AUDITS

SECURITY ASSESSMENT

AVASHARKS

JULY 30 TH 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

The AvaSharks are a collection of 2,605 NFTs, born and bred on the Avalanche Blockchain - first surfacing on December 3, 2021. Since then, the team has been busy building a grassroots community, as well as growing our network, within the Avalanche ecosystem.

As AvaSharks is the first wagering platform to enter this space, they are on their way to becoming a major player within the blockchain betting industry and transforming into a fully-operational online gaming platform.

Project Name *AvaSharks*

Contract Name -

Contract Address -

Contract Chain *Not Yet Deployed on Mainnet*

Contract Type *Smart Contract*

Platform *EVM*

Language *Solidity*

Codebase *GitHub Repository*

INFO & SOCIALS

Network *Avalanche (AVAX)*

Max Token Supply -

Website *https://avasharks.io/*

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

Telegram -

Discord *https://discord.gg/F3v8R55kpM*

AUDIT SCORES

Issues	10
Critical	0
Major	2
Medium	0
Minor	7
Informational	1
Discussion	0

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

AUDIT SCOPE

FILE	LOCATION
AtlantisGate.sol	*GitHub Repository*
NFTLender.sol	*GitHub Repository*

REVIEW METHODOLOGY

TECHNIQUES

This report has been prepared for AvaSharks to discover issues and vulnerabilities in the source code of the AvaSharks 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	2022/07/25
Description	Layout project Automated / Manual review / Static & dynamic security testing Summary
Version	v1.1
Date	2022/07/30
Description	Reaudit Final Summary

KEY FINDINGS

TITLE	SEVERITY	STATUS
Floating Pragma	Minor	*Partially Fixed*
Functions Should Be Declared External	Informational	*Fixed*
Missing Reentrancy Protections	Major	*Fixed*
Missing Multiple Zero Address Validations	Minor	*Fixed*
Missing Events	Minor	*Fixed*
Use Of Multiple Pragma Versions	Minor	*Fixed*
Missing Pausable Modifier	Major	*Fixed*
Missing Constant Attribute in Variables	Minor	*Fixed*
Redundant Statement	Minor	*Fixed*
Incorrect Placement Of require Statements	Minor	*Fixed*

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., ^0.8.4 *and* >=0.7.0 <0.9.0*. This allows the contracts to be compiled with all the solidity compiler version above* 0.8.4*.*

Location: *AtlantisGate.sol L02*
NFTLender.sol L02

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 therefore this is only informational.

Issue: *Floating Pragma*

Type: *Floating Pragma (SWC-103)*

Level: *Minor*

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: *The Avasharks team opted to consider our references and applied the recommendation.*

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: *AtlantisGate.sol*

deposit() - ***L43***
withdraw() - ***L48***
moveBalance() - ***L57***
changeWinningsFee() - ***L68***

NFTLender.sol

addListing() - ***L136***
cancelListing() ***L184***
fundListing() - ***L195***
withdrawBalance()- ***L215***
repayForListing() -***L221***
claimCollateralAsFunder() - ***L237***
setListingPrice() - ***L251***
withdrawToSafe() - ***L276***

Issue: *Functions Should Be Declared* External

Type: *Gas Optimization*

Level: *Informational*

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

Alleviation: *The Avasharks team opted to consider our references and applied the recommendation.*

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.*

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 functions making external calls

Location: *The function claimCollateralAsFunder() is making an external call on L244*

```
            ERC721(listing.nftContract).safeTransferFrom(address(this),
msg.sender, listing.tokenId);
```

After the call, the following state changes are occurring

```
                                idToListing[_listingId].status      =
ListingStatus.FUNDER_CLAIMED_COLLATERAL;
      idToListing[_listingId].datesInfo.funderClaimedDate = _now;
      _listingsRepaid.increment();
```

*If the user controls the address of the externally called address, i.e., "*listing.nftContract*", they might be able to reenter the function without the reentrancy guard and cause unexpected behaviour and token manipulation.*

Issue: *Missing Reentrancy Protections*

Type: *Reentrancy (SWC-107)*

Level: *Major*

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

Alleviation: *The Avasharks team opted to consider our recommendation and applied the recommendation.*

Affected Code:

```
    function claimCollateralAsFunder(uint256 _listingId) public {
        Listing memory listing = idToListing[_listingId];
        uint256 _now = block.timestamp;

            require(listing.fundedBy == msg.sender, "caller must be the
funder");
         require(listing.datesInfo.fundedDate + listing.repayDays < _now,
"loan repayment date must have passed");
```

```
             ERC721(listing.nftContract).safeTransferFrom(address(this),
msg.sender, listing.tokenId);

                                    idToListing[_listingId].status     =
ListingStatus.FUNDER_CLAIMED_COLLATERAL;
        idToListing[_listingId].datesInfo.funderClaimedDate = _now;
        _listingsRepaid.increment();
    }
```

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.

Description: *Multiple Solidity contracts were found to be setting 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 burned 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 forever.

Location: *AtlantisGate.sol*

address _adminAddress, address _serverWithdrawAddress, address _serverJudgeAddress ***- L20***
address payable _to ***- L48***
address _source, address _destination ***- L57***

NFTLender.sol
address _contractSafe ***- L63***
address _nftContract ***- L136***
address payable _destAddress ***- L215***

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 Multiple Zero Address Validations*

Type: *Missing Input Validation*

Level: *Minor*

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

Alleviation: *The Avasharks team opted to consider our recommendation and applied the recommendation.*

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.

Location: *AtlantisGate.sol*

withdraw() ***- L48***
moveBalance() ***- L57***
changeWinningsFee() ***- L68***

NFTLender.sol

cancelListing ***- L184***
setListingPrice ***- L251***
withdrawToSafe ***- L276***

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 Events*

Type: *Missing Best Practices*

Level: *Minor*

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

Alleviation: *The Avasharks team opted to consider our recommendation and applied the recommendation.*

Description:

The contracts were found to be using multiple Solidity Compiler versions across different solidity files. This is not a good coding practice because different versions of the compiler have different caveats, breaking changes and introducing vulnerabilities.

Location: *AtlantisGate.sol L02*
NFTLender.sol L02

Impacts:

Having different pragma versions across multiple contracts increases the chances of introducing vulnerabilities since each solidity version have their own set of issues and coding practices. Some major version upgrades may also break the contract logic if not handled properly.

Issue: *Use of Multiple Pragma Versions*

Type: *Missing Best Practices*

Level: *Minor*

Recommendation: *nstead of using different versions of the Solidity compiler with different bugs and security checks, it is better to use one version across all contracts.*

Alleviation: *The Avasharks team opted to consider our references and applied the recommendation.*

Description: *Openzeppelin's Pausable Library is used as a modifier to check if a contract is paused or not. This is typically used in contracts to protect sensitive functions in the case there's a malicious activity going on or if the contract is compromised by pausing the critical functions of the contract. The contracts were found to be missing a pausable modifier on business-critical functions which can cause state-changing actions on the smart contract if, during an attack, or a compromise, the contract is not paused.*

Location: *NFTLender.sol*

addListing ***- L136***
cancelListing ***- L184***
fundListing ***- L195***
withdrawBalance ***- L215***
repayForListing ***- L221***
claimCollateralAsFunder ***- L237***
setListingPrice ***- L251***
withdrawToSafe ***- L276***

Impacts: *Missing pausable modifier on sensitive functions may be abused in case a malicious actor is able to compromise the contracts or its functions. There needs to be a pausable modifier which can be used on sensitive functions to halt the contract flow.*

Issue: *Missing Pausable Modifier*

Type: *Missing Access Control*

Level: *Major*

Recommendation: *It is recommended to implement the* whenNotPaused *modifier on all the sensitive functions that deal with Ether or tokens or sensitive access roles and their modifications.*

Alleviation: *The Avasharks team opted to consider our recommendation and applied the recommendation.*

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: *AtlantisGate.sol L17*

```
uint256 public withdrawFee = 0.002 ether;
```

PoC:

*1/ Go to the contract "*AtlantisGate.sol*" and note the* uint256 withdrawFee *variable on L17. This is not being modified anywhere throughout the code.*

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: *Minor*

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

Alleviation: *The Avasharks team opted to consider our references and applied the recommendation.*

Description: *Solidity parameter type uint256 stores values from* 0 *to* 2 **256 - 1. *This means that it can never store negative values. This means there's no need to check if the parameter* _fees *can store values greater than or equal to zero.*

Location: require(_fee >= 0, "Winnings fee must be at least 0"); ***L70***

```
  function changeWinningsFee(uint256 _fee) public nonReentrant
whenNotPaused {
        require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender));
        require(_fee >= 0, "Winnings fee must be at least 0");
```

PoC:

1/ In the contract AtlantisGate.sol, it can be seen on L68 that the function changeWinningsFee *accepts a uint256 parameter* _fee*.*
2/ Since this will always take positive values, there's no need for the require statement.

Impacts:

This creates dead and redundant code and also increases gas costs.

Issue: *Redundant Statement*

Type: *Gas Optimization*

Level: *Minor*

Recommendation: *Remove the redundant require statement on L70 since* uint256 *can never be negative.*

Alleviation: *The Avasharks team opted to consider our references and applied the recommendation.*

Description: *The require statements on L200 and L203 in NFTLender.sol are not correctly arranged. These validations should happen before calculating the value for* profitFromInterest*.*

Location: require(msg.sender != listing.owner, "caller must not be listing owner");
require(listing.status == ListingStatus.LISTED, "listing status must be LISTED");

```
        uint256 profitFromInterest = (listing.repayPrice -
listing.price) * repaymentPercentageFee / uint256(100);

        require(msg.sender != listing.owner, "caller must not be
listing owner");
        require(profitFromInterest > 0, "profit from interest
must be greater than 0");
        require(msg.value >= listing.price + profitFromInterest,
"insufficient funds sent");
        require(listing.status == ListingStatus.LISTED, "listing
status must be LISTED");
```

Impacts: *Incorrect placement of require statements will cause the contract to execute unnecessary calculations for the parameter profitFromInterest. If the validations in the require statement happen in the beginning, the function will fail if improper values are supplied.*

Issue: *Incorrect Placement Of* require *Statements*

Type: *Business Logic*

Level: *Minor*

Recommendation: *Change the placement of the* require *statements and keep it above the parameter* profitFromInterest*.*

Alleviation: *The Avasharks team opted to consider our recommendation and applied the recommendation.*

Private GitHub Repository

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 AvaSharks project using the above techniques to examine and discover vulnerabilities and safe coding practices in AvaSharks' 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