ENIGMA DARK

Securing the Shadows



Security Review

Euler Earn

Rescue Strategy

November, 2025

Contents

1. Summary
2. Engagement Overview
3. Risk Classification
4. Vulnerability Summary
5. Findings
6. Annex
7. Disclaimer

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Euler Earn: Rescue Strategy
EulerEarn is a yield aggregation system built on ERC-4626 vaults that automatically allocates user deposits across approved strategies to generate returns. Management and risk controls are handled by designated roles that govern which strategies are enabled and how funds are distributed.
The Rescue Strategy is an emergency safeguard contract that halts user interactions and allows authorized roles to recover or stabilize assets when funds begin flowing into previously emptied EVaults, thereby preventing a potential bank run of Euler Earn underlying assets.

Engagement Overview

Over the course of 1.5 days beginning on November 8 2025, the Enigma Dark team conducted a security review of the Euler Earn: Rescue Strategy project. The review was performed by one Lead Security Researcher: vnmrtz.

The following repositories were reviewed at the specified commits:

Repository	Commit
euler-xyz/euler-earn	`f5bd06f...3103571`

The scope of the review covered the following files:

```
src
└── RescueStrategy.sol
```

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	2	2	0
Informational	0	0	0

Findings

Index	Issue Title	Status
L-01	Unsafe Reliance on tx.origin for Access Control	Fixed
L-02	Missing queue invariant enforcement during rescue	Fixed

Detailed Findings

High Risk

No issues found.

Medium Risk

No issues found.

Low Risk

L-01 - Unsafe Reliance on tx.origin for Access Control

Severity: Low Risk

Context: RescueStrategy.sol#L46-L49

Technical Details:
The `RescueStrategy` contract covers two main mechanics:

- Prevent a bank run of assets by the first withdrawers once funds start dripping into the empty underlying EVaults, since EulerEarn does not socialize losses.
- Ensure that the `rescueAccount` is the only entity able to recover all assets flowing into empty vaults, to later distribute them fairly to users.

To achieve this, the `RescueStrategy` currently relies on `tx.origin` to both block user actions mid-flow (preventing a bank run) and provide access control for the `rescueX` functions. The rationale for using `tx.origin` instead of the usual msg.sender check is to detect the original initiator of the transaction and block `EulerEarn` deposits and withdrawals by users mid-flow. Using `msg.sender` at strategy level can't identify the caller of the EulerEarn deposit and withdrawal actions.

However, relying on `tx.origin` is unsafe, as it always returns the original EOA (externally owned account) that started the transaction, even if the call passes through multiple smart contracts. This is different from msg.sender, which reliably gives the immediate caller of the current function at `deposit`.

Additionally, if a call were executed through a multisig instead of an EOA, the multisig's executor could directly invoke the protocol, bypassing intended restrictions. Depending on single EOAs for authorization is also a weak security practice, as they are vulnerable to operational risks such as key leaks, phishing attacks, and compromised wallets.

Recommendation:
Instead of relying on `tx.origin`, a `msg.sender` check for the `rescueX` functions and a temporary boolean flag to track when a rescue is active could be implemented:

1. Add a private flag bool `rescueActive` to track when a rescue is in progress.
2. In `rescueEuler`, `rescueEulerBatch`, and `rescueMorpho`, replace `onlyRescueAccount` with:

```
require(msg.sender == rescueAccount, "unauthorized");
require(!rescueActive, "rescue ongoing");
```

3. Inside each rescueX function, set rescueActive = true at the start and reset it to false after processing the flashloan.
4. In the callback functions (onFlashLoan, onBatchLoan, onMorphoFlashLoan), add:

```
require(rescueActive, "rescue inactive")`
```

5. For deposit (and any other helpers that used `onlyRescueAccount`), just require rescueActive instead (like above). This ensures that deposits only function while a rescue is active.

Developer Response:
Fixed in commit: `d1b0ddf`

L-02 - Missing queue invariant enforcement during rescue

Severity: Low Risk

Context: RescueStrategy.sol#L188-L203

Technical Details:
The function `_processFlashLoan` doesn't check that the rescue strategy is the only element in the supply queue and first in the withdraw queue, two direct requirements for the flashloan flow to work as expected.

While this doesn't create a direct vulnerability, it could allow the system to enter edge-case states if queues are misconfigured during a rescue operation. For example a curator forgetting to set the `RescueStrategy` first on the withdrawal queue before executing a rescue.

Recommendation:
Consider implementing the following checks at the start of the `rescueX` functions in order to verify the correct configuration of the queues before the rescue is executed:

```
require(vault.supplyQueueLength() == 1, "supply queue length is not 1");
require(address(vault.supplyQueue(0)) == address(this), "supply queue first
element is not the rescue strategy");
require(address(vault.withdrawQueue(0)) == address(this), "withdraw queue
first element is not the rescue strategy");
```

Developer Response:
Fixed in commit: `180c1e8`

Informational

No issues found.

Annex

Annex A: Rescue Strategy - Euler Batch Flashloan Debt Dust Analysis

Context:
While reviewing the alternative flashloan source `RescueStrategy:rescueEulerBatch`, we aimed to confirm that no dust debt remains after borrowing and repaying funds within an EVC deferred-checks batch executed in the same block, as any residual debt would cause the EVC checks to revert the rescue operation.

According to the Euler EVK debt tracking mechanics, debt rounding can only occur if time elapses between the borrow and the repay operations. Debt is denominated in assets 1:1 and internal precision adjustments only apply after time passes, a borrow and repay executed within the same transaction should match exactly, resulting in no rounding and no residual dust.

To confirm this, we extended the fuzzing suite developed for the EVK project to include a test case where a borrow and repay are performed in an EVC batch, imulating a borrow-repay sequence within a single block across several million fuzzing iterations:

BM_INVARIANT_P1: a user without debt should be able to borrow(x) => repay(x) on the same evc batch without reverting

```
function assert_BM_INVARIANT_P1(uint256 amount) external setup {
    address flashLoanVault = address(eTST);

    require(eTST.debtOf(address(actor)) == 0, "BM_INVARIANT_P1: user debt is
not 0");

    IEVC.BatchItem[] memory batchItems = new IEVC.BatchItem[](3);
    batchItems[0] = IEVC.BatchItem({
        targetContract: address(evc),
        onBehalfOfAccount: address(0),
        value: 0,
        data: abi.encodeCall(IEVC.enableController, (address(actor),
flashLoanVault))
    });
    batchItems[1] = IEVC.BatchItem({
        targetContract: flashLoanVault,
        onBehalfOfAccount: address(actor),
        value: 0,
        data: abi.encodeCall(IBorrowing.borrow, (amount, address(actor)))
    });
    batchItems[2] = IEVC.BatchItem({
        targetContract: flashLoanVault,
        onBehalfOfAccount: address(actor),
        value: 0,
        data: abi.encodeCall(IBorrowing.repay, (amount, address(actor)))
    });

    vm.prank(address(actor));
    try evc.batch(batchItems) {}
    catch {
        assertFalse(true, BM_INVARIANT_P1);
    }

    assertEq(eTST.debtOf(address(actor)), 0, BM_INVARIANT_P1);
}
```

After running the fuzzing suite for over 20 million iterations, we confirmed that no dust debt remains when borrowing and repaying funds within the same EVC deferred-checks batch, ensuring the rescue functions execute without unintended reverts.

Disclaimer

This report does not endorse or critique any specific project or team. It does not assess the economic value or viability of any product or asset developed by parties engaging Enigma Dark for security assessments. We do not provide warranties regarding the bug-free nature of analyzed technology or make judgments on its business model, proprietors, or legal compliance.

This report is not intended for investment decisions or project participation guidance. Enigma Dark aims to improve code quality and mitigate risks associated with blockchain technology and cryptographic tokens through rigorous assessments.

Blockchain technology and cryptographic assets inherently involve significant risks. Each entity is responsible for conducting their own due diligence and maintaining security measures. Our assessments aim to reduce vulnerabilities but do not guarantee the security or functionality of the technologies analyzed.

This security engagement does not guarantee against a hack. It is a review of the codebase during a specific period of time. Enigma Dark makes no warranties regarding the security of the code and does not warrant that the code is free from defects. By deploying or using the code, the project and users of the contracts agree to use the code at their own risk. Any modifications to the code will require a new security review.