yAudit OpenState Review

Review Resources:

- Code repository
- flow.md

Auditors:

- fedebianu
- Jackson

Table of Contents

1 Review Summary

2 Scope

3 Code Evaluation Matrix

4 Findings Explanation

5 Medium Findings

1. Medium - `IRM` parameters change can lead to incorrect interest accrual

2. Medium - Missing maximum fee cap in `PSM` could be viewed as a honeypot mechanism

3. Medium - `ProfitModule` loses value to exit fees when selling profits

4. Medium - `osToken` minting in `ProfitModule` isn't backed by actual profits

6 Low Findings

1. Low - `_A` parameter check is not done across all entry points

2. Low - Potential DoS in `getTotalBalance()` due to unbounded loop

3. Low - Theoretical early-stage DoS attack via interest rounding in `getInterest()`

7 Gas Saving Findings

1. Gas - Cache `psm.length`

8 Informational Findings

1. Informational - Remove unchecked increment of the counter

2. Informational - `FEE_PRECISION` can be reduced from 1e18 to 10_000

3. Informational - Fix typos

4. Informational - `psmBalance`'s NatSpec description is misleading

9 Final remarks

Review Summary

OpenState

OpenState is a decentralized protocol designed to create capital-efficient receipt tokens called synths. These synths can represent any underlying asset; provided there's a reliable risk-free rate source and demand for using these receipt tokens in DeFi, restaking, or other collateral-requiring applications.

The OpenState repository was reviewed over six days. Two auditors performed the code review between December 9th and December 16th, 2024. The repository was under active development during the review, but the review was limited to the latest commit f262668f239eb43fb2711d02a1493baae55be9dd.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
src
├── core
│   ├── IRM.sol
│   ├── Minter.sol
│   ├── OSToken.sol
│   ├── PSM.sol
│   ├── PSMLens.sol
│   └── SOSToken.sol
├── interfaces
│   ├── IIRM.sol
│   ├── IMakerUsdcPSM.sol
│   ├── IMinter.sol
│   ├── IOSToken.sol
│   ├── IPSM.sol
│   ├── IPSMLens.sol
│   ├── ISkyDaiUsdsConverter.sol
│   └── external
│       ├── IMetaMorpho.sol
│       ├── IMetaMorphoFactory.sol
│       ├── IMorpho.sol
│       └── IWSTETH.sol
├── lib
│   └── ErrorsLib.sol
└── periphery
    ├── Lens.sol
    ├── MorphoModule.sol
    ├── ProfitModule.sol
    ├── PxETHPSM.sol
    ├── StETHPSM.sol
    ├── UsdsPSM.sol
    ├── cbBTCPSM.sol
    └── zapper
        ├── osETHZapper.sol
        └── osUSDZapper.sol
```

After the findings were presented to the OpenState team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, OpenState and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Authorization is implemented through OZ AccessManaged.
Mathematics	Good	The code doesn't contain complex mathematical formulas.
Complexity	Average	Logic is spread into multiple contracts. Some of the interactions between modules are complex and require careful consideration, particularly around debt accounting, profit distribution, and interest rate mechanics.
Libraries	Good	The codebase uses an updated version of the OZ library.
Decentralization	Average	Restricted roles manage many critical flows. It will be important to see how this will be implemented in practice.
Code stability	Good	The codebase remained stable during the audit.
Documentation	Average	There is a lack of documentation. Some contracts are not even mentioned. Critical flows and accounting mechanisms are not documented.
Monitoring	Good	The contracts emit events for key operations.
Testing and verification	Average	Unit tests are present, but the coverage should be higher. Integration, fuzz, and invariant tests are missing.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Medium Findings

1. Medium - `IRM` parameters change can lead to incorrect interest accrual

Setting new `IRM` parameters without accruing interest first leads to calculating the intermediate period's interest with the new rates instead of those in effect during that period.

Technical Details

When IRM parameters are changed through `setParameters()`, any unclaimed interest for the period between the last accrual and the parameter change would be calculated using the new rates rather than the rates that were actually in effect during that period.

Impact

Medium. Users could receive more or less interest than they should for the period between the last accrual and the parameter change, depending on whether rates were increased or decreased.

Recommendation

Ensure `accrueInterest()` is called before changing `IRM` parameters. A possible approach involves implementing interest accrual enforcement within the `IRM` contract:

```
+    ISOSToken[] public sosTokens;

+    function addSOSToken(ISOSToken token) external restricted {
+        require(address(token) != address(0), "Invalid token");
+        sosTokens.push(token);
+    }
+
+    function removeSOSToken(uint256 index) external restricted {
+        require(index < sosTokens.length, "Index out of bounds");
+        sosTokens[index] = sosTokens[sosTokens.length - 1];
+        sosTokens.pop();
+    }

     function setParameters(uint256 _maxAPR, uint256 _minAPR, uint256 _A)
```

```
external restricted {
+        for (uint256 i = 0; i < sosTokens.length; i++) {
+            sosTokens[i].accrueInterest();
+        }

         _setParameters(_maxAPR, _minAPR, _A);
     }
```

Developer Response

Acknowledged.

Initially we planned to use a multisig to send transactions regarding changes in the protocol, when we do so we assumed that we will batch them in the following order:

1. accrue sosToken interest
2. set irm parameters

2. Medium - Missing maximum fee cap in `PSM` could be viewed as a honeypot mechanism

The `PSM` allows setting fees up to 100%, potentially enabling a malicious admin to steal user funds effectively.

Technical Details

In `PSM.sol`, fees can be set to `FEE_PRECISION` (100%). A malicious or compromised admin could potentially set `entryFee` or `exitFee` equal to `FEE_PRECISION`, trying to steal users' deposits.

Impact

Medium. While this requires admin privilege abuse, the mere possibility of 100% fees could damage the protocol's credibility and user trust. Even if never exploited, such high fee caps could raise concerns about the protocol being a potential exit scam.

Recommendation

Add a reasonable maximum fee cap.

Developer Response

Fixed in this [PR]

3. Medium - `ProfitModule` loses value to exit fees when selling profits

When the `ProfitModule` claims profits by minting and selling `osToken`, it unnecessarily pays exit fees to the PSM, reducing the protocol's profit.

Technical Details

When the protocol realizes profits through the `ProfitModule`, it mints `osToken` and needs to sell them through a PSM to get the underlying asset. Since both the `ProfitModule` and PSM are part of the same protocol, charging exit fees on profit realization unnecessarily reduces the protocol's profits.

Impact

Medium. The protocol reduces its profits by paying fees when realizing profits, leading to accounting inefficiency and value loss.

Recommendation

Add a flag in PSM to bypass fees for the `ProfitModule`. Here's a possible approach:

```
+     mapping(address => bool) public feeExempt;

      function sell(address recipient, uint256 amount) external whenNotPaused
returns (uint256) {
          if (recipient == address(0)) revert ErrorsLib.InvalidAddress();
          if (amount == 0) revert ErrorsLib.ZeroAmountInput();

          _accrueInterest();

          osToken.safeTransferFrom(msg.sender, address(this), amount);
          minter.burn(amount);

-         uint256 withdrawnAmount = withdraw(recipient, deductFee(amount,
exitFee));
+         uint256 feeToApply = feeExempt[msg.sender] ? 0 : exitFee;
+         uint256 withdrawnAmount = withdraw(recipient, deductFee(amount,
feeToApply));

          emit Sell(msg.sender, recipient, amount, withdrawnAmount);

          return withdrawnAmount;
      }
```

```
+     function setFeeExempt(address account, bool exempt) external restricted {
+         feeExempt[account] = exempt;
+     }
```

Developer Response

Fixed in this PR

4. Medium - `osToken` minting in `ProfitModule` isn't backed by actual profits

The `ProfitModule` can mint tokens based on its debt ceiling without verifying actual profits.

Technical Details

`ProfitModule` is created for a specific `osToken`, but currently mints without verifying actual protocol revenues from its multiple sources of income (PSM fees, yield spreads, and Morpho spreads, etc..). The module can mint up to its debt ceiling even if the actual profits are lower, breaking the profit-backing assumption.

Impact

Medium. While a debt ceiling can be aligned with actual profits to help prevent this, the protocol can still mint more tokens than generated revenues. Such an imbalance may lead to token dilution and undermine the protocol's 1:1 backing mechanism. This can also be seen as a potential rug vector.

Recommendation

On-chain tracking and verification of all revenue sources before minting is challenging. To ensure user trust, consider adding a timelock for-profit claims.

Developer Response

Acknowledged. We can add a timelock controller infront of it.

Low Findings

1. Low - `_A` parameter check is not done across all entry points

`_A` parameter validation is only implemented in the constructor, allowing potential zero value setting through `setParameters()` that would break interest rate calculations.

Technical Details

In `IRM.sol`, the validation for parameter `A` is only present in the constructor but missing in the [`setParameters()`](setParameters()). This could lead to setting `A` to zero through `setParameters()`, which would cause division by zero in [`getAPR()`](getAPR()).

Impact

Low. `IRM` will revert if `A` is accidentally set to zero.

Recommendation

Move the zero check for parameter `_A` into the `_setParameters` function:

```
     constructor(address _authority, uint256 _maxAPR, uint256 _minAPR, uint256
_A) AccessManaged(_authority) {
-        if (_A == 0) revert ErrorsLib.InvalidConstructorArgs();
-
         _setParameters(_maxAPR, _minAPR, _A);
     }
```

```
     function _setParameters(uint256 _maxAPR, uint256 _minAPR, uint256 _A)
internal {
         if (_maxAPR < _minAPR) revert IRM__InvalidAPR();
+        if (_A == 0) revert ErrorsLib.IRM__InvalidA();

         emit SetParameters(_maxAPR, _minAPR, _A);

         maxAPR = _maxAPR;
         minAPR = _minAPR;
         A = _A;
     }
```

Developer Response

Fixed in this [PR](PR)

2. Low - Potential DoS in `getTotalBalance()` due to unbounded loop

`getTotalBalance()` iterates over all PSMs without limit, potentially causing out-of-gas errors if too many PSMs are added.

Technical Details

`getTotalBalance()` iterates over all PSMs to sum their balances. Since each iteration requires an external call, this unbounded loop can reach block gas limit constraints. While this can be considered as an extreme scenario, the gas cost of all operations that rely on this function, including all users operations involving `accrueInterest()` will be impacted.

Impact

Medium. If the function becomes unusable due to gas limits:

- IRM can't calculate interest rates
- Interest accrual fails
- Entire protocol functionality breaks

Recommendation

Consider implementing a maximum limit on PSMs that can be added. The limit should be chosen considering:

- Gas costs and block limits
- Planned, supported assets
- Protocol scalability needs
- Economic viability of gas costs
- Future upgrade flexibility

Developer Response

Fixed in this PR

3. Low - Theoretical early-stage DoS attack via interest rounding in `getInterest()`

`accrueInterest()` calculates and mints interest based on the total PSMs total balance and a calculated (APR). In certain conditions, an attacker can block interest accrual by depositing one wei `osToken` every block (~12 seconds). In the early stages, when initial funds are low, users will not receive any interest. Over time, this can reduce user confidence in the protocol's value proposition and discourage further participation until the balance grows sufficiently to yield meaningful interest accrual.

Technical Details

Here is an example scenario. Given an APR of 15%, to ensure that the interest credited to users is at least 1 unit per block, the formula must satisfy:

```
interest = (balance * 0.15 * 12) / 31536000 ≥ 1
```

Solving for balance:

```
balance ≥ 31536000 / (0.15 * 12)
balance ≥ 17,520,000
```

In this scenario, if the total balance is below 17.5M, the interest accrued per block will be less than 1. Due to rounding down, this results in the `accrueInterest()` function effectively crediting zero interest to users each block.

An attacker can repeatedly deposit one wei to force interest calculation, ensuring no interest accrues until this threshold is met. However, given gas costs, maintaining this attack would be economically unviable.

Impact

Low. The protocol could be forced into a zero-interest state during the early stages. However, sustaining this condition would require continuous and costly block-by-block transactions. Gas fees make this prolonged attack economically unrealistic.

Recommendation

Avoid updating `lastUpdate` if interests are not accrued to prevent the attack.

```
    function accrueInterest() public {
        (, uint256 interest, uint256 timeDelta) = previewAccrual();

        if (timeDelta != 0) {
            emit AccruedInterest(timeDelta, interest);

            if (interest != 0) {
                 minter.mint(interest);
+                lastUpdate = block.timestamp;
             }
-
-            lastUpdate = block.timestamp;
         }
     }
```

After this change, you must also initialize `lastUpdate` in the constructor and update `lastUpdate` at the first deposit.

```
     function deposit(uint256 amount) internal virtual override returns (uint256)
{
-        accrueInterest();
+        if (totalSupply() == 0) {
+            lastUpdate = block.timestamp;
+        } else {
+            accrueInterest();
+        }
         return super.deposit(assets, receiver);
     }

     function mint(uint256 shares, address receiver) internal virtual override
returns (uint256) {
-        accrueInterest();
+        if (totalSupply() == 0) {
+            lastUpdate = block.timestamp;
+        } else {
+            accrueInterest();
+        }
         return super.mint(shares, receiver);
     }
```

Developer Response

Fixed in this PR

Gas Saving Findings

1. Gas - Cache `psm.length`

Technical Details

`removePSM()` read `psm.length` two times, which consumes extra gas due to the repeated `SLOAD` operation.

Impact

Gas savings.

Recommendation

Cache the array length in a local variable.

Developer Response

Fixed in this PR

Informational Findings

1. Informational - Remove unchecked increment of the counter

Technical Details

An unchecked increment of the counter is automatic after version 0.8.22. See docs.

Minter.sol#L170

SMLens.sol#L25-L27

Impact

Informational.

Recommendation

Remove unchecked increments to improve readability.

Developer Response

Fixed in this PR

2. Informational - `FEE_PRECISION` can be reduced from 1e18 to 10_000

Technical Details

The `PSM` uses `FEE_PRECISION` of 1e18, which is unnecessarily large for fee calculations. A value of 10_000 would be more intuitive and provide sufficient precision (0.01%), while preventing potential fee avoidance.

Impact

Informational.

Recommendation

Change `FEE_PRECISION` to 10_000.

Developer Response

Fixed in this PR

3. Informational - Fix typos

Technical Details

src/periphery/zapper/osUSDZapper.sol#L59-L75

```
-       uint256 usdsRecieved;
+       uint256 usdsReceived;

        if (tokenFrom == address(USDS)) {
-           usdsRecieved = amount;
+           usdsReceived = amount;
        } else if (tokenFrom == address(sUSDS)) {
-           usdsRecieved = sUSDS.redeem(amount, address(this), address(this));
+           usdsReceived = sUSDS.redeem(amount, address(this), address(this));
        } else if (tokenFrom == address(DAI)) {
-           usdsRecieved = zapInDaiToUSDS(amount);
+           usdsReceived = zapInDaiToUSDS(amount);
        } else if (tokenFrom == address(USDC)) {
-           usdsRecieved = zapInUsdcToUSDS(amount);
+           usdsReceived = zapInUsdcToUSDS(amount);
        } else {
            revert ErrorsLib.InvalidInputToken();
        }

-       USDS.forceApprove(USDSPSM, usdsRecieved);
-       uint256 amountReceived = IPSM(USDSPSM).buy(stake ? address(this) :
msg.sender, usdsRecieved);
```

```
+       USDS.forceApprove(USDSPSM, usdsReceived);
+       uint256 amountReceived = IPSM(USDSPSM).buy(stake ? address(this) :
msg.sender, usdsReceived);
```

Impact

Informational.

Recommendation

Fix typos.

Developer Response

Fixed in this PR

4. Informational - `psmBalance`'s NatSpec description is misleading

Technical Details

The NatSpec currently states that `psmBalance` represents "The balance of the PSM," implying a single `PSM`. In reality, `psmBalance` is the combined aggregated total of all relevant `PSM` balances. This discrepancy may confuse anyone reading the code or relying on the documentation to understand the function's mechanics.

Impact

Informational.

Recommendation

Update the variable name to `psmTotalBalance` and the NatSpec comment to accurately indicate that the variable refers to the cumulative balance across all PSMs, not just a single PSM.

Developer Response

Fixed in this PR

Final remarks

The OpenState protocol demonstrates a solid foundation for synthetic assets with an innovative profit-generating model through PSMs and yield strategies.

While the core mechanics are well implemented using battle-tested libraries, the complex interactions between modules and heavy reliance on off-chain decisions require careful consideration. Testing coverage could be improved, particularly for cross-module interactions and economic assumptions. The protocol would also benefit from better documentation detailing critical flows and accounting mechanisms. Despite these areas for improvement, the codebase provides a strong starting point for a decentralized synthetic asset platform.