OMNISCIA

May 16, 2024

SMART CONTRACT AUDIT REPORT

Euler Finance
Ethereum Vault Connector


omniscia.io


info@omniscia.io


Online report: euler-finance-ethereum-vault-connector

Omniscia.io is one of the fastest growing and most trusted blockchain security firms and has rapidly become a true market leader. To date, our team has collectively secured over 370+ clients, detecting 1,500+ high-severity issues in widely adopted smart contracts.

Founded in France at the start of 2020, and with a track record spanning back to 2017, our team has been at the forefront of auditing smart contracts, providing expert analysis and identifying potential vulnerabilities to ensure the highest level of security of popular smart contracts, as well as complex and sophisticated decentralized protocols.

Our clients, ecosystem partners, and backers include leading ecosystem players such as L'Oréal, Polygon, AvaLabs, Gnosis, Morpho, Vesta, Gravita, Olympus DAO, Fetch.ai, and LimitBreak, among others.

To keep up to date with all the latest news and announcements follow us on twitter @omniscia_sec.

omniscia.io


info@omniscia.io

Ethereum Vault Connector Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
69626eba20	March 27th 2024	a9eee03d03
577d1991de	April 7th 2024	189f419915
577d1991de	April 7th 2024	7b5998275f
c943dcbbb6	May 3rd 2024	33f824c042
c943dcbbb6	May 16th 2024	3baaa02db7

Audit Overview

We were tasked with performing an audit of the Euler Finance codebase and in particular their Ethereum Vault Connector module.

High-Level Description

The EVC module represents a security-conscious router via which users can interact and associate themselves with vaults that integrate with the EVC system.

The EVC system itself exposes multiple mechanisms via which DeFi interactions are performed in a controlled environment, guaranteeing that post-execution checks (such as solvency evaluations for borrowing protocols) can be mandated whenever a user interacts with their collateral which is in the custody of collateral vaults.

Sub-Account Model

Within the EVC, a particular account can ultimately own up to `256` accounts inclusive of their own by using a prefix-association pattern which ignores the `8` rightmost bits of the address's representation.

While the security of a particular address's uniqueness is reduced from `160` to `152` bits as a result of this pattern, the EVC router will associate a particular `256` address range with the "first" address that interacts with the protocol.

As such, even if a theoretical attack was carried out to identify an EOA with a common prefix as the original user who captured a particular range, the identified address would be unusable as the `owner` of the prefix would have already been registered.

Based on the above, a practical attack should never manifest even if the security of `152` bits was considered breached as it would be conditional on an account expressing interest to register its prefix, an on-chain race condition capturing the prefix before them, and them having approved certain collateral vaults with assets beforehand that would be registered by the attacker and would ultimately be compromised.

Specification

The codebase was validated against its whitepaper as well as specification, and three discrepancies were observed one of which is considered an invariant breach under non-standard circumstances; to note, this breach does not constitute a security vulnerability.

From an optimizational perspective, all gas optimizations were evaluated by being applied locally and having the test suites ran to identify the median cost deltas via `forge snapshot`.

Given that gas-specific tests were only present for the `Set` implementation, the other gas-related optimizations should be re-evaluated once gas-specific tests have been introduced for the `EthereumVaultConnector` itself.

Security

The codebase does rely on indirect security checks at times that should be better documented, such as the invocation of `EthereumVaultConnector::authenticateCaller` in `EthereumVaultConnector::setAccountOperator` inferring that the `EthereumVaultConnector::getAccountOwnerInternal` function will yield a non-zero address.

We advise these implicit security checks to be properly documented, aiding future auditing endeavours in understanding the code's operation more efficiently.

The overall security of the EVC is inherently tied to the `IVault` implementations a particular account is attached to, and as such emphasis should be put in ensuring these implementations are adequately audited in the context of the EVC.

In this regard, we identified a potential issue in the way controller validations are performed which may be considered overly restrictive and thus result in an unhandled failure.

Vault Integrations

In addition to the above, we believe proper integration of the EVC is slightly complex and contains a lot of nuisances that may not be immediately apparent.

For example, the result of the `EthereumVaultConnector::getCurrentOnBehalfOfAccount` call should not be trusted if the top-level call was not initiated by the EVC itself as it may have been diverted by another contract.

Multiple similar security checks are expected to be implemented by `IVault` implementations but may not be immediately apparent by external developers.

To this end, we advise a comprehensive development guide to be produced that aids `IVault` developers in understanding the exact ramifications of each transient data point in the EVC as well as how to properly integrate each function within it.

Compilation

As a final note, the codebase is presently compiled with `0.8.24` whilst its `pragma` statements are unrestricted from `0.8.19` upwards.

As specified in the compilation chapter, we advise test suites to be run using multiple compiler versions to ensure no compilation discrepancies remain undetected in previous / future versions.

Overview Conclusion

We advise the Euler Finance team to closely evaluate all minor findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

Post-Audit Conclusion

The Euler Finance team iterated through all findings within the report and provided us with a revised commit hash to evaluate all exhibits on.

We evaluated all alleviations performed by Euler Finance and have validated that all non-informational exhibits have been adequately dealt with.

In relation to the `informational` findings within the audit report, the following have been partially addressed and should be revisited: **`ECT-01C`**, **`EVR-05C`**

The Euler Finance team provided follow-up alleviations and responses that are reflected in the next post-audit conclusion chapter.

Post-Audit Conclusion (c943dcbbb6)

The Euler Finance team proceeded to fully address exhibit `EVR-05C` whilst acknowledging `ECT-01C` explicitly in a manner that aligns with best practices, rendering both exhibits fully addressed.

We consider all outputs of the audit report properly consumed by the Euler Finance team with no further actions expected, thus concluding this audit engagement.

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	0	0	0	0
Informational	19	17	0	2
Minor	3	3	0	0
Medium	0	0	0	0
Major	0	0	0	0

During the audit, we filtered and validated a total of **1 findings utilizing static analysis** tools as well as identified a total of **21 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they can introduce potential misbehaviours of the system as well as exploits.

Scope

The audit engagement encompassed a specific list of contracts that were present in the commit hash of the repository that was in scope. The tables below detail certain meta-data about the target of the security assessment and a navigation chart is present at the end that links to the relevant findings per file.

Target

- Repository: **https://github.com/euler-xyz/ethereum-vault-connector**
- Commit: 69626eba206ae301b919f021e71b49feaf7ea8f5
- Language: Solidity
- Network: Ethereum
- Revisions: **69626eba20**, **577d1991de**, **c943dcbbb6**

Contracts Assessed

File	Total Finding(s)
src/Errors.sol (ESR)	0
src/Events.sol (EST)	0
src/utils/EVCUtil.sol (EVC)	2
src/ExecutionContext.sol (ECT)	1
src/EthereumVaultConnector.sol (EVR)	14
src/Set.sol (STE)	5
src/TransientStorage.sol (TSE)	0

Compilation

The project utilizes `foundry` as its development pipeline tool, containing an array of tests and scripts coded in Solidity.

To compile the project, the `build` command needs to be issued via the `forge` CLI tool:

```bash
forge build
```

The `forge` tool automatically selects Solidity version `0.8.24` based on the version specified within the `foundry.toml` file.

The project contains discrepancies with regards to the Solidity version used as the `pragma` statements of the contracts are open-ended (`^0.8.19`).

As the project is meant to implement a framework, we advise multiple compiler versions to be utilized during testing to ensure the code's robustness in all eligible compiler versions.

We used version `0.8.24` for our static analysis as well as optimizational review of the codebase.

During compilation with the `foundry` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Static Analysis

The execution of our static analysis toolkit identified **10 potential issues** within the codebase of which **9 were ruled out to be false positives** or negligible findings.

The remaining **1 issues** were validated and grouped and formalized into the **1 exhibits** that follow:

ID	Severity	Addressed	Title
EVC-01S	Informational	Yes	Inexistent Sanitization of Input Address

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Euler Finance's Ethereum Vault Connector.

As the project at hand implements a special-purpose interaction router and framework, intricate care was put into ensuring that the **execution flows within the system conforms to the specifications and restrictions** laid forth within the protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed two discrepancies** within the system in relation to its specification which we advise the Euler Finance team to evaluate. More details can be observed in the report's summary as well as in the dedicated findings of each discrepancy.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to an exemplary extent, containing comprehensive in-line documentation, a whitepaper, as well as a technical specification document.

A total of **21 findings** were identified over the course of the manual review of which **9 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
EVC-01M	Minor	Yes	Inexistent Prevention of Modifier Misuse
EVR-01M	Informational	Yes	Ambiguous Specification Invariant (31)
EVR-02M	Informational	Yes	Documentation Discrepancy of Security Flags
EVR-03M	Informational	Yes	Potentially Illegible Revert Error
EVR-04M	Informational	Yes	Potentially Insecure Acceptance of Funds

EVR-05M	Informational	Acknowledged	Potentially Restrictive Push Pattern
EVR-06M	Informational	Nullified	Undocumented Operator Restriction
EVR-07M	Minor	Yes	Arbitrary Consumption of Permit Operations
EVR-08M	Minor	Yes	Invalidation of Specification Invariant (27)

Code Style

During the manual portion of the audit, we identified **12 optimizations** that can be applied to the codebase that will decrease the operational cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
EVR-01C	Informational	Yes	Illegible Authentication Invocation Style
EVR-02C	Informational	Acknowledged	Inefficient `mapping` Lookups
EVR-03C	Informational	Yes	Inexistent Named Arguments
EVR-04C	Informational	Yes	Optimization of Caller Authentication
EVR-05C	Informational	Yes	Repetitive Value Literal
EVR-06C	Informational	Yes	Suboptimal Struct Declaration Style
ECT-01C	Informational	Yes	Optimization of Initialization
STE-01C	Informational	Yes	Discrepancy of Proposed Maximum Element Limitation
STE-02C	Informational	Yes	Inexistent Avoidance of Bit Masking (Stamp & First Element Miss
STE-03C	Informational	Yes	Inexistent Avoidance of Bit Masking (Stamp Missing)

STE-04C	Informational	Yes	Inexistent Correlation of Constants
STE-05C	Informational	Yes	Optimization of Interim Element Removal

EVCUtil Static Analysis Findings

EVC-01S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Informational	EVCUtil.sol:L17-L19

Description:

The linked function accepts an `address` argument yet does not properly sanitize it.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

src/utils/EVCUtil.sol

SOL

```sol
17  constructor(IEVC _evc) {
18      evc = _evc;
19  }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that the `address` specified is non-zero.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The input `_evc` address argument of the **`EVCUtil::constructor`** function is adequately sanitized as non-zero in the latest in-scope revision of the codebase, addressing this exhibit.

EVCUtil Manual Review Findings

EVC-01M: Inexistent Prevention of Modifier Misuse

Type	Severity	Location
Logical Fault	Minor	EVCUtil.sol:L29

Description:

The `EVCUtil::callThroughEVC` modifier co-exists with the `EVCUitl::callThroughEVCPayable` modifier with the latter equipped to handle native funds.

Impact:

Integrators of the `EVCUtil` contract may mistakenly specify the non-payable variant of the `EVCUtil::callThorughEVC` modifier for a `payable` function, an integration not prevented by the code itself.

Example:

src/utils/EVCUtil.sol

SOL

```sol
21  /// @notice Ensures that the msg.sender is the EVC by using the EVC callback
functionality if necessary.
22  /// @dev Optional to use for functions requiring account and vault status checks
to enforce predictable behavior.
23  /// @dev If this modifier used in conjuction with any other modifier, it must
appear as the first (outermost)
24  /// modifier of the function.
25  modifier callThroughEVC() virtual {
26      if (msg.sender == address(evc)) {
27          _;
28      } else {
29          bytes memory result = evc.call(address(this), msg.sender, 0, msg.data);
30
```

Example (Cont.):

```sol
        assembly {
            return(add(32, result), mload(result))
        }
    }
}

/// @notice Ensures that the msg.sender is the EVC by using the EVC callback functionality if necessary.
/// @dev Optional to use for functions requiring account and vault status checks to enforce predictable behavior.
/// @dev If this modifier used in conjuction with any other modifier, it must appear as the first (outermost)
/// modifier of the function.
/// @dev This modifier is used for payable functions because it forwards the value to the EVC.
modifier callThroughEVCPayable() virtual {
    if (msg.sender == address(evc)) {
        _;
    } else {
        bytes memory result = evc.call{value: msg.value}(address(this), msg.sender, msg.value, msg.data);

        assembly {
            return(add(32, result), mload(result))
        }
    }
}
```

Recommendation:

We advise either a single implementation to be present, properly branching its code based on whether `msg.value` is non-zero, or the **`EVCUtil::callThroughEVC`** modifier to prevent a non-zero `msg.value`. Modifiers cannot control whether they are specified in a `payable` environment, and as such their misuse should be validated based on their intended operation.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The functionality of both the **`EVCUtil::callThroughEVC`** and **`EVCUtil::callThroughEVCPayable`** modifiers has been merged under a single implementation per our recommendation, utilizing a low-level `assembly` block to perform the interaction.

We have validated that the `modifier` is securely implemented, with a successful call properly accounting for the `32` byte data offset as well as `32` byte payload length in the return data.

As such, we consider this exhibit fully addressed with a uniform `modifier` for both `payable` and `nonpayable` interactions.

EthereumVaultConnector Manual Review Findings

EVR-01M: Ambiguous Specification Invariant (31)

Type	Severity	Location
Standard Conformity	Informational	EthereumVaultConnector.sol:L528

Description:

The Ethereum Vault Connector **specification** and specifically bullet point 31 uses RFC-2119 terminology and specifies that:

> Only the Checks-Deferrable Calls MUST allow to re-enter the EVC.

This particular point of the specification is relatively ambiguous, and is potentially breached by the `EthereumVaultConnector::permit` implementation.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
473 /// @inheritdoc IEVC
474 function permit(
475     address signer,
476     uint256 nonceNamespace,
477     uint256 nonce,
478     uint256 deadline,
479     uint256 value,
480     bytes calldata data,
481     bytes calldata signature
482 ) public payable virtual nonReentrantChecksAndControlCollateral {
```

Example (Cont.):

SOL

```sol
    // cannot be called within the self-call of the permit function; can occur for nested calls
    if (inPermitSelfCall()) {
        revert EVC_NotAuthorized();
    }

    if (signer == address(0) || !isSignerValid(signer)) {
        revert EVC_InvalidAddress();
    }

    bytes19 addressPrefix = getAddressPrefixInternal(signer);
    uint256 currentNonce = nonceLookup[addressPrefix][nonceNamespace];

    if (currentNonce == type(uint256).max || currentNonce != nonce) {
        revert EVC_InvalidNonce();
    }

    if (deadline < block.timestamp) {
        revert EVC_InvalidTimestamp();
    }

    if (data.length == 0) {
        revert EVC_InvalidData();
    }

    bytes32 permitHash = getPermitHash(signer, nonceNamespace, nonce, deadline, value, data);

    if (
        signer != recoverECDSASigner(permitHash, signature)
```

Example (Cont.):

```sol
            && !isValidERC1271Signature(signer, permitHash, signature)
    ) {
        revert EVC_NotAuthorized();
    }

    if (ownerLookup[addressPrefix].isPermitDisabledMode) {
        revert EVC_PermitDisabledMode();
    }

    unchecked {
        nonceLookup[addressPrefix][nonceNamespace] = currentNonce + 1;
    }

    emit NonceUsed(addressPrefix, nonceNamespace, nonce);

    // EVC address becomes the msg.sender for the duration this self-call, no authentication is required here.
    // the signer will be later on authenticated as per data, depending on the functions that will be called
    (bool success, bytes memory result) = callWithContextInternal(address(this), signer, value, data);

    if (!success) revertBytes(result);
}
```

Recommendation:

We advise this particular point of the specification to be rephrased and refined to a more granular level, permitting its association with the code to be made with ease.

We believe the invariant is "breached" deliberately by the `EthereumVaultConnector::permit` call, and thus the permit-related nuisance should be introduced to the aforementioned bullet point of the project's specification.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The Euler Finance team highlighted that the specifications outlined within the Ethereum Vault Connector repository might be outdated, and that the specifications within the Linear software should be considered up-to-date.

As such, the relevant re-entrancy requirement labelled `CER-49` has been updated in the Linear app to reflect the deliberate permit-related re-entrancy thereby addressing this exhibit.

EVR-02M: Documentation Discrepancy of Security Flags

Type	Severity	Location
Standard Conformity	Informational	EthereumVaultConnector.sol:L285, L303, L473-L531

Description:

The whitepaper of the Ethereum Vault Connector and specifically the **`Permit Disabled Mode`** subsection of the Security Considerations chapter details that the `PERMIT DISABLED MODE` is a subset of the `LOCKDOWN MODE` which is not correct.

While the `LOCKDOWN MODE` itself cannot be affected, **`EthereumVaultConnector::permit`** operations will successfully execute during its duration and can result in operators being mutated, nonces being invalidated, and so on.

Impact:

As no exploitable attack vector arises from this discrepancy, we consider it to be a documentation mismatch rather than vulnerability.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
473 /// @inheritdoc IEVC
474 function permit(
475     address signer,
476     uint256 nonceNamespace,
477     uint256 nonce,
478     uint256 deadline,
479     uint256 value,
480     bytes calldata data,
481     bytes calldata signature
482 ) public payable virtual nonReentrantChecksAndControlCollateral {
```

Example (Cont.):

```sol
    // cannot be called within the self-call of the permit function; can occur for nested calls
    if (inPermitSelfCall()) {
        revert EVC_NotAuthorized();
    }

    if (signer == address(0) || !isSignerValid(signer)) {
        revert EVC_InvalidAddress();
    }

    bytes19 addressPrefix = getAddressPrefixInternal(signer);
    uint256 currentNonce = nonceLookup[addressPrefix][nonceNamespace];

    if (currentNonce == type(uint256).max || currentNonce != nonce) {
        revert EVC_InvalidNonce();
    }

    if (deadline < block.timestamp) {
        revert EVC_InvalidTimestamp();
    }

    if (data.length == 0) {
        revert EVC_InvalidData();
    }

    bytes32 permitHash = getPermitHash(signer, nonceNamespace, nonce, deadline, value, data);

    if (
        signer != recoverECDSASigner(permitHash, signature)
```

Example (Cont.):

```sol
            && !isValidERC1271Signature(signer, permitHash, signature)
    ) {
        revert EVC_NotAuthorized();
    }

    if (ownerLookup[addressPrefix].isPermitDisabledMode) {
        revert EVC_PermitDisabledMode();
    }

    unchecked {
        nonceLookup[addressPrefix][nonceNamespace] = currentNonce + 1;
    }

    emit NonceUsed(addressPrefix, nonceNamespace, nonce);

    // EVC address becomes the msg.sender for the duration this self-call, no authentication is required here.
    // the signer will be later on authenticated as per data, depending on the functions that will be called
    (bool success, bytes memory result) = callWithContextInternal(address(this), signer, value, data);

    if (!success) revertBytes(result);
}
```

Recommendation:

We do not consider these permit consumptions to be of security concern as the `LOCKDOWN MODE` cannot be disabled with a permit and thus only permitted actions can occur via the **`EthereumVaultConnector::permit`**.

In order to alleviate this discrepancy, we advise either the documentation to be updated to not state that the `PERMIT DISABLED MODE` is a subset, or the `isPermitDisabledMode` flag to be set in tandem with the `isLockdownMode` flag during an execution of the **`EthereumVaultConnector::setLockdownMode`** to an `enabled` status of `true`.

We consider either of the two aforementioned solutions as adequate in resolving this whitepaper and code discrepancy.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The documentation discrepancy has been corrected in the Ethereum Vault Connector whitepaper, effectively addressing this exhibit.

EVR-03M: Potentially Illegible Revert Error

Type	Severity	Location
Standard Conformity	Informational	EthereumVaultConnector.sol:L856, L906

Description:

The `EthereumVaultConnector::checkStatusAllWithResult` function is meant to be utilized during simulations to evaluate exactly where a failure occurred, however, the `EthereumVaultConnector::checkAccountStatusInternal` function may `revert` without any indication of the account being checked.

Impact:

The present simulation environment may revert without providing the caller with adequate information to debug the transaction due to an indefinite `EVC_ControllerViolation` error arising.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
851 function checkAccountStatusInternal(address account) internal virtual returns
(bool isValid, bytes memory result) {
852     uint256 numOfControllers = accountControllers[account].numElements;
853     address controller = accountControllers[account].firstElement;
854
855     if (numOfControllers == 0) return (true, "");
856     else if (numOfControllers > 1) revert EVC_ControllerViolation();
857
858     bool success;
859     (success, result) =
860         controller.call(abi.encodeCall(IVault.checkAccountStatus, (account,
accountCollaterals[account].get())));
```

Example (Cont.):

```sol

    isValid = success && result.length == 32
        && abi.decode(result, (bytes32)) ==
bytes32(IVault.checkAccountStatus.selector);

    emit AccountStatusCheck(account, controller);
}
```

Recommendation:

We advise the code to instead yield `false` and the `EVC_ControllerViolation` error encoded as the `result` payload, delegating handling of the error to its callers.

Alternatively, we advise an argument to be introduced to the `error` itself that indicates for which `account` the controller violation occurred ensuring that even if uncaught, the overall transaction's reversion will depict which `account` caused the failure

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

Our primary recommendation has been applied to the codebase, yielding `false` to indicate that the call failed and returning the `EVC_ControllerViolation` error selector thus permitting it to bubble-up properly.

EVR-04M: Potentially Insecure Acceptance of Funds

Type	Severity	Location
Logical Fault	Informational	EthereumVaultConnector.sol:L93-L95

Description:

The `EthereumVaultConnector` is never expected to retain funds at rest, and would only receive native funds as part of a batch operation temporarily.

Impact:

The `EthereumVaultConnector` will presently accept native funds even if they were directly transmitted to it which we consider incorrect as its specification details funds should only be held transiently during a batch operation.

To note, blind acceptance of finds is distinct from funds that are distributed as part of a normal `payable` function call in which case we can assume the caller has deliberately sent them.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
93  receive() external payable {
94      // only receives value, no need to do anything here
95  }
```

Recommendation:

We advise this to be reflected in the code itself, ensuring that checks are deferred whenever the `EthereumVaultConnector::receive` function is triggered and thus ensuring funds are never accidentally at risk.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The `EthereumVaultConnector::receive` function was updated to accept receipt of native funds solely when checks are deferred, implying that the funds were received in between a batch operation.

As such, we consider this exhibit fully addressed.

EVR-05M: Potentially Restrictive Push Pattern

Type	Severity	Location
Standard Conformity	Informational	EthereumVaultConnector.sol:L451-L461, L464-L469

Description:

Once a controller is attached to a particular account, it cannot be removed unless the controller themselves will initiate a transaction and invoke the `EthereumVaultConnector::disableController`.

While this is a necessary security feature to ensure that collaterals cannot be exploited while they are meant to be reserved for a controller, the current mechanism is prohibitive in case of failure of the controller and could result in an inoperative account if the `IVault::checkAccountStatus` function of the controller reverts for any reason.

Impact:

Properly behaving `IVault` controllers will not result in this error, however, we consider it imperative that an account's assets are protected in case the controller misbehaves due to an unintentional mistake, such as a controller being added that does not have the `IVault::checkAccountStatus` function defined.

To note, the expiration should be reset every time a successful validation has occurred to avoid exploitations of reverts enforced by a malicious user.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
851 function checkAccountStatusInternal(address account) internal virtual returns
(bool isValid, bytes memory result) {
852     uint256 numOfControllers = accountControllers[account].numElements;
853     address controller = accountControllers[account].firstElement;
854
855     if (numOfControllers == 0) return (true, "");
856     else if (numOfControllers > 1) revert EVC_ControllerViolation();
857
858     bool success;
859     (success, result) =
860         controller.call(abi.encodeCall(IVault.checkAccountStatus, (account,
accountCollaterals[account].get())));
```

Example (Cont.):

```sol

    isValid = success && result.length == 32
        && abi.decode(result, (bytes32)) ==
bytes32(IVault.checkAccountStatus.selector);

    emit AccountStatusCheck(account, controller);
}
```

Recommendation:

We believe that the specification of vaults meant to integrate with the `EthereumVaultConnector` should be updated to specify that the `IVault::checkAccountStatus` **MUST** not `revert` under any circumstances in RFC-2119 terminology, ensuring that the controllers developed for the EVC are aware of this trait.

Additionally, the `EthereumVaultConnector::checkAccountStatusInternal` function should handle failures differently by potentially implementing a grace period after which continuous account status failures will permit the controller to be removed.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The Euler Finance team deliberated this exhibit internally and has opted to acknowledge the behaviour described.

In detail, the Euler Finance team's response is as follows\:

> We accept the fact that the controller's `IVault::checkAccountStatus` function failure may result in inoperative account. The unopinionated nature of the Ethereum Vault Connector contract prohibit us from implementing a recommended grace period after which continuous account status failures would allow the controller to be removed.

> Additionally, it must be noted that it is desired for the `IVault::checkAccountStatus` to revert deliberately due to invalid account status. Instead of updating the documentation to say that the function must not revert under any circumstances, it was updated to say that this function must only deliberately revert if the account status is invalid. The was a note added to emphasize the fact that if this function reverts due to any other reason, it may render the account unusable with possibly no way to recover funds.

Based on the aforementioned feedback, we consider this exhibit safely acknowledged as the Euler Finance team is fully aware of the acknowledgement's ramifications.

EVR-06M: Undocumented Operator Restriction

Type	Severity	Location
Standard Conformity	Informational	EthereumVaultConnector.sol:L1052, L1054

Description:

The `operator` of a particular account is meant to be able to have access to all `256` accounts of a "master" Account per the project's documentation, however, the codebase's present structure prevents them from doing so.

Specifically, an operator must have a non-zero bitmask which in turn means that they will have access to at most `255` out of the `256` accounts attached to a "master" account. The account that is inaccessible is the owner of the address prefix, and an operator for this account might be desirable in single-account use cases as well as smart contract use cases of the EVC.

Impact:

The whitepaper and specification never explicitly mention that an operator should be assign-able to all `256` accounts of a "master" account, and as such, we cannot reliably assess the severity of this exhibit.

If this turns out to be a desirable use case, the severity will be set to `minor`. Alternatively, the severity will be set to `informational` after the adjustment of the documentation to highlight this trait.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
103
8   function isAccountOperatorAuthorizedInternal(
103
9       address account,
104
0       address operator
104
1   ) internal view returns (bool isAuthorized) {
104
2       address owner = getAccountOwnerInternal(account);
104
3
104
4       // if the owner is not registered yet, it means that the operator couldn't
have been authorized
104
5       if (owner == address(0)) return false;
104
6
104
7       bytes19 addressPrefix = getAddressPrefixInternal(account);
```

Example (Cont.):

```sol
        // The bitMask defines which accounts the operator is authorized for. The bitMask is created from the account
        // number which is a number up to 2^8 in binary, or 256. 1 << (uint160(owner) ^ uint160(account)) transforms
        // that number in an 256-position binary array like 0...010...0, marking the account positionally in a uint256.
        uint256 bitMask = 1 << (uint160(owner) ^ uint160(account));

        return operatorLookup[addressPrefix][operator] & bitMask != 0;
    }
```

Recommendation:

We advise this trait to be evaluated, and potentially reflected in the documentation properly or addressed in the code of the `EthereumVaultConnector` itself.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

After discussions with the Euler Finance team, we confirmed that this finding is **invalid** due to a misunderstanding of how the `bitMask` is calculated.

Specifically, the result of the `XOR` operation between the `owner` and the `account` will be in the `[0, 255]` inclusive range and act as a **bitwise shift input**.

The actual mask is generated by the calculation `1 << xorResult` (i.e. `1 << [0,255]`) which is guaranteed to yield a non-zero positive number that always fits within the `uint256` data type.

As such, the original exhibit is invalid and the code indeed permits the "master" account of an `addressPrefix` to be authorized to operators as well.

EVR-07M: Arbitrary Consumption of Permit Operations

Type	Severity	Location
Logical Fault	Minor	EthereumVaultConnector.sol:L510, L511

Description:

While defined as a desirable trait by the project's specification, the `EthereumVaultConnector::permit` function will permit any caller to activate a signed payload.

This may cause UX hinderances as well as potential Denial-of-Service failures for on-chain contract flows if the `EthereumVaultConnector::permit` function is directly integrated by a smart contract.

Impact:

The `EthereumVaultConnector::permit` function permits any user to submit a signed payload thereby causing an on-chain race condition to manifest if the same payload is submitted by two distinct parties.

This issue can be exacerbated by smart contracts that integrate with the `EthereumVaultConnector::permit` function, resulting in Denial-of-Service attacks if these integrations mandate a successful `EthereumVaultConnector::permit` execution.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
473 /// @inheritdoc IEVC
474 function permit(
475     address signer,
476     uint256 nonceNamespace,
477     uint256 nonce,
478     uint256 deadline,
479     uint256 value,
480     bytes calldata data,
481     bytes calldata signature
482 ) public payable virtual nonReentrantChecksAndControlCollateral {
```

Example (Cont.):

```sol
    // cannot be called within the self-call of the permit function; can occur for nested calls
    if (inPermitSelfCall()) {
        revert EVC_NotAuthorized();
    }

    if (signer == address(0) || !isSignerValid(signer)) {
        revert EVC_InvalidAddress();
    }

    bytes19 addressPrefix = getAddressPrefixInternal(signer);
    uint256 currentNonce = nonceLookup[addressPrefix][nonceNamespace];

    if (currentNonce == type(uint256).max || currentNonce != nonce) {
        revert EVC_InvalidNonce();
    }

    if (deadline < block.timestamp) {
        revert EVC_InvalidTimestamp();
    }

    if (data.length == 0) {
        revert EVC_InvalidData();
    }

    bytes32 permitHash = getPermitHash(signer, nonceNamespace, nonce, deadline, value, data);

    if (
        signer != recoverECDSASigner(permitHash, signature)
```

Example (Cont.):

```sol
            && !isValidERC1271Signature(signer, permitHash, signature)
    ) {
        revert EVC_NotAuthorized();
    }

    if (ownerLookup[addressPrefix].isPermitDisabledMode) {
        revert EVC_PermitDisabledMode();
    }

    unchecked {
        nonceLookup[addressPrefix][nonceNamespace] = currentNonce + 1;
    }

    emit NonceUsed(addressPrefix, nonceNamespace, nonce);

    // EVC address becomes the msg.sender for the duration this self-call, no authentication is required here.
    // the signer will be later on authenticated as per data, depending on the functions that will be called
    (bool success, bytes memory result) = callWithContextInternal(address(this), signer, value, data);

    if (!success) revertBytes(result);
}
```

Recommendation:

We advise the overall permit hash and payload to contain an optional `sender` argument which specifies if a specific address should only be able to activate the `EthereumVaultConnector::permit`, preventing the aforementioned race condition from manifesting.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The `EthereumVaultConnector::permit` function was updated to incorporate an optional `sender` argument that can be specified as `0` in case no dedicated sender should be restricted.

The relevant **EIP-712** hashes have been updated as well to accommodate for the new entry, ensuring that the change has been adopted properly in the signature-based component of the `EthereumVaultConnector::permit` function.

EVR-08M: Invalidation of Specification Invariant (27)

Type	Severity	Location
Logical Fault	Minor	EthereumVaultConnector.sol:L528, L626-L627

Description:

The Ethereum Vault Connector **specification** and specifically bullet point 27 uses RFC-2119 terminology and specifies that:

> Execution Context's Account on behalf of which the current low-level call is being performed MUST be storing address(0) when Account and Vault Status Checks are in progress.

This invariant can be breached by an unusual combination of function calls. In detail, a user can invoke the `EthereumVaultConnector::permit` function which will mutate the `onBehalfOfAccount` entry but will not defer checks.

Should this `EthereumVaultConnector::permit` operation be performed on the `EthereumVaultConnector::batchRevert` function, the account and vault status checks that will be performed in the following call chain will have a non-zero on behalf of account, breaching the invariant:

Impact:

We identified a breach of the Ethereum Vault Connector's invariants under unusual circumstances that should not result in a vulnerability within a production environment.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
/// @inheritdoc IEVC
function batchRevert(BatchItem[] calldata items) public payable virtual nonReentrantChecksAndControlCollateral {
    BatchItemResult[] memory batchItemsResult;
    StatusCheckResult[] memory accountsStatusCheckResult;
    StatusCheckResult[] memory vaultsStatusCheckResult;

    EC contextCache = executionContext;

    if (contextCache.areChecksDeferred()) {
        revert EVC_SimulationBatchNested();
```

Example (Cont.):

```sol
611     }
612
613     executionContext =
contextCache.setChecksDeferred().setSimulationInProgress();
614
615     uint256 length = items.length;
616     batchItemsResult = new BatchItemResult[](length);
617
618     for (uint256 i; i < length; ++i) {
619         BatchItem calldata item = items[i];
620         (batchItemsResult[i].success, batchItemsResult[i].result) =
621             callWithAuthenticationInternal(item.targetContract,
item.onBehalfOfAccount, item.value, item.data);
622     }
623
624     executionContext = contextCache.setChecksInProgress();
625
626     accountsStatusCheckResult = checkStatusAllWithResult(SetType.Account);
627     vaultsStatusCheckResult = checkStatusAllWithResult(SetType.Vault);
628
629     executionContext = contextCache;
630
631     revert EVC_RevertedBatchResult(batchItemsResult, accountsStatusCheckResult,
vaultsStatusCheckResult);
632 }
```

Recommendation:

We consider this invocation sequence to be non-standard and thus the breach of the invariant occurs under circumstances that integrators will practically never rely on.

In any case, we believe that a clarification of the specification would be appropriate given that it uses strict RFC-2119 terminology and it should be accurate in this regard.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The `EthereumVaultConnector::batchRevert` function was updated to properly zero-out the on-behalf-of account in case of the aforementioned scenario, alleviating this exhibit in full.

EthereumVaultConnector Code Style Findings

EVR-01C: Illegible Authentication Invocation Style

Type	Severity	Location
Code Style	Informational	EthereumVaultConnector.sol:L109, L121, L334, L354, L833

Description:

The `EthereumVaultConnector::authenticateCaller` function accepts boolean arguments that are imperative to its secure operation, yet utilizes an index-based invocation style for consecutive boolean variables rendering their distinction difficult.

Example:

src/EthereumVaultConnector.sol

SOL

```
354 address msgSender = authenticateCaller(account, true, false);
```

Recommendation:

We advise the key-value declaration style to be adopted for the function arguments of the `EthereumVaultConnector::authenticateCaller` function (i.e. from `authenticateCaller(account, true, false)` to `authenticateCaller({ account: account, allowOperator: true, checkLockdownMode: false })`), greatly optimizing the legibility of the function's invocations.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

All invocation instances of the `EthereumVaultConnector::authenticateCaller` function have been updated to utilize the key-value argument declaration style we advised, greatly increasing their legibility across the codebase.

EVR-02C: Inefficient `mapping` Lookups

Type	Severity	Location
Gas Optimization	Informational	EthereumVaultConnector.sol:L277, L285, L295, L303, L314, L318,

Description:

The linked statements perform key-based lookup operations on `mapping` declarations from storage multiple times for the same key redundantly.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
// The operatorBitField is a 256-position binary array, where each 1 signals by position the account that the
// operator is authorized for.
uint256 oldOperatorBitField = operatorLookup[addressPrefix][operator];
uint256 newOperatorBitField = authorized ? oldOperatorBitField | bitMask : oldOperatorBitField & ~bitMask;

if (oldOperatorBitField == newOperatorBitField) {
    revert EVC_InvalidOperatorStatus();
} else {
    operatorLookup[addressPrefix][operator] = newOperatorBitField;

```

Example (Cont.):

```sol
388     emit OperatorStatus(addressPrefix, operator, newOperatorBitField);
389 }
```

Recommendation:

As the lookups internally perform an expensive `keccak256` operation, we advise the lookups to be cached wherever possible to a single local declaration that either holds the value of the `mapping` in case of primitive types or holds a `storage` pointer to the `struct` contained.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The Euler Finance team evaluated this exhibit and opted to acknowledge it so as to retain a higher level of legibility in their codebase.

EVR-03C: Inexistent Named Arguments

Type	Severity	Location
Code Style	Informational	EthereumVaultConnector.sol:L80, L82

Description:

The code of the `EthereumVaultConnector` utilizes the named key and value format for `mapping` declarations, however, the `accountCollaterals` and `accountControllers` do not employ this style for their `SetStorage` values.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
74  mapping(bytes19 addressPrefix => OwnerStorage ownerStorage) internal ownerLookup;
75
76  mapping(bytes19 addressPrefix => mapping(address operator => uint256
operatorBitField)) internal operatorLookup;
77
78  mapping(bytes19 addressPrefix => mapping(uint256 nonceNamespace => uint256
nonce)) internal nonceLookup;
79
80  mapping(address account => SetStorage) internal accountCollaterals;
81
82  mapping(address account => SetStorage) internal accountControllers;
```

Recommendation:

We advise a name to be introduced for those declarations, ensuring consistency within the codebase.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The Euler Finance team clarified that named arguments are utilized solely in the case that the data type itself is not self-explanatory, and proceeded to remove the `ownerStorage` named value from the `ownerLookup` declaration.

As such, we consider this exhibit properly addressed.

EVR-04C: Optimization of Caller Authentication

Type	Severity	Location
Gas Optimization	Informational	EthereumVaultConnector.sol:L753, L755, L760, L765

Description:

The caller authentication mechanism in `EthereumVaultConnector::authenticateCaller` is slightly inefficient in its current state as it will utilize an `authenticated` boolean redundantly.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
732 function authenticateCaller(
733     address account,
734     bool allowOperator,
735     bool checkLockdownMode
736 ) internal virtual returns (address) {
737     bytes19 addressPrefix = getAddressPrefixInternal(account);
738     address owner = ownerLookup[addressPrefix].owner;
739     bool lockdownMode = ownerLookup[addressPrefix].isLockdownMode;
740 
741     if (checkLockdownMode && lockdownMode) {
```

Example (Cont.):

```sol
        revert EVC_LockdownMode();
    }

    address msgSender = _msgSender();
    bool authenticated = false;

    // check if the caller is the owner of the account
    if (haveCommonOwnerInternal(account, msgSender)) {
        // if the owner is not registered, register it
        if (owner == address(0)) {
            setAccountOwnerInternal(account, msgSender);
            authenticated = true;
        } else if (owner == msgSender) {
            authenticated = true;
        }
    }

    // if the caller is not the owner, check if it is an operator if operators are allowed
    if (!authenticated && allowOperator) {
        authenticated = isAccountOperatorAuthorizedInternal(account, msgSender);
    }

    // must revert if neither the owner nor the operator were authenticated
    if (!authenticated) {
        revert EVC_NotAuthorized();
    }

    return msgSender;
```

Example (Cont.):

```sol
770 }
```

Recommendation:

We advise the `authenticated` flag to be omitted entirely, and early returns to be performed instead.

Specifically, the `if` branch that validates `EthereumVaultConnector::haveCommonOwnerInternal` can yield the `msgSender` immediately in place of the `true` assignments of the `authenticated` flag. Continuation of execution infers that `authenticated` is `false`, so the validation of the `authenticated` flag can be omitted and only `allowOperator` can be evaluated in the ensuing `if` clause.

The code can be further optimized by evaluating `allowOperator && isAccountOperatorAuthorizedInternal(account, msgSender)` in the `if` clause, yielding the `msgSender` if the clause is `true`.

Finally, the code can unconditionally revert with an `EVC_NotAuthorized` error, as the `authenticated` flag would be guaranteed as `false` after the aforementioned validations have occurred.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The code was precisely updated per our recommendations, optimizing its gas cost in the process.

EVR-05C: Repetitive Value Literal

Type	Severity	Location
Code Style	Informational	EthereumVaultConnector.sol:L108, L333, L1030

Description:

The linked value literal is repeated across the codebase multiple times.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
108 address phantomAccount = address(uint160(uint152(addressPrefix)) << 8);
```

Recommendation:

We advise it to be set to a `constant` variable instead, optimizing the legibility of the codebase.

In case the `constant` has already been declared, we advise it to be properly re-used across the code.

Alleviation (577d1991de):

The `ACCOUNT_ID_OFFSET` constant declaration has replaced the `8` value literal in the first two referenced lines of the exhibit, however, the last line within the `EthereumVaultConnector::getAddressPrefixInternal` function remains as a literal rendering this exhibit partially alleviated.

Alleviation (c943dcbbb6):

The `EthereumVaultConnector::getAddressPrefixInternal` function's literal has been replaced by the `ACCOUNT_ID_OFFSET` as well, addressing this exhibit in full.

EVR-06C: Suboptimal Struct Declaration Style

Type	Severity	Location
Code Style	Informational	EthereumVaultConnector.sol:L915

Description:

The linked declaration style of a struct is using index-based argument initialization.

Example:

src/EthereumVaultConnector.sol

SOL

```sol
915 checksResult[i] = StatusCheckResult(checkedAddress, isValid, result);
```

Recommendation:

We advise the key-value declaration format to be utilized instead, greatly increasing the legibility of the codebase.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The key-value declaration style is now properly in use within the referenced `struct` declaration, addressing this exhibit in full.

ExecutionContext Code Style Findings

ECT-01C: Optimization of Initialization

Type	Severity	Location
Gas Optimization	Informational	ExecutionContext.sol:L42

Description:

The `ExecutionContext::initialize` function is solely invoked in the `TransientStorage::constructor` and thus will always accept an `EC` input argument that is equal to `0`.

Example:

src/ExecutionContext.sol

SOL

```sol
function initialize(EC self) internal pure returns (EC result) {
    // prepopulate the execution context storage slot to optimize gas consumption
    // (it should never be cleared again thanks to the stamp)
    result = EC.wrap(EC.unwrap(self) | (STAMP_DUMMY_VALUE << STAMP_OFFSET));
}
```

Recommendation:

We advise the initial value of the `EC` flag to be set as `library` level `constant` that is assigned to the `executionContext` at the **`TransientStorage::constructor`**, optimizing the code's gas cost.

Alleviation (577d1991de):

The **`ExecutionContext::initialize`** function has been removed from the codebase, and the instance where it was invoked within the **`TransientStorage::constructor`** has been updated to initialize the `executionContext` as a direct statement.

While functionally our recommendation has been applied, we advised the newly introduced expression (`EC.wrap(1 << ExecutionContext.STAMP_OFFSET)`) to be exposed as a `constant` by the `ExecutionContext` library. This would ensure that the `library` can be utilized in other codebases without repeating the initialization statement performed in the **`TransientStorage::constructor`** function.

Alleviation (c943dcbbb6):

The Euler Finance team evaluated our follow-up recommendation and opted to not expose the expression as a `constant` as the `ExecutionContext` is meant to be utilized solely in the context of the EVC. As such, we consider this exhibit adequately addressed to the extent that it aligns with the Euler Finance team's wishes.

Set Code Style Findings

STE-01C: Discrepancy of Proposed Maximum Element Limitation

Type	Severity	Location
Standard Conformity	Informational	Set.sol:L31, L42, L85, L87

Description:

The `Set::MAX_ELEMENTS` constant specifies that the value must not exceed `255`, however, a value equal to `255` would permit writes up to index `254` to occur for the `elements` entry of a `SetStorage`.

As such, the last element of the `SetStorage::elements` key would never be written to whose index would be `255`.

Impact:

The length of the `SetStorage::elements` entry is inefficient and will always contain an extraneous slot.

Example:

src/Set.sol

SOL

```
16  /// @title SetStorage
17  /// @notice This struct is used to store the set data.
18  /// @dev To optimize the gas consumption, firstElement is stored in the same
storage slot as the numElements
19  /// @dev so that for sets with one element, only one storage slot has to be
read/written. To keep the elements
20  /// @dev array indexing consistent and because the first element is stored
outside of the array, the elements[0]
21  /// @dev is not utilized. The stamp field is used to keep the storage slot non-
zero when the element is removed.
22  /// @dev It allows for cheaper SSTORE when an element is inserted.
23  struct SetStorage {
24      /// @notice The number of elements in the set.
25      uint8 numElements;
```

Example (Cont.):

```sol
    /// @notice The first element in the set.
    address firstElement;
    /// @notice The stamp of the set.
    uint88 stamp;
    /// @notice The array of elements in the set. Stores the elements starting from index 1.
    ElementStorage[2 ** 8] elements;
}

/// @title Set
/// @author Euler Labs (https://www.eulerlabs.com/)
/// @notice This library provides functions for managing sets of addresses.
/// @dev The maximum number of elements in the set is defined by the constant MAX_ELEMENTS.
library Set {
    error TooManyElements();
    error InvalidIndex();

    uint8 internal constant MAX_ELEMENTS = 10; // must not exceed 255
```

Recommendation:

As the `SetStorage::numElements` entry is constrained to the `uint8` data type, we advise the `SetStorage::elements` array length to be configured as `2 ** 8 - 1`, or `type(uint8).max`, properly illustrating the maximum elements that the `SetStorage::elements` could practically contain.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

As part of exhibit's **`STE-04C`** remediations, a consistent `constant` is utilized for both the `SetStorage::elements` array as well as the code's original `MAX_ELEMENTS` constant.

As such, we consider this exhibit indirectly addressed as extraneous slots are no longer present.

STE-02C: Inexistent Avoidance of Bit Masking (Stamp & First Element Missing)

Type	Severity	Location
Gas Optimization	Informational	Set.sol:L138, L141

Description:

The referenced statements are meant to avoid Solidity's compiler-generated bit-masking, however, this is not presently achieved. The code will partially assign to a 256-bit slot, and thus will result in bit masking to be generated by the compiler.

Example:

src/Set.sol

SOL

```sol
130 // set numElements for every execution path to avoid SSTORE and bit masking when the element removed is
131 // firstElement
132 if (searchIndex != lastIndex) {
133     if (searchIndex == 0) {
134         setStorage.firstElement = setStorage.elements[lastIndex].value;
135         setStorage.numElements = uint8(lastIndex);
136     } else {
137         setStorage.elements[searchIndex].value = setStorage.elements[lastIndex].value;
138         setStorage.numElements = uint8(lastIndex);
139     }
```

Example (Cont.):

```sol
} else {
    setStorage.numElements = uint8(lastIndex);
}
```

Recommendation:

We advise the referenced statements to be accompanied by assignments to the `setStorage.firstElement` and `setStorage.stamp` as the `firstElement` value already exists in memory, optimizing the code's compiler-incurred gas cost.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The relevant statements have been introduced around the `numElements` assignments, optimizing their compiler-generated gas cost as originally intended.

STE-03C: Inexistent Avoidance of Bit Masking (Stamp Missing)

Type	Severity	Location
Gas Optimization	Informational	Set.sol:L119-L121, L134-L135

Description:

The `Set::remove` function will attempt to minimize gas costs by instructing the compiler to overwrite a storage slot in full instead of overwriting sub-sections of it which entail compiler-generated bit masking.

While the first highlighted instance correctly achieves this by "redundantly" re-assigning the `stamp` thus instructing the compiler that we need a fresh value for it, the second referenced instance does not do this.

Example:

src/Set.sol

SOL

```sol
// write full slot at once to avoid SLOAD and bit masking
if (numElements == 1) {
    setStorage.numElements = 0;
    setStorage.firstElement = address(0);
    setStorage.stamp = DUMMY_STAMP;
    return true;
}

uint256 lastIndex;
unchecked {
```

Example (Cont.):

```sol
    lastIndex = numElements - 1;
}

// set numElements for every execution path to avoid SSTORE and bit masking when the element removed is
// firstElement
if (searchIndex != lastIndex) {
    if (searchIndex == 0) {
        setStorage.firstElement = setStorage.elements[lastIndex].value;
        setStorage.numElements = uint8(lastIndex);
    } else {
        setStorage.elements[searchIndex].value = setStorage.elements[lastIndex].value;
        setStorage.numElements = uint8(lastIndex);
    }
} else {
    setStorage.numElements = uint8(lastIndex);
}
```

Recommendation:

We advise the `stamp` to be assigned to in the second referenced instance as well, optimizing the first element's removal gas cost.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

An assignment of the `stamp` member has been introduced in the highlighted code segment, ensuring that the compiler efficiently generates the `setStorage` data update.

STE-04C: Inexistent Correlation of Constants

Type	Severity	Location
Code Style	Informational	Set.sol:L31, L42

Description:

The `2 ** 8` constant which is utilized to instantiate the length of the `elements` array does not depend on the `MAX_ELEMENTS` member although a relation should be enforced.

Example:

src/Set.sol

SOL

```sol
16  /// @title SetStorage
17  /// @notice This struct is used to store the set data.
18  /// @dev To optimize the gas consumption, firstElement is stored in the same
storage slot as the numElements
19  /// @dev so that for sets with one element, only one storage slot has to be
read/written. To keep the elements
20  /// @dev array indexing consistent and because the first element is stored
outside of the array, the elements[0]
21  /// @dev is not utilized. The stamp field is used to keep the storage slot non-
zero when the element is removed.
22  /// @dev It allows for cheaper SSTORE when an element is inserted.
23  struct SetStorage {
24      /// @notice The number of elements in the set.
25      uint8 numElements;
```

Example (Cont.):

```sol
    /// @notice The first element in the set.
    address firstElement;
    /// @notice The stamp of the set.
    uint88 stamp;
    /// @notice The array of elements in the set. Stores the elements starting from index 1.
    ElementStorage[2 ** 8] elements;
}

/// @title Set
/// @author Euler Labs (https://www.eulerlabs.com/)
/// @notice This library provides functions for managing sets of addresses.
/// @dev The maximum number of elements in the set is defined by the constant MAX_ELEMENTS.
library Set {
    error TooManyElements();
    error InvalidIndex();

    uint8 internal constant MAX_ELEMENTS = 10; // must not exceed 255
```

Recommendation:

We advise the `constant` declaration to be relocated outside the `Set` library, and the `SetStorage` structure to utilize the constant as the length of the `ElementStorage` member optimizing the maintainability of the codebase.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The code was updated to ensure a consistent maximum element limit is imposed across the contract via the `SET_MAX_ELEMENTS` constant which is configured at `10` in accordance to our recommendation.

STE-05C: Optimization of Interim Element Removal

Type	Severity	Location
Gas Optimization	Informational	Set.sol:L137, L144

Description:

An optimization can be achieved by caching the `setStorage.elements[lastIndex]` storage offset which is calculated twice redundantly whenever an element is removed in between the elements array.

Impact:

To note, the adjustment is applicable to all instances but should result in a gas optimization solely for the interim element case as the first-element removal will re-calculate the `setStorage.elements[lastIndex]` offset using a `keccak256` instruction on the same payload which has reduced gas costs at the EVM level. This is not the case for interim removals, which also calculate the `keccak256` offset of the `setStorage.elements[searchIndex]` entry.

Example:

src/Set.sol

SOL

```
130 // set numElements for every execution path to avoid SSTORE and bit masking when
the element removed is
131 // firstElement
132 if (searchIndex != lastIndex) {
133     if (searchIndex == 0) {
134         setStorage.firstElement = setStorage.elements[lastIndex].value;
135         setStorage.numElements = uint8(lastIndex);
136     } else {
137         setStorage.elements[searchIndex].value =
setStorage.elements[lastIndex].value;
138         setStorage.numElements = uint8(lastIndex);
139     }
```

Example (Cont.):

```sol
} else {
    setStorage.numElements = uint8(lastIndex);
}

setStorage.elements[lastIndex].value = address(0);
```

Recommendation:

We applied this optimization and observed a median decrease in gas cost and thus advise it to be applied.

Alleviation (577d1991de2aa6c1b0578e9e45d363e8d6799408):

The `setStorage.elements[lastIndex]` lookup is properly cached and re-used across its previous instances, optimizing the code's median gas cost as advised.

Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted if blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Circom language boasts that discerns it from other conventional programming languages.

Curve Specific

Circom defaults to using the BN128 scalar field (a 254-bit prime field), but it also supports BSL12-381 (which has a 255-bit scalar field) and Goldilocks (with a 64-bit scalar field). However, since there are no constants denoting either the prime or the prime size in bits available in the Circom language, some Circomlib templates like `Sign` (which returns the sign of the input signal), and `AliasCheck` (used by the strict versions of `Num2Bits` and `Bits2Num`), hardcode either the BN128 prime size or some other constant related to BN128. Using these circuits with a custom prime may thus lead to unexpected results and should be avoided.

Code Style

In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a toplevel variable in the circuit.

Mathematical Operations

This category is used when a mathematical issue is identified. This implies an issue with the implementation of a calculation compared to the specifications.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

Privacy Concern

This category is used when information that is meant to be kept private is made public in some way.

Proof Concern

Under-constrained signals are one of the most common issues in zero-knowledge circuits. Issues with proof generation fall under this category.

Severity Definition

In the ever-evolving world of blockchain technology, vulnerabilities continue to take on new forms and arise as more innovative projects manifest, new blockchain-level features are introduced, and novel layer-2 solutions are launched. When performing security reviews, we are tasked with classifying the various types of vulnerabilities we identify into subcategories to better aid our readers in understanding their impact.

Within this page, we will clarify what each severity level stands for and our approach in categorizing the findings we pinpoint in our audits. To note, all severity assessments are performed **as if the contract's logic cannot be upgraded** regardless of the underlying implementation.

Severity Levels

There are five distinct severity levels within our reports; `unknown`, `informational`, `minor`, `medium`, and `major`. A TL;DR overview table can be found below as well as a dedicated chapter to each severity level:

	Impact (None)	Impact (Low)	Impact (Moderate)	Impact (High)
Likelihood (None)	Informational	Informational	Informational	Informational
Likelihood (Low)	Informational	Minor	Minor	Medium
Likelihood (Moderate)	Informational	Minor	Medium	Major
Likelihood (High)	Informational	Medium	Major	Major

Unknown Severity

The `unknown` severity level is reserved for misbehaviors we observe in the codebase that cannot be quantified using the above metrics. Examples of such vulnerabilities include potentially desirable system behavior that is undocumented, reliance on external dependencies that are out-of-scope but could result in some form of vulnerability arising, use of external out-of-scope contracts that appears incorrect but cannot be pinpointed, and other such vulnerabilities.

In general, `unknown` severity level vulnerabilities require follow-up information by the project being audited and are either adjusted in severity (if valid), or marked as nullified (if invalid).

Additionally, the `unknown` severity level is sometimes assigned to centralization issues that cannot be assessed in likelihood due to their exploitation being tied to the honesty of the project's team.

Informational Severity

The `informational` severity level is dedicated to findings that do not affect the code functionally and tend to be stylistic or optimizational in nature. Certain edge cases are also set under `informational` vulnerabilities, such as overflow operations that will not manifest in the lifetime of the contract but should be guarded against as a best practice, to give an example.

Minor Severity

The `minor` severity level is meant for vulnerabilities that require functional changes in the code but tend to either have little impact or be unlikely to be recreated in a production environment. These findings can be acknowledged except for findings with a moderate impact but low likelihood which must be alleviated.

Medium Severity

The `medium` severity level is assigned to vulnerabilities that must be alleviated and have an observable impact on the overall project. These findings can only be acknowdged if the project deems them desirable behavior and we disagree with their point-of-view, instead urging them to reconsider their stance while marking the exhibit as acknowledged given that the project has ultimate say as to what vulnerabilities they end up patching in their system.

Major Severity

The `major` severity level is the maximum that can be specified for a finding and indicates a significant flaw in the code that must be alleviated.

Likelihood & Impact Assessment

As the preface chapter specifies, the blockchain space is constantly reinventing itself meaning that new vulnerabilities take place and our understanding of what security means differs year-to-year.

In order to reliably assess the likelihood and impact of a particular vulnerability, we instead apply an abstract measurement of a vulnerability's impact, duration the impact is applied for, and probability that the vulnerability would be exploited in a production environment.

Our proposed definitions are inspired by multiple sources in the security community and are as follows:

Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.