Pashov Audit Group

YuzuUSD Security Review

August 28th 2025 - September 1st 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out 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. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

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

4. About YuzuUSD

YuzuUSD is a stable ERC-20 token backed 1:1 by USDC and serves as the core unit of value across the Yuzu protocol. The ecosystem includes StakedYuzuUSD, an ERC-4626 vault for staking YuzuUSD, and YuzuILP, an ERC-20 token representing deposits in the Insurance Liquidity Pool, with minting, redemption, and staking mechanisms designed to balance liquidity, yield, and risk.

5. Executive Summary

A time-boxed security review of the **Telos-Consilium/ouroboros-contracts** repository was done by Pashov Audit Group, during which **unforgiven**, **merlinboii**, **IvanFitro**, **ni8mare** engaged to review **YuzuUSD**. A total of **18** issues were uncovered.

Protocol Summary	
Project Name	YuzuUSD
Protocol Type	Stablecoin
Timeline	August 28th 2025 - September 1st 2025

Review commit hash:

- 6dab29807b9e54d2b41cff9ffccbc63b8442d6a8 (Telos-Consilium/ouroboros-contracts)

Fixes review commit hash:

- b4f1d18d8d21b00394b105e44bdab3739fb3037d (Telos-Consilium/ouroboros-contracts)

Scope

`YuzuUSD.sol` `YuzuILP.sol` `StakedYuzuUSD.sol` `YuzuIssuer.sol` `YuzuOrderBook.sol` `YuzuProto.sol` `interfaces/`

6. Findings

Findings count

Severity	Amount
High	3
Medium	2
Low	13
Total findings	**18**

Summary of findings

ID	Title	Severity	Status
[H-01]	Pending withdrawals in `YuzuILP` contract are not considered in totalAssets calculation	High	**Resolved**
[H-02]	Non-atomic updatePool() enables sandwich attacks and extraction	High	**Resolved**
[H-03]	Fee avoidance possible by uncollected fees in pool accounting	High	**Resolved**
[M-01]	Missing slippage protection allows unexpected asset amounts	Medium	**Resolved**
[M-02]	Sandwiching yield distributions allows users to profit from redemption	Medium	**Resolved**
[L-01]	Missing lower bound check for redeem order incentive	Low	**Resolved**
[L-02]	Constant 1:1 peg between `YuzuUSD` and `USDC` may fail after large protocol loss	Low	Acknowledged
[L-03]	Incorrect rounding in `YuzuProto.previewWithdraw()`	Low	**Resolved**
[L-04]	Incorrect rounding on incentive calculation in `_applyFeeOrIncentiveOnTotal()`	Low	**Resolved**
[L-05]	Returned shares token decimals wrong if asset has over 6 decimals	Low	**Resolved**
[L-06]	Dust withdrawals block admin collateral by reducing `liquidityBufferSize`	Low	**Resolved**

ID	Title	Severity	Status
[L-07]	`maxWithdraw()` return values exceed actual withdrawable amounts due to fee exclusion	Low	**Resolved**
[L-08]	Excessively large max values allowed in `setRedeemDelay()` and `setFillWindow()`	Low	**Resolved**
[L-09]	Missing initialization call for `PausableUpgradeable` contract	Low	**Resolved**
[L-10]	Lack of minimum redemption causes spam for `ORDER_FILLER_ROLE`	Low	**Resolved**
[L-11]	Users can frontrun to escape ILP losses by forfeiting small yield	Low	**Resolved**
[L-12]	Lack of inflation attack mitigation in `StakedYuzuUSD`	Low	Acknowledged
[L-13]	StakedYuzuUSD violates the ERC4626 standard	Low	**Resolved**

High findings

[H-01] Pending withdrawals in `YuzuILP` contract are not considered in totalAssets calculation

Severity

Impact: Medium

Likelihood: High

Description

In `YuzuILP`, when a user creates a redeem order, the contract calculates and records the asset amount they are entitled to based on the vault price at that moment.

```
// from YuzuOrderBook file
function createRedeemOrder(uint256 tokens, address receiver, address owner)
    public
    virtual
    returns (uint256, uint256)
{
    if (receiver == address(0)) {
        revert InvalidZeroAddress();
    }
    uint256 maxTokens = maxRedeemOrder(owner);
    if (tokens > maxTokens) {
        revert ExceededMaxRedeemOrder(owner, tokens, maxTokens);
    }

    uint256 assets = previewRedeemOrder(tokens);
    address caller = _msgSender();
    uint256 orderId = _createRedeemOrder(caller, receiver, owner, tokens, assets);

    emit CreatedRedeemOrder(caller, receiver, owner, orderId, assets, tokens);

    return (orderId, assets);
}
```

However, the tokens being withdrawn are not excluded from vault accounting and remain part of `totalAssets()` and `totalSupply()` until the order is finalized. This means that while the user's redemption value is fixed, the tokens continue to accrue yield within the vault. When the user later finalizes the order, only the originally recorded asset amount is transferred, **while the yield that accumulated during the pending period remains in the vault**. This causes the withdrawing user to lose a portion of their rightful profits, and those profits are instead redistributed to the remaining participants.

Recommendations

The vault should prevent pending withdrawals from accruing additional yield. This can be achieved by adjusting accounting to exclude them from `totalAssets()` and `totalSupply()` as soon as the redeem order is created.

[H-02] Non-atomic updatePool() enables sandwich attacks and extraction

Severity

Impact: High

Likelihood: Medium

Description

The `YuzuILP.updatePool()` lacks atomicity when updating the pool state, allowing manipulation of `poolSize` in a way that can be exploited through sandwich attacks.

The function directly sets `poolSize` from the externally provided input without validating against the current state or considering ongoing user operations.

```
function updatePool(uint256 newPoolSize, uint256 newDailyLinearYieldRatePpm) external
onlyRole(POOL_MANAGER_ROLE) {
    if (newDailyLinearYieldRatePpm > 1e6) {
        revert InvalidYield(newDailyLinearYieldRatePpm);
    }

    poolSize = newPoolSize;
    dailyLinearYieldRatePpm = newDailyLinearYieldRatePpm;
    lastPoolUpdateTimestamp = block.timestamp;

    emit UpdatedPool(newPoolSize, newDailyLinearYieldRatePpm);
}
```

Consider the following scenario:

1. User1 and User2 both deposit `100e6` assets into the pool.
2. The treasury funds the liquidity buffer with their deposits (`200e6` total).
3. Pool Manager prepares to call `updatePool(200e6, 0.1e6)` to set a `10%` yield rate.
4. User1 front-runs this transaction and redeems half their shares, reducing the actual `poolSize`.
5. The `updatePool()` transaction executes, artificially restoring `poolSize` to `200e6`.
6. User1 back-runs with another redemption for their remaining shares, now at an inflated value.
7. User1 extracts more assets than their fair share, effectively stealing from User2.

Proof of Concept: test_audit_sandwichPoolUpdate

Recommendation

The `updatePool()` should maintain atomicity because it updates state that is used to price and account across the entire pool.

The possible solutions are to make it callable only while the protocol is paused and/or add a short cooldown for user operations after any admin update. This gives the operator time to verify state consistency before deposits or redemptions resume.

[H-03] Fee avoidance possible by uncollected fees in pool accounting

Severity

Impact: Medium

Likelihood: High

Description

Both `YuzuILP` and `StakedYuzuUSD` contracts have an accounting issue where fees collected during withdrawals and redemptions are not properly separated from the pool's available assets. This creates an opportunity to minimize fees as the remaining shares benefit from the fees collected by previous redemptions.

- In `YuzuILP._withdraw()`, when a user redeems shares, the fee is deducted from the assets they receive, but the fee amount remains in the `poolSize`.
- Similarly, in `StakedYuzuUSD._initiateRedeem()`, when shares are burned during redeem order initiation, the `totalSupply` decreases, but the underlying assets (including any fees) remain in the contract.

This means the fee stays as part of the pool's available assets, benefiting remaining shareholders rather than being collected by the protocol.

Consider the following scenario: 0. User A and User B deposit assets into the vault and receive shares. - User A deposit 100 `yzUSD` and receive 100 `st-yzUSD`. - User B deposit 100 `yzUSD` and receive 100 `st-yzUSD`. - **Redeem fee is 10%**.

1. User A initiate a full redemption of 100 st-yzUSD`.
   - Applying fee as `(100) - (100*0.1/1.1) = 90.9090909 yzUSD`
   - `totalPendingOrderValue` += 90.9 `yzUSD`
   - `totalSupply` -= 100 `st-yzUSD` = 100 `st-yzUSD`
   - `totalAssets` = 200 `yzUSD` - 90.9 `yzUSD` = 109.1 `yzUSD`

2. User B initiate a full redemption of 100 `st-yzUSD`.

- Now 100 `st-yzUSD` worth `totalAssets() = 109.1` yzUSD`
- Applying fee as `(109.1) - (109.1*0.1/1.1) = 99.1090909` yzUSD`
- `totalPendingOrderValue` += 99.1090909 `yzUSD` = 90.9 + 99.1090909 `yzUSD` = 190 `yzUSD`
- `totalSupply` -= 100 `st-yzUSD` = 0 `st-yzUSD`
- `totalAssets` = 200 `yzUSD` - 190 `yzUSD` = 10 `yzUSD`

3. **Result**:

- Net User A paid 10% fee: `((100 - 90.9090909) / 100) * 100`
- Net User B paid 0.8909091% fee: `((100 - 99.1090909) / 100) * 100`

Proof of Concept: - test_audit_feeHasNotBeenCollected_StateProof - test_audit_feeHasNotBeenCollected_MinimizeRedeemFee

Recommendation

- Properly collect fees from the pool's available assets. When fees are collected during withdrawals or redemptions, the fees should be transferred to a separate fee vault or treasury address rather than remaining in the pool.
- Modify the accounting to track collected fees separately and exclude them from the `totalAssets()` and all vault's calculations.

Medium findings

[M-01] Missing slippage protection allows unexpected asset amounts

Severity

Impact: Medium

Likelihood: Medium

Description

The `ERC4626` interaction functions in `StakedYuzuUSD`, `YuzuUSD`, and `YuzuILP` lack slippage protection mechanisms. Users call functions like `initiateRedeem()` expecting to receive a specific amount of assets based on the current preview, but the actual amount received can differ due to state changes between transaction submission and execution.

- In `StakedYuzuUSD` and `YuzuUSD` contracts, users may suffer from changes in redemption fees.
- In `YuzuILP` contract, users may suffer from both changes in fees and changes in pool size, as this pool is expected to also bear risk for `YuzuUSD`.

The lack of slippage protection means users cannot specify a minimum acceptable amount of assets they're willing to receive, potentially leading to unexpected outcomes.

Recommendation

Allow users to specify their slippage tolerance when interacting with `StakedYuzuUSD`, `YuzuUSD`, and `YuzuILP` contracts.

An additional approach is to implement a deadline parameter that ensures transactions are only executed within a specific time window, reducing the likelihood of significant parameter changes between submission and execution.

[M-02] Sandwiching yield distributions allows users to profit from redemption

Severity

Impact: Medium

Likelihood: Medium

Description

The `StakedYuzuUSD` allows users to deposit and initiate redeem requests immediately after the reward distribution. When rewards are distributed to the `StakedYuzuUSD` contract, the `totalAssets()` increases while `totalSupply()` remains constant, which improve the exchange rate for all shareholders.

However, users can sandwich the reward distribution with the deposit and redeem requests to initiate redemption requests. They can then benefit from the yields that larger than the fee they pay, potentially resulting in a net positive outcome even after accounting for redemption fees.

Proof of Concept:: test_fuzz_audit_sandwichYieldDistribution

Recommendation

Implement a cooldown period between when users deposit and when users can initiate redemption requests.

Low findings

[L-01] Missing lower bound check for redeem order incentive

The `setRedeemOrderFee` function in contract only checks that the fee is not greater than `1e6`, but it does not enforce a lower bound. This allows setting excessively negative values that increase the payout during redeem orders beyond safe limits. Although this depends on admin misconfiguration, it could result in incorrect incentives or unintended asset inflation.

```
    function setRedeemOrderFee(int256 newFeePpm) external onlyRole(REDEEM_MANAGER_ROLE) {
        if (newFeePpm > 1e6) {
            revert FeeTooHigh(SafeCast.toUint256(newFeePpm), 1e6);
        }
        int256 oldFee = redeemOrderFeePpm;
        redeemOrderFeePpm = newFeePpm;
        emit UpdatedRedeemOrderFee(oldFee, newFeePpm);
    }
```

Note: `feeBPM` can be negative for incentives:

```
    function _applyFeeOrIncentiveOnTotal(uint256 assets, int256 feePpm) internal pure returns
(uint256) {
        if (feePpm >= 0) {
            /// @dev Positive fee - reduce assets returned
            uint256 fee = _feeOnTotal(assets, SafeCast.toUint256(feePpm));
            return assets - fee;
        } else {
            /// @dev Negative fee (incentive) - increase assets returned
            uint256 incentive = _feeOnRaw(assets, SafeCast.toUint256(-feePpm));
            return assets + incentive;
        }
    }
```

[L-02] Constant 1:1 peg between `YuzuUSD` and `USDC` may fail after large protocol loss

If the protocol suffers a loss greater than the available insurance coverage, the hard-coded 1:1 price assumption between YuzuUSD and USDC becomes problematic. In such a scenario, the system would not be able to reflect the devalued backing of YuzuUSD, leading to potential insolvency risks or unfair redemptions. It is recommended to introduce a dynamic pricing mechanism that accounts for protocol losses and adjusts the YuzuUSD exchange rate accordingly.

```
    function _convertToShares(uint256 assets, Math.Rounding) internal view override returns
(uint256) {
        return assets * 10 ** _decimalsOffset();
    }

    function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view override
```

```
returns (uint256) {
        if (rounding == Math.Rounding.Floor) {
            return shares / 10 ** _decimalsOffset();
        } else {
            return Math.ceilDiv(shares, 10 ** _decimalsOffset());
        }
    }
```

[L-03] Incorrect rounding in `YuzuProto.previewWithdraw()`

The `YuzuProto.previewWithdraw()` incorrectly calculates the number of shares required to withdraw a specific amount of assets by calling `previewDeposit()`. It incorrectly uses `Math.Rounding.Floor` for converting assets to shares, which is appropriate for deposits but incorrect for withdrawals, as it should round up (`Math.Rounding.Ceil`).

```
/// @notice See {IERC4626-previewWithdraw}
function previewWithdraw(uint256 assets) public view virtual override returns (uint256) {
    uint256 fee = _feeOnRaw(assets, redeemFeePpm);
    uint256 tokens = previewDeposit(assets + fee);
    return tokens;
}
```

However, the usage of `YuzuUSD` and `YuzuILP` contracts is correctly applied in the overridden `previewWithdraw()`.

Consider using `Math.Rounding.Ceil` instead of calling to `previewDeposit()`.

```
-   uint256 tokens = previewDeposit(assets + fee);
+   uint256 tokens = _convertToShares(assets + fee, Math.Rounding.Ceil);
    return tokens;
```

or

```
-   uint256 tokens = previewDeposit(assets + fee);
+   uint256 tokens = super.previewWithdraw(assets + fee);
```

[L-04] Incorrect rounding on incentive calculation in `_applyFeeOrIncentiveOnTotal()`

The `_applyFeeOrIncentiveOnTotal()` applies fees or incentives to assets. When handling negative `feePpm` (incentives), it uses `_feeOnRaw()` which rounds up via `Math.Rounding.Ceil`.

This rounding direction favors users at the protocol's expense, as incentives are consistently calculated slightly higher than mathematically precise.

```
function _applyFeeOrIncentiveOnTotal(uint256 assets, int256 feePpm) internal pure returns
(uint256) {
    if (feePpm >= 0) {
        --- SNIPPED ---
    } else {
        /// @dev Negative fee (incentive) - increase assets returned
@>      uint256 incentive = _feeOnRaw(assets, SafeCast.toUint256(-feePpm));
        return assets + incentive;
    }
}

function _feeOnRaw(uint256 assets, uint256 feePpm) internal pure returns (uint256) {
@>   return Math.mulDiv(assets, feePpm, 1e6, Math.Rounding.Ceil);
}
```

Consider using `Math.Rounding.Floor` for incentive calculation.

```
-       uint256 incentive = _feeOnRaw(assets, SafeCast.toUint256(-feePpm));
+       uint256 incentivePpm = SafeCast.toUint256(-feePpm);
+       uint256 incentive = Math.mulDiv(assets, incentivePpm, 1e6, Math.Rounding.Floor);
```

[L-05] Returned shares token decimals wrong if asset has over 6 decimals

The `YuzuProto` contract and its children (`YuzuUSD` , `YuzuILP`) inherit from OpenZeppelin's `ERC20Upgradeable` . This base contract provides a default `decimals()` function that returns `18` .

```
// ERC20Upgradeable.sol
function decimals() public view virtual returns (uint8) {
    return 18;
}
```

```
// YuzuProto.sol
function _decimalsOffset() internal view virtual returns (uint8) {
    return 12;
}
```

However, the share tokens represented by these contracts should have decimals based on their underlying assets plus any offset.

For instance, `YuzuProto` defines a `_decimalsOffset()` of `12` , which should be added to the underlying asset's decimals. This works correctly when the underlying asset has 6 decimals (like `USDC` or `USDT`), resulting in 18 decimals for the share token. But if the underlying asset has more than 6 decimals (like `DAI` with 18 decimals), the share token should return 30 decimals (`18+12`).

The `decimals()` function should be overridden to properly return the underlying asset's decimals plus the `_decimalsOffset()` to ensure accurate representation of share tokens.

[L-06] Dust withdrawals block admin collateral by reducing `liquidityBufferSize`

The `YuzuProto.withdrawCollateral()` allows admins to withdraw excess collateral from the protocol by specifying the amount of assets they want to withdraw, and this function relies on `liquidityBufferSize()` to determine the maximum withdrawable amount.

The `liquidityBufferSize()` calculation subtracts `totalUnfinalizedOrderValue()` from the contract's asset balance, which can be reduced by pending redemption orders.

```
function withdrawCollateral(uint256 assets, address receiver) public virtual {
    uint256 liquidityBuffer = liquidityBufferSize();
    if (assets > liquidityBuffer) {
        revert ExceededLiquidityBuffer(assets, liquidityBuffer);
    }
    SafeERC20.safeTransfer(IERC20(asset()), receiver, assets);
    emit WithdrawnCollateral(receiver, assets);
}

function liquidityBufferSize() public view virtual override returns (uint256) {
    return super.liquidityBufferSize() - totalUnfinalizedOrderValue();
}
```

When users perform instant withdrawals, they can reduce the available `liquidityBufferSize()` by consuming available liquidity. This creates a situation where legitimate admin collateral withdrawals might fail unexpectedly.

This can occur either through normal user operations or by intentionally executing small instant withdrawals (dust amounts) that incur small/zero fees.

Recommendation

Consider allowing admins to withdraw the maximum available amount by implementing a special flag (such as passing `type(uint256).max`) that would withdraw the remaining available liquidity buffer.

[L-07] `maxWithdraw()` return values exceed actual withdrawable amounts due to fee exclusion

The `YuzuIssuer.maxWithdraw()` returns the maximum amount of assets that can be withdrawn by a user without considering withdrawal fees.

```
function maxWithdraw(address _owner) public view virtual returns (uint256) {
    return Math.min(previewRedeem(_maxRedeem(_owner)), _maxWithdraw(_owner));
}

function _maxWithdraw(address) internal view virtual returns (uint256) {
    return liquidityBufferSize();
}
```

```
function liquidityBufferSize() public view virtual override returns (uint256) {
    return super.liquidityBufferSize() - totalUnfinalizedOrderValue();
}
```

When a user attempts to withdraw assets: 1. User has 100 shares in the vault. 2. `previewRedeem(100)` returns ~90 assets (after 10% fee). 3. `_maxWithdraw()` returns 50 assets as the current maximum withdrawable amount. 4. User calls `withdraw(50, user, user)`. 5. The withdrawal process calculates: ```solidity fee = _feeOnRaw(50, redeemFeePpm) // fee = 50 * 0.1e6 / 1e6 = 5 assets

tokens = previewWithdraw(50) // tokens = 55 shares (equivalent to 50 assets + 5 assets fee) `` 6. The actual assets that should be left in the vault are 55 (50 + 5 fee), which exceeds the reported `_maxWithdraw()` value of 50 assets, yet the transaction is still allowed.

Recommendation

Consider updating `maxWithdraw()` to account for fees by calculating the net withdrawable amount after fees are applied.

[L-08] Excessively large max values allowed in `setRedeemDelay()` and `setFillWindow()`

`setRedeemDelay()` in the `StakedYuzuUSD` and `setFillWindow()` in `YuzuProto` contracts set the waiting period users must wait before finalizing their redemption. The issue is that this value can be set up to `type(uint32).max`, which equals 136 years. If mistakenly set to an extremely high value, some redemptions may never be callable within a human lifetime.

Recommendation: Restrict the maximum delay to a reasonable value that can be completed within a human lifespan.

[L-09] Missing initialization call for `PausableUpgradeable` contract

`YuzuProto` and `StakedYuzuUSD` contracts do not invoke the initialization function of the `PausableUpgradeable` contract. As a result, `PausableUpgradeable` remains improperly initialized, which can lead to unexpected behavior when attempting to pause or unpause the contract.

Recommendation: Ensure that `__Pausable_init()` is called within the `initialize()` function to properly initialize the `PausableUpgradeable` contract.

[L-10] Lack of minimum redemption causes spam for `ORDER_FILLER_ROLE`

Currently in the `createRedeemOrder` function in `YuzuOrderBook`, there is a check to ensure that not more than `maxTokens` are being redeemed. But there is no such check for the minimum amount on withdrawals. Users are allowed to withdraw as little as 1 wei. This would allow spammers to create multiple `createRedeemOrder` transactions with extremely small token amounts for redemption. This is a problem because these created redeem orders are supposed to be fulfilled by the `ORDER_FILLER_ROLE`, who would call the `fillRedeemOrder` function to fulfil these orders. As the orders are very small, the `ORDER_FILLER_ROLE` could spend more on gas than the order amount being fulfilled, especially on a chain like Ethereum. Hence, this results in gas waste for the `ORDER_FILLER_ROLE`. Also, this would cause unnecessary delays for other redeem orders to be processed.

Recommendation

Add a minimum amount for redemptions. Any amount lower than this should not be allowed to be redeemed via `createRedeemOrder`.

[L-11] Users can frontrun to escape ILP losses by forfeiting small yield

The `YuzuILP` (Insurance Liquidity Pool) is designed to accrue yield and cover losses for users. According to the documentation, on **redeem** tokens are priced based on the pool share **excluding the yield accrued since the last pool-size update**.

```
    function _convertToSharesRedeemed(uint256 assets, Math.Rounding rounding) internal view
returns (uint256) {
        // slither-disable-next-line incorrect-equality
        if (poolSize == 0) {
            return totalSupply();
        }
        return Math.mulDiv(totalSupply(), assets, poolSize,
rounding);  //Note: poolSize is used instead of totalAssets() to simulate withdraw on last
update
    }
```

This design aims to prevent "yield sniping," where an attacker deposits after an update and withdraws before the next one to capture most of the yield without taking on risk. To achieve this, any accrued yield since the last update is forfeited when redeeming.

However, this mechanism still allows users to frontrun large losses. If an incident occurs that requires the ILP to cover a large loss, users can immediately redeem their position. By doing so, they only forfeit a small amount of yield accrued since the last update (+ small fee) while escaping a much larger loss from the insurance coverage:

This creates an **asymmetric risk exposure**, where rational users can opt out right before large losses are applied, undermining the insurance design of the protocol.

Recommendations

- Prevent one-step redeem/withdrawals on `YuzuILP`.*
- For two-step redeems, ensure to use the share price in the **second step** which reflects losses, so users cannot escape pending losses by exiting early.

[L-12] Lack of inflation attack mitigation in `StakedYuzuUSD`

The `StakedYuzuUSD` contract is vulnerable to an inflation attack where an attacker can manipulate the share price by directly donating `yzUSD` to the contract.

Since the `_assetDecimalOffset` will be `0` and the asset's balance of `StakedYuzuUSD` is used to calculate the number of shares, it would be possible to inflate the share price by donating the amount of `yzUSD` (obtain from depositing to `YuzuUSD`) directly to the `StakedYuzuUSD` contract.

This donation increases the `totalAssets()` in the vault without minting any new shares, dramatically inflating the price per share.

Proof of concept: test_audit_inflationAttack

Recommendation

There are several ways to mitigate this issue: - Set the value of `_decimalsOffset` to a non-zero value (e.g., 6). - or consider mitigating this with an initial deposit of a small amount (dead shares). - Implementing internal accounting for deposited assets, so the donation does not affect the share price.

[L-13] StakedYuzuUSD violates the ERC4626 standard

Look at the specification for maxWithdraw in EIP4626:

> MUST factor in both global and user-specific limits, like if withdrawals are entirely disabled (even temporarily) it MUST return 0.

In `StakedYuzuUSD`, withdrawals are disabled:

```
function withdraw(uint256, address, address) public pure override returns (uint256) {
    revert WithdrawNotSupported();
}
```

But, `maxWithdraw` still returns a value based on `super.maxRedeem`, which is simply the share balance of the user and converts it into asset value:

```
function maxWithdraw(address _owner) public view override returns (uint256) {
    return previewRedeem(super.maxRedeem(_owner)); //@audit - supposed to return 0?
}
```

This is in violation of the ERC4626 standard. The same argument can also be used for `maxRedeem` function as well, as `redeem` function has also been disabled. Although `maxRedeem` is being used in `initiateRedeem` , its recommended to use another function instead of `maxRedeem` to be in line with the specification.

Also, if the protocol intends to use `maxRedeem` in the same manner, there is also a note about returning 0 on temporary disablement:

> MUST factor in both global and user-specific limits, like if redemption is entirely disabled (even temporarily) it MUST return 0.

So, when the protocol decides to pause the contract, `maxRedeem` must return 0 too. As even `initiateRedeem` is paused as well, `maxRedeem` returning 0 will stay true to the standard.

Recommendations

Return 0 from the above-mentioned functions, under the mentioned cases, to stay compliant with the ERC4626 standard.