SMART CONTRACT AUDIT REPORT

for

StableV1Pair

Prepared By: Yiqun Chen

PeckShield

August 28, 2021

Document Properties

Client	StableV1Pair
Title	Smart Contract Audit Report
Target	StableV1Pair
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 28, 2021	Xuxian Jiang	Final Release

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 **4**

1.1 About StableV1Pair . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings **9**

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results **11**

3.1 Potential Reentrancy Risk in Liquidity Changes . . . 11

3.2 Race Condition Between approve() And transferFrom() . . . 12

3.3 Incompatibility with Fee-on-transfer Tokens . . . 13

4 Conclusion **16**

References **17**

1 Introduction

Given the opportunity to review the design document and related source code of the the `StableV1Pair` 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 is well engineered and 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 StableV1Pair

The `StableV1Pair` protocol is a fixed math approach to the stable swap implementation. With its focus on the stablecoin swaps, it is designed for extremely efficient stablecoin trading and low risk. When compared with `Curve`, it has no swap fee and allows for cheaper 1:1 swaps. With sufficient liquidity pools, `StableV1Pair`'s AMM algorithm is tailored specifically for stablecoins and equivalent wrapped tokens.

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	StableV1Pair
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 28, 2021

In the following, we show the repository of reviewed files and the MD5 checksum hash value used in this audit.

- URL: https://gist.github.com/andrecronje/3ea4e6b9d23b2b8967955f30ab8e0462
- MD5: 317a3a77cc34161110058198b254c11a

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 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

Likelihood

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.

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

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

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.4: 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 implementation of the `StableV1Pair` protocol. 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	1	■
Low	1	■
Informational	1	■
Total	3	

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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Audit Findings of StableV1Pair Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Reentrancy Risk in Liquidity Changes	Time and State	Fixed
PVE-002	Low	Race Condition Between approve() And transferFrom()	Time and State	Confirmed
PVE-003	Informational	Incompatibility with Fee-on-transfer Tokens	Business Logic	Confirmed

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 Potential Reentrancy Risk in Liquidity Changes

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: StableV1Pair
- Category: Time and State [7]
- CWE subcategory: CWE-682 [3]

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. Using the StableV1Pair as an example, the add_liquidity() function (see the code snippet below) is provided to externally call a token contract to transfer assets as liquidity. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (lines $103-104$) starts before effecting the update on internal states (lines $106-107$), 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 add_liquidity() function.

```
91      function add_liquidity(uint amount0, uint amount1, uint min_liquidity, address to)
            external returns (uint liquidity) {
92          (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
93          (uint _balance0, uint _balance1) = ((_reserve0+amount0), (_reserve1+amount1));
94
```

```
        if (totalSupply == 0) {
            liquidity = _lp(amount0/decimals0, amount1/decimals1);
        } else {
            liquidity = _lp(_balance0/decimals0, _balance1/decimals1) - _lp(_reserve0/
                decimals0, _reserve1/decimals1);
        }

        require(liquidity > min_liquidity, '< _min_liquidity');

        _safeTransferFrom(token0, msg.sender, address(this), amount0);
        _safeTransferFrom(token1, msg.sender, address(this), amount1);

        _mint(to, liquidity);
        _update(_balance0, _balance1);
    }
```

Listing 3.1: `StableV1Pair::add_liquidity()`

The same `re-entrancy` protection is also applicable to other routines, including `remove_liquidity()` and `exchange()`.

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

Status The issue has been fixed by adding the suggested `re-entrancy` protection.

3.2 Race Condition Between approve() And transferFrom()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StableV1Pair`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

The current `StableV1Pair` is also an ERC20-compliant pool token whose implementation has a known race condition issue between `approve()` and `transferFrom()` [1]. Specifically, when a user intends to reduce the spending allowance amount previously approved from, say, 10 `sAMM` to 1 `sAMM`. The user may race to spend all of the previously approved allowance (the 10 `sAMM`) and then additionally spend the new allowance amount just approved (1 `sAMM`). This breaks the user's intention of restricting the spending allowance to the new amount, **not** the sum of old amount and new amount.

```
    function approve(address spender, uint amount) external returns (bool) {
        allowance[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
```

```
        return true;
    }
```

Listing 3.2: `StableV1Pair::approve()`

```
    function transferFrom(address src, address dst, uint amount) external returns (bool)
         {
        address spender = msg.sender;
        uint spenderAllowance = allowance[src][spender];

        if (spender != src && spenderAllowance != type(uint).max) {
            uint newAllowance = spenderAllowance - amount;
            allowance[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }
```

Listing 3.3: `StableV1Pair::transferFrom()`

In order to properly approve the spending allowance, there also exists a known workaround: users can utilize the non-standard `increaseAllowance()` and `decreaseAllowance()` functions.

Recommendation Add the suggested workaround functions `increaseAllowance()` and `decreaseAllowance()` to mitigate the well-known issues around setting allowances. However, considering the difficulty and possible lean gains in exploiting the race condition, we also think it is reasonable to leave it as is.

Status This issue has been confirmed.

3.3 Incompatibility with Fee-on-transfer Tokens

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StableV1Pair`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `StableV1Pair` contract is designed to be the main entry for interaction with trading users. In particular, one entry routine, i.e., `add_liquidity()`, accepts user deposits of supported assets (e.g., `DAI`) as the liquidity. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `StableV1Pair` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function add_liquidity(uint amount0, uint amount1, uint min_liquidity, address to)
        external returns (uint liquidity) {
        (uint _reserve0, uint _reserve1) = (reserve0, reserve1);
        (uint _balance0, uint _balance1) = ((_reserve0+amount0), (_reserve1+amount1));

        if (totalSupply == 0) {
            liquidity = _lp(amount0/decimals0, amount1/decimals1);
        } else {
            liquidity = _lp(_balance0/decimals0, _balance1/decimals1) - _lp(_reserve0/
                decimals0, _reserve1/decimals1);
        }

        require(liquidity > min_liquidity, '< _min_liquidity');

        _safeTransferFrom(token0, msg.sender, address(this), amount0);
        _safeTransferFrom(token1, msg.sender, address(this), amount1);

        _mint(to, liquidity);
        _update(_balance0, _balance1);
    }
```

Listing 3.4: `StableV1Pair::add_liquidity()`

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. Therefore, these operations 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 contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `StableV1Pair` protocol, which is a fixed math approach to the stable swap implementation. The current code base is well organized and 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] HaleTom. Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack. https://github.com/ethereum/EIPs/issues/738.

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

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

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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