Euler

EulerEarn

Security Assessment Report

Version: 2.1

July, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Euler components in scope. The review focused solely on the security aspects of the Solidity implementation of the contracts, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Euler components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Euler components in scope.

Overview

EulerEarn is a protocol for noncustodial risk management on top of accepted ERC-4626 vaults, especially the EVK vaults. It enables anyone to create a vault depositing liquidity into multiple ERC-4626 vaults.

Users of EulerEarn are liquidity providers who want to earn from borrowing interest without having to actively manage the risk of their position. The active management of the deposited assets is the responsibility of a set of different roles (owner, curator and allocators). These roles are primarily responsible for enabling and disabling accepted ERC-4626 strategy vaults and managing the allocation of users' funds.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the euler-xyz/euler-earn repository.

The scope of this time-boxed review was strictly limited to the following files, assessed at commit 558b15b:

1. `EulerEarn.sol`
2. `EulerEarnFactory.sol`
3. `PublicAllocator.sol`
4. `libraries/*`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 5 issues during this assessment. Categorised by their severity:

- Informational: 5 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Euler components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
EUL-01	Lost Assets Never Decrease On Strategy Recovery	Informational	Closed
EUL-02	Unbounded Memory Allocation	Informational	Closed
EUL-03	Inconsistent Zero Address Validation For Fee Recipient	Informational	Closed
EUL-04	Missing Fee Cap Validation In `PublicAllocator` Fee Setting	Informational	Closed
EUL-05	Miscellaneous General Comments	Informational	Closed

EUL-01	Lost Assets Never Decrease On Strategy Recovery
Asset	`EulerEarn.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `lostAssets` tracking mechanism in `_accruedFeeAndAssets()` creates a one-way ratchet that permanently inflates `totalAssets()` when strategies recover from losses. Once assets are marked as "lost", they can never be unmarked, leading to accounting discrepancies where the vault reports more assets than it actually controls.

The issue occurs in `_accruedFeeAndAssets()` where `lostAssets` can only increase or remain unchanged, but never decrease:

```
// If the vault lost some assets (realTotalAssets decreased), lostAssets is increased.
if (realTotalAssets < lastTotalAssets - lostAssets) newLostAssets = lastTotalAssets - realTotalAssets;
// If it did not, lostAssets stays the same.
else newLostAssets = lostAssets; // @audit lostAssets never decreases on recovery
```

Consider the following scenario:

1. A vault starts with 100 assets.
2. A strategy fails and `realTotalAssets` drops to 80, causing `lostAssets` to become 20.
3. The strategy recovers and `realTotalAssets` returns to 100
4. `lostAssets` remains at 20, making `totalAssets()` return 120 instead of 100 assets.

This creates phantom value that does not correspond to withdrawable assets.

Recommendations

Modify the `_accruedFeeAndAssets()` function to allow `lostAssets` to decrease when strategies recover. The logic should track the current deficit rather than cumulative historical losses.

Consider implementing a mechanism that reduces `lostAssets` proportionally when `realTotalAssets` exceeds the expected baseline (`lastTotalAssets - lostAssets`).

Resolution

The finding has been closed with the following comment provided by the development team, and acknowledged by the testing team as an intentional design decision:

"Firstly, if a strategy suffers a loss but subsequently its shares increase in value, it is impossible to determine whether the gain is due to loss recovery or new profits. From the perspective of a user who deposits into EulerEarn after a loss is realized, they would receive no yield on their deposit until the loss is fully recovered. This is not fair, as they did not deposit during the period when the loss occurred.

Secondly, and perhaps more importantly, if the EulerEarn vault incurs a loss that needs to be recovered, it may trigger a bank run. This is because, in addition to having unwithdrawable assets, depositors would not receive any yield on their deposits until the loss is covered. This could incentivize them to withdraw capital to avoid opportunity cost. While this could be partially mitigated by using only a portion of new profits to cover losses, such a design would still reduce the vault's attractiveness.

The intended method for recovering lost assets, as designed by the original authors, is through the burning of shares by the vault owner. The vault owner may deposit their earned fees to address(1), which can then be read by the frontend to adjust lostAssets."

EUL-02	Unbounded Memory Allocation
Asset	`PublicAllocator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `PublicAllocator.sol::reallocateTo()` function accepts an unbounded `Withdrawal[] calldata` array and allocates a dynamic memory array `(MarketAllocation[] memory allocations)` based on its length.

```
function reallocateTo(address vault, Withdrawal[] calldata withdrawals, IERC4626 supplyId) external payable {
        if (msg.value != fee[vault]) revert ErrorsLib.IncorrectFee();
        if (msg.value > 0) accruedFee[vault] += msg.value;

        if (withdrawals.length == 0) revert ErrorsLib.EmptyWithdrawals();

        if (!IEulerEarn(vault).config(supplyId).enabled) revert ErrorsLib.MarketNotEnabled(supplyId);

        MarketAllocation[] memory allocations = new MarketAllocation[](withdrawals.length + 1);     // @audit this should have an
      ↪  upper bound check
        uint128 totalWithdrawn;
        // ...
}
```

Since calldata size is ultimately limited by block gas limits, a sufficiently large array can:

1. Consume excessive gas during execution (especially during memory allocation or iteration).
2. Cause the transaction to run out of gas, failing execution.
3. Create a potential griefing vector where an attacker submits transactions that always revert due to hitting gas limits or exceeding internal gas constraints, locking functionality if used in critical control paths.

Recommendations

Implement an upped bound check on `withdrawals` before processing, e.g.:

```
uint256 MAX_WITHDRAWALS = 100; // or any safe upper limit
if (withdrawals.length > MAX_WITHDRAWALS) revert ErrorsLib.TooManyWithdrawals();
```

If processing a long list is a valid use case, implement batching or pagination and require stateful processing across multiple calls.

Resolution

The finding has been closed with the following comment provided by the development team, and acknowledged by the testing team as not having direct security implication:

"The griefing vector described would only result in a denial of service for the attacker themselves."

EUL-03	Inconsistent Zero Address Validation For Fee Recipient
Asset	`EulerEarn.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is an inconsistency in how the zero address validation is handled for the fee recipient across different functions, creating a potential vulnerability where fees could be minted to the zero address.

The `setFee()` and `setFeeRecipient()` functions prevent setting a non-zero fee with a zero address fee recipient:

```
// In setFee():
if (newFee != 0 && feeRecipient == address(0)) revert ErrorsLib.ZeroFeeRecipient();

// In setFeeRecipient():
if (newFeeRecipient == address(0) && fee != 0) revert ErrorsLib.ZeroFeeRecipient();
```

However, `_accrueInterest()` does not validate the fee recipient before minting:

```
lostAssets = newLostAssets;
emit EventsLib.UpdateLostAssets(newLostAssets);

if (feeShares != 0) _mint(feeRecipient, feeShares); // @audit: No zero address check
```

Recommendations

Add zero address validation in the `_accrueInterest()` function before minting fee shares.

Resolution

The finding has been closed by the testing team as a false positive. In the reviewed commit, the logic ensures that fee shares cannot be minted to the zero address.

EUL-04	Missing Fee Cap Validation In `PublicAllocator` Fee Setting
Asset	`PublicAllocator.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `setFee()` function in `PublicAllocator` lacks validation against the maximum fee limit defined in `ConstantsLib.MAX_FEE`, unlike the corresponding function in `EulerEarn`. This allows vault owners or admins to set arbitrarily high fees that could exceed reasonable limits.

The `PublicAllocator.setFee()` function only checks for duplicate values, but does not validate the fee amount:

```
function setFee(address vault, uint256 newFee) external onlyAdminOrVaultOwner(vault) {
    if (fee[vault] == newFee) revert ErrorsLib.AlreadySet();
    fee[vault] = newFee; // @audit No validation against MAX_FEE
    emit EventsLib.SetAllocationFee(_msgSender(), vault, newFee);
}
```

In contrast, `EulerEarn.setFee()` properly validates against the maximum:

```
function setFee(uint256 newFee) external onlyOwner {
    if (newFee == fee) revert ErrorsLib.AlreadySet();
    if (newFee > ConstantsLib.MAX_FEE) revert ErrorsLib.MaxFeeExceeded(); // @audit Proper validation
    // ...
}
```

Recommendations

Add validation in `PublicAllocator.setFee()` to ensure `newFee` does not exceed `ConstantsLib.MAX_FEE`, similar to the implementation in `EulerEarn.setFee()`.

Resolution

The finding has been closed with the following comment provided by the development team, and acknowledged by the testing team as an intentional design decision:

> *"The `EulerEarn` fee is defined as a percentage, whereas the `PublicAllocator` charges a fixed fee denominated in the network asset. Because the value of the network asset varies significantly across chains, it is not feasible to define a meaningful universal constant.*
>
> *Moreover, vault owners are free to install or remove `PublicAllocator` instances from their vaults, so it is in their own interest to set fees that do not impair the utility of the contract. If the fees are too high, users will simply avoid using the `PublicAllocator`, which is equivalent to it not being installed."*

EUL-05	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Naming Convention In Custom Error Library**

   Related Asset(s): libraries/ErrorLib.sol

   The current naming of certain custom errors in `ErrorsLib` may reduce code readability and hinder effective debugging:

   ```
   library ErrorsLib {
       /// @notice Thrown when the query is invalid.
       error BadQuery();
        ...
        /// @notice Thrown when there's no pending value to set.
       error NoPendingValue();
   ```

   While functionally correct, the naming lacks descriptive precision. In particular, the error names could be misinterpreted or require developers to reference comments or external documentation to fully understand their intent.

   Consider renaming these errors to more explicit and self-explanatory alternatives, e.g. `invalidQuery()` and `PendingValueNotSet()`.

2. **Typo in Guardian Documentation**

   Related Asset(s): IEulerEarn.sol

   The documentation for the `submitGuardian()` function contains a typo where *"gardian"* is used instead of *"guardian"* in the NatSpec comments:

   ```
   /// @notice Submits a `newGuardian`.
   /// @notice Warning: a malicious guardian could disrupt the Earn vault's operation, and would have the power to revoke
   /// any pending guardian.
   /// @dev In case there is no guardian, the gardian is set immediately.
   ```

   Correct the typo.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team has acknowledged the findings above and will address the typo.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].

σ'