Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

RIBBON

GammaProtocol-OTC | Unwinding

Veridise Inc.

September 1, 2023

► **Prepared For:**

Ribbon Finance
https://www.ribbon.finance

► **Prepared By:**

Benjamin Sepanski
Alberto Gonzalez
Jon Stephens

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

► **Version History:**

Aug. 11, 2023	V1
Aug. 9, 2023	Initial Draft

Contents

1 Executive Summary

From Jul. 31, 2023 to Aug. 2, 2023, Ribbon Finance engaged Veridise to review the security of the new unwinding feature in GammaProtocol-OTC. GammaProtocol-OTC is a Solidity project which facilitates a market for OTC tokens, each representing an option. Whitelisted market makers are matched with would-be OTC buyers, putting down some (reputation-based) amount of collateral to take out a short position. This also included a signature-checking contract, based off of ERC 2612.

Compared to the previous version, which Veridise has audited previously, the new version adds a feature which allows OTC holders to sell their long positions to whitelisted market makers in an off-chain bid, and then recover the funds via signed permits. This audit focused exclusively on the added functionality.

Veridise conducted the assessment over 8 person-days, with 2 engineers reviewing code over 4 days on commit `0xbcbf34e8`. 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 GammaProtocol-OTC developers provided the source code of the GammaProtocol-OTC contracts for review. To facilitate the Veridise auditors' understanding of the code, the GammaProtocol-OTC developers provided a detailed design document outlining the purpose of variables, example execution, desired access controls, and desired properties. The source code also contained some documentation in the form of documentation comments on functions and storage variables.

The source code contained a test suite. The Veridise auditors noted this test suite was very comprehensive, testing the new functionality not only along "happy paths," but checking that the code reverts when expected as well. Several files in the source code also indicate that the developers use linting and static analysis tools such as Slither, solhint, and prettier.

Overall, the Veridise team assessed the code quality to be above average. The code was clear and well documented. Two key properties (such as the two listed below) do much to reduce the possibility of abusing signatures revealed in the mempool.

- Only an option buyer can submit a permit to unwind a position.
- If a market maker buys back their own short, they cannot sell it again.

The Ribbon Finance team also identified actions which should be prevented by their implementation, further helping to focus the audit efforts.

Summary of issues detected. The audit uncovered 9 issues, 0 of which are assessed to be of high or critical severity by the Veridise auditors. The Veridise auditors identified several low-severity issues, each pertaining to frontrunning (V-RBN-VUL-001, V-RBN-VUL-003, and V-RBN-VUL-002), as well as a number of minor issues. The GammaProtocol-OTC developers have fixed 8 of these issues. The remaining unfixed issue is an Info issue which has been partially fixed and does not have direct security implications.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the GammaProtocol-OTC.

First, to make clear the importance of the properties mentioned above (only an option buyer can submit an unwind, and a market maker cannot buy back their own short then resell it), Veridise auditors recommend additional documentation to explain why these are important will help readers of the code understand how the protocol's guarantees are enforced.

Second, Veridise auditors noted that the unwind permits are signed only over the ID of the order. Including the buyer and seller may add an additional layer of security to be extra sure that signed permits cannot be repurposed. See also V-RBN-VUL-003.

Third, Veridise auditors suggested adding an event or timelock to changes in the protocol fees to better protect market makers.

Finally, the Veridise auditors recommend to import contracts directly from OpenZeppelin (as a dependency of the project) whenever possible. See V-RBN-VUL-006.

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.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
GammaProtocol-OTC	`0xbcbf34e8`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 31 - Aug. 2, 2023	Manual & Tools	2	8 person-days

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	0	0
Low-Severity Issues	3	3
Warning-Severity Issues	2	2
Informational-Severity Issues	4	3
TOTAL	9	8

Table 2.4: Category Breakdown.

Name	Number
Maintainability	4
Frontrunning	3
Data Validation	1
Logic Error	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Ribbon Finance's implementation of the unwinding feature in GammaProtocol-OTC. In our audit, we sought to answer the following questions:

- Are position sellers guaranteed to receive the necessary funds upon sale?
- Is the position properly transferred upon a successful bid?
- Can signatures, once revealed to the mempool, be used to steal funds or adversely affect the buyer, seller, or pool?
- Can the sale be prevented by third-party actors?
- Are signatures properly validated?
- If a market maker buys an order they executed, can they redeem the order for more tokens than they are owed?
- Can market makers safely liquidate a position if they own both sides of it?
- Are Solidity best practices followed and common vulnerabilities avoided?

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 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 updates in `0x2e43ab4-0xbcbf34e8`, considering only the portions of `OTCWrapperV2.sol` which differ from `OTCWrapper.sol`. More specifically, the audit scope consists of:

- The diff between `OTCWrapperV2.sol` and `OTCWrapper.sol` in `contracts/core`.
- `contracts/packages/unwind-permit/UnwindPermit.sol`.
- `contracts/interfaces/otcWrapperInterfaces/UnwindPermitInterface.sol`.

The source code is provided by the GammaProtocol-OTC developers at commit `0xbcbf34e8`.

Methodology. Veridise auditors reviewed the reports of previous audits for GammaProtocol-OTC, inspected the provided tests, and read the GammaProtocol-OTC documentation. They then began a manual audit of the code assisted by static analyzers.

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-RBN-VUL-001	Frontrunners can grief USDC permit signatures	Low	Fixed
V-RBN-VUL-002	Frontrunners can grief unwind permit signatures	Low	Fixed
V-RBN-VUL-003	Frontrunning market maker can prevent liquidation	Low	Acknowledged
V-RBN-VUL-004	Hard-coded value depends on constant	Warning	Fixed
V-RBN-VUL-005	Hard-coded value depends on constant	Warning	Fixed
V-RBN-VUL-006	Code Recommendation: Link to OZ Implementation	Info	Partially Fixed
V-RBN-VUL-007	Lack of slippage protection	Info	Acknowledged
V-RBN-VUL-008	Test uses wrong function	Info	Fixed
V-RBN-VUL-009	UnwindPermit does not inherit interface	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-RBN-VUL-001: Frontrunners can grief USDC permit signatures

Severity	Low	**Commit**	bcbf34e
Type	Frontrunning	**Status**	Fixed
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	_deposit()		
Confirmed Fix At	a0daf13		

The `_deposit()` function verifies a permit signature if the asset is USDC (see below code snippet).

```
if (_asset == USDC) {
   // Sign for transfer approval
   IERC20Permit(USDC).permit(
      _signature.acct,
      address(this),
      _signature.amount,
      _signature.deadline,
      _signature.v,
      _signature.r,
      _signature.s
   );
}
```

Snippet 4.1: Signature checking performed in the `_deposit()` function.

Note, however, that anyone may call the `IERC20Permit(USDC).permit` method in the USDC contract. So, front-runners might grief the sale by making the function `_deposit` revert.

Impact Frontrunners may prevent any process that uses `_deposit()` from occurring. For example, unwinding.

Recommendation Check if the contract has sufficient allowance before calling `IERC20Permit(USDC).permit`.

Developer Response We have implemented the recommendation.

4.1.2 V-RBN-VUL-002: Frontrunners can grief unwind permit signatures

Severity	Low	**Commit**	bcbf34e
Type	Frontrunning	**Status**	Fixed
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	sellRedeemRights()		
Confirmed Fix At	dd35454		

The `sellRedeemRights()` function begins by verifying the permits of both the bidder and the seller.

```
UNWIND_PERMIT.checkOrderPermit(
    _sellerOrderSignature.acct,
    _sellerOrderSignature.orderID,
    _sellerOrderSignature.bidValue,
    _sellerOrderSignature.deadline,
    _sellerOrderSignature.v,
    _sellerOrderSignature.r,
    _sellerOrderSignature.s
);
UNWIND_PERMIT.checkOrderPermit(
    _bidderOrderSignature.acct,
    _bidderOrderSignature.orderID,
    _bidderOrderSignature.bidValue,
    _bidderOrderSignature.deadline,
    _bidderOrderSignature.v,
    _bidderOrderSignature.r,
    _bidderOrderSignature.s
);
```

Snippet 4.2: Beginning of the `sellRedeemRights()` function:

Note, however, that anyone may call the `UnwindPermit.checkOrderPermit` method.

```
function checkOrderPermit(
    address owner,
    uint256 orderID,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
```

Snippet 4.3: Signature of `checkOrderPermit()`

Frontrunners who wish to do so may grief the sale by verifying either signature, causing the `checkOrderPermit()` function to revert.

Impact Frontrunners may prevent unwinding from occurring.

Recommendation Give the `OTCWrapperV2` ownership over `UnwindPermit` and make `checkOrderPermit` owner-only. In this case, we also recommend that only short deadlines be accepted, since users will no longer be able to manually invalidate signatures.

A "short deadline" check can be added to `sellRedeemRights` to ensure signatures with long deadlines are unusable.

Whether the "ownership" change is made or not, we still recommend enforcing a "short deadline" requirement as a proactive measure to prevent misuse of signatures.

Developer Response We have added an owner to the `UnwindPermit` who can whitelist a single address. This whitelisted address is the only one which may validate signatures.

4.1.3 V-RBN-VUL-003: Frontrunning market maker can prevent liquidation

Severity	Low	**Commit**	bcbf34e
Type	Frontrunning	**Status**	Acknowledged
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	sellRedeemRights()		
Confirmed Fix At	N/A		

The `sellRedeemRights()` function begins by verifying the permits of both the bidder and the seller.

```
UNWIND_PERMIT.checkOrderPermit(
    _sellerOrderSignature.acct,
    _sellerOrderSignature.orderID,
    _sellerOrderSignature.bidValue,
    _sellerOrderSignature.deadline,
    _sellerOrderSignature.v,
    _sellerOrderSignature.r,
    _sellerOrderSignature.s
);
UNWIND_PERMIT.checkOrderPermit(
    _bidderOrderSignature.acct,
    _bidderOrderSignature.orderID,
    _bidderOrderSignature.bidValue,
    _bidderOrderSignature.deadline,
    _bidderOrderSignature.v,
    _bidderOrderSignature.r,
    _bidderOrderSignature.s
);
```

Snippet 4.4: Beginning of the `sellRedeemRights()` function:

A frontrunning market maker may prevent another bidder from purchasing redeem rights by signing their own permit and submitting it before the initial bidder.

Impact Malicious market makers could prevent other market makers from liquidating a position by preventing them from buying out their positions.

For example, consider the following scenario.

1. Market maker Alice has taken out a short position on volatileCoin against Bob.
2. Alice comes to believe that volatileCoin is going to do very well. Bob disagrees, so Alice convinces him to sell her the rights to the position so she can liquidate it.
3. Market maker Eve wishes to cause financial harm to Alice, but did not wish to drive the price higher so did not bid. Instead, she waits for the bidding to conclude.
4. Eve frontruns the transaction and replaces Alice's `_bidderOrderSignature` and USDC permit with her own.

Although Alice has won the auction so that she may liquidate (what she believes to be) a financially damaging holding, Eve prevented her from liquidating, forcing Alice to suffer the losses associated to her short (if they manifest).

Recommendation Include the bidder and the seller in the unwind permit.

Developer Response A malicious market maker could do this once. After this, they could be blacklisted by the Ribbon. The MM loses a trade, not any money directly.

4.1.4 V-RBN-VUL-004: Substitute hard-coded constant for FEE_PERCENT_MULTIPLIER

Severity	Warning	**Commit**	bcbf34e
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	sellRedeemRights()		
Confirmed Fix At	a0daf13		

In `sellRedeemRights` the variable `orderFee` is computed as follows:

```
uint256 orderFee = (_bidderOrderSignature.bidValue * unwindFee[order.underlying]) / 1
    e6; // divides by 1e6 as bidValue is expected to have 6 decimals (USDC)
```

Snippet 4.5: Computation of `orderFee`

The above computation uses `1e6` as a hard-coded value in the division in order to be consistent with the decimals of `unwindFee` .

Impact Future updates to `FEE_PERCENT_MULTIPLIER` will cause the `orderFee` to be computed incorrectly.

Recommendation Replace `1e6` with `FEE_PERCENT_MULTIPLIER`. And delete the comment:

```
// divides by 1e6 as bidValue is expected to have 6 decimals (USDC)
```

Since `1e6` comes from `unwindFee` and not from USDC.

Developer Response We have implemented the recommendation.

4.1.5 V-RBN-VUL-005: Hard-coded constant depends on FEE_PERCENT_MULTIPLIER

Severity	Warning	**Commit**	bcbf34e
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	_settleFunds()		
Confirmed Fix At	a0daf13		

In `_settleFunds()`, the decimals of the fee are combined with the decimals of the USDC as described in the below code comment.

```
// eg. fee = 4bps = 0.04% , then need to divide by 100 again so (( 4 / 100 ) / 100)
// after the above it is divided again by 1e2 which is the fee decimals
// multiplication by 1e8 is used to compensate the 8 decimals from USDC price
// when aggregated the multiplication becomes by 1e2
uint256 usdcPrice = oracle.getPrice(USDC);
require(usdcPrice > 0, "OTCWrapper: invalid USDC price");
uint256 orderFee = (_notional * (fee[_order.underlying]) * 1e2) / usdcPrice;
```

Snippet 4.6: Snippet from `_settleFunds()`

This value `1e2` is computed as

```
(1 / FEE_PERCENT_MULTIPLIER) * (1 / USDC_DECIMALS)
= USDC_DECIMALS / FEE_PERCENT_MULTIPLIER
= 1e2
```

However, if `FEE_PERCENT_MULTIPLIER` ever changes, this value will also need to change.

Impact Future updates to `FEE_PERCENT_MULTIPLIER` will cause the `orderFee` to be computed incorrectly.

Recommendation Replace `1e2` with `1e8 / FEE_PERCENT_MULTIPLIER`.

Developer Response We have implemented the recommendation.

4.1.6 V-RBN-VUL-006: Code Recommendation: Link to OZ Implementation

Severity	Info	**Commit**	bcbf34e
Type	Maintainability	**Status**	Partially Fixed
File(s)	`contracts/packages/unnwind-permit/UnwindPermit.sol`		
Location(s)	N/A		
Confirmed Fix At	dd35454		

The `UnwindPermit` contract is closely based on the OpenZeppelin ERC20Permit implementation. While this is indicated in documentation provided to the audit team, this is not indicated in the file itself.

Impact Future readers of the file may require additional context, or not realize that parts of the code have been heavily audited by multiple parties.

Recommendation Link to the OpenZeppelin implementation within the file.

We recommend doing the same in the EIP 712 implementation, and any other files imported from OpenZeppelin. Preferably, these would be imported directly via a dependency on OpenZeppelin.

Developer Response We have linked to the OpenZeppelin base implementation in the source code, but decided to keep a copy of OpenZeppelin's EIP712 rather than import it directly using OpenZeppelin as a dependency.

4.1.7 V-RBN-VUL-007: Lack of slippage protection in the premium for the MM

Severity	Info	**Commit**	bcbf34e
Type	Data Validation	**Status**	Acknowledged
File(s)	`contracts/core/OTCWrapperV2.sol`		
Location(s)	_settleFunds()		
Confirmed Fix At	N/A		

The `orderFee` takes into account the scenario when USDC depegs. It does that by querying the price `USD / USDC` and using it on the computation of the fee:

```
uint256 usdcPrice = oracle.getPrice(USDC);
require(usdcPrice > 0, "OTCWrapper: invalid USDC price");
uint256 orderFee = (_notional * (fee[_order.underlying]) * 1e2) / usdcPrice;

// transfer premium to market maker
IERC20(USDC).safeTransfer(_msgSender(), (_premium - orderFee));
```

Snippet 4.7: Snippet from `_settleFunds()`

Then `orderFee` is subtracted from `_premium` , and the result is sent to the MM. However, there is no upper bound in how much `orderFee` can grow. For example, in the event of a USDC depeg, `usdcPrice` will start getting smaller, increasing `orderFee` .

Impact The MM will not receive any premium when `orderFee ~= _premium`.

Recommendation Allow the MM to specify the minimum amount of premium to receive.

Developer Response We have decided not to implement a fix for this as the event of a depeg between the moment the function is called and the moment order premium is calculated is very unlikely, and a minimum premium is enforced in the frontend.

4.1.8 V-RBN-VUL-008: Test uses wrong function

Severity	Info	**Commit**	bcbf34e
Type	Logic Error	**Status**	Fixed
File(s)	`test/upgrades/otcWrapperV2.ts`		
Location(s)	N/A		
Confirmed Fix At	a0daf13		

The upgrade utility script checks the wrong field for `fillDeadline`.

```
1 const fillDeadlineAft = (await otcWrapperProxy.latestOrder()).toString()
```

Snippet 4.8: Snippet from test.

Impact If the `fillDeadline` storage variable were swapped with another, a storage collision may occur undetected.

Recommendation Fix the test to use `.fillDeadline()`.

Developer Response We have fixed the test to use the correct function.

4.1.9 V-RBN-VUL-009: UnwindPermit does not inherit interface

Severity	Info	**Commit**	bcbf34e
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/packages/unwind-permit/UnwindPermit.sol`		
Location(s)	N/A		
Confirmed Fix At	dd35454		

The `UnwindPermit` inherits from `EIP712`

```
contract UnwindPermit is EIP712 {
```

Snippet 4.9: Declaration of `UnwindPermit`.

but not from the `UnwindPermitInterface`.

```
interface UnwindPermitInterface {
    function checkOrderPermit(
        address owner,
        uint256 orderID,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) external;
}
```

Snippet 4.10: Definition of the `UnwindPermitInterface`.

Impact If the signature of `UnwindPermit.checkOrderPermit` changes, `solc` will not automatically require changes in uses of the interface (and vice versa).

Recommendation Have `UnwindPermit` inherit the `UnwindPermitInterface`.

Developer Response We have implemented the recommendation.

Glossary

ERC Ethereum Request for Comment. 19

ERC 20 The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 19

ERC 2612 An Ethereum Request for Comment (ERC) describing a permit extension for ERC 20-signed approvals. See `https://eips.ethereum.org/EIPS/eip-2612` for the full ERC. 1

Ethereum Request for Comment Peer-reviewed proposals describing application-level standards and conventions. Visit `https://eips.ethereum.org/erc` to learn more. 19

OpenZeppelin A security company which provides many standard implementations of common contract specifications. See `https://www.openzeppelin.com`. 2

prettier A code formatting tool, see `https://prettier.io/docs/en/integrating-with-linters.html` to learn more. 1

Slither A static analyzer for Solidity by Crytic, a subsidiary of Trail of Bits. See `https://github.com/crytic/slither` for more information. 1, 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.. 19

solhint An open-source project for linting Solidity code. See `https://protofire.github.io/solhint/` to learn more. 1

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. 1, 19