SMART CONTRACT AUDIT REPORT

for

Astherus BNB Earn

Prepared By: Xiaomi Huang

PeckShield

December 2, 2024

Document Properties

Client	Astherus
Title	Smart Contract Audit Report
Target	Astherus BNB 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 2, 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 BNB 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 BNB Earn

Astherus BNB 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 BNB Earn

Item	Description
Name	Astherus
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 2, 2024

- https://github.com/astherus-contract/ass-bnb-earn-contract.git (870d30e)

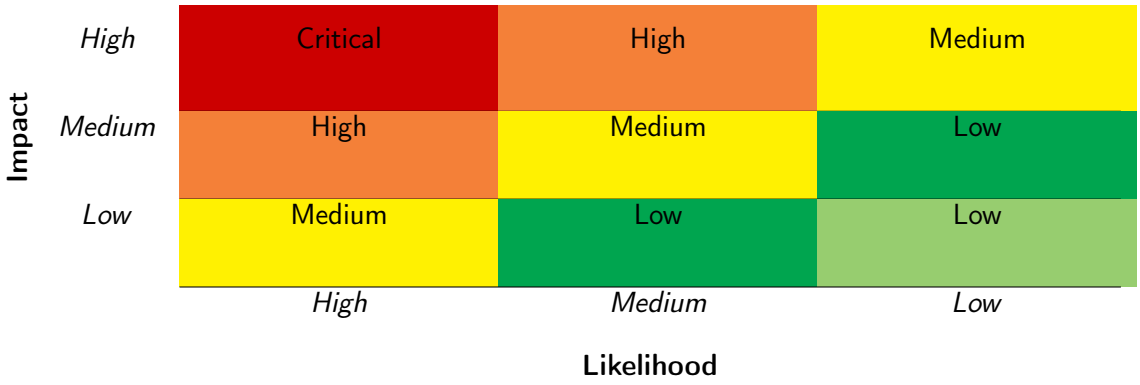
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/astherus-contract/ass-bnb-earn-contract.git (bb35f6f)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 BNB 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	3	■■■
Informational	0	
Total	3	

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 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in YieldProxy	Coding Practices	Resolved
PVE-002	Low	Accommodation of Non-ERC2-Compliant Tokens	Coding Practices	Resolved
PVE-003	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 Improved Constructor/Initialization Logic in YieldProxy

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `YieldProxy`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

To facilitate possible future upgrade, the `YieldProxy` constract is instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers();`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initializer()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
85   function initialize(
86     address _admin,
87     address _manager,
88     address _pauser,
89     address _bot,
90     address _token,
91     address _asBnb,
92     address _stakeManager,
93     address _mpcWallet
94   ) external initializer {
95     require(_admin != address(0), "Invalid admin address");
96     require(_manager != address(0), "Invalid manager address");
97     require(_pauser != address(0), "Invalid pauser address");
```

```
98      require(_bot != address(0), "Invalid bot address");
99      require(_token != address(0), "Invalid token address");
100     require(_asBnb != address(0), "Invalid asBnb address");
101     require(_stakeManager != address(0), "Invalid stakeManager address");
102     require(_mpcWallet != address(0), "Invalid MPC wallet address");

104     __Pausable_init();
105     __ReentrancyGuard_init();

107     _grantRole(DEFAULT_ADMIN_ROLE, _admin);
108     _grantRole(MANAGER, _manager);
109     _grantRole(PAUSER, _pauser);
110     _grantRole(BOT, _bot);

112     token = IERC20(_token);
113     asBnb = IAsBNB(_asBnb);
114     stakeManager = _stakeManager;
115     slisBNBProvider = address(0);
116     mpcWallet = _mpcWallet;
117 }
```

Listing 3.1: `YieldProxy::initialize()`

Moreover, the above `initialize()` routine can also be improved by initializing its parent contracts, including `AccessControlUpgradeable` and `UUPSUpgradeable`.

Recommendation Improve the above-mentioned constructor/initialization routines in `YieldProxy`. Note the initialization routine in `AsBnbMinter` can be similarly improved.

Status This issue has been fixed by the following commit: `05a00c7`.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AsBnbMinter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

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 `transfer()` 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. Specifically, the `transfer()` routine does not have a return value defined and

implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call will be unfortunately reverted. Another transferFrom() routine shares the same design.

```
126     function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
127         uint fee = (_value.mul(basisPointsRate)).div(10000);
128         if (fee > maximumFee) {
129             fee = maximumFee;
130         }
131         uint sendAmount = _value.sub(fee);
132         balances[msg.sender] = balances[msg.sender].sub(_value);
133         balances[_to] = balances[_to].add(sendAmount);
134         if (fee > 0) {
135             balances[owner] = balances[owner].add(fee);
136             Transfer(msg.sender, owner, fee);
137         }
138         Transfer(msg.sender, _to, sendAmount);
139     }
```

Listing 3.2: USDT::transfer()

Because of that, a normal call to transfer()/transferFrom() is suggested to use the safe version, i.e., safeTransfer()/safeTransferFrom(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the AsBnbMinter::_mint() routine that is designed to transfer the funds from the calling user. To accommodate the specific idiosyncrasy, there is a need to user safeTransferFrom(), instead of transferFrom() (line 313).

```
310   function _mint(uint256 amountIn) private returns (uint256) {
311     require(amountIn >= minMintAmount, "amount is less than minMintAmount");
312     // transfer token to YieldProxy
313     token.transferFrom(msg.sender, address(this), amountIn);
314     token.safeIncreaseAllowance(yieldProxy, amountIn);
315     IYieldProxy(yieldProxy).deposit(amountIn);

317     // queue mint request if there is on going activity
318     if (IYieldProxy(yieldProxy).activitiesOnGoing()) {
319       // set queueFront tokenMintReq
320       tokenMintReqQueue[queueRear] = TokenMintReq(msg.sender, amountIn);
321       // increment queueRear
322       ++queueRear;
323       // emit event
324       emit TokenMintReqQueued(msg.sender, amountIn);
325       return 0;
326     } else {
327       // calculate amount to mint
```

```
328         uint256 amountToMint = convertToAsBnb(amountIn);
329         // record token amount increment
330         totalTokens += amountIn;
331         // mint asBnb
332         asBnb.mint(msg.sender, amountToMint);
333         // emit event
334         emit AsBnbMinted(msg.sender, amountIn, amountToMint);
335         return amountToMint;
336       }
337     }
```

Listing 3.3: `AsBnbMinter::_mint()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed by the following commit: `bb35f6f`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Astherus BNB 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.

```
238   function unpause() external onlyRole(MANAGER) {
239     _unpause();
240   }
241   ...
242   function pause() external onlyRole(PAUSER) {
243     _pause();
244   }
245   ...
246   function addActivity(
247     uint256 _startTime,
248     uint256 _endTime,
249     string memory _tokenName
```

```
250   ) external onlyRole(MANAGER) whenNotPaused {
251     require(_startTime > block.timestamp, "Invalid start time");
252     require(_startTime < _endTime, "Invalid time range");
253     require(bytes(_tokenName).length > 0, "Invalid token name");
254     activities.push(Activity(_startTime, _endTime, 0, _tokenName));
255     emit ActivityAdded(_startTime, _endTime, 0, _tokenName);
256   }
257   ...
258   function endActivity(uint256 numberOfActivity) external onlyRole(MANAGER)
          whenNotPaused {
259     require(numberOfActivity > 0, "Invalid number of activities");
260     require(this.activitiesOnGoing(), "No active activity");
261     uint256 idx = lastUpdatedActivityIdx;
262     // starting from lastUpdatedActivityIdx
263     for (uint256 i = idx; i < (lastUpdatedActivityIdx + numberOfActivity); i++) {
264       // break if reach to the end
265       if (i >= activities.length) {
266         break;
267       }
268       activities[i].rewardedTime = block.timestamp;
269       idx = i;
270       // emit event, note that no reward is compounded
271       emit ActivitySettled(block.timestamp, 0, activities[idx].tokenName);
272     }
273     lastUpdatedActivityIdx = idx;
274   }
275   ...
276   function reDelegateTokens() external onlyRole(MANAGER) whenNotPaused {
277     require(slisBNBProvider != address(0), "slisBNBProvider not set");
278     require(mpcWallet != address(0), "mpcWallet not set");
279     ISlisBNBProvider(slisBNBProvider).delegateAllTo(mpcWallet);
280   }
281   ...
282   function convertAllSlisBNBToClisBNB() external onlyRole(MANAGER) whenNotPaused {
283     require(slisBNBProvider != address(0), "slisBNBProvider not set");
284     require(mpcWallet != address(0), "mpcWallet not set");
285     uint256 balance = token.balanceOf(address(this));
286     ISlisBNBProvider(slisBNBProvider).provide(balance, mpcWallet);
287   }
288   ...
289   function setSlisBNBProvider(address _slisBNBProvider) external onlyRole(MANAGER) {
290     require(_slisBNBProvider != address(0) && _slisBNBProvider != slisBNBProvider, "
          Invalid slisBNBProvider address");
291     address oldSlisBNBProvider = slisBNBProvider;
292     slisBNBProvider = _slisBNBProvider;
293     emit SlisBNBProviderSet(oldSlisBNBProvider, _slisBNBProvider);
294   }
295   ...
296   function setMPCWallet(address _mpcWallet) external onlyRole(MANAGER) {
297     require(_mpcWallet != address(0) && _mpcWallet != mpcWallet, "Invalid MPC wallet
          address");
298     address oldMpcWallet = mpcWallet;
```

```
299     mpcWallet = _mpcWallet;
300     emit MPCWalletSet(oldMpcWallet, _mpcWallet);
301   }
302   ...
303   function setRewardsSender(address _rewardsSender) external onlyRole(MANAGER) {
304     require(_rewardsSender != address(0) && _rewardsSender != rewardsSender, "Invalid
          rewards sender address");
305     address oldRewardsSender = rewardsSender;
306     rewardsSender = _rewardsSender;
307     emit RewardsSenderSet(oldRewardsSender, _rewardsSender);
308   }
```

Listing 3.4: Example Privileged Functions in `YieldProxy`

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 BNB 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

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

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

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