HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Diverse Solutions
Date: 07 Jun, 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 Diverse Solutions
Approved By	Marcin Ugarenko | Lead Solidity SC Auditor at Hacken OU
Type	ERC20; Staking;
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.dsolutions.mn/
Changelog	24.04.2023 - Initial Review 09.05.2023 - Second Review 07.06.2023 - Third 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 Diverse Solutions (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 includes review and security analysis of the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/DiverseSolutions/ardmoney-staking-smart-contracts
Commit	f9d3dc20fb0b5b5cdf0f46803c9d0a622b260d17
Whitepaper	Not provided.
Functional Requirements	Not provided.
Technical Requirements	Not provided.
Contracts	File: ./contracts/XARDM.sol SHA3: 0a269e50ff58851c631a8f936a1c3726f680fba2c51816f95c26b1d417679168 File: ./contracts/XARDMStaking.sol SHA3: 3bb73e3519382e31c28631a428898764276dafc64e401741fbbc38a5fc4d9a6a

Second review scope

Repository	https://github.com/DiverseSolutions/ardmoney-staking-smart-contracts
Commit	9a1c12880986471875097f9b10835d5aed509bed
Whitepaper	Not provided.
Functional Requirements	Not provided.
Technical Requirements	Not provided.
Contracts	File: ./contracts/XARDM.sol SHA3: 9b275ced01c54674b0d22211fda43d0ebf8aa0cef65f3b733d28f3de4e4596a2 File: ./contracts/XARDMStaking.sol SHA3: 257256b7766bbf73a227b9371c9d20e31c058364f16c2190432a76eaf86a2454

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

Third review scope

Repository	https://github.com/DiverseSolutions/ardmoney-staking-smart-contracts
Commit	71b0b2e7d5aaa4db31652574cdb48f2081e045a2
Whitepaper	Not provided.
Functional Requirements	https://github.com/DiverseSolutions/ardmoney-staking-smart-contracts/blob/main/README.md
Technical Requirements	https://github.com/DiverseSolutions/ardmoney-staking-smart-contracts/blob/main/README.md
Contracts	File: ./contracts/XARDM.sol SHA3: 8307eb4645e6d56f72160f44ab6ce808e2ccb554feab11e1bfaeebdf85cef247 File: ./contracts/XARDMStaking.sol SHA3: aa8c2b125c2d18f79398c16f4f1c85dd07571683e9157826e8ee1553bb438c1f

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

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

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 detailed:
  - Description of the work of multiple deposits with commissions is not explained.
  - Project overview is detailed
  - All roles in the system are described.
  - NatSpec is presented.
- Technical description is inadequate:
  - Run instructions are not provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **7** out of **10**.

- The development environment is configured.
- Best practice violations.
- CEI pattern violation is found.

Test coverage

Code coverage of the project is **80.95%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is not fully covered.
- Test cases on some operations such as depositing several times are missing.

Security score

As a result of the audit, the code contains **3** high and **1** low severity issues. The security score is **0** 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: **2.1**

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

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

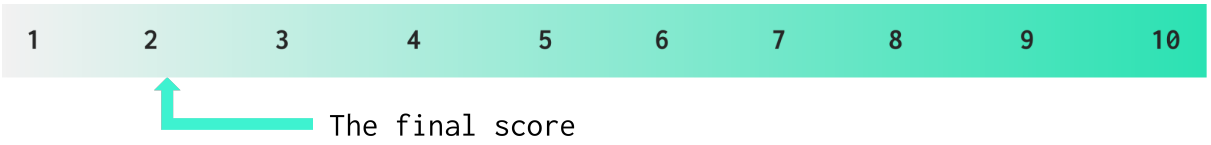

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
24 April 2023	9	2	3	3
08 May 2023	3	0	2	2
07 June 2023	1	0	3	0

Risks

- In addition to the XARDMStaking contract, the system owner has the ability to mint an unlimited number of XARDM tokens. This can lead to a **potential manipulation of the token price by affecting the token supply**.
- The smart contract highly depends on the smart contract owners, they can significantly affect the work and logic of the execution of the smart contract.
- The risk of users being awarded solely based on the deposit amount, without considering the duration, undermines the fairness of the reward distribution and enables manipulation of the penalty fee system, leading to minimal payments of penalty fees.

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

System Overview

Diverse Solutions is a staking pool that uses a modified AMM mechanism and the exchange rate between ARDM and XARDM is determined by the ratio of the total supply of XARDM to the total amount of ARDM held in the exchange contract. The system is explained via the following contracts:

- *XARDM* – an ERC-20 token that does not mint any supply during initialization. Additional minting is allowed and total supply is not capped.
  It has the following attributes:
  - Name: xArdMoney
  - Symbol: *XARDM*
  - Decimals: 18
  - Total supply: Infinite.
- *XARDMStaking* – a staking contract that allows users to deposit ARDM tokens. The staking system runs with the following logic:
  - Staker gets XARDM tokens in exchange for depositing ARDM. The XARDM amount to get = deposited ARDM amount * (total supply of xARDM / total ARDM in the contract)
  - Staker withdraws ARDM tokens by paying back the XARDM tokens.
    The ARDM amount to get = given XARDM amount * (total ARDM in the contract / total supply of xARDM)

Privileged roles

- The MINTER_ROLE of the XARDM contract can mint an arbitrary amount of tokens to any address.
- The DEFAULT_ADMIN_ROLE can grant PAUSER_ROLE or MINTER_ROLE to any user.
- The owner of the XARDMStaking contract can:
  - reset the rewards and withdraw deposited ARDM tokens that cross the ratio 1 of ARDM/XARDM.
  - set a penalty fee and a penalty deadline.
  - set a treasury address.
  - pause/unpause withdrawals or deposits.
  - pause getting a penalty fee.

Recommendations

- Add proper NatSpec documentation for the code.
- Consider merging XARDM and XARDMStaking into one contract, as both contracts are one system. Consider using the tokenized vault standard.
- Provide documentation for the system.

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

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.	Passed
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.	Failed
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

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

Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Failed
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 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.	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 or be locked on the contract.	Failed
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

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

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 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.	Failed
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. 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

Findings

■■■■ Critical

C01. Highly Permissive Role Access

The XARDM token contract has a *mint()* function that can be accessed by any account with *MINTER_ROLE* assigned, including the XARDMStaking contract and _adminAddress.

Any unauthorized minting of XARDM tokens outside of the XARDMStaking system compromises the integrity of user funds staked in the contract.

The presence of any minter role that is not a XARDMStaking contract in the staking system leads to a situation where user funds can be extracted from the contract directly, by minting any amount of XARDM tokens and withdrawing ARDM tokens from the XARDMStaking contract.

Path: ./contracts/XARDM.sol : mint()

Recommendation: only the staking contract should have the authority to mint XARDM tokens, and this should be immutable.

Found in: f9d3dc2

Status: Mitigated (Revised commit: 71b0b2e.

According to documentation provided by the client, the token minting business logic should be presented in the code:

"*xARDM token must have MINTER ROLE and only should point to 1 Staking Contract. IF in the future staking contract needs to be closed then minter role of that staking contract needs to be revoked and new staking contract needs to have minter role. Giving us full flexibility and migration abilities of xARDM token.*")

C02. Highly Permissive Role Access

The owner of the XARDMStaking contract can withdraw the users' deposited ARDM tokens by using the *resetRewards()* function.

When the total balance of ARDM in the XARDMStaking contract is greater than the total supply of xARDM, the owner can withdraw this difference as ARDM tokens.

Although this difference occurs due to an external ARDM transfer to the contract in the form of rewards, shares calculations of deposits made after the transfer will be calculated according to the new rate.

This leads to a situation in which any user who deposits ARDM tokens after the transfer of rewards can suffer losses after the owner calls for a rewards reset.

The owner should not be able to withdraw other users' deposited assets and should not be able to manipulate the profit they will make.

Path: ./contracts/XARDMStaking.sol : resetRewards()

Recommendation: do not reset the ratio and do not allow the owner to withdraw assets that belong to users.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

C03. Front-Running Attack; Inflation Attack

An inflation attack is an attack that allows malicious actors to steal the initial deposits into vulnerable pools, potentially resulting in significant losses for unsuspecting investors.

In the early stages, any contract that utilizes the 'mint shares' function in exchange for deposited assets is susceptible to an inflation attack.

The vulnerability is connected to a rounding issue in the *deposit()* function, as the following equation illustrates:

```
uint256 mintAmount = (_amount * totalxARDM) / totalARDM;
```

An attacker can manipulate the denominator, causing a victim to receive either zero or one share of the vault (XARDM).

At the beginning, when there are no funds in the pool, it is possible to use the front-running attack for instant profit.

Attack scenario:

1. An attacker sends the first deposit to the pool and mints one wei of share (XARDM): deposit(1). As a result, totalAsset() == 1, totalSupply() == 1.
2. An attacker front-runs the deposit of the victim who wants to deposit 20,000 ARDM.
3. An attacker inflates the denominator right in front of the victim: ardm.transfer(20_000e18). Now, totalAsset() == 20_000e18 + 1 and totalSupply() == 1.
4. The victim's transaction takes place. The victim gets 1 * 20_000e18 / (20_000e18 + 1) == 0 shares (XARDM), so the victim gets zero shares.
5. An attacker burns his share and gets all the ARDM.

Path: ./contracts/XARDMStaking.sol : deposit()

Recommendation: consider adding mitigation steps to the deposit() function. The attack vector and recommended mitigation steps are described under this link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3706#issuecomment-1297230505

Found in: f9d3dc2

Status: Fixed (Revised commit: 71b0b2e)

■■■ High

H01. Undocumented Behavior

The staking system is designed to exclusively allow externally owned accounts (EOA) to participate in staking.

Preventing contracts from participating in staking is not a desirable solution as it could limit the functionality and adoption of many applications, particularly in the DeFi space.

For example, Gnosis Safe addresses are created as contracts but can be used by multiple users as a shared wallet. In addition, it may block the possible interactions of other DeFi applications.

Path: ./contracts/XARDMStaking.sol : onlyEOA()

Recommendation: remove the EOA modifier and allow contract addresses to join the staking, or document this behavior and the reasoning behind it.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

H02. Highly Permissive Role Access

The owner of the XARDMStaking contract can change the penalty deadline and penalty fee values after users have deposited ARDM tokens under the previous penalty values.

Changing the penalty deadline and penalty fee will affect the users that have stakes in the system and will result in them paying different penalty fees than promised.

Path: ./contracts/XARDMStaking.sol : deposit()

Recommendation: consider applying a penalty deadline to user deposit directly inside the *deposit()* function, as shown below:

```
_userDeadline[msg.sender] = block.timestamp + penaltyDeadline;
```

Check the deadline in the withdraw() and hasUserDeadlinePassed() functions, simply by comparing:

```
_userDeadline[msg.sender] > block.timestamp
```

The penalty fee variable should be limited to reasonable amounts, for example 10%, when it is assigned in the *constructor()* or in the *setPenaltyFee()* function.

Found in: 128u923

Status: Reported (Revised commit: 71b0b2e.

The limitation on the fee size was not added, so there is no limit on how big it can be.)

H03. Undocumented Behaviour

The *deposit()* function always updates the *_userDeadline* variable to the current *block.timestamp* to track how much time has passed since the last deposit, and if the user needs to pay a penalty fee when withdrawing ARDM tokens.

However, the scenario of making multi-deposits is not considered as individual deposits and their timestamps are not tracked in the system.

Penalty fees are always calculated from the timestamp of the last deposit.

This creates inconsistency and causes users to pay unfair amounts of fees.

Path: ./contracts/XARDMStaking.sol : deposit(), withdraw()

Recommendation: explain the logic of this implementation in the documentation. If it is not the intended behavior of the system, fix the issue.

Found in: f9d3dc2

Status: Reported (Revised commit: 71b0b2e.

The issue is not fixed, multi-deposits are still not supported; applied fix created a new issue described in the H04.)

H04. Requirements Violation; Race Condition

Users' deadlines are not changed when they make a deposit, as long as the deadline for paying the penalty has not yet arrived, and the newly made deposits are recorded to be processed with the same deadline.

Users can wait until the last stage of the deadline by depositing a very small amount of tokens and then deposit the desired amount at the last minute to collect their rewards a few minutes later.

This race condition allows users to pay unfair penalty fees to the system and wait for less than the required period of time by manipulating the system.

Path: ./contracts/XARDMStaking.sol : deposit(), withdraw()

Recommendation: implement logic of multi-deposits, for taking fees from every deposit separately, instead of storing one deadline timestamp for all users' investment.

Found in: 71b0b2e

Status: New

Medium

M01. Unchecked Transfer

The *deposit()*, *withdraw()* and *resetRewards()* functions do not use the *SafeERC20* library for checking the result of *ERC20* token transfers.

Tokens may not follow the *ERC20* standard and return a false in case of transfer failure or not return any value at all.

Path: ./contracts/XARDMStaking.sol : deposit(), withdraw(), resetRewards()

Recommendation: use the SafeERC20 library to interact with tokens safely.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

M02. Highly Permissive Role Access

The account with *PAUSER_ROLE* can pause the transferability of the XARDM token.

This leads to a situation in which the *deposit()* and *withdraw()* functions of the XARDMStaking contract are affected by a Denial of Service vulnerability.

As both systems are tightly connected and there is functionality for pausing deposits and withdrawals directly in the XARDMStaking, the Pausable nature of the XARDM token appears redundant.

Path: ./contracts/XARDMStaking.sol : pause(), unpause()

Recommendation: consider removing the Pausable extension from the XARDM token contract, reduce the impact of privilege roles to a minimum.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

Low

L01. Gas Optimization: Redundant Use of SafeMath

Since Solidity v0.8.0, the overflow/underflow check is implemented via ABIEncoderV2 on the language level - it adds the validation to the bytecode during compilation.

There is no need to use the SafeMath library.

Path: ./contracts/XARDMStaking.sol

Recommendation: remove the SafeMath library.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L02. Switcher Functionality

Functions-switchers which reverse a value are not safe as they may be invoked by several users and the wanted result may not be obtained.

Race conditions and unexpected value can be assigned during the call.

Path: ./contracts/XARDMStaking.sol : toggleWithdrawPause(), toggleDepositPause(), togglePenaltyPause()

Recommendation: remove the switch functionality providing wanted status as a parameter.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L03. Gas Optimization: Unused Variable

The `penaltyToAddress` variable is declared but never used in the project.

Redundant declarations cause unnecessary Gas consumptions and reduce the code readability.

Path: ./contracts/XARDMStaking.sol

Recommendation: either implement the logic for the `penaltyToAddress` variable or remove it.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L04. Missing Zero Address Validation

Address parameters(treasuryAddress and ARDM) are used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Path: ./contracts/XARDMStaking.sol : constructor(), setTreasuryAddress()

Recommendation: implement zero address checks in the mentioned functions.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L05. Gas Optimization: Variables That Can Be Set as Immutable

The variables *ARDM* and *xARDM* are only set in the constructor and can thus be set as *immutable.*

Path: ./contracts/XARDMStaking.sol

Recommendation: it is recommended to set said variables as *immutable* in order to save Gas.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L06. Recommendation: 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.

Path: ./contracts/XARDMStaking.sol

Recommendation: consider adding the indexed keyword to track user addresses in events.

Found in: f9d3dc2

Status: Mitigated (The most important events have indexed parameters.)

L07. Gas Optimization: Unnecessary State Variable Update

The variables *withdrawPaused* and *depositPaused* are set to a *false* value in the smart contract *constructor()*, which is unnecessary since that is their default value.

This leads to unnecessary Gas consumption.

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

Recommendation: remove redundant state variables update.

Found in: f9d3dc2

Status: Fixed (Revised commit: 9a1c128)

L08. Recommendation: Boolean Equality

Boolean constants can be used directly and do not need to be compared to *true* or *false*.

Path: ./contracts/XARDMStaking.sol : deposit(), withdraw()

Recommendation: remove boolean equality.

Found in: f9d3dc2

Status: Fixed (Revised commit: 71b0b2e)

L09. Recommendation: Use of Hard-Coded Values

Hard-coded values are used in computations. The 1e20 and 1e18 values can be converted to constants to increase contract readability and reduce misuse.

Path: ./contracts/XARDMStaking.sol : withdraw(), getXARDMRate()

Recommendation: convert these variables into constants.

Found in: f9d3dc2

Status: Fixed (Revised commit: 71b0b2e)

L10. CEI Pattern Violation

The Checks-Effects-Interactions pattern is violated in several functions.

During the performing *withdraw* and *reward* function, totalARDM is updated after the external calls.

When performing the *deposit* function, first XARDM is sent to the user and then ARDM is taken by the user.

Path: ./contracts/XARDMStaking.sol : withdraw()

Recommendation: Update the state variable before transferring the tokens and always first receive the required tokens to be burned from the users and then transfer the rewards.

Found in: 71b0b2e

Status: New

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

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.