HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Imagine Replay
Date: 14 Aug, 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 Imagine Replay
Approved By	Oleksii Zaiats | SC Audits Head at Hacken OU Arda Usman | Lead Solidity SC Auditor at Hacken OU
Tags	ERC20 token
Platform	EVM
Language	Solidity
Methodology	Link
Website	Not provided
Changelog	25.07.2023 - Initial Review 07.08.2023 - Second Review 11.08.2023 - Third Review 14.08.2023 - Fourth Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Imagine Replay (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

ReplayToken is an ERC20 reward contract with the following contracts:

- *ReplayToken* – simple ERC-20 token that mints initial supply to a passed initDistrWallet_ address and allows to mint new tokens to reward the Subchain validator stakers.

Privileged roles

- PendingAdmin is an Intermediary role that is used for the update of admin.
- Admin is an address that is allowed to mint new tokens, change the pendingAdmin and change the value of rewards that will be shared between stakers.
- Minter roles is an address that is allowed to mint new staker rewards.

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 and technical requirements are provided.

Code quality

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

- The development environment and deployment instructions are sufficient.

Test coverage

Code coverage of the project is **80.56%** (branch coverage)

- The audit Lines of Code do not exceed 250, this does not affect the score.

Security score

As a result of the audit, the code contains **3** low severity 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.

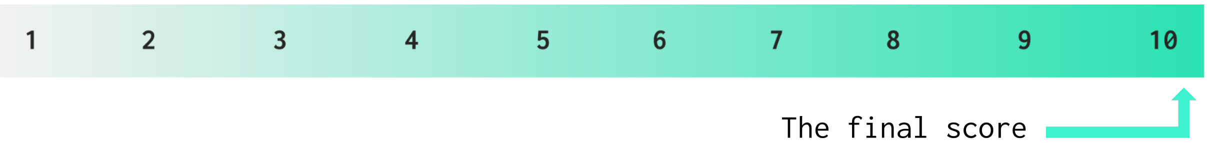

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
25 July 2023	5	1	0	0
07 August 2023	5	1	0	0
11 August 2023	3	0	0	0
14 August 2023	3	0	0	0

Risks

- The out-of-scope ValidatorStakeManager contract and admin are responsible for the minting of tokens. The secureness of the supply depends on the secureness of key storage. If an admin or a minter goes malicious, they will be able to mint all the token supply at once and prevent other users from receiving rewards.
- The total supply of the token is determined during the deployment. It cannot be verified until the contract is deployed.
- There is no strict restriction on when the admin can change the _stakerRewardPerBlock variable. They might set a smaller reward per block by calling the updateStakerRewardPerBlock() function. As a result, the stakers might receive fewer or no rewards, contrary to what they initially expected. It is recommended to use a timelock mechanism for such critical functionality.
- Since ReplayToken will be used as the Governance token, a malicious admin or minter will be able to manipulate the voting power.

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.	Failed	I05
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.	Not Relevant	
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.	Not Relevant	
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.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Not Relevant	
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.	Passed	
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. Contracts shouldn’t rely on values that can be changed in the same transaction.	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.	Failed	
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.	Not Relevant	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

No high severity issues were found.

■■ Medium

M01. Missing Event for Critical Value Updation

Impact	Medium
Likelihood	Medium

setPendingAdmin() does not have events.
Events for critical state changes should be emitted for tracking things off-chain.

This can lead to non-tracking minting tokens and setting a pending admin off-chain.

Paths: ./contracts/ReplayToken.sol: *setPendingAdmin()*

Recommendation: Add Emitting Events to *setPendingAdmin()*

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

■ Low

L01. Redundant Use of SafeMath

Impact	Low
Likelihood	Low

The library SafeMath is generally not needed starting with Solidity 0.8, since the compiler now has built-in overflow checking.

It would lead to not checking for under/overflow at all.

Path: ./contracts/ReplayToken.sol: *mintStakerReward(), mint(),*

Recommendation: Remove the SafeMath library.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

L02. Missing Zero Address Validation

Impact	Low
Likelihood	Low

updateMinter(), setPendingAdmin() functions do not have require() check for zero address validation.

This can lead to incorrect added address.

Paths: ./contracts/ReplayToken.sol: *updateMinter(), setPendingAdmin()*

Recommendation: Add require state (minter_ != address(0), "Error message") in *updateMinter()* function.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

L03. State Variables Can Be Declared Immutable or Constant

Impact	Low
Likelihood	Low

Compared to regular state variables, the Gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed in the ReplayToken contract, variables: _decimals, maxSupply.

Paths: ./contracts/ReplayToken.sol

Recommendation: Declare mentioned variables as immutable.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

L04. Public Variable Read in the External Context

Impact	Low
Likelihood	Low

The contract reads its own variable using *this.* keyword and *totalSupply()*, adding overhead of an unnecessary *STATICCALL*.

Path: ./contracts/ReplayToken.sol: *mintStakerReward()*

Recommendation: Read the variable directly from storage instead of calling the contract.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

L05. Copy Of Well-Known Contract

Impact	Low
Likelihood	Low

Well-known contracts from projects like provider should be imported directly from the source as the projects are in development and may update the contracts in the future.

Path: ./contracts/ReplayToken.sol

Recommendation: Import the Ownable2Step directly from the source, avoid modifying them.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

Informational

I01. Missing Event Indexes

The lack of indexed events makes it difficult for users to track the smart contract’s activity and increases overall Gas.

Path: ./contracts/ReplayToken.sol: UpdateAdmin, UpdateMinter

Recommendation: Use indexed events to keep track of a smart contract's activity after it is deployed, which is helpful in reducing overall Gas.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

I02. Disabled Solidity Optimizer

Disabled Solidity optimizer increases the overall Gas cost.

Path: ./contracts/ReplayToken.sol: truffle-config.js

Recommendation: Enable the Solidity compiler optimizer to minimize the size of the code and the cost of execution via inline operations, deployment costs, and function call costs.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

I03. Optimization by Replacing the require() Statements with Custom Errors

Path: ./contracts/ReplayToken.sol: *constructor(), mint(), adminOnly(), pendingAdminOnly(), minterOnly(), setPendingAdmin(), updateMinter()*

Recommendation: Replace the require() statements with custom errors.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

I04. Redundant Check

The *mintStakerReward()* function has the following redundant *if*, which consumes additional Gas.

```
if (currentSupply >= maxSupply) {
      return false;
  }
```

Path: ./contracts/ReplayToken.sol: mintStakerReward()

Recommendation: Delete the redundant *if* check.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

I05. Outdated Solidity Version

Using an outdated compiler version can be problematic, especially if publicly disclosed bugs and issues affect the current compiler version. Using an old version for deployment prevents access to new Solidity security checks.

Path: ./contracts/ReplayToken.sol
Recommendation: Deploy with any of the following Solidity versions: 0.8.18, 0.8.19, 0.8.20

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

I06. Events Are Missing Relevant Data

In the ReplayToken contract, the UpdateAdmin and UpdateMinter events do not log the previous state of the updated data. This may impede reconstructing the history of generation updates for an account through emitted events.

Path: ./contracts/ReplayToken.sol

Recommendation: Consider adding the previous values to the UpdateAdmin and UpdateMinter events.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Fixed (Revised commit: 0755d4bfb100169b3c1ab4353e9592d6e06a1530)

I07. Code Duplication

The code in the functions *mint()* and mintStakerReward() does the same, thus it is considered duplicated, can be refactored and merged.

Path: ./contracts/ReplayToken.sol: *mint(), mintStakerReward()*

Recommendation: Consider reducing the duplicated code to save Gas.

Found in: d266c1b9c7c6db414a50406582e5d5fee1fe3c33

Status: Reported

I08. Optimization by Replacing the Previous State Values of the Roles with the Values that Are Stored in the Memory

The contract ReplayToken uses 3 new variables for on-chain tracking of previous values of the updated roles: minter, admin, and pendingAdmin. They are stored in storage, so they require additional Gas for that.

Path: ./contracts/ReplayToken.sol: previousMinter, previousPendingAdmin, previousAdmin

Recommendation: Consider using memory instead of storage for previous values of the updated roles.

Found in: 0755d4bfb100169b3c1ab4353e9592d6e06a1530

Status: New

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/imaginereplay/smart-contracts/tree/pre-audit
Commit	d266c1b9c7c6db414a50406582e5d5fee1fe3c33
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	NatSpec
Contracts	File: contracts/ReplayToken.sol SHA3: 442b14dd3b113fcbeccb3e1926e398a150a243df89e481951252c34815698673

Second review scope

Repository	https://github.com/imaginereplay/smart-contracts/tree/pre-audit
Commit	2e6d59c1dc54d2f6caf00daafdf08cd7deba66ce
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	NatSpec
Contracts	File: contracts/ReplayToken.sol SHA3: b164e8daf4b2e73583540883b593369dabd87ecd25c57c47851f1f0029d71fc9

Third review scope

Repository	https://github.com/imaginereplay/smart-contracts/tree/pre-audit
Commit	8d3486b60d5700c3249d3e15a909cbab4531c3ca
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	NatSpec
Contracts	File: contracts/ReplayToken.sol SHA3: 62c338f678bc6be3eb7ba2b8404d779a58a578af376c64ad3ece8901deb9ddd0

Fourth review scope

Repository	https://github.com/imaginereplay/smart-contracts/tree/pre-audit
Commit	0755d4bfb100169b3c1ab4353e9592d6e06a1530
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	NatSpec
Contracts	File: contracts/ReplayToken.sol SHA3: ac999ae51fd2a90df5d16a458fe02a8d618c1ed5c062a606ed072b220c9b9cd3