HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: MAJR INC
Date: November 11, 2022

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 MAJR INC
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; Staking; Vesting; Erc721 system
Platform	EVM
Network	Ethereum
Language	Solidity
Methods	Manual Review, Automated Review, Architecture Review
Website	https://www.majrdao.io/
Timeline	06.10.2022 - 11.11.2022
Changelog	11.10.2022 - Initial Review 27.10.2022 - Second Review 11.11.2022 - 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 MAJR INC (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

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

Initial review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts
Commit:
37c6ed92b9ba7f3e91f82538abfea9f675395b79
Documentation:
Business Logic and Technical Documentation
Contracts:
File: ./contracts/MajrCanon.sol
SHA3: 30179693725049cdf170ca1f13fdd72a7f2a852fdd248deb16c3e8c188a1c250

File: ./contracts/MajrContests.sol
SHA3: 7c568afb7e36408c6176a48ae352ad312ab6540a78394b2d703bbd6c473154ec

File: ./contracts/MajrERC20.sol
SHA3: 1b78e118965588bc566dd11069356b0283491f99f0319180d701e632f6cf3522

File: ./contracts/MajrStaking.sol
SHA3: 089f06831f2e55989f88e7a01eae37ebc1df2695b80c0814754c03096aee6cce

File: ./contracts/MajrVester.sol
SHA3: 5fda8bd93480b7874d5dfa68019fd5ff8574be806ba94f927bc838da917c5e7f

Second review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts
Commit:
376f4b9e9d3d21a9162dd2098ba7ea2e70065689
Documentation:
Business Logic and Technical Documentation
Contracts:
File: ./contracts/MajrCanon.sol
SHA3: c310367cf328d54c6ab18bf72cdb87b1658ac388a17a518548afcb8ba309ae1d

File: ./contracts/MajrContests.sol
SHA3: 40da6163d5fb880b12579bccc4b4b01a08cf2877571615c94521b0943172aa8b

File: ./contracts/MajrERC20.sol
SHA3: 267514e118c0da83d50466c1c8371931c8cdb1a7580cf520e5ebf4b1bc5b6dbf

File: ./contracts/MajrStaking.sol
SHA3: 1f6725da0f5cc2a5de91af0ac00243866de3931c91a8e1525c4a247bc8e8cc25

File: ./contracts/MajrVester.sol
SHA3: 1a6e314d85221c178d2a6d2c7c680dc9cf9e61bedb40e2fd5199ae22cb9e4ca9

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

Third review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts

Commit:
4d5e7193a4a073bd0cf4b1a7c2a9c2af8d7dec8a

Documentation:
Business Logic and Technical Documentation

Contracts:

File: ./contracts/MajrCanon.sol
SHA3: c310367cf328d54c6ab18bf72cdb87b1658ac388a17a518548afcb8ba309ae1d

File: ./contracts/MajrContests.sol
SHA3: 40da6163d5fb880b12579bccc4b4b01a08cf2877571615c94521b0943172aa8b

File: ./contracts/MajrERC20.sol
SHA3: 123d984612e65fc408e5130a8d17a6515709a6da3a6b8b0791ce41eed3a42632

File: ./contracts/MajrStaking.sol
SHA3: 71416f1d452358e60bb389af294a4dae31ae3a59ddb8ed38e42275aea065ae7c

File: ./contracts/MajrVester.sol
SHA3: 1a6e314d85221c178d2a6d2c7c680dc9cf9e61bedb40e2fd5199ae22cb9e4ca9

Severity Definitions

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

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

- Functional requirements are provided.
- Technical description is provided.

Code quality

The total Code Quality score is **9** out of **10**.

- The development environment is configured.
- Code does not fully follow the official guidelines.

Test coverage

Test coverage of the project is **95.6%**.

- Deployment and basic user interactions are covered with tests.
- Negative cases are partially covered.
- Interactions by several users are tested.

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: **9.6**.

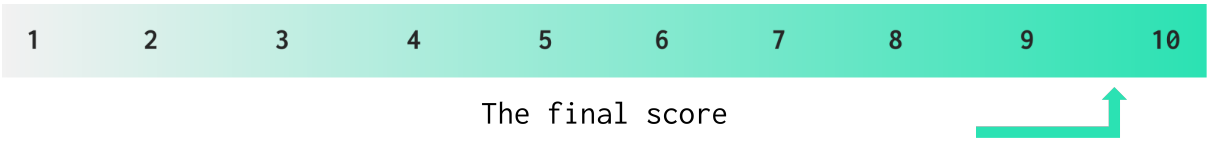

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
10 October 2022	6	3	3	0
26 October 2022	2	0	1	0
11 November 2022	1	0	0	0

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 more specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Not Relevant
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.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed

Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	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	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	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.	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 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.	Not Relevant
Style guide violation	**Custom**	Style guides and best practices should be followed.	Failed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Failed
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

System Overview

MAJR DAO is a cross application membership and ETH engine for the video community. Engineered to accumulate digital assets and create new video experiences for members. It leverages web3 applications that complement existing/new video infrastructure and marketplaces with blockchain technology.
The files in the scope:

- **MajrCanon.sol** - is a governance contract used for making key decisions for the future of the MAJR DAO. There are two primary types of proposals that users can vote on:
    - General proposals: Can be proposed by anyone who has staked at least 200 MAJR IDs and has paid 0.3 ETH as the proposal cost(minimum stake amount and proposal cost can be changed by the contract owner). These proposals are broad and can include anything that can potentially benefit the future of the MAJR DAO, but with the primary focus being put on scalability, practicality and efficiency.
    - Pre-set proposals: Can be proposed only by the Guardian Council (consisting of the MAJR DAO core dev team & community leaders). Here, users vote on pre-set matters on a monthly basis (e.g. which percentage of ETH from the MAJR DAO treasury should be staked this month?)
- **MajrContests.sol** - is a contract used to store the relevant data associated with each MAJR contest held on a weekly basis, with the goal of achieving increased transparency within the MAJR ecosystem.
- **MajrERC20.sol** - is a token issued as a reward for the MAJR ID stakers. In the beginning, users can only get these tokens by staking their MAJR IDs in the staking contract and earning them gradually over time there, as they are made to be initially non-transferable.
- **MajrStaking.sol** - is a staking contract where MAJR DAO members can safely stake their MAJR IDs to receive the following benefits:
    - Voting power: Only staked MAJR IDs have voting power in the MAJR DAO governance system.
    - Staking rewards: Users earn the rewards in form of MAJR XP tokens for staking their IDs in the staking contract.
- **MajrVester.sol** - is a utility contract that vests the specified amount of MAJR ERC20 tokens for an intended recipient over a set period of time.

Privileged roles

- Owner of the MajrCanon - can pause and unpause voting and proposition creation, approves, queues, executes and cancels all proposals and makes pre-set proposals. Additionally, it possesses the ability to configure the key governance variables: quorum votes percentage,

proposal fees, proposal threshold, DAO treasury address and the staking contract address.

- Owner of the MajrContests - can post the contest data for the most recent MAJR contest to the blockchain.
- Owner of the MajrERC20 - can add new addresses to the whitelist before the transfers are enabled. Once transfers are enabled, they cannot be disabled ever again. Owner can change the token's name and symbol, but can do so only once (as a part of the function for enabling transfers). The entire token supply is minted to the owner at the time of the contract deployment.
- Owner of the MajrStaking - can add new amounts of rewards tokens and set new (longer) reward period durations. Admin can pause new staking deposits, but withdrawals and rewards claiming can never be paused.
- Recipient of the MajrVester - able to claim their vested tokens once the pre-defined vesting cliff period has passed. After that, the remaining vesting amount will be unlocked linearly until the end of the vesting period, after which the recipient can claim all vested tokens. The current recipient can set the new recipient at any time if they wish to do so.

Risks

- Owner of the MajrCanon can create and execute pre-set proposals despite the number of votes cast for it.
- Users can only withdraw from staking contract while the voting in the MAJR Canon (governance) contract is not active.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

1. Funds Lock

MajrERC20, MajrStaking, MajrCanon contracts implements *receive* function to be able to receive ETH. Though, there is no functionality that can withdraw those tokens.

In this case, chain native currency will be locked on contract.

Paths: ./contracts/MajrERC20.sol, ./contracts/MajrStaking.sol, ./contracts/MajrCanon.sol

Recommendation: remove the *receive* method from mentioned contracts.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

2. Funds Lock

The owner of the MajrCanon can create a new proposal via method *propose* with the amount of ETH settled. In this case, this ETH will be locked on the contract.

Path: ./contracts/MajrCanon.sol

Recommendation: add *if* statement with a check of the caller and move require statements and ETH transfer inside.

```
        if (msg.sender != owner()) {
            require(
                stakingContract.balanceOf(msg.sender) >= proposalThreshold,
                "MajrCanon: Insufficient number of staked Majr IDs to propose a
proposal."
            );
            require(
                msg.value == proposalCost,
                "MajrCanon: The exact fee for the proposal cost must be sent."
            );
            (bool sent, ) = daoTreasury.call{value: msg.value}("");
            require(sent, "MajrCanon: Failed to send the proposal fee.");
        }
```

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

3. Data Consistency

The contract MajrCanon.sol calculates voting power by the current balance of MAJR ID's staked by user to the MajrStaking.sol. There is a risk that users can manipulate their own voting power inside a single transaction.

It can lead to voting results manipulations.

Paths: ./contracts/MajrCanon.sol, ./contracts/MajrStaking.sol

Functions: vote, votePreSet, getVotingPower

Recommendation: implement snapshot functionality into the MajrStaking.sol and use it in the voting system.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

4. **Denial of Service Vulnerability**

Owner of the contract MajrStaking.sol can change staking token. In this case, users with active stakes would not have the possibility to withdraw their staked tokens.

Additionally, the owner can change the reward token. In this case, users may receive unwanted token rewards.

Paths: ./contracts/MajrStaking.sol

Functions: setRewardsToken, setStakingToken

Recommendation: remove mentioned function or implement a mechanism to prevent described issues.

Status: Fixed (4d5e7193a4a073bd0cf4b1a7c2a9c2af8d7dec8a)

Medium

1. **Using SafeMath in Solidity ^0.8.0**

Integer overflow check is built-in in Solidity ^0.8.0. Due to this, using this library is redundant.

Paths: ./contracts/ MajrStaking.sol, ./contracts/ MajrVester.sol

Recommendation: remove redundant functionality.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

2. **Unchecked Token Transfer**

ERC20 transfer functions return bool after transfers, and it is important to implement a return value check for this return value. This issue leads to unintended behavior of the contract regarding token transfer results.

Paths: ./contracts/ MajrVester.sol, ./contracts/ MajrStaking.sol

Functions: claim, getReward

Recommendation: implement a return value check for token transfers.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

3. **Tautology of Contradiction**

Require statements with checks for `uint256 >= 0` are redundant.

Paths: ./contracts/MajrContests.sol, ./contracts/MajrCanon.sol

Functions: getContestData, validProposalId, validPreSetProposalId

Recommendation: remove redundant parts of code.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

Low

1. Missing Zero Address Validation

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

Path: ./contracts/MajrVester.sol

Function: constructor

Recommendation: implement zero address checks.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

2. State Variables Can Be Declared Immutable

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. State variables majrERC20, vestingAmount, vestingBegin, vestingCliff, vestingEnd in the MajrVester.sol are settled in the constructor and cannot be changed. State variables oldName, oldSymbol are settled in the constructor and cannot be changed.

Paths: ./contracts/ MajrVester.sol, ./contracts/ MajrERC20.sol

Function: constructor

Recommendation: declare mentioned variables as immutable.

Status: Fixed (4d5e7193a4a073bd0cf4b1a7c2a9c2af8d7dec8a)

3. Empty Constructor

A constructor is an optional function. If there is no constructor, the contract will assume the default constructor, which is equivalent to constructor() {}.

This makes redundant parts of code.

Path: ./contracts/ MajrContests.sol

Function: constructor

Recommendation: remove redundant parts of code.

Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

4. Style Guide Violation

The provided projects should follow the official guidelines.

Paths: ./contracts/ MajrStaking.sol, ./contracts/ MajrERC20.sol, ./contracts/ MajrCanon.sol

Recommendation: follow the official Solidity guidelines.

Status: Reported

5. **Assert Violation**

   Properly functioning code should never reach a failing assert statement. It is possible to reach underflow in the function *withdraw* as there is no check for user staked amount.

   Path: ./contracts/ MajrStaking.sol

   Function: withdraw

   Recommendation: add a require statement to check that the withdrawal amount is less or equal to the user staked amount.

   Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

6. **Redundant Import**

   The usage of AccessControl is unnecessary for the contract. In the contract is not implemented role access.

   The use of unnecessary imports will increase the Gas consumption of the code.

   Path: ./contracts/ MajrStaking.sol

   Recommendation: remove the redundant import.

   Status: Fixed (376f4b9e9d3d21a9162dd2098ba7ea2e70065689)

Disclaimers

Hacken Disclaimer

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

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

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

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

Technical Disclaimer

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