HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: WT WORLDWIDE TECH PTE. LTD
Date: November 16th, 2022

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for WT WORLDWIDE TECH PTE. LTD
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 tokens; Vesting; Airdrop; Staking
Platform	EVM
Language	Solidity
Methodology	Link
Changelog	28.10.2022 - Initial Review 14.11.2022 - Second Review 16.11.2022 - Third Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by WT WORLDWIDE TECH PTE. LTD (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project is smart contracts in the repository:

Initial review scope

<table>
<tr><td><b>Repository</b></td><td>https://github.com/onlystables/smart-contracts</td></tr>
<tr><td><b>Commit</b></td><td>8c9679d5bd2dde8b3878922a3d9ac55386cc3c78</td></tr>
<tr><td><b>Functional Requirements</b></td><td>Are not provided</td></tr>
<tr><td><b>Technical Requirements</b></td><td>README.md</td></tr>
<tr><td colspan="2"><b>Contracts:</b>

File: ./contracts/access/OracleManaged.sol
SHA3: 777d19d93371096a6ae2d0d54d80baccf9cc9b816a02b0d57a99cf8b0bc5db9c

File: ./contracts/fees/IOnlyStablesFeeHandler.sol
SHA3: 7937fcc488cdf6feaafb5bf3aa53a0d7fa6b731ad8cc32a821876df551f8f4f1

File: ./contracts/fees/OnlyStablesFeeHandler.sol
SHA3: cbe8760a4d8fa4c131daa1c8d839a47e3630b26ded43605822620fb221568815

File: ./contracts/integrations/common/OnlyStablesLPIntegration.sol
SHA3: 4b92440f17aec09d4843cc865f148f5ffc94cacd663e1354611d38fa6232ed5e

File: ./contracts/integrations/curve/common/ICurve3CRVBasePool.sol
SHA3: 83e4218b4ed33b2a1d0d75b708d2e04158d23c9ee9a9a62a9c4bd53099c714b8

File: ./contracts/integrations/curve/common/ICurve3CRVDepositZap.sol
SHA3: 21728701792951655667 65efab35f07fa4ff95543927552e573984bf03afbebd

File: ./contracts/integrations/curve/common/ICurve3CRVGauge.sol
SHA3: 62c6ac022bb71b70674ab8afdde979f914bd8ceb672a8154263b76bf18b422a5

File: ./contracts/integrations/curve/common/ICurve3CRVMinter.sol
SHA3: 980e34ceca4cf6fd16138ae17157d59bf901e22ab38b6a0bc8fdb51bfb1a8780

File: ./contracts/integrations/curve/common/ICurve3CRVPool.sol
SHA3: 8816fdce1768957398853a5b8b65c6ddf30cdbfd03f0ce396da082c83388fda9

File: ./contracts/integrations/curve/common/ICurveSwap.sol
SHA3: 9bd045078dabf989050be3dcbf9f70d73875dceceeb95615011a065abdf00196

File: ./contracts/integrations/curve/common/
OnlyStables3CRVMetaPoolIntegration.sol
SHA3: cc8273719739720e4b99a46fc57f3542f0299845fbe0f1018d34dfecc845223d</td></tr>
</table>

```
File: ./contracts/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol
SHA3: 09c9e13e5f624c3553e4c6b15a6489617216fd9e434dd79922fff13209fe2ddd

File: ./contracts/integrations/curve/OnlyStablesUSDD3CRVIntegration.sol
SHA3: 975d54937f98a31a3fc9be74893e895a277b9f70c6c12298de5521c384a9311d

File: ./contracts/staking/common/OnlyStablesStaking.sol
SHA3: bdf428f1543ab2aaf3819d2abf7b8f6ff3a410dfb99a610276d86f2f5e9f898a

File: ./contracts/staking/OnlyStablesLiquidityMining.sol
SHA3: 22462164ef5cf6be86c0c1427aba78b979ce08be292579494625b9da9c198555

File: ./contracts/staking/OnlyStablesTokenStaking.sol
SHA3: 900d87b9b66bd1a9eede5c5d7d308a07ff1639aa540074d552c591f1f1a58084

File: ./contracts/token/IOnlyStables.sol
SHA3: d3668656f60086b4c4b4fc30c54fdb217e651a653410a335484dd6caa9f46370

File: ./contracts/token/IStakedOnlyStables.sol
SHA3: 13b3e9602972864d3a33f31ac215c83e1fc64c0f97718d88dc6e75c59ff8cd97

File: ./contracts/token/OnlyStables.sol
SHA3: 694a4a264c847b2e9032d62cd2c3cf66a9ab02133d27662d6a14b1f4cd65a880

File: ./contracts/token/StakedOnlyStables.sol
SHA3: 831fc33056cbd5297a48e62e39a2de8d6e0e8006f984ea04244733b97c4b6a55

File: ./contracts/vesting/OnlyStablesAirdrop.sol
SHA3: 1f3f7dc60763d51bd59f6f156368b2d75402c80a2c5639ccca464c11b89ea87c

File: ./contracts/vesting/OnlyStablesVesting.sol
SHA3: 89ad13a450ea9c38bebdd8f8bcb7bf4217459b0182fbf90b98e6068b8afe424f
```

Second review scope

Repository	https://github.com/onlystables/smart-contracts
Commit	743d48ced4f253ee5385cc175065ceb7d01c4a18
Functional Requirements	README.md
Technical Requirements	README.md

Contracts:

```
File: ./contracts/access/OracleManaged.sol
SHA3: 8cb4ccc29e283ec64602806bb8225c43396361f250514c9347a28aa61e41cdaa

File: ./contracts/fees/IOnlyStablesFeeHandler.sol
SHA3: 3d272799d42aa4152bf99888e70b4a8a184c30a4919ef2482b23fe085b700042

File: ./contracts/fees/OnlyStablesFeeHandler.sol
SHA3: 08f4773d0dfe4340cf8fba3074530e46a3d40a960e76a1154e646fab634c69ac

File: ./contracts/integrations/common/OnlyStablesLPIntegration.sol
SHA3: 244f43ff74267bc078bf0f84de611b90d89e3773ced8d3cc5277d50d6a599b42

File: ./contracts/integrations/curve/common/ICurve3CRVBasePool.sol
SHA3: a940a470bcd176aaf53dd01427e596940857035651c11f03296dc211bfba2e82

File: ./contracts/integrations/curve/common/ICurve3CRVDepositZap.sol
SHA3: d100b8271b22d3a147b113f94171b46f81d7f0dc1132fa4b508d956b2410883b

File: ./contracts/integrations/curve/common/ICurve3CRVGauge.sol
SHA3: a458bed0750f9f5a9f8671859fae907ab087765b9bca98c19505fd0faecd0345

File: ./contracts/integrations/curve/common/ICurve3CRVMinter.sol
SHA3: f2c26bf0fe24c4b44223b2c3b25dc46d3fee48390e77cdfc7cecdc0ddf8919bc

File: ./contracts/integrations/curve/common/ICurve3CRVPool.sol
SHA3: 96ba997faaa490411efc35faa6adb39764f05d0c2dc4afba7e5909b497047983

File: ./contracts/integrations/curve/common/ICurveSwap.sol
SHA3: 08251dea0de1686f0f3cb80a71c8c38f5eea28bfebb2d8ba28b57227a3bc05c0

File: ./contracts/integrations/curve/common/
OnlyStables3CRVMetaPoolIntegration.sol
SHA3: a387fefecddcb7ccf9cce7c38ed246b0fdf0a89e3bc9e4707cd691759954eacf

File: ./contracts/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol
SHA3: f9c64f41ad01495f0780882f530d5c48e1db39c224dca3f5d900d020606b2fb4

File: ./contracts/integrations/curve/OnlyStablesUSDD3CRVIntegration.sol
SHA3: cd381ef298a0fba4154569b98594eb6d575e5be424befe59d778aab8af8066f8

File: ./contracts/staking/common/OnlyStablesStaking.sol
SHA3: 154257aae76b6163149761a68e9c16a856f2fd102c4f23c1feb5f44c93904aec

File: ./contracts/staking/OnlyStablesLiquidityMining.sol
SHA3: 49918ca71e4287dcd4443381a40b2630639f6c66fd8e092e4a644973858a3a8c
```

```
File: ./contracts/staking/OnlyStablesTokenStaking.sol
SHA3: 62b64ddb477f5e0599448291cacf2872aa251ab63550aadf001da2f3fe760d98

File: ./contracts/token/OnlyStables.sol
SHA3: 694a4a264c847b2e9032d62cd2c3cf66a9ab02133d27662d6a14b1f4cd65a880

File: ./contracts/token/StakedOnlyStables.sol
SHA3: 838d1d4592501c92920e71a1ff764e785250223ef5edaddf9bf51a0d3bc68547

File: ./contracts/vesting/OnlyStablesAirdrop.sol
SHA3: 26d40caf24a48835759a0e1020520d012dd66e070cebe3fd4eb2149e672a959b

File: ./contracts/vesting/OnlyStablesVesting.sol
SHA3: 00aea80c3c7f373848b14bbe62e79637e6b54d015839d460a389b4e6fb2def1a
```

Third review scope

Repository	https://github.com/onlystables/smart-contracts
Commit	f7cefdc5eba38ce16b03d1a2f97e7e595cb54755
Functional Requirements	README.md
Technical Requirements	README.md

Contracts:

```
File: ./contracts/access/OracleManaged.sol
SHA3: 8cb4ccc29e283ec64602806bb8225c43396361f250514c9347a28aa61e41cdaa

File: ./contracts/fees/IOnlyStablesFeeHandler.sol
SHA3: 3d272799d42aa4152bf99888e70b4a8a184c30a4919ef2482b23fe085b700042

File: ./contracts/fees/OnlyStablesFeeHandler.sol
SHA3: 08f4773d0dfe4340cf8fba3074530e46a3d40a960e76a1154e646fab634c69ac

File: ./contracts/integrations/common/OnlyStablesLPIntegration.sol
SHA3: 742154f61d5355824f68567f4527f7a0631aabd5da5680b9efca365e2dc952af

File: ./contracts/integrations/curve/common/ICurve3CRVBasePool.sol
SHA3: a940a470bcd176aaf53dd01427e596940857035651c11f03296dc211bfba2e82

File: ./contracts/integrations/curve/common/ICurve3CRVDepositZap.sol
SHA3: d100b8271b22d3a147b113f94171b46f81d7f0dc1132fa4b508d956b2410883b

File: ./contracts/integrations/curve/common/ICurve3CRVGauge.sol
SHA3: a458bed0750f9f5a9f8671859fae907ab087765b9bca98c19505fd0faecd0345

File: ./contracts/integrations/curve/common/ICurve3CRVMinter.sol
SHA3: f2c26bf0fe24c4b44223b2c3b25dc46d3fee48390e77cdfc7cecdc0ddf8919bc

File: ./contracts/integrations/curve/common/ICurve3CRVPool.sol
SHA3: 96ba997faaa490411efc35faa6adb39764f05d0c2dc4afba7e5909b497047983

File: ./contracts/integrations/curve/common/ICurveSwap.sol
SHA3: 08251dea0de1686f0f3cb80a71c8c38f5eea28bfebb2d8ba28b57227a3bc05c0

File: ./contracts/integrations/curve/common/
OnlyStables3CRVMetaPoolIntegration.sol
SHA3: aa89fef7e5026100694f17929fdff7b4adbfa8a0dc61e6c7c1245292d1000fb5

File: ./contracts/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol
SHA3: f9c64f41ad01495f0780882f530d5c48e1db39c224dca3f5d900d020606b2fb4

File: ./contracts/integrations/curve/OnlyStablesUSDD3CRVIntegration.sol
SHA3: cd381ef298a0fba4154569b98594eb6d575e5be424befe59d778aab8af8066f8

File: ./contracts/staking/common/OnlyStablesStaking.sol
SHA3: 154257aae76b6163149761a68e9c16a856f2fd102c4f23c1feb5f44c93904aec

File: ./contracts/staking/OnlyStablesLiquidityMining.sol
SHA3: 49918ca71e4287dcd4443381a40b2630639f6c66fd8e092e4a644973858a3a8c

File: ./contracts/staking/OnlyStablesTokenStaking.sol
```

```
SHA3: 62b64ddb477f5e0599448291cacf2872aa251ab63550aadf001da2f3fe760d98

File: ./contracts/token/OnlyStables.sol
SHA3: 694a4a264c847b2e9032d62cd2c3cf66a9ab02133d27662d6a14b1f4cd65a880

File: ./contracts/token/StakedOnlyStables.sol
SHA3: 838d1d4592501c92920e71a1ff764e785250223ef5edaddf9bf51a0d3bc68547

File: ./contracts/vesting/OnlyStablesAirdrop.sol
SHA3: 26d40caf24a48835759a0e1020520d012dd66e070cebe3fd4eb2149e672a959b

File: ./contracts/vesting/OnlyStablesVesting.sol
SHA3: 88f26ce90e2fcaa196041119b4ffb784302f34979b578ce0696dff5b6ff8291e
```

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions.
Medium	Medium-level vulnerabilities are important to fix; however, they cannot lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Technical description is provided.
- Functional requirements are provided.
- NatSpec comments clearly describe system purposes.

Code quality

The total Code Quality score is **10** out of **10**.

- Code follows the best practices.
- The development environment is configured.

Test coverage

Test coverage of the project is **100%** (branch coverage).

Security score

As a result of the audit, the code contains **1** low severity issue. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **10**.

1
2
3
4
5
6
7
8
9
10
The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
28 October 2022	14	5	5	0
14 November 2022	3	1	1	0
16 November 2022	1	0	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and more specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Not Relevant
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed

Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery	Passed
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP standards violation	EIP	EIP standards should not be violated.	Passed
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Passed
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer.	Passed

Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style guide violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

System Overview

Worldwide OnlyStables is a mixed-purpose smart contract system implementing following features:

- *OracleManaged* — inheritable access managing contract. Functionality:
    - *onlyOracle* modifier — controls function is callable only by oracle
    - *onlyOwnerOrOracle* modifier — controls function is callable only by oracle or owner
    - *setOracle* — allows owner changing oracle address
- *OnlyStablesFeeHandler* (inherits *OracleManaged*) — fee handler. Functionality:
    - *processFee* — distributes its balance to staking and treasury contracts, partially burns tokens
    - setFee — allows owner setting fees (in total no more than 20%)
- *OnlyStablesLPIntegration* (inherits *OracleManaged*) — inheritable floating staking contract. Functionality:
    - *deposit* — allows depositing specified tokens
    - *withdraw* — allows withdrawing deposit
    - *claimRewards* — allows claiming staking rewards
    - *withdrawAfterShutdown* — allows withdrawing deposit after shutdown happens
    - *harvest* — allows owner or oracle to harvest rewards from internal farm
    - *handleFee* — allows owner or oracle to handle fee from internal farm staking rewards
- *OnlyStables3CRVMetaPoolIntegration* (inherits *OnlyStablesLPIntegration*) — implement interactions with CurveFi farm.
- *OnlyStablesPUSD3CRVIntegration* (inherits *OnlyStables3CRVMetaPoolIntegration*) — initialize the contract with PUSD/3CRV pair.
- *OnlyStablesUSDD3CRVIntegration* (inherits *OnlyStables3CRVMetaPoolIntegration*) — initialize the contract with USDD/3CRV pair.
- *OnlyStablesStaking* — inheritable solid staking contract. Functionality:
    - *deposit* — allows depositing specified tokens for specified period and APR
    - *withdraw* — allows withdrawing deposit after staking period finished
    - *claimRewards* — allows claiming rewards during staking period
- *OnlyStablesLiquidityMining* (inherits *OnlyStablesStaking*) — provides a calculating rewards mechanism based on corresponding pool reserves.

- *OnlyStablesTokenStaking* (inherits *OnlyStablesStaking*) - provides simple calculating rewards mechanism based on APR and period.
- *OnlyStables* - ERC20 burnable token:
  - name: "OnlyStables"
  - symbol: "OS"
  - once mints *100_000_000 * (10**decimals())* to deployer
- *StakedOnlyStables* - controlled ERC20 burnable token:
  - name: "stakedOnlyStables"
  - symbol: "sOS"
  - transfers from or to whitelisted users are allowed
  - whitelisted users are able to mint and burn tokens
  - owner is able to whitelist any user
- *OnlyStablesAirdrop* - simple vesting contract, vesting size depends on user deposit share. Functionality:
  - *deposit* - allows depositing tokens while deposit window is open
  - *withdraw* - allows withdrawing vesting token during vesting period
  - *startVestingPeriod* - allows owner to deposit vesting token and start vesting
- *OnlyStablesVesting* - simple vesting contract, owner specifies investors manually, supports TGE amount. Functionality:
  - *importData* - allows owner to import investors and their shares
  - *startVestingPeriod* - allows owner to start vestment period
  - *withdraw* - allows investor to receive funds during vesting period

Privileged roles

- Whitelisted users of *StakedOnlyStables* contract are able to mint/burn any amount of the tokens.
- System owner is able to handle fees from *OnlyStablesLPIntegration* contract.
- System owner is able to harvest rewards from the internal CurveFi farm.
- System owner is able to start airdrop/vesting sessions.

Risks

- **The admin of the contract may never start airdrop or vesting.** However, the deposited BRAIN coins are transferred directly to the treasury.
- The system highly relies on the Curve Finance liquidity pool farming system.
- The admin may set the LP price and affect the reward calculation during the liquidity mining reward calculation. It is necessary to make sure that the price is relevant.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

1. Front-Running Attack

The minimum return amount is not specified, or *0* is specified as a minimum value during operations with the router.

“Sandwich” attack is possible in such a case, which may lead to the loss of tokens by swapping them using the low rates.

Paths:
./contracts/fees/OnlyStablesFeeHandler.sol : _swapUSDTToOnlyStables()
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : _farmDeposit(), _farmWithdrawal(), _farmEmergencyWithdrawal(), _claimRewards(), _handleFee(), _swapMetaTokenTo3CRV(), _swapCRVTo3CRV()

Recommendation: use oracles to calculate the minimum amount expected to get tokens after swap.

Status: Fixed (second scope)

2. Flashloan Attack

The project has a staking contract, which calculates rewards based on the token share in the liquidity pool. It is possible to manipulate the calculated rewards by depositing flashloaned tokens to the pool.

This may significantly increase user rewards and lead to unexpected contract funds loss.

Path: ./contracts/fees/OnlyStablesLiquidityMining.sol : _swapUSDTToOnlyStables()

Recommendation: make the algorithm safe by disabling relying on data from UniSwap pool or use oracles to obtain actual value.

Status: Fixed (second scope)

3. Requirement Violation

According to documentation, the *_farmHarvest* function should harvest all possible rewards in an emergency situation. However, rewards are claimed only if claimable rewards amount and crv balance are bigger than corresponding thresholds.

This may lead to some rewards never being claimed.

Path:
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : _farmHarvest()

Recommendation: neglect thresholds on emergency withdrawal.

Status: Fixed (second scope)

4. Requirement Violation

According to documentation, users should be able to withdraw their deposits. However, it may not be possible if the user has transferred receipt tokens to another account or burned them.

This may lead to inability to withdraw the deposit.

Path: ./contracts/staking/OnlyStablesTokenStaking.sol : _takeReceipt()

Recommendation: mention that users should not transfer the tokens in public documentation or make the functionality safe.

Status: Fixed (second scope) & Mitigated (according to documentation, a user should provide receipt tokens to withdraw deposit)

5. Data Consistency

The staking contract has *stakingToken* and *rewardToken*. The Uniswap V2 token for *stakingToken* should have *rewardToken* as a pair. There are no statements which checks if the *stakingToken* has another token as a pair when *lpToken.token0()* is not equal to *rewardToken*.

This may lead to wrong staking rewards calculations.

Path: ./contracts/staking/OnlyStablesLiquidityMining.sol : _getLPValue

Recommendation: add check if *lpToken.token1()* equals to *rewardToken* before assigning *onlyStablesReserve*.

Status: Fixed (second scope) & Mitigated (oracle or admin may set the lp price)

6. Highly Permissive Role Access

During handling fees, after a shutdown situation happens, the owner may transfer an arbitrary amount of *LP_3CRV_TOKEN* to the treasury.

This may lead to a lack of funds to satisfy user rewards claiming and Denial of Service situations.

Path: ./contracts/integrations/common/OnlyStablesLPIntegration.sol : _handleFeeAfterShutdown()

Recommendation: reset the *totalUnhandledFee* value on fee handling.

Status: Fixed (third scope)

Medium

1. Best Practice Violation

It is considered following best practices to avoid unclear situations and prevent common attack vectors.

The functions do not use *SafeERC20* library for checking the result of ERC20 token transfer. Tokens may not follow ERC20 standard and return false in case of transfer failure or not returning any value at all.

This may lead to denial of service vulnerabilities during interactions with non-standard tokens.

Paths:
./contracts/vesting/OnlyStablesAirdrop.sol : startVestingPeriod()
./contracts/vesting/OnlyStablesVesting.sol : startVestingPeriod()
./contracts/staking/common/OnlyStablesStaking.sol : withdraw(), claimRewards()

Recommendation: follow common best practices, use *SafeERC20* library to interact with tokens safely.

Status: Fixed (second scope)

2. Inconsistent Data

It is considered to keep any data as accurately as possible until losses are quite small.

Critical state changes should emit events for tracking things off-chain.

The functions do not emit events on change of important values.

This may lead to inability for users to subscribe events and check what is going on with the project.

Paths:
./contracts/fees/OnlyStablesFeeHandler.sol : setFee()
./contracts/integrations/common/OnlyStablesLPIntegration.sol : setDepositThreshold()
./contract/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : setSwapThreshold()

Recommendation: keep data actual to the current system state, emit events on critical state changes.

Status: Fixed (second scope)

3. Inefficient Gas Model

It is considered to avoid inefficient Gas models.

The number of iterations of the loop in the function is uncontrolled as it depends on stored data.

This may lead to failures due to the block Gas limit.

Path: ./contracts/vesting/OnlyStablesVesting.sol : withdraw()

Recommendation: design the project to consume a limited amount of Gas regardless of the stored data and the number of users, provide page navigation through the data.

Status: Fixed (second scope)

4. **Contradiction**

It is considered that the project should be consistent and contain no self-contradictions.

The interface includes functions that could be declared as *view*.

This may lead to unexpected state changes.

Path: .contracts/fees/IOnlyStablesFeeHandler.sol : treasury, calculateFee

Recommendation: provide documentation, comments and identifiers in code consciously, declare the functions with a *view* modifier.

Status: Fixed (second scope)

5. **Unscalable Functionality**

It is considered smart contract systems should be easily scalable.

Custom implementation of *_beforeTokenTransfer* function should call *super._beforeTokenTransfer* function to keep the system consistent.

This may lead to new issues during further development, and functionality of the inherited contract is not applied.

Path: ./contracts/token/StakedOnlyStables.sol : _beforeTokenTransfer()

Recommendation: implement abstraction layers consciously, call the *super.function* at the start of custom implementation.

Status: Fixed (second scope)

6. **Inconsistent Data**

It is considered to keep any data as accurately as possible until losses are quite small.

The event *FeeHandlingThresholdUpdated* is emitted in the function, but provided data is not full.

This may lead to the wrong assumptions on the frontend about the current contract state.

Path: ./contracts/integrations/common/OnlyStablesLPIntegration.sol : setFeeHandlingThreshold()

Recommendation: use actual *feeHandlingThreshold* value to emit events.

Status: Fixed (third scope)

Low

1. Best Practice Violation

The Checks-Effects-Interactions pattern is violated. During the functions, some state variables are updated after the external calls.

This may lead to reentrancies, race conditions, and denial of service vulnerabilities during implementation of new functionality.

Paths:
./contracts/staking/common/OnlyStablesStaking.sol : initialize()
./contracts/integrations/common/OnlyStablesLPIntegration.sol: _deposit()

Recommendation: follow common best practices, implement function according to the Checks-Effects-Interactions pattern.

Status: Fixed (second scope)

2. Default Variable Visibility

The lack of variable visibility may cause unexpected variable visibility in derived contracts.

Paths:
./contracts/integrations/common/OnlyStablesLPIntegration.sol : rewardFactorAccuracy, users
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : CRV_META_POOL, CRV_GAUGE, CRV_DEPOSIT_ZAP, CRV_MINTER, CRV_SWAP, CRV_BASE_POOL, META_TOKEN, DAI_TOKEN, USDC_TOKEN, USDT_TOKEN, CRV_TOKEN, LP_3CRV_TOKEN, metaTokenSwapThreshold, crvSwapThreshold, stablecoinIndex
./contracts/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol : PUSD_3CRV_POOL, PUSD_3CRV_GAUGE
./contracts/integrations/curve/OnlyStablesUSDD3CRVIntegration.sol : USDD_3CRV_POOL, USDD_3CRV_GAUGE
./contracts/staking/common/OnlyStablesStaking.sol : userStakes
./contracts/staking/OnlyStablesTokenStaking.sol : receipt
./contracts/vesting/OnlyStablesAirdrop.sol : users
./contracts/vesting/OnlyStablesVesting.sol : users

Recommendation: specify the needed visibility during the variable initialization.

Status: Fixed (second scope)

3. Missing Zero Address Validation

Address parameters are being used without checking against the possibility of *0x0*.

Paths:
./contracts/fees/OnlyStablesFeeHandler.sol : constructor()

./contracts/integrations/common/OnlyStablesLPIntegration.sol : constructor()
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : constructor()
./contracts/staking/OnlyStablesStaking.sol : constructor()
./contracts/vesting/OnlyStablesAirdrop.sol : constructor()

Recommendation: add zero address validation.

Status: Fixed (second scope)

4. Functions that Could Be Declared External

public functions that are never called by the contract should be declared *external* to save Gas.

Path: ./contracts/integrations/common/OnlyStablesLPIntegration.sol : getLPBalance(), getReward()

Recommendation: use the *external* attribute for functions never called from the contract.

Status: Fixed (second scope)

5. Redundant Import Statement

Some contracts have redundant import statements, which are not used in the project.

Paths:
./contracts/staking/OnlyStablesLiquidityMining.sol : ReentrancyGuard, Ownable, IERC20, SafeERC20
./contracts/staking/OnlyStablesTokenStaking.sol : ReentrancyGuard, Ownable, IERC20, SafeERC20
./contracts/integrations/common/OnlyStablesLPIntegration.sol : Ownable
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : Ownable, ReentrancyGuard, IERC20Metadata

Recommendation: rewrite the contract logic to use all the imported contracts or remove the redundant statements.

Status: Fixed (second scope)

6. Floating Pragma

Contracts should be deployed with the same compiler version and flags that have been tested thoroughly. Locking the Pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Paths:
./contracts/fees/IOnlyStablesFeeHandler.sol
./contracts/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol
./contracts/integrations/curve/OnlyStablesUSDD3CRVIntegration.sol

./contracts/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol

Recommendation: use a fixed version of the compiler (^ symbol should be removed from Pragma).

Status: Fixed (second scope)

7. State Variables that Could Be Declared as Immutable

There are variables in the contract that can be declared as immutable to save Gas.

Paths:
./contracts/fees/OnlyStablesFeeHandler.sol : onlyStables, staking, treasury
./contracts/integrations/common/OnlyStablesLPIntegration.sol : rewardFactorAccuracy, feeHandler
./contracts/common/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : CRV_META_POOL, CRV_GAUGE, META_TOKEN
./contracts/staking/OnlyStablesStaking.sol : stakingToken, rewardToken, totalRewards, apr1Month, apr3Month, apr6Month, apr12Month
./contracts/staking/OnlyStablesTokenStaking.sol : receipt
./contracts/vesting/OnlyStablesAirdrop.sol : brains, treasury

Recommendation: declare variables that do not change as immutable.

Status: Fixed (second scope)

8. State Variables that Could Be Declared as Constant

There are variables in the contract that can be declared as constants to save Gas.

Paths:
./contracts/fees/OnlyStablesFeeHandler.sol : _router
./contracts/integrations/common/OnlyStablesLPIntegration.sol :, rewardFactorAccuracy
./contracts/common/integrations/curve/common/OnlyStables3CRVMetaPoolIntegration.sol : CRV_DEPOSIT_ZAP, CRV_MINTER, CRV_SWAP, CRV_BASE_POOL, DAI_TOKEN, USDC_TOKEN, USDT_TOKEN, CRV_TOKEN, LP_3CRV_TOKEN
./contracts/common/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol : PUSD_3CRV_POOL, PUSD_3CRV_GAUGE
./contracts/common/integrations/curve/OnlyStablesPUSD3CRVIntegration.sol : USDD_3CRV_POOL, USDD_3CRV_GAUGE

Recommendation: declare variables that do not change as constants.

Status: Fixed (second scope)

9. Missing Interface Inheritance

The project has contracts which implement interfaces but do not inherit them.

This increases the possibility of implementing the function with an interface violation which may lead to denial of service vulnerabilities.

Paths:
./contracts/token/StakedOnlyStables.sol
./contracts/token/OnlyStables.sol
./contracts/fees/OnlyStablesFeeHandler.sol

Recommendation: the contracts should inherit matching interfaces.

Status: Fixed (second scope)

10. Error Message Typo

The contract has a function with the *require* statements with the typo in the error message: *Depositing is now allowed at this time*.

This may confuse users during the interaction with the contract.

Path: ./contracts/integrations/common/OnlyStablesLPIntegration.sol : deposit()

Recommendation: update the error message by replacing *now* with *not*.

Status: Fixed (second scope)

11. Missing Constant

Consider putting the *1000* value to a special constant to be sure that APR is calculated correctly.

Paths:
./contracts/staking/OnlyStablesTokenStaking.sol : _calculateReward()
./contracts/staking/OnlyStablesLiquidityMining.sol : _calculateReward()

Recommendation: create a constant and use it to be the denominator of APR value.

Status: Fixed (second scope)

12. Redundant Code

The contract sets the mapping value to zero in the constructor, which is redundant because *0* is the default value for uninitialized variables.

This may lead to unnecessary Gas consumption.

Path:
./contracts/integrations/curve/OnlyStables3CRVMetaPoolIntegration.sol : constructor()

Recommendation: remove the assigning *stablecoinIndex[META_TOKEN]* to zero value.

Status: Fixed (second scope)

13. Code Duplications

The functions of the contract have the same *require* statements in two different functions.

Duplication of code may lead to unnecessary Gas consumption.

Path:
./contracts/staking/common/OnlyStablesStaking.sol :
enableDepositing(), disableDepositing()

Recommendation: consider moving the validation to a modifier or separate function.

Status: Fixed (second scope)

14. Inefficient Gas Model

It is considered to avoid inefficient Gas models.

The function returns many data at once, overloading the blockchain and network.

This may lead to failures due to the block Gas limit.

Path: ./contracts/staking/common/OnlyStablesStaking.sol : getUser()

Recommendation: design the project to consume a limited amount of Gas regardless of the stored data and the number of users, provide page navigation through the data.

Status: Fixed (second scope)

15. Redundant Statement

The contract has redundant variable statements. The importData function declares the *i* variable outside of the *for* loop and has a redundant *i* variable statement in the beginning of the loop.

Path: ./contracts/vesting/OnlyStablesVesting.sol : importData

Recommendation: remove the *i* variable statements from the beginning of the *for* loop.

Status: Fixed (third scope)

16. Missing Constant Usage

The contract has an unused constant. It looks like the constant should be used during calculations instead of the *100* value.

This may lead to redundant Gas usage during the contract deployment.

Path: ./contracts/vesting/OnlyStablesVesting.sol :
TGE_PERCENT_DENOMINATOR

Recommendation: rework the contract logic to use the state variable or remove the redundant state variable.

Status: Fixed (third scope)

17. Functions that Could Be Declared External

public functions that are never called by the contract should be declared *external* to save Gas.

Path: ./contracts/access/OracleManaged.sol : oracle()

Recommendation: use the *external* attribute for functions never called from the contract.

Status: Reported

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted to and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, Consultant cannot guarantee the explicit security of the audited smart contracts.