SMART CONTRACT AUDIT REPORT

for

SorbettoFragola

Prepared By: Yiqun Chen

PeckShield

June 28, 2021

Document Properties

Client	Popsicle-Finance
Title	Smart Contract Audit Report
Target	SorbettoFragola
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	June 28, 2021	Yiqun Chen	Final Release
1.0-rc	June 25, 2021	Yiqun Chen	Release Candidate

Contact

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

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

Contents

1 Introduction

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

`SorbettoFragola` is a protocol gathering the funds from users together to provide liquidity to the certain pool in `Uniswap V3`. It then mints their `ERC20` tokens for the user as a proof of share of the whole liquidity. As a new feature updated in `Uniswap V3`, the price range should also be set when the user tries to provide liquidity. The `governance` in the `SorbettoStrategy` develops the strategy (e.g., set the price range) for investors, and it aims to manage the funds by the strategy so that all investors can earn yield from the pool together.

The basic information of SorbettoFragola is as follows:

Table 1.1: Basic Information of SorbettoFragola

Item	Description
Name	Popsicle-Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	June 28, 2021

In the following, we show the Git repository and the commit hash value used in this audit:

- https://github.com/Popsicle-Finance/SorbettoFragola (9eb0ab5)

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

- https://github.com/Popsicle-Finance/SorbettoFragola (1ad2110)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem 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).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.2: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

We perform the audit according to the following procedures:

- 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.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section ??)
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 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.

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.

PeckShield

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of `SorbettoFragola`. 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	4	■■■■
Low	1	■
Informational	1	■
Total	6	

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

Table 2.1: Key SorbettoFragola Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Suggested Use Of Safemath In init()	Coding Practices	Fixed
PVE-002	Informational	Two-Step Transfer Of Privileged Account Ownership	Coding Practices	Fixed
PVE-003	Low	Logic Error In _calcShare()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Fixed
PVE-005	Medium	Incorrect Amount Calculation In burnLiquidityShare()	Coding Practices	Confirmed
PVE-006	Medium	Reentrancy Risk In MultisigWallet::execute()	Security Features	Fixed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Suggested Use Of Safemath In init()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SorbettoFragola`
- Category: Coding Practices [7]
- CWE subcategory: CWE-190 [2]

Description

The `SorbettoFragola` protocol allows the `governance` to set the `tickLower` and `tickUpper` for investors when they try to provide the liquidity in the pool. With that, the protocol retrieves the `currentTick`, adds the `baseThreshold` to compute the `tickUpper`, and subtracts the `baseThreshold` to compute the `tickLower`. And the protocol uses the `tickRangeMultiplier` set by the `governance` to multiply the `tickSpacing` to get the `baseThreshold`.

During our analysis, we notice potential overflow and underflow issues in the above calculation. In the following, we list the related `init()` function.

```
    function init() external onlyGovernance {
        require(!finalized, "F");
        finalized = true;
        int24 baseThreshold = tickSpacing * ISorbettoStrategy(strategy).
            tickRangeMultiplier();
        (uint160 sqrtPriceX96, int24 currentTick, , , , , ) = pool03.slot0();
        int24 tickFloor = PoolVariables.floor(currentTick, tickSpacing);

        tickLower = tickFloor - baseThreshold;
        tickUpper = tickFloor + baseThreshold;
        universalMultiplier = PriceMath.token0ValuePrice(sqrtPriceX96,
            token0DecimalPower);
    }
```

Listing 3.1: `SorbettoFragola::init()`

The problem is introduced when the multiplication of `tickSpacing` and `tickRangeMultiplier` is larger than 24 bits, which leads to an overflow. Besides, when calculating `tickLower` and `tickUpper`, if the `tickFloor` is smaller than the `baseThreshold`, an underflow occurs. And if the sum of `tickFloor` and `baseThreshold` is large enough, an overflow occurs.

Recommendation Use `Safemath` for all the calculations in the above `init()` function.

Status This issue has addressed by the following commit: `6201770`.

3.2 Two-Step Transfer Of Privileged Account Ownership

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SorbettoStrategy`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1109 [1]

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. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

In the following, we list below the `acceptGovernance()` function.

```
    /**
     * @notice Governance address is not updated until the new governance
     * address has called `acceptGovernance()` to accept this responsibility.
     */
    function acceptGovernance() external {
        require(msg.sender == pendingGovernance, "PG");
        governance = msg.sender;
    }
```

Listing 3.2: `SorbettoStrategy::acceptGovernance()`

The current implementation provides a two-step transfer of the privileged account (`governance`). The first step initiates the `governance` update intent and the second step accepts and materializes the update. Both steps should be executed in two separate transactions. This is explicitly designed to prevent unintentional errors in the owner transfer process. However, there is no event emitted here to record the change of the `governance`. As an essential account for the protocol-level safety and operation, we suggest to update the changes with the event. Also, we suggest to set the `pendingGovernance` to `address(0)` once the transfer of the `governance` is done.

Recommendation Add the event to record the transfer of the governance and set the pendingGovernance to address(0) after the transfer done.

Status This issue has been addressed by the following commit: 6201770.

3.3 Logic Error In _calcShare()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: SorbettoFragola
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [8]

Description

The SorbettoFragola protocol gathers the funds from investors together to provide liquidity to the specific pool in Uniswap V3. Once the user calls the deposit() to transfer token0 and token1 to the pool, the SorbettoFragola smart contract will mint a certain amount of ERC20 tokens as a proof of share.

However, there is a logic error in the calculation of the share. In the following, we list below the _calcShare() function.

```
    // Calcs user share depending on deposited amounts
    function _calcShare(uint256 amount0Desired, uint256 amount1Desired)
        internal
        view
        returns (
            uint256 shares
        )
    {
        shares = amount0Desired.mul(universalMultiplier).unsafeDiv(token1DecimalPower).
        add(amount1Desired.mul(1e12)); // Mul(1e12) Recalculated to match precisions
    }
```

Listing 3.3: SorbettoFragola::_calcShare()

The protocol derives the price of the token0 from the pool in a certain time, and uses the fixed prices of the token0 and the token1 to calculate the value of them based on the amount provided by the user. In fact, the share is the sum of value of the token0 and the token1. The purpose of this design is to make sure users come with same amount of tokens receiving the same amount of the share at anytime. However, as the price of the token0 floats greatly, when the price of the token0 is higher than the fixed price set in the protocol, it is unfair for other users to deposit the same amount of token0 but get the same share.

Recommendation Calculate the share based on the liquidity from the user.

Status This issue has been addressed by the following commit: `e45e07f`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SorbettoFragola`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `SorbettoFragola` smart contract, there is a privileged `governance` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters).

In the following, we show the representative function potentially affected by the privilege of the untrusted `governance`.

```
    /// @inheritdoc ISorbettoFragola
    function withdraw(uint256 shares)
        external override nonReentrant checkDeviation
        updateVault(msg.sender) returns (uint256 amount0, uint256 amount1)
    {
        require(shares > 0, "S");


        (amount0, amount1) = pool03.burnLiquidityShare(tickLower, tickUpper,
                                 totalSupply(), shares,  msg.sender);

        // Burn shares
        _burn(msg.sender, shares);
        emit Withdraw(msg.sender, shares, amount0, amount1);
    }
```

Listing 3.4: `SorbettoFragola::withdraw()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `governance` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. Note that a compromised `governance` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `SorbettoFragola` design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

3.5 Incorrect Amount Calculation In burnLiquidityShare()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PoolActions`
- Category: Coding Practices [7]
- CWE subcategory: CWE-682 [5]

Description

The `SorbettoFragola` protocol allows users to withdraw their funds from the pool by calling the `withdraw()` function. The logic behind this implementation is to compute the total liquidity in the position set by the `governance` at the very beginning. Then, based on the percentage of the share to the total share owned by the user, it further derives the amount of the liquidity that belongs to the user. After this, the protocol removes this part of the liquidity from the pool, and transfers the tokens back to the user. The logic is valid, though there exists a calculation error in this process.

In the following, we list below the `burnLiquidityShare()` function.

```
    function burnLiquidityShare(
        IUniswapV3Pool pool,
        int24 tickLower,
        int24 tickUpper,
        uint256 totalSupply,
        uint256 share,
        address to
    ) internal returns (uint256 amount0, uint256 amount1) {
        require(totalSupply > 0, "TS");
        uint128 liquidityInPool = pool.positionLiquidity(tickLower, tickUpper);
        uint256 liquidity = uint256(liquidityInPool).mul(share) / totalSupply;


        if (liquidity > 0) {
            (amount0, amount1) = pool.burn(tickLower, tickUpper, liquidity.toUint128());

            if (amount0 > 0 || amount1 > 0) {
            // collect liquidity share
                (amount0, amount0) = pool.collect(
                    to,
```

```
46                          tickLower,
47                          tickUpper,
48                          amount0.toUint128(),
49                          amount1.toUint128()
50                      );
51                  }
52              }
53          }
```

Listing 3.5: `PoolActions::burnLiquidityShare()`

As we can see in the above function, when the contract calls the `collect()` function from the pool, the outputs are `amount0` and `amount0` (line 44). The `amount0` here represents the amount of `token0`. However, the purpose of the `collect()` function is to transfer the exact amount of the `token0` and the `token1` to the user, so the outputs here should be `amount0` and `amount1`.

Recommendation Revise the statement of `(amount0, amount0)= pool.collect()` to `(amount0, amount1)= pool.collect()`

Status This issue has been addressed by the following commit: `6201770`.

3.6 Reentrancy Risk In MultisigWallet::execute()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SorbettoFragola`
- Category: Coding Practices [7]
- 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` [12] exploit, and the recent `Uniswap/Lendf.Me` hack [11].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Note the `collectFees()` 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 (lines 503 – 504) starts before effecting the update on internal states (lines 511 – 512), 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 `collectFees()` function.

```
492     function collectFees(uint256 amount0, uint256 amount1) external updateVault(msg.
            sender) {
493         UserInfo storage user = userInfo[msg.sender];
494 
495         require(user.token0Rewards >= amount0, "A0");
496         require(user.token1Rewards >= amount1, "A1");
497 
498         uint256 balance0 = _balance0();
499         uint256 balance1 = _balance1();
500 
501         if (balance0 >= amount0 && balance1 >= amount1) {
502 
503             if (amount0 > 0) pay(token0, address(this), msg.sender, amount0);
504             if (amount1 > 0) pay(token1, address(this), msg.sender, amount1);
505         }
506         else {
507 
508             uint128 liquidity = pool03.liquidityForAmounts(amount0, amount1, tickLower,
                    tickUpper);
509             (amount0, amount1) = pool03.burnExactLiquidity(tickLower, tickUpper,
                    liquidity, msg.sender);
510         }
511         user.token0Rewards = user.token0Rewards.sub(amount0);
512         user.token1Rewards = user.token1Rewards.sub(amount1);
513         emit RewardPaid(msg.sender, amount0, amount1);
514     }
```

Listing 3.6: `SorbettoFragola::collectFees()`

Recommendation Add the `nonReentrant` modifier to prevent reentrancy.

Status The issue has been addressed by the following commit: `6201770`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of `SorbettoFragola`. The audited protocol gathers the funds from investors together to provide the liquidity to the specific `Uniswap V3` pool with their investment strategy. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, 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

PeckShield

References

[1] MITRE. CWE-1109: Use of Same Variable for Multiple Purposes. https://cwe.mitre.org/data/definitions/1109.html.

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

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

[8] MITRE. CWE CATEGORY: Logic Error. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

PeckShield