THREE SIGMA

Keyring

Security Review

Disclaimer

Security Review

Keyring

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Security Review

Keyring

Table of Contents

Summary

Security Review

Keyring

Summary

Three Sigma audited Keyring in a 0.4 person week engagement. The audit was conducted on 26/06/2025.

Protocol Description

The KeyringLeveragedPositionUnwindingOperator is a specialized contract that atomically unwinds leveraged positions on the Euler xUSDC-USDC cluster using flash loans from the Multipli vault. When called, it receives flash-loaned USDC, uses it to repay outstanding debt on the Euler platform, withdraws the xUSDC collateral from the leveraged position, and returns the specified amount of xUSDC shares to the Multipli vault to complete the flash loan. The entire unwinding process is executed in a single atomic transaction using Euler's EVC batching functionality, ensuring the leveraged position (created by repeatedly using USDC to mint xUSDC collateral and borrowing more USDC) is safely and efficiently closed.

Scope

Security Review

Keyring

Scope

Filepath	nSLOC
euler-xusdc-usdc-looping/src/KeyringLeveragedPositionUnwindingOperator.sol	121
euler-xusdc-usdc-looping/src/utils/Utils.sol	28
Total	**149**

Methodology

Security Review

Keyring

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Keyring

Project Dashboard

Application Summary

Name	Keyring
Repository	https://github.com/Keyring-Network/keyring-smart-contracts-utils
Commit	6af41df
Language	Solidity
Platform	Avalanche

Engagement Summary

Timeline	26/06/2025
№ of Auditors	2
Review Time	0.4 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	3	3	0
Medium	0	0	0
Low	1	1	0
None	1	1	0

Category Breakdown

Suggestion	1
Documentation	0
Bug	0
Optimization	0
Good Code Practices	0

Risk Section
Security Review
Keyring

Risk Section

- **Dependency on External Contracts**: The system relies on external contracts. Although integrations were reviewed, vulnerabilities or issues within these external systems could still pose risks.

Findings

Security Review

Keyring

Findings

3S-Keyring-H01

maxWithdraw returns 0 when controllers are enabled, causing **withdrawAll** functionality to fail

Id	3S-Keyring-H01
Classification	High
Impact	High
Likelihood	High
Status	Addressed in #632cbb7.

Description

The **KeyringLeveragedPositionUnwindingOperator** contract contains an issue in its **withdrawAll** functionality that renders this feature completely ineffective for users who have enabled controllers on their vault positions. This issue stems from the fundamental design of Euler vaults where the **maxWithdraw** function returns zero when any controller is enabled on an account.

In the **onRedemptionFlashLoan** function at line 91, the contract attempts to determine the maximum withdrawable amount by calling **i_xUsdcVault.maxWithdraw(_initiator)**. However, this call will always return zero when the user has enabled a controller, which is a requirement when using a vault as collateral for borrowing from another vault.

The root cause lies in Euler's vault implementation where the **maxWithdraw()** function internally calls **maxRedeemInternal()** which contains a critical check:

```
function maxRedeemInternal(VaultCache memory vaultCache, address owner)
private view returns (Shares) {
    Shares max = vaultStorage.users[owner].getBalance();

    if (max.isZero() || hasAnyControllerEnabled(owner)) return Shares.wrap(0);

    Shares cash = vaultCache.cash.toSharesDown(vaultCache);
    max = max > cash ? cash : max;

    return max;
```

```
}
```

When a user enables a controller (which is necessary for using a vault as collateral), the **hasAnyControllerEnabled(owner)** check returns true, causing the function to return zero shares regardless of the actual balance. This behavior is documented in Euler's code comments which state that "integrators who handle borrowing should implement custom logic to work with the particular controllers they want to support."

The contract's current implementation does not account for this limitation, leading to a situation where the **withdrawAll** flag will always result in withdrawing zero shares from the vault. This creates a broken user experience where the contract appears to execute successfully but actually performs no withdrawal operation.

The likelihood of this bug is high because leveraged position unwinding typically requires the user to have enabled controllers for their borrowing positions. When users attempt to fully unwind their positions using the **withdrawAll** functionality.

Recommendation

To fix this issue, the contract must implement a proper sequence of operations that handles the controller-enabled state before attempting to determine the maximum withdrawable amount. The solution requires first repaying all outstanding debt and then disabling the controller before calling **maxWithdraw**.

Since we cannot know the maximum withdrawable amount before performing the repay operation (as the controller must be disabled first), when **withdrawAll** is requested, the contract should use **batchSimulate** to determine the correct amount to withdraw after the debt repayment and controller disabling operations. The result from this simulation can then be used to construct the proper batch sequence with the accurate withdrawable amount.

3S-Keyring-H02

Incorrect USDC approval will cause contract to permanently fail

Id	3S-Keyring-H02
Classification	High
Impact	High
Likelihood	High
Status	Addressed in #632cbb7.

Description

The **KeyringLeveragedPositionUnwindingOperator** contract has a flaw in line 97 where USDC approval is made to the wrong address, causing the contract to permanently fail when attempting to repay debt to the Euler USDC vault.

```
i_usdc.approve(address(i_usdc), _underlyingAmount + buffer);
```

The contract is approving USDC spending to **address(i_usdc)** (the USDC token contract itself), but the actual entity that needs to transfer USDC from the contract is the **USDC vault** (**i_usdcVault**) when executing the **repay** operation.

When the Euler vault's **repay** function is called via the EVC batch operation, the vault will attempt to **transferFrom** the USDC from the contract to repay the debt. However, the contract has only approved the USDC token contract itself, not the vault that needs to perform the transfer. This leads the contract logic to be broken.

The issue is evident from the Euler vault's **repay** function implementation:

```
function repay(uint256 amount, address receiver) public virtual nonReentrant returns (uint256) {
    (VaultCache memory vaultCache, address account) = initOperation(OP_REPAY, CHECKACCOUNT_NONE);

    uint256 owed = getCurrentOwed(vaultCache, receiver).toAssetsUp().toUint();

    Assets assets = (amount == type(uint256).max ? owed : amount).toAssets();
    if (assets.isZero()) return 0;

    pullAssets(vaultCache, account, assets);
```

```
    decreaseBorrow(vaultCache, receiver, assets);
    return assets.toUint();
}
function pullAssets(VaultCache memory vaultCache, address from, Assets amount)
internal virtual {
        vaultCache.asset.safeTransferFrom(from, address(this), amount.toUint(),
permit2);
        vaultStorage.cash = vaultCache.cash = vaultCache.cash + amount;
}
```

Recommendation

It is advised to update the approved address to the correct spender.

```
// Line 97: from
i_usdc.approve(address(i_usdc), _underlyingAmount + buffer);
// To
i_usdc.approve(address(i_usdcVault), _underlyingAmount + buffer);
```

3S-Keyring-H03

i_multipliVault not initialized in constructor breaks contract functionality

Id	3S-Keyring-H03
Classification	High
Impact	High
Likelihood	High
Status	Addressed in #632cbb7.

Description

In the **KeyringLeveragedPositionUnwindingOperator** contract, the **i_multipliVault** immutable variable is declared but never initialized in the constructor. This causes the **onRedemptionFlashLoan** function to always revert, making the entire contract non-functional.

In the constructor at lines 47-56, the contract initializes several immutable variables:

```
constructor(address _evc, address _usdcVault, address _xUsdcVault) {
    _requireNotAddressZero(_evc);
    _requireNotAddressZero(_usdcVault);
    _requireNotAddressZero(_xUsdcVault);
    i_evc = IEthereumVaultConnector(payable(_evc));
    i_usdcVault = IEVault(_usdcVault);
    i_xUsdcVault = IEVault(_xUsdcVault);
    // @audit - multipli vault is not set
    i_usdc = IERC20(i_usdcVault.asset());
    i_xUsdc = IERC4626(i_xUsdcVault.asset());
}
```

However, the **i_multipliVault** immutable variable declared at line 35 is never assigned a value:

```
/// @notice The Multipli vault where the flash loan (swap) is initiated.
address public immutable i_multipliVault;
```

This causes the **_requireOnlyMultipliVault** function at line 130 to always revert:

```
function _requireOnlyMultipliVault(address _caller) internal view {
    if (_caller != i_multipliVault) revert
KeyringLeveragedPositionUnwindingOperator__NotMultipliVault();
}
```

Since **i_multipliVault** is uninitialized, it defaults to **address(0)**, and any caller (including the legitimate Multipli vault) will not match this address, causing the function to always revert.

Recommendation

Add the Multipli vault address as a constructor parameter and initialize the **i_multipliVault** variable while also adding a **addreess(0)** check.

3S-Keyring-L01

Faulty **_requireGreaterThan** function logic

Id	3S-Keyring-L01
Classification	Low
Impact	Low
Likelihood	Medium
Status	Addressed in #632cbb7.

Description

The **Utils** contract contains a logic flaw in the **_requireGreaterThan** function that renders this validation ineffective.

In the **_requireGreaterThan** function at lines 29-33 of **Utils.sol**, the contract attempts to validate that **_value** is greater than **_reference**:

```
function _requireGreaterThan(uint256 _value, uint256 _reference) internal pure {
    if (_value < _reference) revert Utils__NotGreaterThan();
}
```

The fundamental issue with this implementation is that it only reverts when **_value < _reference**, but it should revert when **_value <= _reference** to properly enforce the "greater than" condition. The current logic allows the function to pass when **_value == _reference**, which violates the intended "greater than" requirement.

This broken validation is used in the **onRedemptionFlashLoan** function at line 85 where it calls **_requireGreaterThan(_shares, 0)**. Due to the inverted logic, this check will only revert if **_shares** is less than 0, which is impossible for a **uint256** type. This means the validation effectively does nothing and allows the function to proceed even when **_shares** is zero, which could lead to unexpected behavior in the flash loan redemption process.

Recommendation

In order fix this issue, the **_requireGreaterThan** function must be corrected to properly enforce the "greater than" condition. The function should revert when the value is less than or equal to the reference value:

```
function _requireGreaterThan(uint256 _value, uint256 _reference) internal pure {
    if (_value <= _reference) revert Utils__NotGreaterThan();
}
```

3S-Keyring-N01

Redundant **xUsdc** sweep operation

Id	3S-Keyring-N01
Classification	None
Category	Suggestion
Status	Addressed in #632cbb7.

Description

The **KeyringLeveragedPositionUnwindingOperator::onRedemptionFlashLoan** function performs leveraged position unwinding by executing a batch of operations through the Ethereum Vault Connector (EVC). At the end of the function, it calls **_sweep(address(i_xUsdc), _initiator)** to transfer any remaining **xUsdc** tokens from the contract to the initiator. However, this sweep operation is unnecessary because **xUsdc** tokens never flow through this contract during the unwinding process. The batch operations withdraw **xUsdc** directly to the EVC and then transfer the required amounts to the Multipli vault without the tokens ever being held by this contract.

Recommendation

Remove the unnecessary sweep operation for **xUsdc** tokens to improve code quality and gas efficiency.

_sweep(address(i_usdc), _initiator);
- _sweep(address(i_xUsdc), _initiator);