TapiocaDAO Security Review

Pashov Audit Group

Conducted by: dirk_y, peanuts, Dan Ogurtsov

February 12th 2024 - February 19th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **tapioca-periph, tapioca-yieldbox-strategies, Tapioca-bar** and **PermitC** repositories are done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About TapiocaDAO

TapiocaDAO is a cross-chain money market, that includes BoringCrypto's Yieldbox. Yieldbox tracks user's deposits via shares, which are used to account for idle funds, while the same funds are simultaneously applied to strategies. Tapioca Periph smart contracts use PermitC. PermitC is an extended approval system for ERC20, ERC721 and ERC1155 which abstracts the approval process and adds in expirations.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- f968050e8468d712b93fdd37382e24f302b626dd
- c431dc5e80690c1d8c3727f5992d519df3d38254
- d91e5c7472ed74f8d51d400a893df1b4b3c087b7
- d1377d5bbd85a781398eadc6e0c2f2b2594e3688

fixes review commit hashes:

- bc07c576f6da578f9ba336f522c6577ffeb70914
- c431dc5e80690c1d8c3727f5992d519df3d38254
- 38752fc217fe83fafa2d648679c698b9554731bd
- c2031ac2e2667ac8f9ac48eaedae3dd52abef559

Scope

The following smart contracts were in scope of the audit:

For `tapioca-periph`:

- `pearImit/PearImit`
- `pearImit/PearImitHandler`
- `pearImit/PearImitHash`

For `tapioca-yieldbox-strategies`:

- `glp/GlpStrategy`
- `sDai/sDaiStrategy`
- `feeCollector`

For `Tapioca-bar`:

- `markets/leverage/AssetToSGLPLeverageExecutor`
- `markets/leverage/AssetTotsDaiLeverageExecutor`
- `markets/leverage/BaseLeverageExecutor`
- `markets/leverage/SimpleLeverageExecutor`

For `PermitC`:

- `PermitC`
- `Errors`
- `Datatypes`
- `Constants`
- `libraries/PermitHash`
- `interfaces/IPermitC`

7. Executive Summary

Over the course of the security review, dirk_y, peanuts, Dan Ogurtsov engaged with TapiocaDAO to review TapiocaDAO. In this period of time a total of **19** issues were uncovered.

Protocol Summary

Protocol Name	TapiocaDAO
Date	February 12th 2024 - February 19th 2024
Protocol Type	Strategies and permits

Findings Count

Severity	**Amount**
Critical	2
High	3
Medium	6
Low	8
Total Findings	**19**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Attacker can prevent sDaiStrategy fees from ever being collected	Critical	Resolved
[C-02]	Attackers can steal unclaimed rewards from GlpStrategy because the accounted balance does not include unclaimed rewards	Critical	Resolved
[H-01]	Emergency withdrawal allows depositing again	High	Resolved
[H-02]	BaseLeverageExecutor doesn't have a receive() method to receive ETH withdrawals from WETH	High	Resolved
[H-03]	AssetTotsDaiLeverageExecutor.getAsset() should use redeem(collateralAmountIn) instead of redeem(convertToShares(collateralAmountIn))	High	Resolved
[M-01]	Fees are improperly calculated in sDaiStrategy.sol and thus subsequent depositors cannot withdraw their full deposits	Medium	Resolved
[M-02]	GlpStrategy withdrawals can fail if shouldBuyGLP is false	Medium	Resolved
[M-03]	Slippage in GlpStrategy will impact the value of shares for different users	Medium	Resolved
[M-04]	sDaiStrategy can be bricked by a single attacker, requiring manual intervention to resolve	Medium	Resolved
[M-05]	Users can't withdraw from sDaiStrategy when it is paused	Medium	Resolved
[M-06]	Using a permit together with an external call allows for griefing attacks	Medium	Resolved
[L-01]	Setting minimum slippage	Low	Resolved
[L-02]	Dangerous payable function in AssetTotsDaiLeverageExecutor and	Low	Resolved

	AssetToSGLPLeverageExecutor contracts		
[L-03]	FEE_BPS in feeCollector.sol can be set to an arbitrary value	Low	Resolved
[L-04]	permitTransferFromERC20 in PermitC uses uint256 as permitAmount whereas maximum approval is set to uint200 only	Low	Acknowledged
[L-05]	msg.sender is not hashed in the permit	Low	Acknowledged
[L-06]	Setting approval to 0 does not set the validity only to the context of the current block in PermitC.	Low	Acknowledged
[L-07]	permitBatchTransferFrom() in PearImit.sol does not handle errors	Low	Resolved
[L-08]	Pearlmit batch approval deadlines incorrectly used	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Attacker can prevent `sDaiStrategy` fees from ever being collected

Severity

Impact: High, since zero fees will be captured by the strategy

Likelihood: High, since this attack can be easily performed and persisted by a single attacker

Description

In the `sDaiStrategy`, when a user deposits assets into the strategy, the deposited assets are only placed into the underlying pool when the `depositThreshold` (which is set by the owner) is reached:

```
function _deposited(uint256 amount) internal override nonReentrant {
        if (paused) revert Paused();

        // Assume that YieldBox already transferred the tokens to this address
        uint256 queued = IERC20(contractAddress).balanceOf(address(this));
        totalActiveDeposits += queued; // Update total deposits

        if (queued >= depositThreshold) {
            ITDai(contractAddress).unwrap(address(this), queued);
            dai.approve(address(sDai), queued);
            sDai.deposit(queued, address(this));
            emit AmountDeposited(queued);
            return;
        }
        emit AmountQueued(amount);
    }
```

During this flow, the `totalActiveDeposits` variable is incremented by the contract balance of the deposited asset.

Now, during withdrawals, this variable is used to calculate how many fees are still pending:

```
// Compute the fees
        {
            uint256 _totalActiveDeposits = totalActiveDeposits; // Cache total
            // deposits
            (uint256 fees, uint256 accumulatedTokens) = _computePendingFees
            //(_totalActiveDeposits, maxWithdraw); // Compute pending fees
            if (fees > 0) {
                feesPending += fees; // Update pending fees
            }

            // Act as an invariant, totalActiveDeposits should never be lower
            // than the amount to withdraw from the pool
            totalActiveDeposits = _totalActiveDeposits + accumulatedTokens -
            // amount; // Update total deposits
        }
```

This logic is important because it makes sure that we're not charging fees multiple times over the same capital deposited in the strategy. The `_computePendingFees` method only charges fees when `maxWithdraw` is greater than `_totalActiveDeposits` (i.e. the strategy has generated yield, of which the fee recipient is due a percentage):

```
function _computePendingFees(uint256 _totalDeposited, uint256 _amountInPool)
        internal
        view
        returns (uint256 result, uint256 accumulated)
    {
        if (_amountInPool > _totalDeposited) {
            accumulated = _amountInPool - _totalDeposited; // Get the occurred
            // gains amount
            (, result) = _processFees(accumulated); // Process fees
        }
    }
```

So, `totalActiveDeposits` should never be greater than `maxWithdraw` otherwise no fees will ever be charged.

An attacker can force this situation to occur by doing the following:

1. The attacker makes a deposit that brings the queued asset balance of the strategy to just below the `depositThreshold`
2. The attacker can now make multiple tiny deposits that keep the asset balance close to but still below the `depositThreshold`

This is a problem because of the following logic:

```
uint256 queued = IERC20(contractAddress).balanceOf(address(this));
totalActiveDeposits += queued; // Update total deposits
```

The `totalActiveDeposits` variable is increased by `~depositThreshold` every time the attacker makes a tiny deposit, yet there are no deposits being made into the underlying sDai pool. As a result the `totalActiveDeposits` variable is now significantly larger than the `maxWithdraw` from the pool and no fees will be collected from the generated yield. This attack could be performed again in the future once accumulated rewards catch up with the inflated active deposit accounting.

Recommendations

In the `_deposited` method the `totalActiveDeposits` variable should only be incremented if the queued deposits are actually greater than the `depositThreshold` and deposited into the sDai pool.

[C-02] Attackers can steal unclaimed rewards from `GlpStrategy` because the accounted balance does not include unclaimed rewards

Severity

Impact: High, since users can steal unclaimed rewards

Likelihood: High, since this can be performed by any user and the only requirement is that there are some unclaimed rewards. The higher the number of unclaimed rewards and the larger the deposit of the user, the higher the impact. A strategy with less frequent deposits is more likely to suffer from this attack.

Description

When a user wants to withdraw assets they have deposited into a YieldBox, the following logic is used in the YieldBox contract:

```
uint256 totalAmount = _tokenBalanceOf(asset);
        if (share == 0) {

            share = amount._toShares(totalSupply[assetId], totalAmount, true);
        } else {
            // amount may be lower than the value of share due to rounding,
            // that's ok
            amount = share._toAmount(totalSupply[assetId], totalAmount, false);
        }

        _burn(from, assetId, share);

        // Interactions
        asset.strategy.withdraw(to, amount);
```

Here we're mapping the shares owned by the user to their share of the assets in the strategy, and then burning the shares and withdrawing the corresponding number of assets from the strategy. The `deposit` flow uses a similar pattern. The `_tokenBalanceOf` method calls into the strategy, which in the `GlpStrategy` case looks like:

```
function _currentBalance() internal view override returns (uint256 amount) {
        amount = sGLP.balanceOf(address(this));
        amount += pendingRewards();
    }
```

So the asset balance of the strategy is the sum of the GLP balance in the strategy and the pending rewards of the strategy, where the pending WETH rewards are exchanged to GLP based on the current exchange rate.

However, these "pending rewards" are actually not technically pending rewards since they have already been harvested from GMX, they just haven't been converted into GLP yet. As a result, the asset <-> share conversion logic in the YieldBox isn't taking into account the actual pending rewards that are still in GMX waiting to be claimed. This is a problem since it allows an attacker to steal a non-proportional amount of the unclaimed rewards for themselves by:

1. Depositing assets when there are unclaimed rewards. Since the unclaimed rewards are not accounted for, they will obtain more shares than they should. During the deposit process, the unclaimed rewards are harvested so the accounted balance will increase, but this is after the shares have been minted.
2. Now when the attacker withdraws their shares, they obtain a portion of the previously unclaimed rewards. To obtain a greater portion of the unclaimed rewards the attacker should make as large a deposit as possible vs the existing assets in the strategy.

Since this all happens atomically, the attacker could use flashloans to steal a significant portion of unclaimed rewards from the strategy.

Recommendations

The `_currentBalance` method should be updated to include actual pending rewards that are still locked in GMX. This would involve querying the appropriate methods on the GMX contracts.

8.2. High Findings

[H-01] Emergency withdrawal allows depositing again

Severity

Impact: High, tokens can be deposited again after emergency withdrawal, mass withdrawal will not be finalized

Likelihood: Medium, emergency withdrawal is an important admin function, but still not a usual operation

Description

The last step of `sDaiStrategy.emergencyWithdraw()` is the conversion to tDai and setting a pause.

```
function emergencyWithdraw() external onlyOwner {
        paused = true; // Pause the strategy

        // Withdraw from the pool, convert to Dai and wrap it into tDai
        uint256 maxWithdraw = sDai.maxWithdraw(address(this));
        sDai.withdraw(maxWithdraw, address(this), address(this));
        dai.approve(contractAddress, maxWithdraw);
        ITDai(contractAddress).wrap(address(this), address(this), maxWithdraw);
    }
```

It means that the Owner will have to disable pause so that users can withdraw tDai (otherwise tDai will stuck on the strategy). But if the Owner disables pause it will also enable deposits again. Moreover, the first deposit will trigger the strategy to deposit all tDai balance again, requiring `emergencyWithdraw()` again.

Recommendations

Consider either managing different pause types for deposits and withdrawals separately or disabling deposits after `emergencyWithdraw()` is called leaving withdrawals enabled. Also, one of the simplest solutions would be just removing the pause for withdrawals.

[H-02] `BaseLeverageExecutor` doesn't have a `receive()` method to receive ETH withdrawals from WETH

Severity

Impact: Medium, since it breaks the leverage executor when the output token is the tOFT for ETH

Likelihood: High, since it will always happen for this output token

Description

The `BaseLeverageExecutor.sol` contract is the base contract that is inherited by all the other leverage executor contracts. It contains the logic for handling token swapping and tOFT wrapping and unwrapping that is common to all the leverage executors.

When the final token of a leverage action is the ETH tOFT, the WETH from the swap is converted to ETH, and the ETH is wrapped in the tOFT:

```
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 issue here is the call to `weth.withdraw` includes a callback to this contract with the amount of ETH corresponding to the amount of WETH we want to burn. In order to receive this ETH we need a `fallback` or `receive` method, however, this contract has neither. As a result, the `withdraw` call will revert and so any leverage calls that finish with the ETH tOFT will always revert.

This applies to both `AssetTotsDaiLeverageExecutor` and `AssetToSGLPLeverageExecutor`. Any calls to `getAsset` where the `assetAddress` is the tOFT for ETH will always fail.

Recommendations

The following snippet should be added to the `BaseLeverageExecutor.sol` contract:

```
receive() external payable {}
```

It can be removed from the `SimpleLeverageExecutor.sol` contract (since it inherits `BaseLeverageExecutor`).

[H-03] AssetTotsDaiLeverageExecutor.getAsset() should use `redeem(collateralAmountIn)` instead of `redeem(convertToShares(collateralAmountIn))`

Severity

Impact: Medium, users will get lesser DAI when converting their sDAI.

Likelihood: High, will happen every time `getAsset()` is called.

Description

sDai is an ERC4626 contract. Dai is the native asset and sDai is the share. When withdrawing assets, if `withdraw()` is called, the function takes in the asset value and returns the amount of the underlying asset. When `redeem()` is called, the function takes in the share value and returns the amount of the underlying asset. Usually, the shares are worth more than the asset, eg 1 sDai : 1.05 Dai.

In AssetTotsDaiLeverageExecutor.getAsset(), the function intends to unwrap tsDai > withdraw sDai > Dai > USDO. When withdrawing sDai, it calls `redeem(convertToShares())`.

```
//unwrap tsDai
        ITOFT(collateralAddress).unwrap(address(this), collateralAmountIn);
        //redeem from sDai
>       uint256 obtainedDai = ISavingsDai(sDaiAddress).redeem(
>           ISavingsDai(sDaiAddress).convertToShares
  (collateralAmountIn), address(this), address(this)
        );
```

Since sDai is already the share, by calling `convertToShares()`, it returns lesser Dai than intended since `convertToShares()` takes in an asset amount and returns the amount of the share. The user will get back lesser Dai and the remaining shares will be stuck in the AssetTotsDaiLeverageExecutor contract.

Recommendations

Recommend not calling `convertToShares()` in `getAsset()`

```
- uint256 obtainedDai = ISavingsDai(sDaiAddress).redeem(
-             ISavingsDai(sDaiAddress).convertToShares
- (collateralAmountIn), address(this), address(this)
+ uint256 obtainedDai = ISavingsDai(sDaiAddress).redeem
+ (collateralAmountIn, address(this), address(this));
```

8.3. Medium Findings

[M-01] Fees are improperly calculated in sDaiStrategy.sol and thus subsequent depositors cannot withdraw their full deposits

Severity

Impact: Medium, some users may not get their full deposit back.

Likelihood: High, will happen everytime `_withdraw()` is called.

Description

The fees in sDaiStrategy.sol are targeted at the yield from depositing Dai to sDai.

```
> (uint256 fees, uint256 accumulatedTokens) = _computePendingFees
//(_totalActiveDeposits, maxWithdraw); // Compute pending fees
            if (fees > 0) {
                feesPending += fees; // Update pending fees
            }
```

If the protocol deposits 1000 Dai for x sDai and gets back 1010 Dai for x sDai in one year, the yield is 10 Dai and if the fee is set at 10%, then the protocol will earn 1 Dai of fees.

The problem is that it is extremely difficult to calculate how much fees to deduct since there are many depositors to sDaiStrategy but only the sDaiStrategy interacts with the sDai contract. This issue is better explained through an example. Assume fees are set at 10%.

There are two depositors, depositor A and depositor B. Depositor A deposits 9990 Dai through sDaiStrategy, and depositor B deposits 10 Dai through sDaiStrategy. The sDaiStrategy then deposits 10000 Dai and gets back x number of sDai shares.

1 year later, assuming the yield is 5%, the x number of sDai shares can now be redeemed for 10500 Dai. For depositor A, he can get back 10,489.5 Dai and for depositor B, he can get back 10.5 Dai. Depositor A tries to withdraw ~10,489.5 (5% yield) Dai. The sDaiStrategy calculates the fee and since the fee is 10%, the `feesPending` variable will be 50 Dai.

Once depositor A withdraws his sum, the sDaiStrategy has 10.5 Dai left inside. There is not enough Dai to withdraw the fees and subsequent depositors will not be able to get their withdrawal back because the owner will withdraw the fees.

Instead of making depositor A pay the fees, depositor A can withdraw his full amount + yield, leaving subsequent depositors to pay the fee for him.

In summary, this is what should happen, with a 500 Dai yield:

- Depositor A gets 10439.55 Dai back (49.95 Dai goes to fees)
- Depositor B gets 10.45 Dai back (0.05 Dai goes to fees)
- Fees accumulated = 50
- Token left in contract = 0

This is what happens instead

- Depositor A gets 10,489.5 Dai back (0 Dai goes to fees)
- Depositor B gets possibly 10.5 Dai back (50 Dai goes to fees), but if fees are taken first, then depositor B gets nothing
- Fees accumulated = 50
- Token left in contract: 10.5

Recommendations

Recommend the sDaiStrategy to have its own accounting system, so that when a user withdraws his deposited amount + yield, the contract can tax the yield directly, instead of having to use a global `maxWithdraw`.

[M-02] `GlpStrategy` withdrawals can fail if `shouldBuyGLP` is false

Severity

Impact: Medium, since withdrawals can fail until `shouldBuyGLP` is set to true

Likelihood: Medium, since `shouldBuyGLP` has to be set to false by the owner

Description

When a user wants to withdraw assets they have deposited into a YieldBox, the following logic is used in the YieldBox contract:

```
uint256 totalAmount = _tokenBalanceOf(asset);
        if (share == 0) {

            share = amount._toShares(totalSupply[assetId], totalAmount, true);
        } else {
            // amount may be lower than the value of share due to rounding,
            // that's ok
            amount = share._toAmount(totalSupply[assetId], totalAmount, false);
        }

        _burn(from, assetId, share);

        // Interactions
        asset.strategy.withdraw(to, amount);
```

Here we're mapping the shares owned by the user to their share of the assets in the strategy, and then burning the shares and withdrawing the corresponding number of assets from the strategy. The `_tokenBalanceOf` method calls into the strategy, which in the `GlpStrategy` case looks like:

```
function _currentBalance() internal view override returns (uint256 amount) {
        amount = sGLP.balanceOf(address(this));
        amount += pendingRewards();
    }
```

So the asset balance of the strategy is the sum of the GLP balance in the strategy and the pending rewards of the strategy, where the pending WETH rewards are exchanged to GLP based on the current exchange rate.

Now, in the withdraw logic in the strategy, the rewards are claimed and the GLP is wrapped into the tOFT for the withdrawer:

```
function _withdraw(address to, uint256 amount) internal override {
        if (amount == 0) revert NotValid();
        _claimRewards(); // Claim rewards before withdrawing
        if (shouldBuyGLP) {
            _buyGlp(); // Buy GLP with WETH rewards
        }
        sGLP.safeApprove(contractAddress, amount);
        ITapiocaOFTBase(contractAddress).wrap(address
        //(this), to, amount); // wrap the sGLP to tsGLP to `to`, as a transfer
        sGLP.safeApprove(contractAddress, 0);
    }
```

However, the problem here is that the WETH rewards are only used to buy more GLP when `shouldBuyGLP` is true.

So, this is a problem because the last withdrawers from the strategy may be unable to withdraw altogether in the case where there is a large amount of pending rewards relative to the balance of GLP in the contract. This is because they will try to claim more GLP than the contract holds based on the conversion from shares to assets in YieldBox.

Recommendations

I can't see why we would ever want `shouldBuyGLP` to be false. Since there is no method to withdraw WETH (besides accrued fees) we always want the WETH rewards to be

used to buy more GLP.

Either `shouldBuyGLP` should always be true (we could remove this variable altogether), or the WETH balance needs to be accounted for and there needs to be special logic to handle the case where there is not sufficient GLP left in the contract to handle withdrawals entirely in GLP; so the withdrawal would have to be part GLP, part WETH. This latter option adds a fair bit of complexity which is why the former might make more sense.

[M-03] Slippage in `GlpStrategy` will impact the value of shares for different users

Severity

Impact: Medium, since the value of shares changes depending on market conditions and the ordering of withdrawals

Likelihood: Medium, since this behavior will be induced at every withdrawal/harvest, but the impact is dependent on the current slippage

Description

When users want to withdraw their assets from the `GlpStrategy`, the value of their shares is calculated based on their portion of shares and the current balance of the strategy:

```
function _currentBalance() internal view override returns (uint256 amount) {
        amount = sGLP.balanceOf(address(this));
        amount += pendingRewards();
    }
```

The `pendingRewards` method returns an **estimate** of the GLP that could be purchased with the contract balance of WETH:

```
function pendingRewards() public view returns (uint256 amount) {
        uint256 wethAmount = weth.balanceOf(address(this));
        uint256 _feesPending = feesPending;
        if (wethAmount > _feesPending) {
            wethAmount -= _feesPending;
            uint256 fee = (wethAmount * FEE_BPS) / 10_000;
            wethAmount -= fee;

            uint256 glpPrice;
            (, glpPrice) = wethGlpOracle.peek(wethGlpOracleData);

            uint256 amountInGlp = (wethAmount * glpPrice) / 1e18;
            amount = amountInGlp - (amountInGlp * _slippage) / 10_000; //0.5%
        }
    }
```

This is an **estimate** because the slippage is set to 0.5% (since this is the maximum slippage we'll accept on the actual `mintAndStakeGlp` call). However, the problem with this logic is that the slippage is subject to change and is impacted by market conditions and it can also be influenced by other users and protocols interacting with GMX.

For example, a user who wants to withdraw their shares from the strategy will be quoted based on slippage of 0.5%, but if the market conditions dictate that the actual slippage is 0.1%, then the user still receives the same number of assets, but now the proportional value of every share left in the pool has increased. In low slippage conditions, the first users to withdraw are impacted the most.

Recommendations

Arguably this could be the intended behaviour by the protocol since you're incentivising users to remain staked in the strategy during low slippage market conditions. However, the fact that this is unfair to some end users does violate the ethics of the protocol.

A potential solution is to keep all accounting (and reward claiming) in sGLP, including fee recipient rewards. The burden of swapping to WETH (if required) can be moved to the fee recipient, which has 2 benefits:

1. They can choose when to perform a swap, which allows them to do so when slippages are low, maximizing the value retained in the protocol/strategy
2. Users withdrawing from the strategy receive fair asset conversions from their shares at all times

[M-04] `sDaiStrategy` can be bricked by a single attacker, requiring manual intervention to resolve

Severity

Impact: Medium, since the contract can be bricked, and this is most easily achieved when minimal funds are at risk

Likelihood: Medium, since this can be easily achieved with minimal stake by a single attacker or inadvertently during normal operation

Description

For both deposit and withdrawal interactions, the `_currentBalance` method is called from the YieldBox to exchange assets for shares and vice versa. If this method were to revert, then all deposits and withdrawals would fail. One potential scenario that would

cause a revert is in the following line if `feesPending` was greater than the sum preceding it (since we'll revert on the underflow):

```
return queued + maxWithdraw - feesPending;
```

So the pending fees to be withdrawn by the admin would have to be greater than the deposits into the strategy. This isn't especially likely during normal operations, but it can be induced by a single attacker.

This can be achieved most easily if the attacker is the first to deposit into the strategy. At this time the exchange rate between assets and shares is 1:1, so if the attacker deposited 100 assets they would receive 100 shares. The attacker would deposit at least `depositThreshold` assets into the strategy to ensure that the assets are actually deposited into sDai and are accruing yield. Now, after some short period of time, the sDai would have accrued yield and the attacker can now withdraw their assets. Since they own 100% of the shares they are able to withdraw all of the assets where:

```
queued + maxWithdraw - feesPending = 0 + assetsDeposited - 0 = assetsDeposited
```

because there are no assets queued and no fees have been captured yet since no withdrawals have taken place. So now, during the withdrawal, the fees are accounted for by incrementing `feesPending`, but the user still withdrawals all the assets that were deposited into the strategy.

After this withdrawal, there are zero assets in the strategy but the `feesPending` variable is >0. As a result, all calls to `_currentBalance` will revert due to underflow. This can be resolved with a manual interaction by the admin to send assets directly to the strategy, but this scenario should never be allowed to occur in the first place.

Recommendations

The final line of `_currentBalance` should be changed to avoid the potential for underflow. Here is a suggested fix:

```
return
   feesPending > queued + maxWithdraw ? 0 : queued + maxWithdraw - feesPending;
```

The pending fees should also be properly accounted for as suggested in the other issue.

[M-05] Users can't withdraw from `sDaiStrategy` when it is paused

Severity

Impact: Medium, reverts when integrating with strategy shares

Likelihood: Medium, since this only applies when the `sDaiStrategy` is paused, which shouldn't happen too frequently and should be transient

Description

At any point in time the owner of the `sDaiStrategy` can either pause the strategy or emergency withdraw (which enables a pause). When the strategy is marked as paused, both deposits and withdrawals cannot be executed.

However, the problem with this logic is that blocking withdrawals from a YieldBox strategy at any time is a big no-no. Even in the event of an exploit scenario, a user should be able to be able to withdraw their share of assets. Even if the relative value of the shares is lower than it could be, a user might want to make that sacrifice in order to pay back borrows elsewhere or fulfill other commitments.

Recommendations

The following line should be removed from the `_withdraw` method:

```
if (paused) revert Paused();
```

[M-06] Using a permit together with an external call allows for griefing attacks

Severity

Impact: Low, the function caller will be frontrun but no funds will be lost

Likelihood: High, anyone can grief a permit.

Description

When calling a permit, the data of the permit will be logged in the blockchain, and anyone is able to frontrun the permit by duplicating the TX arguments. This is not an issue according to the EIP since the permit creator can just create another permit.

However, if the permit is used in conjunction with an external function call, like a `transfer()` call, frontrunning the permit will cause the function to be griefed.

Reference: https://www.trust-security.xyz/post/permission-denied

Recommendations

Check that the allowance of the tokens is still available when calling `_checkBatchPermitData()`. Make sure the user still has the proper allowance before calling `transfer()`.

8.4. Low Findings

[L-01] Setting minimum slippage

In glpStrategy, setting slippage allowance to a low value is dangerous. An extreme scenario is setting to zero, which increases the risk of reverts. It must be some slippage always allowed.

Consider introducing some `_MIN_SLIPPAGE` and do not allow `_slippage` to be below this value.

[L-02] Dangerous payable function in AssetTotsDaiLeverageExecutor and AssetToSGLPLeverageExecutor contracts

Both contracts have a `getCollateral()` function that has a payable modifier. If users were to accidentally send native tokens together with their assets, the native tokens would be stuck in the contract.

```
function getCollateral(
   addressassetAddress,
   addresscollateralAddress,
   uint256assetAmountIn,
   bytescalldatadata
 )
        external
>       payable
        override
        returns (uint256 collateralAmountOut)
    {
```

Consider removing the payable function or having a way to withdraw their native token that is stuck in the contract.

[L-03] FEE_BPS in feeCollector.sol can be set to an arbitrary value

The FEE_BPS is not checked when being set in the constructor.

```
contract FeeCollector {
>   uint256 public immutable FEE_BPS;
    uint256 internal constant FEE_PRECISION = 10_000;
    address public feeRecipient;
    uint256 public feesPending;


    constructor(address _feeRecipient, uint256 feeBps) {
        feeRecipient = _feeRecipient;
>       FEE_BPS = feeBps;
    }
```

It can be set to a value higher than the FEE_PRECISION.

Have a limit in the constructor so that the fees cannot be set greater than 100%.

```
constructor(address _feeRecipient, uint256 feeBps) {
        feeRecipient = _feeRecipient;
+       require(feeBps < 1000, "Fee cannot be more than 10%");
        FEE_BPS = feeBps;
    }
```

[L-04] permitTransferFromERC20 in PermitC uses uint256 as permitAmount whereas maximum approval is set to uint200 only

In PermitC, the permitTransferFromERC20 uses uint256 as the data type.

```
function permitTransferFromERC20(
        address token,
        uint256 nonce,
        uint256 permitAmount,
        uint256 expiration,
        address owner,
        address to,
 >      uint256 transferAmount,
```

However, it only uses uint200 when approving.

```
function approve(
        address token,
        uint256 id,
        address operator,
>       uint200 amount,
        uint48 expiration) external {
        _storeApproval(token, id, amount, expiration, msg.sender, operator);
```

Make sure the data type is the same as how permit2 uses only uint160 for approval and transfer.

[L-05] msg.sender is not hashed in the permit

In PermitC's PermitHash.sol, its hashSingleUsePermit hashes the msg.sender together with the nonce and master nonce.

```
function hashSingleUsePermit(
        address token,
        uint256 id,
        uint256 amount,
        uint256 nonce,
        uint256 expiration,
        uint256 masterNonce
    ) internal view returns (bytes32) {
        return keccak256(
            abi.encode(
                SINGLE_USE_PERMIT_TYPEHASH,
                token,
                id,
                amount,
>               nonce,
>               msg.sender,
                expiration,
>               masterNonce
            )
        );
```

In PearlmitHash.sol, the individual permits do not have a nonce and master nonce, but the whole batch hash has a nonce and a master nonce. However, the whole batch hash does not have a msg.sender hashed together with it.

```
function hashBatchTransferFrom(
        IPearlmit.SignatureApproval[] memory approvals,
        uint256 nonce,
        uint48 sigDeadline,
        uint256 masterNonce
    ) internal pure returns (bytes32) {
        uint256 numPermits = approvals.length;
        bytes32[] memory permitHashes = new bytes32[](numPermits);
        for (uint256 i = 0; i < numPermits; ++i) {
            permitHashes[i] = _hashPermitSignatureApproval(approvals[i]);
        }


        return keccak256(
            abi.encode(
                _PERMIT_BATCH_TRANSFER_FROM_TYPEHASH,
                keccak256(abi.encodePacked(permitHashes)),
>               nonce,
                sigDeadline,
>               masterNonce
               //@audit - no msg.sender
            )
        );
    }
```

Consider hashing the msg.sender in the batch hash as well to increase security and reduce the chances of any signature replay attack.

[L-06] Setting approval to 0 does not set the validity only to the context of the current block in PermitC.

In PermitC, when approving, it states on Line 125:

> If the expiration is 0, the approval is valid only in the context of the current block

However, `_storeApproval()` does not set the expiration to the context of the current block if 0 is input as the expiration.

```
function _storeApproval(
        address token,
        uint256 id,
        uint200 amount,
        uint48 expiration,
        address owner,
        address operator
    ) private {
        PackedApproval storage allowed = _getPackedApprovalPtr
          (owner, token, id, ZERO_BYTES32, operator);
>       allowed.expiration = expiration;
        allowed.amount = amount;
```

In Permit2.sol, when `approve()` is called which calls `updateAmountAndExpiration()`, the expiration is set to block.timestamp if 0 is input as the expiration.

```
function approve(
      addresstoken,
      addressspender,
      uint160amount,
      uint48expiration
    ) external {
        PackedAllowance storage allowed = allowance[msg.sender][token][spender];
>        allowed.updateAmountAndExpiration(amount, expiration);
        emit Approval(msg.sender, token, spender, amount, expiration);
    }
```

```
function updateAmountAndExpiration(
        IAllowanceTransfer.PackedAllowance storage allowed,
        uint160 amount,
        uint48 expiration
    ) internal {
        // If the inputted expiration is 0, the allowance only lasts the
        // duration of the block.
>       allowed.expiration = expiration == 0 ? uint48
  (block.timestamp) : expiration;
        allowed.amount = amount;
    }
```

Set the expiration to be block.timestamp if the input expiration is 0 like how Permit2 does it.

[L-07] permitBatchTransferFrom() in PearImit.sol does not handle errors

Severity

Impact: Low, because the transaction will fail silently and the user will not know which part of the approval fails in the batch. No funds will be lost.

Likelihood: Low, will happen on failed `permitBatchTransferFrom()` calls.

Description

`permitBatchTransferFrom()` in PearImit.sol first checks the approval and then calls one of the transfers, eg `_transferFromERC20()`.

```
function permitBatchTransferFrom
      (IPearlmit.PermitBatchTransferFrom calldata batch) external {
>       _checkPermitBatchApproval(batch);


        uint256 numPermits = batch.approvals.length;
        for (uint256 i = 0; i < numPermits; ++i) {
            IPearlmit.SignatureApproval calldata approval = batch.approvals[i];
            if (approval.tokenType == uint8(IPearlmit.TokenType.ERC20)) {
>               _transferFromERC20
  (approval.token, batch.owner, approval.operator, 0, approval.amount);
            } else if (approval.tokenType == uint8
              (IPearlmit.TokenType.ERC721)) {
>               _transferFromERC721
  (batch.owner, approval.operator, approval.token, approval.id);
            } else if (approval.tokenType == uint8
              (IPearlmit.TokenType.ERC1155)) {
>               _transferFromERC1155(
  approval.token,
  batch.owner,
  approval.operator,
  approval.id,
  approval.amount
);
            }
        }
```

Note that the `transferFrom()` functions are called from PermitC, and it handles the transfer of tokens.

```
function _transferFromERC20(
        address token,
        address owner,
        address to,
        uint256 /*id*/,
        uint256 amount
      ) internal returns (bool isError) {
        isError = _beforeTransferFrom(token, owner, to, ZERO, amount);


        if (!isError) {
            (bool success, bytes memory data) = token.call(
              boolsuccess,
              bytesmemorydata
            ) = token.call
              (abi.encodeWithSelector(IERC20.transferFrom.selector, owner, to, amount
```

This is how PermitC does its singular transfer. `_transferFromERC20()` will return a boolean value, and `transferFromERC20()` will return the error.

```
function transferFromERC20(
        address owner,
        address to,
        address token,
        uint256 amount
    ) external returns (bool isError) {
        _checkAndUpdateApproval(owner, token, ZERO, amount, false);
>       isError = _transferFromERC20(token, owner, to, ZERO, amount);
    }
```

In PearImit.sol, the error is not checked and returned. If a function fails to execute, the user will not know which approval had failed in the batch.

Recommendations

Recommend catching all the errors when calling `permitBatchTransferFrom()` and logging them in an array so the user knows which approval in the batch has failed.

[L-08] `Pearlmit` batch approval deadlines incorrectly used

In `permitBatchApprove`, the approval deadline for the batch of assets is set as the deadline of the signature:

```
function permitBatchApprove
  (IPearlmit.PermitBatchTransferFrom calldata batch) external {
        _checkPermitBatchApproval(batch);

        uint256 numPermits = batch.approvals.length;
        for (uint256 i = 0; i < numPermits; ++i) {
            IPearlmit.SignatureApproval calldata approval = batch.approvals[i];
            __storeApproval(

                                approval.token, approval.id, approval.amount, batch.sigDeadline
            );
        }
    }
```

The docstring in `Pearlmit.sol` indicates that the `SignatureApproval` struct has the `approvalExpiration` field:

```
// batch.approvals.approvalExpiration - expiration of the approval.
```

However the struct doesn't actually include this field; as suggested in the "Recommendations" section below, I believe it should.

The problem with the current logic is that asset approval expiration times will in the majority of cases be a non-trivial amount of time in the future (e.g. 1 month/1 year/etc), since otherwise the approval mechanism has limited value. However, this now means that the signature deadline also has to be a non-trivial amount of time in the future. It's dangerous to sign permits with long expiration times since it increases the window over

which the permits could be leaked/stolen and used maliciously. In this case, the permits directly control token access so the impact would be fairly high if they were stolen.

Having dangling permits with long deadlines increases the risk of permits being used maliciously if leaked/stolen. This affects all batch approvals but it requires the theft/leakage of a signed permit

The `SignatureApproval` struct should be extended to include the `approvalExpiration` field that is specific to each asset in the batch and this expiration should be used in the `__storeApproval` call.

The `permitBatchTransferFrom` method should also enforce the `approvalExpiration` for each asset is in the future, but realistically the signature deadline check will happen first and the signature deadline will ordinarily be less than the approval expiration of the assets in the batch.