SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: VirtuSwap Foundation
Date: 05 Jun, 2023

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 VirtuSwap Foundation
Approved By	Marcin Ugarenko | Lead Solidity SC Auditor at Hacken OU
Type	ERC20; Vesting; Staking;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://virtuswap.io/
Changelog	08.05.2023 - Initial Review 05.06.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by VirtuSwap Foundation (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.

System Overview

VirtuSwap is a protocol with the following contracts:

- *The `VGlobalMinter` contract is responsible for minting and distributing VRSW and gVRSW tokens. It inherits from the Ownable contract and implements the IvGlobalMinter interface. The contract uses OpenZeppelin's SafeERC20 library for safe token transfers. This contract is deployed only once per whole system.*
- *The `VChainMinter` smart contract is responsible for distributing VRSW and gVRSW tokens to stakers. It implements the IvChainMinter interface and inherits from the Ownable contract. The contract uses OpenZeppelin's SafeERC20 library for safe token transfers. This contract can be deployed one per chain, and the whole system can have several instances of this contract (limited to one per chain).*
- *The `VVestingWallet` contract is a vesting wallet for ERC20 tokens. It allows the release of tokens to a beneficiary following a customizable vesting schedule. The contract is based on OpenZeppelin's VestingWallet contract.*
- *The `GVrsw` smart contract is an ERC20 token contract that is built using the OpenZeppelin library. The contract introduces a "minter" role, which is assigned to an address during the contract deployment. The minter has the ability to mint new tokens and send them to a specified address.*
- *The `VTokenomicsParams` smart contract is a simple contract that inherits from the IvTokenomicsParams interface and the OpenZeppelin Ownable contract. It is designed to store and update tokenomics parameters (r, b, alpha, beta, gamma) used in another contract's calculations. The contract is initialized with default values for these parameters, and they can be updated by the contract owner using the updateParams function.*
- *The `VStakerFactory` smart contract creates and manages instances of another smart contract called vStaker, which is used for staking VRSW tokens. The vStakerFactory contract is an implementation of the IvStakerFactory interface, which defines the functions that need to be implemented by the contract. The contract provides functions for*

getting the staker contract address for the VRSW pool and a specific LP token pool, creating a new staker contract for a given LP token, and setting the pending admin address. Only the admin can create new staker contracts and set the pending admin address. The admin can also accept the pending admin address to update the admin address.

- *The VStaker implements the IvStaker interface. It allows users to stake VRSW tokens and LP tokens, which are ERC20 tokens that represent liquidity pool shares. The staking rewards are distributed in VRSW tokens.*
- *The Vrsw is a smart contract that creates a new ERC20 token called "Vrsw" with the symbol "VRSW". It inherits from the OpenZeppelin ERC20 contract and includes a constructor that mints 1 billion tokens and assigns them to the address provided as the "_minter" argument. The decimals of the token are set to the default of 18.*

Privileged roles

- The owner of the *VTokenomicsParams* contract can update parameters of tokenomics (r, b, alpha, beta, gamma).
- The owner of the *vGlobalMinter* contract can mint new tokens for newly added *ChainMinter,* create new *VVestingWallet* contract, transfer *VRSW* tokens from contract, transfer *VRSW* tokens for new epoch and set parameters of epoch.
- The owner of the *VChainMinter* contract transfers VRSW tokens to contract for next epoch, updates stakeFactory address, changes minting epoch duration and preparation time and sets the allocation points for a list of stakers.
- During construction of *GVrsw* contract, sender of transaction gets minter role. Address with this role can mint new *Governance Virtuswap* tokens.

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**.

- Functional requirements are present in detail.
- Technical description is provided as NatSpec comments.

Code quality

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

- NatSpec covers the code in detail.

Test coverage

Code coverage of the project is **100.00%** (branch coverage).

Security score

As a result of the audit, the code contains no issues. 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**.

The system users should acknowledge all the risks summed up in the risks section of the report.

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
8 May 2023	9	4	3	1
5 June 2023	0	0	0	0

Risks

- The fund flow in the system is centralized, and many functions rely on the admin/owner role and need to be done manually. The staking rewards extracted from the `vGlobalMinter` contract are first deposited in the admin/owner role address and only then bridged to other chains to be deposited in each vChainMinter contract. If used incorrectly or when the admin/owner account is compromised, most of the VRSW token total supply will be lost.
- Owners can mint an unlimited number of gVRSW tokens using `addChainMinter()`. This centralization risk is driven by the fact that gVRSW tokens need to be deposited to each vChainMinter contract deployed on different chains. If used incorrectly the value of the gVRSW will be broken.
- The vTokenomicsParams can be updated by the owner using the `updateParams()` function, which may affect the tokenomics of the VRSW token.

Recommendations

- All admin/owner privilege role accounts should use multi-signature wallets with ⅗ signatures required to protect against the risks described in the Risk section.
- The test suite should be updated to cover all branches, include edge cases, and account for multi-user scenarios.

Checked Items

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

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	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	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Passed	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	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. EIP-712 should be followed during a signer verification.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Not Relevant	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Not Relevant	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	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.	Not Relevant	
Token Supply Manipulation	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	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	Style guides and best practices should be followed.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	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	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.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

C01. Invalid Calculations

Impact	High
Likelihood	High

In the *_availableTokens()* and *_availableTokensForNextEpoch()* functions, calculations are done incorrectly.

Both functions are not taking into account that calculations from the block.timestamp can be greater than the divisor in the function equations.

This can lead to a situation where the amount of tokens calculated is greater than the actual amount of rewards provided if the *prepareForNextEpoch()* function is not used correctly.

In the *_availableTokensForNextEpoch()* function, the *epochDuration* variable is used incorrectly in the dividend in case when the *nextEpochDuration > 0*.

Path: ./contracts/vChainMinter.sol : _availableTokens(), _availableTokensForNextEpoch()

Recommendation: Use the *min()* function from OZ Math library to limit the dividend of the equation to the value of the divisor.

In *_availableTokens()* function use *min((block.timestamp - startEpochTime), epochDuration)*.

In *_availableTokensForNextEpoch()* function use *min((block.timestamp - startEpochTime - _epochDuration), _epochDuration)*, where *_epochDuration = nextEpochDuration > 0 ? nextEpochDuration : epochDuration*.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

■■■ High

H01. Invalid Calculations

Impact	High
Likelihood	Medium

Calling the *setEpochParams()* function with *block.timestamp* which makes the check *block.timestamp >= startEpochTime + epochDuration* pass will cause a partial lock of the rewards paid when calling *nextEpochTransfer()*.

This is driven by the fact that _epochTransition() is triggered and param startEpochTime used for reward calculation is forwarded to a new timestamp.

Path: ./contracts/vGlobalMinter.sol : setEpochParams(), nextEpochTransfer()

Recommendation: Consider updating the *setEpochParams()* by removing the logic that triggers the *_epochTransition()*, as this internal function should only be called from the *nextEpochTransfer()* function.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

H02. Token Supply Manipulation

Impact	High
Likelihood	Medium

In the *newVesting()* and *arbitraryTransfer()* functions, the requirement to only release unlocked tokens is being executed incorrectly.

The *require(amount <= unlockedBalance)* statement only checks that the amount is less than the *unlockedBalance* variable, but both functions should also reduce the remaining unlocked token balance.

Lack of the *unlockedBalance* variable reduction can lead to a situation where more VRSW tokens are released than described in the tokenomy.

Path: ./contracts/vGlobalMinter.sol : newVesting(), arbitraryTransfer()

Recommendation: After each function call, reduce the *unlockedBalance* variable by the amount released.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

H03. Denial of Service; Fund Lock

Impact	High
Likelihood	Medium

In the *setStakerFactory()* function, the *stakerFactory* variable can be changed even when the current vStakerFactory has active vStaker contracts with deposited user funds.

In the event of changing the *stakerFactory* variable, users of the old vStaker contracts will not be able to claim their earned rewards. All calculations done in the vStaker contracts will be incorrect.

Additionally, the *setAllocationPoints()* function will be affected, as the *totalAllocationPoints* variable will contain the old allocation points from the old vStakerFactory vStaker contracts.

Changing the *stakeFactory* after it is set and in use will lead to an unusable staking system.

Path: ./contracts/vChainMinter.sol : setStakerFactory()

Recommendation: Prevent changing of the *stakerFactory* variable after it is set up.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

Medium

M01. Invalid Calculations

Impact	High
Likelihood	Low

In the *nextEpochTransfer()* function, there is a flaw in the logic of the epoch transition.

In case of calling the *nextEpochTransfer()* for the first time after the *emissionStartTs*, the rewards for epoch 0 will be locked inside the contract.

Path: ./contracts/vGlobalMinter.sol : nextEpochTransfer()

Recommendation: Consider updating the logic of the *nextEpochTransfer()* to trigger *_epochTransition()* after the release of the previous epoch's rewards.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

M02. Missing Validation

Impact	Medium
Likelihood	Medium

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

According to implementation, the value *beneficiary* should be different from the 0x0 address. However, in the functions, the validation is missed.

According to implementation, the value *startTs* should be greater than current time (block.timestamp). However, in the functions, the validation is missed.

According to implementation, the value *duration* should be different from 0. However, in the functions, the validation is missed.

According to implementation, the value *amount* should be different from 0. However, in the functions, the validation is missed.

This may lead to unexpected value processed by the contract.

Path: ./contracts/vVestingWallet.sol : newVesting()

Recommendation: Implement validations.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

M03. Non-Finalized Code

Impact	Medium
Likelihood	Medium

The code should not contain TODO comments. Otherwise, it means that the code is not finalized and additional changes will be introduced in the future.

Path: ./contracts/vStakerFactory.sol : createPoolStaker()

Recommendation: Remove TODO comments and resolve unfinalized codes.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

M04. Requirements Violation; Invalid Hardcoded Value

Impact	Medium
Likelihood	Medium

In the EmissionMath library, the constants V and v are declared. These variables are used in the calculation of the decreasing release schedule in the _calculateEmission() function.

In the requirement from the project whitepaper, it is stated that the amount released each year should be based on an annual percentage decrease of 20%.

With the current values, the first year release is around 20% of the initial amount of rewards, but all the consecutive years are around 16.84% decrements.

This leads to a situation where more tokens are released than expected.

Path: ./contracts/libraries/EmissionMath.sol : _calculateEmission()

Recommendation: Consider updating the values or documentation.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

Low

L01. Floating Pragma

Impact	Low
Likelihood	Low

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.

Path: ./contracts/ : *

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L02. Unscalable Functionality - Same Checks In Functions

Impact	Low
Likelihood	Medium

It is considered that smart contract systems should be easily scalable.

Same checks used in several functions overwhelm code and make further development difficult. Checks used multiple times:

- *require(block.timestamp >= emissionStartTs, 'too early');*
- *require(lockDuration > 0, 'insufficient lock duration');*
- *require(amount > 0, 'insufficient amount');*

- *require(lpToken != address(0), 'can stake only vrsw');*

This may lead to new issues during further development.

Path: ./contracts/vStaker.sol : *

Recommendation: Consider moving the checks to special modifiers.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L03. Missing Zero Address Validation

Impact	Medium
Likelihood	Low

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

This can lead to unwanted external calls to 0x0.

Paths:
./contracts/vStakerFactory.sol : constructor()
./contracts/vStaker.sol : unlockVrsw()
./contracts/vChainMinter.sol : constructor(), setStakerFactory(), transferRewards(), mintGVrsw(), burnGVrsw()
./contracts/GVrsw.sol : mint()

Recommendation: Implement zero address checks.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L04. Best Practice Violation

Impact	Low
Likelihood	Medium

The input arrays are not validated for having equal lengths.

This violates the best practices.

Path: ./contracts/vChainMinter.sol : setAllocationPoints()

Recommendation: Validate the input array lengths for the equality.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L05. State Variables Can Be Declared Immutable

Impact	Low
Likelihood	Low

Variable's *gVrsw* and *vrsw* values are only set in the constructor. Those variables can be declared as immutable.

This will lower Gas usage.

Path: ./contracts/vGlobalMinter.sol : gVrsw, vrsw

Recommendation: State variables can be declared immutable.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L06. Missing Validation

Impact	Low
Likelihood	Medium

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

Lack of validation of the *_emissionStartTs* argument in *vGlobalMinter.sol constructor()*. Emission should not start in the past. *_emissionStartTs* should be in the future.

This may lead to unexpected value processed by the contract.

Path: ./contracts/vGlobalMinter.sol : constructor()

Recommendation: Implement the validation.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L07. Unauthorized Access

Impact	Low
Likelihood	Low

The *release()* function can be called by anyone, allowing external users to release tokens on behalf of the beneficiary.

Path: ./contracts/vVestingWallet.sol : release(),

Recommendation: Consider restricting access to the *release()* function only to the beneficiary.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L08. Missing Events

Impact	Low
Likelihood	Low

Events for critical state changes should be emitted for tracking things off-chain.

Missing event inside *constructor()* of *vTokenomicsParams*, tokenomics parameters are updated and *UpdateTokenomicsParams* should be emitted like in *updateParams()*.

Path: ./contracts/vTokenomicsParams.sol : constructor(),

Recommendation: Add *UpdateTokenomicsParams* event inside *constructor()* of *vTokenomicsParams*.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

L09. NatSpec Comment Contradiction

Impact	Low
Likelihood	Low

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

The NatSpec comments of the *transferRewards()* imply that the caller must be a registered staker with a non-zero allocation point.

Actually, staker can have a non-zero allocation point, in case he had rewards before.

This may lead to wrong assumptions about the code's purpose.

Path: ./contracts/IvChainMinter.sol : transferRewards()

Recommendation: Fix the mismatch.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

Informational

I01. Solidity Style Guide Violation - Single Quotes

The provided projects should follow the official guidelines. The project violates the following style guidelines: use double quotes for strings.

Path: ./*

Recommendation: Replace single quotes with double quotes.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I02. Solidity Style Guide Violation - Contract Names

The name of the contracts and types should begin with uppercase letters. Using lowercase letters for types may confuse developers and lead to unintentional errors during further development.

Path: ./*

Recommendation: Consider using CapWords style for contract names.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I03. Indexed Inputs in Events

Events have the possibility to track their inputs as *indexed*. It is recommended to use the *indexed* keyword for better tracking of sensitive data.

Paths:
./contracts/interfaces/IvChainMinter.sol : TransferRewards
./contracts/interfaces/IvStaker.sol : StakeVrsw, StakeLp, RewardsClaimed, UnstakeLp, UnstakeVrsw, LockVrsw, LockStakedVrsw, UnlockVrsw

Recommendation: Consider adding the indexed keyword to track token addresses in events.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I04. Misleading Function Parameter Name

Function parameters should represent the function logic and should not mislead it.

Parameter *to* from *burnGVrsw()* is misleading. Function is burning gVrsw tokens from the provided address. It would be more suitable if this parameter were *from* instead of *to*.

This makes code harder to read.

Path: ./contracts/interfaces/IvChainMinter.sol : burnGVrsw()

Recommendation: Change function parameter name to fit the logic.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I05. Functions That Can Be Declared External

In order to save Gas, public functions that are never called in the contract should be declared as external.

Paths:
./contracts/GVrsw.sol : mint()
./contracts/vVestingWallet.sol : release()

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

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I06. Redundant Events

In order to save Gas, code should not have unused events. Events are declared in *IvGlobalMinter* and never used inside *vGlobalMinter*.

Path: ./contracts/interfaces/IvChainMinter.sol : NewStakerFactory, TransferRewards

Recommendation: Remove unused events.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I07. Unused Variable

Unused variables should be removed from the contracts.

Unused variables are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can cause an increase in computations (and unnecessary Gas consumption) and decrease readability.

The variable TOTAL_PROJECT_EMISSION is never used.

Path: ./contracts/liblaries/EmissionMath.sol : TOTAL_PROJECT_EMISSION

Recommendation: Remove unused variable.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I08. State Variables That Can Be Packed

Since the state variables in the *vChainMinter* contract *currentEpochBalance*, *nextEpochBalance*, *epochDuration*, *epochPreparationTime*, *nextEpochDuration*, *nextEpochPreparationTime*, *startEpochTime*, *startEpochSupply*, totalAllocationPoints represent mostly timestamps, they can be downcast and packed together in order to save Gas.

The state variables in the *vGlobalMinter* contract *startEpochTime*, *epochDuration*, *epochPreparationTime*, *nextEpochDuration*, *nextEpochPreparationTime* and *emissionStartTs* represent mostly timestamps, they can be downcast and packed together in order to save Gas.

The state variables in the *Stake* struct *startTs* and *lockDuration* represent timestamps, they can be downcast and packed together in order to save Gas.

Paths:
./contracts/vChainMinter.sol : *
./contracts/vGlobalMinter.sol : *
./contracts/types.sol : Stake

Recommendation: Consider downcasting the mentioned variables to smaller uint sizes and place them next to each other in order to pack storage.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I09. Redundant Import

Unused imports should be removed from the contracts.

Unused imports are allowed in Solidity and do not pose a direct security issue. It is best practice to avoid them as they can decrease readability.

The usage of *Math* is unnecessary for the *vStaker* and *vGlobalMinter* contracts.

The usage of *types* and *IvStaker* is unnecessary for the *vStakerFactory* contract.

Paths:
./contracts/vStaker.sol : Math.sol
./contracts/vGlobalMinter.sol : Math.sol
./contracts/vStakerFactory.sol : types.sol, IvStaker.sol

Recommendation: Remove the redundant import.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I10. Redundant Payable

Unused function's modifiers should be removed from the contracts. It is best practice to avoid them as they can decrease readability.

Constructor of *vVestingWallet* has a *payable* modifier, but this contract is not designed to receive native coins.

Path: ./contracts/vVestingWallet.sol : constructor()

Recommendation: Remove the redundant payable modifier.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I11. Style Guide Violation - Order of Functions

The project should follow the official code style guidelines.

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)
- external
- public
- internal
- private

Within a grouping, place the *view* and *pure* functions at the end.

Paths:
./contracts/vVestingWallet.sol
./contracts/vStakerFactory.sol
./contracts/stakeVrsw.sol
./contracts/vChainMinter.sol

Recommendation: The official Solidity style guidelines should be followed.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

I12. Missing Getters

Data from *allStakers* and *vestingWallets* can be accessed only by index.

It will be much easier to extract all data from an array using a single function call.

Paths:
./contracts/vStakerFactory.sol : allStakers
./contracts/vGlobalMinter.sol : vestingWallets

Recommendation: Create a getter function to extract whole arrays.

Found in: c23049f1

Status: Fixed (Revised commit: dfb861a)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing 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 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, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/Virtuswap/tokenomics
Commit	c23049f19e3faced5b39a3483e092712f05e1e53
Whitepaper	Not provided
Requirements	https://www.dropbox.com/s/psory6x4ymnuaom/Tokenomics-VirtuSwap-Apr-4-2023.pdf?dl=0
Technical Requirements	The documentation is present in form of NatSpec comments
Contracts	File: GVrsw.sol SHA3: 8862f6969293a2530440151fce2a74ef2a0f64b771630ea5ee809ac2da45ef75 File: types.sol SHA3: 5bd26fedf079f6809a1a7a85262b759737acfca6ec31ffea4672b608f0549196 File: vChainMinter.sol SHA3: b16aa46272c3f82e1151b601124bb8c245e15730403d9a8dda54f81f68004d32 File: vGlobalMinter.sol SHA3: a7a79d1d2e46b90636c7b4e6b5fc86af3666b56dad6cf1095effdcafb3e621af File: Vrsw.sol SHA3: 19cec13f28fb245faed3bcc6c54e9cdbe96c3f572385380c64ada46bf10b3e07 File: vStaker.sol SHA3: 87b543e37cf26353de37038e807ee850688e5c6386e34459153f81560481df8a File: vStakerFactory.sol SHA3: 54f747e6e1fd8f0fc06acb9816d1550b144e0746dcc943b21b70437f0867436a File: vTokenomicsParams.sol SHA3: f1539488d68b6a6b350971c5e148c239357c48cf49ffb2bb2ce8f600c4bfe5bd File: vVestingWallet.sol SHA3: 0f98d1608cc6ee8cde26a36a9814db0fa6b655c62a8b48d6b5bae3ee50828b83 File: interfaces/IvChainMinter.sol SHA3: d998e7ca31f8ddee05f30ca5aa834dc63f418f81902253f851177b2568556486 File: interfaces/IvGlobalMinter.sol SHA3: 5db50c76a126f87335c9f9c58f365f404107b4e8a44745def61c3a2badf86f01 File: interfaces/IvStaker.sol SHA3: d9e3888bc6c2951ecd8c1bd85425fddf666f04fcedd113813600e4ee351cf996 File: interfaces/IvStakerFactory.sol SHA3: 6307dea01e7fca090fb6b293f966de165cdf1a5ca50b30bce677d570c3adc0c7 File: interfaces/IvTokenomicsParams.sol SHA3: 1da85f9d457d5ab5f7fc41c4af39fcd6a6724a2272c0c1c7d1a22e0dd7b8061e

	File: libraries/EmissionMath.sol SHA3: d48707306ee79c7c17deeaa1e25b8d21a0a98c15e76d413a6842588909596649

Second review scope

Repository	https://github.com/Virtuswap/tokenomics
Commit	dfb861a7381c34d1bd3fd366326ed21201b5e388
Whitepaper	Not provided
Requirements	Tokenomics-VirtuSwap-May-24-2023.pdf
Technical Requirements	The documentation is present in the form of NatSpec comments.
Contracts	File: ./contracts/GVrsw.sol SHA3: 3e377bfa90805b61f539864b7adda70ac1f30902fc357d6e85df7dd92c2640ca File: ./contracts/Types.sol SHA3: 315d056d2f4c0515dee2f95c23825bfa35bca053ded45789d72840a80765e554 File: ./contracts/VChainMinter.sol SHA3: 2fbeb20b0ebd681ac838975625a0e1dee998672675427ffa082d9eb9e348e730 File: ./contracts/VGlobalMinter.sol SHA3: 62573f798ff76c1a51f74ff029288dda693005f2c1def2ceb2208bae6a4fb09c File: ./contracts/Vrsw.sol SHA3: 12890edd8edb2accc35bf50430f825353fd1036741c67feb06eba91e97194ef8 File: ./contracts/VStaker.sol SHA3: 17af05b13ffbd73c0212ef79dc3549dd26305d48652463abd108b68abcba7524 File: ./contracts/VStakerFactory.sol SHA3: 647591ca9d9cce73bb910401f1200df84aa99caeeb07e8b04965e4a3ab75bb9c File: ./contracts/VTokenomicsParams.sol SHA3: 8dbb591d0c0c47b88444fcb3eb0ac5c90f4fc47759800470ec6fe4181cb2c909 File: ./contracts/VVestingWallet.sol SHA3: e9644bf5036fe90625ea9f2e7cc1dbcce324533642583352dafaf536fa53d78d File: ./contracts/interfaces/IVChainMinter.sol SHA3: f2d23c4e7b3a8e527c41d78868d9d958d03fc92bef1af119cded089a9ba3b5a4 File: ./contracts/interfaces/IVGlobalMinter.sol SHA3: d08d95c0734f86365d603f406dfaa948180eeede4ab674f408a26344c839abf8 File: ./contracts/interfaces/IVStaker.sol SHA3: 91bf8cec3e5bec91ef47c8a99fadf0a68b796e0dccfdd90a7581bc31f7544ed6 File: ./contracts/interfaces/IVStakerFactory.sol SHA3: 4e4910b547c2fdd73961c345f419c77da055bfed1a5796afed94bd03898317fe File: ./contracts/interfaces/IVTokenomicsParams.sol SHA3: 43aa67c40062770d7f52e7ebfd59d42ecbadc0bc33024f1fb000949b54b8eb0a File: ./contracts/libraries/EmissionMath.sol

	SHA3: d125de1370ae0fecd9b84d53c5d8c0a2a107a52d67153bba901a46f96e655fc8