Polarys

smart contracts
final audit report

July 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Polarys team to perform an audit of their smart contract. The audit was conducted between 20/07/2022 and 25/07/2022.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at the GitHub repositories: @PolarysDAC/polarys-metis-contract and @PolarysDAC/polarys-evm-contract. The code was audited after the commit 9b898e5 (@PolarysDAC/polarys-metis-contract) and the commit 67cbd2b (@PolarysDAC/polarys-evm-contract).

Update: the Polarys team has responded to this report. The updated code is located in the repositories: @PolarysDAC/polarys-metis-contract after the commit 57b2b19 and @PolarysDAC/polarys-evm-contract after the commit c76c99a.

The audited contracts were deployed to addresses:- PolarysNFTContract `0x41FEcb9bA1E142fA332472D4eA66c6f1C9B07b7c` in Metis Chain - DepositContract `0x6b2ec4dfe27dca61444ad9157291b3224f9b9427` in Ethereum, Polygon, BSC, Fantom, Avalanche chains.

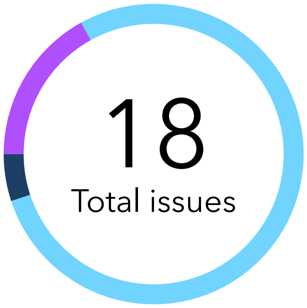
2.1 Summary

Project name	Polarys
URL	https://www.polarys.io/
Platform	Metis
Language	Solidity

2.2 Contracts

Name	Address
ERC721B	
PolarysNFTContract	
DepositContract	

3. Found issues



●	Medium	3 (17%)
●	Low	14 (78%)
●	Info	1 (5%)

C1. ERC721B

ID	Severity	Title	Status
C1-01	Low	Floating pragma	Resolved

C2. PolarysNFTContract

ID	Severity	Title	Status
C2-01	Medium	Missing functionality for prices	Resolved
C2-02	Medium	Withdrawal of undistributed rewards	Resolved
C2-03	Low	Floating Pragma	Resolved
C2-04	Low	AccessControl and Ownable using	Resolved
C2-05	Low	Lack of events	Resolved
C2-06	Low	Lack of parameters validation	Resolved
C2-07	Low	Gas optimization	Resolved

C3. DepositContract

ID	Severity	Title	Status
C3-01	Medium	Replay of depositToken()	Resolved
C3-02	Low	Floating pragma	Resolved
C3-03	Low	Simultaneous usage of AccessControl and Ownable	Resolved
C3-04	Low	depositToken() params check	Resolved
C3-05	Low	Lack of event	Resolved
C3-06	Low	Gas optimization	Resolved
C3-07	Low	Lack of parameter validation	Resolved
C3-08	Low	Changing _acceptToken and further withdrawal	Resolved
C3-09	Low	Excessive MerkleProof check on depositToken()	Resolved
C3-10	Info	Typos	Resolved

4. Contracts

C1. ERC721B

Overview

A fully compliant implementation of IERC721 with significant gas savings for minting multiple NFTs in a single transaction. Includes the Metadata and Enumerable extension.

At the same time, the contract has a very expensive `balanceOf()` function.

Issues

C1-01 Floating pragma — Low — Resolved

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to 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.

C2. PolarysNFTContract

Overview

The NFT contract extends ERC721B functionality and with royalty implementation inherited from the OpenZeppelin ERC2981 contract.

The contract has the functionality to change the price of tokens depending on the sale stage.

Contract's admin role allows to mint tokens to users.

According to the developer, the purchase of NFT is made through the contract DepositContract by calling the `depositToken()` function, and then a third-party application with the Minter role performs the minting of tokens to the buyer. The contract itself does not provide any guarantees that the right amount of tokens will be minted to the account that purchased them, it all depends on accounts with the Minter role.

Issues

C2-01 Missing functionality for prices

Medium Resolved

The functions `setPrivateSalePrice()`, `setPublicSalePrice()` allow to set prices. But these prices are never used when minting (purchasing) tokens.

Without documentation, it is impossible to say about the correct operation of the minting (purchase) of NFT tokens.

Recommendation

a. Make sure you need to use prices.

b. Make sure the mint function should not use prices.

c. Add documentation.

Update

According to developer team prices will be used by the third-party application to generate the buyer's signature.

Note: no third party application has been audited by the audit team.

C2-02 Withdrawal of undistributed rewards

Medium Resolved

The `depositMetis()` function allows transferring native currency to the contract. This native currency is supposed to be paid out to some users at the time of minting. After the minting is completed, part of the currency may not be distributed, and it will be blocked in the contract.

Recommendation

We recommend adding a function to withdraw undistributed native tokens after the minting is ended.

C2-03 Floating Pragma Low Resolved

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to 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.

C2-04 AccessControl and Ownable using Low Resolved

OpenZeppelin's AccessControl authentication model assumes defining specific roles. The contract also uses the Ownable library, although one more role is sufficient instead.

Recommendation

Consider adding one more role instead of using the Ownable library or use the default role from Access control.

C2-05 Lack of events Low Resolved

The functions `setPrivateSalePrice()`, `setPublicSalePrice()`, `setRoyaltyFee()`, `setBaseURI()` don't emit events, which complicates the tracking of important off-chain changes.

C2-06 Lack of parameters validation Low Resolved

Consider adding validation for input parameters of the `setPrivateSalePrice()`, `setPublicSalePrice()`, `setRoyaltyFee()` functions.

C2-07 Gas optimization

Low Resolved

a. The `getRoyaltyFee()`, `getPrivateSalePrice()`, `getPublicSalePrice()`, `depositMetis()`, `exists()` functions can be declared as `external` to save gas.

b. The `require` check on L105 is redundant because the same check already exists inside the ERC721B._mint() function.

```
require(to != address(0), "Should not be zero address");
```

c. No need to pause the `mint()` function using the if statement in L123-125 because the `require` statement in L106 will not allow to mint more than `MAX_SUPPLY` value. Moreover, the inheritance of the Pausable contract is redundant as it does not add any useful functionality.

```
    function mint(address to, uint256 quantity) external whenNotPaused
onlyRole(MINTER_ROLE) nonReentrant {
        ...
        require(_currentSupply + quantity <= MAX_SUPPLY, "Can not mint NFT more than
MAX_SUPPLY");
        ...

        _currentSupply += quantity;

        if (MAX_SUPPLY == _currentSupply) {
            _pause();
        }
        ...
    }
```

d. The state variables `_currentSupply`, `_royaltyFee` are read several times in the `mint()` function. Consider adding local variables for these values to save gas.

e. No need to use a special variable `metisBalance` for contract balance. Using `address(this).balance` in L108 will save gas.

f. Instead of declaring and reading `_currentSupply` storage variable, ERC721B `_owners.length` can be read.

C3. DepositContract

Overview

The contract allows depositing ERC20 tokens with specifying the quantity of NFT tokens. The deposit can only be made by users with the Deposit role.

Users who have admin permission can withdraw deposited ERC20 tokens to any address.

According to the developer, the contract allows accepting payment for NFT in other chains. After payment, the contract emits events that must be processed by third-party applications. Based on these events, users will be able to receive NFT tokens on the Metis network. At the same time, the audit team did not audit third-party applications and warns of the possibility of exceeding the number of purchased NFT tokens on different networks by the maximum possible number (`MAX_SUPPLY = 2500`) of minted tokens by the PolarysNFTContract contract in the Metis network.

Issues

C3-01 Replay of depositToken() ● Medium ✓ Resolved

A user who has a signature that allows the function `depositToken()` to be executed can replay his call several times until the deadline arrives. This will emit multiple events with the same parameters.

Recommendation

We recommend verifying the signature against reuse.

C3-02 Floating pragma ● Low ✓ Resolved

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to 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.

C3-03 Simultaneous usage of AccessControl and Ownable ● Low ✓ Resolved

OpenZeppelin's [AccessControl] authentication model assumes defining specific roles. The contract also uses the Ownable library, although one more role is sufficient instead.

Recommendation

Consider adding one more role instead of using the Ownable library or using the default admin role from AccessControl.

C3-04 depositToken() params check ● Low ✓ Resolved

The params `amount` and `quantity` of the `depositToken()` function are not checked and may content unexpected values.

Update

According to developer the amount parameter is checked on the third-party application (backend).

C3-05 Lack of event

Low Resolved

The function `setupAcceptToken()` doesn't emit events, which complicates the tracking of important off-chain changes.

C3-06 Gas optimization

Low Resolved

a. The `setupAdminRole()`, `setupDepositRole()`, `setupAcceptToken()`, `getAcceptToken()` functions can be declared as external to save gas.

b. The internal function `toBytes32()` is never used in contract code and can be removed to save gas on deployment.

C3-07 Lack of parameter validation

Low Resolved

The setupAcceptToken() function does not check the address `token` for a non-zero value.

C3-08 Changing _acceptToken and further withdrawal

Low Resolved

The contract owner can change `_acceptToken` using the `setupAcceptToken()` function. This will block the ability to withdraw previously deposited tokens with a different address

Recommendation

Consider adding the additional paramater `tokenAddress` into `withdrawToken()` function. It will allow withdrawing any token at any time by the admin.

C3-09 Excessive MerkleProof check on depositToken()

Low Resolved

The parameter `status` of the function `depositToken()` denotes the type of sale: private or public.

This parameter is also included in a `signature` created by a third-party app.

Additionally, the third-party application generates a list of addresses allowed for private sale. This is done using the Merkle proof pattern.

Thus, both the third-party application and the contract do the same work twice. In our opinion, adding the `status` parameter is enough to identify the purchase in the private mode. And there is no need to use on-chain MerkleProof verification.

Update

The Polarys team explained that the Merkle proof check is needed because there are some cases when the backend can create a signature with `status = 1` for non-whitelisted addresses.

C3-10 Typos

Info Resolved

Typos reduce the code's readability:in L96-L98 'receipient' should be replaced with 'recipient'.

5. Conclusion

3 medium, 14 low severity issues were found during the audit. 3 medium, 14 low issues were resolved in the update.

3 medium, 13 low and 1 info severity issues have been resolved in the update. 1 low and 1 informational severity issues were added in the update.

The contracts are highly dependent on third-party applications and the owner's account. Users using the project have to trust the project team, owner and that the owner's account is properly secured, and that the third-party applications work properly.

We strongly suggest adding documentation as well as unit and functional tests for all contracts.

This audit includes recommendations on improving the code and preventing potential attacks.

Note: no third party application has been audited by the audit team.

The audited contracts were deployed to addresses:

- PolarysNFTContract `0x41FEcb9bA1E142fA332472D4eA66c6f1C9B07b7c` in Metis Chain

- DepositContract `0x6b2ec4dfe27dca61444ad9157291b3224f9b9427` in Ethereum, Polygon, BSC, Fantom, Avalanche chains.

Appendix A. Issues severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

contact@hashex.org

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx

BLOCKCHAIN SECURITY