AUDITS

SECURITY ASSESSMENT

POLKAWAR

SEPTEMBER 01 ST 2022

AUDITS

TABLE OF CONTENTS

LEGAL DISCLAIMER

MH Audits are not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts MH Audits to perform a security review.

MH Audits does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

MH Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

The report is provided only for the contract(s) mentioned in the report and does not include any other potential additions and/or contracts deployed by Owner. The report does not provide a review for contract(s), applications and/or operations, that are out of this report scope.

MH Audits' goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

MH Audits represents an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. MH Audits' position is that each company and individual are responsible for their own due diligence and continuous security.

The security audit is not meant to replace functional testing done before a software release. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent manual audits and a public bug bounty program to ensure the security of the smart contracts.

INTRODUCTION

MH Audits is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.

Secure your project with MH Audits

We offer field-proven audits with in-depth reporting and a range of suggestions to improve and avoid contract vulnerabilities.

Industry-leading comprehensive and transparent smart contract auditing on all public and private blockchains.

Vunerability checking

A crucial manual inspection carried out to eliminate any code flaws and security loopholes. This is vital to avoid vulnerabilities and exposures incurring costly errors at a later stage.

Contract verification

A thorough and comprehensive review in order to verify the safety of a smart contract and ensure it is ready for launch and built to protect the end-user.

Risk assessment

Analyse the architecture of the blockchain system to evaluate, assess and eliminate probable security breaches. This includes a full assessment of risk and a list of expert suggestions.

In-depth reporting

A truly custom exhaustive report that is transparent and depicts details of any identified threats and vulnerabilities and classifies those by severity.

Fast turnaround

We know that your time is valuable and therefore provide you with the fastest turnaround times in the industry to ensure that both your project and community are at ease.

Best-of-class blockchain engineers

Our engineers combine both experience and knowledge stemming from a large pool of developers at our disposal. We work with some of the brightest minds that have audited countless smart contracts over the last 4 years.

PROJECT SUMMARY

PROJECT INTRODUCTION

PolkaWar is a cross-chain decentralized fighting game, integrating NFTs and competitive gaming aspects. PolkaWar creates a vivid fighting world for players to develop their characters and engage in combat with each other. There would be different weapons and equipment to arm one's in-game character with.

The game would also involve a stat or points system, allowing for level upgrade and advancement conditions. Therefore, offering more customization and allowing for a highly unique / interactive game play. Though initially launched on the BNB Chain for it's low fees and fast processing times, it would be deployed on Polkadot and other blockchains in the future.

Project Name *PolkaWar*

Contract Name *PWAR Token*

Contract Address *0x16153214e683018d5aa318864c8e692b66e16778*

Contract Chain *Mainnet*

Contract Type *Smart Contract*

Platform *EVM*

Language *Solidity*

Codebase *GitHub Repository*

INFO & SOCIALS

Network *BNB Chain (BEP20)*

Max Supply *100,000,000*

Website *https://polkawar.com/*

Twitter *https://twitter.com/polkawarnft*

Telegram Chat *https://t.me/polkawarchat*

Telegram Ann *https://t.me/polkawar*

Discord *https://discord.gg/NQFjXkMqgk*

Instagram *https://www.instagram.com/polkawarnft/*

Medium *https://polkawar.medium.com/*

GitHub *https://github.com/polkawar*

BSCScan *https://polygonscan.com/token/0x16153214e683018d5aa318864c8e692b66e16778*

AUDIT SCORES

Issues	16
Critical	0
Major	1
Medium	1
Minor	6
Informational	8
Discussion	0

All issues are described in further detail on the following pages.

AUDIT SCOPE

FILE	LOCATION
CorgibStaking.sol	*GitHub Repository polkawar-contract/contracts/*
PolkaWar.sol	*GitHub Repository polkawar-contract/contracts/*

REVIEW METHODOLOGY

TECHNIQUES

This report has been prepared for PolkaWar to discover issues and vulnerabilities in the source code of the PolkaWar project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic, Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts producedby industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from major to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective in the comments below.

TIMESTAMP

Version	v1.0
Date	2022/09/01
Description	Layout project
	Automated / Manual review / Static & dynamic security testing
	Summary

KEY FINDINGS

TITLE	SEVERITY	STATUS
Floating Pragma	Low	*Pending*
Functions Should Be Declared External	Gas	*Pending*
Missing Reentrancy Protections	Major	*Pending*
Missing Zero Address Validations	Low	*Pending*
Missing Events	Low	*Pending*
Missing Return Value Validation	Medium	*Pending*
Missing SPDX License	Informational	*Pending*
Missing Constant Attribute in Variables	Low	*Pending*
Function Should Return Struct	Informational	*Pending*
Use of SafeMath	Gas	*Pending*
Large Number Literals	Gas	*Pending*

KEY FINDINGS

TITLE	SEVERITY	STATUS
Cheaper Inequalities in if()	**Gas**	*Pending*
Cheaper Inequalities in require()	**Gas**	*Pending*
Function State Mutability can be Restricted to Pure	**Informational**	*Pending*
EndDate **Can Be Set Past In Time**	**Low**	*Pending*
Missing Input Validation	**Low**	*Pending*

Description: *Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities. The contracts found in the repository were allowing floating or unlocked pragma to be used, i.e., >=0.6.0.*

This allows the contracts to be compiled with all the solidity compiler versions above 0.6.0.

Location: *CorgibStaking.sol L01*
PolkaWar.sol L01

Impacts: *If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.*

Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic. The likelihood of exploitation is really low.

Issue: *Floating Pragma*

Type: *Floating Pragma (SWC-103)*

Level: *Low*

Recommendation: *Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use 0.8.7 pragma version.*

Reference: https://swcregistry.io/docs/SWC-103

Alleviation / Retest:

Description: *Public functions that are never called by a contract should be declared external in order to conserve gas.*

The following functions were declared as public but were not called anywhere in the contract, making the public visibility useless.

Location: *CorgibStaking.sol*
changeMinimumStakeAmount() - ***L57-L59***
addPool() - ***L65-L92***
setPool() - ***L94-L113***
deposit() - ***L181-L220***
withdraw() - ***L222-248***
emergencyWithdraw() - ***L250-L258***

Impacts: *Smart Contracts are required to have effective Gas usage as they cost real money, and each function should be monitored for the amount of gas it costs to make it gas efficient.*

*"*public*" functions cost more Gas than "*external*" functions.*

Issue: *Functions Should Be Declared* External

Type: *Gas Optimization*

Level: *Gas*

Recommendation: *Use the "*external*" state visibility for functions that are never called from inside the contract.*

Alleviation / Retest:

Description: *In a Reentrancy attack, a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.*

*The smart contract was missing reentrancy protection on the following fuctions making external calls. Both the functions are making calls to the function "*safeTokenTransfer()*" which is then making an external call to the "*transfer()*" function. Multiple state variables are being written after the external call and the address of the external call can also be controlled by the user since it's "*msg.sender*".*

*The function "*emergencyWithdraw()*" is also affected since there's an event emitted after the external call. This is lower in severity when compared to the "*deposit()*" and "*withdraw()*" functions because there's no state change happening here.*

Location: *CorgibStaking.sol L222-L248; L250-L258*

Impacts: *Lacking reentrancy protection could allow threat actors to abuse the functions and reenter the contract. This can lead to excessive interactions with the functions and loss of funds and tokens.*

Issue: *Missing Reentrancy Protections*

Type: *Reentrancy (SWC-107)*
https://swcregistry.io/docs/SWC-107

Level: *Major*

Recommendation: *Add a Reentrancy guard to the function making external calls.*

Alleviation / Retest:

Description: *The contract "*CorgibStaking.sol*" was found to be setting or using new addresses without proper validations for zero addresses.*

Address type parameters should include a zero-address check otherwise contract functionality may become inaccessible or tokens burnt forever.

Depending on the logic of the contract, this could prove fatal and the users or the contracts could lose their funds, or the ownership of the contract could be lost.

Affected Variables: *CorgibStaking.sol L67-L68*
_stakeToken
_rewardToken

Impacts: *If address type parameters do not include a zero-address check, contract functionality may become unavailable or tokens may be burned permanently.*

Issue: *Missing Zero Address Validations*

Type: *Missing Input Validation*

Level: *Low*

Recommendation: *Add a zero address validation to all the functions where addresses are being set.*

Alleviation / Retest:

Description: *Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions.*

The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Functions: *CorgibStaking.sol*

changeMinimumStakeAmount() - ***L57-L59***
addPool() - ***L65-L92***
setPool() - ***L94-L113***
updatePool() - ***L160-L179***

Impacts: *Events are used to track the transactions off-chain and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.*

Issue: *Missing Reentrancy Protections*

Type: *Missing Best Practices*

Level: *Low*

Recommendation: *Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.*

Alleviation / Retest:

Description: *The contract "*CorgibStaking.sol*" is making an external transfer call on lines 269 and 271 inside the function "*safeTokenTransfer()*". Several tokens do not revert and return false. This may cause issues and failed assumptions when making token transfers.*

Affected Code:

```
function safeTokenTransfer(
      address _to,
      uint256 _amount,
      uint256 _pid
  ) internal {
      PoolInfo storage pool = poolInfo[_pid];
      uint256 totalPoolReward = pool.allocPoint;

      if (_amount > totalPoolReward) {
          pool.rewardToken.transfer(_to, totalPoolReward);
      } else {
          pool.rewardToken.transfer(_to, _amount);
      }
  }
```

Impacts: *Missing error handling on transfer return value may cause issues if the call fails as it will create inconsistencies with failed function calls.*

Issue: *Missing Return Value Validation*

Type: *Unchecked Call Return Value (SWC-104)*
https://swcregistry.io/docs/SWC-104

Level: *Medium*

Recommendation: *It is recommended to use "*SafeERC20*" or check the return values of transfer and handle the errors appropriately.*

Alleviation / Retest:

Description: *The contracts "CorgibStaking.sol" and "PolkaWar.sol" were missing an SPDX License identifier in the source code. A smart contract whose source code is available can better establish trust. In order to minimize legal problems relating to copyright, Solidity encourages the use of machine-readable SPDX license identifiers*

Vulnerable Code:

```
pragma solidity >=0.6.0;

import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/ownership/Ownable.sol";
```

Impacts: *SPDX Licenses help in identifying the legal owner of the software, therefore, helping in issues like copyright infringement.*

Issue: *Missing SPDX License*

Type: *Best Practices*

Level: *Informational*

Recommendation: *Every source file should start with a comment indicating its license. Add a necessary license identifier in the contract code like the one shown below:*

// SPDX-License-Identifier: MIT

Alleviation / Retest:

Description: *State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile time.*

The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

Compared to regular state variables, the gas costs of constant and immutable variables are much lower since no SLOAD *is executed to retrieve constants from storage because they're interpolated directly into the bytecode*

Location: *CorgibStaking.sol L08*

Impacts: *Gas usage is increased if the variables that should be constants are not set as constants.*

Issue: *Missing Constant Attribute in Variables*

Type: *Gas Optimization*

Level: *Low*

Recommendation: *A “*constant*” attribute should be added in the parameters that never change to save the gas.*

Alleviation / Retest:

Description: *The contract "*CorgibStaking.sol*" was found to be returning multiple uint256 values in the function "*getPoolInfo()*". Consider using a struct instead of multiple return values as It can improve code readability.*

Location: *CorgibStaking.sol L286-L292*

Impacts: *This affects the overall gas usage and code readability of the contract.*

Issue: *Function Should Return Struct*

Type: *Gas Optimization*

Level: *Informational*

Recommendation: *Consider returning a struct instead of multiple values from a function.*

Alleviation / Retest:

Description: *SafeMath library is found to be used in the contract. This increases gas consumption than traditional methods and validations if done manually.*

Also, Solidity 0.8.0 and above includes checked arithmetic operations by default, and this renders SafeMath unnecessary.

Location: *CorgibStaking.sol L09*

Impacts: *This increases the gas usage of the contract.*

Issue: *Use of SafeMath*

Type: *Gas Optimization*

Level: *Gas*

Recommendation: *We do not recommend using SafeMath library for all arithmetic operations. It is good practice to use explicit checks where it is really needed and to avoid extra checks where overflow/underflow is impossible.*

The compiler should be upgraded to Solidity version 0.8.0+ which automatically checks for overflows and underflows.

Alleviation / Retest:

Description: *Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations. The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation also supported by Solidity.*

Location: *CorgibStaking.sol L54*

Impacts: *Having a large number literals in the code increases the gas usage of the contract while its deployment and when the functions are used or called from the contract. It also makes the code harder to read and audit and increases the chances of introducing code errors.*

Issue: *Large Number Literals*

Type: *Gas & Missing Best Practices*

Level: *Gas*

Recommendation: *Scientific notation in the form of 2e10 is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal MeE is equivalent to M * 10**E. Examples include 2e10, 2e10, 2e-10, 2.5e1, as suggested in official solidity documentation.*

https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals

Alleviation / Retest:

Description: *The contract was found to be doing comparisons using inequalities inside the "*if*" statement. When inside the "*if*" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).*

Location: *CorgibStaking.sol L104; L107; L110; L139; L192; L197; L208; L231; L236; L268*

Impacts: *Using strict inequalities inside "*if*" statements cost more gas.*

Issue: *Cheaper Inequalities in* if()

Type: *Gas & Missing Best Practices*

Level: *Gas*

Recommendation: *It is recommended to go through the code logic, and, if possible, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.*

Alleviation / Retest:

Description: *The contract was found to be performing comparisons using inequalities inside the “*require*” statement. When inside the “*require*” statements, non-strict inequalities (>=, <=) are usually costlier than strict equalities (>, <).*

Location: *CorgibStaking.sol L186; L225*

Impacts: *Using non-strict inequalities inside “*require*” statements cost more gas.*

Issue: *Cheaper Inequalities in* require()

Type: *Gas & Missing Best Practices*

Level: *Gas*

Recommendation: *It is recommended to go through the code logic, and, if possible, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.*

Alleviation / Retest:

Description: *The contract “*CorgibStaking.sol*” was using a function called “*getMultiplier()*” which is neither reading nor writing to the state and therefore, it’s state mutability can be restricted to pure.*

Location: *CorgibStaking.sol L116-L122*

Impacts: *It is a good practice to set the state mutability of a function to pure that is not modifying or reading from the state.*

Issue: *Function State Mutability can be Restricted to Pure*

Type: *Best Practices*

Level: *Informational*

Recommendation: *Change the state mutability of the function “*getMultiplier()*” to pure.*

Alleviation / Retest:

Description: *Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components.*

When the smart contract does not validate the inputs properly, it may introduce a range of vulnerabilities.

*The contracts "*CorgibStaking.sol*" was missing input validation in the functions "*addPool()*" and "*setPool()*" in which the end date of the vesting period - "*_endDate*", has no validation and can be set in the past.*

Location: *CorgibStaking.sol L70; L98*

Impacts: *This issue allows a malicious contract owner to manipulate the release of tokens in the beneficiary's account or maybe by mistake if there is an error in epoch value leading to past dates of vesting period.*

Issue: EndDate *Can Be Set Past In Time*

Type: *Input validation*

Level: *Low*

Recommendation: *Use a* require() *validation in the function that validates if the value of the* _endDate *parameter is not in the past.*

Alleviation / Retest:

Description: *Input validation is a frequently-used technique for checking potentially dangerous inputs in order to ensure that the inputs are safe for processing within the code, or when communicating with other components.*

When the smart contract does not validate the inputs properly, it may introduce a range of vulnerabilities.

*The contracts "*CorgibStaking.sol*" was missing input validation in the function "*addPool()*" on the following parameters:*

Vulnerable Code: *CorgibStaking.sol*
_allocPoint - ***L66***
_rewardPerBlock - ***L69***

Impacts: *Missing input validation on sensitive function parameters may introduce inconsistencies and erroneous logic when the user will interact with the added pools.*

Issue: *Missing Input Validation*

Type: *Input validation*

Level: *Low*

Recommendation: *Use a* require() *input validation in the function parameters shown above.*

Alleviation / Retest:

SOURCE CODE

GitHub Repository - https://github.com/polkawar/polkawar-contract

BNB Chain Deployment - https://bscscan.com/address/0x16153214e683018d5aa318864c8e692b66e16778#code

FINDING CATEGORIES

The assessment process will utilize a mixture of static analysis, dynamic analysis, in-depth manual review and/or other security techniques.

This report has been prepared for PolkaWar project using the above techniques to examine and discover vulnerabilities and safe coding practices in PolkaWar's smart contract including the libraries used by the contract that are not officially recognized.

A comprehensive static and dynamic analysis has been performed on the solidity code in order to find vulnerabilities ranging from minor gas optimizations to major vulnerabilities leading to the loss of funds.

Various common and uncommon attack vectors will be investigated to ensure that the smart contracts are secure from malicious actors. The testing methods find and flag issues related to gas optimizations that help in reducing the overall gas cost It scans and evaluates the codebase against industry best practices and standards to ensure compliance It makes sure that the officially recognized libraries used in the code are secure and up to date.

AUDIT SCORES

MH Audits AuditScores is not a live dynamic score. It is a fixed value determined at the time of the report issuance date.

MH Audits AuditScores are not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. These reports and scores are not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts MH Audits to perform a security review.

AUDITS

WEBSITE
MHAUDITS.IO

TWITTER
@MHAUDITS