BLOCKAPEX

SMART CONTRACT SECURITY

v 1.0

Date: 20-03-2025

Prepared For: Nero Chain - Final Audit Report

About BlockApex

Founded in early 2021 BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

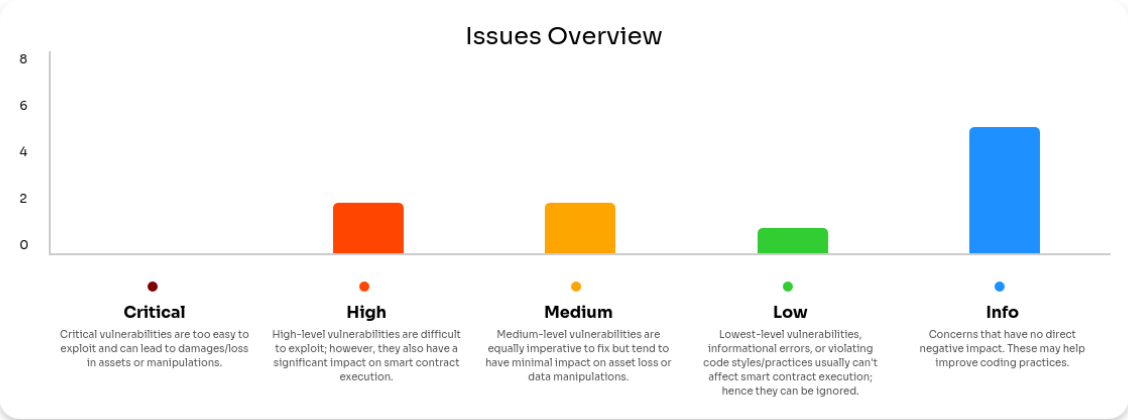
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

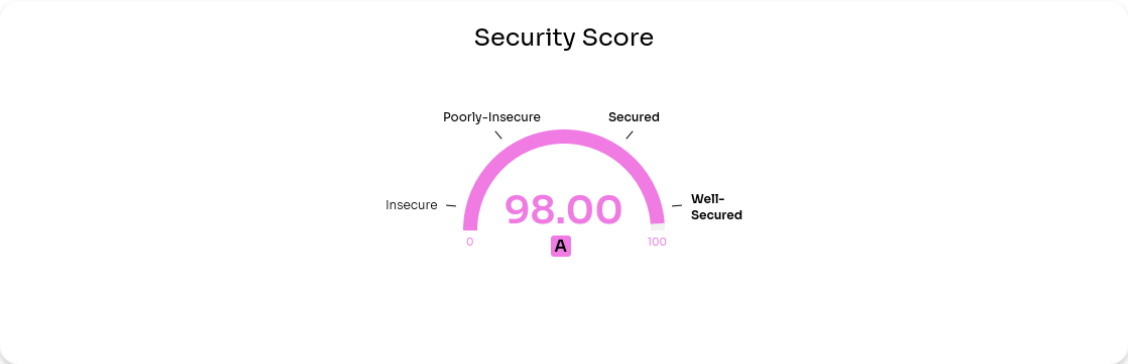
To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

Our team performed a technique called Filtered Audit, where two individuals separately audited the NeroChain System and paymaster Smart Contracts. After a thorough and rigorous manual testing process involving line by line code review for bugs, an automated tool-based review was carried out. All the raised flags were manually reviewed and re-tested to identify any false positives.





1.1 Scope

1.1.1 In Scope

This audit covers the Genesis Lock Staking and paymaster modules within NeroChain's ecosystem, essential for securing and managing token allocations and validator staking.

Genesis Lock Module

The Genesis Lock module establishes a structured token release mechanism for various user types, including early investors, partners, team members, and rewards. This module provides users with different locking periods, tailored to their specific allocations, and a way to claim unlocked tokens as they become available. The audit focuses on verifying the proper implementation of lock periods, user-specific allocations, and a secure claiming process, ensuring that users can only claim what they are entitled to based on the timing and conditions specified by NeroChain's lock-up policies.

Staking Module

The Staking module is designed to manage validator and delegator participation in network consensus. This module allows validators to register, maintain stakes, and earn rewards while securing the network. Delegators can contribute to validator stakes, sharing in the rewards and reinforcing the network's security. The audit examines the staking process, reward distribution, validator penalties, and the mechanisms ensuring secure delegation. Additionally, it assesses how the module handles validator and delegator interactions, including rewards, withdrawals, and penalizations for non-compliance with consensus requirements, ensuring integrity in staking operations and alignment with NeroChain's security goals.

NeroPaymaster Module The NeroPaymaster serves as a versatile gas sponsorship tool that enables developers to sponsor user gas costs in NERO tokens or specified ERC20 tokens. Developers can generate a unique project ID on the platform's backend, register it with the Paymaster contract, and deposit NERO as collateral. Users in developer-backed DApps have flexible payment options: they can opt for free gas, post-paid ERC20, or prepaid ERC20 gas payments. The Paymaster contract validates payment parameters, allowing developers to set custom sponsorship policies and accepted token types. The audit will verify the security of the payment process, the proper handling of token transfers, and the conditions surrounding sponsorship policies to ensure that user gas payments are accurately processed, reflecting the settings developers specify.

Contracts in Scope:

1. system-contracts/contracts/*.sol
2. nerochain/aa-paymaster-contracts/contracts/*.sol

Initial Commit Hash:

- 9e2e3679572689dc6fc80e7438bbb5781b5656ab
- a896e4bfccf7045d772aa836565d1384f5f80717

Fixed Commit Hash: 2962463954ff475691c87c2c9ee14ae4b8f9e4b3

1.1.2 Out of Scope

All features or functionalities not specified in the “In Scope” section are considered outside the boundaries of this audit. This exclusion primarily applies to backend operations and infrastructure related to the NeroChain platform. Specifically, the consensus engine for system contracts, the bundler, entry point, backend management of project IDs, and all backend processes supporting the Paymaster functionality are outside the audit’s purview. Any integrations or external services linked to these backend systems are also not covered within this audit scope.

1.2 Methodology

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning 1.5 weeks. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations,software, and security design patterns, code styles and best practices.

Protocol Assumptions: To focus our audit and define its scope, we have made several key assumptions about how the protocol and contracts should work. These assumptions are based on expected behavior and trusted interac- tions, including areas that were outside the scope of this review. They help clarify the basis for our findings and ensure a secure, consistent operation.

- **Protocol Functionality and Security**: It is assumed that the protocol's underlying components—such as the consensus mechanism, validator setup, and genesis creation—operate securely and as intended. These components, though out of scope for this audit, are critical to the protocol's integrity and assumed to be reliable in all interactions with audited contracts.
- **Engine-Only Function Calls in System Contract**: We assume that all functions marked with the onlyEngine modifier are invoked exclusively by active validators within the consensus mechanism. This assumption includes the expectation that these functions are called consistently on every block and at each epoch by the consensus, ensuring the contract's intended functionality
- **Consensus and Genesis Creation Functionality**: While auditing, we assumed that both the consensus mechanism and genesis creation processes function correctly as designed. This falls outside the audit scope, which is restricted solely to the system contract. Thus, any reliance on these underlying mechanisms was assumed to be secure and operational
- **NeroPaymaster as a Verifying Paymaster**: Our audit assumes that the NeroPaymaster contract oper- ates as a Verifying Paymaster, dependent on backend validation processes, such as token whitelisting and signature verification. These backend verifications are foundational to the Paymaster's security and expected functionality
- **Correct Integration with EntryPoint and Related Contracts**: As the NeroPaymaster contract is central to the audit, it is assumed that the EntryPoint and other integrated contracts interact with NeroPaymaster as intended. This assumption underpins our understanding that the NeroPaymaster will perform reliably within the wider
- **Accurate msg.value Transfers in Consensus Genesis**: It is assumed that the msg.value sent to the System contract during consensus genesis is transferred accurately, enabling proper distribution of fees as designed
- **Order Consistency for onlyEngine Transactions**: It is assumed that onlyEngine transactions are not included into the mempool, as any mempool inclusion could introduce ordering issues. Thus, all transactions with the onlyEngine modifier are assumed to be executed in the correct sequence, maintaining the intended operational order.This assumption is important for correct initializations, such as the initialize function of the staking contract, which must be called prior to initValidator to properly set essential contract parameters. Ensuring the correct sequence in such initializations is vital for the secure and accurate setup of validator information and staking parameters.

1.3 Questions for Security Assessment

1. Are there effective mechanisms in place to prevent unauthorized modifications to staking and slashing rules for validators?
2. Does the staking process include adequate checks to prevent potential mismanagement of staking balances and rewards?
3. How does the protocol ensure that only eligible validators participate in consensus without circumventing validator registration requirements?
4. What safeguards exist to validate the correct application of penalties, ensuring that validator slashing is fair and consistent?
5. Are administrative functions properly restricted to prevent unauthorized access to validator and staking management?
6. Is there a secure mechanism to prevent unauthorized reallocation or transfer of rights during the changeAllRights and acceptAllRights operations?
7. Does the NeroPaymaster contract ensure that ERC20 deposits and withdrawals are securely managed and prevent unauthorized access or misuse of funds?
8. Are all project-specific tokens and NERO deposits within the NeroPaymaster contract protected against accidental or malicious fund depletion?
9. How does the NeroPaymaster verify the authenticity of sponsor requests and maintain security in the validation of user signatures and exchange rates?
10. Are entry point and post-operation functionalities sufficiently validated in NeroPaymaster to mitigate the risks of gas overconsumption or failed post-operation handling?

1.4 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.5 Summary of Findings Identified

S.NO	SEVERITY	FINDINGS	STATUS
#1	HIGH	Potential Front-Running Attack on Project Ownership Registration	FIXED
#2	HIGH	Absence of Random Validator Selection Enables Block Production Monopoly	ACKNOWLEDGED
#3	MEDIUM	Missing Initializer Lock Enables Unauthorized Ownership	FIXED
#4	MEDIUM	Risk of Permanent Contract Lockout due to renounceOwnership	FIXED
#5	LOW	Potential Financial Loss for Paymaster in Post-Paid Mode Due to Revoked User ERC20 Approval	ACKNOWLEDGED
#6	INFO	Block Fee Distribution Must Precede Validator Set Updates and Punishment Operations	ACKNOWLEDGED
#7	INFO	Optimizing GasUsage withPrefix Incrementation in Loops	ACKNOWLEDGED
#8	INFO	Optimizing SmartContract Efficiency by Avoiding Redundant Initializations	ACKNOWLEDGED
#9	INFO	Gas Efficiency Improvement by Substituting Booleans withuint256	ACKNOWLEDGED
#10	INFO	Enhancing Efficiency with !=0 Comparison for Unsigned Integers	ACKNOWLEDGED

2 Findings and Risk Analysis

2.1 Potential Front-Running Attack on Project Ownership Registration

Severity: High

Status: Fixed

Location :

1. `NeroPaymaster.sol`, `regOwner()`

Description In the `regOwner` function of the `NeroPaymaster` contract, a front-running vulnerability exists that could allow malicious users to register themselves as the project owner of a legitimate project before the legitimate owner completes their transaction. When a legitimate user attempts to register their project with a specific `project ID`, a malicious actor could front-run this transaction by submitting the same project ID, setting themselves as the project owner instead. Once set, the project ownership is locked and cannot be reassigned without explicit transfer from the current owner. This vulnerability allows the malicious actor to control deposits, withdrawals, and other operations associated with that project ID.

Vulnerable Code

```
function regOwner(uint256 _projectId) external whenNotPaused {
        // bytes32 hash = ECDSA.toEthSignedMessageHash(keccak256(abi.encodePacked(
            _projectId)));
        // require(verifier == ECDSA.recover(hash, _signature), &quot;NeroPaymaster:
            signature error&quot;);
        require(_projectId &gt; 0, &quot;NeroPaymaster: error project&quot;);
        require(projectOwner[_projectId] == address(0), &quot;NeroPaymaster: project
            initialized&quot;);
        projectOwner[_projectId] = msg.sender;
        emit ProjectOwner(_projectId, address(0), msg.sender);
    }
```

Attacker'S Steps

1. A legitimate user initiates a transaction to call `regOwner` with a unique `_projectId`.
2. A malicious actor observes this transaction in the mempool and submits their own transaction with the same `_projectId` but at a higher gas fee.
3. The malicious transaction is processed first, setting the malicious actor as the owner of `_projectId`
4. Any subsequent attempts by the legitimate user to register the same `_projectId` will fail because the project ownership has already been initialized

Impact :

1. **Financial Risk**: The legitimate user may incur backend setup costs to initialize a project, which would be lost if the project ownership is hijacked by a malicious actor. This includes costs associated with API creation, infrastructure setup, and the fees for backend processes.
2. **Denial of Service (DoS)**: The legitimate user is blocked from registering their project with the intended project ID, preventing them from using the NeroPaymaster service as intended.
3. **Enhanced Attack Potential**: If a user employs automated scripts to both register ownership and immediately deposit Nero, the malicious actor can capture both actions. The attacker's front-run transaction would succeed in claiming the _projectId ownership, followed by the legitimate user's deposit, which the attacker could then withdraw. This setup gives the attacker a greater advantage, allowing them to directly profit from funds deposited by the legitimate user.
4. **Reputation Risk**: This exploit could lead to loss of trust in the NeroPaymaster contract due to the risk of unauthorized project ownership and fund misappropriation.

Recommendation To mitigate this vulnerability, implement a verifier signature requirement for the `regOwner` function. This will ensure that only a legitimate owner can register a project with a given `_projectId`

2.2 Absence of Random Validator Selection Enables Block Production Monopoly

Severity: High

Status: Acknowledged

Location :

1. `Staking.sol`, `updateActiveValidatorSet()`, `getTopValidators()`

Description In the current Delegated Proof of Stake (DPoS) implementation, validators are selected for block production based on their total stake. According to protocol documentation, 15 validators should be chosen directly based on the highest stakes, while the remaining 6 should be selected randomly from the lower-ranked validators to ensure decentralization. However, the contract's `updateActiveValidatorSet` function does not implement this random selection, resulting in a validator set solely determined by the highest stakes.

Without a decentralized random selection mechanism, the same high-stake validators are chosen repeatedly across epochs. This increases the risk of centralization, as larger validators monopolize block production, limiting opportunities for lower-stake validators. Furthermore, this setup weakens the protocol's decentralization and increases vulnerability to potential collusion among dominant validators.

According to the protocol's guidelines, randomness should be generated through a secure Multiparty Computation (MPC) and Publicly Verifiable Secret Sharing (PVSS) scheme to ensure that the random selection process is tamper-proof and not controlled by any individual node. This mechanism is essential to maintaining the protocol's DPoS structure and decentralization goals.

Impact :

1. **Centralization Risk**: Dominant validators with high stakes monopolize block production, leading to a reduction in network decentralization.
2. **Reduced Validator Participation**: Smaller validators are effectively excluded from active selection, which discourages broader participation.

Recommendation Implement a decentralized random selection protocol to randomly select 6 validators from the pool of lower-stake candidates each epoch.This change will better align the implementation with the DPoS model's decentralization goals and mitigate the risk of validator centralization.

References Nero Docs

Developer Response

Noticed. But relative features will be implemented in future versions, not now. Current version will just select validators by staking amount.

2.3 Missing Initializer Lock Enables Unauthorized Ownership

Severity: Medium

Status: Fixed

Location :

1. `NeroPaymaster.sol`, `constructor()`

Description The NeroPaymaster contract lacks a `_disableInitializers()` call in its constructor, which is critical in upgradeable contract patterns to prevent unauthorized initialization. This omission can expose the contract to the following risks:

1. **Ownership Risk**: Without `_disableInitializers()`, any uninitialized contract allows an attacker to directly call the `initialize` function, assuming ownership. This could lead to unauthorized access to administrative functions.
2. **Functionality Exploits**: If an attacker becomes the owner, they could misuse key functions, such as modifying the verifier, withdrawing funds, or pausing critical operations. This can lead to financial losses and potentially disrupt the paymaster functionality.

Vulnerable Code

```
// constructor() {
//     _disableInitializers();
// }
```

Impact :

1. **Loss of Control**: If exploited, attackers gain full control over sensitive administrative functions.
2. **Funds Theft**: With control over `withdrawTo` and `withdrawStake`, an attacker can drain funds from the contract.
3. **Operational Disruption**: Modifications to settings like `entryPoint` and `verifier` could undermine the contract's intended operation and reliability.

Recommendation To prevent these risks, ensure `_disableInitializers()` is implemented in the constructor to lock the initializer. This pattern prevents any subsequent calls to `initialize`, thus mitigating unauthorized access and preserving contract integrity.

References :

- Openzeppelin Docs

2.4 Risk of Permanent Contract Lockout due to renounceOwnership

Severity: Medium

Status: Fixed

Location :

1. `NeroPaymaster.sol`

Description The `NeroPaymaster` contract imports `OwnableUpgradeable` from OpenZeppelin, which includes a renounceOwnership function. This function transfers ownership to the zero address (address(0)), effectively removing the contract owner. As a result, any functions restricted by the onlyOwner modifier will become inaccessible if renounceOwnership is called, either intentionally, by mistake, or due to malicious activity. This can lead to a scenario where the contract is permanently locked, making it impossible to manage or upgrade the contract, or to access critical functionalities.

Impact This vulnerability could lead to a complete loss of control over the contract's critical administrative functions. Once the ownership is renounced, functions restricted to the owner will no longer be accessible, potentially locking the contract indefinitely.

Recommendation override the `renounceOwnership` function in the `NeroPaymaster` contract to disable or restrict it

2.5 Potential Financial Loss for Paymaster in Post-Paid Mode Due to Revoked User ERC20 Approval

Severity: Low

Status: Acknowledged

Location :

1. `NeroPaymaster.sol`, `_postOp()`

Description In the `NeroPaymaster` contract, when operating in post-paid mode (mode = 2), the paymaster deducts gas fees from the user's ERC20 balance after the transaction is executed. However, if a user revokes or reduces the ERC20 token allowance for the paymaster before the gas deduction occurs, the paymaster is still responsible for covering the gas fees, as the `_postOp` function does not halt on insufficient allowance.

Impact This issue could lead to recurring financial losses for the paymaster in cases where users revoke their ERC20 token approvals in post-paid mode, as the gas costs would be covered by the paymaster rather than deducted from the user's ERC20 balance.

Recommendation Refer to the Nero documentation

Developer Response

The contract will not change, but will improve the AA Platform.

2.6 Block Fee Distribution Must Precede Validator Set Updates and Punishment Operations

Severity: Info

Status: Acknowledged

Location :

1. `Staking.sol`, `updateActiveValidatorSet()`, `lazyPunish()`, `doubleSignPunish()`, `distributeBlockFee()`, `onlyOperateOnce()`

Description In the current implementation of the staking contract, the block fees intended for validator rewards are distributed without guaranteed prioritization over validator set updates or punishment actions. Specifically:

1. **Validator Set Update**: When `updateActiveValidatorSet` is called before `distributeBlockFee`, validators who were active in the previous set may not receive the fees they were eligible for.
2. **Punishment Actions**: If `lazyPunish` or `doubleSignPunish` is called before `distributeBlockFee`, the validator's stake is reduced before rewards are calculated, leading to a potential reduction in the rewards they receive due to their lower stake.
3. This issue arises because `distributeBlockFee` does not precede these functions in every scenario, potentially causing validators to miss out on fees they would otherwise receive.

Impact Validators can lose out on rewards due to the order in which functions are called. This can lead to inconsistent and unfair fee distributions, particularly for validators who have contributed to the network up until the block where the set is updated or punishment is enforced. This can undermine validator incentives and the fairness of the staking system.

Recommendation Introduce a conditional check within the `onlyOperateOnce` modifier to ensure that `distributeBlockFee` executes before either `updateActiveValidatorSet` or any punishment function in the current block. If distributeBlockFee has not yet been called, prevent the execution of `updateActiveValidatorSet`, `lazyPunish`, and `doubleSignPunish` until `distributeBlockFee` completes, ensuring proper reward allocation before any validator's stake is modified.

2.7 Optimizing GasUsage withPrefix Incrementation in Loops

Severity: Info

Status: Acknowledged

Location :

1. `staking.sol`, `validator.sol`, `genesisLock.sol`

Description Within the smart contracts, the current use of postfix incrementation (i++) in loops is identified as a less gas-efficient practice. Switching to prefix incrementation (++i) for updating uint variables inside loops can yield gas savings. This optimization opportunity is applicable not only to loop iterators but also to any increment operations within the loop's body. The efficiency gain stems from how Solidity handles these two operations at the bytecode level, with prefix incrementation being slightly more gas-efficient due to its direct modification of the variable's state without the need to store an intermediate value.

Recommendation identify instances where i++ is used within loops.Refactor these to use ++i for direct incrementation.

2.8 Optimizing SmartContract Efficiency by Avoiding Redundant Initializations

Severity: Info

Status: Acknowledged

Location :

1. `staking.sol`, `genesisLock.sol`

Description In the smart contracts , there is a recurring pattern of initializing loop variables and accumulators to their default values. This practice, while common and clear for understanding, introduces unnecessary operations, as Solidity automatically initializes variables to their default values. Specifically, instances of loop counters being set to 0 (uint256 i = 0) and accumulators being initialized to 0, can be optimized by leveraging Solidity's default initialization to reduce bytecode size and potentially gas costs during contract deployment.

2.9 Gas Efficiency Improvement by Substituting Booleans withuint256

Severity: Info

Status: Acknowledged

Location :

1. `staking.sol`, `validator.sol`, `genesisLock.sol`

Description Current implementations within the contracts utilize boolean (bool) types for storage variables, which, while intuitive, lead to higher gas costs due to the Solidity storage model. Specifically, changing a boolean from false to true, especially after it has previously been set to true, incurs a substantial gas overhead. An optimization technique in- volves using uint256 values, such as uint256(1) and uint256(2), to represent true and false, respectively. This approach can significantly reduce gas costs associated with storage access and modification.

2.10 Enhancing Efficiency with !=0 Comparison for Unsigned Integers

Severity: Info

Status: Acknowledged

Location :

1. `staking.sol`, `validator.sol`, `genesisLock.sol`, `NeroPaymaster.sol`

Description In various locations across the smart contracts the comparison > 0 is used to validate unsigned integer values. Given that unsigned integers in Solidity cannot be negative, using != 0 for comparisons is recommended for clarity and potential gas optimization. This approach directly checks for non-zero values, which is the intended validation in these contexts, particularly when verifying that array lengths or token amounts are not zero

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts