# SMART CONTRACT AUDIT REPORT

for

# Cygnus Finance

Prepared By: Xiaomi Huang

PeckShield

December 11, 2023

## Document Properties

| | |
|---|---|
| Client | Cygnus Finance |
| Title | Smart Contract Audit Report |
| Target | Cygnus |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Colin Zhong, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | December 11, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc1 | December 7, 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 Cygnus 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 audited protocol 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 Cygnus

Cygnus is a native stablecoin protocol on Base by collateralizing short-term US Treasury bonds to mint an interest-bearing stablecoin cgUSD. Users are allowed to mint cgUSD using assets from multiple chains. Through the rebase mechanism, cgUSD passes on the pure interest income from the US Treasury bonds to users. As interest is distributed, the balance of cgUSD increases correspondingly. The total supply of cgUSD is kept in sync with the net value of its asset portfolio, including on-chain stablecoins, off-chain US Treasury bonds, and interest, on every New York banking day. This ensures that Cygnus consistently supports the redemption of cgUSD for USDC at a 1:1 ratio. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Cygnus

| Item | Description |
|---:|---|
| Name | Cygnus Finance |
| Type | EVM Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | December 11, 2023 |

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

- https://github.com/arks-labs/cygnus-contracts.git (ef629fd)

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

- https://github.com/arks-labs/cygnus-contracts.git (33637b0)

## 1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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.

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 |

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.

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) [5], 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 `Cygnus` implementation. 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 | 0 | |
| Medium | 1 | ■ |
| Low | 2 | ■ ■ |
| Informational | 1 | ■ |
| Total | 4 | |

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 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Cygnus Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Revisited Caller Validation in Burner | Security Features | Resolved |
| PVE-002 | Low | Asset Consistency Enforcement in WithdrawVault | Coding Practices | Resolved |
| PVE-003 | Medium | Trust Issue of Admin Keys | Security Features | Mitigated |
| PVE-004 | Informational | Revisited Bunker Mode Design in WithdrawQueue | Business Logic | Resolved |

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 Revisited Caller Validation in Burner

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Burner`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

### Description

The `Cygnus` protocol has a dedicated `Burner` contract for `CgUSD` burning requests scheduling. Naturally, this contract exports a number of privileged functions. In the process of reviewing these functions, we notice two of them may be relaxed for their access control restriction.

To elaborate, we show below the code snippet of these two routines, i.e., `requestBurnMyStTokenForCover()` and `requestBurnMyStToken()`. As their names indicate, they are used to request the burn of certain `CgUSD` tokens. Since these tokens are transferred from the calling user, we can simply avoid the caller verification guarded with the `requiresAuth` modifier.

```
    function requestBurnMyStTokenForCover(uint256 _stTokenAmountToBurn) external requiresAuth {
        IERC20(stToken).safeTransferFrom(msg.sender, address(this), _stTokenAmountToBurn);
        uint256 sharesAmount = IStToken(stToken).convertToShares(_stTokenAmountToBurn);
        _requestBurn(sharesAmount, _stTokenAmountToBurn, true);
    }

    function requestBurnMyStToken(uint256 _stTokenAmountToBurn) external requiresAuth {
        IERC20(stToken).safeTransferFrom(msg.sender, address(this), _stTokenAmountToBurn);
        uint256 sharesAmount = IStToken(stToken).convertToShares(_stTokenAmountToBurn);
        _requestBurn(sharesAmount, _stTokenAmountToBurn, false);
    }
```

Listing 3.1: Burner::requestBurnMyStTokenForCover()/requestBurnMyStToken()

**Recommendation** Relax the call restriction in the the above two routines.

**Status** This issue has been fixed in the following commit: 33637b0.

## 3.2 Asset Consistency Enforcement in WithdrawVault

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WithdrawVault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### Description

The `Cygnus` protocol has a number of contracts that work closely to ensure `cgUSD` can always be redeemed `USDC` at a `1:1` ratio. While examining the related contracts, we notice the asset consistency can be better enforced.

To elaborate, we show below the constructors of three related contracts: `CgUSD`, `WithdrawVault`, and `WithdrawQueue`. These three contracts have the common `asset`. However, their consistency is not being enforced. With that, we can enhance them by explicitly enforcing their assets are identical, i.e., `require( ICgUSD(_cGUSD).asset()== _asset)` or `require( ICgUSD(stToken).asset()== _underlyingToken )`.

```
    constructor(
        address _asset,
        address _owner,
        Authority _authority
    ) Auth(_owner, _authority) {
        asset = _asset;
    }
```

Listing 3.2: `CgUSD::constructor()`

```
    constructor(
        address _asset,
        address _cgUSD,
        address _treasury,
        address _owner,
        Authority _authority
    ) Auth(_owner, _authority) {
         if (_cgUSD == address(0)) {
            revert LidoZeroAddress();
        }
        if (_treasury == address(0)) {
            revert TreasuryZeroAddress();
        }
```

```
        asset = _asset;
        CGUSD = _cgUSD;
        TREASURY = _treasury;
    }
```

Listing 3.3: WithdrawVault::constructor()

```
    constructor(
        address _wstToken,
        address _underlyingToken,
        address _owner,
        address _authority
    ) Auth(_owner, Authority(_authority)) {
        wstToken = _wstToken;
        underlyingToken = _underlyingToken;
        stToken = IWstToken(_wstToken).stToken();
        _initialize();
    }
```

Listing 3.4: WithdrawQueue::constructor()

**Recommendation** Revise the above constructors to ensure they share the same `asset`.

**Status** This issue has been fixed in the following commit: 33637b0.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

### Description

In the `Cygnus` protocol, there is a privileged account (`owner`). This account plays critical roles in governing and regulating the protocol-wide operations (e.g., configure protocol parameters and upgrade protocol implementations). Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the `CgUSD` contract as an example and show the representative functions potentially affected by the privileged account.

```
    function invest(address _to, uint256 _assetsAmount) external requiresAuth {
        IERC20(asset).safeTransfer(_to, _assetsAmount);

        uint256 postBufferedAssets = _getBufferedAssets() - _assetsAmount;
```

```
        uint256 postInvestedAssets = _getInvestedAssets() + _assetsAmount;
        _setBufferedAssets(postBufferedAssets);
        _setInvestedAssets(postInvestedAssets);
        emit Invested(_assetsAmount, postBufferedAssets, postInvestedAssets);
    }

    function resume() external requiresAuth {
        _unpause();
    }

    function pause() external requiresAuth {
        _pause();
    }
```

Listing 3.5: Example Privileged Operations in `CgUSD`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

**Recommendation** Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

**Status** The issue has been mitigated as the team confirmed that all the privileged accounts will be multi-sig wallets.

## 3.4 Revisited Bunker Mode Design in WithdrawQueue

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `WithdrawQueue`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

### Description

`Cygnus` has a key `WithdrawQueue` contract to handle `cgUSD` withdrawal request queue. While reviewing its logic, we notice the presence of a so-called `bunker` mode. However, the exact consequence of enabling or disabling the `bunker` mode remains unclear.

To elaborate, we show below the implementation of the related `onOracleReport()` routine. This routine is designed to update `bunker` mode state as well as the last report timestamp on oracle report. However, this routine is not actually invoked in other contracts and is presumably exercised by an external entity (guarded with `requiresAuth`). Moreover, after entering the `bunker` mode, the protocol itself does not specify the exact consequence or the impact on staking users and their funds.

```
    function onOracleReport(bool _isBunkerModeNow, uint256 _bunkerStartTimestamp, uint256 _currentReportTimestamp)
        external
        requiresAuth
    {
        if (_bunkerStartTimestamp >= block.timestamp) revert InvalidReportTimestamp();
        if (_currentReportTimestamp >= block.timestamp) revert InvalidReportTimestamp();

        _setLastReportTimestamp(_currentReportTimestamp);

        bool isBunkerModeWasSetBefore = isBunkerModeActive();

        if (_isBunkerModeNow != isBunkerModeWasSetBefore) {
            // write previous timestamp to enable bunker or max uint to disable
            if (_isBunkerModeNow) {
                BUNKER_MODE_SINCE_TIMESTAMP_POSITION.setStorageUint256(_bunkerStartTimestamp);

                emit BunkerModeEnabled(_bunkerStartTimestamp);
            } else {
                BUNKER_MODE_SINCE_TIMESTAMP_POSITION.setStorageUint256(BUNKER_MODE_DISABLED_TIMESTAMP);

                emit BunkerModeDisabled();
            }
        }
    }
```

Listing 3.6: `WithdrawQueue::onOracleReport()`

**Recommendation** Revise the `bunker` mode design to make it explicit to the staking users.

**Status** This issue has been fixed in the following commit: 33637b0.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the `Cygnus` protocol, which is a native stablecoin protocol on `Base`. It collateralizes short-term `US Treasury` bonds to mint an interest-bearing stablecoin `cgUSD`. Users are allowed to mint `cgUSD` using assets from multiple chains. Through the rebase mechanism, `cgUSD` passes on the pure interest income from the `US Treasury` bonds to users. As interest is distributed, the balance of `cgUSD` increases correspondingly. The total supply of `cgUSD` is kept in sync with the net value of its asset portfolio, including on-chain stablecoins, off-chain `US Treasury` bonds, and interest, on every `New York` banking day. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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.

PeckShield

# References


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

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

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

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

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

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

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


PeckShield