Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Catalyst

Veridise Inc.

January 18, 2024

► **Prepared For:**

Cata Labs
https://catalabs.org/

► **Prepared By:**

Benjamin Sepanski
Nicholas Brown

► **Contact Us:** contact@veridise.com

► **Version History:**

Jan. 18, 2024	V1
Jan. 17, 2024	Initial Draft

Contents

1 Executive Summary

From Jan. 8, 2024 to Jan. 12, 2024, Cata Labs engaged Veridise to review the security of Catalyst*. The review covered an update to the smart contracts of Catalyst, an Automated Market Maker (AMM) for cross-chain swaps. Compared to the previous version, which Veridise also audited†, the new version refactored the Vaults, switched to Cata Labs's new `GeneralisedIncentives`‡ cross-chain incentive structure, and added support for underwriting of swaps by a third party. Underwriting is taken on at the underwriter's own risk, but can facilitate faster cross-chain swaps by providing an incentive to the underwriter.

Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week on commit `e7cd23fcc`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The Catalyst developers provided the source code of the Catalyst contracts for review. This code is original, written by the Catalyst developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the Catalyst developers also shared some internal documentation of the Catalyst protocol, and were very responsive to questions from the auditing team.

The source code contained a test suite, which the Veridise auditors noted covered a variety of scenarios, including successful and unsuccessful calls to the protocol. Several failure cases were carefully and thoroughly tested. The auditors found the code to be high quality, well-tested, and well-documented.

Summary of issues detected. The audit uncovered 5 issues, none of which are assessed to be of high or critical severity by the Veridise auditors. One warning and 4 informational findings were identified. Of the 5 issues, Cata Labs has fixed 3 issues. Of the remaining 2 issues, Cata Labs has acknowledged both but deemed them too minor to fix.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* `https://github.com/catalystdao/catalyst`
† The previous audit report, if it is publicly available, can be found on Veridise's website at `https://veridise.com/audits/`
‡ Also audited by Veridise, see `https://github.com/catalystdao/GeneralisedIncentives`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Catalyst	`e7cd23fcc`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 8 - Jan. 12, 2024	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed	Acknowledged
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	0	0	0
Warning-Severity Issues	1	0	1
Informational-Severity Issues	4	3	4
TOTAL	5	3	5

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Frontrunning	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Catalyst's smart contracts in the `evm/src` directory. The Veridise auditors validated the changes to the Vault contracts since the previous audit, as well as the new underwriting functionality and switch to `GeneralisedIncentives`. In our audit, we sought to answer questions such as:

- Is it possible for a malicious underwriter to damage/profit from the protocol or its users?
- Is it possible for a malicious user to take advantage of an underwriter?
- Is it ensured that a transaction cannot be underwritten multiple times?
- Are underwriters guaranteed to receive their funds if the underwritten message arrives?
- Are there any new vulnerabilities introduced by the refactoring of the Vaults?
- Can underwriters influence the callback requested by a user when receiving a swap?
- Is the switch to `GeneralisedIncentives` properly implemented, with all required security checks in place (e.g. can replayed acks have a negative effect)?
- Are any common Solidity vulnerabilities present (reentrancy, front-running risks, etc.)?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the `evm/src` folder of the source code provided by the Catalyst developers, which contains the smart contract implementation of the Catalyst. Specifically, the Veridise auditors audited the new file: `evm/src/CatalystChainInterface.sol` and verified that the changes to the Vault contracts either didn't change the behavior of the protocol, or were safe changes to make.

Methodology. Veridise auditors reviewed the reports of previous audits for Catalyst, inspected the provided tests, and read the Catalyst documentation. They then began a manual audit of the code assisted by static analyzers. During the audit, the Veridise auditors regularly met with the Catalyst developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-CAT-VUL-001	Underwriter may underwrite more than intended	Warning	Acknowledged
V-CAT-VUL-002	No domain separator for escrows	Info	Acknowledged
V-CAT-VUL-003	Unused functions/parameters/errors	Info	Fixed
V-CAT-VUL-004	catalyst Typos and Comment clarifications	Info	Fixed
V-CAT-VUL-005	_setLiquidityEscrow should document assumptior	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-CAT-VUL-001: Underwriter may underwrite more than intended

Severity	Warning	**Commit**	e7cd23f
Type	Frontrunning	**Status**	Acknowledged
File(s)	`evm/src/CatalystChainInterface.sol`		
Location(s)	underwrite()		
Confirmed Fix At			

When calling `underwrite()`, the underwriter has no manner by which to specify a maximum amount of tokens to underwrite.

```
function underwrite(
    address targetVault,  // -- Swap information
    address toAsset,
    uint256 U,
    uint256 minOut,
    address toAccount,
    uint16 underwriteIncentiveX16,
    bytes calldata cdata
) public returns(bytes32 identifier) {
```

Snippet 4.1: Function signature of `underwrite()`.

Impact If an underwriter approves the `CatalystChainInterface` for the maximum amount of `type(uint256).max`, then they will be vulnerable to front-running. Front-runners may lead the underwriter to take on more risk than they intend when underwriting swaps by performing large swaps to alter the nominal value of `U`.

Recommendation Add slippage protection for the underwriter.

Developer Response We are still deciding what to do about this. For the time being, the underwriter may mitigate risk by approving only a limited amount to the `CatalystChainInterface`.

4.1.2 V-CAT-VUL-002: No domain separator for escrows

Severity	Info	**Commit**	e7cd23f
Type	Maintainability	**Status**	Acknowledged
File(s)	`evm/src/CatalystVaultCommon.sol`, `evm/src/CatalystChainInterface.sol`		
Location(s)	_getUnderwriteIdentifier(), _computeSendAssetHash()		
Confirmed Fix At			

Both underwrite identifiers and asset hashes are used to identify escrowed funds for underwrites and pending swaps. These are stored in the same mapping: `_escrowedTokens`.

The hashes are computed on strings of different lengths, so a collision should not occur. As can be seen in the below code snippets, the send-asset hash is computed from 769 bytes, and the underwrite identifier is computed from at least 1,008 bytes.

```
function _computeSendAssetHash(
    bytes memory toAccount,
    uint256 U,
    uint256 amount,
    address fromAsset,
    uint32 blockNumberMod
) internal pure returns(bytes32) {
```

Snippet 4.2: Signature of `_computeSendAssetHash()`: all the elements are packed together and then hashed. The `toAccount` argument is a 65-byte object, as all callers verify its length.

```
function _getUnderwriteIdentifier(
    address targetVault,
    address toAsset,
    uint256 U,
    uint256 minOut,
    address toAccount,
    uint16 underwriteIncentiveX16,
    bytes calldata cdata
) internal pure returns (bytes32 identifier) {
```

Snippet 4.3: Signature of `_getUnderwriteIdentifier()`. All elements are packed together and then hashed.

Impact If the signatures of these identifier-derivation functions are ever changed, it may be possible to create a collision, leading to unexpected or damaging behavior.

Recommendation Add a domain-separator (i.e. a deterministic prefix which is different for `_computeSendAssetHash()` than the one used for `_getUnderwriteIdentifier()`) to the hash.

Developer Response While it technically is possible to collect the information from inputs to `_getUnderwriteidentifier` to provide to `_computeSendAssetHash` it wouldn't make sense.

Considering the very stark difference in information that is hashed (and length), it isn't deemed necessary.

Updated Veridise Response The scenario we would be more concerned with the reverse scenario, e.g. of a malicious underwriter being able to release a send-asset hash.

However, as said above, there is only a security issue if the identifiers are changed in future updates.

4.1.3 V-CAT-VUL-003: Unused functions/parameters/errors

Severity	Info	**Commit**	e7cd23f
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d4dbc1d		

1. evm/src/CatalystChainInterface.sol:
   a) The errors `SubcallOutOfGas` and `MaliciousVault` are unused.
2. evm/src/CatalystVaultCommon.sol:
   a) The following functions are defined, but never used.
      - `_setUnderwriteEscrow()`
      - `_releaseUnderwriteEscrow()`

      The identically-defined functions `_setTokenEscrow()` and `_releaseAssetEscrow()` are used instead.
   b) The `U` parameter is unused in `deleteUnderwriteAsset()`.

Impact

1. evm/src/CatalystChainInterface.sol:
   a) Off-chain actors may listen for the wrong error.
2. evm/src/CatalystVaultCommon.sol:
   a) Developers may be confused about the functions, or they may become out-of-date.
   b) The code may be less maintainable.

Recommendation

1. evm/src/CatalystChainInterface.sol:
   a) Remove the unused errors.
2. evm/src/CatalystVaultCommon.sol:
   a) Remove the unused functions.
   b) The unused parameter name should be removed in the function definition.

Developer Response We applied the recommendations.

4.1.4 V-CAT-VUL-004: catalyst Typos and Comment clarifications

Severity	Info	**Commit**	e7cd23f
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	see issue description		
Confirmed Fix At	0bea6b1d		

The following locations in the code have minor typos or potentially confusing comments.

1. evm/src/CatalystChainInterface.sol:
   a) `INITIAL_MAX_UNDERWRITE_DURATION`, `MAX_UNDERWRITE_DURATION`, and `MIN_UNDERWRITE_DURATION` are all variables representing time measured in seconds. However, they are defined as the ratio of two time intervals. For example, see the below code snippet.

```
uint256 constant INITIAL_MAX_UNDERWRITE_DURATION = 24 hours / 6 seconds;
```

Snippet 4.4: Definition of `INITIAL_MAX_UNDERWRITE_DURATION`.

   An hour divided by seconds is a number with no associated units.
   b) In `sendCrossChainAsset()`, at most the first 2^{16} of calldata are passed along. However, this is not documented in the function natspec.

```
uint16(calldata_.length),   // max length of calldata is 2**16-1 = 65535 bytes which
    should be more than plenty.
```

Snippet 4.5: Explicit cast and documented assumption about `calldata._length`.

   c) In `receiveAck()`, the `messageIdentifier` argument is unused.
   d) "collatoral" is used in place of "collateral."
   e) "sclies" is used in place of "slices."

Impact

1. evm/src/CatalystChainInterface.sol:
   a) Users may be unclear on the intended units of the `*_DURATION` variables.
   b) In rare cases, users may be surprised by unexpected failures.
   c) When the parameter name is present, developers will expect the argument to be used in the function.
   d) Clearer comments/variable names will improve maintainability for future developers.
   e) Clearer comments/variable names will improve maintainability for future developers.

Developer Response

1. evm/src/CatalystChainInterface.sol:
   a) These all represent time measured in blocks, not seconds.

b) Based on our testing, the cost associated with 65535 bytes of additional calldata would be so expensive it isn't worth it.

Unlike "traditional calldata" that doesn't have to be copied into memory, our calldata has to be copying into memory so you pay an expoential cost after a certain point (which 65535 bytes is larger than, not to mention the other data in memory).
We will add documentation. 3. We will comment variable out. messageIdentifier is intended for other Generalised Incentives applications. For Catalyst we use a stricter security assumptions which makes messageIdentifier untrusted. 4. and e.: We applied the fixes.

Updated Veridise Response On point 1.a, would it be possible to add documentation in the code that these time intervals are measured in blocks? We implicitly interpreted them that way when they were used in the code, but understanding their definitions would be made easier if the "blocks" unit were noted.

4.1.5 V-CAT-VUL-005: _setLiquidityEscrow should document assumptions

Severity	Info	Commit	e7cd23f
Type	Maintainability	Status	Fixed
File(s)	evm/src/CatalystVaultCommon.sol		
Location(s)	_setLiquidityEscrow()		
Confirmed Fix At	fc7c22a		

`fallbackUser` address is assumed to be non-zero because other functions use this as a check for the existence of the escrow. Since this function doesn't check that the value for `fallbackUser` is non-zero, all callers of this function need to make this check.

Impact Currently, all callers make the necessary check so there are no issues. However, this requirement should be documented in `_setLiquidityEscrow()` so that future developers won't forget to add this check for callers, which could lead to unexpected behavior.

Recommendation Add a comment documenting the need for a zero address check in all callers of this function.

Developer Response We applied the recommended fix.

Glossary

AMM Automated Market Maker. 1

front-running A vulnerability in which a malicious user takes advantage of information about a transaction while it is in the mempool. 5

reentrancy A vulnerability in which a smart contract hands off control flow to an unknown party while in an intermediate state, allowing the external party to take advantage of the situation. 5

smart contract A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure.. 15

Solidity The standard high-level language used to develop smart contracts on the Ethereum blockchain. See `https://docs.soliditylang.org/en/v0.8.19/` to learn more. 5