SMART CONTRACT AUDIT REPORT

for

RemixDao (Ewe Technology)

Prepared By: Xiaomi Huang

PeckShield

Aug 4, 2023

Document Properties

Client	Ewe Protocol
Title	Smart Contract Audit Report
Target	Ewe
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Aug 4, 2023	Jing Wang	Final Release
1.0-rc	July 28, 2023	Jing Wang	Release Candidate

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 `Ewe` design document and related smart contract source code, 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 branch of `Ewe` 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 Ewe

Ewe is a farming protocol which provides interfaces for users to deposit tokens and earn rewards from `Uniswap V3`. Each strategy manages only one pair token and pool fee (correspond to `Uniswap V3 Pool` `contract`). The protocol provides automatically earning and rescaling mechanisms to collect rewards and rebalance the positions of user funds. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Ewe

Item	Description
Name	Ewe Protocol
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Aug 4, 2023

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

- https://github.com/ewe-technology/pancakeswap-V3-strategy-contract (85bbca3)

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

- https://github.com/ewe-technology/pancakeswap-V3-strategy-contract (51ca86d)

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

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 [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 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

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

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) [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.

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 Ewe protocol 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	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 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 low-severity vulnerabilities.

Table 2.1: Key Ewe Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Sandwich-Based MEV With Imbalanced Positions	Time and State	Fixed
PVE-002	Low	Possible Costly LPs From Improper Strategy Initialization	Time and State	Fixed
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Confirmed

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 Potential Sandwich-Based MEV With Imbalanced Positions

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

As mentioned earlier, the protocol supports liquidity by adding single-side tokens and requires the timely invocation to rebalance current positions. Because of above requirement, there is a constant need of swapping one asset to another. With that, the protocol has provided several interfaces to facilitate the asset conversion.

```
function swapToken(
  ...
  ) public payable override returns (uint256 outputAmount) {
  ...

      // get minimum swap out amount
      uint256 minimumSwapOutAmount = getMinimumSwapOutAmount(
    ...
      );
      require(minimumSwapOutAmount > 0, "inputAmount too small");

      uint256 pathLength = swapPathArray.length;
      if (pathLength == 2) {
          // statement for "single swap path", swap by exactInputSingle function
          outputAmount = ISmartRouter(SMART_ROUTER_ADDRESS).exactInputSingle(
              ISmartRouter.ExactInputSingleParams(
        ...
                  minimumSwapOutAmount,
                  0
              )
          );
```

```
    }
    ...
    }

    function getMinimumSwapOutAmount(
    ...
    ) public view override returns (uint256 minimumSwapOutAmount) {
        uint256 estimateSwapOutAmount = getEstimateSwapOutAmount(
      ...
        );
    ...
    }

  function getEstimateSwapOutAmount(
    ...
    ) public view returns (uint256 estimateSwapOutAmount) {
    ...
            (
                address token0,
                address token1,
                uint256 tokenPriceWith18Decimals // (token1/token0) * 10**DECIMALS_PRECISION
            ) = getTokenExchangeRate(tokenIn, tokenOut);

    ..
    }

    function getTokenExchangeRate(
    ...
    {
    ...
        // calculate token price with 18 decimal precision
        tokenPriceWith18Decimals = PoolHelper.getTokenPriceWithDecimalsByPool(
            poolAddress,
            ZapConstants.DECIMALS_PRECISION
        );
    ...
    }
```

Listing 3.1: Zap::swapToken()

To elaborate, we show above the `swapToken()` helper routine. We notice the conversion is routed to `UniswapV3` in order to swap one asset to another. And the swap operation specifies some restrictions on possible slippage, however, it is based on one spot price, and is therefore vulnerable to be manipulated and possible front-running attacks, resulting in a smaller gain for this round of conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the TWAP or time-weighted average price of UniswapV3. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective slippage control mechanism (e.g., TWAP) against above sandwich attacks to better protect the interests of protocol users.

Note this is a protocol wise issue and all routines which are related to token swaps share the same issue.

Status This issue has been fixed in the following commit: 51ca86d.

3.2 Possible Costly LPs From Improper Strategy Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Strategy
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

As mentioned before, the Ewe protocol provides a platform for users to deposit tokens to a Strategy and the controller can manage the funds. The depositor will get their pro-rata share based on their deposited amount. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the share extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the depositLiquidity() routine. This depositLiquidity() routine is used for participating users to deposit the supported asset (e.g., BNB) and get respective profits in return. The issue occurs when the Strategy is being initialized under the assumption that the current Strategy is empty.

```
    function depositLiquidity
    {
        ...
        // update userShare & totalUserShare
        uint256 increasedShare = calculateIncreasedShareAndUpdateUserShare(
            userAddress,
            increasedLiquidity
        );
        \dots
    }

```

```
150     function calculateIncreasedShareAndUpdateUserShare(
151         address userAddress,
152         uint128 increasedLiquidity
153     ) internal returns (uint256 increasedShare) {
154         // update userShare & totalUserShare
155         uint128 totalLiquidity = getNftLiquidityAmount();
156 
157         if (totalUserShare == 0) {
158             increasedShare = totalLiquidity;
159         } else {
160             increasedShare = uint256(increasedLiquidity)
161                 .mul(totalUserShare)
162                 .div(uint256(totalLiquidity).sub(increasedLiquidity));
163         }
164         require(increasedShare > 0, "deposit amount too small");
165 
166         userShare[userAddress] = userShare[userAddress].add(increasedShare);
167         totalUserShare = totalUserShare.add(increasedShare);
168     }
169 }
```

Listing 3.2: `Strategy::depositLiquidity()`

Specifically, when the `Strategy` is being initialized, the share value directly takes the value of `increasedShare = totalLiquidity` (line 158), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `getNftLiquidityAmount()= 1 WEI`. With that, the actor can further deposit a huge amount of asset into the position with the goal of making the share extremely expensive.

An extremely expensive 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 vault tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the vault without returning any valut tokens.

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 execution logic of share calculation to defensively calculate the share amount when the vault is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been fixed in the following commit: `51ca86d`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Ewe` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., configure system parameters). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Zap` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `Ewe` protocol allow the `owner` to set swap path.

```
    function setSwapPath(
        address inputToken,
        address outputToken,
        address[] memory newSwapPath
    ) public onlyOwner {
        // parameter verification
    ...
        for (uint i = 0; i < pathLength; i++) {
            ParameterVerificationHelper.verifyNotZeroAddress(newSwapPath[i]);
        }

        // verify inputToken is not outputToken
        require(inputToken != outputToken, "inputToken == outputToken");

        // verify input path is valid swap path
        require(pathLength >= 2, "path too short");

        // verify first token in newSwapPath is inputToken
        require(newSwapPath[0] == inputToken, "path not start from inputToken");

        // verify last token in newSwapPath is outputToken
        require(
            newSwapPath[(pathLength - 1)] == outputToken,
            "path not end with outputToken"
        );
    ...
    }
```

Listing 3.3: Example Privileged Operations in the `Zap` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that 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.

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 confirmed.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Ewe` protocol, which provides interfaces for user to deposit tokens and earn rewards from trading and staking from `Uniswap V3`. 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.

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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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