



SMART CONTRACT AUDIT REPORT

for

STONE PROTOCOL



Prepared By: Shuxiao Wang

PeckShield
February 25, 2021

Document Properties

Client	Stone
Title	Smart Contract Audit Report
Target	Stone
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 25, 2021	Xuxian Jiang	Final Release
1.0-rc	February 25, 2021	Xuxian Jiang	Release Candidate #1
0.3	February 20, 2021	Xuxian Jiang	Additional Findings #2
0.2	February 17, 2021	Xuxian Jiang	Additional Findings #1
0.1	February 8, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Stone	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Reentrancy Risk in yVault	12
3.2	Incompatibility with Deflationary/Rebasing Tokens	14
3.3	Possible Costly yVault LPs From Improper Liquidity Initialization	15
3.4	Suggested Governance-Restricted earn()	17
3.5	Potential Front-Running/MEV With Reduced Buyback	17
3.6	Improved Sanity Checks For System Parameters	19
3.7	Improved Asset Consistency Among Vaults, Controller, and Strategies	20
3.8	Revisited Assumption on Trusted Governance	21
4	Conclusion	23
	References	24

1 | Introduction

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

The `stone` protocol acts as the central part of the incentive structure of `stone` DeFi ecosystem. The audited protocol is a modular one that is heavily influenced by `yfi`. In particular, it has three main components, i.e., `vaults`, `strategies`, and `controller`. `Vaults` are in essence token containers and are used 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 `controller` is the central brain behind the entire protocol operation and maintenance. The `stone` protocol is developed with its own `strategies` working with specific pools, including `DForce`.

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

Table 1.1: Basic Information of The `stone` Protocol

Item	Description
Issuer	Stone
Website	https://www.stonedefi.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 25, 2021

In the following, we show the contract file and the MD5/SHA checksum value of the contract file:

- File: [stone-contract-reduced.zip](#)
- MD5: [111adb4866811a2029434c41c247a320](#)
- SHA: [1b2243d4bb60309f6908f1357bbd0930a940e090](#)

And this is the contract file and its MD5/SHA checksum value after all fixes for the issues found in the audit have been checked in:

- File: [contracts 2.zip](#)
- MD5: [95e588449f79f9a15c5ff92b1cf82100](#)
- SHA: [fcb4d854a1e4fd7f11f32d29c59fbe69f562a029](#)

1.2 About PeckShield

PeckShield Inc. [17] 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	High	Medium	Low
	High	Medium	Low
	High	Medium	Low
Likelihood	High	Medium	Low
	High	Medium	Low
	High	Medium	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [16]:

- 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 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) [15], 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

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

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.




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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Stone 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	3	
Low	3	
Informational	2	
Total	8	

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 medium-severity vulnerabilities, and 3 low-severity vulnerabilities, and and 2 informational recommendations.

Table 2.1: Key Stone Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Reentrancy Risk in yVault	Time and State	Fixed
PVE-002	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Medium	Possible Costly yVault LPs From Improper Liquidity Initialization	Time and State	Confirmed
PVE-004	Medium	Suggested Governance-Restricted earn()	Security Features	Fixed
PVE-005	Low	Potential Front-running/MEV With Reduced Buyback	Time and State	Confirmed
PVE-006	Low	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-007	Low	Improved Asset Consistency Among Vaults, Controller, and Strategies	Coding Practices	Fixed
PVE-008	Medium	Revisited Assumption on Trusted Governance	Security Features	Mitigated

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 Reentrancy Risk in yVault

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Vault
- Category: Time and State [13]
- CWE subcategory: CWE-663 [6]

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 [20] exploit, and the recent Uniswap/Lendf.Me hack [19].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `yVault` as an example, the `deposit()` 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 77) starts before effecting the update on internal states (line 86), 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 very same `deposit()` function.

```
74     function deposit(uint256 _amount) public {
75         uint256 _pool = balance();
76         uint256 _before = token.balanceOf(address(this));
77         token.safeTransferFrom(msg.sender, address(this), _amount);
78         uint256 _after = token.balanceOf(address(this));
```

```

79     _amount = _after.sub(_before); // Additional check for deflationary tokens
80     uint256 shares = 0;
81     if (totalSupply() == 0) {
82         shares = _amount;
83     } else {
84         shares = (_amount.mul(totalSupply())).div(_pool);
85     }
86     _mint(msg.sender, shares);
87 }

```

Listing 3.1: yVault::deposit()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

```

74     function deposit(uint256 _amount) public nonReentrant {
75         uint256 _pool = balance();
76         uint256 _before = token.balanceOf(address(this));
77         token.safeTransferFrom(msg.sender, address(this), _amount);
78         uint256 _after = token.balanceOf(address(this));
79         _amount = _after.sub(_before); // Additional check for deflationary tokens
80         uint256 shares = 0;
81         if (totalSupply() == 0) {
82             shares = _amount;
83         } else {
84             shares = (_amount.mul(totalSupply())).div(_pool);
85         }
86         _mint(msg.sender, shares);
87     }

```

Listing 3.2: yVault::deposit()

Status The issue has been fixed by adding the suggested `nonReentrant` modifier.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Vault
- Category: Business Logics [12]
- CWE subcategory: CWE-841 [8]

Description

In the Stone protocol, the `yVault` contract is designed to be the main entry for interaction with farming users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., DAI). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `yVault` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```

64     function earn() public {
65         uint256 _bal = available();
66         token.safeTransfer(controller, _bal);
67         IController(controller).earn(address(token), _bal);
68     }
69
70     function depositAll() external {
71         deposit(token.balanceOf(msg.sender));
72     }
73
74     function deposit(uint256 _amount) public {
75         uint256 _pool = balance();
76         uint256 _before = token.balanceOf(address(this));
77         token.safeTransferFrom(msg.sender, address(this), _amount);
78         uint256 _after = token.balanceOf(address(this));
79         _amount = _after.sub(_before); // Additional check for deflationary tokens
80         uint256 shares = 0;
81         if (totalSupply() == 0) {
82             shares = _amount;
83         } else {
84             shares = (_amount.mul(totalSupply())).div(_pool);
85         }
86         _mint(msg.sender, shares);
87     }

```

Listing 3.3: `yVault.sol`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet

the assumption behind these low-level asset-transferring routines.

Note that the `deposit()` routine is enhanced to properly support deflationary tokens. However, other functions are not! For example, the `earn()` routine transfers the assets from the current vault to the `controller` and the amount involved in `transfer()` has not been properly adjusted for the support of deflationary tokens. Therefore, these operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the `yVault`. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., `USDT`) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted `USDT`.

Status This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

3.3 Possible Costly yVault LPs From Improper Liquidity Initialization

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Vault
- Category: Time and State [10]
- CWE subcategory: CWE-362 [5]

Description

The `Stone` protocol allows users to deposit supported assets and get in return `st`-wrapped tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token, i.e., `stUSDC`, extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. This routine is used for participating users to deposit the supported assets (e.g., `USDC`) and get respective `stUSDC` pool tokens in return. The

issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```

74     function deposit(uint256 _amount) public {
75         uint256 _pool = balance();
76         uint256 _before = token.balanceOf(address(this));
77         token.safeTransferFrom(msg.sender, address(this), _amount);
78         uint256 _after = token.balanceOf(address(this));
79         _amount = _after.sub(_before); // Additional check for deflationary tokens
80         uint256 shares = 0;
81         if (totalSupply() == 0) {
82             shares = _amount;
83         } else {
84             shares = (_amount.mul(totalSupply())).div(_pool);
85         }
86         _mint(msg.sender, shares);
87     }

```

Listing 3.4: yVault::deposit()

Specifically, when the pool is being initialized (line 81), the share value directly takes the value of amount (line 82), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of USDC assets with the goal of making the `stUSDC` pool token extremely expensive.

An extremely expensive `stUSDC` pool token can be very inconvenient to use as a small number of `1WEI` may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool 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 `deposit()` to defensively calculate the share amount when the pool 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 confirmed. The team will exercise extra caution in properly initializing the vault.

3.4 Suggested Governance-Restricted `earn()`

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Vault
- Category: Time and State [14]
- CWE subcategory: CWE-682 [7]

Description

In the `stone` protocol, a number of new `strategy` contracts have been designed and implemented to invest farmers' assets (held in `vaults`), harvest growing yields, and sell any gains, if any, to the original asset. In order to have a smooth investment experience, the `yVault` contract opens up a public function, i.e., `earn()`, that can be invoked by anyone to kick off the investment.

```

64     function earn() public {
65         uint256 _bal = available();
66         token.safeTransfer(controller, _bal);
67         IController(controller).earn(address(token), _bal);
68     }

```

Listing 3.5: `yVault::earn()`

Unfortunately, this public entry has been exploited in a number of recent incidents (`yDAI` and `BT` hacks [18, 1]) that prompt the need of a guarded call to the `earn()`. By doing so, it ensures the assets in `yVault` will not blindly deposited into a faulty strategy that is currently not making any profit.

Recommendation Ensure the `earn()` can only be called via a trusted entity.

Status This issue has been fixed by properly validating the caller of `earn()`.

3.5 Potential Front-Running/MEV With Reduced Buyback

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Strategies
- Category: Time and State [14]
- CWE subcategory: CWE-682 [7]

Description

As mentioned in Section 3.4, the `stone` protocol has designed a number of new `strategy` contracts to invest farmers' assets (held in `vaults`), harvest growing yields, and sell any gains, if any, to the

original asset.

Using the StrategyDForceUSDC strategy as an example, the authorized strategist or the governance can call `harvest()` that basically collects any pending rewards (via `dRewards(pool).getReward()` - line 128) and swaps them to the designated `want` (line 139) for the next round of investment (via `deposit()` - line 145).

```

126     function harvest() public {
127         require(msg.sender == strategist || msg.sender == governance, "!authorized");
128         dRewards(pool).getReward();
129         uint256 _df = IERC20(df).balanceOf(address(this));
130         if (_df > 0) {
131             IERC20(df).safeApprove(uni, 0);
132             IERC20(df).safeApprove(uni, _df);
133
134             address[] memory path = new address[](3);
135             path[0] = df;
136             path[1] = weth;
137             path[2] = want;
138
139             Uni(uni).swapExactTokensForTokens(_df, uint256(0), path, address(this), now.add(1800));
140         }
141         uint256 _want = IERC20(want).balanceOf(address(this));
142         if (_want > 0) {
143             uint256 _fee = _want.mul(performanceFee).div(performanceMax);
144             IERC20(want).safeTransfer(IController(controller).rewards(), _fee);
145             deposit();
146         }
147     }

```

Listing 3.6: StrategyDForceUSDC::harvest()

We notice the collected yields are routed to `UniswapV2` in order to swap them to `want` as rewards. And the swap operation does not specify any restriction 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 or the strategy contract in our case 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

protect the interests of farming users.

Status This issue has been confirmed. However, as mentioned earlier, the front-running attack is inherent in current DEXes and there is still a need to search for more effective countermeasures.

3.6 Improved Sanity Checks For System Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Controller, Strategies
- Category: Coding Practices [11]
- CWE subcategory: CWE-1126 [3]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Stone` protocol is no exception. Specifically, if we examine `StrategyDForceUSDC`, it has defined a number of protocol-wide risk parameters, e.g., `performanceFee` and `withdrawalFee`. The first fee parameter affects the yielding gains while the last one is related to the deduction that is charged on the invested funds upon withdrawal. In the following, we show the corresponding routines that allow for their changes.

```

53     function setWithdrawalFee(uint256 _withdrawalFee) external {
54         require(msg.sender == governance, "!governance");
55         withdrawalFee = _withdrawalFee;
56     }
57
58     function setPerformanceFee(uint256 _performanceFee) external {
59         require(msg.sender == governance, "!governance");
60         performanceFee = _performanceFee;
61     }

```

Listing 3.7: `StrategyDForceUSDC::setWithdrawalFee()`

Our result shows the update logic on these fee parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large fee parameter (say more than 100%) will revert the `harvest()` operation. Moreover, in `StrategyDForceUSDC`, these two parameters take the values of 50%, 5%, respectively. These two preset values are considered too high for participating users.

Also, we notice that the `setMin()` routine in the `yVault` contract can be improved by validating the given `_min` argument. For example, we can at least ensure the following: `require (_min<max)`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status The issue has been confirmed.

3.7 Improved Asset Consistency Among Vaults, Controller, and Strategies

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Controller
- Category: Coding Practices [11]
- CWE subcategory: CWE-1099 [2]

Description

In the Stone protocol, there is a one-to-one mapping between a vault and its strategy. To properly link a vault with its strategy, it is natural for the two to operate on the same underlying asset. For example, the `yVault` allows for USDC-based deposits and withdraws. The associated strategy, i.e., a `StrategyDForceUSDC`-based instance, naturally has USDC as the underlying asset. If these two have different underlying assets, the link should not be successful.

If we examine the `setStrategy()` routine in the `controller` contract, this routine allows for dynamic binding of the vault with a new strategy (line 98). A successful binding needs to satisfy a number of requirements. One specific example is shown as follows: `require(IVault(vaults[_token]).token() == Strategy(_strategy).want())`. Apparently, this requirement guarantees the consistency of the underlying asset between the vault and its associated strategy.

```

90     function setStrategy(address _token, address _strategy) public {
91         require(msg.sender == strategist, "strategist");
92         require(approvedStrategies[_token][_strategy] == true, "approved");
93
94         address _current = strategies[_token];
95         if (_current != address(0)) {
96             Strategy(_current).withdrawAll();
97         }
98         strategies[_token] = _strategy;
99     }

```

Listing 3.8: Controller :: setStrategy()

However, if we examine the `constructor()` of various strategy contracts (e.g., `StrategyDForceUSDC` and `StrategyDForceUSDT`), the requirement of having the same underlying asset is not enforced. A new

strategy deployment with an ill-provided list of arguments with an unmatched underlying asset may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring the consistency of the underlying asset when a new strategy is being deployed or linked.

Recommendation Ensure the consistency of the underlying asset between the vault and its associated strategy. An example revision is shown below.

```

90     function setStrategy(address _token, address _strategy) public {
91         require(msg.sender == strategist, "strategist");
92         require(approvedStrategies[_token][_strategy] == true, "approved");
93         require(IVault(vaults[_token]).token() == Strategy(_strategy).want(), "asset")
94
95         address _current = strategies[_token];
96         if (_current != address(0)) {
97             Strategy(_current).withdrawAll();
98         }
99         strategies[_token] = _strategy;
100     }

```

Listing 3.9: Controller :: setStrategy()

Status The issue has been fixed by ensuring the linked vault and strategy have the same underlying asset.

3.8 Revisited Assumption on Trusted Governance

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Vault, Controller
- Category: Security Features [9]
- CWE subcategory: CWE-287 [4]

Description

In the Stone protocol, the governance account plays a critical role in governing and regulating the system-wide operations (e.g., vault/strategy addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., vault, controller, and strategy.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we examine the current privilege management graph in the Stone protocol (Figure 3.1).

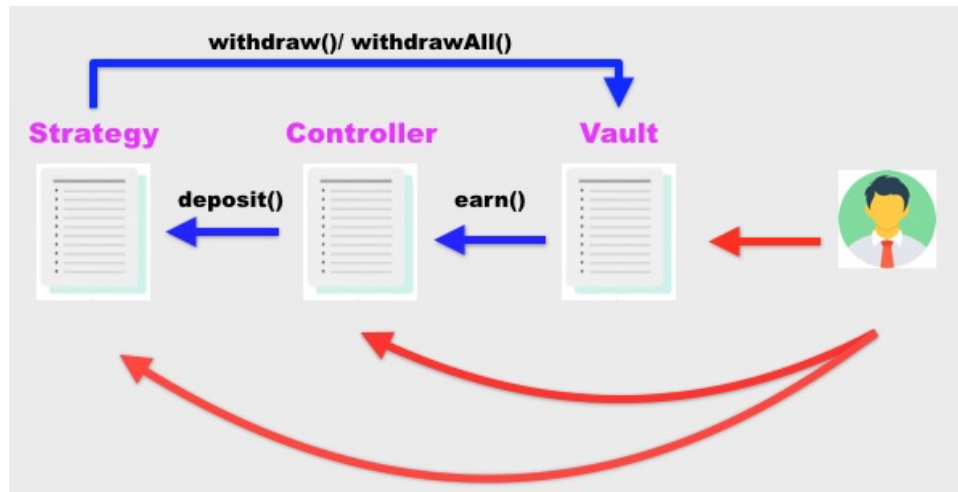


Figure 3.1: The Privilege Management Chain in Stone

We emphasize that the privilege assignment among `vault`, `controller`, and `strategy` is properly administrated. However, it is worrisome that `governance` is not governed by a DAO-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `governance` account would allow the attacker to add a malicious `controller` to steal all funds whenever the `earn()` call is made. It could also allow for the dynamic addition of a new malicious `strategy`, which directly undermines the assumption of the `Stone` protocol.

Recommendation Promptly transfer the `governance` privilege to the intended DAO-like governance contract. And 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 and partially mitigated with a multi-sig account to regulate the `governance/controller` privileges.

4 | Conclusion

In this audit, we thoroughly analyzed the design and implementation of the `stone` protocol. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. Developed on top of `yfi`, the `stone` 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] BT Finance. BT.Finance Exploit Analysis Report. <https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28>.
- [2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. <https://cwe.mitre.org/data/definitions/1099.html>.
- [3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [4] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). <https://cwe.mitre.org/data/definitions/362.html>.
- [6] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.html>.
- [7] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [8] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [9] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.

-
- [10] MITRE. CWE CATEGORY: 7PK - Time and State. <https://cwe.mitre.org/data/definitions/361.html>.
- [11] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [12] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [13] MITRE. CWE CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [14] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [15] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [16] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [17] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [18] PeckShield. The yDAI Incident Analysis: Forced Investment. <https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5>.
- [19] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [20] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.