SMART CONTRACT AUDIT REPORT

for

StoneDefi Protocol

Prepared By: Yiqun Chen

PeckShield

September 18, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 18, 2021	Xuxian Jiang	Final Release
1.0-rc	September 17, 2021	Xuxian Jiang	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `StoneDefi` 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 StoneDefi

The `StoneDefi` protocol acts as the central part of the incentive structure of the `StoneDefi` ecosystem. The audited protocol is a modular one that is heavily influenced by `YFI` with `Vaults` acting as token containers to keep track of an ever-growing pool with additional gains returned back to users. The gains are harvested by employing various `strategies` that are designed to automate the best yield farming opportunities available. The `StoneDefi` protocol is developed with its own `strategies`.

The basic information of the `StoneDefi` protocol is as follows:

Table 1.1: Basic Information of The `StoneDefi` Protocol

Item	Description
Name	StoneDefi
Website	https://www.stonedefi.io/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	September 18, 2021

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

- https://github.com/stonedefi/stone_defi_contract_v2.git (1c05b98)

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

- https://github.com/stonedefi/stone_defi_contract_v2.git (054bcfa)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [11], 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 `StoneDefi` 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 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	1	■
Low	5	■ ■ ■ ■ ■
Informational	2	■ ■
Undetermined	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 5 low-severity vulnerabilities, 2 informational suggestions, and 1 undetermined issue.

Table 2.1: Key StoneDefi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-002	Informational	Meaningful Events For Important States Change	Coding Practices	Fixed
PVE-003	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-004	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-005	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-006	Informational	Removal of Unused Code	Coding Practices	Fixed
PVE-007	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-008	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-009	Low	Proper Estimate Logic in _estimateDebtLimitDecrease()	Business Logic	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in 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: `WETHGateway`
- Category: Coding Practices [8]
- CWE subcategory: CWE-628 [4]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
```

```
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.1: ZRX.sol

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. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `emergencyTokenTransfer()` routine in the `WETHGateway` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).transfer(to, amount)` (line 128) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
116  /**
117   * @dev transfer ERC20 from the utility contract, for ERC20 recovery in case of stuck tokens due
118   * direct transfers to the contract address.
119   * @param token token to transfer
120   * @param to recipient of the transfer
121   * @param amount amount to send
122   */
123  function emergencyTokenTransfer(
124    address token,
125    address to,
126    uint256 amount
127  ) external onlyOwner {
128    IERC20(token).transfer(to, amount);
129  }
```

Listing 3.2: `WETHGateway::emergencyTokenTransfer()`

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

Status This issue has been fixed in the following commit: `bb0be57`.

3.2 Generation of Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SharerV4, CommonHealthCheck
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the SharerV4 contract as an example. This contract has public functions that are used to transfer the governance. While examining the events that reflect the governance changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the governance is being updated in SharerV4::acceptGovernance(), there is no respective event being emitted to reflect the update of governance (line 497).

```
490     function setGovernance(address _governance) external {
491         require(msg.sender == governance);
492         pendingGovernance = _governance;
493     }

495     function acceptGovernance() external {
496         require(msg.sender == pendingGovernance);
497         governance = pendingGovernance;
498     }
```

Listing 3.3: SharerV4::setGovernance()/acceptGovernance()

Recommendation Properly emit respective events when a new governance becomes effective.

Status This issue has been fixed in the following commit: bb0be57.

3.3 Duplicate Pool Detection and Prevention

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Staking
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The StoneDefi protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) external onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accPerShare: 0
        }));
    }
```

Listing 3.4: Staking::add()

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
         do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) external
        onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accPerShare: 0
        }));
    }
```

Listing 3.5: Revised `Staking::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been fixed in the following commit: 959156d.

3.4 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the `StoneDefi` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's TOKEN allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) external onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.6: Staking::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern. Note another routine `setTokenPerBlock()` shares the same issue.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's TOKEN allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) external onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.7: Revised Staking::set()

Status This issue has been fixed in the following commit: af7bc05.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

The StoneDefi protocol has a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., vault/strategy addition, reward adjustment, and parameter setting). In the following, we show representative privileged operations in the protocol's core Staking contract.

```
    // Update the given pool's TOKEN allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) external onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    function setTokenPerBlock(uint256 _tokenPerBlock) external onlyOwner {
        tokenPerBlock = _tokenPerBlock;
    }

    // Sweep all STN to owner
    function sweep(uint256 _amount) external onlyOwner {
        token.safeTransfer(owner(), _amount);
    }
```

Listing 3.8: Staking::set()/setTokenPerBlock()/sweep()

In addition, we also notice the privileged function updateSlot() that can be used to update any storage slot, which in essence allows the owner to update any contract state.

```
    /**
     * Allow storage slots to be manually updated
     */
    function updateSlot(bytes32 slot, bytes32 value) external {
        require(msg.sender == ownerAddress, "Ownable: Admin only");
        assembly {
            sstore(slot, value)
        }
```

```
    }
```

Listing 3.9: `Curve::updateSlot()`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Meanwhile, we point out that a compromised `owner` account would allow the attacker to add a malicious strategy or change other settings, which directly undermines the assumption of the `StoneDefi` protocol. Also notice that if the main protocol logic is deployed behind a proxy, which can be upgradeable via the authorized admin account, the trust of this admin account is also paramount to the entire protocol.

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 by removing the privileged `updateSlot()` function from affected contracts. Also, the team confirms that the owner privilege will be properly managed with multisig.

3.6 Removal of Unused Code

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GenericLogic`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

StoneDefi makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Ownable`, to facilitate its code implementation and organization. For example, the smart contract `StrategyLenderYieldOptimiser` has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `StoneVaults` contract, there is a function `getPriceStoneVault()` that is designed to retrieve the price from the given `stone vault`. However, it contains unreachable code (line 70) with `revert()` after the `return` statement.

```
54    function getPriceStoneVault(address tokenAddress)
55        public
56        view
```

```
        returns (uint256)
    {
        // v2 vaults use pricePerShare scaled to underlying token decimals
        IVault vault = IVault(tokenAddress);
        if (isStoneVault(tokenAddress) == false) {
            revert("CalculationsStoneVaults: Token is not a stone vault");
        }
        address underlyingTokenAddress = vault.token();
        uint256 underlyingTokenPrice =
            oracle.getPriceUsdcRecommended(underlyingTokenAddress);
        uint256 sharePrice = vault.pricePerShare();
        uint256 tokenDecimals = IERC20Metadata(underlyingTokenAddress).decimals();
        return (underlyingTokenPrice * sharePrice) / 10**tokenDecimals;
        revert();
    }
```

Listing 3.10: StoneVaults::getPriceStoneVault()

Recommendation Consider the removal of the unreachable code in the above `getPriceStoneVault()` function.

Status This issue has been fixed in the following commit: bb0be57.

3.7 Staking Incompatibility With Deflationary Tokens

- ID: PVE-007
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Staking`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In the `StoneDefi` protocol, the `Staking` contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., `deposit()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., `deposit()` and `withdraw()`, the contract using the `safeTransfer()/safeTransferFrom()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    // Deposit LP tokens.
    //  In normal case, _recipient should be same as msg.sender
    //  In deposit and stake, _recipient should be address of initial user
```

```
    function deposit(uint256 _pid, uint256 _amount, address _recipient) public returns (
        uint256) {

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_recipient];
        uint256 rewards = _harvest(_recipient, _pid);
        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accPerShare).div(1e12);
        emit Deposit(_recipient, _pid, _amount);
        return rewards;
    }
```

Listing 3.11: `Staking::deposit())`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accPerShare` via dividing `tokenReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 176). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accPerShare` as the final result, which dramatically inflates the pool's reward.

```
170     // Update reward variables of the given pool to be up-to-date.
171     function updatePool(uint256 _pid) internal {
172         PoolInfo storage pool = poolInfo[_pid];
173         if (block.number <= pool.lastRewardBlock) {
174             return;
175         }
176         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
177         if (lpSupply == 0) {
178             pool.lastRewardBlock = block.number;
179             return;
180         }
181         uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
182         uint256 tokenReward = multiplier.mul(tokenPerBlock).mul(pool.allocPoint).div(
                totalAllocPoint);
183         pool.accPerShare = pool.accPerShare.add(tokenReward.mul(1e12).div(lpSupply));
184         pool.lastRewardBlock = block.number;
185     }
```

Listing 3.12: `Staking::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Stone` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

3.8 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `Staking` as an example, the `withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 244) starts before effecting the update on the internal state (line 246), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
235     // Withdraw LP tokens.
236     function withdraw(uint256 _pid, uint256 _amount) external returns (uint256) {
237         address sender = msg.sender;
238         PoolInfo storage pool = poolInfo[_pid];
239         UserInfo storage user = userInfo[_pid][sender];
240         require(user.amount >= _amount, "withdraw: not good");
241         uint256 rewards = _harvest(sender, _pid);
242         if(_amount > 0) {
243             user.amount = user.amount.sub(_amount);
244             pool.lpToken.safeTransfer(address(sender), _amount);
245         }
246         user.rewardDebt = user.amount.mul(pool.accPerShare).div(1e12);
247         emit Withdraw(sender, _pid, _amount);
248         return rewards;
249     }
```

Listing 3.13: `Staking::withdraw()`

Note that other routines share the same issue, including `deposit()`, `withdraw()`, and `unstakeAndWithdraw()`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in the following commit: d332af6.

3.9 Proper Estimate Logic in _estimateDebtLimitDecrease()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StrategyLenderYieldOptimiser`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

In order for the governance module to better decide proper debt limits, the `StoneDefi` protocol provides a helper routine to estimate future APR with a change of debt limit. The helper routine `estimatedFutureAPR()` accepts a new debt limit, which is then compared with current debt limit and gauge the resulting new APR. Based on the increase or decrease of the debt limit, this routine further delegates the calculation task to two internal routines, i.e., `_estimateDebtLimitIncrease()` and `_estimateDebtLimitDecrease()` respectively.

```
244     //Estimates debt limit decrease. It is not accurate and should only be used for very broad decision making
245     function _estimateDebtLimitDecrease(uint256 change) internal view returns (uint256) {
246         uint256 lowestApr = uint256(-1);
247         uint256 aprChoice = 0;
248 
249         for (uint256 i = 0; i < lenders.length; i++) {
250             uint256 apr = lenders[i].aprAfterDeposit(change);
251             if (apr < lowestApr) {
252                 aprChoice = i;
253                 lowestApr = apr;
254             }
255         }
256 
257         uint256 weightedAPR = 0;
258 
259         for (uint256 i = 0; i < lenders.length; i++) {
260             if (i != aprChoice) {
261                 weightedAPR = weightedAPR.add(lenders[i].weightedApr());
262             } else {
263                 uint256 asset = lenders[i].nav();
264                 if (asset < change) {
265                     //simplistic. not accurate
266                     change = asset;
267                 }
268                 weightedAPR = weightedAPR.add(lowestApr.mul(change));
269             }
270         }
271         uint256 bal = estimatedTotalAssets().add(change);
272         return weightedAPR.div(bal);
273     }
```

Listing 3.14: `StrategyLenderYieldOptimiser::_estimateDebtLimitDecrease()`

To elaborate, we show above the `_estimateDebtLimitDecrease()` routine that is used to measure the new APR when the debt limit is decreased. It comes to our attention that this internal routine does not properly adjust the `weightedAPR` for the lender with the `lowestApr` (line 268). The calculation formula needs to be revised as `weightedAPR = weightedAPR.add(lowestApr.mul(asset - change))`.

Recommendation Revised the afore-mentioned `_estimateDebtLimitDecrease()` function to properly estimate the future APR when the debt limit is decreased.

Status This issue has been fixed in the following commit: 1574017.

4 | Conclusion

In this audit, we thoroughly analyzed the design and implementation of the `StoneDefi` protocol. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. Developed on top of `YFI`, the `StoneDefi` protocol has been equipped with additional home-made `strategies` that work with different yield-generating pools. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

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-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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[5] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.