SMART CONTRACT AUDIT REPORT

for

SMOOTHY FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China
February 15, 2021

Document Properties

Client	Smoothy Finance
Title	Smart Contract Audit Report
Target	SmoothyV1
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 15, 2021	Xuxian Jiang	Final Release
1.0-rc	December 18, 2020	Xuxian Jiang	Release Candidate
0.3	December 17, 2020	Xuxian Jiang	Additional Findings #2
0.2	December 15, 2020	Xuxian Jiang	Additional Findings #1
0.1	December 10, 2020	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 **SmoothyV1** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given 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 Smoothy Finance

Smoothy Finance is a novel automated market maker (AMM) that supports 10+ same-backed assets (such as stablecoins) in a single pool with low swapping fee and high interest earning. The single pool feature greatly maximizes the liquidity of all stablecoins without worrying about fragmented liquidity caused by multiple pools - a common way used by existing protocols (such as `Curve`). Moreover, thanks to its bonding curve, the gas cost of swapping is extremely low - even about $10x$ lower than that of `mStable/Curve yPool`. Finally, its `dynamic cash reserve` (DSR) algorithm can further maximize the interest earned from the underlying lending platforms without incurring extra gas cost for normal transactions.

The basic information of SmoothyV1 is as follows:

Table 1.1: Basic Information of SmoothyV1

Item	Description
Issuer	Smoothy Finance
Website	https://www.smoothy.finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 15, 2021

In the following, we show the reviewed file and the `md5` checksum value used in this audit. We clarify that this is not an economic audit and the pros/cons of the adopted bonding curve as well as its applicability are not part of this audit.

- audit.zip (md5: d282eba17c9f4d14e92ba36f4758441a)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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 accordingly classified into four categories, 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) [10], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit 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 SmoothyV1 design and 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	6	■■■■■■
Informational	2	■■
Total	10	

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 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, 6 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Token Information Initialization	Business Logic	Fixed
PVE-002	Low	Possible Failed rebalanceReserve() For Non-YEnable Tokens	Business Logic	Fixed
PVE-003	Informational	Consistent Usage of _systemParameter	Coding Practices	Fixed
PVE-004	Low	Possible Inconsistency From Non-Full yToken Withdrawal	Business Logic	Confirmed
PVE-005	Informational	Improved Event Generation With Indexed Assets	Time and State	Fixed
PVE-006	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Confirmed
PVE-007	Medium	Revisited Trust on Admin Keys	Security Features	Confirmed
PVE-008	Low	Incompatibility With Deflationary/Rebasing Tokens	Time And State	Confirmed
PVE-009	Medium	Accommodation of approve() Idiosyncrasies	Business Logic	Fixed
PVE-010	Low	Possible Outdated Balance For Share Calculation	Business Logic	Confirmed

Beside the identified issues, due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, we always suggest to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly specify the Solidity compiler version, e.g., `pragma solidity 0.6.0`, instead of `pragma solidity ^0.6.0`.

In the meantime, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Token Information Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SmoothyV1
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

The design of SmoothyV1 makes the best from two popular DeFi protocols – mStable and Curve. The single pool feature also consolidates all liquidities without being fragmented. By design, it is able to support $10x$ same-backed assets (such as stablecoins) in a single pool.

For gas efficiency and improved scalability, each same-backed asset (such as a stablecoin) is effectively packed in an internal data structure, i.e., _tokenInfos. Each occupies 256-bit size and can be divided into the following 7 segments: token address, soft weight, hard weight, yEnable bit, decimal multiplier, token index and unused. The above segments occupy 160 bits, 20 bits, 20 bits, 1 bit, 1 bit, 5 bits, 8 bits, and 41 bits, respectively.

During our analysis on its constructor routine (see the code snippet below), we notice that the routine properly initializes nearly all segments for the pre-configured set of same-backed assets. However, it forgets to initialize their hard weights.

```
    constructor (
        address[] memory tokens,
        address[] memory yTokens,
        uint256[] memory decMultipliers
    )
        public
        ERC20("Smoothy LP Token", "syUSD")
    {
        require(tokens.length == yTokens.length, "tokens and ytokens must have the same
            length");
        _rewardCollector = msg.sender;
```

```

        uint256 para = 0;

        for (uint8 i = 0; i < tokens.length; i++) {
            uint256 info = uint256(tokens[i]);
            info = _setSoftWeight(info, W_ONE);
            info = _setDecimalMultiplier(info, decMultipliers[i]);
            info = _setTID(info, i);
            _yTokenAddresses[i] = yTokens[i];
            // _balances[i] = 0; // no need to set
            if (yTokens[i] != address(0x0)) {
                info = _setYEnabled(info, true);
            }
            _tokenInfos[i] = info;
        }
        para = _setNTokens(para, tokens.length);
        _systemParameter = para;
    }
```

Listing 3.1: SmoothyV1::constructor()

Recommendation Initialize the `hard weights` of same-backed assets as well in the constructor routine. An example revision is shown below:

```
    constructor (
        address[] memory tokens,
        address[] memory yTokens,
        uint256[] memory decMultipliers
    )
        public
        ERC20("Smoothy LP Token", "syUSD")
    {
        require(tokens.length == yTokens.length, "tokens and ytokens must have the same
            length");
        _rewardCollector = msg.sender;

        uint256 para = 0;

        for (uint8 i = 0; i < tokens.length; i++) {
            uint256 info = uint256(tokens[i]);
            info = _setSoftWeight(info, W_ONE);
            info = _setHardWeight(info, W_ONE);
            info = _setDecimalMultiplier(info, decMultipliers[i]);
            info = _setTID(info, i);
            _yTokenAddresses[i] = yTokens[i];
            // _balances[i] = 0; // no need to set
            if (yTokens[i] != address(0x0)) {
                info = _setYEnabled(info, true);
            }
            _tokenInfos[i] = info;
        }
        para = _setNTokens(para, tokens.length);
```

```
        _systemParameter = para;
    }
```

Listing 3.2: SmoothyV1::constructor()

Status The issue has been fixed by adding the missing `_setHardWeight()` call in the constructor.

3.2 Possible Failed rebalanceReserve() For Non-YEnable Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmoothyV1`
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

Smoothy Finance features a unique `dynamic cash reserve` (DSR) algorithm to maximize the interest earned from the underlying lending platforms without incurring extra gas cost for normal transactions. The algorithm kicks in when any operation from `mint/redeem/swap` leads to insufficient cash balance for withdrawal or high cash percentage ($> 20\%$). After the adjustment, the algorithm ensures the cash percentage remains at 10%.

In the following, we show the key rebalance routine, i.e., `_rebalanceReserve`. This routine relies on another helper function – `_getBalanceDetail()` to obtain current balance details.

```
    function _rebalanceReserve(
        uint256 info
    )
        internal
    {
        uint256 pricePerShare;
        uint256 cashUnnormalized;
        uint256 yBalanceUnnormalized;
        (pricePerShare, cashUnnormalized, yBalanceUnnormalized) = _getBalanceDetail(info
            );
        uint256 tid = _getTID(info);

        // Update _totalBalance with interest
        _updateTotalBalanceWithNewYBlance(tid, yBalanceUnnormalized.mul(
            _normalizeBalance(info)));

        uint256 targetCash = yBalanceUnnormalized.add(cashUnnormalized).div(10);
        if (cashUnnormalized > targetCash) {
            uint256 depositAmount = cashUnnormalized.sub(targetCash);
            IERC20(address(info)).approve(_yTokenAddresses[tid], depositAmount);
            YERC20(_yTokenAddresses[tid]).deposit(depositAmount);
```

```
            _yBalances[tid] = yBalanceUnnormalized.add(depositAmount).mul(
                _normalizeBalance(info));
        } else {
            YERC20(_yTokenAddresses[tid]).withdraw(targetCash.sub(cashUnnormalized).mul(
                W_ONE).div(pricePerShare));
            _yBalances[tid] = yBalanceUnnormalized.sub(targetCash.sub(cashUnnormalized))
                .mul(_normalizeBalance(info));
        }
    }
```

Listing 3.3: SmoothyV1::_rebalanceReserve()

```
    function _getBalanceDetail(
        uint256 info
    )
        internal
        view
        returns (uint256 pricePerShare, uint256 cashUnnormalized, uint256
            yBalanceUnnormalized)
    {
        address yAddr = _yTokenAddresses[_getTID(info)];
        pricePerShare = YERC20(yAddr).getPricePerFullShare();
        cashUnnormalized = IERC20(address(info)).balanceOf(address(this));
        uint256 share = YERC20(yAddr).balanceOf(address(this));
        yBalanceUnnormalized = share.mul(pricePerShare).div(W_ONE);
    }
```

Listing 3.4: SmoothyV1::_getBalanceDetail()

It comes to our attention that the balance details require an external interaction with the underlying `ytoken` for its `getPricePerFullShare()`. However, if the `yEnable` flag is off, the `_getBalanceDetail()` call will be reverted, resulting in the rebalance failure.

Recommendation If an token has not turned on `yEnable`, the `rebalance` policy should not be applied to this particular token.

Status The issue has been fixed by requiring `_isYEnabled()` for the rebalanced token.

3.3 Consistent Usage of _systemParameter

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SmoothyV1`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

In current codebase of SmoothyV1, there is a global state `_systemParameter` that is designed to configure the total number of tokens supported in current pool. If we visit the `getTokenStats()`, this `_systemParameter` state returns the number of tokens via `_getNTokens()`. To elaborate, we show below these two routines – `getTokenStats()` and `_getNTokens()`.

```
    function getTokenStats(uint256 bTokenIdx)
        public
        view
        returns (uint256 softWeight, uint256 hardWeight, uint256 balance, uint256
            decimals)
    {
        uint256 para = _systemParameter;
        uint256 ntokens = _getNTokens(para);
        require(bTokenIdx < ntokens, "Backed token is not found!");

        uint256 info = _tokenInfos[bTokenIdx];

        balance = _getBalance(info).div(_normalizeBalance(info));
        softWeight = _getSoftWeight(info);
        hardWeight = _getHardWeight(info);
        decimals = ERC20(address(info)).decimals();
    }
```

Listing 3.5: SmoothyV1::getTokenStats()

```
    /***************************************
     * Methods to change system parameters
     ***************************************/
    function _getNTokens(uint256 para) internal pure returns (uint256 ntokens) {
        return para & ((U256_1 << 6) - 1);
    }

    function _setNTokens(uint256 para, uint256 ntokens) internal pure returns (uint256
        newPara) {
        require(ntokens < (U256_1 << 6), "ntoken is too large");

        newPara = para & ~((U256_1 << 6) - 1);
```

```
        newPara = newPara   ntokens;
```

Listing 3.6: SmoothyV1::_getNTokens()

The wrapper routines of `_getNTokens()` and `_setNTokens()` are helpful in avoiding the exposure of internal details and provide a clean interface for operation. However, it also comes to our attention that `_systemParameter` is directly used for `ntokens` purposes. An example is the following `_getBalancesAndWeights()` routine.

```
    function _getBalancesAndWeights()
        internal
        view
        returns (uint256[] memory balances, uint256[] memory softWeights, uint256[]
            memory hardWeights, uint256 totalBalance)
    {
        uint256 ntokens = _systemParameter;
        balances = new uint256[](ntokens);
        softWeights = new uint256[](ntokens);
        hardWeights = new uint256[](ntokens);
        totalBalance = 0;
        for (uint8 i = 0; i < ntokens; i++) {
            uint256 info = _tokenInfos[i];
            balances[i] = _getCashBalance(info);
            if (_isYEnabled(info)) {
                balances[i] = balances[i].add(_yBalances[i]);
            }
            totalBalance = totalBalance.add(balances[i]);
            softWeights[i] = _getSoftWeight(info);
            hardWeights[i] = _getHardWeight(info);
        }
    }
```

Listing 3.7: SmoothyV1::_getBalancesAndWeights()

Such an inconsistent usage of `_systemParameter` brings an unnecessary confusion and may introduce issues if `_systemParameter` also includes other information besides the number of supported tokens. Accordingly, we suggest to adhere to a unified interface for access and manipulation.

Recommendation Be consistent when accessing and interpreting `_systemParameter`. If it only contains the number of supported tokens, simply change the name to `_ntokens`.

Status The issue has been fixed by renaming `_systemParameter` as `_ntokens`.

3.4 Possible Inconsistency From Non-Full yToken Withdrawal

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmoothyV1`
- Category: Business Logic [9]
- CWE subcategory: CWE-837 [5]

Description

By design, SmoothyV1 makes good use of `yToken` to accumulate additional revenue for the pool assets. The use of `yToken` is configured via the so-called `yEnable` (Section 3.1). Note that the `yToken` has been widely used in a number of DeFi protocols. For example, the `yPool` in `Curve` has proved the success of such approach. However, in the unlikely situation, `yToken` may suffer from loss and the current usage of `yToken` assumes it is always reliable and has not taken the loss into account yet.

In the following, we show the `_rebalanceReserve()` routine that is used during rebalance. In particular, it assumes the `withdraw()` operation (line 541) from `yToken` returns the expected amount. This may not be true as evident from a number of real-world incidents.

```
520     function _rebalanceReserve(
521         uint256 info
522     )
523         internal
524     {
525         uint256 pricePerShare;
526         uint256 cashUnnormalized;
527         uint256 yBalanceUnnormalized;
528         (pricePerShare, cashUnnormalized, yBalanceUnnormalized) = _getBalanceDetail(info
            );
529         uint256 tid = _getTID(info);

531         // Update _totalBalance with interest
532         _updateTotalBalanceWithNewYBlance(tid, yBalanceUnnormalized.mul(
            _normalizeBalance(info)));

534         uint256 targetCash = yBalanceUnnormalized.add(cashUnnormalized).div(10);
535         if (cashUnnormalized > targetCash) {
536             uint256 depositAmount = cashUnnormalized.sub(targetCash);
537             IERC20(address(info)).approve(_yTokenAddresses[tid], depositAmount);
538             YERC20(_yTokenAddresses[tid]).deposit(depositAmount);
539             _yBalances[tid] = yBalanceUnnormalized.add(depositAmount).mul(
                _normalizeBalance(info));
540         } else {
541             YERC20(_yTokenAddresses[tid]).withdraw(targetCash.sub(cashUnnormalized).mul(
                W_ONE).div(pricePerShare));
542             _yBalances[tid] = yBalanceUnnormalized.sub(targetCash.sub(cashUnnormalized))
                .mul(_normalizeBalance(info));
```

```
543             }
544         }
```

Listing 3.8: SmoothyV1::_rebalanceReserve()

When the `withdraw()` operation does not result in the expected amount, the internal record in `_yBalances[]` (line 542) may unfortunately become inconsistent and bring negative impact on the entire protocol.

Recommendation Better handle the situation when the withdrawal amount is not expected and gracefully recover from it at the protocol level.

Status This issue has been confirmed.

3.5 Improved Event Generation With Indexed Assets

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SmoothyV1`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [3]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. The`events` are typically emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

We examine the support of system-wide parameters in SmoothyV1 and pay special attention to that configuration-related `getter/setter` routines in the protocol. In the following, we list a few representative events that have been defined in SmoothyV1.

```
    event Swap(
        address buyer,
        uint256 bTokenIdIn,
        uint256 bTokenIdOut,
        uint256 inAmount,
        uint256 outAmount
    );

    event SwapAll(
        address provider,
        uint256[] amounts,
        uint256 inOutFlag,
        uint256 sTokenMintedOrBurned
```

```
    );

    event Mint(
        address provider,
        uint256 inAmounts,
        uint256 sTokenMinted
    );

    event Redeem(
        address provider,
        uint256 bTokenAmount,
        uint256 sTokenBurn
    );
```

Listing 3.9: Various Events Defined in SmoothyV1

It comes to our attention that the above list of events makes no use of `indexed` in the emitted address information. Note that each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, which means it will be attached as data (instead of a separate topic). Considering that the address information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Revise the above events by properly indexing the emitted asset information.

Status The issue has been fixed by indexing the asset information in the emitted events.

3.6 Improved Sanity Checks For System/Function Parameters

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SmoothyV1`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The SmoothyV1 protocol is no exception. Specifically, if we examine the `SmoothyV1` contract, it has defined a number of system-wide risk parameters: `_swapFee`, `_redeemFee`, `_adminFeePct`, and `_adminInterestPct_`. In the following, we show corresponding routines that allow for their changes.

```
    function changeSwapFee(uint256 swapFee) external onlyOwner {
        require(swapFee <= W_ONE, "Swap fee must <= 1");
```

```
        _swapFee = swapFee;
    }

    function changeRedeemFee(
        uint256 redeemFee
    )
        external
        onlyOwner
    {
        require(redeemFee <= W_ONE, "Redeem fee must <= 1");
        _redeemFee = redeemFee;
    }

    function changeAdminFeePct(uint256 pct) external onlyOwner {
        require (pct <= W_ONE, "Admin fee pct must <= 1");
        _adminFeePct = pct;
    }

    function changeAdminInterestPct(uint256 pct) external onlyOwner {
        require (pct <= W_ONE, "Admin interest fee pct must <= 1");
        _adminInterestPct = pct;
    }
```

Listing 3.10: FeeRateModel.sol

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, an unlikely mis-configuration of `_redeemFee` may use up all funds in the redeem operation, hence incurring cost to trading users.

In addition, a number of functions can benefit from more rigorous validation on their arguments. For example, the `_setTID()` (see the code below) can be improved by requiring the given `tid` is no more than the number of supported tokens in the protocol. The same issue is also applicable in `SmoothyV1::setYEnabled()`.

```
    function _setTID(uint256 info, uint256 tid) internal pure returns (uint256) {
        require (tid < 256, "tid is too large");
        require (_getTID(info) == 0, "tid cannot set again");
        return info | (tid << (160 + TID_OFF));
    }
```

Listing 3.11: SmoothyV1::_setTID()

Recommendation 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 confirmed.

3.7 Revisited Trust on Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SmoothyV1`
- Category: Business Logic [9]
- CWE subcategory: CWE-754 [4]

Description

In SmoothyV1, there is a privileged contract, i.e., `owner`, that plays a critical role in configuring and regulating the system-wide operations (e.g., `soft/hard weight` adjustment and `yToken` assignment). Note the `yToken` assignment directly affects the investment of deposited assets in the pool.

In the following, we show the contract's `setYEnabled()` implementation. This routine may retrieve back the full investment on previous `yToken` (line 314). The returned assets will be deposited into the new `yToken` during the rebalance operation occurred in next `swap()/mint()/redeem()/rebalance()` call.

```
309    function setYEnabled(uint256 tid, address yAddr) external onlyOwner {
310        uint256 info = _tokenInfos[tid];
311        if (_yTokenAddresses[tid] != address(0x0)) {
312            // Withdraw all tokens from yToken, and clear yBalance.
313            uint256 cash = _getCashBalance(info);
314            YERC20(_yTokenAddresses[tid]).withdraw(YERC20(_yTokenAddresses[tid]).
                   balanceOf(address(this)));
315            uint256 dcash = _getCashBalance(info).sub(cash);
316
317            // Update _totalBalance with interest
318            _updateTotalBalanceWithNewYBlance(tid, dcash);
319            _yBalances[tid] = 0;
320        }
321
322        info = _setYEnabled(info, yAddr != address(0x0));
323        _yTokenAddresses[tid] = yAddr;
324        _tokenInfos[tid] = info;
325        // If yAddr != 0x0, we will rebalance in next swap/mint/redeem/rebalance call.
326    }
```

Listing 3.12: SmoothyV1::setYEnabled()

We emphasize that the current privilege assignment to `owner` is appropriate and necessary[1]. However, it is worrisome if `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multisig account. To further eliminate

[1]The `initialize()` routine can be used directly to add same-backed assets into the pool without going through the `weight-based penalty` functions. And such addition requires the `owner` to operate only.

the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

We point out that a compromised owner account would allow the attacker to add a malicious yToken to steal all funds in the pool, which directly undermines the integrity of the entire protocol.

Recommendation Promptly transfer the owner 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.

3.8 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SmoothyV1
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

In Smoothy Finance, the SmoothyV1 contract is designed to be the main entry for interaction with trading users. In particular, one entry routine, i.e., mint(), accepts user deposits of supported assets (e.g., USDC). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the pool. These asset-transferring routines work 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.

```
    /* @dev Transfer the amount of token in.  Rebalance the cash reserve if needed */
    function _transferIn(
        uint256 info,
        uint256 amountUnnormalized
    )
        internal
    {
        uint256 amountNormalized = amountUnnormalized.mul(_normalizeBalance(info));
        IERC20(address(info)).safeTransferFrom(
            msg.sender,
            address(this),
            amountUnnormalized
        );
        _totalBalance = _totalBalance.add(amountNormalized);

        // If there is saving ytoken, save the balance in _balance.
```

```
        if (_isYEnabled(info)) {
            uint256 tid = _getTID(info);
            /* Check rebalance if needed */
            uint256 cash = _getCashBalance(info);
            if (cash > cash.add(_yBalances[tid]).mul(2).div(10)) {
                _rebalanceReserve(info);
            }
        }
    }
```

Listing 3.13: SmoothyV1::_transferIn()

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. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `transfer()` or `transferFrom()` 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 SmoothyV1 for borrowing/lending. In fact, SmoothyV1 is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

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.

3.9 Accommodation of approve() Idiosyncrasies

- ID: PVE-009
- Severity: medium
- Likelihood: medium
- Impact: medium
- Target: SmoothyV1
- Category: Business Logics [9]
- CWE subcategory: N/A

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 `approve()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.14: USDT Token **Contract**

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. An example is shown below. It is in the `_rebalanceReserve()` routine that is designed to rebalance the pool assets. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
    function _rebalanceReserve(
        uint256 info
    )
        internal
    {
        uint256 pricePerShare;
        uint256 cashUnnormalized;
        uint256 yBalanceUnnormalized;
        (pricePerShare, cashUnnormalized, yBalanceUnnormalized) = _getBalanceDetail(info
            );
        uint256 tid = _getTID(info);

        // Update _totalBalance with interest
        _updateTotalBalanceWithNewYBlance(tid, yBalanceUnnormalized.mul(
            _normalizeBalance(info)));

        uint256 targetCash = yBalanceUnnormalized.add(cashUnnormalized).div(10);
        if (cashUnnormalized > targetCash) {
            uint256 depositAmount = cashUnnormalized.sub(targetCash);
            IERC20(address(info)).approve(_yTokenAddresses[tid], depositAmount);
            YERC20(_yTokenAddresses[tid]).deposit(depositAmount);
            _yBalances[tid] = yBalanceUnnormalized.add(depositAmount).mul(
                _normalizeBalance(info));
        } else {
            YERC20(_yTokenAddresses[tid]).withdraw(targetCash.sub(cashUnnormalized).mul(
                W_ONE).div(pricePerShare));
            _yBalances[tid] = yBalanceUnnormalized.sub(targetCash.sub(cashUnnormalized))
                .mul(_normalizeBalance(info));
        }
    }
```

Listing 3.15: SmoothyV1::_rebalanceReserve()

Note that the accommodation of the `approve()` idiosyncrasy is necessary to ensure a smooth rebalance. Otherwise, the rebalance attempt with inconsistent token contracts may always be reverted.

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been fixed by taking the suggested approach of applying `approve()` twice.

3.10 Possible Outdated Balance For Share Calculation

- ID: PVE-010
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: SmoothyV1
- Category: Business Logics [9]
- CWE subcategory: CWE-841 [6]

Description

Throughout the entire Smoothy Finance protocol, the SmoothyV1 contract performs the actual mint /redeem operations, calculates the pool share for mint/redeem, and distributes protocol revenue to pool share holders. All these operations rely on precise measurement and calculation of current pool assets (that is needed to calculate the actual amount during mint or redeem for example). We notice that the pool assets may be distributed in various yToken contracts. The contract exhibits a public function named rebalanceReserve() to allow anyone to pull latest assets from current yToken contracts. Specifically, rebalanceReserve() calls _getBalanceDetail() to get the latest assets from yToken (line 528).

```
    /*
     * @dev Given the token id and the amount to be deposited, mint lp token
     */
    function mint(
        uint256 bTokenIdx,
        uint256 bTokenAmount,
        uint256 lpTokenMintedMin
    )
        public
        nonReentrantAndUnpaused
    {
        uint256 lpTokenAmount = getMintAmount(bTokenIdx, bTokenAmount);
        require(
            lpTokenAmount >= lpTokenMintedMin,
            "lpToken minted should >= minimum lpToken asked"
        );

        _transferIn(_tokenInfos[bTokenIdx], bTokenAmount);
        _mint(msg.sender, lpTokenAmount);
        emit Mint(msg.sender, bTokenAmount, lpTokenAmount);
    }
```

Listing 3.16: SmoothyV1::mint()

To elaborate, we show above the mint() routine that calls getMintAmount() to compute the expected amount of pool tokens. We notice that the mint() operation may not get real-time full assets (see the code snippets below at lines 713 714): the calculation of asset balance is performed with a cached `_yBalances[]`. The use of outdated asset balances likely lead to inaccurate measurement of system-wide assets. Other affected operations include `redeem()`, `redeemByLpToken()` and `swap()`. We consider the freshness of these pool assets critical even though their guarantee may introduce additional gas cost.

```
    /*
     * @dev Given the token id and the amount to be deposited, return the amount of lp
         token
     */
    function getMintAmount(
        uint256 bTokenIdx,
        uint256 bTokenAmount
    )
        public
        view
        returns (uint256 lpTokenAmount)
    {
        require(bTokenAmount > 0, "Amount must be greater than 0");

        uint256 info = _tokenInfos[bTokenIdx];
        require(info != 0, "Backed token is not found!");

        // Obtain normalized balances
        uint256 bTokenAmountNormalized = bTokenAmount.mul(_normalizeBalance(info));
        uint256 totalBalance = _totalBalance;
        uint256 sTokenAmount = _getMintAmount(
            bTokenAmountNormalized,
            totalBalance,
            _getBalance(info),
            _getSoftWeight(info),
            _getHardWeight(info)
        );

        require(sTokenAmount <= bTokenAmountNormalized, "penalty should be positive");

        return sTokenAmount.mul(totalSupply()).div(totalBalance);
    }
```

Listing 3.17: SmoothyV1::getMintAmount()

Recommendation Ensure the freshness of pool assets. To mitigate possible gas cost, an alternative is to implement daily-based `rebalance` mechanism such that it dynamically rebalances the assets on a daily basis. With that, there is no need to always invoke gas-heavy `getPricePerFullShare()` routine before the calculation of the exact `_yBalances[]`.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the SmoothyV1 documentation and implementation. The audited system presents a unique innovation and makes the best use of `mStable/Curve` design. We are impressed by the overall design and solid implementation. 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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