HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Majr Dao
Date: September 01st, 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 Dao
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU Noah Jelich | Lead Solidity SC Auditor at Hacken OU
Type	ERC721A token
Platform	EVM
Network	Ethereum
Language	Solidity
Methods	Manual Review, Automated Review, Architecture Review
Website	https://www.majrdao.io
Timeline	01.08.2022 - 01.09.2022
Changelog	05.08.2022 - Initial Review 24.08.2022 - Second review 30.08.2022 - Third Review 01.09.2022 - Fourth Review

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

Table of contents

Introduction 4

Scope 4

Severity Definitions 6

Executive Summary 7

Checked Items 9

System Overview 12

Findings 13

Disclaimers 18

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

Introduction

Hacken OÜ (Consultant) was contracted by Majr Dao (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:
5c852ab03ec4aa773898a3bb9f1a0549451ddb83
Technical Documentation:
Documentation

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/EtherDisperser.sol
SHA3: 29ad9bca2d712a17b07d6a40b2ad71c6f170fa2233ae8d08d7e75ae323d6ce06

File: ./contracts/MajrNFT.sol
SHA3: 970a4da0196314b8b9e928410e7c6c534f39f71e78985c96e35f4731d879de76

File: ./contracts/Splitter.sol
SHA3: 048901098b46928a3b3b5d6085289a6912eec0e34ed4daa9dc425a487ae05ffb

Second review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts
Commit:
5c852ab03ec4aa773898a3bb9f1a0549451ddb83
Technical Documentation:
Documentation

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/EtherDisperser.sol
SHA3: 4c798aed80bd02cf39b525e4849f58b8280d144c491c219f5e9758d30ca71e05

File: ./contracts/MajrNFT.sol
SHA3: 115462fc81bb018afcd46e4d7e874ed1c3dc34b468f5c8c33797d31eff27dd5a

File: ./contracts/Splitter.sol
SHA3: d8c675c32fc5670952b6b04ad0dcdf4b2eeade8ab97c5b67d047803a52e7af38

Third review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts
Commit:
2be6bdd0660b09d18d841ff8320e6a0cbae64ea8
Technical Documentation:
Documentation

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/EtherDisperser.sol
SHA3: b65df3774a4a4bc04e42487b37c9bf86136dad4d50e9d7d41e34f9d260990fe6

File: ./contracts/MajrNFT.sol
SHA3: f9bef6d75d67d6f35b720a3dd76a6c54e1a79a4a8d4e22c4e009bfbd584c269c

File: ./contracts/Splitter.sol
SHA3: e69574871d23b2391ff1590f32f91e8e71431d2c34aa5356ad266bc2548a356d

Fourth review scope

Repository:
https://github.com/MAJR-Inc/majr-dao-contracts
Commit:
0d414f4712458866e5d5b161a8f38973243a38bf
Technical Documentation:
Documentation

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/EtherDisperser.sol
SHA3: 038693621c6cf083f010755bab6de943e61d61b860b674284d34108734c72d25

File: ./contracts/MajrNFT.sol
SHA3: 450514a1e055c9f5f4f2dec48a195b976164f09d24951caebdb6b898c2876157

File: ./contracts/Splitter.sol
SHA3: e69574871d23b2391ff1590f32f91e8e71431d2c34aa5356ad266bc2548a356d

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 methodology.

Documentation quality

The Customer provided a whitepaper with technical documentation that includes functional requirements. The project has technical documentation on the web. The total Documentation Quality score is **10** out of **10**.

Code quality

The total CodeQuality score is **5** out of **10**. Deployment and basic user interactions are covered with tests. **Test coverage is 56.98%.** Negative cases coverage is missed, and interactions by several users are not tested thoroughly.

Architecture quality

The architecture quality score is **10** out of **10**. The project has clear and clean architecture. The project contains development instructions.

Security score

As a result of the audit, the code contains **1** medium and **1** low severity issue. The security score is **9** 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: **8.8**.

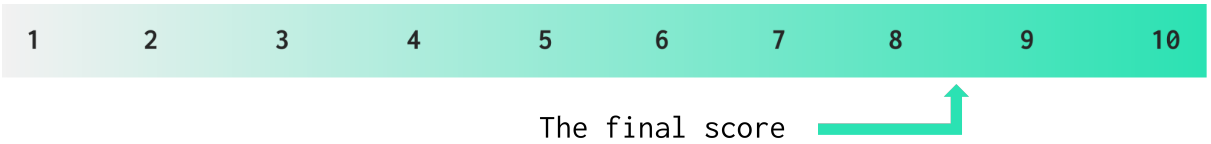

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
4 August 2022	5	4	2	1
22 August 2022	3	1	1	1
30 August 2022	1	0	1	1
01 September 2022	1	1	0	0

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

Checked Items

We have audited provided smart contracts for commonly known and more specific vulnerabilities. Here are some of the items that are 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.	Passed
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.	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 it is 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	SWC-116	Block numbers should not be used for time	Not Relevant

a proxy for time		calculations.	
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 identifier 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.	Not Relevant
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.	Failed
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	**Custom**	The project should contain a configured	Passed

Consistency		development environment with a comprehensive description of how to compile, build and deploy the code.	
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 mixed-purpose system with the following contracts:

- *MajrNFT* — simple ERC-721A non-fungible token with unlimited minting. It has the following attributes:
  - Name: Majr ID
  - Symbol: MAJR
  - Total supply: Unlimited.
- *EtherDisperser* — a contract that allows users to claim rewards earned from contests.
- *Splitter* - an abstract contract within the MajorNFT contract that distributes the amount of ETH to specific wallet addresses during the mint function execution.

Privileged roles

- The owner of the *MajrNFT* contract can pause, unpause the contract.
- The owner of the *MajrNFT* contract can withdraw ETH, ERC20 tokens sent to the contract.
- The owner of the *MajrNFT* contract can change the tokenBaseURI and contractMetadataURI.
- The owner of the *EtherDisperser* contract can withdraw ETH, ERC20 tokens sent to the contract.

Risks

- According to the documentation, the split address list can have a maximum of 3 elements, and the referral address list can have a maximum of 4 elements. Two lists, Referral and Split, are dynamically defined in the *Splitter* contract. The size of these lists is limited by the require control with the *uint256 cap* variable defined inside. With the setCap function, the admin can change the size of these lists, exceeding the maximum values specified in the documentation. If *uint256 cap* is set as a large number for loops used within the *split*, *referralSplit*, *_checkForReferralAddress*, *_checkForInvalidAddress*, *_getSum* functions will cause a problem.

- ERC20 tokens being used in the competitions should adopt standard ERC20 implementations. Any implementation with a fallback logic can lead to unexpected behavior during token transfers, such as reverts.
- NFTs will be minted as a random rarity to the users, and rarity properties will be stored in the IPFS. If the IPFS is not well protected, users can see NFT rarity before the NFT is minted.

Findings

Critical

1. Referral address manipulation

Since it is a public function, everyone can call the *mintWithReferrer* and give their wallet as an input and get the 5% referral amount themselves.

This can lead anybody to take 5% referral fee.

Path: ./contracts/MajrNFT.sol : mintWithReferrer()

Recommendation: The project benefits from the ERC721A standard which has an internal function that checks the number of NFTs minted by an address, _numberMinted(). The important thing is that this function returns the number of NFTs minted by an address, even if the address does not hold that NFT anymore. Based on this, adding require check to mintWithReferrer() which checks if the number of NFTs minted by the referrer address is greater than zero. If it is, then this means that the given referrer address is a valid user of the system and it will get the referral fee. If it's not the contract will revert.

Status: Fixed (0d414f4712458866e5d5b161a8f38973243a38bf)

High

1. Requirements violation

If the EtherDisperser contract is not funded with ETH it will be impossible to claim the reward.

When the user tries to claim the rewards, this can lead to an error in the *claim* function as the contract does not check for sufficient funds.

The *EtherDisperser* update balance function adds to users' current balance each time. Suppose the admin runs the *recoverEther* function after updating the user's balance address. In that case, the admin can get all the Ether in the contract, but the previously updated user balances will still be active. When the admin executes the *updateBalance* Since the mapping keeping the user balances is not deleted, it will add to the existing balance of the users.

Path: ./contracts/EtherDisperser.sol : updateBalance() recoverEther()

Recommendation: Keep the user's total reward in a variable. Inside the *RecoverEther*, check the admin only withdraws the excess amount.

Status: Fixed (0d414f4712458866e5d5b161a8f38973243a38bf)

2. Highly permissive role access

The contract owner can withdraw all the ETH and tokens from the contract. This withdrawal can be done anytime without informing the users, leading to sudden balance changes.

This can lead to sudden ETH depletion in the contract.

Paths: ./contracts/EtherDisperser.sol : recoverERC20()

/contracts/EtherDisperser.sol : recoverETH()

Recommendation: Remove this functionality or inform users in public documentation.

Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

Medium

1. 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 contract about token transfer result.

Paths: ./contracts/MajrNFT.sol: recoverERC20()

./contracts/Splitter.sol : recoverERC20()

Recommendation: Implement a return value check for token transfers.

Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

2. Unoptimized loop usage

tokensOfUser() function returns an array of all Major NFTs owned by the user. Using a for loop depending on *totalSupply* can cause an Out Of Gas error as the amount of tokens users have can be unlimited.

The updateBalances function accepts user-supplied arrays as inputs and iterates over them.

This can lead to excessive Gas consumption.

The *removeBalances()* function takes and array with a user supplied length as a parameters. In addition to that, the function iterates this array and performs external calls.

This can lead to excessive Gas consumption.

Paths: ./contracts/EtherDisperser.sol : removeBalances()

Recommendation: Implement array size limitations.

Status: New

3. **Requirements violation**

   In the provided documentation, there are two different split methods with different percentages for users. However, in the provided whitepaper, there is a completely different split method.

   Recommendation: Review and fix the documentation.

   Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

4. **Requirements violation**

   In the provided documentation, there are three different types of rarities for an NFT. The minted NFT will have a random rarity set to it. However, this feature is not implemented.

   Recommendation: Either remove this feature from the documentation or implement it.

   Status: Mitigated.

5. **Wrong Addresses can be Registered to the Contract**

   To register an address to the contracts, admin needs to run updateBalances function. However, if the admin registers a wrong address there are no mechanisms to remove this address from the contracts

   Paths: ./contracts/EtherDisperse.sol : updateBalances()

   Recommendation: Implement a way to remove addresses that are added by a mistake.

   Status: Fixed (2be6bdd0660b09d18d841ff8320e6a0cbae64ea8)

Low

1. **Redundant condition check**

   Since `require(_targets.length==_amounts.length)` and `require(_targets.length >0)` are checked, `require(_amounts.length > 0)` does not need to be checked.

   Path: ./contracts/EtherDisperser.sol : updateBalances()

   Recommendation: Remove unnecessary `require(_amounts.length > 0)` condition check.

   Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

2. **Floating pragma**

   Contracts should be deployed with the same compiler version and flags that have been tested thoroughly. 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.

Paths: ./contracts/MajrNFT.sol, ./contracts/EtherDisperser.sol, ./contracts/Splitter.sol

Recommendation: Use a fixed version of the compiler (^ symbol should be removed from pragma). Consider using the same compiler version for all contracts.

Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

3. **Redundant use of external calls**

In the mint functions, the MAJRNFT contract uses external calls to split fees to users. Since MAJRNFT inherits Splitter contract, such external calls are redundant and consume Gas.

Paths: ./contracts/MajrNFT.sol: mint()

./contracts/MajrNFT.sol: mintWithReferrer()

Recommendation: Do a simple transfer to the referrer, then split the remainder of funds during an admin withdrawal.

Status: Reported

4. **Missing zero address validation**

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

This can lead to unwanted external calls to 0x0.

Path: ./contracts/EtherDisperse.sol : recoverERC20()

Recommendation: Implement zero address checks.

Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

6. **Missing zero-check on NFT price**

Admin can set the price of NFT with the `setPrice` function. There are no zero-checks when setting the price here.

This can lead to zero-priced NFTs.

Path: ./contracts/MajrNFT.sol : setPrice()

Recommendation: Add `require(_price > 0);` beginning of the function.

Status: Fixed (6204a80c7a27a50a99637ec8d30e8ec08eb54f05)

7. **Using memory instead of calldata**

Function parameters are defined as memory. Using memory instead of calldata will cause the function to consume more gas.

Path: ./contracts/EtherDisperser.sol : updateBalances()

Recommendation: Use calldata instead of memory to save gas.

Status: Fixed (2be6bdd0660b09d18d841ff8320e6a0cbae64ea8)

8. **Missing update of a State Variable**

   The state variable `totalEtherRewarded`is stores the ETH gathered form the users. This value is not updated after a calling recoverEther(). Since this value is not used anywhere contracts, it is possible to assume it is used in the front-end.

   Path: ./contracts/EtherDisperser.sol : -

   Recommendation: Update this variable after ETH transfers from this contract .

   Status: Fixed (2be6bdd0660b09d18d841ff8320e6a0cbae64ea8)

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.