SMART CONTRACT AUDIT REPORT

for

Venus Token Converter

Prepared By: Xiaomi Huang

PeckShield

September 27, 2023

Document Properties

Client	Venus
Title	Smart Contract Audit Report
Target	Venus Token Converter
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 27, 2023	Xuxian Jiang	Final Release
1.0-rc	September 4, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Venus Token Converter` 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 Venus Token Converter

The `Venus` protocol enables a complete algorithmic money market protocol on `BNB Smart Chain`. `Venus` enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. The audited `Venus Token Converter` supports the conversion from the generated protocol incomes and distribution to different targets (`RiskFund`, `Prime`, `VTreasury`, or `XVSVault`). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Venus Token Converter

Item	Description
Name	Venus
Website	https://venus.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 27, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. This audit covers on the changes from `PR #9`: `https://github.com/VenusProtocol/protocol-reserve/pull/9`.

- https://github.com/VenusProtocol/protocol-reserve.git (cc3d674)

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

- https://github.com/VenusProtocol/protocol-reserve.git (5d1c6b3)

1.2 About PeckShield

PeckShield Inc. [12] 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 OWASP Risk Rating Methodology [11]:

- 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 List of Check Items

Category	Check Item
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 list of check 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) [10], 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.

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 Logics	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 design and implementation of the `Venus Token Converter` 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	3	
Medium	1	
Low	2	
Informational	0	
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 that 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 3 high-severity vulnerabilities, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Venus Token Converter Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Public Exposure of State-Modifying Functions in RiskFundConverter	Security Features	Resolved
PVE-002	Medium	Possible Underflow in RiskFundConverter::postSweepToken()	Business Logic	Resolved
PVE-003	Low	Improved Precision in AbstractTokenConverter::getAmountIn()	Numeric Errors	Resolved
PVE-004	Low	Trust Issue of Admin Keys	Security Features	Resolved
PVE-005	High	Inconsistent Storage Layout For RiskFund Upgrade	Coding Practices	Resolved
PVE-006	High	Incorrect RiskFundConverter::postConversionHook() Logic	Business Logic	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Public Exposure of State-Modifying Functions in RiskFundConverter

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `RiskFundConverter`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The audited `Venus Token Converter` contract converts and distributes the protocol incomes to different recipients. While examining the specific `RiskFundConverter` contract, we notice the exposure of a public function and this exposure may be abused to corrupt the intended token conversion functionality.

In the following, we show the code snippet from the exposed `postSweepToken()` routine. This routine is designed to be an internal helper for the public `sweepToken()` function. In other words, this routine should not be exposed to public. Its public exposure without any caller authentication will corrupt the internal asset states and thus cripple the token conversion functionality.

```
    function postSweepToken(address tokenAddress, uint256 amount) public override {
        uint256 balance = IERC20Upgradeable(tokenAddress).balanceOf(address(this));
        uint256 balanceDiff = balance - assetsReserves[tokenAddress];

        uint256 amountDiff = amount - balanceDiff;
        if (amountDiff > 0) {
            address[] memory pools = getPools(tokenAddress);
            uint256 assetReserve = assetsReserves[tokenAddress];
            for (uint256 i; i < pools.length; ++i) {
                uint256 poolShare = (poolsAssetsReserves[pools[i]][tokenAddress] *
                    EXP_SCALE) / assetReserve;
                if (poolShare == 0) continue;
                updatePoolAssetsReserve(pools[i], tokenAddress, amount, poolShare);
```

```
            }
            assetsReserves[tokenAddress] -= amountDiff;
        }
    }
```

Listing 3.1: `RiskFundConverter::postSweepToken()`

Recommendation Declare the above `postSweepToken()` routine as an internal function.

Status This issue has been fixed in the following commit: `a1e6697`.

3.2 Possible Underflow in RiskFundConverter::postSweepToken()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `RiskFundConverter`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

In last Section 3.1, we have examined the public exposure of an internal helper routine. In this section, we further examine the logic of this helper and notice its logic needs to be improved.

Specifically, we show below the code snippet from the `postSwepToken()` routine. Within this routine, there is a `assetsReserves` state that keeps track of the amount for each asset that is transferred to `ProtocolShareReserve` combined (for all pools). And the information about the pool-specific asset amount in `ProtocolShareReserve` is saved in another state `poolsAssetsReserves`. It comes to our attention that when this helper `postSweepToken()` is called, the `balanceDiff` variable computes the amount of tokens that is free to sweep. If the requested amount is larger than `balanceDiff`, there is a need to reduce the pool-specific asset reserve to fulfill the sweep request. However, the calculation of `amount - balanceDiff` (line 177) may lead to a possible arithmetic underflow. Moreover, the pool-specific asset reserve needs to be updated as `updatePoolAssetsReserve(pools[i], tokenAddress, amountDiff, poolShare)`, instead of the current approach of `updatePoolAssetsReserve(pools[i], tokenAddress, amount, poolShare)` (line 184).

```
173     function postSweepToken(address tokenAddress, uint256 amount) public override {
174         uint256 balance = IERC20Upgradeable(tokenAddress).balanceOf(address(this));
175         uint256 balanceDiff = balance - assetsReserves[tokenAddress];
176 
177         uint256 amountDiff = amount - balanceDiff;
178         if (amountDiff > 0) {
```

```
            address[] memory pools = getPools(tokenAddress);
            uint256 assetReserve = assetsReserves[tokenAddress];
            for (uint256 i; i < pools.length; ++i) {
                uint256 poolShare = (poolsAssetsReserves[pools[i]][tokenAddress] *
                    EXP_SCALE) / assetReserve;
                if (poolShare == 0) continue;
                updatePoolAssetsReserve(pools[i], tokenAddress, amount, poolShare);
            }
            assetsReserves[tokenAddress] -= amountDiff;
        }
    }
```

Listing 3.2: `RiskFundConverter::postSweepToken()`

Recommendation Revise the above `postSweepToken()` logic to properly keep track of the pool-specific reserve.

Status This issue has been fixed in the following commit: `fd20124`.

3.3 Improved Precision in AbstractTokenConverter::getAmountIn()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `AbstractTokenConverter`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the mixed multiplication (`mul`) and division (`div`).

In particular, we use the `AbstractTokenConverter::getAmountIn()` as an example. This routine is used to calculate the amount of input tokens sender would send on receiving the requested `amountOutMantissa` of output tokens.

```
    function getAmountIn(
        uint256 amountOutMantissa,
        address tokenAddressIn,
        address tokenAddressOut
    ) public returns (uint256 amountConvertedMantissa, uint256 amountInMantissa) {
```

```
427         if (amountOutMantissa == 0) {
428             revert InsufficientInputAmount();
429         }

431         ConversionConfig memory configuration = convertConfigurations[tokenAddressIn][
                tokenAddressOut];

433         if (!configuration.enabled) {
434             revert ConversionConfigNotEnabled();
435         }

437         priceOracle.updateAssetPrice(tokenAddressIn);
438         priceOracle.updateAssetPrice(tokenAddressOut);

440         uint256 maxTokenOutReserve = balanceOf(tokenAddressOut);
441         uint256 tokenInUnderlyingPrice = priceOracle.getPrice(tokenAddressIn);
442         uint256 tokenOutUnderlyingPrice = priceOracle.getPrice(tokenAddressOut);

444         /// amount of tokenAddressOut after including incentive
445         uint256 conversionWithIncentive = MANTISSA_ONE + configuration.incentive;
446         /// conversion rate after considering incentive(conversionWithIncentive)
447         uint256 tokenInToOutConversion = (tokenInUnderlyingPrice *
                conversionWithIncentive) / tokenOutUnderlyingPrice;

449         amountInMantissa = ((amountOutMantissa * EXP_SCALE) / tokenInToOutConversion);
450         amountConvertedMantissa = amountOutMantissa;

452         /// If contract has less Liquity for tokenAddressOut than amountOutMantissa
453         if (maxTokenOutReserve < amountOutMantissa) {
454             amountInMantissa = ((maxTokenOutReserve * EXP_SCALE) /
                    tokenInToOutConversion);
455             amountConvertedMantissa = maxTokenOutReserve;
456         }
457     }
```

Listing 3.3: `AbstractTokenConverter::getAmountIn()`

We notice the calculation of the resulting amountInMantissa (line 449) involves mixed multiplication and devision. For improved precision, it is better to round-up the computation in favor of the protocol. Currently, it simply takes a round-down approach to compute the result. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note the `getAmountOut()` routine can be similarly improved.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commits: `92342eb` and `057dcee`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the new Venus Token Converter protocol, there is a privileged owner as well as the associated access control module, i.e., AccessControlledV8. They play a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role authorization). They may also affect the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine the privileged accounts and their related privileged accesses in current contracts.

```
    function setConvertibleBaseAsset(address convertibleBaseAsset_) external onlyOwner {
        ensureNonzeroAddress(convertibleBaseAsset_);
        emit ConvertibleBaseAssetUpdated(convertibleBaseAsset, convertibleBaseAsset_);
        convertibleBaseAsset = convertibleBaseAsset_;
    }

    /// @dev Risk fund converter setter
    /// @param riskFundConverter_ Address of the risk fund converter
    /// @custom:event RiskFundConverterUpdated emit on success
    /// @custom:error ZeroAddressNotAllowed is thrown when risk fund converter address is zero
    function setRiskFundConverter(address riskFundConverter_) external onlyOwner {
        ensureNonzeroAddress(riskFundConverter_);
        emit RiskFundConverterUpdated(riskFundConverter, riskFundConverter_);
        riskFundConverter = riskFundConverter_;
    }

    /// @dev Shortfall contract address setter
    /// @param shortfallContractAddress_ Address of the auction contract
    /// @custom:error ZeroAddressNotAllowed is thrown when shortfall contract address is zero
    function setShortfallContractAddress(address shortfallContractAddress_) external onlyOwner {
        ensureNonzeroAddress(shortfallContractAddress_);
        emit ShortfallContractUpdated(shortfall, shortfallContractAddress_);
        shortfall = shortfallContractAddress_;
    }
```

Listing 3.4: Example Privileged Operations in the RiskFundV2 Contract

If the privileged admins are managed by a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been resolved as the `owner` is managed by the `Venus: Timelock` contract deployed at `0x939bd8d64c0a9583a7dcea9933f7b21697ab6396`.

3.5 Inconsistent Storage Layout For RiskFund Upgrade

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: `RiskFund`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

The current `Venus Token Converter` protocol will upgrade an earlier contract `RiskFundV1` contract. Naturally, there is a requirement on the storage consistency between the old version and the new version. Our analysis shows that the new version has a different storage layout from the old version and this inconsistency needs to be resolved before the upgrade.

To elaborate, we show below the storage layout of of the new version `RiskFundV2Storage`. We notice it inherits five states from the earlier version, i.e., `pancakeSwapRouter`, `minAmountToConvert`, `convertibleBaseAsset`, `shortfall`, and `poolReserves`. However, our analysis on the old version indicates the following layout: `convertibleBaseAsset`, `shortfall`, `pancakeSwapRouter`, and `minAmountToConvert`. Beside the different layout, we also notice an extra `poolReserves` state that does not show up in the old version. Last but not least, for consistency, the `RiskFundV2` contract inherits from `Ownable2StepUpgradeable`, `AccessControlledV8`, `RiskFundV2Storage`, and `ReentrancyGuardUpgradeable`. The first `Ownable2StepUpgradeable` inheritance is redundant as it is already part of `AccessControlledV8` inheritance.

```
contract RiskFundV1Storage is ReserveHelpersStorage, MaxLoopsLimitHelpersStorage {
    /// @notice This state is deprecated, using it to prevent storage collision
    address private pancakeSwapRouter;
    /// @notice This state is deprecated, using it to prevent storage collision
    uint256 private minAmountToConvert;

    address public convertibleBaseAsset;
    address public shortfall;

    /// @notice Store base asset's reserve for specific pool
    mapping(address => uint256) public poolReserves;
}

/// @title RiskFundV2Storage
/// @author Venus
/// @dev Risk fund V2 storage
contract RiskFundV2Storage is RiskFundV1Storage {
    /// @notice Risk fund converter address
    address public riskFundConverter;
}
```

Listing 3.5: Key Storage States Defined in `RiskFundV2`

```
contract RiskFund is AccessControlledV8, ExponentialNoError, ReserveHelpers,
    MaxLoopsLimitHelper, IRiskFund {
    using SafeERC20Upgradeable for IERC20Upgradeable;
    address public convertibleBaseAsset;
    address public shortfall;
    address public pancakeSwapRouter;
    uint256 public minAmountToConvert;
    ...
}
```

Listing 3.6: Key Storage States Defined in `RiskFundV1`

Recommendation Ensure the storage consistency between the old version and the new version.

Status This issue has been fixed in the following commit: `74f9cb2`.

3.6 Incorrect RiskFundConverter::postConversionHook() Logic

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `RiskFundConverter`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.1, the `Venus Token Converter` contract converts and distributes the protocol incomes to different recipients. While examining the token conversion logic, we notice there is a post-conversion hook and this hook logic needs to be revisited.

In the following, we show the code snippet from the related `postConversionHook()` routine inside the `RiskFundConverter` contract. This routine is designed to perform bookkeeping tasks after the conversion. However, we notice the current implementation mixes `tokenInAddress` with `tokenOutAddress` and updates the wrong asset reserve numbers as well. Specifically, the `amountIn` is expected to be the protocol incoming amount, which will be sent to the `destinationAddress`.

```
    function postConversionHook(
        address tokenInAddress,
        address tokenOutAddress,
        uint256 amountIn,
        uint256 amountOut
    ) internal override {
        address[] memory pools = getPools(tokenInAddress);
        uint256 assetReserve = assetsReserves[tokenInAddress];
        for (uint256 i; i < pools.length; ++i) {
            uint256 poolShare = (poolsAssetsReserves[pools[i]][tokenInAddress] *
                EXP_SCALE) / assetReserve;
            if (poolShare == 0) continue;
            updatePoolAssetsReserve(pools[i], tokenInAddress, amountIn, poolShare);
            uint256 poolAmountOutShare = (poolShare * amountOut) / EXP_SCALE;
            IRiskFund(destinationAddress).updatePoolState(pools[i], tokenOutAddress,
                poolAmountOutShare);
        }

        assetsReserves[tokenInAddress] -= amountIn;
    }
```

Listing 3.7: `RiskFundConverter::postSweepToken()`

Recommendation Revise the above `postConversionHook()` routine to properly update the token amount. An updated version is shown in the following:

```
    function postConversionHook(
```

```
        address tokenInAddress,
        address tokenOutAddress,
        uint256 amountIn,
        uint256 amountOut
    ) internal override {
        address[] memory pools = getPools(tokenOutAddress);
        uint256 assetReserve = assetsReserves[tokenOutAddress];
        for (uint256 i; i < pools.length; ++i) {
            uint256 poolShare = (poolsAssetsReserves[pools[i]][tokenOutAddress] *
                EXP_SCALE) / assetReserve;
            if (poolShare == 0) continue;
            updatePoolAssetsReserve(pools[i], tokenOutAddress, amountOut, poolShare);
            uint256 poolAmountInShare = (poolShare * amountIn) / EXP_SCALE;
            IRiskFund(destinationAddress).updatePoolState(pools[i], tokenInAddress,
                poolAmountInShare);
        }

        assetsReserves[tokenOutAddress] -= amountOut;
    }
```

Listing 3.8: `RiskFundConverter::postSweepToken()`

Status This issue has been fixed in the following commit: `15c252d`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Venus Token Converter` protocol, which allows for the conversion from the generated protocol incomes and distribution to different targets (`RiskFund`, `Prime`, `VTreasury`, or `XVSVault`). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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