# SMART CONTRACT AUDIT REPORT

for

# Astherus USDF Earn

Prepared By: Xiaomi Huang

PeckShield

December 3, 2024

## Document Properties

| | |
|---|---|
| Client | Astherus |
| Title | Smart Contract Audit Report |
| Target | Astherus USDF Earn |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Daisy Cao, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | December 3, 2024 | Xuxian Jiang | Final Release |
| 1.0-rc | November 30, 2024 | Xuxian Jiang | Release Candidate #1 |

## Contact

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



| | |
|---|---|
| Name | Xiaomi Huang |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related source code of the Astherus USDF Earn 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 Astherus USDF Earn

Astherus USDF Earn is a novel platform within the Astherus ecosystem, designed to aggregate a diverse array of earning strategies across multiple blockchains and protocols. It simplifies the process of asset staking with a user-friendly, one-click feature, enabling users to effortlessly engage in the most lucrative opportunities available in the decentralized finance (DeFi) space. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Astherus USDF Earn

| Item | Description |
|---|---|
| Name | Astherus |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | December 3, 2024 |

- https://github.com/astherus-contract/ass-usdf-earn-contract.git (c4b4a48)

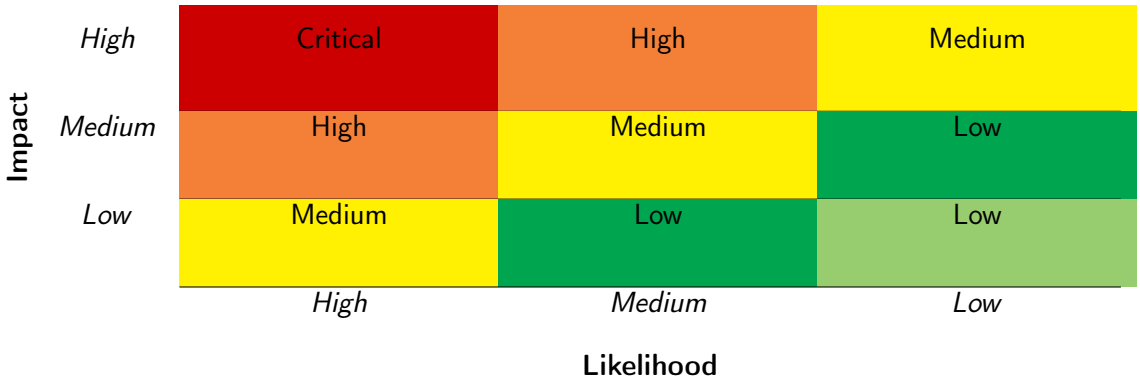
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/astherus-contract/ass-usdf-earn-contract.git (a660303, 7c20b84)

## 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




## 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.

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 accordingly classified into four categories, 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

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 |

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. 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 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 `Astherus USDF Earn` 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 | 0 | |
| Medium | 0 | |
| Low | 2 | ■■ |
| Informational | 0 | |
| Total | 2 | |

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 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Possible Costly Exchange Price From Unintended Pool Initialization | Time and State | Resolved |
| PVE-002 | Low | Trust Issue of Admin Keys | Security Features | Mitigated |

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 Possible Costly Exchange Price From Unintended Pool Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `asUSDFEarn`
- Category: Time and State [4]
- CWE subcategory: CWE-362 [2]

### Description

The `asUSDFEarn` contract allows users to deposit supported assets and get in return the share to represent the pool ownership. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool share extremely expensive and bring hurdles (or even causes loss) for later depositors.

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

```
100     function _mintasUSDF(uint256 amountIn) private {
101         require(USDFAddress != address(0), "USDFAddress cannot be a zero address");
102         require(asUSDFAddress != address(0), "asUSDFAddress cannot be a zero address");
103         require(USDFDepositEnabled == true, "Deposit is paused");
104         require(amountIn > 0, "invalid amount");
106         //getting exchange price must be in front of transferToVault.
107         uint256 exchange_price = exchangePrice();
109         amountIn = _transferToVault(msg.sender, USDFAddress, amountIn);
110         uint256 asUSDFAmount = amountIn * EXCHANGE_PRICE_DECIMALS / exchange_price;
111         require(asUSDFAmount > 0, "invalid amount");
113         IAs(asUSDFAddress).mint(msg.sender, asUSDFAmount);
```

```
114         emit MintasUSDF(msg.sender, USDFAddress, asUSDFAddress, amountIn, asUSDFAmount,
                exchange_price);
115     }
```

Listing 3.1: asUSDFEarn::_mintasUSDF()

```
125     function exchangePrice() public view returns (uint256){
126         require(USDFAddress != address(0), "USDFAddress cannot be a zero address");
127         require(asUSDFAddress != address(0), "asUSDFAddress cannot be a zero address");

129         IERC20 USDFToken = IERC20(USDFAddress);
130         uint256 USDFBalance = USDFToken.balanceOf(address(this));

132         IERC20 asUSDFToken = IERC20(asUSDFAddress);
133         uint256 totalSupply = asUSDFToken.totalSupply();
134         if (totalSupply <= 0){
135             return EXCHANGE_PRICE_DECIMALS;
136         }
137         if (USDFBalance <= 0){
138             return EXCHANGE_PRICE_DECIMALS;
139         }
140         return Math.mulDiv(USDFBalance,EXCHANGE_PRICE_DECIMALS,totalSupply);
141     }
```

Listing 3.2: asUSDFEarn::exchangePrice()

Specifically, when the pool is being initialized, the exchange price is calculated as EXCHANGE_PRICE_DECIMALS (line 135) and the share value directly takes the value of amountIn (line 109), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated asUSDFAmount = amountIn = 1 WEI. With that, the actor can further deposit a huge amount of the underlying assets with the goal of making the pool share extremely expensive.

An extremely expensive pool share can be very inconvenient to use as a small number of 1 Wei may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets.

This is a known issue that has been mitigated in popular Uniswap. 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 deposit logic 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 resolved as the pool has been properly initialzied without being empty, and admin have enough asset deposit in it.

## 3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

### Description

In the `Astherus USDF Earn` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, assign roles, and upgrade proxies). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
87      function updateUSDTDepositEnabled(bool enabled) external onlyRole(ADMIN_ROLE) {
88          bool oldUSDTDepositEnabled = USDTDepositEnabled;
89          require(oldUSDTDepositEnabled != enabled, "newUSDTDepositEnabled can not be
                equal oldUSDTDepositEnabled");
90
91          USDTDepositEnabled = enabled;
92          emit UpdateUSDTDepositEnabled( oldUSDTDepositEnabled, USDTDepositEnabled);
93      }
94
95      function updateCommissionRate(uint256 _commissionRate) external onlyRole(ADMIN_ROLE)
             {
96          require(_commissionRate >= 0, "commissionRate cannot less than zero");
97          require(_commissionRate <= 1e4, "commissionRate cannot greater than 10000");
98
99          uint256 oldCommissionRate = commissionRate;
100         require(oldCommissionRate != _commissionRate, "newCommissionRate can not be
                equal oldCommissionRate");
101
102         commissionRate = _commissionRate;
103         emit UpdateCommissionRate(oldCommissionRate, commissionRate);
104     }
105
106     function updateUSDFMaxSupply(uint256 _USDFMaxSupply) external onlyRole(ADMIN_ROLE) {
107         require(_USDFMaxSupply > 0, "USDFMaxSupply cannot be a zero");
108
109         uint256 oldUSDFMaxSupply = USDFMaxSupply;
110         require(oldUSDFMaxSupply != _USDFMaxSupply, "newUSDFMaxSupply can not be equal
                oldUSDFMaxSupply");
111
112         USDFMaxSupply = _USDFMaxSupply;
113         emit UpdateUSDFMaxSupply(oldUSDFMaxSupply, USDFMaxSupply);
114     }
```

```
115
116     function updateCeffuAddress(address _ceffuAddress) external onlyRole(ADMIN_ROLE) {
117         require(_ceffuAddress != address(0), "ceffuAddress cannot be a zero address");
118
119         address oldCeffuAddress = ceffuAddress;
120         require(oldCeffuAddress != _ceffuAddress, "newCeffuAddress can not be equal
                oldCeffuAddress");
121
122         ceffuAddress = _ceffuAddress;
123         emit UpdateCeffuAddress( oldCeffuAddress, ceffuAddress);
124     }
125
126     function updateTransferToCeffuEnabled(bool enabled) external onlyRole(ADMIN_ROLE) {
127         bool oldTransferToCeffuEnabled = transferToCeffuEnabled;
128         require(oldTransferToCeffuEnabled != enabled, "newTransferToCeffuEnabled can not
                be equal oldTransferToCeffuEnabled");
129
130         transferToCeffuEnabled = enabled;
131         emit UpdateTransferToCeffuEnabled(oldTransferToCeffuEnabled,
                transferToCeffuEnabled);
132     }
```

Listing 3.3: Example Privileged Functions in USDFEarn

Note that if the privileged owner account is 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 may 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 mitigated as the team makes use of a timelock to act as the privileged owner.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the Astherus USDF Earn protocol, which is a novel platform within the Astherus ecosystem, designed to aggregate a diverse array of earning strategies across multiple blockchains and protocols. It simplifies the process of asset staking with a user-friendly, one-click feature, enabling users to effortlessly engage in the most lucrative opportuni 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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