Security Audit Report

Neutron & Timewave Q2 2025:
Valence Protocol

Last Revised
2025/07/18

Authors:
Aleksandar Ignjatijevic, Carlos Rodriguez

Contents

Audit overview **2**
- The project . . . 2
- Scope of this report . . . 2
- Audit plan . . . 2
- Conclusions . . . 2

Audit Dashboard **3**
- Target Summary . . . 3
- Engagement Summary . . . 3
- Severity Summary . . . 3

System Overview **5**
- **Core components** . . . 5
- Execution flow . . . 6
- Assumptions . . . 7

Threat Model **9**
- General . . . 9
- Account system . . . 9
- Authorization system . . . 10
- Processor system . . . 14
- `OneWayVault.sol` . . . 17

Findings **24**
- target lacks zero-check in Account execute() function . . . 26
- Unexpected payable functions in OneWayVault . . . 27
- Lack of proper validation for withdraw request receiver . . . 28
- Malicious strategist could manipulate redemption rate to its own advantage . . . 29
- Unpausing vault does not make it usable in the event of stale rate . . . 30
- Unexpected payable function in LiteProcessor . . . 32
- Missing validation of redemption rate in contract initialisation . . . 33
- Stale rate auto-pause mechanism wastes gas for unlucky users . . . 34
- Malicious users can spam withdrawals with small amounts to overwhelm destination . . . 35
- VerificationGateway might be zero address when adding/removing registries . . . 36
- Requests will be piling up and take up contract storage . . . 37
- Differently sized arrays will cause panic in execute function . . . 38
- Processor functions are not protected by nonReentrant modifier . . . 39
- Deposit and mint functions are not protected by nonReentrant modifier . . . 40
- Using msgSender function in contracts that do not support meta-transactions . . . 41
- Miscellaneous code improvements . . . 42
- Recommendations for gas optimisation . . . 44

Appendix: Vulnerability classification **45**

Disclaimer **48**

Audit overview

The project

In June 2025, Timewave engaged Informal Systems↗ to to conduct a security audit of several competes of their Valence Protocol, a trust-minimized cross-chain execution environment.

Scope of this report

The audit focused on evaluating the correctness and security properties of:

- Several sub-systems of the Valence Protocol:
  - Account system
  - Authorization system
  - Processor system
- A ERC4626 tokenized one-way vault contract that enables deposits on one chain with withdrawal requests processed on another domain/chain.

Audit plan

The audit was conducted between June 11th, 2025 and July 8th, 2025 by the following personnel:

- Aleksandar Ignatijevic
- Carlos Rodriguez

Conclusions

The Valence Protocol demonstrates exceptional architectural design and implementation quality, showcasing a sophisticated approach to trust-minimized cross-chain DeFi operations. The codebase exhibits strong engineering principles with well-structured modular components including the Account system, Authorization framework, Processor system, and innovative One-Way Vault implementation. The protocol's integration of zero-knowledge proof verification through the `SP1VerificationGateway` represents cutting-edge cryptographic security practices, while the comprehensive separation of concerns between different system components demonstrates mature software architecture. The implementation of ERC4626 standards in the vault system and the thoughtful design of atomic and non-atomic execution models in the processor system reflect deep understanding of DeFi security requirements and operational flexibility needs.

Our audit identified two high-severity issues were discovered: the absence of zero-address validation in the `Account execute()` function's `_target` parameter, which could lead to failed transactions and potential fund locks, and unexpected payable functions in the `OneWayVault` contract that could result in unintended fund loss scenarios.

The audit also identified 5 medium-severity and 5 low-severity issues that, while not immediately exploitable, could impact protocol operations and user experience, along with 5 informational findings that provide recommendations for enhanced security practices and gas optimization opportunities.

Audit Dashboard

Target Summary

- **Type:** Protocol and implementation
- **Platform:** Solidity
- **Artifacts:**

```
solidity/
|-- src/
    |-- accounts/
    |   |-- Account.sol
    |   |-- BaseAccount.sol
    |-- authorization/
    |   |-- Authorization.sol
    |-- processor/
    |   |-- ProcessorBase.sol
    |   |-- LiteProcessor.sol
    |   |-- libs/
    |   |   |-- ProcessorErrors.sol
    |   |   |-- ProcessorEvents.sol
    |   |   |-- ProcessorMessageDecoder.sol
    |   |-- interfaces/
    |       |-- ICallback.sol
    |       |-- IProcessor.sol
    |       |-- IProcessorMessageTypes.sol
    |-- vaults/
    |   |-- OneWayVault.sol
    |-- verification
        |-- SP1VerificationGateway.sol
        |-- VerificationGateway.sol
```

Of `timewave-computer/valence-protocol`↗ at commit hash `113a188`↗.

Engagement Summary

- **Dates**: June 11th, 2025 → July 8th, 2025
- **Method**: Manual code review, Foundry testing, Quint modelling↗

Severity Summary

Finding Severity	Number
Critical	0

Finding Severity	Number
High	2
Medium	5
Low	5
Informational	5
Total	17

System Overview

The Valence Protocol is a unified development environment for trust-minimized cross-chain DeFi applications, enabling developers to build **Valence Programs**. These programs simplify cross-chain operations, offering extensibility and rapid deployment without requiring complex smart contracts or multisig setups.

Core components

In the next sections we go over the core architecture components of the protocol under scope, which are coloured green in the following diagram:

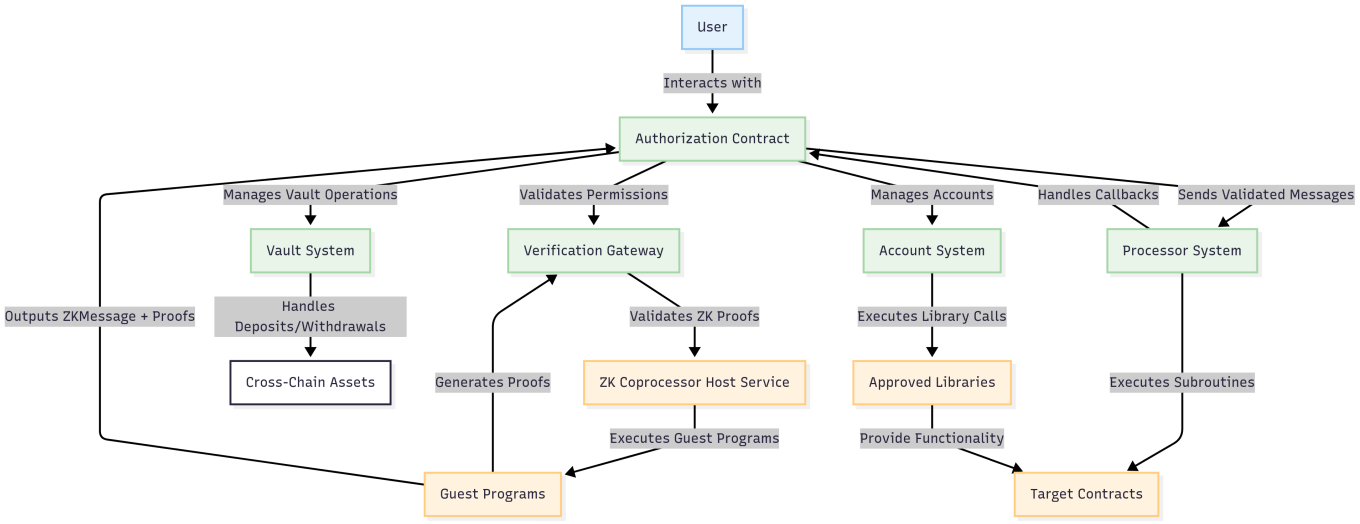

Figure 1: Architecture diagram

Account system

The account system in the Valence Protocol is designed to securely manage funds and enable controlled execution of operations through approved libraries. At its core is the `Account.sol` contract, which serves as an abstract base for account management. This contract introduces a mechanism for approving specific library addresses, ensuring that only trusted libraries can interact with the account's funds or execute operations. Additionally, `Account.sol` implements ownership control, allowing the account owner to manage approvals and execute privileged actions.

Building on this foundation, `BaseAccount.sol` provides a concrete implementation of the abstract `Account.sol`. It inherits the functionality of library approval and ownership control, making it ready for deployment and integration into the Valence Protocol. These account contracts act as secure containers for funds and execution logic.

Authorization & access control

This system enables secure and trust-minimized interactions within the Valence Protocol. The `Authorization.sol` contract serves as the entry point for managing permissions and submitting ZK proofs for verification. It supports two types of authorizations: **Standard authorizations**, which rely on address-based permissions, and **ZK authorizations**, which validate permissions using cryptographic proofs.

Standard authorizations allow specific addresses to interact with the system, providing access control for known entities such as administrators or trusted users. ZK authorizations validate permissions through cryptographic

proofs generated off-chain by the ZK Coprocessor Service and verified on-chain.

The `Authorization.sol` contract integrates with the `VerificationGateway.sol` and its specialized implementation, `SP1VerificationGateway.sol`, to perform cryptographic verification of ZK proofs. These gateways store verification keys for registered guest programs and use them to validate proofs submitted by authorized entities. Once a proof is verified, the `Authorization.sol` contract dispatches the associated `processorMessage` to the `Processor` contract for execution, ensuring that only authenticated and authorized actions are performed.

Processor system

This system is responsible for executing program subroutines and handling cross-chain messages. It consists of three main contracts: `ProcessorBase.sol`, `LiteProcessor.sol`, and `Processor.sol`. These contracts enable flexible and efficient execution of subroutines.

At the core is `ProcessorBase.sol`, which provides shared functionality for processors, including message handling and execution logic. It supports atomic and non-atomic execution models, allowing flexibility in how subroutines are processed. Atomic execution ensures that either all functions within a subroutine succeed or none are executed, providing strong guarantees for critical operations. Non-atomic execution, on the other hand, allows partial completion of subroutines, with individual retry logic applied to failed functions. Building on this foundation, `LiteProcessor.sol` offers a lightweight implementation designed for immediate execution of subroutines without queuing.

The Processor system integrates seamlessly with the Authorization contract, which manages the addition of message batches to the queues and controls the processor's state.

Verification system

This system enables the validation of zero-knowledge (ZK) proofs and it consists of two main contracts: `VerificationGateway.sol` and its specialized implementation, `SP1VerificationGateway.sol`.

At its core, `VerificationGateway.sol` serves as an abstract base for verification gateways, providing foundational functionality for managing verification keys (VKs) and performing proof validation. It stores VKs associated with specific registry IDs, which are used to verify proofs submitted by authorized entities. The contract also supports updates to the verifier address and domain verification key. Building on this foundation, `SP1VerificationGateway.sol` implements the verification logic specific to the SP1 zkVM. It integrates with the SP1 verifier to validate proofs against the stored VKs.

This works in tandem with the Authorization contract, which delegates proof validation to the `VerificationGateway.sol`. Upon successful verification, the Authorization contract proceeds with executing the associated processor message.

One way vault

The `OneWayVault.sol` contract implements the ERC4626 tokenized vault standard to enable deposits and withdrawals of assets. It supports one-way withdrawal requests, allowing assets deposited on one chain to be withdrawn on another. The contract also includes mechanisms for fee distribution, ensuring that both the platform and strategist receive their respective shares of collected fees.

To enhance security and operational control, the vault includes pausability features, allowing the owner or strategist to pause operations in case of emergencies or stale redemption rates. Additionally, the contract enforces deposit caps to prevent excessive asset accumulation.

Execution flow

The following diagram shows the complete interaction patterns within the Valence Protocol, showing how users interact with the system through both standard and ZK authorization flows, as well as vault operations for cross-

chain asset management.

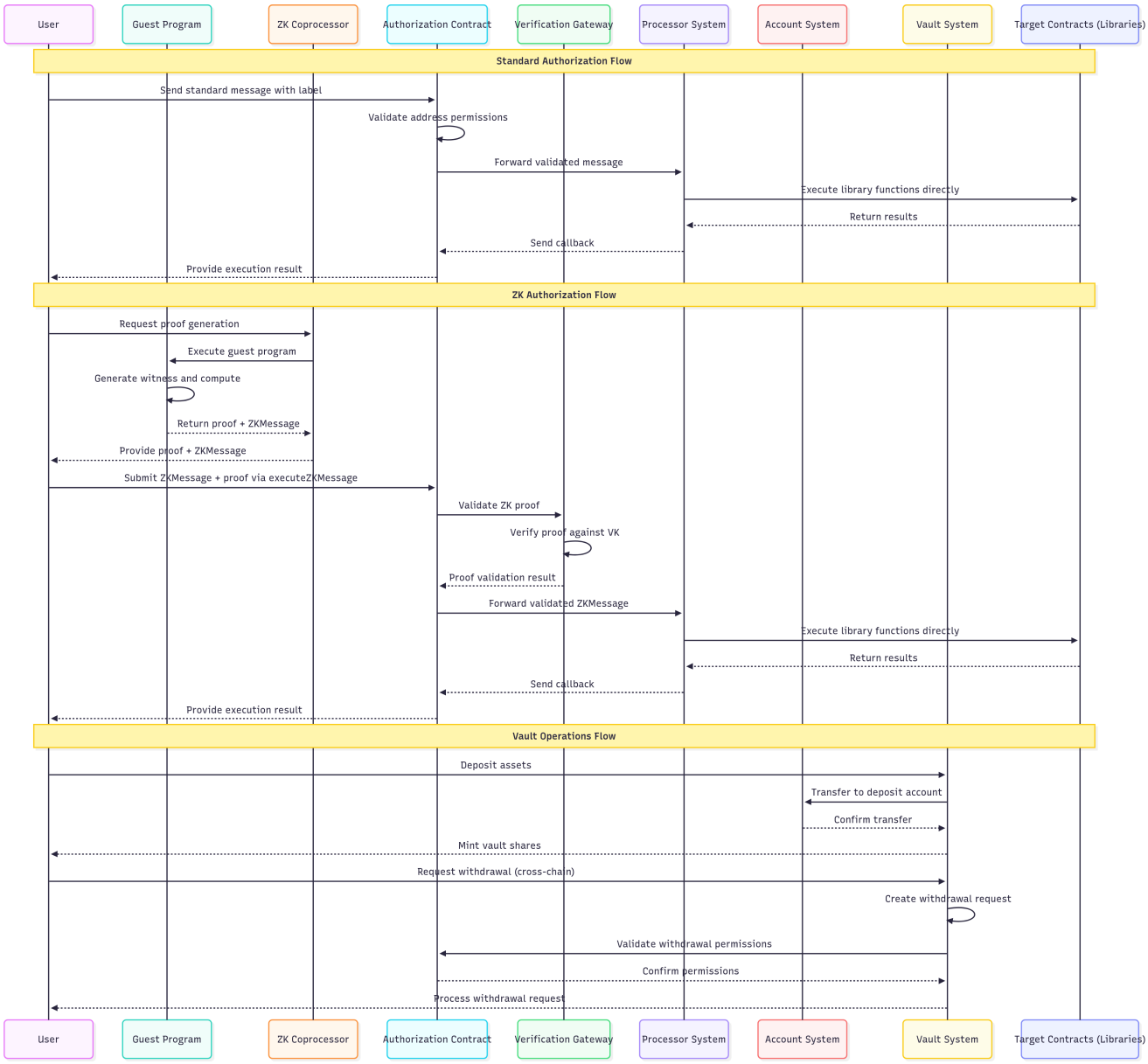

Figure 2: Execution flow

Assumptions

In order to understand the system we had several assumptions:

- Processor's authorized addresses - our assumption is that the authorized addresses are addresses of authorized contracts that are to be trusted. However, since they will be calling external contracts and perform certain logic, we still left some room for potential bugs in the code that could lead to authorized addresses performing certain actions in ways that could lead to bugs.
- Strategist - our assumption is that the strategist is a trusted party, as well as the vault owner. However, there are still ways to gain control over trusted party's keys or to act maliciously so we left some room for that.
- Approved libraries - approved libraries are a part of accounting system and our assumption is that they are trusted by the owner of an account. However, since those libraries are out of our scope, we cannot rule out bugs

in that code that can lead to bugs in account system that we are inspecting.

Threat Model

General

- Property GEN-01: Events are emitted for all state changes

  Conclusion: The contracts already emit events on several state changes (pause/unpause, when a withdrawal request is created, when the vault configuration is updated, etc), however there are other state changes that could benefit from emitting an event, and we list some recommendatiosn in finding "Miscellaneous code improvements".

- Property GEN-02: All functions are marked with appropriate visibility

  Conclusion: Function visibility appears appropriate for almost all functions. Some of the functions that are marked as public are not used internally. We have mentioned that in finding "Miscellaneous code improvements".

- Property GEN-03: Checks-Effects-Interactions pattern is applied consistently

  Conclusion: Overall the contracts adhere to the CEI pattern consistently and we would only recommend to apply it in function `sendProcessorMessage()` of `Authorization`, where the `executionId` is incremented↗ after the external call to `processor.execute(message)`.

- Property GEN-04: Division operations are protected against zero denominators

  Conclusion: As reported in finding "Missing validation of redemption rate in contract initialisation", we recommend checking that input parameter `startingRate`↗ is not zero.

- Property GEN-05: Rounding errors are not accumulated in ways that can be exploited

  Conclusion: We haven't identified ways in which rounding error accumulation could be exploited.

Account system

Definitions

Approved libraries: Mapping to track approved library addresses. Maps library address to approval status (true = approved, false = not approved).

Properties

- Property ACC_AUTH-01: Admin operations can only be performed by contract owner

  - Threat a: Other address than contract owner can add or remove approved libraries

    Conclusion: The property holds. Due to placed `onlyOwner` modifier, only owner is able to interact with `approveLibrary()` and `removeLibrary()` functions.

- Property ACC_EXEC-01: Execute function can only be called by owner or by approved libraries

  - Threat a: Malicious address can call execute and drain the contract

    Conclusion: The property holds. Check placed here↗ ensures that only approved libraries and contract owner can call `execute()`.

  - Threat b: Error data is not returned in the proper way

    Conclusion: During the audit, development team has run into an issue around reverting and returning `returnData` if `.call` returned no data. Development team fixed this issue in PR#404↗.

- Property ACC_APL-01: Once library is removed it cannot call `execute` function

  Conclusion: The property holds. Only owner can remove the library and the state is updated straight away by deleting entry from the map which will set the value to `false`.

- Property ACC_NFT-01: Contract has to be able to receive ERC721

  Conclusion: The property holds. `Account` implements `IERC721Receiver` interface and implements `onERC721Received` function which returns `this.onERC721Received.selector` therefore ensuring that `ERC721` can be transferred to the account.

- Property ACC_ETHER-01: Contract has to be able to receive ether needed for approved libraries

  Conclusion: The property holds. The contract implements `receive() external payable {}` which allows contract to receive native tokens.

Authorization system

Definitions

Guest program: A user-developed application that runs on the ZK coprocessor, consisting of a controller (a Wasm-compiled Rust program) and a ZK circuit.

Registry: A unique identifier (`uint64`) that:

- Links a specific guest program to its verification key
- Controls which addresses can submit proofs for that program
- Acts as an access control mechanism in the Authorization contract
- Each registry has its own verification key and set of authorized users

ZK coprocessor: An off-chain service responsible for running and managing guest programs in a secure and isolated environment. By leveraging an underlying zkVM, such as SP1, the service generates cryptographic proofs that attest to the correctness of program execution. Once proofs are generated, the service makes them readily available for submission to the blockchain.

Coprocessor root hash: The first 32 bytes of a ZK message that represent a cryptographic commitment to the current state of the ZK coprocessor.

Domain: A specific blockchain environment defined by: 1) The blockchain name (e.g., Neutron, Osmosis) 2) The execution environment (e.g., CosmWasm, EVM) 3) The bridge type used for cross-chain communication.

Domain proof: A cryptographic proof of the state of a specific domain, stored in the SMT of the ZK coprocessor. It is used to ensure that the state of the domain used in the computation of the guest program in the ZK coprocessor matches the proven state from the other chain.

Properties

- Property AUTH-01: Admin operations can only be performed by contract owner

  - Threat a: Other address than contract owner can update the verification gateway

    Conclusion: The property holds. The function `updateOwner()`↗ is decorated with the`onlyOwner` modifier (implemented in `Ownable.sol` of `openzeppelin-contracts@v5.2.0`).

  - Threat b: Other address than contract owner can update the processor

    Conclusion: The property holds. The function↗ `updateProcessor()` is decorated with the`onlyOwner` modifier.

  - Threat c: Other address than contract owner can add/remove authorisation configurations

    Conclusion: The property holds. The functions `addStandardAuthorizations()`↗ and `removeStandardAuthorizations()`↗ are both decorated with the `onlyOwner` modifier.

  - Threat d: Other address than contract owner can add/remove admin addresses

    Conclusion: The property holds. The functions `addAdminAddress()`↗ and `removeAdminAddress()`↗ are both decorated with the `onlyOwner` modifier.

  - Threat e: Other address than contract owner can add/remove registries with authorised senders and verification keys

    Conclusion: The property holds. The functions `addRegistries()`↗ and `removeRegistries()`↗ are both decorated with the `onlyOwner` modifier.

- Property AUTH-02: Only callbacks initiated by the processor must be executed

  - Threat a: Callbacks are triggered by other addresses than the processor

    Conclusion: The property holds. The function `handleCallback()`↗ is decorated with the `onlyProcessor` modifier↗.

- Property AUTH-03: For standard authorisation a strict binding between authorized addresses, target contracts, and their allowed function calls is maintained, ensuring messages can only be executed by permitted actors within their defined scope

  Standard authorizations are stored by calling function `addStandardAuthorizations()`↗ (only owner allowed). They consist of an array of addresses and an array of `AuthorizationData`↗, both indexed by a `label`. The addresses specify what accounts are authorised to send a message for a specific label. The array of `AuthorizationData` specifies the contract addresses and functions that are authorised by be executed for a specific label.

  - Threat a: A different function than authorised is executed by manipulating the function selector or call hash

    Conclusion: The property holds. Authorised accounts can execute function `sendProcessorMessage()`↗ with a specific `label` and an ABI-encoded `message` of type `ProcessorMessage`↗. The message includes itself a field that encodes a message of one of the possible values of the `ProcessorMessageType`↗ enumeration. Both `SendMsgs` and `InsertMsgs` types include information about the contract addresses (in the `address` field of `AtomicFunction` and `NonAtomicFunction`) and functions that should be executed (encoded in each of the `messages` of `InsertMsgs` and `SendMsgs`). These values are then checked in function `_checkAddressIsAuthorized()`↗ against the stored values for the specific label:

    → It is checked that the account that initiated the call is authorised↗ (either explicitly or implicitly is `address(0)` is stored).
    → It is checked that the contract address that should be executed is authorised↗.
    → It is checked whether the function selector↗ or the hash of the function call↗ of the function that should be executed is authorised.

  - Threat b: Authorisation data from one label is reused for another label

    Conclusion: The property holds. Standard authorizations are stored in a map indexed by the label↗, therefore labels are unique. If the owner executes function `addStandardAuthorizations()` with duplicate labels in the `_labels` input parameter, then only the information for the last label of the same name will be stored.

- Property AUTH-04: For ZK authorisation a strict binding between registry, authorised senders, and target authorization contract must be maintained, ensuring messages can only be executed by permitted actors within their defined scope

  ZK authorisations are stored by calling function `addRegistries()` (only owner allowed). Each consists of a registry ID, an array of addresses, and a verification key (the cryptographic key used to verify that proofs are valid for a given guest program and its inputs). The addresses specify what accounts are authorised to send a message for a specific registry ID. When adding registries, it is also possible to specify if block number validation should be performed when executing a ZK message. If enabled, ZK messages will only be executed if the block number of the domain when the ZK message was generated is greater than the block number of the domain for the previous ZK message that was executed.

  Therefore each registry is bound to a specific guest program by the verification key that is associated with it, and a list of accounts that are allowed to execute ZK messages generated by running the guest program on the ZK coprocessor. ZK messages consists of the registry ID, the block number of the domain when the message was created, the address of the authorization contract that the message is for, and the actual message of type `ProcessorMessage` to be processed and that was proven on the ZK coprocessor.

  - Threat a: A message associated with a verification key is executed for a different registry

    Conclusion: The property holds. Every ZK message includes a reference to the associated registry. The registry is used to fetch the verification key in function `verify()`. Therefore the registry connects with the verification key that is used to verify the message and domain proofs. If the ZK message references a registry that is different from the one that it should, then the verification will fail since the wrong verification key will be used.

  - Threat b: If permissioned access is enabled, a message from an unauthorised sender is executed

    Conclusion: The property holds. When a list of addresses is associated with a registry, the address of the account that initiates the call to `executeZKMessage()` is checked against the authorised addresses. If the message sender is not authorised, the function reverts.

  - Threat c: A message is executed through an authorisation contract that is not the intended one

    Conclusion: The property holds. If the ZK message includes a non-zero address for `authorizationContract`, then function `executeZKMessage()` checks that the address matches the contract's address.

  - Threat d: If block number validation is enabled, block number validation is bypassed

    Conclusion: The property holds. If block number validation is enabled for a registry, then in function `executeZKMessage()` the `blockNumber` of the ZK message will be checked against the latest block number stored and the function reverts if the block number is not greater. The `blockNumber` cannot be tampered with, because the verification of the input `_proof` for `_message` would fail.

  - Threat e: An un-authorised user changes the verification key associated with an exiting registry

    Conclusion: The property holds. The verification key for a registry is added by calling function `addRegistry()` of `VerificationGateway` contract. The verification key for the registry is keyed by the address of the account that initiates the call. In the expected flow, the `Authorization` contract will call `addRegistry()` (and thus the address of the contract will be `msg.sender`). There is no other way to update the verification key for the registry.

- Property AUTH-05: The coprocessor root hash present in both main and domain messages must match, ensuring that all proven computations and state transitions are based on the same underlying state context

  - Threat a: Messages generated with coprocessor state `a` are executed with domain messages generate with a different coprocessor state `b`

Conclusion: The property holds. The coprocessor root hash is included in the message and domain message when they are generated by the ZK coprocessor and compared in function `executeZKMessage()` ↗. If they don't match, then call reverts.

- Threat b: A domain proof for block `h` is used with a message proof for `h'`, or vice versa

  Conclusion: The property holds. Both the `_domainMessage` and `_domainProof` inputs to function `executeZKMessage()` are generated by the ZK coprocessor when it executes the guest program. Both of them are generated for a specific block of the domain. If a domain message for block `h'` is used together with a domain proof for block `h`, then the verification here ↗ would fail and the function would revert.

- Property AUTH-06: Messages and their associated ZK proofs cannot be replayed or executed out of order
  - Threat a: Messages are replayed for the same registry of an authorization contract

    Conclusion: It is by design possible to execute the same message both for standard authorization and ZK authorization. For ZK authorization the only check that would prevent the same exact message to execute would be the validation of block number execution ↗ if the value of mapping `validateBlockNumberExecution` is true for the given registry key.

  - Threat b: Messages are replayed across different registries for the same authorization contract

    Conclusion: The property holds. The registry that links to the verification key used to verify the ZK message is included in the ZK message ↗, and the `_proof` generated by the ZK coprocessor would fail verification ↗ if the ZK message is tampered with a different registry, even if the account initiating the transaction passes the authorization validation for the different registry ↗ (i.e., the account is also authorised for the different registry).

  - Threat c: Messages are replayed across different authorization contracts

    Conclusion: The property holds. The address of the authorization contract expected to execute the ZK message is part of the ZK message ↗, and the `_proof` generated by the ZK coprocessor would fail verification ↗ if the ZK message is tampered with a different authorization contract address, even if the check ↗ on the `Authorization` contract is bypassed.

- Property AUTH-07: For any given `executionId`, neither the processor nor the callback must be executed for the associated message multiple times
  - Threat a: If validation of latest executed block is enabled, block numbers of ZK messages are not strictly increasing

    Conclusion: The property holds. If block number validation is enabled for a specific registry, the block number in the ZK messages must be increasing ↗. Only possibility to change this would be to set the value of the mapping `validateBlockNumberExecution` for a specific registry to `false`, but such functionality is not implemented, and the only way to achieve something similar would be to delete the registry by calling function `removeRegistries()` ↗, but this would effectively disable the execution of any ZK messages for the specific registry.

  - Threat b: If validation of latest executed block is enabled, latest executed block is not tracked accurately

    Conclusion: The property holds. When executing a ZK message, the latest block number executed is updated if block number validation is enabled ↗.

  - Threat c: There is no check on `executionId` to prevent that processor execution is triggered multiple times for the same message

    Conclusion: The development team informed us that the use of `executionId` is purely for informational purposes and to keep track of executions, thus it is not used to prevent message replay.

- Threat d: There is no check on `executionId` to prevent that callback is triggered multiple times for same message

  Conclusion: The development team informed us that the use of `executionId` is purely for informational purposes and to keep track of executions, thus it is not used to prevent message replay.

Processor system

Definitions

Authorization contract: Authorization contract is address of the deployed sub-authorization contract containing access control elements. Authorization contract is the only address allowed to deliver messages to this processor. Stored as bytes32 to handle cross-chain address representations.

Mailbox: Hyperlane's Mailbox contract, but this will not be used in the actual use-case. The only address allowed to deliver messages to this processor

Origin domain: The origin domain ID for sending the callbacks via Hyperlane, but it will not be used in the actual use-case.

Authorized addresses: The addresses authorized to interact with the processor contract directly.

Paused: State variable of the contract which indicates wether the processor can process messages or not.

Properties

- Property PROC_PAUSE-01: Paused processor should revert and not process messages
  - Threat a: Paused processor continues to process messages leading to unexpected behavior
    Conclusion: The property holds. Messages are handled with `_handleMessageType()` function which will check that processor is not paused when `SendMsg` type of message arrives.
- Property PROC_PAUSE-02: Not paused processor should not revert messages in a way that leads to confusion about paused state
  - Threat a: Not paused processor reverts messages in away that indicates that it is paused
    Conclusion: The property holds. Paused processor will revert with error `ProcessorErrors.ProcessorPaused()` which undoubtedly indicates that the processor has been paused. Other possible reverts do not return the same type of error.
- Property PROC_MSGD-01: Unknown message types should not get processed by the processor
  - Threat a: Processing unknown messages leads to unexpected behavior
    Conclusion: The property holds. `_handleMessageType()` function implements `if-else` statement which ensures that every type that is not supported (`Pause`, `Resume` or `SendMsg`) results in revert with error `ProcessorErrors.UnsupportedOperation()`.
- Property PROC_MSGD-02: `LiteProcessor` must process only messages of type Pause, Resume and SendMsgs
  - Threat a: Processing of `IProcessorMessageTypes.InsertMsgs` or `IProcessorMessageTypes.EvictMsgs` leads to unexpected behavior
    Conclusion: The property holds. `_handleMessageType()` function implements `if-else` statement which ensures that every type that is not supported (`Pause`, `Resume` or `SendMsg`) results in revert with error `ProcessorErrors.UnsupportedOperation()`.
- Property PROC_ATOM-01: Atomic execution ensures that either all of messages have to be executed successfully or everything should be reverted
  - Threat a: Message executions result in silent failures that went undetected therefore breaking the atomicity
    Conclusion: The property holds. All failures are accounted for and are reverting (here↗, here↗ and here↗). These reverts are caught↗ in `try-catch` and regarded as an `err` which means that

`IProcessor.SubroutineResult({succeeded: false, expired: false, executedCount: 0, errorData: err})` will be returned.

- Threat b: Systematic failures prevent messages from being executed at all
  Conclusion: **The property is doesn't hold.** If `processorMessage` contains differently sized arrays `processorMessage.messages.length < processorMessage.subroutine.functions.length` the following panic may happen: `panic: array out-of-bounds access (0x32)`. Based on threat inspeciton, we have written the finding "Differently sized arrays will cause panic in execute function".
- Threat c: Subroutine results are not representing the actual execution state
  Conclusion: The property holds. This was discussed in "Property PROC_CLBCK-02: Callbacks are built to represent the adequate situation".
- Threat d: Improper implementation of try-catch pattern
  Conclusion: The property holds. `try-catch` pattern is implemented in the correct way. There are no silent failures. Every `revert` that happens in `_executeAtomicSubroutine()` function will be caught in `catch` block and result in `IProcessor.SubroutineResult({succeeded: false, expired: false, executedCount: 0, errorData: err})` being returned. Panics should not be caught because it will make behaviour inconsistent across non-atomic and atomic operations.
- Threat e: Error data is not returned in the proper way
  Conclusion: During the audit, development team has run into an issue around reverting and returning `returnData` if `.call` returned no data. Development team fixed this issue in PR#404↗.
- Threat f: Number of executed subroutines is not the exact number of executed subroutines
  Conclusion: The property holds. `executedCount` will either be `subroutine.functions.length` or `0` if there has been an `revert` in `_executeAtomicSubroutine()` function.
- Threat g: Messages are executed on the address which is not a contract
  Conclusion: The property holds. Check placed here↗ ensures that the contract is the target.

- Property PROC_ATOM-02: Subatomic execution ensures that messages are to be executed until one of them fails
  - Threat a: Systematic failures prevent messages from being executed at all
    Conclusion: **The property doesn't hold.** If `processorMessage` contains differently sized arrays `processorMessage.messages.length < processorMessage.subroutine.functions.length` the following panic may happen: `panic: array out-of-bounds access (0x32)`. Based on threat inspection, we have written a finding "Differently sized arrays will cause panic in execute function".
  - Threat b: Subroutine results are not representing the actual execution state
    Conclusion: The property holds. This was discussed in "Property PROC_CLBCK-02: Callbacks are built to represent the adequate situation".
  - Threat c: Error data is not returned in the proper way
    Conclusion: The property holds. If error occurs, other than `panic`, it will be caught with `catch (bytes memory err)`. However, if empty `revert` happens during `call`, returned error will be `""` which will later used as `SubroutineResult.errorData`.
  - Threat d: Number of executed subroutines is not the exact number of executed subroutines
    Conclusion: The property holds. If `.call` succeeds `executedCount` is increased by one, while if one fails `succeeded` is marked as `false` and error is used as `errorData`.
  - Threat e: Messages are executed on the address which is not a contract
    Conclusion: The property holds. Check placed here↗ ensures that the contract is the target.
- Property PROC_CLBCK-01: Callbacks should only be returned if the sender is authorization contract
  - Threat a: Callbacks are returned to an arbitrary address
    Conclusion: The property holds. The address that initiated the `execute()` call is used when callback is built as the callback destination.
  - Threat b: Callbacks are returned to an address that is not authorization contract
    Conclusion: The property holds. The address that initiated the `execute()` call is used when callback is built

as the callback destination. If the sender is not the contract, callback will not be sent due to check placed here↗.

- Threat c: Callbacks intended for the authorization contract does not arrive at the destination
  Conclusion: The property holds.. It is possible that the atomic or subatomic `calls` spend certain amount of gas which can lead to less gas needed to execute sending callback to the callback destination. However, this will revert the whole transaction.

- Property PROC_CLBCK-02: Callbacks are built to represent the adequate situation
  - Threat a: Callback build does not represent the adequate result of operations
    → Threat analysis:
    Callbacks are returned with `result` which is supposed to encapsulate result of atomic or subatomic operation. Result consists of:

```
{
    succeeded: bool,
    expired: bool,
    executedCount: uint256,
    errorData: bytes
}
```

    Assumptions are:

    1) if the subroutine achieved its goal - `succeeded` must be `true`,
    2) if messages are deemed expired - `expired` must be `true`,
    3) `executedCount` must represent the exact number of subroutines that has finished successfully,
    4) in case of error - `errorData` should represent that error.

    → Threat inspection:
    Scenario 1 - Messages are expired - OK
    In that event result will be `{succeeded: false, expired: true, executedCount: 0, errorData: bytes("")}` adequately encapsulating the state of the execution. Callback will be built with adequate data since `executionResult` will be set to `IProcessor.ExecutionResult.Expired`.
    Scenario 2 - Atomic - OK
    `_executeAtomicSubroutine()` will return adequate number of executed routines which is the total number of `atomicSubroutine.functions` or it will result in `revert` which will be caught `_handleAtomicSubroutine()`'s `try/catch` block. First subscenario results in adequate data because `success` will be `true` and `executedCount` will be `atomicSubroutine.functions.length`. Second subscenario also yields correct data.
    Scenario 3 - Subatomic - OK
    `_handleNonAtomicSubroutine()` does the following: uses `succeeded` as flag that either the `nonAtomicSubroutine.functions[i].contractAddress` is not a contract or that the `call` towards it failed. Based on that `errorData` is updated as well. `executedCount` is updated only if `call` is a success.

    Conclusion: Based on our threat inspection, we confirm that the property holds.

- Property PROC_CLBCK-03: Callbacks must correspond to unique `executionId`
  - Threat a: `executionId` collision when callbacks are executed
    Conclusion: The property holds. `Authorization` contract increments the `executionId` when a message is executed, so that next the message uses the updated value. Every message will have a unique `executionId` that will be passed on to the callback data when it's constructed in `ProcessorBase`.

- Property PROC_CLBCK-04: Callback emitted events correspond to the actual state
  - Threat a: `CallbackSent` event is emitted without callback being sent
    Conclusion: The property holds. As discussed in "Threat CLBCK-01.b: Callbacks intended for the authorization contract does not arrive at the destination" callbacks can end up without being sent. However, even though `success` flag is ignored the contract will revert and event `ProcessorEvents.CallbackSent` won't be emitted.

- Property PROC_AUTH-01: Only owner of processor contract can update authorized addresses
  - Threat a: Malicious users manipulate authorized addresses to gain elevated access
    Conclusion: The property holds. `onlyOwner` modifier placed here↗ and here↗ ensures that only the contract owner can edit authorized addresses mapping.
- Property PROC_AUTH-02: Only authorized addresses can perform `execute`
  - Threat a: Unauthorized addresses perform `execute` with malicious calldata leads to unexpected behavior
    Conclusion: The property holds. Due to check placed (here↗), if `msg.sender` is not present in `authorizedAddresses` mapping, `_handleMessageType` function will result in revert with error `ProcessorErrors.UnauthorizedAccess()`. This means that `calldata` can only be supplied by authorized address.
- Property PROC_SMSG-01: Expired messages should not be processed
  - Threat a: Expired messages get processed as if they are not expired
    Conclusion: The property holds. There↗ is a check which ensures that if the messages are expired not to process them.
- Property PROC_SMSG-02: Messages are deemed expired in a correct way
  - Threat a: `block.timestamp` manipulation by malicious validators can lead to messages failing, or expired messages being executed
    Conclusion: The property holds. Even though the `block.timestamp` could be manipulated, it would be manipulated by a small margin (~15 seconds), which would not have much of an effect to message execution.
- Property PROC_SMSG-03: Messages of type `Atomic` are executed in atomic way, and messages of type `NonAtomic` are executed in a subatomic way
  - Threat a: Atomic messages are executed in subatomic way
    Conclusion: The property holds. Placed `if-else` statement here↗ ensures that if decoded `IProcessorMessageTypes.SendMsgs` has `subroutine.subroutineType` equal to `IProcessorMessageTypes.SubroutineType.Atomic` it will call `_handleAtomicSubroutine` (here↗).
  - Threat b: Subatomic messages are executed in atomic way
    Conclusion: The property holds. Placed `if-else` statement here↗ ensures that if decoded `IProcessorMessageTypes.SendMsgs` does not have `subroutine.subroutineType` equal to `IProcessorMessageTypes.SubroutineType.Atomic` it will call `_handleNonAtomicSubroutine` (here↗).

`OneWayVault.sol`

Definitions

Redemption rate: A parameter controlled by the strategist that determines the conversion ratio between shares and assets. The redemption rate can be updated to adjust the economics of cross-chain transfers.

Asset: An ERC20 token that users can deposit into the vault. The asset represents the actual underlying token that is being transferred cross-chain. Assets are held in a separate deposit account specified in the vault's configuration.

Share: A tokenized representation of a user's deposit in the vault, conforming to the ERC4626 standard. Shares represent a claim on the underlying assets and are minted when assets are deposited and burned when withdrawal requests are created. The value of shares relative to assets is determined by the redemption rate.

Relationship:

```
// Assets to Shares
shares = assets * ONE_SHARE / redemptionRate

// Shares to Assets
assets = shares * redemptionRate / ONE_SHARE
```

Total assets: The theoretical total value of assets that all shares represent, calculated by converting the total supply of shares to assets using the current redemption rate. This is **NOT** the actual balance of assets in the deposit account.

Total supply: The total number of share tokens currently in existence. This represents the sum of all shares that have been minted through deposits minus all shares that have been burned through withdrawal requests. Total supply is automatically tracked by the ERC20 implementation.

Relationship:

```
totalAssets = totalSupply * redemptionRate / ONE_SHARE
```

Where:

- `ONE_SHARE = 10 ** decimals()`
- `redemptionRate` is controlled by the strategist

Properties

General

- Property VAULT-01: All configuration parameters of the vault must be valid
  - Threat a: Deposit account is set to zero address

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - Threat b: Strategist account is set to zero address

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - Threat c: Platform account is set to zero address

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - Threat d: Strategist account intended for fee distribution is set to zero address

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - Threat e: Deposit fees in basis points is larger than 100%

    The property holds. Check placed here↗ prevents this from happening.
  - Threat f: Withdrawal fees in basis points is larger than 100%

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - Threat g: Strategist's share of total fees in basis points is larger than 100%

    Conclusion: The property holds. Check placed here↗ prevents this from happening.
  - ***Caveat***: *It is possible for `config.feeDistribution.strategistAccount` to be different than `config.strategist` if that is intended behavior.*
- Property VAULT-02: Only contract owner or strategist can pause/unpause the vault

  Conclusion: The property holds. Pause and unpause are protected by `onlyOwnerOrStrategist` modifier↗ (here↗ and here↗) which ensures that only owner or strategist are the ones who can pause or unpause the contract. However, there is a potential issue with pausing and unpausing mechanisms. We have written a finding about that in "Unpausing vault does not make it usable in the event of stale rate".
- Property VAULT-03: The redemption rate can only be modified by the strategist and must be non-zero

Conclusion: The property holds. Modifier placed here↗ ensures that only strategist can update redemption rate and check placed here↗ prevents new rate being zero.

- Property VAULT-04: Redemption rate updates must be atomic and distribute accumulated fees

  Conclusion: The property holds. Updating rate is done after fees have been already distributed within the same transaction. As there are no external calls, the update is executed atomically.

- Property VAULT-05: The total distributed fees between platform and strategist must exactly equal the accumulated fees

  Conclusion: **The property doesn't hold.** Total fees denominated in assets are distributed to the strategist and the platform. However, due to the rounding strategist and platform might get less shares to be minted. If this is the case, then some dust will be left in `_feeAccruedInAsset` and that will basically be overwritten with zero (here↗). This amount can accumulate over time and present a certain portion of the fees. But in order to get exact number of assets that are covered with shares, the vault would have to recalculate it and therefore spend gas that would eat up that dust, so in the end there would be no economic incentive not to annul the `_feesAccruedInAsset`.

```
uint256 platformAssets = _feesAccruedInAsset - strategistAssets;
```

- Property VAULT-06: Fees must be split between platform and strategist according to the configured rate

  Conclusion: The property holds. Strategist's allocated fees are calculated using `feesAccruedInAsset.mulDiv(feeDistribution.strategistRatioBps, BASIS_POINTS, Math.Rounding.Floor` formula.

- Property VAULT-07: Asset/share conversions must maintain value conservation across any sequence of operations and solely be based on current redemption rate, with any precision loss strictly favouring the protocol rather than users

  - Threat a: Precision loss in the share-to-asset conversion is exploited through many small withdrawals instead of one large withdrawal

    Conclusion: The property holds. When withdraw is split into smaller withdraws, sum of withdrawal fee for a group of small withdrawals will be bigger or equal to the withdrawal fee of the big withdrawal created. This would mean that total sum of net assets for all withdrawals will be less than net assets for one big withdrawal, leading to less post fee shares. Also, due to ceil rounding when `sharesToBurn` is calculated, total amount of shares to burn for all small withdrawals would be bigger or equal to the shares to burn for one big withdrawal.

  - Threat b: Rounding differences between `_convertToAssets` and `_convertToShares` are exploited

    Conclusion: The property holds. Rounding differences are done in the favour of the protocol. Several scenarios can be identified:

    → Rounding during `deposit()`: first rounding would allow depositing more assets, but then during `previewDeposit` the result is floored, so it will cancel it.
    → Rounding during `mint()`: getting maximum amount allowed to mint is done through rounding twice (using floor), and assets needed to be transferred and fee to be paid is done with ceiling.
    → Rounding during `withdraw()`: getting maximum amount available to withdraw is done using floor rounding, limiting the amount. Withdrawal fee is calculated using ceiling, as well as shares to be burned for the amount of assets wanted to be withdrawn.
    → Rounding during `redeem()`: Gross amount of assets is calculated using floor rounding, and withdrawal fee is calculated using ceil rounding, meaning that net assets would be a little less. Then when post fee shares is calculated, ceil is used meaning that it would be rounded up. However, off-chain component calculates amount while rounding down.

  - Threat c: Multiple deposit/withdrawal cycles could accumulate rounding errors to extract more assets than originally deposited

    Conclusion: The property holds. Sequence would go like this: deposit gives you less shares due to rounding down, then withdraw would burn more shares due to rounding up.

- Property VAULT-08: For any sequence of operations (deposits, withdrawals, rate updates) and any operation parameters (timing, size, frequency), the economic outcome for an attacker should never be better than a honest user conducting standard operations

  - Threat a: An advantage is gained by breaking large operations into smaller ones.

    In order to cover this threat, our approach was to break:

    Scenario a.1: Big deposit into smaller deposits

    Scenario a.2: Big mint into smaller mints

    Scenario a.3: Big withdraw into smaller withdraws

    Scenario a.4: Big redeem into smaller redeems

    Scenario a.1: Big deposit into smaller deposits

    → Having one big deposit split into smaller ones the fees could be bigger, since the fees will be calculated by ceiling the division. If this is the case, user would end up with less assets deposited, and therefore less shares minted, since there is flooring when shares are calculated.

    Scenario a.2: Big mint into smaller mints

    → When mint fee is calculated it is done as a result of subtraction between gross assets and base assets. Both gross and base assets are calculated using ceil rounding. In some scenarios it is possible that more assets get transferred to mint the same amount of shares. This way the user does not profit from smaller mints.

    Scenario a.3: Big withdraw into smaller withdraws

    → When withdraw is split into smaller withdraws, sum of withdrawal fee for a group of small withdrawals will be bigger or equal to the withdrawal fee of the big withdrawal created. This would mean that total sum of net assets for all withdrawals will be less than net assets for one big withdrawal, leading to less post fee shares. Also, due to ceil rounding when `sharesToBurn` is calculated, total amount of shares to burn for all small withdrawals would be bigger or equal to the shares to burn for one big withdrawal.

    Scenario a.4: Big redeem into smaller redeems

    → When one big redeem is split into smaller redeems, it can happen that the sum of all small redeem's post fee shares be smaller than the actual `postFeeShares` of one big redeem. However, the users would also pay less fee. One of the examples we run splits one big redeem with `shares=3690 * 10 ** 18` into 30 `shares=123 * 10 ** 18` redeems. Result of that was: 1) sum of all `postFeeShares` when redeem was split was smaller by `122_754 * 10 ** 15` if redeem was not split, 2) total fee paid when redeem was split was smaller but by `738 * 10 ** 15` if redeem was not split, 3) having that in mind, there would be no economic incentive to split redeems.

    Conclusion: The property holds.

  - Threat b: An advantage is gained by combining, ordering or timing operations in specific sequences around rate updates

    Conclusion: The property doesn't hold under the assumption that the strategist may behave maliciously. A malicious strategist, knowing that the redemption rate will increase, could incentivise accomplices to

deposit assets into the vault just before the rate update. This creates a dual benefit: (1) the depositors benefit from the more favorable exchange rate after the update, and (2) the deposit fees generated benefit the strategist who receives a portion of all fees distributed at the old (lower) rate, making those fee shares more valuable after the rate increase. This coordinated attack exploits the timing vulnerability in the `update()` function where fee distribution occurs before the rate change↗, allowing the strategist to extract additional value through strategic timing and collusion.

- Threat c: The redemption rate is manipulated to create arbitrage opportunities between deposit and withdrawal operations

  Conclusion: The property holds. The vault's architecture mitigates arbitrage opportunities between deposit and withdrawal operations due to its cross-chain, request-based withdrawal mechanism. Unlike conventional ERC4626 vaults where users can immediately redeem shares for assets, the `OneWayVault` locks the redemption rate at the time of withdrawal request creation, preventing post-request rate manipulation from affecting payouts. The vault's one-way design effectively breaks the arbitrage loop by separating the timing of share burning from asset delivery, rendering redemption rate manipulation ineffective for creating deposit-withdrawal arbitrage opportunities.

Deposit & mint

- Property VAULT-09: Deposits must only be processed when vault is operational

  Conclusion: **The property doesn't hold.** Operational vault will accept deposit operation and first check the stale rate, and if the rate becomes stale during user's operation, its deposit will return zero shares, while wasting gas. We have discussed this with the development team and they have informed us that this is by design. However, since we do not find it fair that users pay gas and get nothing in the return, we have written the finding "Stale rate auto-pause mechanism wastes gas for unlucky users".

- Property VAULT-10: Mint requests must only be processed when vault is operational

  Conclusion: **The property doesn't hold.** Operational vault will accept mint operation and first check the stale rate, and if the rate becomes stale during user's operation, its mint will return zero assets, while wasting gas. We have discussed this with the development team and they have informed us that this is by design. However, since we do not find it fair that users pay gas and get nothing in the return, we have written the finding "Stale rate auto-pause mechanism wastes gas for unlucky users".

- Property VAULT-11: If the vault is configured with a non-zero deposit cap, then deposits must never cause total assets to exceed the deposit cap

  Conclusion: The property holds. If `config.depositCap != 0` , then assets that are larger than total assets calculated from total shares using current redemption rate will result in revert.

- Property VAULT-12: If the vault is configured with a non-zero deposit cap, then desired shares must never cause total assets to exceed the deposit cap

  Conclusion: The property holds. If `config.depositCap != 0` , then amount of shares that are larger than amount of shares calculated from total amount of deposited assets deducted using current redemption rate will result in revert.

- Property VAULT-13: Minted shares must correspond exactly to assets deposited minus deposit fees divided by the redemption rate

  Conclusion: The property holds. Minting shares is done for the amount of shares calculated using `assetsAfterFee` variable. `assetsAfterFee` represents amount of assets deposited deducted by calculated fee.

- Property VAULT-14: Required assets to mint desired shares must include the minting fee

Conclusion: The property holds. Required assets to mint desired shares are calculated using `calculateMintFee(shares)` which calculates `grossAssets` using the following function `baseAssets.mulDiv(BASIS_POINTS, BASIS_POINTS - feeBps, Math.Rounding.Ceil)`.

- Property VAULT-15: All deposited assets must be transferred to deposit account

  Conclusion: The property holds. Assets deposited, including the fee will be transferred to the address of `depositAccount` and tracking of how much fees are in the contract is done with `feesAccruedInAsset` state variable.

- Property VAULT-16: Designated receiver must receive the minted shares

  Conclusion: **The property doesn't hold.** `_deposit` function is called using `_msgSender()` function instead of `msg.Sender`. This can cause issues and it is generally not recommended to use `_msgSender()` if not supporting EIP-2771. If `_msgSender()` is overridden in the underlying ERC20, msg.sender might not equal to `_msgSender()`. Therefore, we have written the finding "Using `_msgSender()` in contracts that do not support metatransactions**".**

- Property VAULT-17: Converting assets to shares and back to assets should never result in a value larger than the original assets

  Conclusion: The property holds. Amount of shares generated by `previewDeposit(amount)` function will yield the amount of shares, which when converted to assets using `previewMint(shares)` results in the starting amount.

Withdraw & redeem

- Property VAULT-18: Withdrawals must only be processed when vault is operational

  Conclusion: **The property doesn't hold.** Operational vault will accept withdraw operation and first check the stale rate, and if the rate becomes stale during user's operation, its withdraw will return early, while wasting gas. We have discussed this with the development team and they have informed us that this is by design. However, since we do not find it fair that users pay gas and get nothing in the return, we have written the finding "Stale rate auto-pause mechanism wastes gas for unlucky users".

- Property VAULT-19: Redeem requests must only be processed when vault is operational

  Conclusion: **The property doesn't hold.** Operational vault will accept redeem operation and first check the stale rate, and if the rate becomes stale during user's operation, its redeem will return early, while wasting gas. We have discussed this with the development team and they have informed us that this is by design. However, since we do not find it fair that users pay gas and get nothing in the return, we have written the finding "Stale rate auto-pause mechanism wastes gas for unlucky users".

- Property VAULT-20: Withdrawal amount must not exceed total shares of owner

  Conclusion: The property holds. During withdraw process, amount of shares supplied as shares to be burned is calculated by using amount of assets that the user would like to withdraw, using floor rounding. This means that due to the check here↗, there is no possibility that the user withdraws more assets than they have.

- Property VAULT-21: Withdrawn net assets must equal the requested assets minus the withdrawal fee

  Conclusion: **The property doesn't hold.** Withdrawal request stores `sharesAmount` and `redemptionRate`. These values are used to calculate amount that should be withdrawn. Since there can be rounding due to integer division on the destination side, the users might end up with less assets than they have requested and burned the shares for. We have disclosed this to the development team and they have informed us that that is by design and that they will be of level 1e-6 for USDC, and they have decided to go with that due to flexibility reasons.

- Property VAULT-22: Redeemed net assets must equal the assets of requested shares minus the withdrawal fee

  Conclusion: The property holds. Net amount of assets equals to amount of assets converted from shares that the user wants to redeem deducted by the withdrawal feel (here↗).

- Property VAULT-23: Withdrawal requests must only be created by burning the corresponding shares

  Conclusion: The property holds. Withdrawal request is created with `postFeeShares` value which corresponds to the total amount of assets calculated from the supplied shares, deducted by withdrawal fee and then converted back to shares that are recorded as `sharesAmount`.

- Property VAULT-24: Withdrawal request IDs must be unique and sequential

  Conclusion: The property holds. `currentWithdrawRequestId` value is used for withdrawal request ID (here↗), and it is increased after each withdrawal request is created (here↗).

Findings

Finding	Type	Severity	Status
target lacks zero-check in Account execute() function	Implementation	High	Patched Without Reaudit
Unexpected payable functions in OneWayVault	Implementation	High	Patched Without Reaudit
Lack of proper validation for withdraw request receiver	Implementation	Medium	Risk Accepted
Malicious strategist could manipulate redemption rate to its own advantage	Implementation	Medium	Patched Without Reaudit
Unpausing vault does not make it usable in the event of stale rate	Implementation	Medium	Patched Without Reaudit
Unexpected payable function in LiteProcessor	Implementation	Medium	Patched Without Reaudit
Missing validation of redemption rate in contract initialisation	Implementation	Medium	Patched Without Reaudit
Stale rate auto-pause mechanism wastes gas for unlucky users	Implementation	Low	Acknowledged
Malicious users can spam withdrawals with small amounts to overwhelm destination	Implementation	Low	Acknowledged
VerificationGateway might be zero address when adding/removing registries	Implementation	Low	Patched Without Reaudit
Requests will be piling up and take up contract storage	Implementation	Low	Acknowledged
Differently sized arrays will cause panic in execute function	Implementation	Low	Patched Without Reaudit

Finding	Type	Severity	Status
Processor functions are not protected by nonReentrant modifier	Implementation	Informational	Patched Without Reaudit
Deposit and mint functions are not protected by nonReentrant modifier	Implementation	Informational	Patched Without Reaudit
Using msgSender function in contracts that do not support meta-transactions	Implementation	Informational	Patched Without Reaudit
Miscellaneous code improvements	Implementation	Informational	Patched Without Reaudit
Recommendations for gas optimisation	Implementation	Informational	Patched Without Reaudit

target lacks zero-check in Account execute() function

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/accounts/Account.sol`↗

Description

The `Account` contract's `execute()` function does not validate that the `_target` parameter is not the zero address before making external calls. This allows approved libraries or the contract owner to make calls to the zero address, which can lead to unexpected behavior, gas waste, and potential security implications depending on the EVM implementation.

Problem scenarios

Libraries or owners may accidentally pass `address(0)` as target due to programming errors in library implementations, uninitialized address variables being passed or integration mistakes where target address is not properly set. This can lead to ETH loss risks. If `_value > 0` and `_target` is zero address, ETH gets sent to the zero address (permanently burned) and there is no way to recover the sent ETH.

Also, calls to zero address return `success = true` and libraries may incorrectly assume operations completed successfully, therefore committing state changes based on false success.

Recommendation

We recommend implementing a check to ensure that `_target` cannot be `address(0)`.

```
error ZeroAddressTarget();

function execute(address _target, uint256 _value, bytes calldata _data) external
↪  returns (bytes memory result) {
  if (!approvedLibraries[msg.sender] && msg.sender != owner()) {
    revert NotOwnerOrLibrary(msg.sender);
  }

  if (_target == address(0)) {
    revert ZeroAddressTarget();
  }

  (bool success, bytes memory returnData) = _target.call{value: _value}(_data);

  // ... rest of function
}
```

Status

The development has implemented a fix in PR#413↗.

Unexpected payable functions in OneWayVault

Severity High **Impact** 3 - High **Exploitability** 2 - Medium

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `OneWayVault` contract declares the `withdraw()`↗ and `redeem()`↗ functions as `payable`, but these functions have no logic to handle ETH payments. This creates confusion and potential for users to accidentally send ETH along with their withdrawal requests, causing permanent loss of funds as the ETH becomes locked in the contract with no recovery mechanism.

Problem scenarios

Here are some of the scenarios of interest.

Accidental ETH loss: Users may mistakenly send ETH when calling these functions:

- User calls `withdraw()` with `msg.value > 0`, the function executes normally but ETH is trapped in contract and there is no mechanism in place to recover the ETH.

Potential wallet integration issues: Some wallet interfaces may automatically add ETH:

- Wallet software may interpret `payable` as requiring ETH payment and users may not notice ETH being sent along with transaction, making ETH becomes permanently locked.

Smart contract integration: Other contracts integrating with the vault may send ETH:

- Integration assumes payable functions require ETH payment and multi-call transactions may batch ETH transfers with withdrawals making ETH accumulate in contract over time.

MEV Bot Confusion: MEV bots may misinterpret the payable functions leaving ETH locked during automated operations.

Recommendation

We recommend to remove payable modifiers, since these functions don't need ETH.

Status

The development has implemented a fix in PR#412↗.

Lack of proper validation for withdraw request receiver

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Risk Accepted

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The vault accepts arbitrary string inputs as receiver addresses↗ for withdrawals (`withdraw()` and `redeem()` functions) but does not validate whether these strings are correctly formatted `bech32` addresses. These strings are stored↗ and used for transfers after shares are irreversibly burned↗. If the address is malformed or incorrect, funds may be lost permanently with no way to recover or modify the request.

Problem scenarios

Because the withdrawal request is created and shares are burned without validating the receiver address, any user mistake—such as typos, incorrect formats, or wrong chain prefixes—can result in the vault sending funds to an invalid or unreachable address. There is no format enforcement, or ability to cancel or update requests after submission.

Recommendation

Implement strict validation to ensure the receiver string matches the expected Neutron `bech32` format. Consider introducing a two-phase withdrawal process, or mechanisms to cancel or update pending requests, to reduce the risk of irreversible user errors.

Malicious strategist could manipulate redemption rate to its own advantage

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The strategist has exclusive authority to update the redemption rate through the `update()` function↗, which is protected by the `onlyStrategist` modifier. This makes the strategist the sole entity responsible for maintaining accurate asset valuations in the vault.

Problem scenarios

Even though the strategist is a trusted actor in the Valence Protocol (thus our assessment of low exploitability), malicious behaviour can affect the safety of the protocol if, for example, the strategist owns shares and manipulates the redemption rate to its own advantage to maximise the amount of assets that it can withdraw.

Recommendation

The Timewave team has added (see PR#407↗) extra configuration parameters to `OneWayVaultConfig` structure↗ to prevent that the strategist quickly updates the rate to a very high value to cash out its shares and then brings back the rate down. The PR adds a minimum update delay that has to pass before strategist can update redemption rate, and also adds a maximum delta for rate increases and a maximum delta for rate decreases.

Status

The development has implemented a fix in PR#407↗.

Unpausing vault does not make it usable in the event of stale rate

Severity Medium **Impact** 2 - Medium **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `OneWayVault`'s `unpause()`↗ function fails to update `lastRateUpdateTimestamp` when unpausing a vault that was automatically paused due to stale redemption rates. This creates an immediate re-pause cycle where the vault gets paused again on the very next user interaction, violating the fundamental property that users should be able to interact with an unpaused vault.

Problem scenarios

When a vault is automatically paused due to stale rates, the owner can call `unpause()` to restore operations:

```
function unpause() external onlyOwnerOrStrategist {
  VaultState memory _vaultState = vaultState;
  if (_vaultState.pausedByStaleRate && msg.sender != owner()) {
    revert("Only owner can unpause if paused by stale rate");
  }
  delete vaultState; // Clears pause flags
  emit PausedStateChanged(false);
  // MISSING: lastRateUpdateTimestamp is NOT updated
}
```

However, since `lastRateUpdateTimestamp` remains unchanged, the next user operation triggers `_checkAndHandleStaleRate()`↗ again:

```
function _checkAndHandleStaleRate() internal returns (bool) {
  if (uint64(block.timestamp) - lastRateUpdateTimestamp >
  ↪  config.maxRateUpdateDelay) {
    vaultState.paused = true;
    vaultState.pausedByStaleRate = true;
    return true; // Immediately pauses again
  }
  return false;
}
```

This creates a deterministic cycle: 1. unpause, 2. user interaction, 3. immediate re-pause.

The vault becomes effectively unusable during periods when the strategist is unavailable. Users waste gas on transactions that appear to succeed (unpause events are emitted) but accomplish nothing, as subsequent operations immediately trigger re-pausing. This undermines the emergency unpause capability and creates a frustrating user experience.

You can find the test confirming the described problem bellow.

```
function test_ManualPauseAndUnpauseThenStaleRateThenUnpauseAndFail() public {
↪
    vm.warp(block.timestamp + maxRateUpdateDelay + 1 days); // Warp past max
↪  delay
```

```
    // After warping, rate becomes stale, deposit pauses the contract
    vm.prank(user1);
    uint256 shares = vault.deposit(1000 * 10 ** 18, user1);
    assertEq(shares, 0); // No error is returned, only 0 as result of deposit

    // Owner can still unpause
    vm.prank(owner);
    vault.unpause();

    (bool paused, bool pausedByOwner, bool pausedByStaleRate) =
↪   vault.vaultState();

    // Assert contract not paused
    assertFalse(paused);
    assertFalse(pausedByOwner);
    assertFalse(pausedByStaleRate);

  // The user1 should be able to deposit something since the contract is not
   ↪  paused
  // However, this does not happen, since _checkAndHandleStaleRate decides that
   ↪  the rate is stale and pauses the contract
  vm.prank(user1);
  shares = vault.deposit(1000 * 10 ** 18, user1);
  assertEq(shares, 0); // No error is returned, only 0 as result of deposit,
   ↪  since the contract gets paused

  (paused, pausedByOwner, pausedByStaleRate) = vault.vaultState();

    // Assert contract is paused by stale rate
  assertTrue(paused);
  assertFalse(pausedByOwner);
  assertTrue(pausedByStaleRate);
}
```

Recommendation

Our recommendation is to prevent owner or strategist unpausing the vault in the event of stale rate, unless the rate is updated. When rate is updated, if the vault was not paused by the owner, vault should be unpaused by setting `pausedByStaleRate` field and `paused` as `false`. This way there is no need for two transactions: one to update rate, and one to unpause, therefore saving gas and bringing vault back to operational state faster.

Status

The development has implemented a fix in PR#414↗.

Unexpected payable function in LiteProcessor

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/processor/LiteProcessor.sol`↗

Description

The `LiteProcessor` contract's `execute()`↗ and `handle()`↗ functions are declared as `payable`, but the functions have no logic to handle ETH payments or native token transfers. This creates a risk where users or integrated systems may accidentally send ETH along with their execution calls, causing permanent loss of funds as the ETH becomes locked in the contract with no recovery mechanism.

Problem scenarios

Authorized callers may mistakenly send ETH when calling the function, and function will execute normally, but ETH is trapped in contract and there is no mechanism exists to recover the ETH.

Recommendation

Our recommendation is to remove payable modifier, since the function doesn't need ETH.

Status

The development has implemented a fix in PR#415↗.

Missing validation of redemption rate in contract initialisation

Severity Medium **Impact** 3 - High **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `OneWayVault` contract allows the redemption rate to be set to zero during initialization, which breaks all share/asset conversion functions. The `initialize()`↗ function accepts a `startingRate`↗ parameter without validation, and if set to zero, it causes division-by-zero errors in critical conversion functions.

The redemption rate is fundamental to the vault's operation as it determines the exchange rate between shares and assets. A zero redemption rate makes the vault completely unusable, until it has been updated.

Problem scenarios

When `redemptionRate`↗ is zero, the `_convertToShares()`↗ function will cause division by zero:

```
function _convertToShares(uint256 assets, Math.Rounding rounding) internal view
↪  override returns (uint256) {
  return assets.mulDiv(ONE_SHARE, redemptionRate, rounding); // Division by zero
  ↪  if redemptionRate == 0
}
```

All `deposit`, `mint`, `withdraw`, and `redeem` operations depend on these conversion functions that depend on `redemptionRate`: `_convertToShares()` and `_convertToAssets()` (which will always return zero since it uses `shares.mulDiv(redemptionRate, ONE_SHARE, rounding)`).

Recommendation

We recommend implementing a `require` which would prevent zero value being set during initialisation.

```
function initialize(
  address _owner,
  bytes memory _config,
  address underlying,
  string memory vaultTokenName,
  string memory vaultTokenSymbol,
  uint256 startingRate
) public initializer {
  require(startingRate > 0, "Starting redemption rate cannot be zero");
  // ... rest of initialization ...
  redemptionRate = startingRate;
}
```

Status

The development has implemented a fix in PR#416↗.

Stale rate auto-pause mechanism wastes gas for unlucky users

Severity Low **Impact** 1 - Low **Exploitability** 2 - Medium

Type Implementation **Status** Acknowledged

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `OneWayVault` contract implements an automatic pause mechanism that triggers when the redemption rate becomes stale↗ (hasn't been updated within `maxRateUpdateDelay` seconds). However, this mechanism creates a situation where an unfair gas amount was wasted for unlucky users who managed to be the first to interact with the pool after rate went stale. Functions return early with zero values without proper notification or revert behavior.

The `_checkAndHandleStaleRate()`↗ function is called at the beginning of critical user operations like `deposit()`, `mint()`, `withdraw()`, and `redeem()`, and can silently pause the vault mid-operation.

Problem scenarios

This can cause several problems. If the redemption rate is stale, the functions will return 0↗ (or nothing↗) without any indication to the user that their operation failed or that the vault was paused. Users may think their transaction succeeded but received no shares. Users still pay gas fees for transactions that effectively do nothing, as the function returns early without reverting.

The vault state changes from "not paused" to "paused↗" during the user's transaction, creating inconsistent behavior where the `whenNotPaused` modifier↗ passes but the operation still fails.

Recommendation

In order to prevent users from paying gas for the transactions that effectively only pause the contract, our recommendation is to rethink other approaches to handling stale rate.

Malicious users can spam withdrawals with small amounts to overwhelm destination

Severity Low **Impact** 2 - Medium **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `OneWayVault` allows unlimited withdrawal requests without minimum amount thresholds or rate limiting, enabling malicious users to spam thousands of micro-withdrawals that overwhelm destination chain processing infrastructure, while paying small fee for them.

Problem scenarios

An attacker can repeatedly call `withdraw()` with minimal amounts, creating thousands of individual withdrawal requests. Each micro-withdrawal creates individual cross-chain processing overhead while consuming minimal source chain gas, creating an asymmetric attack where destination chain resources are disproportionately consumed. As a result, users may experience delays in their withdrawals, leading to frustration and degraded trust in the system. Additionally, the overload of events can put strain on monitoring and analytics infrastructure, making it harder to maintain visibility into the system's actual behavior.

Recommendation

Our recommendation is to rethink ways of dealing with this potential issue. One of the ways would be to introduce minimum amount required for the withdraw, while another solution would be rate limiting the caller.

VerificationGateway might be zero address when adding/removing registries

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/authorization/Authorization.sol`↗

Description

The `_verificationGateway` address parameter of the constructor↗ and of the function `updateVerificationGateway()`↗ are allowed to be zero addresses (as explained here↗). In `executeZKMessage()` function there is a check↗ to revert in case the address is zero, however the same check is not present in functions `addRegistries()`↗ and `removeRegistries()`↗, where the functions `addRegistry()` and `removeRegistry()` of `VerificationGateway` are called.

Problem scenarios

Even if zero-knowledge proofs are not used, a call to `addRegistries()`↗ and `removeRegistries()`↗ will fail and unnecessarily consume gas.

Recommendation

Add a check to functions `addRegistries()`↗ and `removeRegistries()`↗ to revert early if the address of `verificationGateway` is zero.

Status

The development has implemented a fix in PR#417↗.

Requests will be piling up and take up contract storage

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Acknowledged

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The vault uses the `withdrawRequests` mapping keyed by an incrementing `currentWithdrawRequestId` to store each withdrawal request. This allows tracking of individual withdrawal intents and facilitates post-processing.

Problem scenarios

Each call to `_withdraw()`↗ creates a new `WithdrawRequest`↗ and stores it in the `withdrawRequests` mapping↗. Over time, especially in high-volume systems, this can lead to unbounded growth of on-chain storage. Without a mechanism for pruning or archiving fulfilled or expired requests, this pattern introduces long-term storage bloat and increasing gas costs for related operations (e.g., iteration, checkpoints, or migrations).

Recommendation

We recommend re-evaluating how `withdrawRequests` are managed to avoid unbounded storage growth over time. One possible approach could be to remove or archive fulfilled or expired requests after processing (which has the added benefit of issuing gas refunds for releasing storage).

Differently sized arrays will cause panic in execute function

Severity Low **Impact** 1 - Low **Exploitability** 1 - Low

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- solidity/src/processor/LiteProcessor.sol↗
- solidity/src/processor/ProcessorBase.sol↗

Description

During `execute()` function `_body` function argument is decoded into `ProcessorMessage`. If `ProcessorMessage` is of type `ProcessorMessageType.SendMsgs` then `_handleSendMsgs` is called with that decoded message as function argument. That argument is later decoded into `SendMsgs` structure which is then passed into either `_handleAtomicSubroutine` or `_handleNonAtomicSubroutine` functions.

```
struct SendMsgs {
  uint64 executionId;
  Priority priority;
  Subroutine subroutine;
  uint64 expirationTime;
  bytes[] messages;
}
```

Problem scenarios

During `_handleAtomicSubroutine` and `_handleNonAtomicSubroutine` functions iterate over `subroutine.functions` array and for i-th function they are executing `.call` on `subroutine.functions[i].contractAddress` with `messages[i]`. However, there is no check which will ensure that `subroutine.functions` array and `messages` array are the same size, which can lead to `panic: array out-of-bounds access (0x32)`.

This `panic` will not be caught in `try-catch` block and will revert the contract in both cases.

Recommendation

Even though the processor messages will arrive from authorized addresses, this can still happen and it would be ideal to stop execution right at the beginning with `require`.

Status

The development has implemented a fix in PR#418↗.

Processor functions are not protected by nonReentrant modifier

Severity Informational **Impact** 3 - High **Exploitability** 0 - None

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/processor/LiteProcessor.sol`↗
- `solidity/src/processor/ProcessorBase.sol`↗

Description

The `LiteProcessor` contract contains a cross-function reentrancy vulnerability in its message handling flow. The vulnerability arises from external calls made during subroutine execution that can potentially re-enter the contract through different entry points before the execution state is fully resolved.

While we identified the problem, we had in mind that only authorized addresses will be interacting with the `LiteProcessor` contract. But since we cannot rely on the fact that they would not reenter, we decided to report this as a finding.

Problem scenarios

The reentrancy occurs in the `_handleSendMsgs()`↗ function through the following call chain:

1. **Subroutine execution calls**: Both atomic and non-atomic subroutines make arbitrary external calls
2. **Callback dispatch**: The callback mechanism makes additional external calls

There are several vectors of interest:

Subroutine execution reentrancy

A malicious contract included in the subroutine execution can re-enter the `LiteProcessor` through:

- The `handle()`↗ function (if it controls the Hyperlane mailbox)
- The `execute()`↗ function (if it's an authorized address)

Callback reentrancy

The callback receiver can re-enter during the `_sendCallback()`↗ execution, potentially causing:

- State inconsistencies in execution tracking
- Double execution of subroutines
- Manipulation of execution results

Recommendation

Our recommendation is to implement reentrancy protection.

Status

The development has implemented a fix in PR#419↗.

Deposit and mint functions are not protected by nonReentrant modifier

Severity Informational **Impact** 3 - High **Exploitability** 0 - None

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The `deposit()`↗ and `mint()`↗ functions in the `OneWayVault` contract are not protected by a `nonReentrant` modifier. These functions involve asset transfers and state updates, which are common targets for reentrancy vulnerabilities in general ERC4626 vault patterns.

Problem scenarios

While the ERC20 asset used in the vault is set by a trusted party and is assumed to behave correctly, omitting reentrancy protection could pose a risk if the vault logic is reused with untrusted tokens or extended in the future. If an ERC20 token with reentrancy behavior is mistakenly integrated, it could exploit internal state inconsistencies by calling back into `deposit()`↗ or `mint()`↗ mid-execution.

Recommendation

Given the current trust assumptions, the risk is very low. However, as a defense-in-depth measure, we recommend marking `deposit()` and `mint()` with the `nonReentrant` modifier. This provides future-proofing and prevents accidental vulnerabilities in case the vault is upgraded or reused with less trusted tokens.

Status

The development has implemented a fix in PR#419↗.

Using msgSender function in contracts that do not support meta-transactions

Severity Informational **Impact** 3 - High **Exploitability** 0 - None

Type Implementation **Status** Patched Without Reaudit

Involved artifacts

- `solidity/src/vaults/OneWayVault.sol`↗

Description

The vault contract uses `_msgSender()` for caller identification (here↗ and here↗). This function is inherited from OpenZeppelin's `Context` via `ERC4626` and then via `ERC20`, and while safe by default, it introduces unnecessary abstraction when meta-transactions are not in use.

Problem scenarios

If the ERC20 token passed to the vault were to override `_msgSender()`, it could alter the vault's behavior — potentially impacting access control or attribution logic. While this is unlikely with a trusted deployer, future misuse or extension of the vault with third-party ERC20s could expose it to subtle bugs or privilege escalation.

Recommendation

Since the ERC20 asset is set by a trusted party and meta-transaction support is not required, we recommend replacing `_msgSender()` with `msg.sender` throughout the vault. This improves clarity, avoids inheritance risks, and aligns with the contract's current trust model.

Status

The development has implemented a fix in PR#420↗.

Miscellaneous code improvements

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Patched Without Reaudit

- The storage variable `storeCallbacks` in `Authorization` can be made immutable, since it is only set once in `initialize` and then read in `handleCallback`.
- Remove the `coprocessorRoot` storage variable of `VerificationGateway`, since it is only set but not used.
  - This was done in PR#406.
- Generally consider changing `public` functions for `external` if they are not used in the code. This is relevant for `OneWayVault` and `Processor`.
- Consider having this as one line.
- Consider having this done in a single line.
- `WithdrawRequested` event is emitted before the state changes (here).
- Consider emitting events:
  - In `Account`, when approving / removing libraries.
  - In `Authorization`, when updating the `Processor` or when updating the `VerificationGateway`.
  - In `VerificationGateway`, when updating the verifier, updating the domain VK, and adding / removing registries.
- Consider adhere to CEI pattern and move the increment of `executionId` before `processor.execute(message)`.
- Redundant checks in `_withdraw` function (here). `_withdraw` function calls `_spendAllowance` which then performs the same checks as previously performed in `_withdraw` function. Openzeppelin's common workflow example shows that after performing caller check it is enough to call `_spendAllowance` (here) since spending allowance performs other checks (here).

Contract upgradability

- The `OneWayVault` contract implements the UUPS (Universal Upgradeable Proxy Standard) pattern by inheriting OpenZeppelin's `UUPSUpgradeable` contract. However, the `initialize()` function of the contract lacks the call to `__UUPSUpgradeable_init()`. It is the recommended practice to call `__UUPSUpgradeable_init()` for a number of reasons:
  - It signals to readers and tools that this contract supports UUPS upgrades.
  - It future-proofs your contracts, should OpenZeppelin introduce or change any setup logic in later version.
  - Since the constructor of the contracts is disabled, calling `__UUPSUpgradeable_init()` will correctly initialise the parent contracts, ensuring that the upgrade mechanism is not broken.
- Consider implementing a pure `version()` function to `OneWayVault` and `VerificationGateway` that returns the version of the contracts. For example:

```
function version() public pure returns (uint256) {
  return 1;
}
```

Future versions of the contract can also implement it by returning higher version number, so that it is easy to identify what version of the contract is deployed. Additionally, you could consider checking the version of the new contract in the `_authorizeUpgrade()` function, to make sure that only upgrades to newer versions are allowed.

Status

The development has implemented a fix in PR#420↗.

Recommendations for gas optimisation

Severity Informational **Impact** 0 - None **Exploitability** 0 - None

Type Implementation **Status** Patched Without Reaudit

- Use custom error in `require` statements. In all files under scope only the `require` statement in `LiteProcessor`'s `execute` function uses a custom error↗, while the rest use a `string`. Using custom errors is more gas efficient and. can make the code more readable and maintainable.
- In `OneWayVault.sol`:
  - The storage layout of the contract may be optimised by moving the declaration of `BASIS_POINTS` constant↗ together with the definition of `currentWithdrawRequestId` and `lastRateUpdateTimestamp`:

```
uint64 public currentWithdrawRequestId;

uint64 public lastRateUpdateTimestamp;

/**
 * @dev Constant for basis point calculations (100% = 10000)
 */
uint32 private constant BASIS_POINTS = 1e4;
```

    All these variables can be them packed together in the same storage slot.
  - Writing the whole structure in the storage wastes more gas than only updating certain fields, since the whole structure will update changed fields as well as unchanged ones (here↗).
  - No need to have the whole struct created in the memory↗ and then update the struct in the memory, only to update one or two fields of the struct.
  - No need to load whole struct from the storage↗ just to pass one `uint256` field to the function.

Status

The development has implemented a fix in PR#422↗.

Appendix: Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1↗, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score↗, and the Exploitability score↗. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale↗, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)

ImpactScore	Examples
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

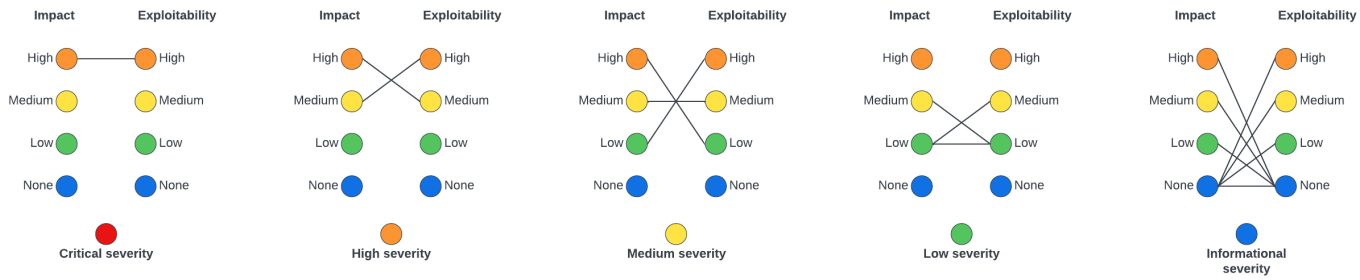

Figure 3: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	Examples
Critical	Halting of chain via a submission of a specially crafted transaction
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an "as is" basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an "endorsement", "approval" or "disapproval" of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client's business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.