PUBLIC

Code Assessment

of the Ethereum Vault Connector Smart Contracts

6 May, 2024

Produced for

by

Contents

1 Executive Summary

Dear Euler team,

Thank you for trusting us to help Euler with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Ethereum Vault Connector according to Scope to support you in forming an opinion on their security risks.

Euler implements Ethereum Vault Connector, a general framework for vaults interoperability for the purpose of arbitrary lending markets creation.

The critical subjects covered in our audit are authentication, checks enforcement, and adherence to the specification. Security regarding all the aforementioned subjects is high.

Some issues of low severity have been addressed by Euler by accepting them as part of the specification and improving the documentation.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed.

We have furthermore included notes on peculiar aspects of the systems, that will hopefully be of interest to integrators and future users of Ethereum Vault Connector.

We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3
• Risk Accepted	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Ethereum Vault Connector repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	7 April 2024	c30606b44f1d1cabba303dfd13046e5444cab775	Initial Version
2	22 April	9bc0c17afd6bed51cd9126d9f3b8fc31614d29a9	After Intermediate Report

For the solidity smart contracts, the compiler version `0.8.24` was chosen.

2.1.1 Included in scope

This report covers the Euler Ethereum Vault Connector (EVC) contracts.

- src/ExecutionContext.sol
- src/EthereumVaultConnector.sol
- src/TransientStorage.sol
- src/Errors.sol
- src/Events.sol
- src/Set.sol
- src/utils/EVCUtil.sol

2.1.2 Excluded from scope

Any contracts inside the repository that are not mentioned in `Scope` are not part of this assessment.

Tests and deployment scripts are excluded from the scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Euler offers Ethereum Vault Connector (EVC), a system that enables coordination between vaults to build composable and arbitrary lending markets. The EVC serves as a foundation for vaults to interact with each other in a unified manner. The EVC acts as a central registry of which collaterals are enabled

for a given user, and which lending vault has the power to act on a user's collateral in case of need. The EVC allows batching Vault operations and is tasked to enforce the solvency of lenders.

2.2.1 EthereumVaultConnector

2.2.1.1 Controller

The main functionality provided by the EVC is to enable borrowing from vaults, by allowing vaults to interact with the borrower's collateral, which is provided in the form of shares of EVC-compatible vaults. When an account takes a loan from a vault, this vault has to be set as the controller of the account, and it will gain control over the collateral until the vault itself renounces the controller role (in general when the loan is fully repaid). At any time, the controller vault can invoke `controlCollateral()` to move the account's collateral. Typically this is used to transfer the collateral to the liquidator during the liquidation of the borrow position.

2.2.1.2 Collateral management

An account owner or operator can `enableCollateral()` and `disableCollateral()` for it. A maximum of 10 different collaterals can be enabled for an account. Furthermore, the account owner or operator can `reorderCollaterals()` to optimize collateral checks performed by controllers.

2.2.1.3 `call`, `batch` *and* `permit`

The EVC allows any user to `call()` into any `targetContract` with arbitrary `data`. A `call()` can also be performed `onBehalfOfAccount` which requires additional authentication to be performed as only account owners or assigned operators can perform calls for an account. If `targetContract` is the `msg.sender`, no authentication is performed as the contract is supposed to trust itself. Similarly, `batch()` allows for multiple `calls` to be executed one after another.

A `signer` can sign arbitrary `data`, only allowing `sender` to invoke `permit()` with the signed data, which will verify the validity of the signature and ensure that the contents of the signature are valid. The signature contains a `deadline` after which it is no longer valid and a `nonce` to prevent the signature from being replayed. The signature can either be an ECDSA signature (used by EOAs), in which case it is verified using `ecerecover()`, or an ERC1271 signature (used by smart contracts) that is verified by calling `signer.isValidSignature()`. Once the validity of the signature has been assessed, a self-call into the EVC is performed with the arbitrary data. Furthermore, the execution context is modified to set `onBehalfOfAccount` to `signer`. During a self-call into the EVC the `_msgSender` is set to the `onBehalfOfAccount` address stored in the execution context instead of being `msg.sender`. This allows the EVC to act on behalf of the `signer`. This kind of self-call is detected because the `msg.sender` is the EVC itself, and it can happen only through the `permit` function. Calls in `batch` or `call` that target the EVC are performed through `delegatecall` and maintain the original `msg.sender`.

2.2.1.4 Checks

The EVC is tasked to enforce the solvency of the lending markets that are implemented through it by performing checks on user position every time a critical action is performed. For example, a user who borrows through the EVC does not transfer its collateral to a lending pool, instead the balance is maintained with the user and whenever an outgoing transfer is performed, the collateral contract is tasked to request a check on the solvency of the sending user. The solvency check is requested through the EVC's `requireAccountStatusCheck` function.

Checks can be performed immediately, in case no ongoing EVC execution is running, or they can be deferred, meaning they are performed after batches of actions have been performed. This allows a generalized flash-loan logic, enabling simpler and more powerful position management.

As the last action of a `call()` or a `batch()`, checks are performed on the accounts and the vaults that have been involved in the batch. While checks are performed, no call can re-enter the EVC. The checks are first performed on the accounts and then on the vaults. An account check is performed by delegating

the check to its controller, if there is any. The controller ensures that the account is not under-collateralized. Checks can be required explicitly by systems that rely on the EVC by calling `requireAccountStatusCheck()` or `requireVaultStatusCheck()`. Anyone can request a check for an account, and vaults can request checks for themselves. Checks are also added when using EVC functionality that modifies a user position, such as `disableCollateral()`.

Calls, batches and permits can be nested in which case the checks are all deferred to the end of the outermost call. This is achieved by setting the `checksDeferred` flag in the execution context. Vault status checks are performed by calling `checkVaultStatus()` on the vaults. The vault status check ensures that the vaults are not exceeding global limits (such as supply limits) after the actions have been performed.

Controllers also have the ability to `forgiveAccountStatusCheck()` for accounts they control. This is used to allow accounts to still be in an under-collateralized state following a partial liquidation, as long as the overall health of the system is improved. If EVC checks are forgiven, the controller vault must implement its custom checking logic.

2.2.1.5 Accounts

Every Ethereum address that interacts with the EVC owns 256 accounts. Every account range (0-255) has an owner assigned to it stored in the `ownerLookup` mapping. Each account's address is computed from the first 19 bytes of the user's address and the first byte of the account id. The owner of an account is the owner of the 19-byte address prefix. This allows for up to 256 borrow positions per user. In the unlikely case that two users share the same address prefix, the first one to have interacted with the EVC and be authenticated by it will become the owner of the `addressPrefix`. The owner of an address prefix is saved the first time the owner is authenticated during a `call` or a `batch` in the EVC.

The owner of an account can set operators for it. Operators can act on behalf of the owner on the account. Operators cannot add other operators to accounts they manage but can renounce their operator role. Accounts can have multiple operators. Accounts cannot be operators for other accounts in the same address prefix.

2.2.1.6 Nonces

Every `addressPrefix` owns 2**256 `nonceNamespaces` each initialized to a nonce of 0. A nonce in a given `nonceNamespace` is used by a user in a signed `permit` message to prevent replay attacks by ensuring that the message is only valid once. The nonce is incremented after the message has been processed. The nonce is stored in the `nonceNamespaces` mapping. To invalidate a nonce, the user can `setNonce` to a value greater than the current nonce. To invalidate a namespace its nonce can be set to `type(uint256).max`. Nonces for every `nonceNamespace` for every `addressPrefix` are stored in the `nonceLookup` mapping.

2.2.1.7 LockdownMode & PermitDisableMode

Lockdown mode can only be enabled and disabled by the owner. It will prevent any call with authentication from being performed that is not from the owner. Lockdown mode is enforced during authentication of the caller in `callWithAuthenticationInternal()`.

`PermitDisableMode` can only be enabled and disabled by the owner. It will prevent any permit action from being performed. It can be used in case the owner signs by mistake a permit action that should not be executed. However, this is not enough to prevent the permit action from never being executed. The owner should then invalidate the nonce used in the malicious permit action. It is only enforced during the execution of a permit action in `permit()`.

Both modes cannot be disabled during a self-call of `permit()` or during `call` or `batch` actions. This is enforced by checking that the execution context `areChecksDeferred` flag is not set.

2.2.1.8 Simulation

The EVC allows users to simulate their batches using `batchSimulation()` and `batchRevert()`. A simulation will always revert to the initial state and yield the results of each batch item operation and the result of all account and vault checks that have been performed for the batch to the user. This allows users to test the outcome of their batch before executing it.

2.2.2 EVCUtil

`EVCutil` is an abstract contract that provides utility functions required by vaults to interact with the EVC. It must be initialized with the address of the EVC contract. It provides two modifiers: `callThroughEVC` which will redirect the function call through the EVC and `onlyEVCWithChecksInProgress` which checks that the EVC is the sender and that checks are in progress. Furthermore, it provides a `_msgSender()` function that either returns `msg.sender` or the `OnBehalfOfAccount` stored in the EVC's execution context if the sender is the EVC itself. Last, `_msgSenderForBorrow()` is a function similar to `_msgSender()` but it additionally ensures that the sender account has the vault specified as argument as one of its controllers.

2.2.3 ExecutionContext

`ExecutionContext` is a library that provides functions for managing the EVC's execution context (EC). The EC is represented as a `uint256` and holds the following information: `onBehalfOfAccount`, `checksDeferred`, `checksInProgress`, `controlCollateralInProressLock`, `operatorAuthenticated`, `simulation` and `stamp`.

2.2.4 Set

`Set` is a library that provides functions for managing `SetStorage` structures that contain a maximum of `SET_MAX_ELEMENTS` of `ElementStorage` structures. Each `ElementStorage` contains a `value` of type address (20 bytes) and a `stamp` of size 12 bytes, i.e one element occupies exactly one slot. The `stamp` has value either `DUMMY_VALUE` (1), or 0. `DUMMY_VALUE` is used to keep a dirty bit in the storage slot for optimization purposes. The `SetStorage` structure contains a `numElements` field that keeps track of the number of elements in the set, a `firstElement` field, which contains the first element of the set, and a `stamp` value which is initialized to `DUMMY_VALUE` on the first time an element is inserted in the set. The `initialize()` function allows setting the `stamp` values of the array of `ElementStorage` to `DUMMY_VALUE`. The library provides functions to `insert()`, `remove()`, `reorder()` and `get()` elements. Additionally, a `contains()` function and two functions to iterate over the set and apply a function to each element are provided: `forEachAndClear()` and `forEachAndClearWithResult()`. The latter function also returns a result.

2.2.5 TransientStorage

Euler plans to use the transient storage recently added to the EVM. As for now, `TransientStorage` does not yet use transient storage but instead uses regular storage to store the execution context and the two sets that keep track of deferred account status and vault status checks. The `initialize()` function of `Set` is used to mark all the `stamp` values of a transient set to `DUMMY_VALUE`, so that the expensive storage slot initialization cost is not triggered.

2.2.6 Trust Model

The controller for an account is fully trusted by the account, as it can arbitrarily act on the account's collateral, it can arbitrarily prevent the checks from passing, and only it has the power to disable itself as the controller.

Collaterals are by default untrusted, and it is up to the controller implementation to decide how to handle them.

Operators for an account are fully trusted by the account owner not to perform actions against the account.

Operations directly submitted by users through `call()` and `batch()` are assumed to be intentional and correct. Nested operations resulting from the first layer of user operations are considered arbitrary.

No authentication is performed regarding `onBehalfOf` when calling the `msg.sender` through `call()` or `batch()`. The `msg.sender` is expected to perform the authentication.

Systems integrating with the EVC should not use `getCurrentOnBehalfOfAccount()` unless called directly by the EVC.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

- Calls That Sweep All ETH in EVC Can Fail Silently Risk Accepted
- Execution of Arbitrary Code Can Cause Denial of Service Risk Accepted
- Simulations Can Be Tricked by Malicious Systems Risk Accepted

5.1 Calls That Sweep All ETH in EVC Can Fail Silently

Correctness Low Version 1 Risk Accepted

EULEVC-001

Calls and batch items in the `EthereumVaultConnector` can transfer the whole balance of the EVC by setting the `value` to `type(uint256).max`. This can carry unintended consequences when calls are nested, even in the presence of trusted systems only:

Let's consider the following setup, where the user performs a batch call with three actions: A, B, and C.

- *A* withdraws some ether into the EVC
- *B* performs some arbitrary operation on trusted vaults
- *C* deposits the ether somewhere, using `type(uint256).max` as the value.

If *B* is to perform some action on the EVC that uses its ether balance, then *C* would fail to deposits the whole amount received in *A*, but the failure will in general not result in a revert.

This can be of course problematic if *B* triggers malicious code (the EVC documentation addresses this in the security paragraph), but it can also fail when the action performed by *B* is correct but also performs an EVC calls with `type(uint256).max` value. In the latter case, the nested EVC call performed in *B* would unintentionally use the whole ether amount from *A*.

The problem is not present when using a specified ETH value in *C*, because the *C* action would cause a revert. The use of `type(uint256).max` as value is therefore safe only when no intermediate action exists that transfers ETH to the EVC.

Risk accepted:

Euler accepts the risk with the following statement:

> We acknowledge this issue. Considering that oftentimes users might not know exactly how much value the EVC has received as a result of an operation, the EVC provides users with a convenient method to sweep all the available value by passing a special parameter of type(uint256).max. We consider this feature to outweigh potential risks associated.
>
> As per “EVC Contract Privileges” section of the EVC white paper, it is not advisable for the EVC to hold any native currency. The documentation emphasizes potential risks regarding untrusted code execution, but we agree it does not mention any side effects that may arise from multiple operations using special type(uint256).max and an input parameter for the EVC. The EVC white paper has been refined to sufficiently describe this behavior.

5.2 Execution of Arbitrary Code Can Cause Denial of Service

Security | Low | Version 1 | Risk Accepted

EULEVC-003

The execution of arbitrary code when checks are deferred can be exploited to cause denial of service of the EVC. If a user initiates a call through the EVC which triggers the execution of malicious code, the whole EVC execution can be forced to revert by introducing an account or vault status checks that fails.

Arbitrary non malicious code can also introduce EVC failures by including a number of vault or account status checks that exceeds the maximum of 10 (`SET_MAX_ELEMENTS`).

Risk Accepted:

Euler accepts the risk with the following statement:

> We acknowledge this issue. The EVC has been designed to function as a glorified multi-call contract allowing the user to execute calls into any other addresses, including contracts containing malicious code. As with any other system of such a type, it is the user's responsibility to carefully select contracts they interact with. If not careful, it is true that malicious contracts can cause denial of service attacks. However, such attacks should never pose a greater security threat to the system as a whole and with user’s care, can easily be avoided. The white paper has been refined to sufficiently describe this behavior.

5.3 Simulations Can Be Tricked by Malicious Systems

Security | Low | Version 1 | Risk Accepted

EULEVC-004

A user can simulate the effects of a batch by using `batchSimulation()` or `batchRevert()`. However, during a simulated batch, the execution context of the EVC is updated to indicate that it is in simulation mode, by setting the `SimulationInProgress` flag. This flag can be checked by any vault or external system that the EVC interacts with. Therefore, malicious vaults or external systems could use this information to act differently during simulation mode, in order to trick the user into thinking that the vault/external system is not malicious. Simulations should not be used as a security measure to determine the effects of a batch if the systems with which the batch interacts are untrusted.

Risk Accepted:

Euler accepts the risk with the following statement:

> We acknowledge this issue. The `simulationInProgress` flag, same as `operatorAuthenticated` flag, has been introduced in the system on purpose. None of them is used internally by the EVC, they both have been introduced so that they can be observed by the external smart contracts the user interacts with through the EVC. Although, as noticed in the issue description, those flags allow the contracts called to modify the behavior and execution path, their existence may increase the UX and hence we consider this feature to outweigh potential risks associated.
>
> For example, the `simulationInProgress` flag can be used by a vault so that the user is able to determine the outcome of the operation even if they do not currently hold tokens required to carry out such an operation, i.e. deposit into a vault.
>
> As with any other EVC feature, users should only use the EVC simulation with trusted and recognized smart contracts that do not aim to trick or harm them in any way. Considering the EVC simulation features are mostly meant to be used by the UI applications, we believe this is the natural place where user protection should be applied. If the user aims to faithfully evaluate the outcome of the simulation to assess the security of the to be executed transaction, they should resort to other methods and available commercial solutions. The white paper has been refined to sufficiently describe this behavior.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0
Informational Findings	3

- Gas Optimizations Code Corrected
- setAccountOwnerInternal() Naming Is Not Accurate Code Corrected
- Unused Variable Code Corrected

6.1 Gas Optimizations

Informational Version 1 Code Corrected

EULEVC-005

In the `EthereumVaultConnector` contract, public functions `requireAccountStatusCheck`, `requireVaultStatusCheck`, and `requireAccountAndVaultStatusCheck` are decorated with the `nonReentrantChecks` modifier. However, the functions perform different actions depending if checks are deferred or not. Since `areChecksDeferred()` and `areChecksInProgress()` are mutually exclusive (except transiently in the body of these functions), the reentrancy check can be moved to the internal version of the functions which is called if checks are not deferred. This saves 3 storage accesses every time one of these functions is called.

Several gas optimizations can be implemented in the `Set` library, all pertaining to writing values into structs that share a storage slot. If a and b share a storage slot, writing a new value into a requires first loading b from storage, so that the new `[a,b]` value can be then written in storage. If a and b are written together, the `SLOAD` is prevented. The gas optimizations in question are:

- At the end of function `insert`, around line 94 (`setStorage.numElements = uint8(numElements + 1)`), a storage load can be prevented by also setting `setStorage.firstElement`, which is known, and `setStorage.stamp`, which is always `DUMMY_STAMP` in the `setStorage` struct.
- In function `insert` when inserting at the end of the array, line 91, the `stamp` value can also be written, therefore saving a storage read. To know which value to set for `stamp`, the element-searching loop that is performed just before (lines 85-87) can also be used to query the `stamp` values of the array. They will either all be set (for transient sets), or all unset (for persistent sets), so when setting stamp at index `i`, the value of `stamp` at index `i - 1` can be used (`i >= 1`). If the second element is being inserted (`i == 0`), then the extra `SLOAD` can't be avoided, since the old value of `stamp` must be retrieved.
- In function `remove`, when replacing the removed element with the last element, at line 143, the `stamp` value can also be written to prevent an `SLOAD`. The `stamp` value to write can be known at no extra storage load costs.

- In function `reorder`, if `index1 == 0`, `setStorage.numElements` and `setStorage.stamp` can be set to their known values to prevent an extra SLOAD.
- In functions `forEachAndClear` and `forEachAndClearWithResult`, when clearing `setStorage.numElements` and `setStorage.firstElement`, `setStorage.stamp` can be set to `DUMMY_STAMP` to prevent an extra SLOAD.

Because some functions are only used on `transient` sets (`forEachandClear`), and some others only on persistent sets (`reorder`), extra optimizations are available if we accept tighter coupling between the `Set` implementation and the `EthereumVaultConnector`:

- When clearing the array elements in `forEachandClear` (and `forEachAndClearWithResult`), we can also write `setStorage.elements[i].stamp = DUMMY_STAMP`, since `forEachAndClear()` is only used on transient sets which are known to have every stamp set to `DUMMY_STAMP`.
- `reorder()` is only used on persistent sets of `accountCollaterals`, which are known to have `stamp` value `0` for entries of the `elements` array. Therefore, the `stamp` value can be set to `0` when writing the `value` of entries, saving extra SLOADs

---

Code corrected:

After evaluation by Euler, some of the optimizations were implemented while others were considered to slightly complicate the logic of the contract or increase the gas consumption.

The following optimizations were implemented:

- two additional internal functions, `requireAccountStatusCheckInternalNonReentrant` and `requireVaultStatusCheckInternalNonReentrant`, that wrap `requireAccountStatusCheckInternal` and `requireVaultStatusCheckInternal` accordingly, have been added to the `EthereumVaultConnector` and used in `requireAccountStatusCheck`, `requireVaultStatusCheck` and `requireAccountAndVaultStatusCheck` functions.
- forEachAndClear and forEachAndClearWithResult have been modified.

6.2 Unused Variable

Informational | Version 1 | Code Corrected

EULEVC-008

The variable `STAMP_MASK` of `ExecutionContext` is currently unused.

Code corrected:

This variable has been removed

6.3 `setAccountOwnerInternal()` Naming Is Not Accurate

Informational | Version 1 | Code Corrected

EULEVC-009

`setAccountOwnerInternal()` seems to indicate that the function is setting the account owner for a single account. However, the function is setting the owner of all 256 accounts (for the whole address prefix), and not just a single account.

Code corrected:

This function has been inlined and removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Account Check Can Fail When Enabling Collateral

Note Version 1

`enableCollateral()` will always `requireAccountStatusCheck()`. This can be problematic in the case a position is below the LTV and above the liquidation threshold. In this case, it will prevent the account from improving the position by enabling a new collateral if the amount of enabled collateral is too small to improve the position above the LTV. The same issue can arise in `reorderCollateral()`.

7.2 Differences Between Call to Collateral and `controlCollateral`

Note Version 1

EULEVC-002

`controlCollateral()` enables the controller of an account to act on the account's collateral while impersonating the account. This is expected to be used for example in liquidations, and the controller should be able to use the collateral at their discretion. However, since `controlCollateral()` sets the `setControlCollateralInProgress` flag, a controller has more restrictions when interacting with a collateral than the owner, because the EVC can't be re-entered.

The implementation of complex collateral vaults is therefore restricted to not interact with other EVC vaults in methods used in liquidations, and controller vaults are restricted to use methods of collateral vaults that are known not to interact with the EVC. This imposes design restrictions on how the liquidating vault (controller) interacts with the collateral. When asked about it, Euler stated that operations performed as part of the liquidation flow will most likely involve share transfers or asset withdrawals, and should not contain any complex logic requiring them to perform additional EVC calls.

If a vault implementation performs EVC calls when transferring shares (`callThroughEVC`), in case the liquidation flow includes asset withdrawal from a vault, the withdrawn asset cannot be a share token of another vault (vaults nesting). This restriction is however unlikely to cause problems because nested vaults make a poor choice of collateral from a risk management perspective.

7.3 Inefficient ETH Transfer in `callThroughEVC()`

Note Version 1

In `callThroughEVC()` a substantial amount of gas is spent if the message value is positive because the ether is sent to the EVC and back to the Vault resulting in the ether being moved three times instead of once. Every call that transfers ether costs at least 6800 gas, so at least an additional 13600 gas is spent.

7.4 Inter-dependencies in Checks

Note Version 1

Checks on accounts and vaults are performed in the order they were added to the set. These checks might read and modify the state of third-party smart contracts such that subsequent checks will behave differently due to the modified state. Therefore, the order in which the checks are performed can matter.

While this is not a problem for the EVC itself, it should be considered by any vault implementation that relies on the EVC. Vault should not depend on the order of checks execution as it is not guaranteed, since different call nesting can influence it.

7.5 Nonces Are Shared by Addresses With Colliding Prefix

Note Version 1

EULEVC-007

While extremely unlikely, two different addresses may share the same address prefix. In this case, the first address to be authenticated with the EVC will be the owner of the address prefix preventing the other address from authenticating on behalf of the address prefix. However, the second address can still sign permit messages using nonces for the address prefix. Therefore, the second address can invalidate nonces used by the owner's address by signing a permit message with the same nonce and front running the call to `permit` by the owner's address.

7.6 Vault Composability Is Limited by the Maximum Amount of Deferred Checks

Note Version 1

The amount of vaults and accounts that can be checked during the check phase is limited by the size of the respective sets. Therefore, any nested operation that results in more than 10 deferred vault or account checks will fail.

Therefore, vaults containing other vaults could stop working with the EVC if the contained vaults change their behavior by requesting additional checks, which would lead to the above-mentioned limit being exceeded. This is a limitation of the EVC that vault developers should be aware of.