SMART CONTRACT AUDIT REPORT

for

STACKER VENTURES

Prepared By: Shuxiao Wang

PeckShield
March 11, 2021

Document Properties

Client	Stacker Ventures
Title	Smart Contract Audit Report
Target	Stacker.VC
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	March 11, 2021	Xuxian Jiang	Final Release
1.0-rc1	January 31, 2021	Xuxian Jiang	Release Candidate #1
0.2	January 23, 2021	Xuxian Jiang	Additional Findings
0.1	January 18, 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

Given the opportunity to review the design document and related smart contract source code of the `Stacker.VC` 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 Stacker Ventures

`Stacker Ventures` is a decentralized, community-owned venture capital protocol and accelerator that focuses on early-stage investment with aligned incentives of community investors with founding teams. By creating full functionality for a trust-minimized, decentralized VC investment fund, the protocol provides holders proportional claim over assets in the VC investment fund managed by a decentralized "fund council". These "fund council" members are community-elected to manage subsequent funds and consist of publicly available and auditable individuals. There are policies enforced in the protocol to delineate the responsibilities and limit the power of the fund council members.

The basic information of the Stacker.VC module is as follows:

Table 1.1: Basic Information of The Stacker.VC Module

Item	Description
Issuer	Stacker Ventures
Website	https://stacker.vc/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 11, 2021

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

- https://github.com/jack0x-tech/StackerVC_VentureFund001/ (3cfb98b)

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

- https://github.com/jack0x-tech/StackerVC_VentureFund001/ (87b1581)

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.

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 1.3: 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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the Stacker.VC 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	2	■■
Low	4	■■■■
Informational	1	■
Total	7	

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 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Stacker.VC Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Safe-Version Replacement of safeTransfer()/safeTransferFrom()	Coding Practices	Fixed
PVE-002	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-003	Low	Incompatibility with Deflationary/Rebasing Token	Business Logics	Confirmed
PVE-004	Medium	Non-Functional setEndBlock()/deposit() in LPGauge	Business Logics	Fixed
PVE-005	Low	Asset Consistency in VaultGaugeBridge::newBridge()	Business Logics	Fixed
PVE-006	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-007	Informational	Inconsistency Between Document and Implementation	Coding Practices	Fixed

Besides recommending specific countermeasures to mitigate these issues, based on the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.11` instead of specifying a range, e.g., `pragma solidity ^0.6.11`.

In addition, 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 Safe-Version Replacement of safeTransfer()/safeTransferFrom()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GaugeD1`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

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 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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

As an example, we show below the `deposit()` routine in the `GaugeD1` contract. If the `USDT` token is supported as `acceptToken`, the unsafe version of `IERC20(acceptToken).transfer(vcHolding, _amountCommitHard)` (line 122) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
99      function deposit(uint256 _amountCommitSoft, uint256 _amountCommitHard, address
            _creditTo) nonReentrant external {
100         require(block.number <= endBlock, "GAUGE: distribution 1 over");
101         require(fundOpen    _amountCommitHard == 0, "GAUGE: !fundOpen, only soft commit
                allowed"); // when the fund closes, soft commits are still accepted
102         require(msg.sender == _creditTo    msg.sender == vaultGaugeBridge, "GAUGE: !bridge
                 for creditTo"); // only the bridge contract can use the "creditTo" to
                credit !msg.sender

104         _claimSTACK(_creditTo); // new deposit doesn't get tokens right away

106         // transfer tokens from sender to account
107         uint256 _acceptTokenAmount = _amountCommitSoft.add(_amountCommitHard);
108         if (_acceptTokenAmount > 0){
109             IERC20(acceptToken).safeTransferFrom(msg.sender, address(this),
                    _acceptTokenAmount);
110         }

112         CommitState memory _state = balances[_creditTo];
113         // no need to update _state.tokensAccrued because that's already done in
                _claimSTACK
114         if (_amountCommitSoft > 0){
115             _state.balanceCommitSoft = _state.balanceCommitSoft.add(_amountCommitSoft);
116             depositedCommitSoft = depositedCommitSoft.add(_amountCommitSoft);
```

```
        }
        if (_amountCommitHard > 0){
            _state.balanceCommitHard = _state.balanceCommitHard.add(_amountCommitHard);
            depositedCommitHard = depositedCommitHard.add(_amountCommitHard);

            IERC20(acceptToken).transfer(vcHolding, _amountCommitHard);
        }

        emit Deposit(_creditTo, _amountCommitSoft, _amountCommitHard);
        balances[_creditTo] = _state;
    }
```

Listing 3.2: GaugeD1::deposit()

Recommendation Accommodate the above-mentioned idiosyncrasies about ERC20-related `transfer()` and `transferFrom()`.

Status The issue has been fixed by this commit: `2c9d64b`.

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GaugeD1`, `LPGuarge`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

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 are several occasions the `checks-effects-interactions` principle is violated. Using the `LPGuarge` 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 80) starts before effecting the update on internal states (line 88), hence violating the principle. In this particular case, if the external contract has some hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function.

```
    function deposit(uint256 _amount) nonReentrant external {
        require(block.number <= endBlock, "LPGAUGE: distribution over");

        _claimSTACK(msg.sender);

        IERC20(token).safeTransferFrom(msg.sender, address(this), _amount);

        DepositState memory _state = balances[msg.sender];
        _state.balance = _state.balance.add(_amount);
        deposited = deposited.add(_amount);

        emit Deposit(msg.sender, _amount);

        balances[msg.sender] = _state;
    }
```

Listing 3.3: LPGuarge::deposit()

Another similar violation can be found in `deposit()`, `withdraw()` and `upgradeCommit()` routines within the `GaugeD1` contract.

In the meantime, we should mention that the related routines are properly protected with `nonReentrant` and the related token contract is not vulnerable or exploitable for `re-entrancy`.

Recommendation Suggest to follow the `checks-effects-interactions` best practice in addition to current reentrancy protection.

Status The issue has been confirmed.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [6]

Description

In the `Stacker.VC` protocol, the `GaugeD1` contract is designed to be the main entry for interaction with investing 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 `GaugeD1` 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.

```
    function deposit(uint256 _amountCommitSoft, uint256 _amountCommitHard, address
        _creditTo) nonReentrant external {
        require(block.number <= endBlock, "GAUGE: distribution 1 over");
        require(fundOpen  _amountCommitHard == 0, "GAUGE: !fundOpen, only soft commit
            allowed"); // when the fund closes, soft commits are still accepted
        require(msg.sender == _creditTo  msg.sender == vaultGaugeBridge, "GAUGE: !bridge
             for creditTo"); // only the bridge contract can use the "creditTo" to
            credit !msg.sender

        _claimSTACK(_creditTo); // new deposit doesn't get tokens right away

        // transfer tokens from sender to account
        uint256 _acceptTokenAmount = _amountCommitSoft.add(_amountCommitHard);
        if (_acceptTokenAmount > 0){
            IERC20(acceptToken).safeTransferFrom(msg.sender, address(this),
                _acceptTokenAmount);
        }

        CommitState memory _state = balances[_creditTo];
        // no need to update _state.tokensAccrued because that's already done in
            _claimSTACK
        if (_amountCommitSoft > 0){
            _state.balanceCommitSoft = _state.balanceCommitSoft.add(_amountCommitSoft);
            depositedCommitSoft = depositedCommitSoft.add(_amountCommitSoft);
        }
        if (_amountCommitHard > 0){
            _state.balanceCommitHard = _state.balanceCommitHard.add(_amountCommitHard);
            depositedCommitHard = depositedCommitHard.add(_amountCommitHard);

            IERC20(acceptToken).transfer(vcHolding, _amountCommitHard);
        }

        emit Deposit(_creditTo, _amountCommitSoft, _amountCommitHard);
        balances[_creditTo] = _state;
    }
```

Listing 3.4: GaugeD1::deposit()

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.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol. 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.

3.4 Non-Functional setEndBlock()/deposit() in LPGauge

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LPGauge`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Stacker.VC protocol is no exception. Specifically, if we examine the `LPGauge` contract, it has defined a number of system-wide risk parameters: `emissionRate`, `startBlock`, and `endBlock`.

To elaborate, we show below the related configuration routines in the `LPGauge` contract for the risk parameter – `setEndBlock()`.

```
    function setEndBlock(uint256 _block) external {
        require(msg.sender == governance, "LPGAUGE: !governance");
        require(block.number <= endBlock, "LPGAUGE: distribution already done, must start another");
        require(block.number <= _block, "LPGAUGE: can't set endBlock to past block");

        endBlock = _block;
    }

    function deposit(uint256 _amount) nonReentrant external {
        require(block.number <= endBlock, "LPGAUGE: distribution over");

        _claimSTACK(msg.sender);

        IERC20(token).safeTransferFrom(msg.sender, address(this), _amount);

        DepositState memory _state = balances[msg.sender];
        _state.balance = _state.balance.add(_amount);
```

```
        deposited = deposited.add(_amount);

        emit Deposit(msg.sender, _amount);

        balances[msg.sender] = _state;
    }
```

Listing 3.5: Two LPGauge Routines: setEndBlock() and deposit()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these 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, the above configuration of `endBlock` is always reverted as `endBlock = startBlock + 100` where `startBlock` is hardcoded as 300! Therefore, every call of `setEndBlock()` and `deposit()` fails. In other words, no one is able to deposit into `LPGauge`, which is clearly not the purpose.

Recommendation Revise the hardcoded `startBlock` to be an appropriate number. In the meantime, validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed in the following commit: `87b1581`.

3.5 Asset Consistency in VaultGaugeBridge::newBridge()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `VaultGaugeBridge`
- Category: Business Logics [8]
- CWE subcategory: CWE-837 [5]

Description

In Stacker.VC, the `VaultGaugeBridge` contract facilitates users in allowing them to deposit into `yEarn vault` and then into the `GaugeD1` in a single transaction. Naturally, there is a one-to-one mapping between a `yEarn vault` and its `gauge`. To properly link a `vault` with its `gauge`, it is a need for the two to operate on the same underlying asset. For example, the `yEarn vault` allows for `DAI`-based deposits and withdraws. The associated `gauge` naturally has `DAI` as the underlying asset. If these two have different underlying assets, the link should not be successful.

If we examine the `newBridge()` routine in the `VaultGaugeBridge` contract, this routine allows for dynamic binding of the `vault` with a new `gauge` (line 55). A successful binding needs to satisfy a number of requirements. One specific example is shown as follows: `_gauge.acceptToken == _vault`.

Apparently, this requirement guarantees the consistency of the underlying asset between the `vault` and its associated `gauge`.

```
    // create a new bridge, warning this allows an overwrite
    function newBridge(address _vault, address _gauge) external {
        require(msg.sender == governance, "BRIDGE: !governance");

        bridges[_vault] = _gauge;
    }
```

Listing 3.6: VaultGaugeBridge::newBridge()

However, if we examine the above logic, the requirement of having the same underlying asset is not enforced. An unmatched deployment or configuration between `vault` and `gauge` may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring their consistency when the mapping is being linked.

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

```
    // create a new bridge, warning this allows an overwrite
    function newBridge(address _vault, address _gauge) external {
        require(msg.sender == governance, "BRIDGE: !governance");
        require{_gauge.acceptToken == _vault},
        bridges[_vault] = _gauge;
    }
```

Listing 3.7: VaultGaugeBridge::newBridge()

Status This issue has been resolved as the team decides to only support `ibETHv2` deposits into the `GaugeD1` contract, instead of many different `yEarn` tokens.

3.6 Improved Precision By Multiplication And Division Reordering

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [3]

Description

In Stacker.VC, there is a `VCTreasuryV1` contract that creates full functionality for a trust-minimized, decentralized VC investment fund. Within this contract, there is a routine, i.e., `startFund()`. This routine is called once all tokens have been issued. By calling this routine, the "fund council" actually seeds the fund with `ETH`, and the seeded fund will go into an `ACTIVE` state. At this point, new investments can then be made.

To elaborate, we show below this routine's implementation. It comes to our attention that this routine computes the `maxInvestment` threshold as `maxInvestment = msg.value.div(max).mul(investmentCap)` (line 180).

```
170     // seed the fund with ETH and start it up. 3 years until the fund is dissolved
171     function startFund() payable external {
172         require(currentState == FundStates.setup, "TREASURYV1: !FundStates.setup");
173         require(msg.sender == councilMultisig, "TREASURYV1: !councilMultisig");
174         require(totalSupply() > 0, "TREASURYV1: invalid setup"); // means fund tokens
                were not issued

176         fundStartTime = block.timestamp;
177         fundCloseTime = block.timestamp.add(ONE_YEAR);

179         initETH = msg.value;
180         maxInvestment = msg.value.div(max).mul(investmentCap);

182         _changeFundState(FundStates.active); // set fund active!
183     }
```

Listing 3.8: VCTreasuryV1::startFund()

It is important to note that the lack of `float` support in `Solidity` may introduce subtle, but troublesome issue: precision loss. One possible precision loss stems from the computation when both multiplication (`mul`) and division (`div`) are involved. Specifically, the computation at line 180 is better performed as follows: `maxInvestment = msg.value.mul(investmentCap).div(max)`.

A better approach is to avoid any unnecessary division operation that might lead to precision loss. In other words, the computation of the form `A / B * C` can be converted into `A * C / B` under the condition that `A * C` does not introduce any overflow.

Similar precision improvements can also be observed in the arithmetic operations in other routines, e.g, `getUtilization()`, `killQuorumRequirement()`, `pauseQuorumRequirement()`, and `_assessFee()`.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in `Solidity`. An example revision to the above computation is shown as: `maxInvestment = msg.value.mul(investmentCap).div(max)`.

Status The issue has been fixed by this commit: `2c65e20`.

3.7 Inconsistency Between Document and Implementation

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `VCTreasuryV1`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

There exist a misleading comment embedded among lines of solidity code, which brings unnecessary hurdles to understand and/or maintain the software.

The comment can be found in line 170 of `VCTreasuryV1::startFund()`. the preceding function header indicates that the fund will be dissolved after 3 years. However, the `fundCloseTime` state (line 177) indicates it takes one year to dissolve.

```
170     // seed the fund with ETH and start it up. 3 years until the fund is dissolved
171     function startFund() payable external {
172         require(currentState == FundStates.setup, "TREASURYV1: !FundStates.setup");
173         require(msg.sender == councilMultisig, "TREASURYV1: !councilMultisig");
174         require(totalSupply() > 0, "TREASURYV1: invalid setup"); // means fund tokens
                were not issued

176         fundStartTime = block.timestamp;
177         fundCloseTime = block.timestamp.add(ONE_YEAR);

179         initETH = msg.value;
180         maxInvestment = msg.value.div(max).mul(investmentCap);

182         _changeFundState(FundStates.active); // set fund active!
183     }
```

Listing 3.9: VCTreasuryV1::startFund()

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been fixed by removing the aforementioned inconsistency.

4 Conclusion

In this audit, we have analyzed the Stacker.VC design and implementation. The protocol aims to develop a decentralized, community-owned venture capital protocol and accelerator with a focus on early-stage investment that aligns incentives of community investors with founding teams. During the audit, we notice that the current implementation is well 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[5] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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