# SMART CONTRACT AUDIT REPORT

for

# RIBBON FINANCE



Prepared By: Shuxiao Wang

PeckShield

March 31, 2021

## Document Properties

| | |
|---|---|
| Client | Ribbon Finance |
| Title | Smart Contract Audit Report |
| Target | Ribbon Finance |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Xuxian Jiang, Huaguo Shi |
| Reviewed by | Jeff Liu |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | March 31, 2021 | Xuxian Jiang | Final Release |
| 1.0-rc1 | March 30, 2021 | Xuxian Jiang | Release Candidate #1 |
| 0.3 | March 26, 2021 | Xuxian Jiang | Add More Findings #2 |
| 0.2 | March 19, 2021 | Xuxian Jiang | Add More Findings #1 |
| 0.1 | March 15, 2021 | Xuxian Jiang | Initial Draft |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.


| | |
|---|---|
| Name | Shuxiao Wang |
| Phone | +86 173 6454 5338 |
| Email | contact@peckshield.com |

# Contents

# 1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the **Ribbon Finance** protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

## 1.1 About Ribbon Finance

`Ribbon Finance` is building on-chain option vaults that use smart contracts to automate various options strategies. Users can simply deposit their assets into a smart contract and will automatically start running a specific options strategy. The first product that `Ribbon` is creating is called a `Theta Vault`, which is a yield-generating strategy on ETH. `Theta Vaults` run a covered call strategy, which earns yield on a weekly basis through writing out of the money covered calls and collecting the premiums.

The basic information of `Ribbon Finance` is as follows:

Table 1.1: Basic Information of Ribbon Finance

| Item | Description |
|---:|---|
| Issuer | Ribbon Finance |
| Website | https://ribbon.finance/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | March 31, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/ribbon-finance/audit.git (93ef69f)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/ribbon-finance/audit.git (5311c6f)

## 1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full Audit Checklist

| Category | Checklist Items |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| **Semantic Consistency Checks** | Semantic Consistency Checks |
| **Advanced DeFi Scrutiny** | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logic** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Ribbon Finance` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 1 | ■ |
| Medium | 1 | ■ |
| Low | 2 | ■ ■ |
| Informational | 2 | ■ ■ |
| Total | 6 | |

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendation.

Table 2.1: Key Ribbon Finance Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Informational | Asset Consistency Check Between Instrument And Option | Coding Practices | Fixed |
| PVE-002 | Medium | Possible Costly Pool Tokens From Improper Initialization | Time and State | Fixed |
| PVE-003 | Low | Improved Sanity Checks For System Parameters | Coding Practices | Fixed |
| PVE-004 | High | Possible Sandwich/MEV Attacks To Collect Most Profits | Time And State | Fixed |
| PVE-005 | Informational | Hardcoded Decimal Assumption in purchaseWithZeroEx() | Coding Practices | Fixed |
| PVE-006 | Low | Accommodation of approve() Idiosyncrasies | Business Logic | Fixed |

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Asset Consistency Check Between Instrument And Option

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: RibbonCoveredCall
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

### Description

The Ribbon Finance protocol develops new on-chain option vaults that allow for automating various options strategies. At the core of the protocol are the provided instruments (e.g., RibbonCoveredCall) that provide the main interaction with protocol users.

In the following, we elaborate one specific function from the RibbonCoveredCall contract. This function setNextOption() can be used to set the next option against which a short can be opened. This function takes a single argument, i.e., optionTerms, then queries and records the corresponding option address (lines 244 – 246), and next updates the timestamp for the option to be eligible for activation (line 247).

```
237     /**
238      * @notice Sets the next option address and the timestamp at which the admin can call `rollToNextOption` to open a short for the option
239      * @param optionTerms is the terms of the option contract
240      */
241     function setNextOption(
242         ProtocolAdapterTypes.OptionTerms calldata optionTerms
243     ) external onlyManager nonReentrant {
244         address option = adapter.getOptionsAddress(optionTerms);
245         require(option != address(0), "!option");
246         nextOption = option;
247         nextOptionReadyAt = block.timestamp.add(delay);
248     }
```

Listing 3.1: RibbonCoveredCall::setNextOption()

Based on the protocol design, there is a consistency in terms of the underlying assets between the `instrument` and the supported `option`. Specifically, to properly link a new `option` with the `instrument`, it is natural for the two to operate on the same underlying asset. With that, we suggest to improve the `setNextOption()` routine by enforcing the implied consistency so that they operate on the same underlying asset and strike asset with the validity of option expiry at `nextOptionReadyAt` (line 247).

**Recommendation** Add necessary consistency checks on the same sets between the instrument And supported options.

**Status** The issue has been fixed by this commit: `3254263`.

## 3.2 Possible Costly Pool Tokens From Improper Initialization

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RibbonCoveredCall`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [3]

### Description

The `Ribbon Finance` protocol allows users to deposit supported assets and get in return `rETH-THETA` pool tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., `rETH-THETA`, extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This routine is used for participating users to deposit the supported assets (e.g., `WETH`) and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    /**
     * @notice Mints the vault shares to the msg.sender
     * @param amount is the amount of ‘asset‘ deposited
     */
    function _deposit(uint256 amount) private {
        uint256 totalWithDepositedAmount = totalBalance();
        require(totalWithDepositedAmount < cap, "Cap exceeded");

        // amount needs to be subtracted from totalBalance because it has already been
        // added to it from either IWETH.deposit and IERC20.safeTransferFrom
        uint256 total = totalWithDepositedAmount.sub(amount);

        // Following the pool share calculation from Alpha Homora: https://github.com/
            AlphaFinanceLab/alphahomora/blob/340653c8ac1e9b4f23d5b81e61307bf7d02a26e8/
            contracts/5/Bank.sol#L104
```

```
191         uint256 share =
192             total == 0 ? amount : amount.mul(totalSupply()).div(total);
193
194         emit Deposit(msg.sender, amount, share);
195
196         _mint(msg.sender, share);
197     }
```

Listing 3.2: RibbonCoveredCall::_deposit()

Specifically, when the pool is being initialized (line 191), the share value directly takes the value of `amount` (line 192), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `share = amount = 1 WEI`. With that, the actor can further deposit a huge amount of `WETH` assets with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `UniswapV2`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

**Recommendation** Revise current execution logic of `deposit()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

**Status** The issue has been fixed by the following commits: `c46afd2c` and `5311c6f2`.

## 3.3 Improved Sanity Checks For System Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RibbonCoveredCall`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Ribbon Finance` protocol is no exception. Specifically, if we examine the `BaseVault` contract, it has defined a number of protocol-wide risk parameters, e.g., `instantWithdrawalFee` and `cap`. In the following, we show the corresponding routines that allow for their changes.

```
    /**
     * @notice Sets the new withdrawal fee
     * @param withdrawalFee is the fee paid in tokens when withdrawing
     */
    function setWithdrawalFee(uint256 withdrawalFee) external onlyManager {
        instantWithdrawalFee = withdrawalFee;
    }
```

Listing 3.3: setWithdrawalFee()

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large fee parameter (say more than $100\%$) will revert any `withdrawal()` operation, effectively locking down user assets in the contract.

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

**Status** The issue has been fixed by this commit: `3254263`.

## 3.4 Possible Sandwich/MEV Attacks To Collect Most Profits

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GammaAdapter`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

### Description

As mentioned in Section 3.1, the `Ribbon Finance` protocol develops new on-chain option vaults that allow for automating various options strategies. The strategy involves adapters to interact with different yield-generating protocols. Specifically, if we examine the `GammaAdapter` implementation, there is a `swapExercisedProfitsToUnderlying()` routine that is part of `exercise()` function so that an expired option can be exercised to claim the option profits, if any.

```
    /**
     * @notice Swaps the exercised profit (originally in the collateral token) into the
         `underlying` token.
```

```
     *          This simplifies the payout of an option. Put options pay out in USDC, so
        we swap USDC back
     *          into WETH and transfer it to the recipient.
     * @param otokenAddress is the otoken's address
     * @param profitInCollateral is the profit after exercising denominated in the
        collateral - this could be a token with different decimals
     * @param recipient is the recipient of the underlying tokens after the swap
     */
    function swapExercisedProfitsToUnderlying(
        address otokenAddress,
        uint256 profitInCollateral,
        address recipient
    ) private returns (uint256 profitInUnderlying) {
        OtokenInterface otoken = OtokenInterface(otokenAddress);
        address collateral = otoken.collateralAsset();
        IERC20 collateralToken = IERC20(collateral);

        require(
            collateralToken.balanceOf(address(this)) >= profitInCollateral,
            "Not enough collateral from exercising"
        );

        IUniswapV2Router02 router = IUniswapV2Router02(UNISWAP_ROUTER);

        IWETH weth = IWETH(WETH);

        if (collateral == address(weth)) {
            profitInUnderlying = profitInCollateral;
            weth.withdraw(profitInCollateral);
            (bool success, ) = recipient.call{value: profitInCollateral}("");
            require(success, "Failed to transfer exercise profit");
        } else {
            address[] memory path = new address[](2);
            path[0] = collateral;
            path[1] = address(weth);

            uint256[] memory amountsOut =
                router.getAmountsOut(profitInCollateral, path);
            profitInUnderlying = amountsOut[1];
            require(profitInUnderlying > 0, "Swap is unprofitable");

            router.swapExactTokensForETH(
                profitInCollateral,
                profitInUnderlying,
                path,
                recipient,
                block.timestamp + SWAP_WINDOW
            );
        }
    }
```

Listing 3.4: GammaAdapter::swapExercisedProfitsToUnderlying()

We notice the collected profits are routed to `UniswapV2` in order to swap them to `WETH`. And the swap operation essentially does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the `GammaAdapter` contract in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

**Recommendation** Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

**Status** The issue has been fixed by this commit: `3254263`.

## 3.5 Hardcoded Decimal Assumption in purchaseWithZeroEx()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GammaAdapter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

### Description

In Section 3.4, we have examined the `GammaAdapter` contract that allows protocol users to directly buy `Opyn`-based options. In the following, we further examine the same contract, but on a different function, i.e., `purchaseWithZeroEx()`.

To elaborate, we show below the function implementation. This function is designed to allow for buying `otokens` using a `0x` order. Our analysis shows that this function makes an implicit assumption that may not always hold. In particular, the amount of `soldETH` is calculated by the following equation: `zeroExOrder.takerAssetAmount.mul(uint256(latestPrice)).div(10**6)`, which somehow implies the `sellTokenAddress` has the decimal of 6. With that, the calculation can then compute `soldETH` to be properly denominated at `WETH`.

```
    /**
     * @notice Purchases otokens using a 0x order struct
```

```
     * @param optionTerms is the terms of the option contract
     * @param zeroExOrder is the 0x order struct constructed using the 0x API response
         passed by the frontend.
     */
    function purchaseWithZeroEx(
        ProtocolAdapterTypes.OptionTerms calldata optionTerms,
        ProtocolAdapterTypes.ZeroExOrder calldata zeroExOrder
    ) external payable {
        require(
            msg.value >= zeroExOrder.protocolFee,
            "Value cannot cover protocolFee"
        );

        IUniswapV2Router02 router = IUniswapV2Router02(UNISWAP_ROUTER);

        address[] memory path = new address[](2);
        path[0] = WETH;
        path[1] = zeroExOrder.sellTokenAddress;

        (, int256 latestPrice, , , ) = USDCETHPriceFeed.latestRoundData();

        uint256 soldETH =
            zeroExOrder.takerAssetAmount.mul(uint256(latestPrice)).div(10**6);

        router.swapETHForExactTokens{value: soldETH}(
            zeroExOrder.takerAssetAmount,
            path,
            address(this),
            block.timestamp + SWAP_WINDOW
        );

        require(
            IERC20(zeroExOrder.sellTokenAddress).balanceOf(address(this)) >=
                zeroExOrder.takerAssetAmount,
            "Not enough takerAsset balance"
        );

        IERC20(zeroExOrder.sellTokenAddress).safeApprove(
            zeroExOrder.allowanceTarget,
            zeroExOrder.takerAssetAmount
        );

        require(
            address(this).balance >= zeroExOrder.protocolFee,
            "Not enough balance for protocol fee"
        );

        (bool success, ) =
            ZERO_EX_EXCHANGE_V3.call{value: zeroExOrder.protocolFee}(
                zeroExOrder.swapData
            );

```

```
        require(success, "0x swap failed");

        require(
            IERC20(zeroExOrder.buyTokenAddress).balanceOf(address(this)) >=
                zeroExOrder.makerAssetAmount,
            "Not enough buyToken balance"
        );

        emit Purchased(
            msg.sender,
            _name,
            optionTerms.underlying,
            soldETH.add(zeroExOrder.protocolFee),
            0
        );
    }
```

Listing 3.5: GammaAdapter::purchaseWithZeroEx()

Since the `0x` order structure is provided by the user, it is strongly suggested to enforce the assumption will always hold in all cases.

**Recommendation** Make the implicit assumption of `sellTokenAddress`'s decimal explicit.

**Status** The issue has been fixed by this commit: `3254263`.

## 3.6 Accommodation of approve() Idiosyncrasies

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GammaAdapter`
- Category: Business Logic [7]
- CWE subcategory: N/A

### Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.6: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. For example, the `GammaAdapter::createShort()` routine is designed to allow the `MARGIN_POOL` to pull funds from itself. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function createShort(
        ProtocolAdapterTypes.OptionTerms calldata optionTerms,
        uint256 depositAmount
    ) external override returns (uint256) {
        IController controller = IController(gammaController);
        uint256 newVaultID =
            (controller.getAccountVaultCounter(address(this))).add(1);

        address oToken = lookupOToken(optionTerms);
        require(oToken != address(0), "Invalid oToken");

        address collateralAsset = optionTerms.collateralAsset;
        if (collateralAsset == address(0)) {
            collateralAsset = WETH;
        }
        IERC20 collateralToken = IERC20(collateralAsset);

        uint256 collateralDecimals = assetDecimals(collateralAsset);
        uint256 mintAmount;

        if (optionTerms.optionType == ProtocolAdapterTypes.OptionType.Call) {
            mintAmount = depositAmount;
            if (collateralDecimals >= 8) {
                uint256 scaleBy = 10**(collateralDecimals - 8); // oTokens have 8
                    decimals
                mintAmount = depositAmount.div(scaleBy); // scale down from 10**18 to
                    10**8
```

```
                require(
                    mintAmount > 0,
                    "Must deposit more than 10**8 collateral"
                );
            }
        } else {
            mintAmount = wdiv(depositAmount, optionTerms.strikePrice)
                .mul(OTOKEN_DECIMALS)
                .div(10**collateralDecimals);
        }

        collateralToken.safeApprove(MARGIN_POOL, depositAmount);

        IController.ActionArgs[] memory actions =
            new IController.ActionArgs[](3);

        actions[0] = IController.ActionArgs(
            IController.ActionType.OpenVault,
            address(this), // owner
            address(this), // receiver -  we need this contract to receive so we can
                swap at the end
            address(0), // asset, otoken
            newVaultID, // vaultId
            0, // amount
            0, //index
            "" //data
        );

        actions[1] = IController.ActionArgs(
            IController.ActionType.DepositCollateral,
            address(this), // owner
            address(this), // address to transfer from
            collateralAsset, // deposited asset
            newVaultID, // vaultId
            depositAmount, // amount
            0, //index
            "" //data
        );

        actions[2] = IController.ActionArgs(
            IController.ActionType.MintShortOption,
            address(this), // owner
            address(this), // address to transfer to
            oToken, // deposited asset
            newVaultID, // vaultId
            mintAmount, // amount
            0, //index
            "" //data
        );

        controller.operate(actions);
```

```
        return mintAmount;
    }
```

Listing 3.7: GammaAdapter::createShort()

Meanwhile, it is important to highlight that the current implementation is safe as far as the one-time `safeApprove()` is always followed by the full transfer of the approved amount, which effectively reduces the approved amount back to zero. However, to accommodate various situations, it is always suggested to follow the convention of applying the `approve()` call twice to ensure the operation always runs smoothly.

**Recommendation** Accommodate the above-mentioned idiosyncrasy of `approve()`.

**Status** The issue has been fixed by this commit: `3254263`.

# 4 Conclusion

In this audit, we have analyzed the `Ribbon Finance` design and implementation. The system presents a unique, robust offering as a decentralized protocol for automating various options strategies. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.