HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Seaside Club
Date: August 19th, 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 Seaside Club
Approved By	Noah Jelich | Senior Solidity SC Auditor at Hacken OU
Type	ERC721 token; ERC721 token Marketplace
Platform	EVM
Network	Ethereum
Language	Solidity
Methods	Manual Review, Automated Review, Architecture review
Website	seaside.club
Timeline	29.06.2022 - 19.08.2022
Changelog	09.07.2022 - Initial Review 28.07.2022 - Second Review 19.08.2022 - 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 Seaside Club (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://gitlab.4irelabs.com/seaside/seas-contracts/
Commit:
0fd64ad585f3589c7ae93c962bedbe2b47370082
Technical Documentation:
Type: Whitepaper
Link

Type: Technical and functional requirements

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/ApexHumanity.sol

File: ./contracts/Marketplace.sol

File: ./contracts/helpers/MarketplaceSignature.sol

File: ./contracts/interfaces/IERC721.sol

File: ./contracts/mock/TestToken.sol

File: ./contracts/mock/chainLinkMock/LinkToken.sol

File: ./contracts/mock/chainLinkMock/MockOracle.sol

File: ./contracts/mock/chainLinkMock/MockV3Aggregator.sol

File: ./contracts/mock/chainLinkMock/VRFCoordinatorV2Mock.sol

File: ./contracts/mock/TestMarketplace/MockMarketplace.sol

File: ./contracts/mock/TestMarketplace/MockMarketplaceV2.sol

Second review scope

Repository:
https://gitlab.4irelabs.com/seaside/seas-contracts/
Commit:
e9249e0fbbc50001860290a38dcf2b6b441bbfd9
Technical Documentation:
Type: Whitepaper
Link

Type: Technical and functional requirements

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/ApexHumanity.sol

File: ./contracts/Marketplace.sol

File: ./contracts/helpers/MarketplaceSignature.sol

File: ./contracts/interfaces/IERC721.sol

File: ./contracts/mock/TestToken.sol

File: ./contracts/mock/chainLinkMock/LinkToken.sol

File: ./contracts/mock/chainLinkMock/MockOracle.sol

File: ./contracts/mock/chainLinkMock/MockV3Aggregator.sol

File: ./contracts/mock/chainLinkMock/VRFCoordinatorV2Mock.sol

File: ./contracts/mock/TestMarketplace/MockMarketplace.sol

File: ./contracts/mock/TestMarketplace/MockMarketplaceV2.sol

Third review scope

Repository:
https://gitlab.4irelabs.com/seaside/seas-contracts/
Commit:
e3256b33cec0012ed2f9309a5ae866115b4b9c07
Technical Documentation:
Type: Whitepaper
Link

Type: Technical and functional requirements

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/ApexHumanity.sol

File: ./contracts/Marketplace.sol

File: ./contracts/helpers/MarketplaceSignature.sol

File: ./contracts/interfaces/IERC721.sol

File: ./contracts/mock/TestToken.sol

File: ./contracts/mock/chainLinkMock/LinkToken.sol

File: ./contracts/mock/chainLinkMock/MockOracle.sol

File: ./contracts/mock/chainLinkMock/MockV3Aggregator.sol

```
File: ./contracts/mock/chainLinkMock/VRFCoordinatorV2Mock.sol

File: ./contracts/mock/TestMarketplace/MockMarketplace.sol

File: ./contracts/mock/TestMarketplace/MockMarketplaceV2.sol
```

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

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 total Documentation Quality score is **10** out of **10**. Functional requirements and technical description were provided.

Code quality

The total CodeQuality score is **9,5** out of **10**. Code partially follows the official guidelines, the unit tests were provided.

Architecture quality

The architecture quality score is **8** out of **10**. The architecture is clear. The configured development environment was provided. However, the decision to mint through the Marketplace.sol instead of directly increases Gas costs.

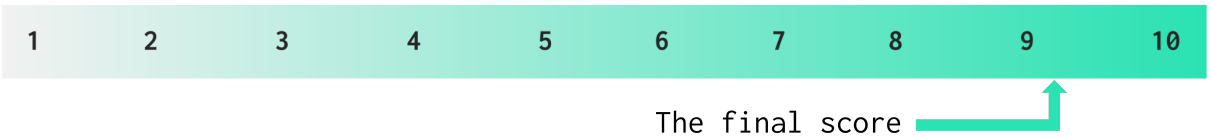
Security score

As a result of the audit, the code contains **1** medium, and **1** low severity issues. 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: **9.05**.



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.	Passed
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.	Failed
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 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.	Passed
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Failed
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	Passed
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.	Passed
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.	Passed
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 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.	Passed
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.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

System Overview

SeaSide Investment Club is an ERC-721 selling system with the following contracts:

- *ApexHumanity* — is an ERC-721 token with ERC2981 support. Token name and symbol are defined when deploying.
- *Marketplace* - is an NFT Marketplace contract with 10010 tokens available for selling. There are two phases of selling:
    - Whitelist sale: users should provide the signature, and the presale period should not be exceeded. The maximum amount of tokens that can be bought by the user per presale is 2.
    - Public sale: the presale phase should be finished. The maximum amount of tokens that can be bought by the user per public sale is 3.

  There is the blacklist functionality. The token price is 0.2 ETH. The owner can mint tokens after the presale period exceeds (within the limit).
  After all the tokens are sold and minted, the lottery is triggered by the "OWNER_MARKETPLACE_ROLE". There are two phases of the lottery:
    - topDraw: 110 tokens are the winners. Their URIs are changed with the ones defined by "OWNER_MARKETPLACE_ROLE". The randomness is implemented with Chainlink Random. The random values are requested before the lottery.
    - bronzeDraw: is executed after the topDraw is finished. All the remaining token URIs are changed with the ones defined by "OWNER_MARKETPLACE_ROLE".

  The contract has a pausing functionality: token selling may be stopped and unstopped.
- *MarketplaceSignature* - is a contract for the signature verifications inherited by the *Marketplace* contract.
- *IERC721* - is an interface for the ERC-721 tokens used in *Marketplace* contract.
- *MockMarketplace, MockMarketplaceV2, MockOracle, TestToken, LinkToken, MockV3Aggregator, VRFCoordinatorV2Mock* - are the testing contracts.

Privileged roles

- The *ApexHumanity* contract has privileged roles of the owner and "OWNER_MARKETPLACE_ROLE":
    - The owner can set and update royalty information: receiver address and the fee; tokens base URI.
    - The "OWNER_MARKETPLACE_ROLE" can mint tokens, set and update token URIs.

- The *Marketplace* contract has privileged roles of the "OWNER_MARKETPLACE_ROLE":
    - The "OWNER_MARKETPLACE_ROLE" can set and update the token, request random values from Chainlink, trigger the lotteries execution, add and remove users from blacklist, pause and unpause tokens selling, mint tokens during the public sale phase, withdraw ETH and ERC-20 tokens from the contract, increase LINK tokens for the Chainlink, cancel the Chainlink subscription.
- The *MockMarketplace* and *MockMarketplaceV2* contracts have the same privileged roles as the *Marketplace* contract does.

Risks

- The statuses (URIs) of tokens that won the lottery and those that did not are determined by the user with the "OWNER_MARKETPLACE_ROLE" role at the time of the lottery and cannot be verified.
- The "OWNER_MARKETPLACE_ROLE" role allows mint tokens that will participate in the lottery, pause and unpause tokens selling, and change the marketplace token.

Findings

Critical

1. Requirements violation

According to the documentation, the distribution of lottery prizes is as follows:

- 10 users will become winners of additional presents (flat, money etc).
- 10 users will get Diamond status
- 20 users will get Golden status
- 70 users will get Silver status

According to an addendum provided regarding changes in requirements:

- 20 users will get Diamond status
- 20 users will get Golden status
- 20 users will get Silver statuses

The “topDraw” function sets the URIs to 110 winners.

There are more winners than the total number of required winners, according to the addendum with the latest updates (60).

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: topDraw

Recommendation: Implement the code according to requirements, add the latest requirements to the documentation.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

2. Ambiguous third-party integration

According to the documentation, the contracts will be deployed to the Ethereum network, but the “keyHash” Chainlink value is hardcoded with the value available for Rinkeby testnet. (0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc)

Therefore, the current Chainlink random implementation will not function on the Ethereum network.

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: requestRandomNumbers

Recommendation: Use the available Gas lane (Key Hash) on Ethereum network: Contract Addresses | Chainlink Documentation. Do not hardcode the value, convert it into constant.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

3. **Denial of Service vulnerability**

There is a defined Gas value in the loop for minting (50000). This value is insufficient to cover the amount required to complete the transaction. The “should successfully execute bronze random” test fails with the “TransactionExecutionError: Transaction ran out of Gas” error.

Therefore, setting "Bronze" status to the tokens is impossible as the transaction always fails.

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: bronzeDraw

Recommendation: Increase the defined Gas value, do not hardcode it as the Gas costs of opcodes can change in the future. Convert the value into a changeable variable or function parameter.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

High

1. **Denial of Service vulnerability**

The `transfer` function has a built-in Gas limit.

Execution will fail if the receiver is a contract with fallback functionality.

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: unlockETH

Recommendation: Transfer using “call”.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

2. **Denial of Service vulnerability and silent execution fail**

The Gas values are hardcoded for the “bronzeDraw”, “ownerPurchase” and functions use “gasLeft”.

Execution will silently fail if there is not enough Gas.

The Gas value is hardcoded for “requestRandomNumbers”.

If the Gas costs of opcodes increases in the future, the “bronzeDraw”, and the callback from Chainlink will fail, the “ownerPurchase” function will fail to start with big enough “quantity”.

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: bronzeDraw

Recommendation: Do not use “gasleft” for the loop condition. Use the number of iterations that can be changed. Do not hardcode Gas values, convert them into changeable variables or function parameters.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

3. **Undocumented behavior**

The “OWNER_MARKETPLACE_ROLE” role allows mint tokens that will participate in the lottery, pause and unpause tokens selling, and change the marketplace token.

The contract has a blacklist functionality. The comment in the code: “On the backend we use a service for AML policy. And if the address is not recommended by AML, we add this address to the blacklist”. (blacklisted users can participate in the public sale and then in the lottery.)

The code should not contain undocumented functionality. The existence of such functionality can lead to unexpected behavior of the contract.

Contract: Marketplace

Path: contracts/Marketplace.sol

Functions: setFreeze, ownerPurchase, init

Recommendation: Remove undocumented functionality or add it into the documentation. Clarify the blacklist functionality rules in the documentation.

Status: Mitigated. The functionality description was added to the documentation.

4. **Denial of Service vulnerability**

The functionality allows the user with “OWNER_MARKETPLACE_ROLE” to cancel the subscription, after which subscription creation is impossible.

If the subscription is canceled before the needed random values are obtained from Chainlink, the lottery will not be possible to execute.

Contract: Marketplace

Path: contracts/Marketplace.sol

Function: cancelSubscription

Recommendation: Check if the required amount of random values for the lottery is obtained before the subscription canceling or add the ability to create new subscriptions.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

5. **Data consistency**

“tokenURI” function calls the function of the same name from the ERC721URIStorage contract, which calls the “_baseURI()” function from the ERC721 contract. “_baseURI()” is empty by default.

Therefore, base URI is not concatenated with token URI in the “tokenURI” function.

When a token is minted, the “_baseTokenURI” is set as the token URI value.

Thus, the incorrect token URI will be returned if changing base URI or the concatenation with base URI works.

Contract: ApexHumanity

Path: contracts/ApexHumanity.sol

Functions: tokenURI, mint

Recommendation: Override “_baseURI” function from ERC721 function with `return _baseTokenURI`, do not set the “_baseTokenURI” as the token URI when minting.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

6. **Requirements violation**

According to the documentation, the royalties must be implemented (royalty 7.5% of ETH price). The ApexHumanity contract implements ERC2981 standard and has the functionality for the royalty information (receiver, fee) defining, but the royalty payment is not enforced.

Thus, the royalty receiver never gets the royalties.

Contracts: ApexHumanity, Marketplace

Paths: contracts/ApexHumanity.sol, contracts/Marketplace.sol

Functions: ApexHumanity.setRoyaltyInfo, Marketplace.purchase

Recommendation: Pay royalties together with sale in the token marketplace (Marketplace.purchase).

Status: Mitigated. The royalty will be paid in the secondary marketplaces. The information from the documentation: “After sale of NFT at secondary market royalty of 7.5% deducted from sell price and sent to the contract creator address”.

7. **Denial of Service vulnerability**

The “topDraw” function uses random values for choosing the winners' IDs; the “bronzeDraw” function checks if there are 110 winners; if not, the function reverts.

Therefore, if there are not enough unique random values, there will be fewer winners than required, and the “bronzeDraw” function will be inoperable.

Contract: Marketplace

Path: contracts/Marketplace.sol

Functions: topDraw, bronzeDraw

Recommendation: Ensure that there will be the required number of winners.

Status: Fixed (Revised commit: e3256b33cec0012ed2f9309a5ae866115b4b9c07)

Medium

1. **Hardcoded values**

The token price is *0.2 Ether,* and this value is constant. (“PRICE”)

If the price of the native currency goes up or down, such value may not be relevant.

Path: contracts/Marketplace.sol

Contract: Marketplace

Recommendation: Allow changing the price by an admin account.

Status: Mitigated. The Customer comment: “Price is hardcoded as per clients request”.

2. **Redundant random values requests**

The 140 random values are obtained from Chainlink per 2 requests.

30 values are redundant, as only 110 random values are used. Creating multiple requests increases the amount of Gas.

Path: contracts/Marketplace.sol

Contract: Marketplace

Function: requestRandomNumbers

Recommendation: Request the required number of random values per one request as it is less than 500 (Chainlink limit), do not hardcode the value, convert it into constant. If the increase of winners is implied for future sales, divide the request into chunks with 500 random values.

Status: Mitigated. The Customer`s comment: “I take 140 random numbers, as Chainlink sometimes sends the same numbers and 140 will be enough for a 100% guarantee that all numbers will be unique. Same way. I do this with two requests of 70, since the callback is only 2.5 million available. And this is not enough to save 140 uint256 in storage at a time.”

3. **Checks-Effects-Interaction pattern violation**

The state variables are updated after the external calls.

This can lead to reentrancies, race conditions, or denial of service vulnerabilities.

Path: contracts/Marketplace.sol

Contract: Marketplace

Functions: topDraw, bronzeDraw, ownerPurchase, purchase, cancelSubscription

Recommendation: Implement functions according to the Checks-Effects-Interactions pattern.

Status: Reported

4. **Failing tests**

The “should successfully execute bronze random” test is failing with the “Headers Timeout Error” error.

Recommendation: Ensure that all the test cases are passing.

Status: Fixed (Revised commit: e3256b33cec0012ed2f9309a5ae866115b4b9c07)

Low

1. **Never emitted event**

Event “SetLocked” is never emitted.

Path: contracts/ApexHumanity.sol

Contract: ApexHumanity

Recommendation: Add the tokens locking according to the requirements and emit the event.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

2. **Use of hard-coded values**

The parameters for "__Signature_init" in the constructor are hardcoded.

Using hardcoded values is not a good practice.

Path: contracts/Marketplace.sol

Contract: Marketplace

Recommendation: Convert the hardcoded values into the constructor parameters.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

3. **Redundant check**

Checking if the total token supply is equal to the "MAX_PARTICIPANTS_NUM" is redundant as it is already checked in the "topDraw" phase.

Path: contracts/Marketplace.sol

Contract: Marketplace

Function: bronzeDraw

Recommendation: Remove the redundant check.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

4. **Boolean equality checks**

Boolean constants can be used directly and do not need to be compared to true or false.

Path: contracts/Marketplace.sol

Contract: Marketplace

Functions: topDraw, purchase

Recommendation: Remove the redundant checks.

Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

5. **Block values as a proxy for time using**

   The contract uses block.timestamp for time calculations. It is not precise and safe.

   Path: contracts/Marketplace.sol

   Contract: Marketplace

   Functions: constructor, ownerPurchase, purchase,

   Recommendation: It is recommended to avoid using block.timestamp. Alternatively, it is safe to use oracles.

   Status: Reported

6. **Default visibility usage**

   There are variables in the contract whose visibilities are not defined explicitly.

   This may lead to incorrect access to variables.

   File: ./contracts/helpers/MarketplaceSignature.sol

   Contract: MarketplaceSignature

   Paths: EIP712DOMAIN_TYPEHASH, SIGNDATA_TYPEHASH, EIP712DOMAIN_SEPARATOR

   Recommendation: Define the variables' visibilities explicitly.

   Status: Fixed (Revised commit: e9249e0fbbc50001860290a38dcf2b6b441bbfd9)

7. **Outdated Solidity version usage**

   Using an outdated compiler version can be problematic, especially if publicly disclosed bugs and issues affect the current compiler version. The project uses compiler versions ^0.4.24 and ^0.6.6.

   Files: ./contracts/mock/chainLinkMock/LinkToken.sol, ./contracts/mock/chainLinkMock/MockV3Aggregator.sol, ./contracts/mock/chainLinkMock/MockOracle.sol

   Recommendation: Use a contemporary compiler version.

   Status: Mitigated. The Customer`s comment: “I cannot change these contracts, they are used for Mock contracts.”

8. **Floating pragma**

   Contracts with unlocked pragmas may be deployed by the latest compiler, which may have higher risks of undiscovered bugs.

Files: ../contracts/mock/chainLinkMock/LinkToken.sol, ./contracts/mock/chainLinkMock/MockV3Aggregator.sol, ./contracts/mock/chainLinkMock/MockOracle.sol, ./contracts/mock/chainLinkMock/VRFCoordinatorV2Mock.sol

Recommendation: Consider locking the pragma version whenever possible.

Status: Mitigated. The Customer`s comment: “I cannot change these contracts, they are used for Mock contracts.”

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.