SMART CONTRACT AUDIT REPORT

for

Infusion

Prepared By: Xiaomi Huang

PeckShield

February 19, 2024

Document Properties

Client	Infusion
Title	Smart Contract Audit Report
Target	Infusion
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 19, 2024	Xuxian Jiang	Final Release
1.0-rc	February 17, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Infusion` 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 Infusion

`Infusion` is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from `Solidly/Velodrome` with a unique `AMM`. The DEX is compatible with all the standard features as popularized by `UniswapV2` with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Infusion

Item	Description
Name	Infusion
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	February 19, 2024

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

- https://github.com/Infusion-Finance/infusion-contracts.git (5b2b8d4)

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

- https://github.com/Infusion-Finance/infusion-contracts.git (3ece057)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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.

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

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

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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Infusion` protocol smart contracts. 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	3	■■■
Informational	0	
Total	5	

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 and 3 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Pair Initialization With Minimal Liquidity Enforcement	Time And State	Resolved
PVE-002	Low	Incorrect getAmountOut() Logic in InfusionLibrary	Business Logic	Resolved
PVE-003	Low	Improved _lockerFeesP Validation in Pair Construction	Coding Practices	Resolved
PVE-004	Low	Inconsistent K Invariants Between Pair And Router	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Beside the identified issues, 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 Pair Initialization With Minimal Liquidity Enforcement

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Pair`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

The `Infusion` protocol has the built-in curve ($x^3y + xy^3 = k$) for efficient stable swaps. While examining the k calculation, we notice a possible denial-of-service issue that may allow the first stable swap LP to brick the (new) pair.

In the following, we show the implementation of the related `_k()` routine, which basically computes the k value from the above curve. However, it comes to our attention that the internal variable `_a` (line 665) may yield 0, which effectively computes the return value to be 0 and cascadingly meets the k invariant maintained in the `swap()` routine. As a result, the first LP of a stable swap pair may abuse it to initialize the pair with $k = 0$ and then empty the pool by resetting `reserve0` and/or `reserve1` to be 0. With that, any later LP may find infeasible to add new liquidity to the pair, hence effectively bricking the pair.

```
661     function _k(uint x, uint y) internal view returns (uint) {
662         if (stable) {
663             uint _x = (x * 1e18) / decimals0;
664             uint _y = (y * 1e18) / decimals1;
665             uint _a = (_x * _y) / 1e18;
666             uint _b = ((_x * _x) / 1e18 + (_y * _y) / 1e18);
667             return (_a * _b) / 1e18; // x3y+y3x >= k
668         } else {
669             return x * y; // xy >= k
670         }
```

```
    }
```

Listing 3.1: `Pair::_k()`

Recommendation To fix the above issue, there is a need to ensure the initial k upon the stable swap pair initialization will be larger than `MINIMUM_LIQUIDITY*MINIMUM_LIQUIDITY`. An example revision is shown as below:

```
    function mint(address to) external lock returns (uint liquidity) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        uint _balance0 = IERC20(token0).balanceOf(address(this));
        uint _balance1 = IERC20(token1).balanceOf(address(this));
        uint _amount0 = _balance0 - _reserve0;
        uint _amount1 = _balance1 - _reserve1;

        uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
        if (_totalSupply == 0) {
            liquidity = Math.sqrt(_k(_amount0, _amount1)) - MINIMUM_LIQUIDITY;
            _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
        } else {
            liquidity = Math.min(
                (_amount0 * _totalSupply) / _reserve0,
                (_amount1 * _totalSupply) / _reserve1
            );
        }
        require(liquidity > 0, "ILM"); // Pair: INSUFFICIENT_LIQUIDITY_MINTED
        _mint(to, liquidity);

        _update(_balance0, _balance1, _reserve0, _reserve1);
        emit Mint(msg.sender, _amount0, _amount1);
    }
```

Listing 3.2: Revised `Pair::mint()`

Status This issue has been fixed in the following commit: `3ece057`.

3.2 Incorrect getAmountOut() Logic in InfusionLibrary

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `InfusionLibrary`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the user interaction, the `Infusion` protocol has an `InfusionLibrary` contract that provides a number of user convenience functions. While examining a specific `getAmountOut()` helper routine, we notice its implementation needs to be revised.

```
    function getAmountOut(
        uint amountIn,
        address tokenIn,
        address tokenOut,
        bool stable
    ) external view returns (uint) {
        (uint dec0, uint dec1, uint r0, uint r1, bool st, address t0, ) = IPair(
            router.pairFor(tokenIn, tokenOut, stable)
        ).metadata();
        return
            (_getAmountOut(amountIn, tokenIn, r0, r1, t0, dec0, dec1, st) *
                1e18) / amountIn;
    }
```

Listing 3.3: InfusionLibrary ::getAmountOut()

To elaborate, we show above the implementation of this `getAmountOut()` routine. As the name indicates, this routine is used to compute the `tokenOut` amount after swapping the given `amountIn` of `tokenIn`. However, the computed amount is equal to the expected `tokenOut` amount by scaling it to the input amount of `1e18`, not `amountIn`.

Recommendation Revise the above routine to compute the intended `tokenOut` amount based on the given `amountIn` of `tokenIn`.

Status This issue has been fixed in the following commit: `3ece057`.

3.3 Improved Validation on Protocol Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pair
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Infusion protocol is no exception. Specifically, if we examine the Pair contract, it has defined a number of protocol-wide risk parameters, such as _lockerFeesP and _feeDistributor. In the following, we show the corresponding constructor routine that initializes their values.

```
    constructor() {
        factory = msg.sender;
        (
            address _token0,
            address _token1,
            bool _stable,
            uint _lockerFeesP,
            address _feeDistributor
        ) = PairFactory(msg.sender).getInitializable();
        require(
            _lockerFeesP == 0
                (_lockerFeesP != 0 && _feeDistributor != address(0)),
            "MISS_FEE_DIST"
        );
        (token0, token1, stable, lockerFeesP, feeDistributor) = (
            _token0,
            _token1,
            _stable,
            _lockerFeesP,
            IFeeDistributor(_feeDistributor)
        );
        fees = address(new PairFees(_token0, _token1));
        ...
    }
```

Listing 3.4: Pair :: constructor()

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, we can improve the above constructor by further enforcing the following requirement: `require(_lockerFeesP < LOCKER_FEES_SCALE)`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. In the meantime, the above constructor in essence allows the first LP can decide `_lockerFeesP`, which may need to be revisited.

Status This issue has been fixed in the following commit: `3ece057`.

3.4 Inconsistent K Invariants Between Pair And Router

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Router`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned earlier, the `Infusion` protocol has a built-in curve, which is different from $xy = k$. While analyzing the k usages between `Pair` and `Router` routines, we notice certain inconsistency that needs to be resolved before deployment.

To elaborate, we show below the related `quoteLiquidity()` routine from the `Router` contract. While it always follows the traditional curve $xy = k$ to compute the output amount, it does not take into account the new curve when the given pair is a stable swap one.

```
    function quoteLiquidity(
        uint amountA,
        uint reserveA,
        uint reserveB
    ) internal pure returns (uint amountB) {
        require(amountA > 0, "Router: INSUFFICIENT_AMOUNT");
        require(reserveA > 0 && reserveB > 0, "Router: INSUFFICIENT_LIQUIDITY");
        amountB = (amountA * reserveB) / reserveA;
    }
```

Listing 3.5: `Router::quoteLiquidity()`

Recommendation Revise the above logic to properly compute the quote amount for both stable and volatile pairs.

Status This issue has been resolved as it intends to compute the normal quote for liquidity.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Infusion` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and pausing swaps). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setPauser(address _pauser) external {
        require(msg.sender == pauser);
        pendingPauser = _pauser;
    }

    function acceptPauser() external {
        require(msg.sender == pendingPauser);
        pauser = pendingPauser;
    }

    function setPause(bool _state) external {
        require(msg.sender == pauser);
        isPaused = _state;
    }

    function setFeeManager(address _feeManager) external {
        require(msg.sender == feeManager, "not fee manager");
        pendingFeeManager = _feeManager;
    }

    function acceptFeeManager() external {
        require(msg.sender == pendingFeeManager, "not pending fee manager");
        feeManager = pendingFeeManager;
    }

    function setFee(bool _stable, uint256 _fee) external {
        require(msg.sender == feeManager, "not fee manager");
        require(_fee <= MAX_FEE, "fee too high");
        require(_fee != 0, "fee must be nonzero");
        if (_stable) {
            stableFee = _fee;
        } else {
```

```
            volatileFee = _fee;
        }
    }
```

Listing 3.6: Example Privileged Functions in `PairFactory`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

Status This issue has been mitigated as the team makes use of a multisig to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Infusion` protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. It is in essence a DEX that is built starting from `Solidly/Velodrome` with a unique `AMM`. The DEX is compatible with all the standard features as popularized by `UniswapV2` with a number of novel improvements, including price oracles without upkeeps, a new curve ($x^3y + xy^3 = k$) for efficient stable swaps. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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

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