certora

Security Analysis and Formal Verification Report

Euler EVC

December 2023

Prepared for
Euler

certora

Table of content

Table of Contents **2**

Project Summary **3**

Project Team 3

Project Timeline 4

Findings Summary 4

Detailed Findings **5**

EU-L-01 -Contracts called by vaults can DOS status checks 5

EU-INFO-01- An if/else if statement can be simplified 7

EU-INFO-02-constant STAMP_MASK is unused 8

Formal Verification **9**

Verification Notations 9

General Assumptions 9

Properties 9

About Certora **12**

Disclaimer **12**

certora

Project Summary

This document describes the specification and verification of the new **EthereumVaultConnector** contract from the Euler project using the Certora Prover and manual code review findings. The work was undertaken from 22.11.2023 to 14.12.2023.

The Certora Prover demonstrated that the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. The formal rules are not a full specification, but only a selection of interesting property used as a demonstration project. In addition, the team performed a manual audit of the Solidity contract.

Project Team

Name	Role
Andrew Ferraiuolo	Project Lead
Tomer Ganor, Dravee	Security researcher
Andrew Ferraiuolo, Gereon, Johannes Spath	Formal Verification expert

Project Timeline

Event	Timeline
Kickoff Meeting	Nov 22, 2023
Update Meeting #1	Nov 30, 2023
Update Meeting #2	Dec 7, 2023
Draft Report Delivery	Dec 14, 2023
Final Report Delivery	Dec 28, 2023

Findings Summary

Severity	Discovered	Acknowledged	Fixed
Critical	0	-	-
High	0	-	-
Medium	0	-	-
Low	1	-	-
Informational	2	2	2
Total	**3**	**2**	**2**

Detailed Findings

1.

Contracts called by vaults can DOS status checks		
Low	Probability: **Rare**	
Category: DOS Liquidation	Files: EthereumVaultConnector.sol	ID: EU-L-01

Description

It's expected that:

- All funds are under control of the enabled Controller Vaults
- Checks are deferrable towards the end of the execution's state

However, there's a particular scenario that can occur when, mid-execution, an arbitrary contract is called (whether directly, such as with a `.call()`, or with a callback, such as with a `safeMint()`, an `ERC777` token or ERC20 Plugins). As a reminder, the spec mentions the following:

`NOTE: The protocol deliberately doesn't enforce specific properties about the assets being used as collateral or liabilities. EVC users can therefore create vaults backed by irregular asset classes, such as NFTs, uncollateralized IOUs, or synthetics.`

When this happens, the called contract can gain control of the execution flow and do one of the following (as these functions are **permissionless** and **callable by anyone**):

- Either call `requireVaultStatusCheck` and insert themselves as an **arbitrary vault** to be checked. As a reminder, they **cannot be forgiven** by the controller (only a vault can add/forgive themselves).
- Or call `requireAccountStatusCheck()` with an **arbitrary account**, which will make`forgiveAccountStatusCheck` **revert** due to the modifier `onlyController(account)` not passing the check that the arbitrarily added account's controller would be the current controller.

In both cases, the transaction will always end up **reverting at the end** when the checks are being made.

Exploit Scenario

As a scenario, let's imagine that the **controller is trying to liquidate** a user in violation in a **trusted and non-malicious vault**.

As a courtesy, sometimes, as part of the vault's flow, there's a callback to liquidated users that can occur (as part of a `try/catch`, otherwise the user could just revert on callback to prevent the liquidation).

In our particular situation, it's possible for that user to either call `requireVaultStatusCheck()` or `requireAccountStatusCheck()` so that the transaction would end up reverting at the top-level of the execution flow, during the checks.

Such vaults making those kinds of callbacks (`safeMint()`, `ERC777`, `ERC1820` etc.) or direct calls (`.call()`) to the liquidated users would be working fine on their own and would be successfully preventing any DOS attempt from the user the liquidation. However, down the execution flow, with the EVC, the **liquidation can be prevented**.

Recommendation

controllers need to be very careful with the vault they rely on. Even if the vault is safe by itself, the external call from the vault (in a safe way) can cause denial of liquidation.

Customer response

Consider the fact that the EVC is a special-purpose multicall contract and, ultimately, it's the user that defines which contracts are being called in the transaction. Therefore, reverting the transaction due to a call to a contract that unexpectedly schedules the account/vault status checks (which are either a result of safeMint()/ERC777 callback or ERC20 Plugin) is no different

than reverting the transaction due to a call of a function that unconditionally reverts. It's the user being in control.

Please note that in most systems, allowing a user to install any kind of a callback, i.e. on transfer, such a callback is only called if a user is a party relevant to the operation. I.e. in case of a callback on transfer, the callback can only occur if the user's address is either from or to. Hence, considering the collateral vault itself is not malicious, there's no risk to the liquidation flow as described:

- in case the impersonate performs shares transfer, there are no callback side effects possible unless the collateral vault implements the callback system on its shares transfer. If that's the case, it should be scrutinized whether such a vault can become collateral for a controller before recognizing it as such.

- in case the impersonate performs withdrawal of the underlying or shares redemption (which in either case means transfer of the underlying asset), even if the callback system is in place on the underlying asset contract, the from and to addresses are not user-defined. In liquidation, the from address is always a collateral vault which is considered non-malicious if recognized as collateral. The to address is either the controller itself or the liquidator. Neither of those two has any incentive to enable a malicious callback in order to prevent liquidation.

2.

An if/else if statement can be simplified

Severity: **Informational**	Probability:	
Category: Gas Optimization	Files: EthereumVaultConnector.sol	ID: EU-INFO-01

Description

Change those lines from:

```
714:                 if (value > 0 && value != type(uint256).max && value > address(this).balance) {
715:                     revert EVC_InvalidValue();
716:                 } else if (value == type(uint256).max) {
```

```
717:             value = address(this).balance;
718:         }
```

To:

```
        if (value == type(uint256).max) {
             value = address(this).balance;
         } else if (value > address(this).balance) {
             revert EVC_InvalidValue();
        }
```

Indeed: `value > 0 && value > address(this).balance` is a tautology as `address(this).balance >= 0`.
Also, first checking `if (value == type(uint256).max)` will implicitly make the `else if` statement verify `value != type(uint256).max`.

Customer response

Acknowledged and fixed (code fix)

3.

constant STAMP_MASK is unused

Severity: **Informational**	Probability: **-**	
Category: Unused variables	Files: EthereumVaultConnector.sol	ID: EU-INFO-02

Description

This constant can be removed as it isn't used.

```
uint256 internal constant STAMP_MASK = 0xFFFFFFFFFFFFFF0000000000000000000000000000000000000000;
```

Customer response

Acknowledged. Used or not, the `STAMP_MASK` constant was defined for consistency so that the full word is covered.

certora

Formal Verification

Verification Notations

Formally Verified
Violated

General Assumptions

1. Loop unrolling: We assume any loop can have at most 3 iterations

Properties

Rule#	Rule Name	Description	Link to rule report
1	Verified **topLevelFunctionDontChangeTransientStorage**	All the storage variables declared in the TransientStorage contract must return to the default value after the top-level EVC call	*opLevelFunctionDontChangeTransientStorage*
2	Verified **noFunctionChangesExecutionContext**	Each external call that the EVC performs, restores the value of the execution context so that it's equal to the value just before the external call was performed	*noFunctionChangesExecutionContext*

3	Verified ***onlyEVCCanCallCriticalMethod***	EVC can only be msg.sender during the self-call in the permit() function. Verifies we can not call into EVC with msg.sender == EVC unless via permit(). We do that by verifying all of EVM's call opcodes: a. we verify that CALLCODE is never used. b. we verify that CALL is never used with EVC as the target contract. c. we verify that DELEGATECALL is only used with EVC as the target contract so that we never leak write access to our storage to external code (which could then DELEGATECALL back into EVC, violating the property). d. we ignore STATICCALL because such calls are read-only.	*onlyEVCCanCallCriticalMethod*
4	Verified **check_have_commonPrefix**	Two accounts have a common owner according to haveCommonOwner exactly if their address prefix is identical	*check_have_commonPrefix*
5	Verified **check_callDepth_zero_means_checksAreDeferred**	Checks are deferred exactly if the call depth is not zero	*check_callDepth_zero_means_checksAreDeferred*
6	Verified **onlyOwnerCanCallSetOperator**	Only the owner of an address prefix can call setOperator	*onlyOwnerCanCallSetOperator*

7	Verified **onlyOneController**	Each Account can have at most one Controller Vault enabled at a time unless it's a transient state during a Checks-deferrable Call	*onlyOneController*
8	Verified **nonRevertFunctions & mustRevertFunctions**	Only batchSimulation and batchRevert always revert, all other external functions have non-reverting paths	nonRevertFunctions & mustRevertFunctions
9	Verified **setAccountOperatorSandboxed**	Calling setAccountOperator does not affect the state for any operator other than the target of the function call	setAccountOperatorSandboxed
10	Verified **Check_call_depth** **Check_bitmasks_coverall** **Check_call_depth_maximum** **Check_on_behalf_of_account** **envfreeFuncsStaticCheck** **Check_bitmasks_offsets** **check_bitmasks_disjoint**	Sanity checks about the ExecutionContext	several rules
11	Verified **envfreeFuncsStaticCheck** **Not_contained_if_removed** **Removed_iff_not_contained** **Removed_then_length_decrease** **mirrorIsCorrect** **Contained_if_inserted** **containsIntegrity** **validSet**	Sanity checks about Set data structure	several rules

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.
Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.