HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: DinoWars
Date: April 28, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for DinoWars
Approved By	Marcin Ugarenko | Lead Solidity SC Auditor at Hacken OÜ
Type	ERC20 token; Vesting
Platform	EVM
Language	Solidity
Methodology	Link
Website	dino-wars.com
Changelog	30.01.2023 - Initial Review 22.02.2023 - Second Review 17.03.2023 - Third Review 13.04.2023 - Fourth Review 28.04.2023 - Fifth Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by DinoWars (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 review and security analysis of smart contracts in the repository:

Initial review scope

Repository	https://github.com/QBergSolution/DinoWars-EconomySystem
Commit	a5279efb219affc7aac9690e94f44bc17b75c9a8
Whitepaper	whitepaper.dino-wars.com
Functional Requirements	Link
Contracts	File: ./DG.sol SHA3: c36df89b1660c3c493ed14f21148718d43cfe63e43f25afe3a3572e2b32dc44b File: ./DINW.sol SHA3: 31ffc8cd40c4c15b16b328144d39de13113bee3c77c49efe68641aeaf489bbc1

Second review scope

Repository	https://github.com/QBergSolution/DinoWars-EconomySystem
Commit	6e54bcf418b60c5768f6ff2f37a4752a15290605
Whitepaper	whitepaper.dino-wars.com
Functional Requirements	Link Contract methods description
Technical Requirements	Link
Contracts	File: ./contracts/DG.sol SHA3: 5aa62550a3aba76e998271d930fccecb2acdabb0c637ec121d8db686411d71bf File: ./contracts/DINW.sol SHA3: fe7d1561473691e61414dffb9ef35a0b5c497c9cc15f9e6bbfae0dbb0a4af475

Third review scope

Repository	https://github.com/QBergSolution/DinoWars-EconomySystem
Commit	ee8bbca4dcfc575b248fba5572d1ee61a6191668
Whitepaper	whitepaper.dino-wars.com
Functional Requirements	Link Contract methods description

Technical Requirements	Link
Contracts	File: ./contracts/DG.sol SHA3: c3bd3cdab59f96a1e7bb533709470ef4612101d63955edca5383a7a208812f93 File: ./contracts/DINW.sol SHA3: df14bf4ed037edca68a59181eb8225d6fec5131b87f83405c79463e5322a72cf

Fourth review scope

Repository	https://github.com/QBergSolution/DinoWars-EconomySystem
Commit	767a6cceec41fef03e78da1801515f12b2139be3
Whitepaper	whitepaper.dino-wars.com
Functional Requirements	Link Contract methods description
Technical Requirements	Link
Contracts	File: ./contracts/DG.sol SHA3: c3bd3cdab59f96a1e7bb533709470ef4612101d63955edca5383a7a208812f93 File: ./contracts/DINW.sol SHA3: 1f5aac545f4d849bc08822391199070b8b5213ca8cfd34171876e78b6c635e3f

Fifth review scope

Repository	https://github.com/QBergSolution/DinoWars-EconomySystem
Commit	735586b03a08d75bf6c7006e32c2d9b5c6338a17
Whitepaper	whitepaper.dino-wars.com
Functional Requirements	Link Contract methods description
Technical Requirements	Link
Contracts	File: ./contracts/DG.sol SHA3: 7b5a85b81fb318847a5bd5e62f5336ee0838febdd3685ec9aa752284a6f64f96 File: ./contracts/DINW.sol SHA3: f1bf3fefb921eb6bcd463a4d9f6b039a9294fa698fe23ff0370d7d45b717fd50

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
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 by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor gas optimization. These issues won't have a significant impact on code execution but affect code quality.

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 **7** out of **10**.

- Functional requirements are not finalized:
  - The “change user’s vesting plan” functionality is implemented but not mentioned in the documentation.
  - NatSpec comments for most of the code objects are incorrect and do not represent their purposes. Thus, *DG.docx* and *DINW.docx* files contain outdated/misleading information.
  - The Dino Wars. Documentation for economics file contains unimplemented blurry requirements:
    - *A separate pool of tokens for the game - divided into 4 wallets.*
    - *Commissions - covered by token users.*
    - *Share of owners - divided into 4 wallets.*
  - The Dino Wars. Documentation for economics file contains empty auto-generated fields and markdown formatting in the “Vesting” section. It is recommended to use native formatting in Google Documents.
  - The *README.md* file refers to an unfinalized non-English Копия requirements file.
  - In the *README.md* file *owner* and *user* as system actors are mixed.
  - A NatSpec comment for the *unlock* function refers to an inexistent *availLockedTokens* function (should be *availableLockedTokens*).
- Technical description is clear.

Correct functionality description could be found in the System Overview section of the report.

Code quality

The total Code Quality score is **8** out of **10**.

- Code is not formatted.
- Revert messages thrown in the same cases are designed in different ways:
  - *Invalid lock amount data* ~ *Invalid lock user data*
- Nondeclarative revert messages are found:
  - *Invalid data* - The data that is invalid is not defined.
- Code duplication is found.

- Dynamic length array is checked to have static length.
  Static length array usage would be more optimal there.
- A *uint40* variable is used as a counter. However, Solidity works faster operating 32-bytes length (*uint256*) types.
- Incorrect NatSpecs found.
- It is possible to set locked user balance below the unlocked one (by providing duplicates in *constructor(unlockWallets)* or providing low values in *lock(lockAmounts)*). This looks inconsistent and is not mentioned in the documentation.
- The development environment is configured.

Test coverage

Code coverage of the project is **56.25%** (branch coverage).

- Negative cases are not tested thoroughly.

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: **9.3**.

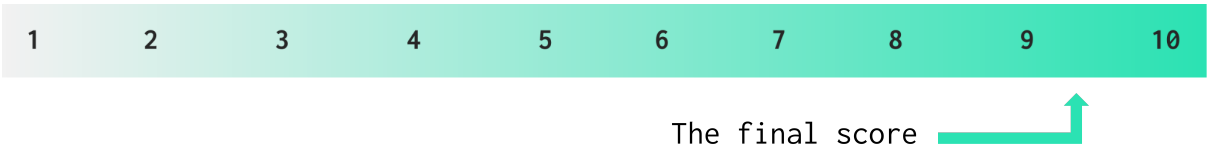

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
30 January 2023	7	6	4	1
22 February 2023	7	1	5	0
17 March 2023	5	3	0	1
13 April 2023	6	2	1	0
28 April 2023	1	0	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and 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.	Not Relevant
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Not Relevant
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.	Not Relevant
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Not Relevant
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	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	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.	Not Relevant
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Not Relevant
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 or be locked on the contract.	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.	Not Relevant

Token Supply Manipulation	**Custom**	Tokens can be minted only according to the 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.	Failed
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.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. The usage of contracts by multiple users should be tested.	Failed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

System Overview

DinoWars is a mobile crypto game project which contains the following smart contracts covered by the audit:

- *DG* – ERC20 token with the following properties:
  - Name: Dino game
  - Symbol: DG

  Additional features:
  - Ability to mint *DG* tokens (only specific role owners).
  - Ability to burn owned *DG* tokens (only specific role owners).
- *DINW* – ERC20 token with the following properties:
  - Name: DINO WARS
  - Symbol: DINW

  Additional features:
  - Tokens are initially locked in several wallets and are unlocked over 40 stages according to the tokenomics defined on contract deployment.
  - TGE for special wallets happens on deployment.
  - Ability to mint *DINW* tokens (only specific role owners).
  - Ability to burn owned *DINW* tokens (only specific role owners).
  - Ability to change/create/remove vesting plans (only specific role owners).

Privileged roles

DG – *DEFAULT_ADMIN_ROLE*:

- Set any other roles on the contract.

DG – *MINTER_ROLE*:

- Mint funds.
- Burn owned funds.

DINW – *DEFAULT_ADMIN_ROLE*:

- Set any other roles on the contract.

DINW – *MINTER_ROLE*:

- Change/Create/Remove user vesting plan.

Risks

- According to the documentation, the system contains exchange, staking, and lottery contracts, which are outside of the audit scope. The security of these systems should be ensured by an appropriate audit.
- Owners of *MINTER_ROLE* on the *DINW* contract may change users' vesting plans to arbitrary ones (for example, revoking vestments at all).
- There is no maximum total supply of *DG* and *DINW* tokens. Tokens could be infinitely inflated by the system owners.

Findings

■■■■ Critical

C01. EIP Standard Violation

According to the ERC20 standard, the *totalSupply* function should return the sum of all user balances. However, during the initialization, some user balances are set and the *_totalSupply* is not.

This may lead to unexpected assumptions for the total cap size.

The standard violation may lead to an integer underflow/overflow during calls to the *_burn*/*_mint* functions.

Path:
./DINW.sol: initialize()

Recommendation: process the *_totalSupply* variable value consciously.

Status: Fixed (second scope)

C02. Data Consistency & Token Supply Manipulation

The *lock* function implementation contradicts requirements.

The algorithm is corrupted:

- The *source* unlocked balance is not taken into account.
- The function mints tokens to the Gateway address and increases the *_unlocked_balances[Gateway]* variable value independently of the *_locked_balances[Gateway]* value.
- The function increases the *_locked_balances[target]* value in proportion to the *source* address vesting plan. However, the *_locked_balances[source]* value is not changed.
- The function emits an *Unlocked(source, amount)* event. However, nothing was unlocked for the *source* address.

This may lead to:

- Duplication of already withdrawn tokens from the *source* to the *target* address.
- Unexpected double minting (to the Gateway and the target address).
- Unexpected *_unlocked_balances[Gateway]* value.
- Unexpected wallet balances after execution (Gateway receives tokens, *target* receives locked tokens, *source* does not lose tokens).
- Wrong assumptions on user unlocked amount.

Path:
./contracts/DINW.sol: lock()

Recommendation: implement the algorithm according to the requirements.

Status: Fixed (fourth scope)

■■■ High

H01. Requirement Violation

According to the documentation, system owners of the contract should be able to mint/burn the asset. However, the functionality is not implemented.

Path:
./DINW.sol

Recommendation: make the documentation and code consistent with each other.

Status: Mitigated (on behalf of H05)

H02. Requirement Violation

According to the documentation, TGE amounts should be distributed at launch. However, the functionality is not implemented.

Instead, amounts that should be distributed after 1 month are distributed at launch and the total vesting length decreases to 39 months (a 40-month length is expected).

Path:
./DINW.sol

Recommendation: make the documentation and code consistent with each other.

Status: Fixed (second scope)

H03. Invalid Hardcoded Value

According to the documentation, the data stored in the _unlock mapping should be consistent with the project's Tokenomics.

However, it is not right for several cases:
`_unlock[REWARDS_WALLET][23] = 12973333` (should be `12973333e18`)
`_unlock[MARKETING_WALLET][20] = 1250000e18` (should be `1500000e18`)
`_unlock[MARKETING_WALLET][21] = 1375000e18` (should be `1500000e18`)
`_unlock[MARKETING_WALLET][30] = 1250000e18` (should be `1500000e18`)
`_unlock[MARKETING_WALLET][31] = 1375000e18` (should be `1500000e18`)

This may lead to a temporary DoS state for certain users due to an integer underflow in the `unlock` function.

Path:
./DINW.sol: _unlock, unlock()

Recommendation: copy the values to the code carefully or implement a corresponding calculation algorithm.

Status: Fixed (second scope)

H04. Highly Permissive Role Access

Owners should not have access to funds that belong to users.

In order to keep the funds flow clear any actions with other users' funds should be authorized with allowances.

Path:
./DG.sol: burn()

Recommendation: remove the ability to burn other users' funds, use an *ERC20Burnable* pattern from OpenZeppelin to implement burning funds using allowances.

Status: Fixed (second scope)

H05. Requirements Violation

The doc files are outdated / contain contradictory information.

- A lock for user functionality is implemented but not mentioned in the general overview.
- In the *DINW.docx* document the *lock* function description contains documentation for the *avail_locked_tokens* function.
- In the *DINW.docx* document it is stated that the *constructor* has no parameters. However, it has some.
- The general overview file states that the token name is "Dino game". However, "Dino Game" is implemented.
- The general overview file states that contract owners are able to mint/burn tokens. However, the functionality is not implemented.
- The NatSpec comment on the *lock* function does not describe what it should do.
- The technical description contains a screenshot of successfully running tests. However, some of the tests/contracts are not present in the repository.

Recommendation: update the documentation and resolve contradictions.

Status: Fixed (third scope)

H06. Requirement Violation

The *avail_locked_tokens* function relies on *unlock_plan[x] <= unlock_plan[x+1]*. However, the fact is not validated during the setup.

The *avail_locked_tokens* function relies on *_locked_balances[w][x] <= _locked_balances[w] [x+1]*. However, it is not checked during the setup.

This may lead to:

- the contract receiving a broken vesting plan
- users unable to receive locked funds

- *avail_locked_tokens* unexpected behavior

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

Recommendation: process input data carefully.

Status: Fixed (third scope)

H07. Data Consistency & Requirement Violation

The *_unlock* function relies on *_unlocked_balances[w] + amount <= _locked_balances[w][last_acceptable]*. However, the fact is not validated. The function may be called without pre-validation from the *lock* function.

This may lead to unexpected behavior of the *lock* and *unlock* functions.

Path:
./contracts/DINW.sol: _unlock(), lock(), unlock()

Recommendation: process input data carefully.

Status: Mitigated (on behalf of C02)

H08. Requirement Violation & Denial of Service

According to the documentation it should be possible for the owner to call the *setGateway* function. However, it is not.

The zero-check is broken as it checks the state variable, not the incoming parameter. As *_gateway* equals *0x0* by default, the function is inaccessible.

This may lead to an inability to use the *lock* function.

Path:
./contracts/DINW.sol: setGateway()

Recommendation: fix the zero-check.

Status: Fixed (third scope)

H09. Requirement Violation & Data Consistency

The *lock* function implementation contradicts requirements.

It is possible to withdraw most of the locked funds before the vesting period ends.

The initial tokens holder is able to unlock funds up to the current period and then transfer (using the *lock* function) the least funds to another address. There the funds are distributed through the vesting period and it is possible to unlock some funds up to the current period again. After a continuous number of iterations, most of the locked funds would be unlocked.

This may lead to users being able to unlock funds before the vesting period ends.

Path:
./contracts/DINW.sol: lock()

Recommendation: rework the logic, prevent *target* users from being able to unlock more funds than the *source* address is able to.

Status: Fixed (fifth scope)

Medium

M01. Best Practice Violation - Missing Initialization

According to the upgradable contracts pattern documentation, all inherited contracts should be initialized by a corresponding initializer. However, the *AccessControlUpgradeable* contract's initialization is missing.

Path:
./DG.sol: initialize()

Recommendation: invoke the *__AccessControl_init* method.

Status: Fixed (second scope)

M02. Best Practice Violation - Uninitialized Implementation

According to the upgradable contracts pattern documentation, it is recommended to disable the possibility of initialization on the logic contract.

It may be done by adding the constructor to the target code.

```
constructor() {
    _disableInitializers();
}
```

Paths:
./DG.sol
./DINW.sol

Recommendation: add the constructor to the target code to prevent logic contracts from being overtaken.

Status: Fixed (second scope)

M03. Contradiction - Documentation Mismatch

According to the documentation, tokens should be unlocked once a month. The month length is defined in the code as *30 days*. However, the average month length is *~30.437 days*.

This may lead to a full unlock 17 days earlier than expected.

Path:
./DINW.sol

Recommendation: consider the approximate month length in the documentation or implement more accurate calculations in the code.

Status: Fixed (second scope)

M04. Unscalable Functionality - Copy-Pasted Functionality

The contract contains copy-pasted functionality of the OpenZeppelin ERC20 contract.

This may lead to unexpected issues during further development (such as C01 and L03).

Path:
./DINW.sol

Recommendation: import the ERC20 contract from the source and inherit the target contract with it.

Status: Fixed (second scope)

M05. Contradiction - Documentation Mismatch

According to the documentation, the token Name property should be equal to “Dino game”. However, in the code, it is “Dino Game”.

Path:
./DG.sol: initialize()

Recommendation: update the documentation according to the code or implement the code according to the requirements.

Status: Mitigated (on behalf of H05)

M06. Contradiction - Documentation Mismatch

According to the documentation, users should be able to unlock distributed funds. However, there is no possibility for a user to get an available amount to unlock.

This may lead to high Gas waste while guessing the available unlock amount.

Path:
./DINW.sol: _balances, _locked_balances, _unlocked_balances

Recommendation: implement functionality to check the available unlock amount.

Status: Fixed (second scope)

M07. Best Practice Violation - Missing Event Emit

According to the documentation, a *GatewayChanged* event should be emitted on *_gateway* variable changes. However, the event is not emitted.

Path:
./contracts/DINW.sol: setGateway()

Recommendation: emit the event as required.

Status: Fixed (third scope)

M08. Contradiction - Documentation Mismatch

The function returns `0` for `timestamp < block.timestamp`. However, according to requirements, it should return the value available at the timestamp moment.

This may lead to users receiving incorrect data.

Path:
./contracts/DINW.sol: availableLockedTokens()

Recommendation: implement code according to the requirements or update documentation on the function purpose.

Status: Fixed (fourth scope)

M09. Best Practice Violation - Immutable Ownership

The contract is designed in a way that ownership cannot be transferred.

This may lead to the impossibility to update the owner in critical situations.

Path:
./contracts/DINW.sol

Recommendation: implement an ability to transfer the contract owner.

Status: Fixed (fourth scope)

M10. Contradiction - Invalid Calculations

In the `lock()` function, there is a possibility of an invalid calculation due to a rounding error.

As the `denominator` is rounded down, the fraction result (`lock_amount`) may be greater than expected.

This may lead to the `target` address receiving more funds than the `amount` value.

As the `lock_amount > amount` may happen, the statement `lock_amount + _unlocked[w] <= _locked[w][39]` may not be true and a Token Supply Manipulation issue may appear.

Path:
./contracts/DINW.sol: lock()

Recommendation: remove the *denominator* variable and calculate the fraction correctly (put all multiplications at the start and divisions at the end).

Status: Fixed (fifth scope)

M11. Contradiction - Inconsistent Data

During deployment it is possible to provide duplicates in the *unlockWallets* parameter. This may lead to the check being bypassed *_locked[w][0] < _unlocked[w]* as the *require* check implemented does not consider the *_unlocked* value.

This may lead to incorrect behavior of the *lock* function if a previously overfunded wallet is provided as the *target*.

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

Recommendation: consider *_unlocked* variable value during the “overfund” check.

Status: Fixed (fifth scope)

Low

L01. Unconscious Design

ERC20 token contracts are considered to have an unlimited life cycle. Vesting contracts with hardcoded receivers and a limited distribution amount are considered to have a limited life cycle.

Uniting the patterns may lead to additional Gas waste and obsolete dead code in a live contract.

Path:
./contracts/DINW.sol

Recommendation: separate the logic across different contracts or make the vesting functionality renewable.

Status: Mitigated (according to the requirements, the functionality should be implemented on the same contract)

L02. Floating Pragma

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.

The project uses floating pragmas *^0.8.17*.

Paths:
./contracts/DG.sol
./contracts/DINW.sol

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

Status: Fixed (third scope)

L03. 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:
./DG.sol: initialize()
./DINW.sol: initialize(), decimals(), decreaseAllowance()

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

Status: Fixed (second scope)

L04. Dead Code & Redundant State Variable

The functionality is redundant as it is never used.

Path:
./DINW.sol: _mint(), _burn(), contract_wallets

Recommendation: get rid of unused functionality or finalize it.

Status: Fixed (second scope)

L05. Redundant Code Block

There is no need to reserve storage slots in top-level contracts.

uint256[...] private __gap at the end of the DG and DINW is redundant.

Paths:
./DG.sol: __gap
./DINW.sol: __gap

Recommendation: get rid of the redundant functionality.

Status: Fixed (second scope)

L06. Explicit Size

Across the contracts, *uint* type is sometimes used for *uint256* variables.

Using *uint256* improves the readability and consistency of the code.

Mixing *uint256* and *uint* types makes code messy.

Paths:
./DINW.sol
./DG.sol

Recommendation: rename *uint* to *uint256*.

Status: Fixed (third scope)

L07. Style Guide Violation

The variables and functions are not implemented in the mixed case.

It is considered best practice to start *private* or *internal* objects with the "_" symbol and *external* or *public* objects with a lowercase letter.

Path:
./contracts/DINW.sol: unlock_plan, _locked_balances, _unlocked_balances, avail_locked_tokens(), _locked_balances(), _unlocked_balances(), unlock_plan(), constructor(lock_wallets, lock_amounts, unlock_wallets, unlock_amounts)

Recommendation: follow the official Solidity guidelines.

Status: Fixed (third scope)

L07-1. Style Guide Violation

The variables and functions are not implemented in the mixed case.

It is considered best practice to use *mixedCase* for variable names and start *private* or *internal* objects with the "_" symbol.

Path:
./contracts/DINW.sol: _locked_balances, _unlocked_balances, lock(lock_amount)

Recommendation: follow the official Solidity guidelines naming convention.

Status: Fixed (fifth scope)

L08. Missing zero value validation

The function accepts any value for the *amount* parameter. However, only a positive value is reasonable.

This may lead to the *DG* token being deployed with zero total supply.

Path:
./contracts/DG.sol: constructor(amount)

Recommendation: check if the mint amount is not *0*.

Status: Fixed (third scope)

L09. Redundant statement

The *require(uint40[40] == 40, ...)* statement is redundant as it is always *true*.

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

Recommendation: get rid of redundant statements.

Status: Fixed (third scope)

L10. Checks-Effects-Interactions Pattern Violation

During the call, some sanity checks are performed after storage writes.

This may lead to additional Gas waste on provided wrong input data.

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

Recommendation: move compare array lengths check to the start of the function.

Status: Fixed (third scope)

L11. Wrong NatSpecs

The NatSpec comments of the state variables are outdated. The variables are *private* so the *Returns ...* statements make no sense.

Path:
./contracts/DINW.sol: _locked_balances, _unlocked_balances

Recommendation: provide comments consciously.

Status: Mitigated (on behalf of Documentation Quality)

L12. Wrong Error Message

The revert message is *Incorrect caller*. However, a zero address check is performed there.

Path:
./contracts/DINW.sol: setGateway()

Recommendation: provide error messages consciously.

Status: Fixed (fourth scope)

L13. Redundant Return Parameter

The *amount* return parameter is declared, but it is never used in the code as the function ends with an explicit return statement.

Path:
./contracts/DINW.sol: availableLockedTokens(amount)

Recommendation: remove the redundant variable.

Status: Fixed (fifth scope)

L14. Second Lost

The function should return available funds for the wallet at the moment in time. However, a strict `timestamp > unlockPlan[t]` check is used, so the wrong value would be returned if the provided timestamp equals one of the `unlockPlan` elements.

Path:
./contracts/DINW.sol: availableLockedTokens()

Recommendation: make the comparison non-strict.

Status: Fixed (fifth scope)

L15. Wrong NatSpec

The NatSpec states that the function `emits an {Unlocked} event indicating the unlocked balance `amount` for `source``. However, it is not true.

Path:
./contracts/DINW.sol: lock()

Recommendation: remove the wrong statement.

Status: Mitigated (on behalf of Documentation Quality)

L14-01. Code Duplication

The pattern is implemented twice in the function.

```
if (_locked[w][t] < _unlocked[w]) return 0;
return _locked[w][t] - _unlocked[w];
```

Path:
./contracts/DINW.sol: availableLockedTokens()

Recommendation: make the `t > 0` comparison non-strict and remove the pattern duplication after the `for` cycle body.

Status: Reported

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.