SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Angel Block
Date: 18 May, 2023

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Angel Block
Approved By	Marcin Ugarenko | Lead Solidity SC Auditor at Hacken OU
Type	Staking
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://angelblock.io/
Changelog	10.04.2023 - Initial Review 18.05.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Angel Block (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 includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/angel-block/angelblock-contracts
Commit	ea726f7
Whitepaper	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/
Functional Requirements	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/
Technical Requirements	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/
Contracts	File: ./contracts/staking/AbstractStaking.sol SHA3: 78fddfe57d130b80d28d2367c0766bb46bb86926c7039118c8021cd9c62eec57 File: ./contracts/staking/NFTDataOperator.sol SHA3: 3ecf9e322ea6eed29ab0a2728be4a2c084693dca7f1399ea713d95112d08290b File: ./contracts/staking/TholosStaking.sol SHA3: 4efa4fe71acbcbac5ec6e248f0cbd77c2cc1423aa810eafce8f0f7599a5537cf File: ./contracts/pools/AbstractPool.sol SHA3: ba714443e71fe30e866c598cbc5db9bb00ed3bd6d644dcf394178d0cbbf19286 File: ./contracts/pools/DepositPool.sol SHA3: 8d58e9125c0bbe6a6309b1fe6daa446b6c88412440d94a2406d30de87cf8cf28 File: ./contracts/utils/Configurable.sol SHA3: 77c4c02f76ae5a3e60fad15fc0b3f105e60cc98f3cca83b59834d7a2c8f1aacb File: ./contracts/interfaces/INFTDataOperator.sol SHA3: 8286505c002b73742d63a99d0f11466cd8e51443e048ed131f6959b35597accf File: ./contracts/interfaces/IPool.sol SHA3: 8c4ce6698e2ed8c36cde856fdf97c279244f92e5293e3579b04150be34a13902 File: ./contracts/interfaces/IPool721.sol SHA3: 78f30fa1dd6ff1932ee775e1a0e9e68fcbcdc3d224d9de8dee17de6f31690178 File: ./contracts/interfaces/IStaking.sol SHA3: 0a630fa3048ffd1c7982cb935b9d0b47ff0673d7a7dbfb20e101f6d9ee4db6da File: ./contracts/interfaces/ITholosStaking.sol

	SHA3: e85a7c6d32ad617bb545660c8b303982312e60f3ae1143daddadb8dd24731109

Second review scope

Repository	https://github.com/angel-block/angelblock-contracts
Commit	90c3de501
Whitepaper	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/
Functional Requirements	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/ https://angelblock.io/blog/introducing-thol-and-nft-staking/ https://github.com/angel-block/angelblock-contracts/blob/master/docs/contracts/README.md https://github.com/angel-block/angelblock-contracts/blob/master/docs/contracts/staking/README.md https://github.com/angel-block/angelblock-contracts/blob/master/docs/contracts/pools/README.md
Technical Requirements	https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/
Contracts	File: ./contracts/staking/AbstractStaking.sol SHA3: 88cabb5446fd10b81b6c74dcaf9e9fe3c8166ce0db0b7616d2d8d15816c9c180 File: ./contracts/staking/NFTDataOperator.sol SHA3: 8c0c203d1f08cc025307620138690e676e083b4c0993a579970949d65e2c1b0f File: ./contracts/staking/TholosStaking.sol SHA3: 9549f225e8f463aef37848e9de126ad520d63525f43538959c2b7f9bc26e9846 File: ./contracts/pools/AbstractPool.sol SHA3: c0d17f81437b3dca70e068e20203b98814298e8670cb276d32deca87987fe071 File: ./contracts/pools/DepositPool.sol SHA3: 4b61b58f2d91aff48035e0b397c0042c67c6d4d309633a821ea6e07cc148eaf5 File: ./contracts/utils/Configurable.sol SHA3: cea2b2e858ad5a382779e3931cf730d236a53493b77409fed32cab3588ff4473 File: ./contracts/interfaces/INFTDataOperator.sol SHA3: ef9ccb15961ee32a02b7f692ecece502c1cf5e491aef78acb77e5f6b722d8a09 File: ./contracts/interfaces/IPool.sol SHA3: 3248617e1455b9c78de4de9b7fab7a942f4686e9f4c849d676314004de065943 File: ./contracts/interfaces/IPool721.sol SHA3: 654ca7211f426dabb64afab0006396863fafc914c6911d6060056a0119bfc6ca File: ./contracts/interfaces/IStaking.sol SHA3: f5234a40d2267778f584e330544bdc6893a3623c78b323a422dbfde6ce30351f File: ./contracts/interfaces/ITholosStaking.sol SHA3: 43fcad84166fbc3fd945438f5b51d49ee5c49f49e3b3cedf783d0860a9c1686a

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to asset loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect code quality

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided and detailed.
- Technical description is sufficient:
  - NatSpec is provided and sufficient.
  - Run instructions are provided.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- The code is well organized and follows best practices.

Test coverage

Code coverage of the project is **100.0%** (branch coverage).

- Test coverage is sufficient.

Security score

As a result of the audit, the code contains **2** low severity issues. The security score is **10** 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: **10**.

The system users should acknowledge all the risks summed up in the risks section of the report.

1
2
3
4
5
6
7
8
9
10
The final score

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
10 April 2023	12	1	2	0
18 May 2023	2	0	0	0

Risks

- The *TholosStaking* smart contract uses a queue to handle unstake requests. If the unstake queue size equals ten (10), it is impossible to withdraw assets.
- The deposit and withdrawal of ERC721 assets are only supported for the EOA wallets. In case of using Multi-sig or other smart contracts for interaction with staking contracts, it is a user responsibility to check if ERC721 tokens are supported.
- The vulnerability in AbstractStaking.sol could manifest when the rewardPool's balance is depleted or its transfer allowance is insufficient, potentially preventing users from withdrawing their funds. Additionally, this system presents a risk of gas inefficiency due to the unnecessary storage updates and token transfers in its design.

System Overview

AngelBlock is a non-custodial, protocol based fundraising infrastructure that allows to conduct token based raises in a more transparent, decentralized, and democratized manner.

The audit is focused on the Staking part of the system. It consists of following contracts:

- *AbstractPool* - an abstract smart contract which basically describes the pool for accumulating deposited funds.
- *DepositPool* - a pool smart contract inherited *AbstractPool*, which stores deposited assets and ERC721(NFTs) from staking. Tokens and NFTs are part of the ecosystem and are clearly defined in the smart contract. A smart contract has a keeper who can withdraw funds from the smart contract.
- *AbstractStaking* - an abstract smart contract which basically describes a staking mechanism. It is available to update the state of the contract based on the compounding interest and rewards. It ensures that the compounding process only happens if there are sufficient rewards and an hour has passed since the last compounding action. As a deposit are accepted ERC20 and ERC721(NFTs) tokens.
- *TholosStaking* - a staking smart contract inherited *AbstractStaking.* It has extended functionality for calculating rewards. Unstaking is only possible after a 10 day period after the unstake request.
- *NFTDataOperator* - a smart contract calculates the ERC20 tokens equivalent for a given amount of WETH tokens based on the current and previous NFT collection floor prices and volumes. It also ensures that the calculated ERC20 per ERC721(NFTs) value is within the allowed local cap range. The contract allows updating the local cap values and is configurable by the owner..
- *INFTDataOperator* - an interface for *NFTDataOperator* smart contract.
- *ITholosStaking* - an interface for *TholosStaking* smart contract.
- *IStaking* - an interface describes basic staking functions. *TholosStaking* smart contract inherits this interface.
- *IPool* - an interface for the deposit pool. Describes ERC20 interactions.
- *IPool721* - an interface for the deposit pool. Describes ERC721(NFTs) interactions.

Privileged roles

- The `*keeper*` of the *DepositPool* smart contract is able to withdraw native ERC20 and ERC721(NFTs) tokens from the balance of the smart contract.
- The `*owner*` of the *NFTDataOperator* is available to change cap range.

- The `*admin*` of the *TholosStaking* is available to configure the state of the smart contract.
- The `*manager*` of the *TholosStaking* is available to set *maxNftRewardCap* value.
- The `*nft operator*` of the *TholosStaking* is available to set the rate between ERC20 native token and ERC721(NFT).
- The `*upgrader*` of the *TholosStaking* is available to upgrade the smart contract.

Recommendations

- Provide more documentation and explanation of the project's technical part.

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items 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.	Not Relevant
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.	Passed
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.	Passed
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed

Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Not Relevant
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	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 or be locked on the contract.	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.	Not Relevant
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 sufficient, 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, which may be changed in the future.	Passed

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. Denial Of Service

The *rewardPool* value is an address of a smart contract or EOA wallet with some balance of THOL ERC20 tokens. The flow of the staking system is dependent on the balance of this address and the allowance given to the staking contract.

There are many places in the staking system where this dependency leads to a Denial of Service vulnerability, in which users, in the worst case, are unable to withdraw deposited funds.

When the *compound()* function is performed, the balance of the *rewardPool* address is checked and rewards are calculated based on its value. Unfortunately, rewards are not transferred to the *depositPool* during *compound()* execution, but are only stored virtually in the *pendingSum* variable.

If the THOL balance of the *rewardPool* is depleted by actors other than the staking contract, there will be no tokens to transfer for the execution of the *accrue()* internal function, which will block the *deposit()* and *withdraw()* external functions.

Additionally, if the allowance to transfer THOL tokens from the *rewardPool* is dropped or insufficient, the same issue will happen.

The design choice to transfer the user pending rewards during the *accrue()* internal function is Gas-inefficient for the users, as many unneeded storage updates and token transfers are made.

Path:
./contracts/staking/AbstractStaking.sol : compound(), deposit(), withdraw()

Recommendation: Rewards calculated during the execution of the *compound()* function should be transferred to the *depositPool* at the end of that execution. The transfer of rewards tokens in the *accrue()* internal function should be removed.

This will prevent Denial of Service on the user-facing functions *deposit()* and *withdraw()*.

The lack of rewards in the *rewardsPool* or missing approval will only affect the *compound()* function, but this will be desired and expected.

The increment of the *depositSum* should be done when rewards are added, and the use of *pendingSum* can be omitted as it will not be needed.

Found in: ea726f7

Status: Mitigated (with Customer notice:

That's intentional security design. rewardPool is a Gnosis Multisig 3 out of 5, that we are regularly using and maintaining appropriate level of allowance. We plan on monthly basis to extend allowance for more rewards needed for staking contract.

Denial of Service will never happen, since Gnosis Multisig will always own 28M tokens and we will even top up the balance with earned $THOL from Treasury Multisig. When we will reach around 10-15% of free tokens available for rewards we will introduce governance token (xTHOL, planned for future) to ensure staking is able to offer rewards afterwards.) (Revised commit: 90c3de501)

H02. Invalid Calculations; Requirements Violation

Rewards from staking are not occurring passively; they are calculated on demand using the *compound()* function.

There is a flawed logic/invalid design inside the *compound()* function.

For users to receive the desired APY, the function needs to be called every hour. When called at larger intervals (e.g. 2 hours, 24 hours), the rewards will only be calculated for one hour.

This design leads to enormous costs (e.g., if each transaction costs $10, the year of compounding will result in $88,000 in spending) of maintaining the "auto-compounding" requirement from the documentation:

"Stakers will periodically receive auto-compounded $THOL"

In addition, if *withdraw()* is done without performing the compounding before, the user will not receive any rewards that have occurred in the last period.

Path:
./contracts/staking/AbstractStaking.sol : compound()

Recommendation: Re-examine the auto-compounding design, add the passage of time to the rewards calculation, and calculate the accrued rewards for users when withdrawing.

Found in: ea726f7

Status: Mitigated (The code was updated to a reasonable value of 24 hours, and based on the on-chain activity, the project is calling the compound() function at that interval.

Based on the changes and current price of calling the compound() function by the protocol, the costs were reduced approximately by 95%.) (Revised commit: 90c3de501)

■■ Medium

M01. Requirements Violation

In the project documentation:

https://angelblockprotocol.gitbook.io/angelblock-protocol-overview-documentation/staking-mechanism-and-implications

It is stated that “AngelBlock does not plan to lock, limit or take fees on staked goods.”, however, there is a 10-day lock mechanism on funds withdrawal.

Path:
./contracts/staking/TholosStaking.sol : _requestUnstake()

Recommendation: Consider following the requirements or updating the documentation.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

■ Low

L01. Solidity Style Guide Violation

The layouts of the *AbstractPool*, *DepositPool*, *AbstractStaking*, *TholosStaking*, *NFTDataOperator* contracts violate the order of functions convention.

Path:
./contracts/*

Recommendation: Follow the official Solidity code style guide.

Found in: ea726f7

Status: Mitigated (The project uses its own layout, but it is clean and easily readable.) (Revised commit: 90c3de501)

L02. Missing Zero Address Validation

Address parameters are used without checking against the possibility of 0x0. This issue is found in constructors and methods of every file in the audit scope.

Path:
./contracts/*

Recommendation: Implement zero address checks.

Found in: ea726f7

Status: Mitigated (Not all zero address checks were implemented, but contracts were correctly deployed and configured.) (Revised commit: 90c3de501)

L03. State Variables Default Visibility

The contract should specify a visibility level for all functions and state variables. The state variable *unstakeQueue* has a default visibility.

Path:
./contracts/staking/TholosStaking.sol : unstakeQueue

Recommendation: Specify variables as public, internal, or private. Explicitly define visibility for all state variables.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

L04. State Variables That Can Be Declared As Immutable

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

This will lower the Gas taxes.

Paths:
./contracts/pools/AbstractPool.sol : erc20, keeper
./contracts/pools/DepositPool.sol : nft
./contracts/staking/NFTDataOperator.sol : staking

Recommendation: Declare mentioned variables as immutable.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

L05. Typo in Comments

There are multiple spelling errors in the comments:

fullfiled -> fulfilled
begining -> beginning
compouund -> compound
calcuate -> calculate
necesarry -> necessary

Paths:
./contracts/utils/FixedSizeQueue.sol
./contracts/staking/AbstractStaking.sol
./contracts/staking/NFTDataOperator.sol

Recommendation: Spellings should be fixed.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501).

L06. Redundant Code

In the ITholosStaking.sol interface, the *NotStakedNFT* error is declared but is never used in the code.

Path:
./contracts/interfaces/ITholosStaking.sol : NotStakedNFT

Recommendation: Consider removing redundant code for better readability.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501).

L07. Gas Optimization

The *ts* member of the *Compounding* struct can be a smaller uint size and be packed together with the *freeRewards* member.

Path:
./contracts/interfaces/IStaking.sol : Compounding

Recommendation: Consider packing the Compounding struct more efficiently.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

L08. Code Consistency

It is best practice to write code uniformly.

The data emitted in the *UnstakeRequested* event is inconsistent with that emitted in the *UnstakeClaimed* event.

Path:
./contracts/interfaces/ITholosStaking.sol : UnstakeRequested, UnstakeClaimed

Recommendation: Consider including *address indexed sender* also in the UnstakeRequested event.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

L09. OpenZeppelin Deprecated Function

The AccessControl OpenZeppelin contract's *_setRole()* function is deprecated in favor of the *_grantRole()* function.

Paths:
./contracts/staking/AbstractStaking.sol : constructor()
./contracts/staking/TholosStaking.sol : configure()

Recommendation: Consider updating the said function.

Found in: ea726f7

Status: Fixed (Revised commit: 90c3de501)

L10. Code Clarity

The place where the *_requestUnstake()* function is called creates confusion, similar to the missing interaction with ERC20 tokens in the *_withdraw()* function.

Path:
./contracts/staking/TholosStaking.sol : _withdraw(), _decreaseBalance()

Recommendation: Consider moving the *_requestUnstake()* function call from the *_decreaseBalance()* to the *_withdraw()* function to increase the readability of the code and the flow of funds.

Found in: ea726f7

Status: Reported (Code was not changed) (Revised commit: 90c3de501)

L11. Unchecked Transfer

In the *_deposit()* and *_rewardPoolWithdraw()* functions, the return value of the *.transferFrom()* function calls is not checked.

Tokens may not follow ERC20 standard and return false in case of transfer failure or not returning any value at all.

Even when interacting with your own token, it is best practice to use SafeERC20 library.

Paths:
./contracts/staking/TholosStaking.sol : _rewardPoolWithdraw()
./contracts/staking/TholosStaking.sol : _deposit()

Recommendation: Use *SafeERC20* library to interact with tokens safely.

Found in: ea726f7

Status: Mitigated (Contracts only operate with the THOL token, which is a correct ERC20 token, and will revert on a failed transferFrom call.) (Revised commit: 90c3de501)

L12. Gas Optimization

Inside the *deposit()* function in the *AbstractStaking.sol* contract, *balances[_account]* is updated two times. First update occurs in *accrue()* and second update occurs in *_increaseBalance()*. Additionally, only the first value of the Balance struct should be updated second time, *rate* and *extraRate* will have the same value during *deposit()* execution.

Similar situation occurs in the *withdraw()* function, *Balance* struct for given address is updated 2 times in *withdraw()* execution.

Path:
./contracts/staking/AbstractStaking.sol : deposit(), withdraw()

Recommendation: Update members of the *balances[_account]* only once during *deposit()* and *withdraw()* execution.

Found in: ea726f7

Status: Reported (Double time state variable updating left in the code) (Revised commit: 90c3de501)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing 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 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, the Consultant cannot guarantee the explicit security of the audited smart contracts.