SMART CONTRACT AUDIT REPORT

for

ElephantReserve And Stampede

Prepared By: Xiaomi Huang

PeckShield

August 28, 2022

Document Properties

Client	Elephant Money
Title	Smart Contract Audit Report
Target	ElephantReserve And Stampede
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	August 28, 2022	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 two smart contracts of the `Elephant Money` protocol, i.e., `ElephantReserve` and `Stampede`, 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 Elephant Money

The `Elephant Money` protocol aims to be the global decentralized community bank of its kind. By design, it is a permissionless system for economic inclusion and helps its community accumulate wealth through active and passive cash flows. This audit only covers to specific smart contracts, i.e., `ElephantReserve` and `Stampede`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of ElephantReserve And Stampede

Item	Description
Name	Elephant Money
Website	https://elephant.money/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 28, 2022

In the following, we show the given two files with the source contract for audit and the MD5/SHA checksum values of the given files:

- File-1/2: Stampede.sol/ElephantReserve-v5.sol
- MD5-1/2: d7e7d9bc1f52c1d8170d4aa9f9ecdc6a/b107a6ab17bd1d44ede47fb421fc209e

- SHA256-1: d8fb29c0d3e4d3ac8ce25a7639780e3ddb20dadd7db4556e103c3179eed1eb57
- SHA256-2: f92fbfc4eca7f05c060904f1fdce117e2d1e206f8c77c9a64905fa6cba5b9453

1.2 About PeckShield

PeckShield Inc. [11] 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 the OWASP Risk Rating Methodology [10]:

- 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 checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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) [9], 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 Logic	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 implementation of the two specific contracts of the `Elephant Money` protocol, i.e., `ElephantReserve` and `Stampede`. 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	2	■ ■
Low	2	■ ■
Informational	0	
Total	4	

We have so far identified a potential issue for improvement: it involves an unused import of the `Ownable` smart contract, which can be safely removed without affecting the normal functionality. More information can be found in the next subsection, and its detailed discussions can be found 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 issue (shown in Table 2.1), including 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key ElephantReserve And Stampede Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-002	Medium	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	
PVE-004	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	

Besides 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 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

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.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.1: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `Stampede::sponsor()` routine, it is designed to sponsor the given user with the specified amount. To accommodate the specific idiosyncrasy, there is a need to user `safeTransferFrom()`, instead of `transferFrom()` (line 597).

```
582  function sponsor(address _addr, uint256 _amount) external {


585    address _sender = msg.sender;

587    User memory sUser = getUser(_sender);

589    //Checks
590    require(_addr != address(0), "Can't send to the zero address");
591    require(_addr != _sender, "Can't send to yourself");
592    require(sUser.deposits > 0, "Sender must be active");
593    require(_amount >= minimumAmount, "Minimum deposit");

595    //Transfer TRUNK to the contract FROM SENDER //This is a sponsorship
596    require(
597      backedToken.transferFrom(
598        _sender,
599        address(backedTreasury),
600        _amount
601      ),
602      "TRUNK token transfer failed"
603    );

605    //We operate side effect free and just add to pending sponsorships

607    sponsorData.add(_addr, _amount);

609    emit NewSponsorship(_sender, _addr, _amount);

611    flowData.total_txs_incr();

613  }
```

Listing 3.2: `Stampede::sponsor()`

In the meantime, we also suggest to use the safe-version of `transfer()/transferFrom()` in other related routines, including `Stampede::_claim_out()` and `ElephantReserve::redeem()`.

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

Status The issue has been resolved as the team confirms the use of only ERC20-compliant tokens.

3.2 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: ElephantReserve
- Category: Time and State [7]
- CWE subcategory: CWE-682 [4]

Description

The ElephantReserve contract has a user-facing routine, i.e., redeem(), which can be used to redeem backed tokens for collateral. It has a rather straightforward logic in computing the intended redeemed collateral amount after conversion and then performing the actual swap via the collateralRouter (line 1098).

```
1073    function redeem(uint256 backedAmount) public returns (uint collateralAmount,  uint
            feeAmount) {

1075        address msgSender = _msgSender();

1077        require(mintData.ready(msgSender), "Mutable reserve calls can not be made
                multiple times in a block window");

1079        require(backedAmount >= 1e18, "Backed amount must be greater than 1 unit");

1081        //the system will naturally balance itself based on redemptions and payout the
                core asset based on the
1082        require(backedToken.transferFrom(msgSender,address(this), backedAmount), "Backed
                 token must be approved and available");

1084        //If we are trying to avoid burning we can use the Pancake LP to avoid redeeming
                 TRUNK within slippage tolerance
1085        (collateralAmount, feeAmount) = estimateRedemption(backedAmount);

1087        //If the estimate doesn't include core we just swap

1089        uint initialBalance = collateralToken.balanceOf(msgSender);

1091        //Convert from backed to collateral using the core's Oracle
1092        address[] memory path = new address[](2);
1093        path[0] = address(backedToken);
1094        path[1] = address(collateralToken);

1096        require(backedToken.approve(address(collateralRouter), collateralAmount));

1098        collateralRouter.swapExactTokensForTokens(
```

```
            collateralAmount, //swap the backed amount - fees
            0, //accept any amount of core tokens
            path,
            msgSender, //send to msgSender
            block.timestamp
        );

        collateralAmount = collateralToken.balanceOf(msgSender).sub(initialBalance);

        //transfer fee or remaining balance to the TRUNK Treasury
        backedToken.transfer(address(backedTreasury), feeAmount.min(backedToken.
            balanceOf(address(this))));

        //touch so redeem can't be looped in a smart contract / flashloan
        mintData.touch(msgSender);

        //Fire event

        emit onRedemption(
            msgSender,
            backedAmount,
            collateralAmount,
            feeAmount,
            block.timestamp
        );


    }
```

Listing 3.3: `ElephantReserve::redeem()`

To elaborate, we show above the `redeem()` routine. We notice the token swap is routed to `collateralRouter` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any effective restriction [1] on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

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 `UniswapV2`. 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 mitigation to the above front-running attack to better

[1] The current approach of computing the expected return amount via `collateralRouter.getAmountsOut(backedAmount.sub(feeAmount), path)` does not apply any slippage control at all.

protect the interests of protocol users.

Status

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the two audited contracts, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., set the various parameters, as well as related percentage, etc). 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.

To elaborate, we show below example privileged routines from `ElephantReserve`. These routines allow the `owner` account to set new `collateralRouter` contract address, set the `liquidityThreshold/liquidityFrequency/daily_apr`, etc.

```
    //Core collateral liquidity can move from one contract location to another across
        major PCS releases
    function updateCollateralRouter(address _router) onlyOwner public {
        require(_router != address(0), "Router must be set");
        collateralRouter = IUniswapV2Router02(_router);

        emit UpdateCollateralRouter(_router);
    }

    //Mint data is kept across reserves so updates can happen at any time
    function updateMintData(address mintDataAddress) onlyOwner external {
        require(mintDataAddress != address(0), "Require valid non-zero addresses");

        mintData = MintData(mintDataAddress);

        emit UpdateMintData(mintDataAddress);
    }
```

Listing 3.4: `ElephantReserve::updateCollateralRouter()/updateMintData()`

It would be 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

3.4 Improved Precision By Multiplication And Division Reordering

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `ElephantReserve`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [1]

Description

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

In particular, we use the `Stampede::payoutOf()` as an example. This routine is used to calculate the current payout and max-payout of a given address.

```
function payoutOf(address _addr) public view returns(uint256 payout, uint256
    max_payout) {

    User memory _user = getUser(_addr);

    //The max_payout is a function of deposits
    max_payout = maxPayoutOf(_user.deposits);

    uint256 share;

    // No need for negative fee

    if(_user.payouts < max_payout) {
```

```
831        //Using 1e18 we capture all significant digits when calculating available divs
832        share = _user.deposits.mul(payoutRate * 1e18).div(100e18).div(24 hours); //divide
               the profit by payout rate and seconds in the day
833        payout = share * block.timestamp.safeSub(_user.deposit_time);

835        // payout remaining allowable divs if exceeds
836        if(_user.payouts + payout > max_payout) {
837          payout = max_payout.safeSub(_user.payouts);
838        }

840      }
841    }
```

Listing 3.5: Stampede::payoutOf()

We notice the calculation of the resulting payout (line 833) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `payout = user.deposits.mul(payoutRate).mul(elapsed_time).div(24 hours).div(100)`, where `uint256 elapsed_time = block.timestamp.safeSub(_user.deposit_time)`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

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

Status

4 | Conclusion

In this audit, we have analyzed the design and implementation of two specific contracts of the Elephant Money protocol, i.e., ElephantReserve and Stampede. The protocol itself aims to be the global decentralized community bank of its kind. By design, it is a permissionless system for economic inclusion and helps its community accumulate wealth through active and passive cash flows. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

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

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

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

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

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