Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Ribbon

Aevo OTC

Veridise Inc.

March 27, 2023

► **Prepared For:**

Ribbon Finance
https://www.ribbon.finance/

► **Prepared By:**

Jon Stephens
Kostas Ferles
► **Contact Us:** contact@veridise.com

► **Version History:**

Mar 27, 2023 V1

Contents

1 Executive Summary

From Mar. 20 to Mar. 26, Ribbon engaged Veridise to review the security of their Aevo OTC protocol. The review covered the additions to the on-chain contracts of Rysk Finance's Gamma Protocol. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 weeks on commit `4a230df`. 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 Aevo OTC protocol is a fork of Rysk Finance's Gamma Protocol that allows whitelisted market makers to execute user option orders. To do so, a user first places an options order for some asset with a given strike price, expiry and premium. Such an order is not executed until a whitelisted market maker accepts the order. When the order is accepted and executed, the market maker deposits sufficient collateral to cover a percentage of the order's nominal value which is stored in a naked vault in the Gamma protocol backend. Additionally, the market maker submits a permit signature from the user to pay the premium for the order and, in return, options tokens are minted to the user. As orders are only partially collateralized, at times it may be determined that the order margin is insufficient, requiring market makers to deposit additional collateral. Upon expiry, users can choose to exercise their options by redeeming their tokens. Market makers, on the other hand, can settle their vault to receive any remaining collateral or proceeds.

Ribbon provided the source code for the Aevo OTC protocol, for review. In addition, they provided documentation describing the intended behavior of the protocol and a collection of tests built on top of the truffle testing framework.

Summary of issues detected. The audit uncovered 10 issues, 3 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-RAO-VUL-001 allows market makers to steal funds to pay collateral, V-RAO-VUL-002 identifies the potential for uncollateralized orders and V-RAO-VUL-003 identifies risks to the reputation of well-behaved market makers. In addition, the auditors identified 3 moderate-severity issues, including the potential for market makers to front-run order execution (V-RAO-VUL-006) and the possibility that market makers could use permit signatures to fill undesired orders (V-RAO-VUL-004). Finally, the auditors identified several other security concerns, including no validation on the protocol's premium fee (V-RAO-VUL-008).

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Aevo OTC	`4a230df`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 20 - Mar. 26, 2023	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	1	1
High-Severity Issues	2	2
Medium-Severity Issues	3	2
Low-Severity Issues	0	0
Warning-Severity Issues	2	2
Informational-Severity Issues	2	2
TOTAL	10	9

Table 2.4: Category Breakdown.

Name	Number
Data Validation	3
Usability Issue	2
Configuration Error	1
Logic Error	1
Frontrunning	1
Maintainability	1
Gas Optimization	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment Ribbon's additions to the on-chain contracts of Rysk Finance's Gamma Protocol. In our audit, we sought to answer the following questions:

- Can market makers avoid providing the required initial margin when executing an order?
- Can funds be mis-appropriated from users?
- Can a shortfall occur even if a market maker is well-behaved?
- Can users identify the source of potential shortfalls?
- Can users control which market maker executes their order?
- Can users avoid paying premiums?
- Can a user pay a larger premium than what was specified when placing an order?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. 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 other open-source tools. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The audit reviewed the additions that Ribbon made to the fork of Rysk Finance's Gamma Protocol for their Ribbon protocol. This included behaviors corresponding to order placement, order execution, collateral management and vault settlement. When conducting the audit, Veridise engineers first reviewed the provided documentation and test cases to understand the high-level design and intended behavior of the protocol. The auditors then performed a week-long security audit of the code with the assistance of both static analyzers and automated testing. In terms of the audit, the following files were in-scope:

- `contracts/core/OTCWrapper.sol`
- `contracts/core/MarginRequirements.sol`
- `contracts/lib/SupportsNonCompliantERC20.sol`

In addition, the changes made to the following files in Rysk Finance's Gamma Protocol were in-scope:

- `contracts/core/AddressBook.sol`
- `contracts/core/Controller.sol`

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:

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:

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., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-RAO-VUL-001	Market Maker can Steal Funds to Pay Collateral	Critical	Fixed
V-RAO-VUL-002	Potential for Uncollatoralized Order	High	Fixed
V-RAO-VUL-003	Well-Behaved Market Makers face Risk	High	Fixed
V-RAO-VUL-004	Market Maker fan Fill Undesirable Order	Medium	Fixed
V-RAO-VUL-005	Positions can be Opened without OTCWrapper	Medium	Fixed
V-RAO-VUL-006	Market Makers can Front-run Order Execution	Medium	Acknowledged
V-RAO-VUL-007	Use Constant instead of Magic Number	Warning	Fixed
V-RAO-VUL-008	No Constraints on Premium Fee	Warning	Fixed
V-RAO-VUL-009	SafeMath Unnecessary in Solidity 0.8	Info	Fixed
V-RAO-VUL-010	Order Premium must Equal Signature Amount	Info	Fixed

4.1 Detailed Description of Bugs

4.1.1 V-RAO-VUL-001: Market Maker can Steal Funds to Pay Collateral

Severity	Critical	Commit	4a230df
Type	Data Validation	Status	Fixed
Files	OTCWrapper.sol		
Functions	executeOrder, _settleFunds, _deposit		

When a Market Maker executes an order, they must provide collateral funds proportionate to the size of the order. They do so by providing a permit to the protocol so that it may transfer the required funds. As shown below, when an execute order is received, the permit for the user is validated to ensure the funds are from the requested user, but there is no similar validation for the owner of the Market Maker's permit.

```
function executeOrder(
    uint256 _orderID,
    Permit calldata _userSignature,
    Permit calldata _mmSignature,
    uint256 _premium,
    address _collateralAsset,
    uint256 _collateralAmount
) external nonReentrant {
    require(orderStatus[_orderID] == OrderStatus.Pending, "...");
    require(isWhitelistedMarketMaker[_msgSender()], "...");
    require(_userSignature.amount >= _premium, "...");

    Order memory order = orders[_orderID];

    require(_userSignature.acct == order.buyer, "...");
    require(block.timestamp <= order.openedAt.add(fillDeadline), "...");
    require(whitelist.isWhitelistedCollateral(_collateralAsset), "...");

    ...
}
```

The permit is later used by the protocol to transfer funds from the user who created the permit, to the protocol itself as shown below.

```
function _settleFunds(
    Order memory _order,
    Permit calldata _userSignature,
    Permit calldata _mmSignature,
    uint256 _premium,
    address _collateralAsset,
    uint256 _collateralAmount
) private {
    ...

    // market maker inflow
    _deposit(
        _mmSignature.acct,
```

```
        _collateralAsset,
        _collateralAmount,
        _mmSignature.deadline,
        _mmSignature.v,
        _mmSignature.r,
        _mmSignature.s
    );

    ...
}
```

As a result, the collateral paid by the market maker can come from any user for which the market maker has a signature. In addition, as shown below, this signature is only used in cases where the user pays with USDC.

```
function _deposit(
    address _acct,
    address _asset,
    uint256 _amount,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) private {
    require(_amount > 0, "OTCWrapper: amount cannot be 0");

    if (_asset == USDC) {
        // Sign for transfer approval
        IERC20Permit(USDC).permit(_acct, address(this), _amount, _deadline, _v, _r,
    _s);
    }

    // An approve() or permit() by the _msgSender() is required beforehand
    IERC20(_asset).safeTransferFrom(_acct, address(this), _amount);
}
```

As a result, a Market Maker can execute an order using funds from **any user** that has granted the OTCWrapper approval over their funds.

Impact Due to the lack of validation on the collateral payment account, a malicious market maker could steal funds from users to pay the collateral for executed orders by using permit signatures that users have provided them. In addition, since the only token that uses the permit is USDC, a market maker could also steal funds from any user that has granted approval to the OTCWrapper, such as a competing market maker. While this would likely come with repetitional risk, it can also provide significant financial gain as the protocol could claim user premiums without incurring any financial risk.

Recommendation Validate that `_mmSignature.acct` is `msg.sender`.

4.1.2 V-RAO-VUL-002: Missing Validation allows Uncollatoralized Order

Severity	High	Commit	4a230df
Type	Data Validation	Status	Fixed
Files	MarginRequirements.sol		
Functions	checkMintCollateral, _getInitialMargin		

When a market maker executes an order, they must provide sufficient collateral to cover a percentage of the order's notational value. This percentage is set by the owner and is unique to the order asset, the collateral asset and the market maker that is executing the order. Doing so allows the protocol to customize the required collateral to the perceived risk of the assets and market maker itself. As it stands, however, if the margin has not been set, a market maker can execute an order while providing essentially no collateral. This is because in the `checkMintCollateral` function, no validation is performed when retrieving a value from the `initialMargin` mapping. Since mappings return zero by default if an entry in the mapping does not exist, effectively this causes the required margin to be 0%.

```
function checkMintCollateral(
    address _account,
    uint256 _notional,
    address _underlying,
    bool _isPut,
    uint256 _collateralAmount,
    address _collateralAsset
) external view returns (bool) {
    // retrieve collateral decimals
    uint256 collateralDecimals = uint256(ERC20Interface(_collateralAsset).decimals())
    ;

    // retrieve initial margin
    uint256 initialMarginRequired = initialMargin[keccak256(abi.encode(_underlying,
    _collateralAsset, _isPut))][
        _account
    ];

    return
        _notional.mul(initialMarginRequired).mul(10**collateralDecimals).mul(10**
    ORACLE_DECIMALS) <=
        _collateralAmount.mul(oracle.getPrice(_collateralAsset)).mul(
    MAX_INITIAL_MARGIN).mul(10**NOTIONAL_DECIMALS);
}
```

Snippet 4.1: Location in `checkMintCollateral` where `initialMarginRequired` is not validated

Similarly, when checking how much collateral may be withdrawn a similar check is performed. As with `checkMintCollateral` no validation is performed either when the value is retrieved from the mapping in `_getInitialMargin` or when it is used in `checkWithdrawCollateral`.

```
function _getInitialMargin(address _otoken, address _account) internal view returns (
    uint256) {
    OtokenInterface otoken = OtokenInterface(_otoken);

    return
        initialMargin[keccak256(abi.encode(otoken.underlyingAsset(), otoken.
    collateralAsset(), otoken.isPut()))][
            _account
        ];
}
```

Snippet 4.2: Location in `_getInitialMargin` where the value retrieved from `initialMargin` is not validated

Impact This allows a market maker to accept an order while providing essentially no collateral (as eventually there is a check that collateral is not 0) to back their position in the order. A greedy market maker could therefore accept many orders to receive the order premium while incurring no financial risk.

Recommendation Add a check that `initialMargin[x][y] != 0`.

4.1.3 V-RAO-VUL-003: Reputation Risk for Well-Behaved Market Makers

Severity	High	**Commit**	4a230df
Type	Logic Error	**Status**	Fixed
Files	`OTCWrapper.sol`		
Functions	depositCollateral		

The protocol allows Market Makers to execute options trades without requiring full collateralization. Rather, for a market maker to accept an order, they must partially collateralize the order based on the perceived risk. In addition, keepers of the protocol can request that additional collateral be provided. Since the order is already active, however, a market maker doesn't have to deposit the additional requested collateral, but this comes with risks to the reputation of the market maker if the order cannot be filled at expiry. However, this mechanism also places well-behaved market makers at risk as all funds are collected in a single pool. Additionally, when OTokens are redeemed, funds are paid out of the pool regardless of the collateral that backs those tokens. As a result, an option that is not properly collateralized will still be paid out using pool funds.

Impact A collateral shortfall from a single market maker may not directly impact users as the option trade may still proceed, but can have an **indirect** impact as it could cause option trades executed by well-behaved Market Makers to fail. Additionally, as mentioned in another issue the underlying OTC protocol and therefore the pool is accessible to other users allowing similar shortfalls to occur. It therefore seems difficult for the user to accurately determine where the blame for a shortfall should be placed which disincentives Market Makers to be well-behaved.

Recommendation If reputational risk is intended to ensure Market Makers are properly collateralizing executed trades, provide a mechanism that allows users to more accurately determine when one is to blame for a shortfall. For example, one could provide a pool for each individual market maker so that a shortfall from one cannot impact another.

4.1.4 V-RAO-VUL-004: Market Maker can use Permit to Fill Undesirable Order

Severity	Medium	Commit	4a230df
Type	Usability Issue	Status	Fixed
Files	`OTCWrapper.sol`		
Functions	executeOrder		

When an order is executed, the market maker provides information about the premium and collateral for the order in addition to a permit signature from the user allowing the premium funds to be transferred from the user to the protocol. When the user provides the market maker with their signature, however, there is no guarantee that the market maker will use it as intended by the user. For example, consider a user that has two pending orders `o1` and `o2`. If a user provides their permit signature for order `o1`, nothing prevents the market maker from using that signature to instead execute `o2`.

```
function executeOrder(
    uint256 _orderID,
    Permit calldata _userSignature,
    Permit calldata _mmSignature,
    uint256 _premium,
    address _collateralAsset,
    uint256 _collateralAmount
) external nonReentrant {
    require(orderStatus[_orderID] == OrderStatus.Pending, "...");
    require(isWhitelistedMarketMaker[_msgSender()], "...");
    require(_userSignature.amount >= _premium, "...");

    Order memory order = orders[_orderID];

    require(_userSignature.acct == order.buyer, "...");
    require(block.timestamp <= order.openedAt.add(fillDeadline), "...");
    require(whitelist.isWhitelistedCollateral(_collateralAsset), "...");

    ...
}
```

Snippet 4.3: The `executeOrder` function where market makers provide user signatures to execute orders

Impact As the market maker is the one responsible for executing the order and providing the user's permit signature, it is possible for the market maker to execute orders on undesirable terms (e.g. execute an order with a large premium) without a user's consent.

Recommendation To prevent such cases, it may be useful to either restrict users such that they may only have one pending order or to have a user approval stage where the user provides their approval after a market maker proposes the terms (after which the order is executed).

4.1.5 V-RAO-VUL-005: Positions can be Opened outside of OTCWrapper

Severity	Medium	Commit	4a230df
Type	Usability Issue	Status	Fixed
Files	`Controller.sol`		
Functions	operate		

This protocol builds on top of Rysk Finance's Gamma Protocol to track the executed option trades. It does so by creating naked vaults to store the collateral that is provided by market makers. Users, however, are not restricted to interacting with Ribbon's OTCWrapper to create these vaults as no restrictions were added to the Gamma Protocol other than to restrict access to liquidations.

```
1    function operate(Actions.ActionArgs[] memory _actions) external nonReentrant
     notFullyPaused {
2        (bool vaultUpdated, address vaultOwner, uint256 vaultId) = _runActions(
     _actions);
3        if (vaultUpdated) {
4            _verifyFinalState(vaultOwner, vaultId);
5            vaultLatestUpdate[vaultOwner][vaultId] = now;
6        }
7    }
```

Snippet 4.4: The unprotected `operate` function in the Controller

Impact The additional fees and restrictions enforced by the OTCWrapper can be bypassed by interacting directly with `Controller` and these liabilities are inherited by users of Ribbon's OTC protocol as all funds are located in a single pool. In addition, since liquidations have been restricted such that they can only be performed by the liquidation manager, Ribbon must ensure that they are capable of liquidating positions from users that did not interact with the OTCWrapper.

Recommendation Consider restricting access to the Controller so that only approved users can interact with the controller.

4.1.6 V-RAO-VUL-006: Market Makers can Front-run Order Execution

Severity	Medium	**Commit**	4a230df
Type	Frontrunning	**Status**	Acknowledged
Files	`OTCWrapper.sol`		
Functions	executeOrder		

For a market maker to execute an order, the user must provide them with their permit signature to pay the requested premium. The market maker then submits a transaction to execute the order. While the transaction is in the mempool, though, information about the order execution is exposed, including the user's signature.

```
function executeOrder(
    uint256 _orderID,
    Permit calldata _userSignature,
    Permit calldata _mmSignature,
    uint256 _premium,
    address _collateralAsset,
    uint256 _collateralAmount
) external nonReentrant {
    require(orderStatus[_orderID] == OrderStatus.Pending, "...");
    require(isWhitelistedMarketMaker[_msgSender()], "...");
    require(_userSignature.amount >= _premium, "...");

    Order memory order = orders[_orderID];

    require(_userSignature.acct == order.buyer, "...");
    require(block.timestamp <= order.openedAt.add(fillDeadline), "...");
    require(whitelist.isWhitelistedCollateral(_collateralAsset), "...");

    ...
}
```

Snippet 4.5: The `executeOrder` function that can be front-run

Impact With this information, another market maker can front-run the transaction to execute the order themselves. As the user could have provided the same signature to multiple market makers and many users likely only care that their order is executed, it's possible that this could go unnoticed. In addition, it could let less reputable market-makers out compete others and increase the likelihood of a shortfall.

Recommendation Include a step where the user approves the execution of an order

Developer Response While this is a possibility, we believe users will not be as concerned with the identity of the market maker that executes their order as with the fact that their order has been executed. We always trust our partners/market makers to be on their best behaviour. As frontrunning another market maker is highly visible, such behaviour may incur in reputation risk or potential loss of whitelist status to operate in the protocol.

4.1.7 V-RAO-VUL-007: Use Constant instead of Magic Number

Severity	Warning	**Commit**	4a230df
Type	Maintainability	**Status**	Fixed
Files	`MarginRequirements.sol`		
Functions	setInitialMargin		

The `MarginRequirements` contract defines the constant `MAX_INITIAL_MARGIN` to represent 100% when computing the percentage of the nominal value that must be collateralized. However there are locations where a magic number equivalent to the constant is used rather than the constant itself.

```
function setInitialMargin(
    address _underlying,
    address _collateralAsset,
    bool _isPut,
    address _account,
    uint256 _initialMargin
) external onlyOwner {
    require(
        _initialMargin > 0 && _initialMargin <= 100 * 10**2,
        "MarginRequirements: initial margin cannot be 0 or higher than 100%"
    );

    ...
}
```

Snippet 4.6: Location where a magic number is used over the constant

Impact If the developers change the value of this constant, some checks that use a magic number may be incorrect.

Recommendation Replace uses of magic numbers with appropriate constants.

4.1.8 V-RAO-VUL-008: No Constraints on Premium Fee

Severity	Warning	**Commit**	4a230df
Type	Configuration	**Status**	Fixed
Files	`OTCWrapper.sol`		
Functions	setFee		

Market makers charge a premium to users as compensation for executing their order. From this premium, the protocol takes a fee which is sent to the beneficiary. As it stands, when this fee is set by the contract owner, no validation is performed on the new fee.

```
function setFee(address _underlying, uint256 _fee) external onlyOwner {
    require(_underlying != address(0), "OTCWrapper: asset address cannot be 0");

    fee[_underlying] = _fee;
}
```

Snippet 4.7: Location where the fee is set by the owner

Impact As no validation is performed, it is possible that a configuration error could set the fee to an undesirable value. For example, if a fee is set to > `1e6` valid `executeOrder` transactions will revert.

Recommendation To reduce the possibility of configuration errors, place restrictions on the values to which `fee` can be set.

4.1.9 V-RAO-VUL-009: SafeMath Unnecessary in Solidity 0.8

Severity	Info	**Commit**	4a230df
Type	Gas Optimization	**Status**	Fixed
Files	`OTCWrapper.sol`		
Functions	N/A		

The OTCWrapper uses solidity version `0.8.10` which checks for overflows/underflows by default. As a result, the SafeMath library is no longer needed to prevent such issues but is still used by this contract.

Impact Using the SafeMath library in Solidity 0.8 wastes gas as it simply performs the requested operation.

4.1.10 V-RAO-VUL-010: Order Premium must Equal Signature Amount

Severity	Info	Commit	4a230df
Type	Data Validation	Status	Fixed
Files	`OTCWrapper.sol`		
Functions	executeOrder		

When validating the user's permit signature in `executeOrder` the protocol checks that `_userSignature.amount >= _premium`. When processing the permit, though, `_premium` is passed in as the permit amount causing the `permit` function to revert if `_premium` does not match the permit amount.

```
function executeOrder(
    uint256 _orderID,
    Permit calldata _userSignature,
    Permit calldata _mmSignature,
    uint256 _premium,
    address _collateralAsset,
    uint256 _collateralAmount
) external nonReentrant {
    require(orderStatus[_orderID] == OrderStatus.Pending, "...");
    require(isWhitelistedMarketMaker[_msgSender()], "...");
    require(_userSignature.amount >= _premium, "...");

    ...
}
```

Snippet 4.8: The check in `executeOrder` that doesn't enforce `_userSignature.amount == _premium`

Recommendation Check that `_userSignature.amount == _premium`