the closing factor.
5. As the loan is still unhealthy, the bots will then be able to repeatly liquidate the loan.
6. Borrower is unfairly penalized and suffers losses due to the liquidations.

Recommended Mitigation Steps

Implement the liquidation threshold as a separate state variable and ensure it is higher than LTV to provide a safety buffer for borrowers.

cryptotechmaker (Tapioca) confirmed and commented:

> The user is not liquidated for his entire position but only for the amount necessary for the loan to become solvent again.

> Loaning up to the collateralization rate threshold is up to the user and opening such an edging position comes with some risks that the user should be aware of.

> However, adding the buffer seems fair. It can remain as a 'High'.

[H-23] Refund mechanism for failed cross-chain transactions does not work

Submitted by **peakbolt**, *also found by* **Kaysoft**, **windhustler**, *carrotsmuggler (*1, 2*),* **xuwinnie**, *and* **cergyk**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOLeverageModule.sol#L180-L185

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOMarketModule.sol#L178-L186

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/mo

dules/USDOOptionsModule.sol#L187-L197

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L195-L200

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTMarketModule.sol#L170-L175

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTOptionsModule.sol#L202-L212

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L163-L168

There is a refund mechanism in `USDO` and `TOFT` modules that will return funds when the execution on the destination chain fails.

It happens when `module.delegatecall()` fails, where the following code (see below) will trigger a refund of the bridged fund to the user. After that a revert is then 'forwarded' to the main executor contract (`BaseUSDO` or `BaseTOFT`).

However, the issue is that the revert will also reverse the refund even when the revert is forwarded.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOLeverageModule.sol#L180-L185

```
            if (!success) {
                if (balanceAfter - balanceBefore >= amount) {
                    IERC20(address(this)).safeTransfer(leverageFor,
                }

                //@audit - this revert will actually reverse the ref
                revert(_getRevertMsg(reason)); //forward revert beca
```

```
        }
```

Although the main executor contract will `_storeFailedMessage()` to allow users to `retryMessage()` and re-execute the failed transaction, it will not go through if the error is permanent. That means the `retryMessage()` will also revert and there is no way to recover the funds.

Impact

User will lose their bridged fund if the cross chain execution encounters a permanent error, which will permanently lock up the bridged funds in the contract as there is no way to recover it.

Proof of Concept

1. Add a `revert()` in `leverageUpInternal()` within `USDOLeverageModule.sol#L197` as follows, to simulate a permanent failure for the remote execution at destination chain.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOLeverageModule.sol#L197.

```
    function leverageUpInternal(
            uint256 amount,
            IUSDOBase.ILeverageSwapData memory swapData,
            IUSDOBase.ILeverageExternalContractsData memory external
            IUSDOBase.ILeverageLZData memory lzData,
            address leverageFor
        ) public payable {

            //@audit - to simulate a permanent failure for this remc
            revert();

            ...
        }
```

2. Add the following `console.log` to singularity.test.ts#L4113

```
console.log("USDO_10 balance for deployer.address (e
```

3. Run the test case `'should bounce between 2 chains'` under `'multiHopBuyCollateral()'` tests in `singularity.test.ts`. It will show that the `deployer.address` fails to receive the refund amount.

Recommended Mitigation Steps

Implement a 'pull' mechanism for users to withdraw the refund instead of 'pushing' to the user.

That can be done by using a a new state variable within `USDO` and `TOFT` to store the refund amount for the transaction with the corresponding `payloadHash` for `failedMessages` mapping.

Checks must be implemented to ensure that if user withdraws the refund, the corresponding `failedMessages` entry is cleared so that the user cannot retry the transaction again.

Similarly, if `retryMessage()` is used to re-execute the transaction successfully, the refund amount in the new state variable should be cleared.

0xRektora (Tapioca) confirmed via duplicate issue #1410

[H-24] Incorrect formula used in function `Market.computeClosingFactor()`

Submitted by **KIntern_NA**, *also found by* **carrotsmuggler** *and* **0xRobocop**

Incorrect amount of assets that will be liquidated

Proof of Concept

Function `BigBang._liquidateUser()` is used to liquidate an under-collateralization position in the market. This function calls `BigBang._updateBorrowAndCollateralShare()` to calculate the amount of

`borrowPart` and `collateralShare` that will be removed from the user's position and update the storage.

The amount of `borrowPart` to be removed can be calculated using the function `Market.computeClosingFactor()`. This amount will then be converted to `borrowAmount`, which is the corresponding elastic amount, and be used to determine the amount of `collateralShare` that needs to be removed.

- Link to function

However, the returned value from `Market.computeClosingFactor()` is incorrect, which leads to the wrong update for the user's position.

To prove the statement above, let's denote:

- `x`: The elastic amount that will be removed to execute the liquidation.
- `userElastic` and `userElastic'`: The elastic amount corresponding to `userBorrowPart[user]` before and after the liquidation.
- `collateralShare` and `collateralShare'`: The value of `userCollateralShare[user]` before and after the liquidation.
- Following the implementation of `yieldBox.toAmount()` and `yieldBox.toShare()`, in one transaction we can denote that:
    - `yieldBox.toAmount()`: A multiplication expression with a constant `C`.
    - `yieldBox.toShare()`: A division expression with constant `C`.

Following the update of these variables depicted in the function `BigBang._updateBorrowAndCollateralShare()`, we have:

- $userElastic' = userElastic - x$
- $collateralShare' = collateralShare - \frac{x \times (1+liquidationMultiplier)*\frac{exchangeRate}{10^{18}}}{C}$

After the liquidation, the function `Market._isSolvent(user)` must return true. In other words, at least the following equation should hold:

- $C \times (collateralShare' \times \frac{collateralRate}{10^5} \times \frac{10^{18}}{exchangeRate}) = userElastic'$

Solving the equation, we get:

1. $C \times (collateralShare' \times \frac{collateralRate}{10^5} \times \frac{10^{18}}{exchangeRate}) = userElastic'$
2. $C \times collateralShare \times \frac{collateralRate}{10^5} \times \frac{10^{18}}{exchangeRate} - x \times (1 + \frac{liquidationMultiplier}{10^5}) \times \frac{collateralizationRate}{10^5} = userElastic - x$
3. $x = \frac{userElastic - C \times collateralShare \times \frac{collateralRate}{10^5} \times \frac{10^{18}}{exchangeRate}}{1 - (1 + \frac{liquidationMultiplier}{10^5}) * \frac{collateralizationRate}{10^5}}$

So, the returned value of the function `Market.computeClosingFactor()` should be the corresponding base amount of `x` (`totalBorrow.toBase(x, false)`).

Comparing it to the current implementation of `computeClosingFactor()` , we can see the issues are:

- The implementation uses the `borrowPart` in the numerator instead of the corresponding elastic amount of `borrowPart` .
- The multiplication with `borrowPartDecimals` and `collateralPartDecimals` doesn't make sense since these decimals can be different and may cause the numerator to underflow.

Recommended Mitigation Steps

Correct the formula of function `computeClosingFactor()` following the section "Proof of Concept".

cryptotechmaker (Tapioca) confirmed

[H-25] Overflow risk in Market contract

Submitted by KIntern_NA

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L415-L421

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L390-L396

Actions of users (borrow, repay, removeCollateral, ...) in Martket contract might be reverted by overflow, resulting in their funds might be frozen.

Proof of concept

Function `_isSolvent` in `Market` contract use conversion from share to amount of yieldBox.

```
yieldBox.toAmount(
    collateralId,
    collateralShare *
        (EXCHANGE_RATE_PRECISION / FEE_PRECISION) *
        collateralizationRate,
    false
)
```

It will trigger `_toAmount` function in `YieldBoxRebase` contract

```
function _toAmount(
    uint256 share,
    uint256 totalShares_,
    uint256 totalAmount,
    bool roundUp
) internal pure returns (uint256 amount) {
    totalAmount++;
    totalShares_ += 1e8;

    amount = (share * totalAmount) / totalShares_;

    if (roundUp && (amount * totalShares_) / totalAmount < share
        amount++;
    }
```

```
    }
```

The calculation `amount = (share * totalAmount) / totalShares_` might be overflow because `share * totalAmount = collateralShare * (EXCHANGE_RATE_PRECISION / FEE_PRECISION) * collateralizationRate * totalAmount`

In the default condition,
`EXCHANGE_RATE_PRECISION` = 1e18,
`FEE_PRECISION` = 1e5,
`collateralizationRate` = 0.75e18

The `collateralShare` is equal to around `1e8 * collateralAmount` by default (because `totalAmount++; totalShares_ += 1e8;` is present in the `_toAmount` function).

=> `share * totalAmount` **~= (collateralAmount * 1e8) * (1e18 / 1e5) * 0.75e18 * totalAmount = collateralAmount * totalAmount * 0.75e39**

This formula will overflow when `collateralAmount * totalAmount` > 1.5e38. This situation can occur easily with 18-decimal collateral. As a consequence, user transactions will revert due to overflow, resulting in the freezing of market functionalities.

The same issue applies to the calculation of `_computeMaxBorrowableAmount` in the Market contract.

Recommended Mitigation Steps

Reduce some variables used to trigger yieldBox.toAmount(), such as `EXCHANGE_RATE_PRECISION` and `collateralizationRate`, and use these variables to calculate with the obtained amount. Example, the expected amount can be calculated as:

```
    yieldBox.toAmount(
        collateralId,
        collateralShare
```

```
        false
    ) * (EXCHANGE_RATE_PRECISION / FEE_PRECISION) * collateralizatic
```

[0xRektora (Tapioca) confirmed]

[H-26] Not enough TAP tokens to exercise if a user participates and exercises in the same epoch

Submitted by [KIntern_NA]

Users were unable to purchase their deserved amount of TAPs

Proof of Concept

During each `epoch` and for a specific `sglAssetID`, there is a fixed amount of TAP tokens that will be minted and stored in the STORAGE mapping `singularityGauges[epoch][sglAssetID]`. Users have the option to purchase these TAP tokens by first calling the function `TapiocaOptionBroker.participate()` and then executing `TapiocaOptionBroker.exerciseOption()` before the position expires to buy TAPs at a discounted price. The amount of TAP tokens that a user can purchase with each position can be calculated using the formula:

```
eligibleTapAmount = position.amount * gaugeTotalForEpoch / total

- position.amount: The locked amount of the position in `sglAsse
- gaugeTotalForEpoch: The total number of TAP tokens that can be
- totalPoolDeposited: The total locked amount of all positions i
```

The flaw arises when a user who participates in `sglAssetId` in the current epoch can immediately call `exerciseOption()` to purchase the TAP tokens. This results in a situation where the participants cannot exercise their expected TAP tokens.

For example:

- Both Alice and Bob participate in the broker with `position.amount = 1`.

- The amount of TAP tokens allocated for the current epoch is `gaugeTotalForEpoch = 60`.
- Alice calls `exerciseOption()` to buy `eligibleAmount = 1 * 60 / 2 = 30` TAPs.
- In the same epoch, Candice participates in the broker with `position.amount = 1` and immediately calls `exerciseOption()`. She will buy `eligibleAmount = 1 * 60 / 3 = 20` TAPs.
- When Bob calls `exerciseOption`, he can buy `eligibleAmount = 1 * 60 / 3 = 20` TAPs, but this cannot happen since if Bob decides to buy 20 TAPs, the total minted amount of TAPs will exceed `gaugeTotalForEpoch` (30 + 20 + 20 = 70 > 60), resulting in a revert.

Recommended Mitigation Steps

Consider developing a technique similar to the one implemented in `twTAP.sol` for storing the `netAmounts`. When a user participates in the broker, perform the following actions:

- `netAmounts[block.timestamp+1] += lock.amount`
- `netAmounts[lockTime+lockDuration] += lock.amount`

0xRektora (Tapioca) confirmed

[H-27] Attacker can pass duplicated reward token addresses to steal the reward of contract `twTAP.sol`

Submitted by **KIntern_NA**, *also found by* **bin2chen** *and* **glcanvas**

The attacker can exploit the contract `twTAP.sol` to steal rewards.

Proof of Concept

The function `twTAP.claimAndSendRewards() -> twTAP._claimRewardsOn()` is intended for users who utilize the cross-chain message of `BaseTOFT.sol` to claim a specific set of reward tokens.

```
519
520 function _claimRewardsOn(
521     uint256 _tokenId,
522     address _to,
523     IERC20[] memory _rewardTokens
524 ) internal {
525     uint256[] memory amounts = claimable(_tokenId);
526     unchecked {
527         uint256 len = _rewardTokens.length;
528         for (uint256 i = 0; i < len; ) {
529             uint256 claimableIndex = rewardTokenIndex[_rewar
530             uint256 amount = amounts[i];
531
532             if (amount > 0) {
533                 // Math is safe: `amount` calculated safely
534                 claimed[_tokenId][claimableIndex] += amount;
535                 rewardTokens[claimableIndex].safeTransfer(_t
536             }
537             ++i;
538         }
539     }
540 }
```

The internal function iterates through the list of reward tokens specified by the user after calculating the claimable amount for each token in the STORAGE array `twTAP.rewardTokens[]`. Unfortunately, there is no check if the `_rewardTokens` contain duplicated reward tokens, and the function `claimable(_tokenId)` is not called after each iteration, which allows the attacker to manipulate the function call using the same reward address repeatedly.

For example,

- STORAGE array `rewardTokens[] = [usdc, usdt]`
- The function `_claimRewardsOn()` is called with `_rewardTokens[] = [usdt, usdt]`. In each iteration, the `claimableIndex` will be `rewardTokenIndex[usdc] = 0`, which transfers the usdt two times to the attacker.

Recommended Mitigation Steps

One solution to mitigate this issue is to require the MEMORY array `_rewardTokens` to be sorted in ascending order.

```
function _claimRewardsOn(
    uint256 _tokenId,
    address _to,
    IERC20[] memory _rewardTokens
) internal {
    uint256[] memory amounts = claimable(_tokenId);
    unchecked {
        uint256 len = _rewardTokens.length;
        for (uint256 i = 0; i < len; ) {
            // CHANGE HERE
            if (i != 0) {
                require(_rewardTokens[i] > _rewardTokens[i-1]);
            }

            uint256 claimableIndex = rewardTokenIndex[_rewardTok
            uint256 amount = amounts[i];

            if (amount > 0) {
                // Math is safe: `amount` calculated safely in `
                claimed[_tokenId][claimableIndex] += amount;
                rewardTokens[claimableIndex].safeTransfer(_to, a
            }
            ++i;
        }
    }
}
```

By ensuring that the reward tokens are sorted in ascending order, we can prevent the exploit where the attacker claims the same reward token multiple times and effectively mitigate the vulnerability.

0xRektora (Tapioca) confirmed via duplicate issue 1304

[H-28] TOFT and USDO Modules Can Be Selfdestructed

Submitted by Ack, *also found by* BPZ, Breeje, ladboy233, offside0011, Kaysoft, 0x73696d616f, 0xrugpull_detector, carrotsmuggler, CrypticShepherd, ACai,

kodyvim, *and* cergyk

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L184-L193

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTMarketModule.sol#L160-L168

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTOptionsModule.sol#L189-L200>

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTStrategyModule.sol#L152-L162

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOLeverageModule.sol#L169-L1788

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOMarketModule.sol#L168-L176

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/usd0/modules/USDOOptionsModule.sol#L174-L185

All TOFT and USDO modules have public functions that allow an attacker to supply an address `module` that is later used as a destination for a delegatecall. This can point to an attacker-controlled contract that is used to selfdestruct the module.

```
// USDOLeverageModule:leverageUp
function leverageUp(
    address module,
    uint16 _srcChainId,
    bytes memory _srcAddress,
```

```
        uint64 _nonce,
        bytes memory _payload
    ) public {
        // .. snip ..
        (bool success, bytes memory reason) = module.delegatecal
            abi.encodeWithSelector(
                this.leverageUpInternal.selector,
                amount,
                swapData,
                externalData,
                lzData,
                leverageFor
            )
        );

        if (!success) {
            if (balanceAfter - balanceBefore >= amount) {
                IERC20(address(this)).safeTransfer(leverageFor,
            }
            revert(_getRevertMsg(reason)); //forward revert beca
        }
        // .. snip ..
    }
```

Impact

Both BaseTOFT and BaseUSDO initialize the module addresses to state variables in the constructor. Because there are no setter functions to adjust these variables post-deployment, the modules are permanently locked to the addresses specified in the constructor. If those addresses are selfdestructed, the modules are rendered unusable and all calls to these modules will revert. This cannot be repaired.

[BaseUSDO.sol:constructor]

```
    // BaseUSDO.sol:constructor
    constructor(
        address _lzEndpoint,
        IYieldBoxBase _yieldBox,
        address _owner,
        address payable _leverageModule,
        address payable _marketModule,
        address payable _optionsModule
    ) BaseUSDOStorage(_lzEndpoint, _yieldBox) ERC20Permit("USDO'
```

```
            leverageModule = USDOLeverageModule(_leverageModule);
            marketModule = USDOMarketModule(_marketModule);
            optionsModule = USDOOptionsModule(_optionsModule);


            transferOwnership(_owner);
        }
```

Proof of Concept

Attacker can deploy the `Exploit` contract below, and then call each of the vulnerable functions with the address of the `Exploit` contract as the `module` parameter. This will cause the module to selfdestruct, rendering it unusable.

```
pragma solidity ^0.8.18;

contract Exploit {
    address payable constant attacker = payable(address(0xbadbak
    fallback() external payable {
        selfdestruct(attacker);
    }
}
```

Recommended Mitigation Steps

The `module` parameter should be removed from the calldata in each of the vulnerable functions. Since the context of the call into these functions are designed to be delegatecalls and the storage layouts of the modules and the Base contracts are the same, the `module` address can be retreived from storage instead. This will prevent attackers from supplying arbitrary addresses as delegatecall destinations.

0xRektora (Tapioca) confirmed via duplicate issue 146

[H-29] Exercise option cross chain message in the (m)TapiocaOFT will always revert in the destination, losing debited funds in the source chain

Submitted by **0x73696d616f**, *also found by* **KIntern_NA** *and* **bin2chen**

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L539-L545

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTOptionsModule.sol#L153-L159

Exercise option cross chain message in the `(m)TapiocaOFT` will always revert in the destination, but works in the source chain, where it debits the funds from users. Thus, these funds will not be credited in the destination and are forever lost.

Proof of Concept

In the `BaseTOFT`, if the packet from the received cross chain message in `lzReceive()` is of type `PT_TAP_EXERCISE`, it delegate calls to the `BaseTOFTOptionsModule`:

```
    function _nonblockingLzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) internal virtual override {
        uint256 packetType = _payload.toUint256(0);
        ...
        } else if (packetType == PT_TAP_EXERCISE) {
            _executeOnDestination(
                Module.Options,
                abi.encodeWithSelector(
                    BaseTOFTOptionsModule.exercise.selector,
                    _srcChainId,
                    _srcAddress,
                    _nonce,
                    _payload
                ),
                _srcChainId,
                _srcAddress,
                _nonce,
                _payload
            );
```

```
        ...
```

In the `BaseTOFTOptionsModule`, the `exercise()` function is declared as:

```
    function exercise(
        address module,
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) public {
        ...
    }
```

Notice that the `address module` argument is specified in the `exercise()` function declaration, but not in the `_nonBlockingLzReceive()` call to it. This will make the message always revert because it fails when decoding the arguments to the function call, due to the extra `address module` argument.

The following POC illustrates this behaviour. The `exerciseOption()` cross chain message fails on the destination:

▶ Details

Tools Used

Vscode, Foundry

Recommended Mitigation Steps

Adding the extra module parameter when encoding the function call in `_nonBlockingLzReceive()` would be vulnerable to someone calling the `BaseTOFTOptionsModule` directly on function `exercise()` with a malicious `module` argument. It's safer to remove the `module` argument and call `exerciseInternal()` directly, which should work since it's a `public` function.

```
    function _nonblockingLzReceive(
        uint16 _srcChainId,
```

```
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) internal virtual override {
        uint256 packetType = _payload.toUint256(0);
        ...
        } else if (packetType == PT_TAP_EXERCISE) {
            _executeOnDestination(
                Module.Options,
                abi.encodeWithSelector(
                    BaseTOFTOptionsModule.exercise.selector,
                    address(optionsModule), // here
                    _srcChainId,
                    _srcAddress,
                    _nonce,
                    _payload
                ),
                _srcChainId,
                _srcAddress,
                _nonce,
                _payload
            );
        ...
```

[0xRektora (Tapioca) confirmed]

[H-30] `utilization` for `_getInterestRate()` does not factor in interest

Submitted by [ItsNio], *also found by* [ItsNio] *and* [SaeedAlipoor01988]

The calculation for `utilization` in `_getInterestRate()` does not factor in the accrued interest. This leads to `_accrueInfo.interestPerSecond` being under-represented, and leading to incorrect interest rate calculation and potentially endangering conditions such as `utilization > maximumTargetUtilization` on line [124].

Proof of Concept

The calculation for `utilization` in the `_getInterestRate()` function for `SGLCommon.sol` occurs on lines 61-64 as a portion of the `fullAssetAmount` (which is also problematic) and the `_totalBorrow.elastic`. However, `_totalBorrow.elastic` is accrued by interest on line 99. This accrued amount is not factored into the calculation for `utilization`, which will be used to update the new interest rate, as purposed by the comment on line 111.

Recommended Mitigation Steps

Factor in the interest accrual into the `utilization` calculation:

```
...
        // Accrue interest
        extraAmount =
            (uint256(_totalBorrow.elastic) *
                _accrueInfo.interestPerSecond *
                elapsedTime) /
            1e18;
        _totalBorrow.elastic += uint128(extraAmount);

    +    uint256 fullAssetAmount = yieldBox.toAmount(
    +        assetId,
    +        _totalAsset.elastic,
    +        false
    +    ) + _totalBorrow.elastic;
        //@audit utilization factors in accrual
    +    utilization = fullAssetAmount == 0
    +    ? 0
    +        : (uint256(_totalBorrow.elastic) * UTILIZATION_PREC
    +        fullAssetAmount;
...
```

OxRektora (Tapioca) confirmed

[H-31] Collateral can be locked in BigBang contract when `debtStartPoint` is nonzero

Submitted by zzzitron, *also found by* minhtrng, RedOneN, kutugu, bin2chen, OxSky, Oxrugpull_detector, mojito_auditor, plainshift, KIntern_NA,

carrotsmuggler, zzebra83, 0xRobocop, *and* chaduke

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/Penrose.sol#L395-L397

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L242-L255

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L180-L201

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L512-L520

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L309-L317

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L263-L271

Following conditions have to be met for this issue to happen:

- This issue occurs when the BigBang market is not an ETH market.
- `Penrose.registerBigBang()` being called with `data` param where `data.debtStartPoint` is nonzero.
- The first borrower borrows using `BigBang.borrow()`, with function param `amount` (borrow amount) has to be less than `debtStartPoint`.

Now `BigBang.getDebtRate()` will always revert and the collateral from the first borrower is locked, because `BigBang.getDebtRate()` is used in

`BigBang._accrue()`, and `BigBang._accrue()` is used in every function that involves totalBorrow like in `BigBang.liquidate()`, `BigBang.repay()`.

The reason for the revert is that in `BigBang.getDebtRate()`, `totalBorrow.elastic` which gets assigned to the variable `_currentDebt` (line 186 BigBang.sol) will not be 0, and then on line 192 in the BigBang contract, the `_currentDebt` is smaller than `debtStartPoint` which causes the revert.

As a consequence the collateral is trapped as repay or liquidate requires to call accrue before hand.

Proof of Concept

The following gist contains a proof of concept to demonstrate this issue. A non-ETH bigbang market (wbtc market) is deployed with `Penrose::registerBigBang`. Note that the `debtStartPoint` parameter in the init data is non-zero (set to be 1e18).

First we set up the primary eth market: Some weth is minted and deposited to the ETH market. Then some assets were borrowed against the collateral. This is necessary condition for this bug to happen, which is the ETH market to have some borrowed asset. However, this condition is very likely to be fulfilled, as the primary ETH market would be deployed before any non-eth market.

Now, an innocent user is adding collateral and borrows in the non-eth market (the wbtc market). The issue occurs when the user borrows less than the `debtStartPoint`. If the user should borrow less than the `debtStartPoint`, the `BigBang::accrue` will revert and the collateral is trapped in this Market.

https://gist.github.com/zzzitron/a6d6377b73130819f15f1e5a2e2a2ba9

The bug happens here in the line 192 in the `BigBang`.

```
179     /// @notice returns the current debt rate
180     function getDebtRate() public view returns (uint256) {
181         if (_isEthMarket) return penrose.bigBangEthDebtRate(
182         if (totalBorrow.elastic == 0) return minDebtRate;
183 
184         uint256 _ethMarketTotalDebt = BigBang(penrose.bigBar
```

```
185             .getTotalDebt();
186         uint256 _currentDebt = totalBorrow.elastic;
187         uint256 _maxDebtPoint = (_ethMarketTotalDebt *
188             debtRateAgainstEthMarket) / 1e18;
189
190         if (_currentDebt >= _maxDebtPoint) return maxDebtRat
191
192         uint256 debtPercentage = ((_currentDebt - debtStartI
193             DEBT_PRECISION) / (_maxDebtPoint - debtStartPoir
194         uint256 debt = ((maxDebtRate - minDebtRate) * debtPe
195             DEBT_PRECISION +
196             minDebtRate;
197
```

Recommended Mitigation Steps

Consider adding a require statement to `BigBang.borrow()` to make sure that the borrow amount has to be >= `debtStartPoint`.

```
// BigBang
// borrow
247         require(amount >= debtStartPoint);
```

0xRektora (Tapioca) confirmed

[H-32] Reentrancy in `USDO.flashLoan()`, enabling an attacker to borrow unlimited USDO exceeding the max borrow limit

Submitted by zzzitron, *also found by* RedOneN, unsafesol, GalloDaSballo, kodyvim, ayeslick, andy, *and* dirk_y

Due to an reentrancy attack vector, an attacker can flashLoan an unlimited amount of USDO. For example the attacker can create a malicious contract as the `receiver`, to execute the attack via the `onFlashLoan` callback (line 94 USDO.sol).

The exploit works because `USDO.flashLoan()` is missing a reentrancy protection (modifier).

As a result an unlimited amount of USDO can be borrowed by an attacker via the flashLoan exploit described above.

Proof of Concept

Here is a POC that shows an exploit:

https://gist.github.com/zzzitron/a121bc1ba8cc947d927d4629a90f7991

To run the exploit add this malicious contract into the contracts folder:

https://gist.github.com/zzzitron/8de3be7ddf674cc19a6272b59cfccde1

Recommended Mitigation Steps

Consider adding some reentrancy protection modifier to `USDO.flashLoan()`.

0xRektora (Tapioca) confirmed, but disagreed with severity and commented:

> Should be `High` severity, could really harm the protocol.

LSDan (Judge) increased severity to High

[H-33] `BaseTOFTLeverageModule.sol`: `leverageDownInternal` tries to burn tokens from wrong address

Submitted by carrotsmuggler*, also found by* xuwinnie

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L212

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTLeverageModule.sol#L269-L277

The function `sendForLeverage` is used to interact with the USDO token on a different chain. Lets assume the origin of the tx is chain A, and the destination is chain B. The BaseTOFT contract sends a message through the lz endpoints to make a call in the destination chain.

The flow of control is as follows:

Chain A : user -call-> BaseTOFT.sol: `sendForLeverage` -delegateCall-> BaseTOFTLeverageModule.sol: `sendForLeverage` -call-> lzEndpointA

Chain B : lzEndpointB -call-> BaseTOFT.sol: `_nonblockingLzReceive` -delegateCall-> BaseTOFTLeverageModule.sol: `leverageDown` -delegateCall-> `leverageDownInternal`

For the last call to `leverageDownInternal` , the `msg.sender` is the lzEndpointB. This is because all the calls since then have been delegate calls, and thus msg.sender has not been able to change. We analyze the `leverageDownInternal` function in this context.

```
    function leverageDownInternal(
            uint256 amount,
            IUSDOBase.ILeverageSwapData memory swapData,
            IUSDOBase.ILeverageExternalContractsData memory external
            IUSDOBase.ILeverageLZData memory lzData,
            address leverageFor
        ) public payable {
            _unwrap(address(this), amount);
```

The very first operation is to do an unwrap of the mTapiocaOFT token. This is done by calling `_unwrap` defined in the **same contract** as shown.

```
    function _unwrap(address _toAddress, uint256 _amount) private {
        _burn(msg.sender, _amount);

        if (erc20 == address(0)) {
            _safeTransferETH(_toAddress, _amount);
        } else {
            IERC20(erc20).safeTransfer(_toAddress, _amount);
```

```
        }
    }
```

Here we see the contract is trying to burn tokens from the `msg.sender` address. But the issue is in this context, the `msg.sender` is the lzEndpoint on chain B who is doing the call, and they dont have any TOFT tokens there. Thus this call will revert.

The TOFT tokens are actually held within the same contract where the execution is happening. This is because in the `leverageDown` function, we see the contract credit itself with TOFT tokens.

```
    if (!credited) {
        _creditTo(_srcChainId, address(this), amount);
        creditedPackets[_srcChainId][_srcAddress][_nonce] = true;
    }
```

Thus the tokens are actually present in `address(this)` and not in `msg.sender`. Thus the burn should be done from `address(this)` and not `msg.sender`. Thus all cross chain calls for this function will fail and revert.

Since this leads to broken functionality, this is considered a high severity issue.

Proof of Concept

Since no test exists for the `sendForLeverage` function, no POC is provided. However the flow of control and detailed explanation is provided above.

Recommended Mitigation Steps

Run `_burn(address(this),amount)` to burn the tokens instead of unwrapping. Then do the eth/erc20 transfer from the contract.

0xRektora (Tapioca) confirmed via duplicate issue 725

[H-34] `BaseTOFT.sol : retrieveFromStrategy` can be used

to manipulate other user's positions due to absent approval check

Submitted by carrotsmuggler*, also found by* xuwinnie*,* peakbolt*, and* 0x73696d616f

The function `retrieveFromStrategy` is used to trigger a removal of TOFT tokens from a strategy on a different chain. The function takes the parameter `from`, which is the account whose tokens will be retrieved.

The main issue is that anyone can call this function with any address passed to the `from` parameter. There is no allowance check on the chain, allowing this operation. Let's walk through the steps to see how this is executed.

`BaseTOFT.sol : retrieveFromStrategy` is called by the attacker with a `from` address of the victim. This function calls the `retrieveFromStrategy` function in the strategy module.

```
    _executeModule(
        Module.Strategy,
        abi.encodeWithSelector(
            BaseTOFTStrategyModule.retrieveFromStrategy.selector,
            from,
            amount,
            share,
            assetId,
            lzDstChainId,
            zroPaymentAddress,
            airdropAdapterParam
        ),
        false
    );
```

`BaseTOFTStrategyModule.sol : retrieveFromStrategy` is called. This function packs some data and sends it forward to the lz endpoint. Point to note, is that no approval check is done for the `msg.sender` of this whole setup yet.

```
    bytes memory lzPayload = abi.encode(
        PT_YB_RETRIEVE_STRAT,
```

```
        LzLib.addressToBytes32(_from),
        toAddress,
        amount,
        share,
        assetId,
        zroPaymentAddress
    );
    // lzsend(...)
```

After the message is sent, the **lzendpoint** on the receiving chain will call the TOFT contract again. Now, the `msg.sender` is **not** the attacker, but is instead the lzendpoint! The endpoint call gets delegated to the `strategyWithdraw` function in the Strategy module.

```
     (
        ,
        bytes32 from,
        ,
        uint256 _amount,
        uint256 _share,
        uint256 _assetId,
        address _zroPaymentAddress
    ) = abi.decode(
            _payload,
            (uint16, bytes32, bytes32, uint256, uint256, uint256, ac
        );
```

Here we see the unpacking. note that the second unpacked value is put in the `from` field. This is an address determined by the attacker and passed through the layerzero endpoints. The contract then calls `_retrieveFromYieldBox` to take out tokens from the Yieldbox. They are then sent cross chain back to the `from` address.

```
    _retrieveFromYieldBox(_assetId, _amount, _share, _from, address(
    _debitFrom(
        address(this),
        lzEndpoint.getChainId(),
        LzLib.addressToBytes32(address(this)),
        _amount
    );
    bytes memory lzSendBackPayload = _encodeSendPayload(
```

```
            from,
            _ld2sd(_amount)
        );
        _lzSend(
            _srcChainId,
            lzSendBackPayload,
            payable(this),
            _zroPaymentAddress,
            "",
            address(this).balance
        );
```

Thus it is evident from this call that the YieldBox contract being called has no idea that the original sender was the attacker. Instead, for the YieldBox contract, the `msg.sender` is the current TOFT contract. If users want to use the cross chain operations, they have to give allowance to the TOFT address. Thus we can assume that the victim has already given allowance to this address. Thus the YieldBox thinks the `msg.sender` is the TOFT contract, who is allowed, and thus executes the operations.

Thus we have demonstrated that the attacker is able to call a function on the victim's YieldBox position **without** being given any allowance by setting the victim's address in the `from` field. Thus this is a high severity issue since the victim's tokens are withdrawn and send to a different chain without their consent.

Proof of Concept

Two lines from the test in test/TapiocaOFT.test.ts is changed to show this issue. Below is the full test for reference. The changed bits are marked with arrows.

▶ Details

The only relevant change is that the function `retrieveFromStrategy` is called from another address. The test passes, showing that an attacker, in this case `randomUser` can influence the operations of the victim, the `signer`.

Recommended Mitigation Steps

Add an allowance check for the `msg.sender` in the `strategyWithdraw` function.

0xRektora (Tapioca) confirmed, but disagreed with severity and commented:

> Should be medium. Although annoying, attacker can't steal the user's asset, and will have to pay gas without profit for both chains in order to do this trick. Should be grouped in #1037.

> Same answer as https://github.com/code-423n4/2023-07-tapioca-findings/issues/1009.

[H-35] `BaseTOFT.sol : removeCollateral` can be used to manipulate other user's positions and steal tokens due to absent approval check

Submitted by carrotsmuggler, also found by KIntern_NA

The function `removeCollateral` is used to trigger a removal of collateral on a different chain. The function takes the parameter `from`, which is the account whose collateral will be sold. It also takes the parameter `to` where these collateral tokens will be transferred to.

The main issue is that anyone can call this function with any address passed to the `from` parameter. There is no allowance check on the chain, allowing this operation. Let's walk through the steps to see how this is executed. Lets assume both `from` and `to` are the victim's address for reasons explained at the end.

`BaseTOFT.sol : removeCollateral` is called by the attacker with a `from` and `to` address of the victim. This function calls the `removeCollateral` function in the market module.

```
_executeModule(
    Module.Market,
    abi.encodeWithSelector(
        BaseTOFTMarketModule.removeCollateral.selector,
        from,
        to,
        lzDstChainId,
        zroPaymentAddress,
        withdrawParams,
        removeParams,
```

```
            approvals,
            adapterParams
        ),
        false
    );
```

`BaseTOFTMarketModule.sol : removeCollateral` is called. This function packs some data and sends it forward to the lz endpoint. Point to note, is that no approval check is done for the `msg.sender` of this whole setup yet.

```
    bytes memory lzPayload = abi.encode(
        PT_MARKET_REMOVE_COLLATERAL,
        from,
        to,
        toAddress,
        removeParams,
        withdrawParams,
        approvals
    );
    // lzsend(...)
```

After the message is sent, the **lzendpoint** on the receiving chain will call the TOFT contract again. Now, the `msg.sender` is **not** the attacker, but is instead the lzendpoint! The endpoint call gets delegated to the `remove` function in the Market module.

```
     (
        ,
        ,
        address to,
        ,
        ITapiocaOFT.IRemoveParams memory removeParams,
        ICommonData.IWithdrawParams memory withdrawParams,
        ICommonData.IApproval[] memory approvals
    ) = abi.decode(
            _payload,
            (
                uint16,
                address,
                address,
```

```
                bytes32,
                ITapiocaOFT.IRemoveParams,
                ICommonData.IWithdrawParams,
                ICommonData.IApproval[]
            )
        );
```

Here we see the unpacking. note that the third unpacked value is put in the `to` field. This is an address determined by the attacker and passed through the layerzero endpoints. The contract then calls a market contract's `removeCollateral` function.

```
    IMarket(removeParams.market).removeCollateral(
        to,
        to,
        removeParams.share
    );
```

Thus it is evident from this call that the Market contract being called has no idea that the original sender was the attacker. Instead, for the Market contract, the `msg.sender` is the current TOFT contract. If users want to use the cross chain operations, they have to give allowance to the TOFT contract address. Thus we can assume that the victim has already given allowance to this address. Thus the market thinks the `msg.sender` is the TOFT contract, who is allowed, and thus executes the operations.

Thus we have demonstrated that the attacker is able to call a function on the victim's market position **without** being given any allowance by setting the victim's address in the `to` field. While it is a suspected bug that the `removeCollateral` removes collateral from the `to` field's account and not the `from` field, since both these parameters are determined by the attacker, the bug exists either way. Thus this is a high severity issue since the victim's collateral is withdrawn, dropping their health factor.

Proof of Concept

A POC isnt provided since the test suite does not have a test for the `removeCollateral` function. However the function `retrieveFromStrategy`

suffers from the same issue and has been addressed in a different report. The test for that function can be used to demonstrate this issue.

Two lines from the test in test/TapiocaOFT.test.ts is changed to show this issue. Below is the full test for reference. The changed bits are marked with arrows.

▶ Details

The only relevant change is that the function `retrieveFromStrategy` is called from another address. The test passes, showing that an attacker, in tthis case `randomUser` can influence the operations of the victim, the `signer`.

Recommended Mitigation Steps

Add an allowance check for the `msg.sender` in the `removeCollateral` function.

0xRektora (Tapioca) confirmed

LSDan (Judge) commented:

> This attack can be summed up as "approvals are not checked when operating cross-chain." There are several instances of this bug with varying levels of severity all reported by warden carrotsmuggler. Because they all use the same attack vector and all perform undesired/unrequested acts on behalf of other users, I have grouped them and rated this issue as high risk.

[H-36] `twTAP.sol` : Reward tokens stored in index 0 can be stolen

Submitted by **carrotsmuggler**, *also found by* **KIntern_NA**

The function `claimAndSendRewards` can be called to collect rewards accrued by the twTAP position. This function can only be called by the `TapOFT.sol` contract during a crosschain operation. Thus a user on chain A can call `claimRewards` and on chain B, the function `_claimRewards` will be called and a bunch of parameters will be passed in that message.

```
    (
    ,
    ,
    address to,
    uint256 tokenID,
    IERC20[] memory rewardTokens,
    IRewardClaimSendFromParams[] memory rewardClaimSendParams
    ) = abi.decode(
        _payload,
        (
            uint16,
            address,
            address,
            uint256,
            IERC20[],
            IRewardClaimSendFromParams[]
        )
    );
```

All these parameters passed here comes from the original lz payload sent by the user from chain A. Of note is the array `rewardTokens` which is a user inputted value. This function then calls the twtap.sol contract as shown below.

```
    try twTap.claimAndSendRewards(tokenID, rewardTokens) {
```

In the twTAP contract, the function `claimAndSendRewards` eventually calls `_claimRewardsOn` , the functionality of which is shown below.

```
    function _claimRewardsOn(
        uint256 _tokenId,
        address _to,
        IERC20[] memory _rewardTokens
    ) internal {
        uint256[] memory amounts = claimable(_tokenId);
        unchecked {
            uint256 len = _rewardTokens.length;
            for (uint256 i = 0; i < len; ) {
                uint256 claimableIndex = rewardTokenIndex[_rewardTok
                uint256 amount = amounts[i];
```

```
                if (amount > 0) {
                    // Math is safe: `amount` calculated safely in `
                    claimed[_tokenId][claimableIndex] += amount;
                    rewardTokens[claimableIndex].safeTransfer(_to, a
                }
                ++i;
            }
        }
    }
```

Here we want to investigate a case where a user sends some random address in the array `rewardTokens`. We already showed that this value is set by the user, and the above quoted snippet receives the same value in the `_rewardTokens` variable.

In the for loop, the indexes are found. But if `rewardTokens[i]` is a garbage address, the mapping `rewardTokenIndex` will return the default value of 0 which will be stored in `claimableIndex`. The array `amounts` stores the amounts of the different tokens that can be claimed by the user. But since `claimableIndex` is now 0, the `safeTransfer` function in the end is always called on the token `rewardTokens[0]`. Thus a user can withdraw the rewardtoken in index 0 multiple times and in amounts based on the values stored in the `amounts` array.

Thus we have shown that a user can steal an unfair amount of tokens stored in the 0 index of the `rewardTokens` array variable in the twTAP.sol contract. This will mess up the reward distribution for all users, and can lead to an insolvent contract. Thus this is deemed a high severity issue.

Proof of Concept

The attack can be done in the following steps:

1. Attacker calls `claimRewards` on chain A. The attacker is assumed to have a valid position on chain B with pending rewards.
2. The attacker passes an array of garbage addresses in the `rewardTokens` parameter.
3. The contract sends forth the message to the destination chain, where the twTAP contract is called to collect the rewards.

4. As shown above, the reward token stored in index 0 is sent multiple times to the caller, which is the `TapOFT` contract.
5. In the TapOFT contract, the contract then sends all the collected rewards in the contract cross chain back to the attacker. Thus the tokens in index 0 were claimed and collected.

Thus the attacker can claim an unfair number of tokens present in the 0th index. If this token is more valuable than the other rewward tokens, the attacker can profit from this exploit.

Recommended Mitigation Steps

Mitigation can be done by not using the 0th index. The zero index of the `rewardTokens` array in `twTAP.sol`, if left empty, will point to the zero address, and if an unknown address is encountered, the contract will try to claim the tokens from the zero address which will revert.

This can be enforced by using the statement `rewardTokens.push(address(0))` in the constructor. However changes will need to be made on other operations in the contract which loops over this array to skip operations on this zero address now present in the array.

0xRektora (Tapioca) confirmed via duplicate issue 1093

[H-37] Liquidation transactions can potentially fail for all markets

Submitted by **zzzitron**, *also found by* **GalloDaSballo**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L315-L316

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L340-L344

As an example, when calling `BigBang.liquidate()` the tx potentially fails, because the subsequent call to `Market.updateExchangeRate` (BigBang.sol line 316) can face a revert condition on line 344 in Market.sol.

In `Market.updateExchangeRate()` a revert is triggered if `rate` is not bigger than 0 - see line 344 in Market.sol.

Liquidations should never fail and instead use the old exchange rate - see BigBang.sol line 315 comment:

> // Oracle can fail but we still need to allow liquidations

But the liquidation transaction can potentially fail when trying to fetch the new exchange rate via `Market.updateExchangeRate()` as shown above.

This issue applies for all markets (Singularity, BigBang) since the revert during the liquidation happens in the Market.sol contract from which all markets inherit.

Because of this issue user's collateral values may fall below their debt values, without being able to liquidate them, pushing the protocol into insolvency.

This is classified as high risk because liquidation is an essential functionality of the protocol.

Proof of Concept

Here is a POC that shows that a liquidation tx reverts according to the issue described above:

https://gist.github.com/zzzitron/90206267434a90990ff2ee12e7deebb0

Recommended Mitigation Steps

Instead of reverting on line 344 in Market.sol when fetching the exchange rate, consider to return the old rate instead, so that the liquidation can be executed successfully.

0xRektora (Tapioca) confirmed

[H-38] Magnetar contract has no approval checking

Submitted by carrotsmuggler, also found by 0xStalin

The `Magnetar.sol` contract has a lot of useful helper function to carry out operations on user market positions. If a user wishes to use the helper functions, they have to first give approval to the Magnetar contract to manipulate their positions. As an example, for the big bang markets, this is done by calling the `updateOperator` function.

```
    function updateOperator(address operator, bool status) external
        operators[msg.sender][operator] = status;
    }
```

Since this is a helper function, we can expect users to give this approval in order to use these functions. However the issue is that any attacker can use these approvals to manipulate and drain positions of other users.

As an example, let us look at the `withdrawToChain` function. Lets assume an attacker is calling this function, and the victim's address is passed in the `from` field. Assume the victim has given all approvals to the Magnetar contracts. The function delegates this to the `withdrawToChain` in the Market module.

In `withdrawToChain` function, there are no checks on the `msg.sender` address. The function interacts with yieldbox and does a crosschain send to the `erceiver` address passed by the attacker.

```
    if (dstChainId == 0) {
        yieldBox.withdraw(
            assetId,
            from,
            LzLib.bytes32ToAddress(receiver),
            amount,
            share
        );
        return;
    }
```

```
    yieldBox.withdraw(assetId, from, address(this), amount, 0);

    ISendFrom(address(asset)).sendFrom{value: gas}(
        address(this),
        dstChainId,
        receiver,
        amount,
        callParams
    );
```

This sends the tokens to the `receiver` address either in the same chain or cross-chain. This lets any user steal tokens from any other user, exploiting the approval given to the magnetar address.

While this report only discusses the issue with this one function, the same issue is present for every function in the magnetar contract. This allows attackers to manipulate bigbang markets and singularity markets as well. Thus this is a high severity issue.

Proof of Concept

A POC is developed by editing the test present in magnetar.test.ts. Only a single change is made to the test. The last `withdrawToChain` call is done from the `eoa1` address instead of the deployer address.

▶ Details

This test passes, showing that the `eoa1` address is able to withdraw tokens belonging to the deployer.

Tools Used

Hardhat

Recommended Mitigation Steps

Add approval checks to all functions in the Magnetar contract.

[0xRektora (Tapioca) confirmed](#)

[H-39] `AaveStrategy.sol` : Changing swapper breaks the contract

Submitted by carrotsmuggler, *also found by* Oxfuje, Vagner, kaden, rvierdiiev, *and* ladboy233

The contract `AaveStrategy.sol` manages wETH tokens and deposits them to the aave lending pool, and collects rewards. These rewards are then swapped into WETH again to compound on the WETH being managed by the contract. this is done in the `compound` function.

```
uint256 calcAmount = swapper.getOutputAmount(swapData, "");
uint256 minAmount = calcAmount - (calcAmount * 50) / 10_000; //(
swapper.swap(swapData, minAmount, address(this), "");
```

To carry out these operations, the swapper contract needs to be given approval to use the tokens being stored in the strategy contract. This is required since the swapper contract calls transferFrom on the tokens to pull it out of the strategy contract. This allowance is set in the constructor.

```
rewardToken.approve(_multiSwapper, type(uint256).max);
```

The issue arises when the swapper contract is changed. The change is done via the `setMultiSwapper` function. This function however does not give approval to the new swapper contract. Thus if the swapper is upgraded/changed, the approval is not transferred to the new swapper contract, which makes the swappers dysfunctional.

Since the swapper is critical to the system, and `compound` is called before withdrawals, a broken swapper will break the withdraw functionality of the contract. Thus this is classified as a high severity issue.

Proof of Concept

The bug is due to the absence of `approve` calls in the `setMultiSwapper` function. This can be seen from the implementation of the function.

```
function setMultiSwapper(address _swapper) external onlyOwner {
        emit MultiSwapper(address(swapper), _swapper);
        swapper = ISwapper(_swapper);
    }
```

Recommended Mitigation Steps

In the `setMultiSwapper` function, remove approval from the old swapper and add approval to the new swapper. The same function has the proper implementation in the `ConvexTricryptoStrategy.sol` contract which can be used here as well.

```
function setMultiSwapper(address _swapper) external onlyOwner {
    emit MultiSwapper(address(swapper), _swapper);
    rewardToken.approve(address(swapper), 0);
    swapper = ISwapper(_swapper);
    rewardToken.approve(_swapper, type(uint256).max);
}
```

0xRektora (Tapioca) confirmed via duplicate issue 222

[H-40] `BalancerStrategy.sol : _withdraw` withdraws insufficient tokens

Submitted by carrotsmuggler, also found by kaden, n1punp, and chaduke

The funciton `_withdraw` in the balancer strategy contract is called during withdraw operations to withdraw WETH from the balancer pool. The function calculats the amount to withdraw, and then calls `_vaultWithdraw` function.

```
if (amount > queued) {
            uint256 pricePerShare = pool.getRate();
            uint256 decimals = IStrictERC20(address(pool)).decim
            uint256 toWithdraw = (((amount - queued) * (10 ** de
                pricePerShare);

            _vaultWithdraw(toWithdraw);
```

```
        }
```

The function `_vaultWithdraw` submits an exit request with the following userData.

```
    exitRequest.userData = abi.encode(
            2,
            exitRequest.minAmountsOut,
            pool.balanceOf(address(this))
        );
```

A value of 2 here corresponds to specifying the exact number of tokens coming out of the contract. Thus the function `_vaultWithdraw` will withdraw the exact number of tokens passed to it in its parameter.

The issue however is that the function `_vaultWithdraw` is not called with the amount of tokens needed to be withdrawn, it is called by the amount scaled down by `pricePerShare`. Thus if the actual withdrawn amount is less the amounts the user actually wanted. This causes a revert in the next step.

```
    require(
            amount <= wrappedNative.balanceOf(address(this)),
            "BalancerStrategy: not enough"
        );
```

Since an insuffucient amount of tokens are withdrawn, this step will revert if there arent enough spare tokens in the contract. Since the contract incorrectly scales doen the withdraw amount and causes a revert, this is classified as a high severity issue.

Proof of Concept

The following exercise shows that passing the same `exitRequest` data to the balancerPool actually extracts the exact number of tokens as specified in `minamountsOut`.

A position is created on optimism's weth-reth pool. The `userData` is generated using the following code.

```solidity
function temp() external pure returns(bytes memory){
    uint256[] memory amts = new uint256[](2);
    amts[0] = 500;
    amts[1] = 0;
    uint256 max = 20170422329691;
    return(abi.encode(2,amts,max));
}
```

Min amount out of WETH is set to 500 wei. The `exitRequest` is then constructed as follows with the userData from above.

```
{
    "assets": [
        "0x4200000000000000000000000000000000000006",
        "0x9bcef72be871e61ed4fbbc7630889bee758eb81d"
    ],
    "minAmountsOut": ["500", "0"],
    "toInternalBalance": false,
    "userData": "0x000000000000000000000000000000000000000000000000(
}
```

This is an exit request of type 2, which specifies the exact amount of tokens to be withdrawn. This transaction was then run on tenderly to check how many tokens are withdrawn. From the screenshot here from tenderly we can see only 500 wei of WETH is withdrawn.

This proves that the `_vaultWithdraw` function withdraws the exact amount of tokens passed to it as a parameter. Since the passed parameter is scaled down by `pricePerShare`, this leads to an insufficient amount withdrawn, and eventually a revert.

Tools Used

Tenderly

Recommended Mitigation Steps

Pass the amount to be withdrawn without scaling it down by `pricePerShare`.

[0xRektora (Tapioca) confirmed via duplicate issue 51]

[H-41] Rewards compounded in AaveStrategy are unredeemable

Submitted by **Ack**, *also found by* **kaden** *and* **rvierdiiev**

The AaveStrategy contract is designed to:

1. Receive depositor's ERC20 tokens from yieldBox
2. Deposit those tokens into an AAVE lending pool
3. Allow anyone to call `compound()`, which: a. Claims AAVE rewards from the `incentivesController` b. Claims staking rewards from the `stakingRewardToken` (stkAAVE) c. Redeeming staking rewards is only possible within a certain cooldown window that is set by AAVE governance. The function resets the cooldown if either 12 days have passed since the cooldown was last initiated, or if the strategy has a stakedRewardToken balance d. Swaps any received `rewardToken` ($AAVE) for `wrappedNative` e. Deposits the `wrappedNative` received in the swap into the lending pool

There are several issues with this flow, but this vulnerability report is specific to redeeming staked rewards. The incentives controller specified in the `mainnet.env` file is at address 0xd784927Ff2f95ba542BfC824c8a8a98F3495f6b5 (proxy). Its `claimRewards` function stakes tokens directly:

```
function _claimRewards(
  ...
) internal returns (uint256) {
  ...
  uint256 accruedRewards = _claimRewards(user, userState);
  ...
  STAKE_TOKEN.stake(to, amountToClaim); //@audit claimed rewar
  ...
```

```
        }
```

The only way to retrieve tokens once staked is via a call to `stakingRewardToken#redeem()`, which is not present in the AaveStrategy contract. As a result, any rewards accumulated via the incentiveController would not be claimable.

Impact

High - Loss of funds

Proof of concept

This is unfortunately difficult to PoC as the AAVE incentive/staking rewards do not accumulate properly in the fork tests, and the mocks do not exhibit the same behavior.

Recommended Mitigation Steps

Include a call to redeem in `compound()`.

0xRektora (Tapioca) confirmed via duplicate issue 243

[H-42] Attacker can steal victim's oTAP position contents via `MagnetarMarketModule#_exitPositionAndRemoveCollateral()`

Submitted by **Ack**, *also found by* **zzebra83**

NOTE: This vulnerability relies on the team implementing an `onERC721Received()` function in Magnetar. As is currently written, attempts to exit oTAP positions via Magnetar will always revert as Magnetar cannot receive ERC721s, despite this being the clear intention of the function. This code path was not covered in the tests. Once implemented, however, an attack vector to steal twAML-locked oTAP positions opens.

Also, this is a similar but distinct attack vector from #933.

`MagnetarMarketModule#_exitPositionAndRemoveCollateral()` is a complex function used to perform any combination of: exiting an oTAP position, unlocking a locked tOLP position, removing assets and collateral from Singularity/bigBang, and repaying loans. The function achieves this by employing separate “if” clauses for each task that the caller would like to perform. These clauses are entered based on flags the caller provides in the argument struct `removeAndRepayData`.

Along with the set of operations to perform, the caller also provides:

- address `user` to operate on
- address externalData.bigBang
- address externalData.singularity
- address yieldbox (obtained by calling the user-provided `ISingularity(externalData.singularity).yieldBox()`)

As the caller has full control of all of these parameters, he can execute attacks to steal assets that have been approved to Magnetar.

Impact

High - Theft of funds

Proof of concept

Unfortunately the Magnetar tests do not cover the case where we wish to exit our oTAP position, and the periphery testing infrastructure does not include helper functions for the oTAP workflows (at least that I was able to find). This makes a coded PoC difficult and time consuming. Please consider the following walkthrough and reach out if a coded example is necessary.

In this case, we’re going to assume that a victim wants to use this function as-designed to exit his twAML-locked oTAP position. In order to do so he needs to grant approval in for Magnetar to transfer his position.

Once approved, anyone can call `_exitPositionAndRemoveCollateral()` with his own set of target addresses (some valid Tapioca addresses, some attacker-owned contracts) and the approver as `user`.

The attack is as follows:

1. Victim approves Magnetar to control his oTAP positions
2. Attacker first calls `_exitPositionAndRemoveCollateral()`, with:
    - `removeAndRepayData.exitData.exit = true`
    - `removeAndRepayData.unlockData.unlock = true`
    - The `user` to steal from
    - the `tokenId` to exit + steal
    - removeAndRepayData.unlockData.target = an attacker-controlled contract that just passes when called with `.unlock()`
3. Magnetar transfers the oTAP position to itself and exits the position. It retains the yieldbox shares at the end of the call
4. Attacker calls `_exitPositionAndRemoveCollateral()`, with:
    - removeAndRepayData.exitData.exit = false
    - removeAndRepayData.unlockData.unlock = true
    - The `user` is the attacker's address for receiving the unlocked shares
    - the tokenId to exit + steal
    - removeAndRepayData.unlockData.target = the real tOLP contract

(The attacker could have alternately used a similar bigBang attacker contract approach for removing the yieldbox shares in step 4 as in #933)

▶ Details

Recommended Mitigation Steps

- Do not allow arbitrary address input in these complex, multi-use functions.
- Consider breaking this into multiple standalone functions
- Require user == msg.sender

> 0xRektora (Tapioca) confirmed

[H-43] Accounted balance of GlpStrategy does not match withdrawable balance, allowing for attackers to steal unclaimed rewards

Submitted by kaden, *also found by kaden (*1, 2*) and* cergyk

Attackers can steal unclaimed rewards due to insufficient accounting.

Proof of Concept

Pricing of shares for Yieldbox strategies is dependent upon the total underlying balance of the strategy. We can see below how we mint an amount of shares according to this underlying amount.

```
// depositAsset()
uint256 totalAmount = _tokenBalanceOf(asset);
if (share == 0) {
        // value of the share may be lower than the amount due t
        share = amount._toShares(totalSupply[assetId], totalAmou
} else {
        // amount may be lower than the value of share due to ro
        amount = share._toAmount(totalSupply[assetId], totalAmou
}

_mint(to, assetId, share);
```

The total underlying balance of the strategy is obtained via `asset.strategy.currentBalance`.

```
function _tokenBalanceOf(Asset storage asset) internal view retu
        return asset.strategy.currentBalance();
}
```

`GlpStrategy._currentBalance` does not properly track all unclaimed rewards.

```
function _currentBalance() internal view override returns (uint2
        // This _should_ included both free and "reserved" GLP:
        amount = IERC20(contractAddress).balanceOf(address(this)
```

```
    }
```

As a result, attackers can:

- Deposit a high amount when there are unclaimed rewards
    - Receiving a higher amount of shares than they would if accounting included unclaimed rewards
    - Harvests unclaimed rewards, increasing `_currentBalance`, only after they received shares
- Withdraw all shares
    - Now that the balance is updated to include previously unclaimed rewards, the attacker profits their relative share of the unclaimed rewards
        - The more the attacker deposits relative to the strategy balance, the greater proportion of interest they receive

Recommended Mitigation Steps

It's recommended that `_currentBalance` include some logic to retrieve the amount and value of unclaimed rewards to be included in it's return value.

cryptolyndon (Tapioca confirmed)

[H-44] `BigBang::repay` and `Singularity::repay` spend more than allowed amount

Submitted by **zzzitron**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L263-L268

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/

bigBang/BigBang.sol#L721-L732

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/a4793e75a79060f8332927f97c6451362ae30201/contracts/markets/singularity/SGLLendingCommon.sol#L83-L95

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/a4793e75a79060f8332927f97c6451362ae30201/contracts/markets/singularity/SGLBorrow.sol#L45-L52

When an user allows certain amount to a spender, the spender can spend more than the allowance.

Note that this is a different issue from the misuse of `allowedBorrow` for the share amount (i.e. issue "`BigBang::repay` uses `allowedBorrow` with the asset amount, whereas other functions use it with share of collateral"), as the fix in the other issue will not mitigate this issue. This issue is the misuse of `part` and `elastic`, whereas the other issue is the misuse of the `share` and `asset`.

Proof of Concept

The spec in the `MarketERC20::approve` function specifies that the approved amount is the maximum amount that the spender can draw.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/364dead3a42b06a34c802eee951cea1a654d438e/contracts/markets/MarketERC20.sol#L189-L200

```
/// @notice Approves `amount` from sender to be spend by `sp
/// @param spender Address of the party that can draw from m
/// @param amount The maximum collective amount that `spende
/// @return (bool) Returns True if approved.
function approve(
    address spender,
    uint256 amount
) public override returns (bool) {
```

However, the spender can draw more than the allowance if the `totalBorrow.base` is more thant `totalBorrow.elastic`, which is likely condition.

The proof of concept below demonstrates that more asset was pulled than allowed. It is only a part of proof of concept; to see the full proof of concept see https://gist.github.com/zzzitron/8dd809c0ea39dc0ea727534c3ba804f9 To use it, put it in the `test/bigBang.test.ts` in the tapiocabar-audit repo

The eoa1 allows deployer 1e18. After the `timeTravel`, the elastic of `totalBorrow` is more than the `base`. Under the condition, the deployer uses the allowance with the `BigBang::repay` function. As the result, more asset than allowance was pulled from eoa1.

```
it('should not allow repay more PoCRepayMoreThanAllowed'

    ////////
    // setup steps are omitted
    // the full proof of concept is
    // https://gist.github.com/zzzitron/8dd809c0ea39dc0e
    ////////

    // eoa1 allows deployer (it should be `approve`, if
    const allowedPart = ethers.BigNumber.from((1e18).to
    await wethBigBangMarket.connect(eoa1).approveBorrow

    //repay from eoa1
    // check more than the allowed amount is pulled from

    timeTravel(10 * 86400);

    // repay from eoa1 the allowed amount
    // balance before repay of eoa in the yieldBox for t
    const usdoAssetId = await wethBigBangMarket.assetId
    const eoa1ShareBalanceBefore = await yieldBox.balanc
    const eoa1AmountBefore = await yieldBox.toAmount(usd
    await wethBigBangMarket.repay(
        eoa1.address,
        deployer.address,
        false,
        allowedPart,
    );
    const eoa1ShareBalanceAfter = await yieldBox.balance
```

```
            const eoa1AmountAfter = await yieldBox.toAmount(usdc
            console.log(eoa1AmountBefore.sub(eoa1AmountAfter).to
            expect(eoa1AmountBefore.sub(eoa1AmountAfter).gt(allo
        });
```

The result of the poc is below, which shows that `1000136987569097987` is pulled from the eoa1, which is more than the allowance (i.e. 1e18).

```
  BigBang test
    poc
1000136987569097987
      ✓ should not allow repay more PoCRepayMoreThanAllowed (11
```

The same issue is also in the `Singularity`. In the same manner shown above, the spender will pull more than allowed when the `totalBorrow.elastic` is bigger than the `totalBorrow.base`.

Details of the bug

The function `BigBang::repay` uses `part` to check for the allowance.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L263-L268

However, the `BigBang::_repay` draws actually the corresponding `elastic` of the `part` from the `from` address.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L721-L732

```
    function _repay(
        address from,
        address to,
        uint256 part
    ) internal returns (uint256 amount) {
        (totalBorrow, amount) = totalBorrow.sub(part, true);
```

```
            userBorrowPart[to] -= part;

            uint256 toWithdraw = (amount - part); //acrrued
            uint256 toBurn = amount - toWithdraw;
            yieldBox.withdraw(assetId, from, address(this), amount,
```

The similar lines of code is also in the `Singularity`. The `Singularity::repay` will delegate call on the `SGLBorrow::repay`, which has the modifier of `allowedBorrow(from, part)`:

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/a4793e75a79060f8332927f97c6451362ae30201/contracts/markets/singularity/SGLBorrow.sol#L45-L51

```
    // SGLBorrow

        function repay(
            address from,
            address to,
            bool skim,
            uint256 part
        ) public notPaused allowedBorrow(from, part) returns (uint25
            updateExchangeRate();
```

Then, `amount` is calculated from the `part`, and the `amount` is pulled from the `from` address in the below code snippet.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/a4793e75a79060f8332927f97c6451362ae30201/contracts/markets/singularity/SGLLendingCommon.sol#L83-L95

```
        function _repay(
            address from,
            address to,
            bool skim,
            uint256 part
        ) internal returns (uint256 amount) {
            (totalBorrow, amount) = totalBorrow.sub(part, true);
```

```
            userBorrowPart[to] -= part;

            uint256 share = yieldBox.toShare(assetId, amount, true);
            uint128 totalShare = totalAsset.elastic;
            _addTokens(from, to, assetId, share, uint256(totalShare)
            totalAsset.elastic = totalShare + uint128(share);
            emit LogRepay(skim ? address(yieldBox) : from, to, amour
        }
```

The `amount` is likely to be bigger than the `part`, since the calculation is based on the `totalBorrow`'s ratio between `elastic` and `base`. Then the `amount` is used to withdraw from `from` address, meaning that more than the allowance is withdrawn.

The discrepancy between the allowance and actually spendable amount is going to grow in time, as the `totalBorrow`'s elastic will outgrow the base in time.

Tools Used

Hardhat

Recommended Mitigation Steps

Instead of using the `part` to check the allowance, calculate the actual amount to be pulled and use the amount to check the allowance.

0xRektora (Tapioca) confirmed

[H-45] `SGLLiquidation::_computeAssetAmountToSolvency`, `Market::_isSolvent` and `Market::_computeMaxBorrowableAmount` may overestimate the collateral, resulting in false solvency

Submitted by **zzzitron**

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L415-L421

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L385-L399

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L781-L785

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/singularity/SGLLiquidation.sol#L80-L87

An user can borrow via `BigBang::borrow` when there is no collateral amount from the user's share. The `BigBang` will falsely consider the position as solvent, when it is not, resulting in a loss.

A similar issue presents in the Singularity as `SGLLiquidation::_computeAssetAmountToSolvency` will overestimate the collateral, therefore liquidate less than it should.

Proof of Concept

The following proof of concept demonstrates that au user could borrow some assets, even though the collateral share will not give any amount of collateral.

Put the full PoC in the following gist into `test/bigBang.test.ts` in tapiocabar-audit.

https://gist.github.com/zzzitron/14482ea3ab35b08421e7751bac0c2e3f

▶ Details

The result of the test is:

```
  BigBang test
    poc
userCollateralShare:  99999999
userCollateralShareToAmount:  0
collateralPart in asset times exchangerRate 749999992500000000
exchangeRate: 1000000000000000
```

```
can borrow this much: 749
748
      ✓ considers me solvent even when I have enough share for
```

In the scenario above, The deployer is adding share of collateral to the bigbang using `BigBang::addCollateral`. The added amount in the below example is (1e8 - 1), which is too small to get any collateral from the YieldBox, as the `yieldBox.toAmount` is zero.

However, due to the calculation error in the `Market::_isSolvent`, the deployer could borrow 748 of asset. Upon withdrawing the yieldBox will give zero amount of collateral, but the BigBang let the user borrow non zero amount of asset.

Similarly one can show that `Singularity` will liquidate less than it should, due to similar calculation error.

details of the bug

The problem stems from the calculation error, where multiplies the user's collateral share with `EXCHANGE_RATE_PRECISION` and `collateralizationRate` before calling `yieldBox.toAmount`. It will give inflated amount, resulting in false solvency.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L415-L421

```
            return
                yieldBox.toAmount(
                    collateralId,
                    collateralShare *
                        (EXCHANGE_RATE_PRECISION / FEE_PRECISION) *
                        collateralizationRate,
                    false
                ) >=
```

The same calculation happens in the `Market::_computeMaxBorrowableAmount`

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/Market.sol#L385-L399

In the BigBang's liquidating logic (e.i. in the `BigBang::_updateBorrowAndCollateralShare`), the conversion from the share of collateral to the asset is calculated correctly:

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/bigBang/BigBang.sol#L781-L785

```
            uint256 collateralPartInAsset = (yieldBox.toAmount(
                collateralId,
                userCollateralShare[user],
                false
            ) * EXCHANGE_RATE_PRECISION) / _exchangeRate;
```

However, the position in question will not get to this logic, even if `BigBang::liquidate` is called on the position, since the `_isSolvent` will falsely consider the position as solvent.

Similarly the `Singularity` will overestimate the collateral in the same manner.

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/singularity/SGLLiquidation.sol#L70-L90

```
    function _computeAssetAmountToSolvency(
        address user,
        uint256 _exchangeRate
    ) private view returns (uint256) {
        // accrue must have already been called!
        uint256 borrowPart = userBorrowPart[user];
        if (borrowPart == 0) return 0;
        uint256 collateralShare = userCollateralShare[user];

        Rebase memory _totalBorrow = totalBorrow;
```

```
uint256 collateralAmountInAsset = yieldBox.toAmount(
    collateralId,
    (collateralShare *
        (EXCHANGE_RATE_PRECISION / FEE_PRECISION) *
        lqCollateralizationRate),
    false
) / _exchangeRate;
// Obviously it's not `borrowPart` anymore but `borrowAm
borrowPart = (borrowPart * _totalBorrow.elastic) / _tota
```

Recommended Mitigation Steps

The `Market::_isSolvent` and `Market::_computeMaxBorrowableAmount` should evaluate the value of collateral like `BigBang::_updateBorrowAndCollateralShare` function, (e.i. calculate the exchangeRate and collateralizationRate after converting the share to asset).

0xRektora (Tapioca) confirmed

[H-46] TOFT leverageDown always fails if TOFT is a wrapper for native tokens

Submitted by **windhustler**

Pathway for `sendForLeverage` -> `leverageDown` always fails if the `TapiocaOFT` or `mTapiocaOFT` holds the native token as underlying, i.e. `erc20 == address(0)`. This results in loss of gas, airdropped amount, and burned TOFT on the sending side for the user. The failed message if retried will always fail and result in permanent loss for the user.

Proof of Concept

TapiocaOFT/mTapiocaOFT is deployed with **erc20 being address(0)** in case if it holds the native token as an underlying token. However, it still allows anyone to execute the `sendForLeverage` which always results in reverts when receiving the message.

The revert happens at `IERC20(erc20).approve(externalData.swapper, amount);` since `address(0)` doesn't have an `approve` function.

The message if retried will just keep on reverting because of the same reason due to the way the `failedMessages` are stored, e.g. you can just retry the same exact payload. This way anyone invoking this function will lose his TOFT tokens forever.

Recommended Mitigation Steps

Disable `sendForLeverage` function if the `TapiocaOFT` or `mTapiocaOFT` holds the native token as underlying, e.g. revert on the sending side.

0xRektora (Tapioca) confirmed

[H-47] User's assets can be stolen when removing them from the Singularity market through the Magnetar contract

Submitted by **0xStalin**, *also found by* **Ack**

An Attacker can remove user's assets from Singularity Markets and steal them to an account of his own by abusing a vulnerability present in the Magnetar contract

Proof of Concept

- The `Magnetar::exitPositionAndRemoveCollateral()` can be used to exit from tOB, unlock from tOLP, remove assets from Singularity markets, repay on BigBang markets, remove collateral from BigBang markets and withdraw, each of these steps are optional.
- When users wants to execute any operation through the Magnetar contract, the Magnetar contracts requires to have the user's approvals/permissions, that means, when the Magnetar contract executes something on behalf of the user, the Magnetar contract have already been granted permission/allowance on the called contract on the user's behalf.
- **When using the Magnetar contract to remove user assets from the Singularity market and use those assets to repay in a BigBang contract**, the Magnetar contract will receive the removed assets from the Singularity Market, grant ALL

allowance to the BigBang contract in the YieldBox, and finally will call the BigBang.repay().

- The problem is that none of the two markets are checked to ensure that they are valid and supported contracts by the Protocol.
- This attack requires that an attacker creates a FakeBigBang contract (see Step 2 of the Coded PoC mini section!), and passes the address of this Fake BigBang contract as the address of the BigBang where the repayment will be done.

    - When the execution is forwarded to the FakeBigBang contract, the Magnetar contract had already granted ALL allowance to this Fake contract in the YieldBox, which makes possible to do a `YieldBox.transfer()` from the Magnetar contract to an account owned by the attacker.

        - The transferred assets from the Magnetar contract are the assets of the user that were removed from the Singularity market and that they were supposed to be used to repay the user's debt on the BigBang contract

Coded PoC

- I coded a PoC using the `magnetar.test.ts` file as the base for this PoC.
- The first step is to add the `attacker` account in the `test.utils.ts` file

```
> git diff --no-index test.utils.ts testPoC.utils.ts

diff --git a/test.utils.ts b/testPoC.utils.ts
index 00fc388..83107e6 100755
--- a/test.utils.ts
+++ b/testPoC.utils.ts
@@ -1023,8 +1023,14 @@ export async function register(stagir
         ethers.provider,
     );

+    const attacker = new ethers.Wallet(
+        ethers.Wallet.createRandom().privateKey,
+        ethers.provider,
+    );
+
     if (!staging) {
         await setBalance(eoa1.address, 100000);
+        await setBalance(attacker.address, 100000);
```

```
        }

        // ------------------- Deploy WethUSDC mock oracle ----
    @@ -1314,6 +1320,7 @@ export async function register(staging

        if (!staging) {
            await setBalance(eoa1.address, 100000);
    +       await setBalance(attacker.address, 100000);
        }

        const initialSetup = {
    @@ -1341,6 +1348,7 @@ export async function register(staging
            _sglLeverageModule,
            magnetar,
            eoa1,
    +       attacker,
            multiSwapper,
            singularityFeeTo,
            liquidationQueue,
```

2. Now, let's create the `FakeBigBang` contract, make sure to create it under the `tapioca-periph-audit/contract/` folder

▶ Details

3. Create a new file to reproduce this PoC, magnetar*remove*assets*from*singularity_PoC.test.ts
4. Make sure to create this new test file under the `tapioca-periph-audit/test/` folder

▶ Details

4. After all the 3 previous steps have been completed, everything is ready to run the PoC.

> npx hardhat test magnetar*remove*assets*from*singularity_PoC.test.ts

```
  MagnetarV2
    repay
asset in YieldBox owned by Magnetar - BEFORE:  BigNumber { value
asset in YieldBox owned by User - BEFORE:  BigNumber { value: "1
asset in YieldBox owned by FakeBigBang - BEFORE:  BigNumber { va
```

```
==========================================================================

asset in YieldBox owned by Magnetar - AFTER:  BigNumber { value:
asset in YieldBox owned by User - AFTER:  BigNumber { value: "1(
asset in YieldBox owned by FakeBigBang - AFTER:  BigNumber { val
      ✓ should remove asset from Singularity and Attacker will
```

Recommended Mitigation Steps

- Use the Penrose contract to validate that the provided markets as parameters are real markets supported by the protocol (Both, BB & Singularity markets)
- Add the below checks on the `_exitPositionAndRemoveCollateral()` function

```
function _exitPositionAndRemoveCollateral(
    address user,
    ICommonData.ICommonExternalContracts calldata externalData,
    IUSDOBase.IRemoveAndRepay calldata removeAndRepayData
) private {
+   require(penrose.isMarketRegistered(externalData.bigBang), "E
+   require(penrose.isMarketRegistered(externalData.singularity)

    IMarket bigBang = IMarket(externalData.bigBang);
    ISingularity singularity = ISingularity(externalData.singula
    IYieldBoxBase yieldBox = IYieldBoxBase(singularity.yieldBox(

    ...
    ...
    ...
}
```

0xRektora (Tapioca) confirmed

[H-48] triggerSendFrom() will send all the ETH in the destination chain where sendFrom() is called to the refundAddress in the LzCallParams argument

Submitted by 0x73696d616f

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L99

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/BaseTOFT.sol#L551

https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/tOFT/modules/BaseTOFTOptionsModule.sol#L142

https://github.com/Tapioca-DAO/tapioca-sdk-audit/blob/90d1e8a16ebe278e86720bc9b69596f74320e749/src/contracts/token/oft/v2/BaseOFTV2.sol#L18

All the ETH in the destination chain where `sendFrom()` is called is sent to the `refundAddress` in the `LzCallParams`. Thus, for `TapiocaOFT`s which have ETH as the underlying asset `erc`, all the funds will be lost if the `refundAddress` is an address other than the `TapiocaOFT`.

Proof of Concept

`sendFrom()` uses the `msg.value` as native fees to LayerZero, being the excess sent refunded to the `refundAddress`. In `BaseTOFTOptionsModule`, `sendFromDestination()`, which is called when there was a `triggerSendFrom()` from a source chain which is delivered to the current chain, the value sent to the `sendFrom()` function is `address(this).balance`:

```
    function sendFromDestination(bytes memory _payload) public {
        ...
        ISendFrom(address(this)).sendFrom{value: address(this).balar
            from,
            lzDstChainId,
            LzLib.addressToBytes32(from),
            amount,
            callParams
        );
        ...
```

```
    }
```

This means that all the balance but the LayerZero message fee will be refunded to the `refundAddress` in the `callParams`, as can be seen in the `sendFrom()` function:

```
    function sendFrom(address _from, uint16 _dstChainId, bytes32
        _send(_from, _dstChainId, _toAddress, _amount, _callPara
    }
```

The following POC shows that a user that specifies the `refundAddress` as its address will receive all the ETH balance in the `TapiocaOFT` contract minus the LayerZero message fee.

▶ Details

Tools Used

Vscode, Foundry

Recommended Mitigation Steps

The value sent in the `sendFrom()` call in the `BaseTOFTOptionsModule` should be sent and forwarded from the `triggerSendFrom()` call in the source chain. This way, the user pays the fees from the source chain.

0xRektora (Tapioca) confirmed and commented:

> Good finding. Technically speaking the `sendFrom()` will fail if the call was made to the host chain, the one holding the Ether, since LZ have a limit to the amount of value you can send between chain, but nonetheless valid.

LSDan (Judge) decreased severity to Medium

0x73696d616f (Warden) commented:

> Hi everyone,

> This issue should be a valid high as there is no limit on the ETH transferred. The ETH is sent to the attacker (or normal user) on the refund of the LayerZero UltraLightNodeV2, here, in the source chain where `sendFrom()` is called, not in the destination chain. The only cross chain transaction required for this exploit is the `triggerSendFrom()`, which sends no ETH to the chain where `sendFrom()` is called. Thus, the ETH is not actually sent as a cross chain transaction, but sent directly as a refund in the source chain (see the test in the POC) where `sendFrom()` is called, not having any limit.

> Kindly request a review from the judge.

LSDan (Judge) increase severity to High and commented:

> Thank you for the clarification. You are correct. This is a valid high risk issue.

[H-49] User can give himself approval for all assets held by `MagnetarV2` contract

Submitted by **0xTheC0der**, *also found by* **Ack** *and* **dirk_y**

When calling **MagnetarV2._permit(...)** through invoking a permit (or permit all) action via **MagnetarV2.burst(...)**, one can also execute other calls than `ERC20.permit(...)` due to the following reasons / under the following constraints:

- The `target` address can be chosen freely, can be any contract, asset, token, NFT, etc.
- The function selector in `actionCalldata` is not checked, i.e. not required to be `ERC20.permit(...)`
- The first parameter in the encoded `actionCalldata` **must be equal to** `msg.sender`
- The length of the `actionCalldata` should match the length of an encoded call to `ERC20.permit(...)` to avoid issues on `abi.decode(...)`

Given this information, an attacker can easily craft calls to give him approval for any assets held by the `MagnetarV2` contract or directly invoke a transfer. There are

potentially other malicious calls that can be crafted and executed via the permit action, therefore the mentioned approve/transfer calls are only an example.

In order for this to cause loss of funds for the DAO, the `MagnetarV2` contract needs to hold (be the owner of) assets in the first place which seems likely since it is a main entry point and interacts with other important parts of the protocol like Singularity, BigBang, TapiocaOptionBroker and MagnetarMarketModule (trough `delegatecall` in some cases).

Proof of Concept

The following PoC is based on an existing test case and demonstrates that an attacker can give himself the approval of the `MagnetarV2` contract for an ERC20 token.

Just apply the *diff* below in `tapioca-periph-audit` and run the test case with `npx hardhat test test/magnetar.test.ts`:

<details>
<summary>Details</summary>
</details>

Tools Used

VS Code, Hardhat

Recommended Mitigation Steps

Require the function selector (first 4 bytes of `actionCalldata`) to match an `ERC*.permit(...)` call in MagnetarV2._permit(...).

0xRektora (Tapioca) confirmed

[H-50] CompoundStrategy attempts to transfer out a greater amount of ETH than will actually be withdrawn, leading to DoS

Submitted by kaden

Withdrawals will revert whenever there is not sufficient wrapped native tokens to cover loss from integer truncation.

Proof of Concept

`CompoundStrategy._withdraw` calculates the amount of cETH to redeem with `cToken.exchangeRateStored` based on a provided amount of ETH to receive from the withdrawal.

```
    function _withdraw(
            address to,
            uint256 amount
    ) internal override nonReentrant {
            uint256 available = _currentBalance();
            require(available >= amount, "CompoundStrategy: amount r

            uint256 queued = wrappedNative.balanceOf(address(this));
            if (amount > queued) {
                    uint256 pricePerShare = cToken.exchangeRateStore
                    uint256 toWithdraw = (((amount - queued) * (10 *
                            pricePerShare);

                    cToken.redeem(toWithdraw);
```

To understand the vulnerability, we look at the `CEther` contract which we are attempting to withdraw from.

```
    /**
     * @notice Sender redeems cTokens in exchange for the underlying
     * @dev Accrues interest whether or not the operation succeeds,
     * @param redeemTokens The number of cTokens to redeem into unde
     * @return uint 0=success, otherwise a failure (see ErrorReporte
     */
    function redeem(uint redeemTokens) external returns (uint) {
            return redeemInternal(redeemTokens);
    }
```

`redeem` returns the result from `redeemInternal`

```
    /**
     * @notice Sender redeems cTokens in exchange for the underlying
     * @dev Accrues interest whether or not the operation succeeds,
     * @param redeemTokens The number of cTokens to redeem into unde
     * @return uint 0=success, otherwise a failure (see ErrorReporte
     */
    function redeemInternal(uint redeemTokens) internal nonReentrant
            uint error = accrueInterest();
            if (error != uint(Error.NO_ERROR)) {
                    // accrueInterest emits logs on errors, but we s
                    return fail(Error(error), FailureInfo.REDEEM_ACC
            }
            // redeemFresh emits redeem-specific logs on errors, so
            return redeemFresh(msg.sender, redeemTokens, 0);
    }
```

After accruing interest and checking for errors, `redeemInternal` returns the result from `redeemFresh` with the amount of tokens to redeem passed as the second param.

```
    function redeemFresh(address payable redeemer, uint redeemTokens

            ...

            /* exchangeRate = invoke Exchange Rate Stored() */
            (vars.mathErr, vars.exchangeRateMantissa) = exchangeRate

            ...

            /* If redeemTokensIn > 0: */
            if (redeemTokensIn > 0) {
                    /*
                     * We calculate the exchange rate and the amount
                     *  redeemTokens = redeemTokensIn
                     *  redeemAmount = redeemTokensIn x exchangeRate
                     */
                    vars.redeemTokens = redeemTokensIn;

                    (vars.mathErr, vars.redeemAmount) = mulScalarTru
                    if (vars.mathErr != MathError.NO_ERROR) {
                            return failOpaque(Error.MATH_ERROR, Fail
                    }
```

```
            ...

            vars.err = doTransferOut(redeemer, vars.redeemAmount);

            ...

    }
```

`redeemFresh` retrieves the exchange rate (the same one that `CompoundStrategy._withdraw` uses to calculate the amount to redeem), and uses it to calculate `vars.redeemAmount` which is later transferred to the `redeemer`. This is the amount of underlying ETH that we are redeeming the `CEther` tokens for.

We can carefully copy over the logic used in `CEther` to calculate the amount of underlying ETH to receive, as well as the logic used in `CompoundStrategy._withdraw` to determine how many `CEther` to redeem for the desired output amount of underlying ETH, creating the following test contract in Remix.

```
pragma solidity 0.8.19;

contract MatchCalcs {
    uint constant expScale = 1e18;

    struct Exp {
        uint mantissa;
    }

    function mulScalarTruncate(Exp memory a, uint scalar) pure e
        return truncate(mulScalar(a, scalar));
    }

    function mulScalar(Exp memory a, uint scalar) pure internal
        uint256 scaledMantissa = mulUInt(a.mantissa, scalar);

        return Exp({mantissa: scaledMantissa});
    }

    function mulUInt(uint a, uint b) internal pure returns (uint
        if (a == 0) {
            return 0;
```

```
            }

            uint c = a * b;

            if (c / a != b) {
                return 0;
            } else {
                return c;
            }
        }

        function truncate(Exp memory exp) pure internal returns (uir
            // Note: We are not using careful math here as we're per
            return exp.mantissa / expScale;
        }

        function getToWithdraw(uint256 amount, uint256 exchangeRate)
            return amount * (10 ** 18) / exchangeRate;
        }
    }
```

We run the following example with the current `exchangeRateStored` (at the time of writing) of `200877136531571418792530957` and an output amount of underlying ETH to receive of 1e18 on the function `getToWithdraw`, receiving an output of `4978167337` `CEther`. This is the amount of `CEther` that would be passed to `CEther.redeem`.

Next we can see the output amount of underlying ETH according to `CEther` s logic. We pass the same `exchangeRateStored` value and the amount of `CEther` to redeem: `4978167337`, receiving an output amount of `999999999831558306`, less than the intended amount of 1e18.

Since we receive less than intended to receive from `CEther.redeem`, the `_withdraw` call likely fails at the following check:

```
    require(
            wrappedNative.balanceOf(address(this)) >= amount,
            "CompoundStrategy: not enough"
    );
```

Recommended Mitigation Steps

Rather than computing an amount of `CEther` to redeem, we can instead use the `CEther.redeemUnderlying` function to receive our intended amount of underlying ETH.

cryptotechmaker (Tapioca) confirmed

[H-51] Funds are locked because borrowFee is not correctly implemented in BigBang

Submitted by **0x007**, *also found by* **Koolex**, **0xrugpull_detector**, **0xnev**, *and* **SaeedAlipoor01988**

There's borrowOpeningFee for markets. In Singularity, this fee is accumulated over assets as a reward to asset depositors. In BigBang, assets is USDO which would be minted and burned on borrow, and repay respectively. BigBang does not collect fees, because it uses the same mechanism as Singularity and therefore it would demand more than minted amount from user when it's time to repay.

This results in a bird and egg situation where Users can't fully repay a borrowed amount unless they borrow even more.

Proof of Concept

Let's look at how **borrow** works

```
    function _borrow(
        address from,
        address to,
        uint256 amount
    ) internal returns (uint256 part, uint256 share) {
        uint256 feeAmount = (amount * borrowOpeningFee) / FEE_PRECIS

        (totalBorrow, part) = totalBorrow.add(amount + feeAmount, tr
        require(
            totalBorrowCap == 0 || totalBorrow.elastic <= totalBorrc
            "BigBang: borrow cap reached"
        );
```

```
        userBorrowPart[from] += part;

        //mint USDO
        IUSDOBase(address(asset)).mint(address(this), amount);

        //deposit borrowed amount to user
        asset.approve(address(yieldBox), amount);
        yieldBox.depositAsset(assetId, address(this), to, amount, 0)

        share = yieldBox.toShare(assetId, amount, false);

        emit LogBorrow(from, to, amount, feeAmount, part);
    }
```

As can be seen above, amount would be minted to user, but the userBorrowPart is `amount + fee`. When it's time to repay, user have to return `amount + fee` in other to get all their collateral.

Assuming the user borrowed `1,000 USD0` and borrowOpeningFee is at the default value of `0.5%`. Then the user's debt would be `1,005`. If there's only 1 user, and the totalSupply is indeed `1,000`, then there's no other way for the user to get the extra `5 USD0`. Therefore he can't fully redeem his collateral and would have at least `5 * (1 + collateralizationRate) USD0` worth of collateral locked up. This fund cannot be accessed by the user, nor is it used by the protocol. It would be sitting at yieldbox forever earning yields for no one.

This issue becomes more significant when there are more users and minted amount. If more amount is minted more funds are locked.

It might seem like user Alice could go to the market to buy `5 USD0` to fully repay. But the reality is that he is transferring the unfortunate disaster to another user. Cause no matter what, Owed debts would always be higher than `totalSupply`.

This debt would keep accumulating after each mint and every burn. For example, assuming that one 1 billion of USD0 was minted and 990 million was burned in the first month. totalSupply and hence circulating supply would be 10 million, but user debts would be 15 million USD0. That's 5 million USD that can't be accessed by user nor fee collector.

Recommended Mitigation Steps

borrowOpeningFee should not be added to userBorrowPart. If fee is to implemented, then fee collector should receive collateral or USDO token.

0xRektora (Tapioca) confirmed via duplicate issue 739

[H-52] Attacker can prevent rewards from being issued to gauges for a given epoch in TapiocaOptionBroker

Submitted by **Ruhum**, *also found by* **0xRobocop**, **bin2chen**, **KIntern_NA**, **carrotsmuggler**, **c7e7eff**, **0xnev**, **glcanvas**, **marcKn**, *and* **dirk_y**

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/options/TapiocaOptionBroker.sol#L426

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/tokens/TapOFT.sol#L201

An attacker can prevent rewards from being issued to gauges for a given epoch

Proof of Concept

`TapOFT.emitForWeek()` is callable by anyone. The function will only return a value > 0 the first time it's called in any given week:

```
    ///-- Write methods --
    /// @notice Emit the TAP for the current week
    /// @return the emitted amount
    function emitForWeek() external notPaused returns (uint256)
        require(_getChainId() == governanceChainIdentifier, "cha

        uint256 week = _timestampToWeek(block.timestamp);
        if (emissionForWeek[week] > 0) return 0;

        // Update DSO supply from last minted emissions
        dso_supply -= mintedInWeek[week - 1];
```

```
        // Compute unclaimed emission from last week and add it
        uint256 unclaimed = emissionForWeek[week - 1] - mintedIr
        uint256 emission = uint256(_computeEmission());
        emission += unclaimed;
        emissionForWeek[week] = emission;

        emit Emitted(week, emission);

        return emission;
    }
```

In `TapiocaOptionBroker.newEpoch()` the return value of `emitForWeek()` is used to determine the amount of tokens to distribute to the gauges. If the return value is 0, it will assign 0 reward tokens to each gauge:

```
    /// @notice Start a new epoch, extract TAP from the TapOFT c
    ///         emit it to the active singularities and get the
    function newEpoch() external {
        require(
            block.timestamp >= lastEpochUpdate + EPOCH_DURATION,
            "tOB: too soon"
        );
        uint256[] memory singularities = tOLP.getSingularities()
        require(singularities.length > 0, "tOB: No active singul

        // Update epoch info
        lastEpochUpdate = block.timestamp;
        epoch++;

        // Extract TAP

        // @audit `emitForWeek` can be called by anyone. If it's
        // week, subsequent calls will return `0`.
        //
        // Attacker calls `emitForWeek` before it's executed thr
        // The call to `newEpoch()` will cause `emitForWeek` to
        // That will prevent it from emitting any of the TAP to
        // For that epoch, no rewards will be distributed to use
        uint256 epochTAP = tapOFT.emitForWeek();
        _emitToGauges(epochTAP);

        // Get epoch TAP valuation
```

```
        (, epochTAPValuation) = tapOracle.get(tapOracleData);
        emit NewEpoch(epoch, epochTAP, epochTAPValuation);
    }
```

An attacker who frontruns the call to `newEpoch()` with a call to `emitForWeek()` will prevent any rewards from being distributed for a given epoch.

The reward tokens aren't lost. TapOFT will roll the missed epoch's rewards into the next one. Meaning, the gauge rewards will be delayed. The length depends on the number of times the attacker is able to frontrun the call to `newEpoch()`.

But, it will cause the distribution to be screwed. If Alice is eligible for gauge rewards until epoch x + 1 (her lock runs out), and the attacker manages to keep the attack running until x + 2, she won't be able to claim her reward tokens. They will be distributed in epoch x + 3 to all the users who have an active lock at that time.

Here's a PoC:

```
// tOB.test.ts
    it.only("should fail to emit rewards to gauges if attacker f
        const {
            tOB,
            tapOFT,
            tOLP,
            sglTokenMock,
            sglTokenMockAsset,
            tapOracleMock,
            sglTokenMock2,
            sglTokenMock2Asset,
        } = await loadFixture(setupFixture);

        // Setup tOB
        await tOB.oTAPBrokerClaim();
        await tapOFT.setMinter(tOB.address);

        // No singularities
        await expect(tOB.newEpoch()).to.be.revertedWith(
            'tOB: No active singularities',
        );

        // Register sgl
```

```
        const tapPrice = BN(1e18).mul(2);
        await tapOracleMock.set(tapPrice);
        await tOLP.registerSingularity(
            sglTokenMock.address,
            sglTokenMockAsset,
            0,
        );

        await tapOFT.emitForWeek();

        await tOB.newEpoch();

        const emittedTAP = await tapOFT.getCurrentWeekEmission()

        expect(await tOB.singularityGauges(1, sglTokenMockAsset)
            emittedTAP,
        );
    })
```

Test output:

```
  TapiocaOptionBroker
    1) should fail to emit rewards to gauges if attacker frontru


  0 passing (1s)
  1 failing

  1) TapiocaOptionBroker
       should fail to emit rewards to gauges if attacker frontru

      AssertionError: expected 0 to equal 469157964000000000000
      + expected - actual

      -0
      +469157964000000000000000

      at Context.<anonymous> (test/oTAP/tOB.test.ts:606:73)
      at processTicksAndRejections (node:internal/process/task_q
      at runNextTicks (node:internal/process/task_queues:65:3)
      at listOnTimeout (node:internal/timers:528:9)
      at processTimers (node:internal/timers:502:7)
```

Recommended Mitigation Steps

`emitForWeek()` should return the current week's emitted amount if it was already called:

```
function emitForWeek() external notPaused returns (uint256)
    require(_getChainId() == governanceChainIdentifier, "cha

    uint256 week = _timestampToWeek(block.timestamp);
    if (emissionForWeek[week] > 0) return emissionForWeek[we
    // ...
```

0xRektora (Tapioca) confirmed via duplicate issue 192

LSDan (Judge) increase severity to High

[H-53] Potential 99.5% loss in `emergencyWithdraw()` of two Yieldbox strategies

Submitted by Oxfuje, *also found by* Madalad, paweenp, carrotsmuggler, kaden, c7e7eff, Brenzee, SaeedAlipoor01988, *and* Vagner

https://github.com/Tapioca-DAO/tapioca-yieldbox-strategies-audit/blob/05ba7108a83c66dada98bc5bc75cf18004f2a49b/contracts/lido/LidoEthStrategy.sol#L108

https://github.com/Tapioca-DAO/tapioca-yieldbox-strategies-audit/blob/05ba7108a83c66dada98bc5bc75cf18004f2a49b/contracts/convex/ConvexTricryptoStrategy.sol#L154

99.5% of user funds are lost to slippage in two Yieldbox strategies in case of `emergencyWithdraw()`

Description

Slippage is incorrectly calculated where `minAmount` is intended to be 99.5%, however it's calculated to be only 0.5%, making the other 99.5% sandwichable. The

usual correct `minAmount` slippage calculation in other Yieldbox strategy contracts is `uint256 minAmount = calcAmount - (calcAmount * 50) / 10_000;`

Calculation logic

In `ConvexTriCryptoStrategy` and `LidoEthStrategy` - `emergencyWithdraw()` allows the owner to withdraw all funds from the external pools. the amount withdrawn from the corresponding pool is calculated to be: `uint256 minAmount = (calcWithdraw * 50) / 10_000;` . This is incorrect and only 0.5% of the withdrawal.

Let's calculate with `calcWithdraw = 1000` as the amount to withdrawn from the pool. `uint256 incorrectMinAmount = (1000 * 50) / 10_000 = 5`

The correct calculation would look like this: `uint256 correctMinAmount = calcWithdraw - (calcWithdraw * 50) / 10_000` aka `uint256 correctMinAmount = 1000 - (1000 * 50) / 10_000 = 995`

Withdrawal logic

`emergencyWithdraw()` of Yieldbox Strategy contracts is meant to remove all liquidity from the corresponding strategy contract's liquidity pool.

In the case of `LidoStrategy` the actual withdraw is `curveStEthPool.exchange(1, 0, _toWithdraw, _minAmount)` which directly withdraws from the Curve StEth pool.

In the case of `ConvexTriCryptoStrategy` it's `lpGetter.removeLiquidityWeth(lpBalance, minAmount)` and lpGetter withdraws from the Curve Tri Crypto (USDT/WBTC/WETH) pool via `removeLiquidityWeth()` -> `_removeLiquidity()` -> `liquidityPool.remove_liquidity_one_coin(_amount, __index, __min)` .

These transactions are vulnerable to front-running and **sandwich attacks** so the amount withdrawn is only guaranteed to withdraw the `minAmount` aka 0.5% from the pool which makes the other 99.5% user funds likely to be lost.

Recommended Mitigation Steps

Fix the incorrect `minAmount` calculation to be `uint256 minAmount = calcAmount - (calcAmount * 50) / 10_000;` in ConvexTriCryptoStrategy and LidoEthStrategy.

0xRektora (Tapioca) confirmed via duplicate issue 408

[H-54] Anybody can buy collateral on behalf of other users without having any allowance using the multiHopBuyCollateral()

Submitted by 0xStalin*, also found by* peakbolt*,* plainshift*,* KIntern_NA*,* Ack*, and* rvierdiiev

- Malicious actors can buy collateral on behalf of other users without having any allowance to do so.
- No unauthorized entity should be allowed to take borrows on behalf of other users.

Proof of Concept

- The `SGLLeverage::multiHopBuyCollateral()` function allows users to level up cross-chain: Borrow more and buy collateral with it, the function receives as parameters the account that the borrow will be credited to, the amount of collateral to add (if any), the amount that is being borrowed and a couple of other variables.
- The `SGLLeverage::multiHopBuyCollateral()` function only calls the `solvent()` modifier, which will validate that the account is solvent at the end of the operation.
- The `collateralAmount` variable is used to compute the required number of shares to add the specified `collateralAmount` as extra collateral to the borrower account, then there is a check to validate that the caller has enough allowance to add those shares of collateral, and if so, then the collateral is added and debited to the `from` account

```
...
//add collateral
```

```
uint256 collateralShare = yieldBox.toShare(
    collateralId,
    collateralAmount,
    false
);
_allowedBorrow(from, collateralShare);
_addCollateral(from, from, false, 0, collateralShare);
...
```

- After adding the extra collateral (if any), the **execution proceeds to call the `_borrow()` to ask for a borrow specified by the `borrowAmount` parameter, and finally calls the USDO::sendForLeverage().**
- **The problem is that the function only validates if the caller has enough allowance for the `collateralAmount` to be added, but it doesn't check if the caller has enough allowance for the equivalent of shares of the `borrowAmount` (which is the total amount that will be borrowed!).**
- **The exploit occurs when** a malicious actor calls the `multiHopBuyCollateral()` sending the values of the parameters as follows:
    - `from` => The account that will buy collateral and the borrow will be credited to
    - `collateralAmount` => **Set as 0**
    - `borrowAmount` => **The maximum amount that the `from` account can borrow without falling into insolvency because of the borrowing**
        - What will happen is that a malicious actor without any allowance will be able to skip the check that validates if it has enough allowance to add more collateral, and will be able to take the borrow on behalf of the `from` account, because the `borrowShare` (which represents the equivalent shares to take a borrow of `borrowAmount`) is not used to validate if the caller has enough allowance to take that amount of debt on behalf of the `from` account

Coded a Poc

- I used the `tapioca-bar-audit/test/singularity.test.ts` as the base for this PoC.

- If you'd like to use the original `tapioca-bar-audit/test/singularity.test.ts` file, just make sure to update these two lines as follow:

```
diff --git a/singularity.test.ts b/singularity.test.ts.modified
index 9c82d10..9ba9c76 100755
--- a/singularity.test.ts
+++ b/singularity.test.ts.modified
@@ -3440,6 +3440,7 @@ describe('Singularity test', () => {
         it('should bounce between 2 chains', async () => {
             const {
                 deployer,
+                eoa1,
                 tap,
                 weth,
                 createTokenEmptyStrategy,
@@ -4082,7 +4083,7 @@ describe('Singularity test', () => {
                 ethers.constants.MaxUint256,
             );

-            await SGL_10.multiHopBuyCollateral(
+            await SGL_10.connect(eoa1).multiHopBuyCollateral(
                 deployer.address,
                 0,
                 bigDummyAmount,
```

- I highly recommend to create a new test file with the below code snippet for the purpose of validating this vulnerability, ***make sure to create this file in the same folder as the*** `tapioca-bar-audit/test/singularity.test.ts` ***file.***

▶ Details

- The PoC will demonstrate how an attacker can take borrows on behalf of other users without having any allowance by exploiting a vulnerability in the multiHopBuyCollateral()

Recommended Mitigation Steps

- Make sure to validate that the caller has enough allowance to take the borrow specified by the `borrowAmount`.

- Use the returned amount `borrowShare` from the `_borrow()` to validate if the caller has enough allowance to take that borrow.

```
function multiHopBuyCollateral(
    address from,
    uint256 collateralAmount,
    uint256 borrowAmount,
    IUSDOBase.ILeverageSwapData calldata swapData,
    IUSDOBase.ILeverageLZData calldata lzData,
    IUSDOBase.ILeverageExternalContractsData calldata externalDa
) external payable notPaused solvent(from) {
    ...

    //borrow
    (, uint256 borrowShare) = _borrow(from, from, borrowAmount);

+   //@audit => Validate that the caller has enough allowance to
+   _allowedBorrow(from, borrowShare);

    ...
}
```

[0xRektora (Tapioca) confirmed via duplicate issue 121]

[H-55] `_sendToken` implementation in `Balancer.sol` is wrong which will make the underlying erc20 be send to a random address and lost

Submitted by [Vagner]

The function `_sendToken` is called on `rebalance` to perform the rebalance operation by the owner which will transfer native token or the underlying ERC20 for a specific tOFT token to other chains. This function uses the `router` from Stargate to transfer the tokens, but the implementation of the `swap` is done wrong which will make the tokens to be lost.

Proof of Concept

`_sendToken` calls Stargate's router `swap` function with the all the parameters needed as can be seen here https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/Balancer.sol#L322-L332, but the problem relies that the destination address is computed by calling `abi.encode(connectedOFTs[_oft][_dstChainId].dstOft)` instead of the `abi.encodePacked(connectedOFTs[_oft][_dstChainId].dstOft)` https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/Balancer.sol#L316-L318. Per Stargate documentation https://stargateprotocol.gitbook.io/stargate/developers/how-to-swap , the address of the swap need to casted to bytes by using `abi.encodePacked` and not `abi.encode` , casting which is done correctly in the `_sendNative` function https://github.com/Tapioca-DAO/tapiocaz-audit/blob/bcf61f79464cfdc0484aa272f9f6e28d5de36a8f/contracts/Balancer.sol#L291 . The big difference between `abi.encodePacked` and `abi.encode` is that `abi.encode` will fill the remaining 12 bytes of casting a 20 bytes address with 0 values. Here is an example of casting the address `0x5B38Da6a701c568545dCfcB03FcB875f56beddC4`

```
    bytes normalAbi = 0x0000000000000000000000005b38da6a701c568545dc
    bytes packedAbi = 0x5b38da6a701c568545dcfcb03fcb875f56beddc4;
```

This will hurt the whole logic of the `swap` since when the `lzReceive` function on the `Bridge.sol` contract from Startgate will be called, the address where the funds will be sent will be a wrong address. As you can see here the `lzReceive` on `Bridge.sol` for Abitrum for example uses assembly to load 20 bytes of the `payload` to the `toAddress` https://arbiscan.io/address/0x352d8275aae3e0c2404d9f68f6cee084b5beb3dd#code#F1#L88 which in our case, for the address that I provided as an example it would be

```
    toAddress = 0x0000000000000000000000005b38Da6A701c5685;
```

because `abi.encode` was used instead of `abi.ecnodePacked` . Then it will try to swap the tokens to this address, by calling `sgReceive` on it, which will not exist in

most of the case and the assets will be lost, as specified by Stargate documentation https://stargateprotocol.gitbook.io/stargate/composability-stargatecomposed.sol

Recommended Mitigation Steps

Use `abi.encodePacked` instead of `abi.encode` on `_sendToken`, same as the protocol does in `_sendNative`, so the assumptions will be correct.

0xRektora (Tapioca) confirmed

[H-56] Tokens can be stolen from other users who have approved Magnetar

Submitted by dirk_y, *also found by* Madalad, bin2chen, kutugu, Ack, *0xStalin (*1, 2*),* 0xTheC0der, *cergyk (*1, 2*),* rvierdiiev, *and* erebus

https://github.com/Tapioca-DAO/tapioca-periph-audit/blob/023751a4e987cf7c203ab25d3abba58f7344f213/contracts/Magnetar/MagnetarV2.sol#L622-L635

https://github.com/Tapioca-DAO/tapioca-periph-audit/blob/023751a4e987cf7c203ab25d3abba58f7344f213/contracts/Magnetar/MagnetarV2Storage.sol#L336-L338

https://github.com/Tapioca-DAO/tapioca-periph-audit/blob/023751a4e987cf7c203ab25d3abba58f7344f213/contracts/Magnetar/modules/MagnetarMarketModule.sol#L70

https://github.com/Tapioca-DAO/tapioca-periph-audit/blob/023751a4e987cf7c203ab25d3abba58f7344f213/contracts/Magnetar/modules/MagnetarMarketModule.sol#L212-L241

https://github.com/Tapioca-DAO/tapioca-bar-audit/blob/2286f80f928f41c8bc189d0657d74ba83286c668/contracts/markets/MarketERC20.sol#L84-L91

The `MagnetarV2.sol` contract is a helper contract that allows users to interact with other parts of the Tapioca ecosystem. In order for Magnetar to be able to perform

actions on behalf of a user, the user has to approve the contract as an approved spender (or equivalent) of the relevant tokens in the part of the Tapioca ecosystem the user wants to interact with.

In order to avoid abuse, many of the actions that Magnetar can perform are protected by a check that the owner of the position/token needs to be the `msg.sender` of the user interacting with Magnetar. However, there are some methods that are callable through Magnetar that don't have this check. This allows a malicious user to use approvals other users have made to Magnetar to steal their underlying tokens.

Proof of Concept

As I mentioned above, many of the Magnetar methods have a check to ensure that the `msg.sender` is the "from" address for the subsequent interactions with other parts of the Tapioca ecosystem. This check is performed by the `_checkSender` method:

```
function _checkSender(address _from) internal view {
    require(_from == msg.sender, "MagnetarV2: operator not a
}
```

This function does what it is designed to do, however there are some methods that don't include this protection when they should.

One example is the `MARKET_BUY_COLLATERAL` action that allows a user to buy collateral in a market:

```
else if (_action.id == MARKET_BUY_COLLATERAL) {
    HelperBuyCollateral memory data = abi.decode(
        _action.call[4:],
        (HelperBuyCollateral)
    );

    IMarket(data.market).buyCollateral(
        data.from,
        data.borrowAmount,
        data.supplyAmount,
        data.minAmountOut,
```

```
                    address(data.swapper),
                    data.dexData
                );
            }
```

In the market contract there is an underlying call to check whether the sender has the allowance to buy collateral:

```
    function _allowedBorrow(address from, uint share) internal {
        if (from != msg.sender) {
            if (allowanceBorrow[from][msg.sender] < share) {
                revert NotApproved(from, msg.sender);
            }
            allowanceBorrow[from][msg.sender] -= share;
        }
    }
```

Since the `msg.sender` from the perspective of the market is Magnetar, the user would need to provide a borrow allowance to Magnetar to perform this action through Magnetar.

However, you can see above in the `MARKET_BUY_COLLATERAL` code snippet that there is no call to `_checkSender`. As a result, a malicious user can now pass in an arbitrary `data.from` address to use the allowance provided by another user to perform an unauthorised action. In this case, the malicious user could lever up the user's position to increase the user's LTV and therefore push the user closer to insolvency; at which point the user can be liquidated for a profit.

Another example of this issue is with the `depositRepayAndRemoveCollateralFromMarket` method in `MagnetarMarketModule.sol`. In this instance a malicious user can drain approved tokens from any other user by depositing into the Magnetar yield box:

```
        // deposit to YieldBox
        if (depositAmount > 0) {
            _extractTokens(
                extractFromSender ? msg.sender : user,
                assetAddress,
```

```
                depositAmount
            );
            IERC20(assetAddress).approve(address(yieldBox), depo
            yieldBox.depositAsset(
                assetId,
                address(this),
                address(this),
                depositAmount,
                0
            );
        }
```

This is a small snippet from the underlying `_depositRepayAndRemoveCollateralFromMarket` method that doesn't include a call to `_checkSender` and therefore the malicious user can simply set `extractFromSender` to false and specify an arbitrary user address.

Recommended Mitigation Steps

The `_checkSender` method should be used in every method in `MagnetarV2.sol` and `MagnetarMarketModule.sol` if it isn't already.

0xRektora (Tapioca) confirmed via duplicate issue 106

[H-57] twAML::participate - reentrancy via _safeMint can be used to brick reward distribution

Submitted by **cergyk**

A malicious user can use reentrancy in twAML to brick reward distribution

Proof of Concept

As we can see in `participate` in twAML, the function `_safeMint` is used to mint the voting position to the user;

However this function executes a callback on the destination contract: `onERC721Received`, which can then be used to reenter:

```
// Mint twTAP position
tokenId = ++mintedTWTap;
_safeMint(_participant, tokenId);
```

The `_participant` contract can reenter in `exitPosition`, and release the position since,

```
require(position.expiry <= block.timestamp, "twTAP: Lock not exp
```

`position.expiry` is not set yet.

However we see that the following effects are executed after `_safeMint`:

```
weekTotals[w0 + 1].netActiveVotes += int256(votes);
weekTotals[w1 + 1].netActiveVotes -= int256(votes);
```

And these have a direct impact on reward distribution; The malicious user can use reentrancy to increase `weekTotals[w0 + 1].netActiveVotes` by big amounts without even locking her tokens;

Later when the operator wants to distribute the rewards:

```
function distributeReward(
    uint256 _rewardTokenId,
    uint256 _amount
) external {
    require(
        lastProcessedWeek == currentWeek(),
        "twTAP: Advance week first"
    );
    WeekTotals storage totals = weekTotals[lastProcessedWeek];
    IERC20 rewardToken = rewardTokens[_rewardTokenId];
    // If this is a DBZ then there are no positions to give the
    // Since reward eligibility starts in the week after lockinc
    // no way to give out rewards THIS week.
    // Cast is safe: `netActiveVotes` is at most zero by constru
    // weekly totals and the requirement that they are up to dat
```

```
        // TODO: Word this better
        totals.totalDistPerVote[_rewardTokenId] +=
            (_amount * DIST_PRECISION) /
            uint256(totals.netActiveVotes);
        rewardToken.safeTransferFrom(msg.sender, address(this), _amo
    }
```

totals.totalDistPerVote[_rewardTokenId] becomes zero

Recommended Mitigation Steps

Use any of these:

- Move effects before _safeMint
- Use nonReentrant modifier

0xRektora (Tapioca) confirmed

[H-58] A user with a TapiocaOFT allowance >0 could steal all the underlying ERC20 tokens of the owner

Submitted by dirk_y, *also found by* bin2chen, carrotsmuggler, 0x73696d616f, *and* chaduke

The `TapiocaOFT.sol` contract allows users to wrap ERC20 tokens into an OFTV2 type contract to allow for seamless cross-chain use.

As with most ERC20 tokens, owners of tokens have the ability to give an allowance to another address to spend their tokens. This allowance should be decremented every time a user spends the owner's tokens. However the `TapiocaOFT.sol _wrap` method contains a bug that allows a user with a non-zero allowance to keep using the same allowance to spend the owner's tokens.

For example, if an owner had 100 tokens and gave an allowance of 10 to a spender, that spender would be able to spend all 100 tokens in 10 transactions.

Proof of Concept

When a user wants to wrap a non-native ERC20 token into a TapiocaOFT they call `wrap` which calls `_wrap` under the hood:

```
function _wrap(
    address _fromAddress,
    address _toAddress,
    uint256 _amount
) internal virtual {
    if (_fromAddress != msg.sender) {
        require(
            allowance(_fromAddress, msg.sender) >= _amount,
            "TOFT_allowed"
        );
    }
    IERC20(erc20).safeTransferFrom(_fromAddress, address(thi
    _mint(_toAddress, _amount);
}
```

If the sender isn't the owner of the ERC20 tokens being wrapped, the allowance of the user is checked. However this isn't checking the underlying ERC20 allowance, but the allowance of the current contract (the TapiocaOFT).

Next, the underlying ERC20 token is transferred from the owner to this address. This decrements the allowance of the sender, however the sender isn't the original message sender, but this contract.

In order to use this contract as an owner (Alice) I would have to approve the `TapiocaOFT` contract to spend my ERC20 tokens, and it is common to approve this contract to spend all my tokens if I trust the contract. Now let's say I approved another user (Bob) to spend some (let's say 5) of my `TapiocaOFT` tokens. Bob can now call `wrap(aliceAddress, bobAddress, 5)` as many times as he wants to steal all of Alice's tokens.

Recommended Mitigation Steps

In my opinion you shouldn't be able to wrap another user's ERC20 tokens into a different token, because this is a different action to spending. Also, there is no way to decrement the allowance of the user (of the TapiocaOFT token) in the same call as

we aren't actually transferring any tokens; there is no function selector in the ERC20 spec to decrease an allowance from another contract.

Therefore I would suggest the following change:

```
diff --git a/contracts/tOFT/BaseTOFT.sol b/contracts/tOFT/BaseTO
index 5658a0a..e8b7f63 100644
--- a/contracts/tOFT/BaseTOFT.sol
+++ b/contracts/tOFT/BaseTOFT.sol
@@ -350,12 +350,7 @@ contract BaseTOFT is BaseTOFTStorage, ERC20
         address _toAddress,
         uint256 _amount
     ) internal virtual {
-        if (_fromAddress != msg.sender) {
-            require(
-                allowance(_fromAddress, msg.sender) >= _amount,
-                "TOFT_allowed"
-            );
-        }
+        require (_fromAddress == msg.sender, "TOFT_allowed");
         IERC20(erc20).safeTransferFrom(_fromAddress, address(th
         _mint(_toAddress, _amount);
     }
```

OxRektora (Tapioca) confirmed

[H-59] The BigBang contract take more fees than it should

Submitted by OxRobocop*, also found by* mojito_auditor*,* KIntern_NA*,* xuwinnie*, and* rvierdiiev

The repay function in the BigBang contract is used for users to repay their loans. The mechanics of the function are simple:

- 1. Update the exchange rate
- 2. Accrue the fees generated
- 3. Call internal function _repay

The internal function _repay handles the state changes regarding the user debt. Specifically, fees are taken by withdrawing all the user's debt from yieldbox and burning the proportion that does not correspond to fees. The fees stay in the contract's balance to later be taken by the penrose contract. The logic can be seen here:

```
    function _repay(
            address from,
            address to,
            uint256 part
        ) internal returns (uint256 amount) {
            (totalBorrow, amount) = totalBorrow.sub(part, true);
            userBorrowPart[to] -= part;

            uint256 toWithdraw = (amount - part); //acrrued
            // @audit-issue Takes more fees than it should
            uint256 toBurn = amount - toWithdraw;
            yieldBox.withdraw(assetId, from, address(this), amount,
            //burn USDO
            if (toBurn > 0) {
                IUSDOBase(address(asset)).burn(address(this), toBurr
            }

            emit LogRepay(from, to, amount, part);
        }
```

The problem is that the function burns less than it should, hence, taking more fees than it should.

Proof of Concept

I will provide a symbolic proof and coded proof to illustrate the issue. To show the issue clearly we will assume that there is no opening fee, and that the yearly fee is of 10%. Hence, for the coded PoC it is important to change the values of `bigBangEthDebtRate` and `borrowOpeningFee`:

```
    // Penrose contract
    bigBangEthDebtRate = 1e17;

    // BigBang contract
```

```
borrowOpeningFee;
```

Symbolic

How much fees do the protocol should take?. The answer of this question can be represented in the following equation:

```
ProtocolFees = CurrentUserDebt - OriginalUserDebt
```

The fees accrued for the protocol is the difference of the current debt of the user and the original debt of the user. If we examine the implementation of the _repay function we found the next:

```
//uint256 amount;
(totalBorrow, amount) = totalBorrow.sub(part, true);
userBorrowPart[to] -= part;

uint256 toWithdraw = (amount - part); //acrrued
// @audit-issue Takes more fees than it should
uint256 toBurn = amount - toWithdraw;
yieldBox.withdraw(assetId, from, address(this), amount, 0);
//burn USDO
if (toBurn > 0) {
    IUSDOBase(address(asset)).burn(address(this), toBurn);
}
```

The important variables are:

- 1. `part` represents the base part of the debt of the user
- 2. `amount` is the elastic part that was paid giving `part` , elastic means this is the real debt.

At the following line the contract takes `amount` which is the real user debt from yield box:

```
yieldBox.withdraw(assetId, from, address(this), amount, 0);
```

Then it burns some tokens:

```
if (toBurn > 0) {
   IUSDOBase(address(asset)).burn(address(this), toBurn);
}
```

But how `toBurn` is calculated?:

```
uint256 toWithdraw = (amount - part); //acrrued
uint256 toBurn = amount - toWithdraw;
```

`toBurn` is just `part`. Hence, the contract is computing the fees as:

`ProtocolFees = amount - part`. Rewriting this with the first equation terms will be:

`ProtocolFees = CurrentDebt - part`.

But it is `part` equal to `OriginalDebt` ?. Remember that `part` is not the actual debt, is just the part of the real debt to be paid, this can be found in a comment in the code:

```
elastic = Total token amount to be repayed by borrowers, base =
```

So they are equal only for the first borrower, but for the others this wont be the case since the relation of `elastic` and `part` wont be 1:1 due to accrued fees, making `part < OriginalDebt`, and hence the protocol taking more fees. Let's use some number to showcase it better:

```
TIME = 0

First borrower A asks 1,000 units, state:

part[A] = 1000
total.part = 1000
```

```
total.elastic = 1000


TIME = 1 YEAR

part[A] = 1000 --> no change from borrower A
total.part = 1000 --> no change yet
total.elastic = 1100 --> fees accrued in one year 100 units

Second borrower B asks 1,000 units, state:

part[B] = 909.09
total.part = 1909.09
total.elastic = 2100

B part was computed as:

1000 * 1000 / 1100 = 909.09

TIME = 2 YEAR

Fees are accrued, hence:

total.elastic = 2100 * 1.1 = 2310.

Hence the total fees accrued by the protocol are:

2310 - 2000 = 310.

These 310 are collected from A and B in the following proportior

A Fee = 210
B Fee = 100

Borrower B produced 100 units of fees, which makes sense, he ask
```

When B repays its debt, he needs to repay 1,100 units. Then the contract burns the proportion that was real debt, the problem as stated above is that the function burns the `part` and not the original debt, hence the contract will burn 909.09 units. Hence it took:

1100 - 909.09 = 190.91 units

The contract took 190.91 in fees rather than 100 units.

Coded PoC

Follow the next steps to run the coded PoC:

- 1.- Make the contract changes described at the beginning.
- 2.- Add the following test under `test/bigBang.test.ts` :

▶ Details

Tools Used

Hardhat

Recommended Mitigation Steps

Not only store the user borrow part but also the original debt which is `debtAsked + openingFee` . So, during repayment the contract can compute the real fees generated.

0xRektora (Tapioca) confirmed

[H-60] twTAP.claimAndSendRewards() will claim the wrong amount for each reward token due to the use of wrong index

Submitted by **chaduke**, *also found by* **bin2chen**, **KIntern_NA**, **0xRobocop**, *and* **rvierdiiev**

Detailed description of the impact of this finding. twTAP.claimAndSendRewards() will claim the wrong amount for each reward token due to the use of wrong index. As a result, some users will lose some rewards and others will claim more rewards then they deserve.

Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

twTAP.claimAndSendRewards() allows the tapOFT to claim and send a list of rewards indicated in `_rewardTokens`.

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/governance/twTAP.sol#L361-L367

It calls the function `_claimRewardsOn()` to achieve this:

https://github.com/Tapioca-DAO/tap-token-audit/blob/59749be5bc2286f0bdbf59d7ddc258ddafd49a9f/contracts/governance/twTAP.sol#L499-L519

Unfortunately, at L509, it uses the index of `i` instead of the correct index of `claimableIndex`. As a result, the amount that is claimed and transferred for each reward is wrong.

Tools Used

VSCode

Recommended Mitigation Steps

We need to use index `claimableIndex` instead of `i` for function `_claimRewardsOn()`:

```
    function _claimRewardsOn(
            uint256 _tokenId,
            address _to,
            IERC20[] memory _rewardTokens
        ) internal {
            uint256[] memory amounts = claimable(_tokenId);
            unchecked {
                uint256 len = _rewardTokens.length;
                for (uint256 i = 0; i < len; ) {
                    uint256 claimableIndex = rewardTokenIndex[_rewar
-                       uint256 amount = amounts[i];
+                       uint256 amount = amounts[claimableIndex];

                    if (amount > 0) {
```

```
                    // Math is safe: `amount` calculated safely
                    claimed[_tokenId][claimableIndex] += amount;
                    rewardTokens[claimableIndex].safeTransfer(_t
                }
                ++i;
            }
        }
    }
```

[0xRektora (Tapioca) confirmed]

Medium Risk Findings (99)

[M-01] `getDebtRate()` is view and reads `ethMarket.getTotalDebt` allowing for manipulations

Submitted by [GalloDaSballo]

Status: [Sponsor Confirmed]

[M-02] Single UniswapV3Swapper using a single fee makes it highly likely to be suboptimal

Submitted by [GalloDaSballo] *(View multiple* [reports] *submitted by additional wardens)*

Status: [Sponsor Confirmed]

[M-03] `StargateStrategy#_currentBalance` calculation is incorrect and may lead to DoS

Submitted by [Madalad]*, also found by* [bin2chen]

Status: [Sponsor Confirmed]

[M-04] `StargateStrategy#_withdraw` : ether becomes trapped in the contract whenever a user withdraws

Submitted by [Madalad]

Status: [Sponsor Confirmed]

[M-05] `MagnetarV2#burst` double counts `msg.value` for `TOFT_WRAP` operation, making the transaction revert unless the user overpays

Submitted by Madalad *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 207

[M-06] Oracle is susceptible to manipulation if deployed on Optimism

Submitted by ladboy233*, also found by* 0xSmartContract

Status: Sponsor Confirmed

[M-07] `YearnStrategy` is ignoring the `lockedProfits` , giving away all of the Yield to laggard depositors

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-08] In case of Loss to the Yearn Vault, the Contract will stop working until the loss is repaid

Submitted by GalloDaSballo *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 96

[M-09] LidoETHStrategy buys stETH at 1-1 instead of buying it from the Pool at Discount

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-10] LidEthStrategys Hardcoded 2.5% slippage allows stealing all tokens above $2MLN

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-11] Curve Strategy Yield can be Lost by Griefing due to Delta Balance Check

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-12] Convex `BaseRewardPool` allows Claim on Behalf which causes delta to break - Loss of all Rewards

Submitted by GalloDaSballo, *also found by* minhtrng

Status: Sponsor Confirmed via duplicate issue 1688

[M-13] Missing deadline checks allow pending transactions to be maliciously executed

Submitted by Sathish9098 *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 1513

[M-14] `exitPositionAndRemoveCollateral()` will fail as `MagnetarV2` does not implement `onERC721Received()`

Submitted by peakbolt

Status: Sponsor Confirmed, but disagreed with severity

[M-15] `multiHopSell` and `multiHopBuy` can be frontrunned with high slippage tolerance

Submitted by xuwinnie

Status: Sponsor Confirmed

[M-16] `TapiocaOptionLiquidityProvision.registerSingularity()` not checking for duplicate assetIds leading to multiple issues

Submitted by zzzitron

Status: Sponsor Confirmed

[M-17] Incorrect accounting for yieldBoxShares in SGLLiquidation results in wrongly read values

Submitted by unsafesol

Status: Sponsor Confirmed

[M-18] User could be forced to withdraw more amount than desired when calling `retrieveFromStrategy`

Submitted by xuwinnie

Status: Sponsor Confirmed

[M-19] token mights stuck in MagnetarMarketModule contract if the asset doesn't support cross-chain operation

Submitted by jasonxiale, *also found by* Madalad

Status: Sponsor Confirmed

[M-20] Tricrypto on arbitrum should not be used as collateral due to virtual_price manipulation due to Vyper 2.15, .16 and 3.0 bug

Submitted by GalloDaSballo, *also found by* carrotsmuggler

Status: Sponsor Acknowledged

[M-21] CompoundStrategy `_currentBalance` uses `exchangeRateStored` which is leaks value

Submitted by GalloDaSballo *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-22] MEV Attack on `stkAAVE` causes the AaveStrategy to give away all of the Yield

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-23] Airdropped tokens can be stolen by a bot

Submitted by windhustler

Status: Sponsor Confirmed

[M-24] Cannot use CurveSwapper when calling compound due to mismatched data parameter

Submitted by ayeslick

Status: Sponsor Confirmed

[M-25] Using `setBigBangEthMarketDebtRate` or `setBigBangConfig` cause incorrect interest calculation due to retroactively applying the interest rate

Submitted by GalloDaSballo, *also found by rvierdiiev (*1, 2*)*

Status: Sponsor Confirmed via duplicate issue 120

[M-26] Burning FlashFee breaks a core protocol invariant

Submitted by GalloDaSballo, *also found by* jaraxxus

Status: Sponsor Confirmed

[M-27] Multihop buying and selling of collateral will fail due to missing gas payment

Submitted by peakbolt

Status: Sponsor Confirmed

[M-28] TOFT `exerciseOption` fails due to not passing `msg.value` properly

Submitted by windhustler *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-29] `TapiocaOptionLiquidityProvision` stores amount which cause Socialization of Loss when unlocking

Submitted by GalloDaSballo

Status: Sponsor Confirmed

[M-30] `TapiocaOptionLiquidityProvision` causes Loss of Yield when depositing and withdrawing from Singularity - should use shares to track balances

Submitted by GalloDaSballo *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-31] `extractTAP()` function can allow minting an infinite amount in one week, leading to a DoS attack in `emitForWeek()`

Submitted by mojito_auditor *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 728

[M-32] YearnStrategy rounding down when calculating `toWithdraw` could result in insufficient withdrawal amount

Submitted by mojito_auditor

Status: Sponsor Confirmed, but disagreed with severity

[M-33] `emitForWeek` will lose `emissionForWeek` if one week is skipped

Submitted by GalloDaSballo *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 549

[M-34] `BaseTOFT.sendToYBAndBorrow()` will fail when withdrawing the borrowed asset to another chain

Submitted by peakbolt

Status: Sponsor Confirmed

[M-35] `ARBTriCryptoOracle` is vulnerable to read-only reentrancy

Submitted by IIIIIII *(View multiple* reports *submitted by additional wardens)*
Status: Sponsor Acknowledged via duplicate issue 704

[M-36] `BaseTOFTSTrategyModule.strategyWithdraw()` cross chain call will fail due to missing approvals

Submitted by peakbolt *also found by* xuwinnie
Status: Sponsor Confirmed via duplicate issue 759

[M-37] Seer.get uses a view fetcher, breaking the intended use

Submitted by GalloDaSballo
Status: Sponsor Confirmed

[M-38] Incorrect `eligibleAmount` for `AirdropBroker` Phase 3

Submitted by peakbolt*, (View multiple* reports *submitted by additional wardens)*
Status: Sponsor Confirmed via duplicate issue 173

[M-39] Incorrect refund address for `BaseTOFT.retrieveFromStrategy()` prevents gas refund to user

Submitted by peakbolt
Status: Sponsor Confirmed, but disagreed with severity

[M-40] BigBang and Singularity should not pause repay() and liquidate()

Submitted by peakbolt *(View multiple* reports *submitted by additional wardens)*
Status: Sponsor Confirmed

[M-41] Tapioca Bar: Unusable Market Add Functions in Penrose Contract

Submitted by Limbooo *(View multiple* reports *submitted by additional wardens)*
Status: Sponsor Confirmed via duplicate issue 79

[M-42] BigBang liquidation share is not distributed 100%

Submitted by plainshift *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-43] `_getInterestRate` function in SGLCommon contract accrues incorrect fee

Submitted by KIntern_NA

Status: Disputed

[M-44] SGLLeverage/BigBang `buyCollateral` Can Be Exploited to Steal Asset Approvals & Collateral

Submitted by Ack *(View multiple* reports *submitted by additional wardens)*

Status: Disputed via duplicate issue 147

[M-45] Users can borrow funds without any allowance

Submitted by BPZ

Status: Sponsor Confirmed

[M-46] `totalCollateralShare` state variable not updated in `Singularity` market upon liquidation, resulting in an error on `addCollateral` with skim functionality

Submitted by zzzitron *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed, but disagreed with severity

[M-47] `BaseTOFTMarketModule.sol : removeCollateral` removes collateral from the wrong account

Submitted by carrotsmuggler

Status: Sponsor Confirmed, but disagreed with severity

[M-48] liquidation will fail if the Seer or Oracle reverts instead of returning false

Submitted by zzzitron

Status: Sponsor Confirmed via duplicate issue 34

[M-49] [MC01] Market liquidations can revert due to arithmetic underflow

Submitted by carrotsmuggler, *also found by* Koolex

Status: Sponsor Confirmed

[M-50] [HC07] `SGLLiquidation` : Liquidations will fail if `liquidationAddress` is set

Submitted by carrotsmuggler

Status: Sponsor Acknowledged

[M-51] [MB01] Inadvised hardcoding of pool address in `AaveStrategy.sol`

Submitted by carrotsmuggler, *also found by* 0x007

Status: Sponsor Confirmed via duplicate issue 888

[M-52] [HB09] `emergencyWithdraw` on all strategy contracts useless without a pause mechanism

Submitted by carrotsmuggler *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 1522

[M-53] `SGLBorrow::repay` and `BigBang::repay` uses `allowedBorrow` with the asset amount, whereas other functions use it with share of collateral

Submitted by zzzitron *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 578

[M-54] `YieldBox::deposit` , `YieldBox::withdraw` might lock ERC1155 NFT if deposited/withdrawn with less than 1e8 share

Submitted by zzzitron

Status: Sponsor Acknowledged

[M-55] The sending failure of _lzSend is not considered

Submitted by zhaojie

Status: Sponsor Confirmed

[M-56] read-only reentrancy in Curve Eth pool can lead to funds being stolen from the Lido strategy

Submitted by c7e7eff

Status: Sponsor Acknowledged via duplicate issue 704

[M-57] `_getDiscountedPaymentAmount` doesn't work for tokens with more than 18 decimals

Submitted by GalloDaSballo *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 1104

[M-58] mTapiocaOFT can't be rebalanced because the Balancer in tapiocaz-audit calls swapETH() or swap() of the RouterETH but does not forward ether for the message fee

Submitted by 0x73696d616f *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-59] A portion of stargate token rewards earned by StargateStrategy are permanently locked in the contract

Submitted by kaden *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-60] possible reeentrancy if rewardToken is ERC777 or execute arbitrary code on senders/receivers using hooks

Submitted by adeolu

Status: Sponsor Confirmed via duplicate issue 587

[M-61] [M-01] `SGLCommon._getInterestRate()` : feeFraction multiplied by wrong base amount

Submitted by 0xnev

Status: Sponsor Confirmed

[M-62] SGLLendingCommon.sol: The totalBorrowCap validation is incorrect

Submitted by 0xRobocop, *also found by* xuwinnie

Status: Sponsor Confirmed

[M-63] tOLP tokens that are not unlocked after they have expired cause the reward distribution to be flawed

Submitted by Ruhum, *also found by* KIntern_NA

Status: Sponsor Confirmed, but disagreed with severity

[M-64] Potential loss of value in YieldBox's `depositETHAsset()`

Submitted by 0xadrii *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 983

[M-65] Loss of possible rewards in Curve Gauge

Submitted by SaeedAlipoor01988, *also found by* kaden

Status: Sponsor Confirmed

[M-66] FullMath and TickMath libraries desire overflow behavior

Submitted by 0xfuje *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 138

[M-67] Magnetar V2 - mintFromBBAndLendOnSGL can not lock singularity assets to generate TOLP

Submitted by zzebra83

Status: Sponsor Confirmed

[M-68] Compounding mechanism is broken/flawed in ConvexTricryptoStrategy

Submitted by dirk_y, *also found by* ladboy233

Status: Sponsor Confirmed via duplicate issue 297

[M-69] Inconsistent `deposits` into `lendingPool` in `AaveStrategy.withdraw()` and `AaveStrategy.compound()`

Submitted by LosPollosHermanos

Status: Sponsor Confirmed

[M-70] Swapper contract isn't validated for cross-chain leverage operations

Submitted by dirk_y, *also found by* carrotsmuggler

Status: Sponsor Confirmed

[M-71] `Seer.sol` inherits `OracleMulti.sol` which calls `_getQuoteAtTick` from `OracleMath.sol` , function which would revert when `_getRatioAtTick` is called since it doesn't allow overflow behavior

Submitted by Vagner

Status: Sponsor Confirmed

[M-72] oTAP::participate - Call will always revert if msg.sender is approved but not owner

Submitted by cergyk, *also found by* bin2chen

Status: Sponsor Confirmed

[M-73] All liquidated collateral can be stolen from Singularity and Big Bang

Submitted by Ack, *also found by* Koolex

Status: Sponsor Confirmed, but disagreed with severity

[M-74] Stargate swap parameters perform unnecessary airdrop when rebalancing mTapiocaOFT tokens

Submitted by dirk_y

Status: Sponsor Confirmed

[M-75] Rebalancing mTapiocaOFT of native token forces admin to pay for rebalance amount

Submitted by dirk_y *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed, but disagreed with severity

[M-76] TapiocaOptionBroker::newEpoch - An epoch can be skipped leading for unclaimed tap to distribute to be lost

Submitted by cergyk, *also found by* Udsen

Status: Sponsor Confirmed

[M-77] Loss of COMP reward in CompoundStragety.sol

Submitted by ladboy233, *also found by* rvierdiiev

Status: Sponsor Confirmed via duplicate issue 247

[M-78] AaveStragety#withdraw and emergecyWithdraw can revert if the supply cap is reached or isFrozen flag is on when compounding

Submitted by ladboy233

Status: Sponsor Confirmed

[M-79] MagnetarMarketModule::_exitPositionAndRemoveCollateral - Impossible to exitPosition without unlocking tOlp

Submitted by cergyk

Status: Sponsor Confirmed

[M-80] BigBang/Singularity::sellCollateral - Surplus of collateral with regards to repay amount is never returned to user

Submitted by cergyk, *also found by* ladboy233

Status: Sponsor Confirmed

[M-81] averageMagnitude in TapiocaOptionBroker is updated wrongly

Submitted by 0xWaitress

Status: Sponsor disputed

[M-82] Some actions inside MagnetarV2.burst will not work because msg.value is used inside delegate call

Submitted by rvierdiiev *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-83] USDOOptionsModule.exercise doesn't send refund to user

Submitted by rvierdiiev

Status: Sponsor Confirmed

[M-84] SGLLeverage.multiHopSellCollateral checks swapper on wrong chain

Submitted by rvierdiiev

Status: Sponsor Confirmed

[M-85] User can exercise oTAP options for 3 weeks from a 1 week lock

Submitted by dirk_y, *also found by* cergyk

Status: Sponsor Confirmed, but disagreed with severity

[M-86] Option brokers don't handle oracle decimals correctly when calculating payment amounts

Submitted by dirk_y *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-87] The twTAP multiplier can be compromised with manipulated deposits of low value cost and high duration

Submitted by rokinot *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed

[M-88] Oracle Manipulation using Uniswap V3 pool that is not yet deployed

Submitted by SaeedAlipoor01988

Status: Sponsor Confirmed

[M-89] all deposit and withdraw function in Convex and Curve nativeLP Strategy, apply slippage on internal pricing; which call real-time on chain price from Curve directly and subject to MEV

Submitted by 0xWaitress *(View multiple* reports *submitted by additional wardens)*

Status: Sponsor Confirmed via duplicate issue 245

[M-90] ConvexTricryptoStrategy does not count CVX reward into compoundAmount and thus _currentBalance leading to an under-estimate of TVL

Submitted by 0xWaitress*, also found by* minhtrng

Status: Sponsor Confirmed

[M-91] There is no mechanism to track and resolve bad debt

Submitted by rvierdiiev*, also found by* 0x007

Status: Sponsor Confirmed

[M-92] Executing transfers before reverting (AKA bad execution flow/logic design)

Submitted by erebus

Status: Sponsor Confirmed

[M-93] BigBang Contract: The repay function can be DoSed

Submitted by 0xRobocop, *also found by* peakbolt

Status: Sponsor Confirmed

[M-94] Blocking the receiving channel by claiming long arbitrary rewards token from a twTAP position

Submitted by HE1M

Status: Sponsor Confirmed

[M-95] reverting with long message leads to DoS attack

Submitted by HE1M

Status: Sponsor Confirmed

[M-96] DoS attack by consuming all the gas during minting NFT callback

Submitted by HE1M

Status: Sponsor Confirmed

[M-97] The `owner` is a single point of failure and a centralization risk

Submitted by lllllll-bot

Status: Sponsor Acknowledged

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit competition, but is being included here for completeness.*

[M-98] Unsafe use of `transfer()` / `transferFrom()` with `IERC20`

Submitted by lllllll-bot
Status: Sponsor Confirmed

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit competition, but is being included here for completeness.*

[M-99] Return values of `transfer()` / `transferFrom()` not checked

Submitted by lllllll-bot
Status: Sponsor Confirmed

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit competition, but is being included here for completeness.*

Low Risk and Non-Critical Issues

For this audit, 79 reports were submitted by wardens detailing low risk and non-critical issues. This report by **0xSmartContract** received the top score from the judge.

View all Low Risk and Non-Critical submissions here*.*

Gas Optimizations

For this audit, 19 reports were submitted by wardens detailing gas optimizations. This report by **Sathish9098** received the top score from the judge.

View all Gas Optimization submissions here*.*

Audit Analysis

For this audit, 20 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such

topics as architecture, mechanism, or approach. The report highlighted below by **GalloDaSballo** received the top score from the judge.

View all Audit Analyses here.

Executive Summary

The Tapioca system is comprised of multiple interdependent smart-contract systems.

At the highest level we can separate the system into:

Core:

- Market -> Lending Logic (Includes yield box as it's underlying accounting system) (Assumes using Yield Box strategies that do nothing to mitigate composability risks)

Extra:

- Tokenomics

Periphery:

- Yield Strategies
- Oracle
- Swappers

The interrelation between Market, Tokenomics and Periphery has mostly to do with determining value and exchanging it, this creates additional risks at these "edges".

That said, from a thorough analysis it seems to me like the system has grown to be quite complex, with different levels of risk and attacks that can be performed.

My analysis focuses on the additional risk that comes from the composability within the in-scope systems.

The goal of the analysis is to show my adversarial thought process and to suggest a robust security process to ship the codebase in a state that is maintainable and safe for people to use.

Re-Audit the Market, separately as the Core Primitive

For this reason, it seems reasonable to suggest the following:

- Re-audit Market, with Oracles and Swappers separately Iterate on the Market logic until it's extremely solid and battle tested

Once the Market logic is fully ready, adding additional pieces, such as Yield Generation becomes achievable.

Most importantly, the Market will require:

- Monitoring
- Periphery Contract Creation (optimal swaps, liquidations, MEV, arbitrage)
- Documentation and Evangelizing to ensure enough MEV actors are present to allow efficient markets

All of this to suggest that the Market aspect of the system is already "risky enough" on it's own, it would be beyond rekless for me to suggest deploying the system as a whole as of today due to how likely it is for some periphery aspects to add further attack vectors or break invariant at the core level.

Gradually Introduce Risk, in a segregated manner

After the Market is lindy, each new token, oracle and strategy could be added separately with a capped amount to reduce risk.

This addition should be looked at as a separate security exercise, which could be lead by a separate team.

The extra pieces should be looked very thoroughly as to avoid adding vulnerabilities to the core.

As of today, due to the interrelation between Market, Yield Box, Strategy and Pricing being so uniquely and tightly coupled means that a vulnerability at the Strategy Level (e.g. mispricing), will leak value to a degree that allows bankrupting the market.

Separate the Pricing from the Harvesting

Reducing this could be performed by separating the pricing aspect of the underlying yield strategy from the pricing of the underlying asset.

An example would be spinning up an oracle for the TriCrypto Strategy while allowing the redemption of rewards as pro-rata via a SNX like contract.

Another consideration is pricing of about to be harvested collateral, which is a very low hanging fruit for arbitrage and MEV.

Analysis on Test Coverage, not by line but by scenario

From skimming the tests it seems like they do cover most happy paths, but they don't seem to test for adversarial situations, for example:

- Loss to the strategy
- DOS of the strategy

Systemic Risks and Privileges

The Core System does require governance for managing of collaterals, and debtRates.

These seem to be part a requirement for most Lending Protocols.

That said, the separation of each Collateral into separate Singularities does seem to offer a reduced risk as each pool has to be actively opted in by participants of the marketplace.

When it comes to strategies on the other hand, a higher degree of trust and risks are involved.

While there seems to be direct protection for the principal, the Admin Privilege of triggering Emergency Withdrawals could potentially be used to leak value, especially for strategies that are Single Sided (from Token to LP back to Token), for which Sandwiching is always possible, but FlashLoan imbalance attacks become possible mostly only to the Admin.

Risks to end users

Main risks beside invariants being broken (Why I recommend a second audit of Core and then continous security efforts on periphery), are going to be related to composability.

The strategies integrating into different protocols create an ever moving attack surface that will require active risk management, some of which will also require understanding the tradeoff between risking the invested funds and gaining Yield.

Concerns around the complexity of the system

The main threat I can see is how the system is being “sold as one”, in the idea that all of these components will be launched together, which creates exponentially more complexity and risk than if we were dealing with each component separately.

A great example of the Sponsor understanding this was in Formally Verifying YieldBox:

- YieldBox is done
- We know the risk is at the Strategy Level
- We can focus on the rest

I believe a similar exercise could be done for BigBang and Singularity and would give a very different level of confidence in the Core of the system.

From my years of experience in DeFi I believe I have identified gotchas and risks for all the strategies, some of which just come with the territory, however, building on non fully battle-tested core, adds further uncertainty that we wouldn’t have to explore if BigBang and Singularity were audited and verified independently.

Low hanging fruits for attackers and gotchas

I hope I made a clear case that the biggest area of attack is in the Periphery and in how it relates to the rest of the system.

Being able to manipulate a single price feed can cause extreme damage, even when the system fully works.

Collateral separation is definitely a positive there, however, some collaterals are more popular than others, which may still allow attackers to severely damage the system.

On one hand this can be mitigated via having tiered amounts of capital (raise interest rates if capital goes too high).

On the other hand this is part of the bigger challenge in Decentralized Money Markets.

On Formal Verification

The fact that Formal Verification was done for YieldBox is laudable, however, it's important to keep in mind this is not a silver bullet.

Per the Certora Report, the safety and consistency of YieldBox is reliant on the correct accounting of Strategies, which were Out Of Scope at the time of Formal Verification.

This further highlights how additional code doesn't just add new risks to itself, but also to more foundational code.

From my perspective it's extremely important that each piece is looked at individually first and then the composability of each part is further explored.

Process Risks

The process that has been followed seems very risky and I wouldn't be surprised if a lot of valid findings were found in the CodeArena Audit.

From my experience, any audit with even one High Severity, should be followed by another audit, to make sure that the mitigations have been addressed and that no second-order consequences have been created due to minor changes.

Ideal Process

Due to the inherently high surface area, the first set of exercises should be focused exclusively in securing the BigBang, USD0 and Singularity System.

These contracts would need to be audited to ensure all invariants are addressed, and external risks should be properly modelled (e.g. Oracle as single point of failure).

Realized Process

Only YieldBox has been Formally Verified by Certora, this means that BigBang, USD0 and Singularity haven't.

This to me indicates a lower level of maturity in terms of the security of the system.

Only once these core contracts have been secured, by performing multiple audits and security contests, you should opt to perform similar level of security reviews for the Periphery and Oracle Contracts.

Balancing growth and risk

On one hand guarded launches can offer a way to reduce total value at risk, at the same time, we all know that any exploit can severely undermine the reputation of a project.

From my experience in DeFi there is no such thing as a "small review", any minor change (e.g. the Euler Change for Dust Amounts), can have dramatic second, third or even higher order impacts.

For this reason new types of oracles, and strategies should be introduced after serious scrutiny and reviews are performed, this means that no "small tweak" should ever be added.

At the same time, live monitoring, live fuzzing and a rapid response bug bounty can help mitigate actual value loss while allowing for faster experimentation.

In Summary

Unless no High Severity was found, do not limit yourself to a Mitigation Review and then launch, the downside and the risks are too high.

Instead, consider doing an additional audit for all to participate in, and then proceed with a Guarded Launch and a Bug Bounty.

Allow ample time for SRs to check your code, do not rush these phases as the downside massively outweighs the benefits of rushing.

Suggested Next Steps

- Based on the findings, determine the weaker areas (periphery most likely).

- Harden the Core first, as a separate security exercise.
- Develop a plan to progressively stress test the periphery, while allowing safety.
- Due to the variability of the Strategies, you should have Active Monitoring Setup with the goal of Pausing and Deprecating Strategies as fast as possible.

Time spent

50 hours

[cryptotechmaker (Tapioca) acknowledged]

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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