hashlock.

Security Audit

Naoris Protocol (DeFi)

July 2025

info@hashlock.com.au

Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

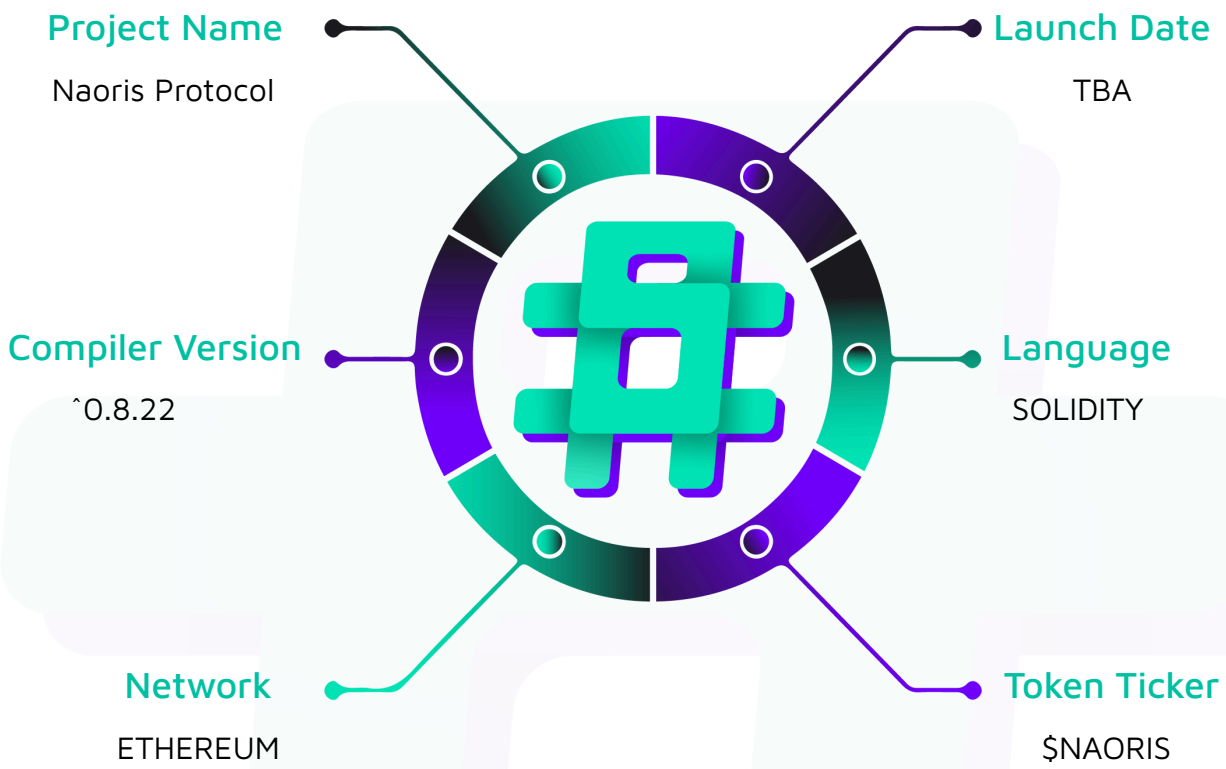
The Naoris Protocol partnered with Hashlock to conduct a security audit of their Naoris.sol and Governance.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

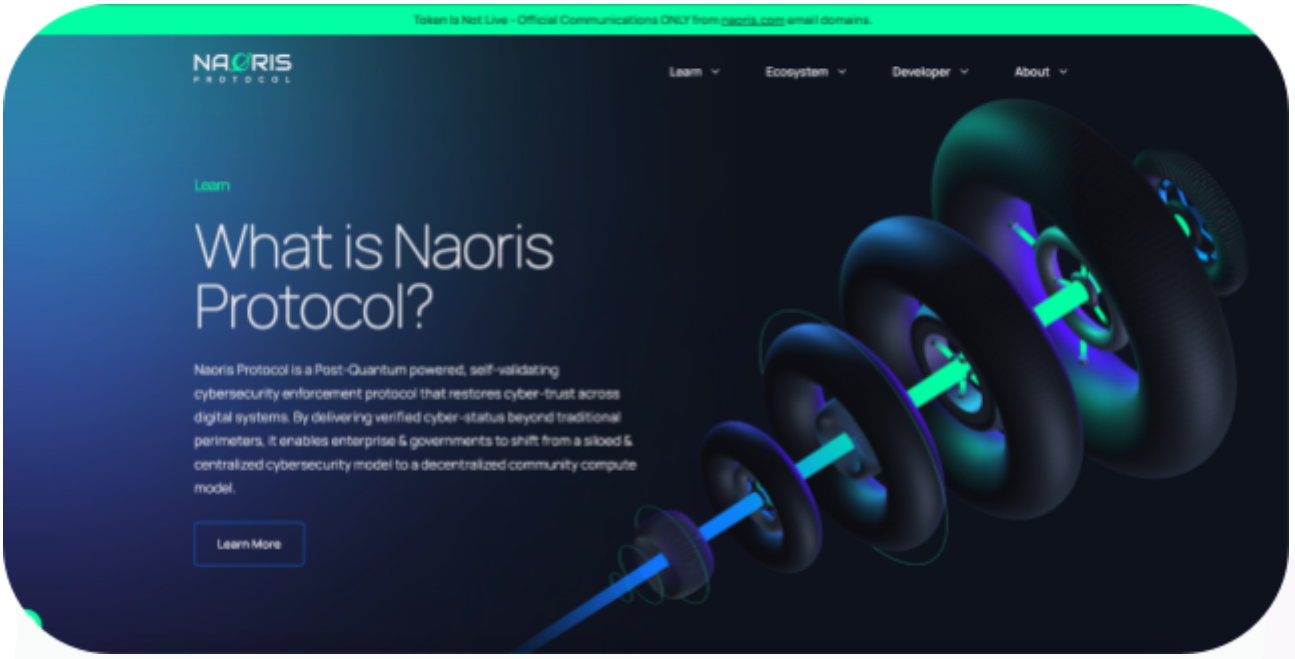
Naoris Protocol is a decentralized cybersecurity infrastructure that applies post quantum cryptography, blockchain, and distributed AI to secure both Web2 and Web3 systems. It transforms each connected device into an incentivized security node, collectively forming a "trust mesh" that continuously validates system integrity, eliminating single points of failure.

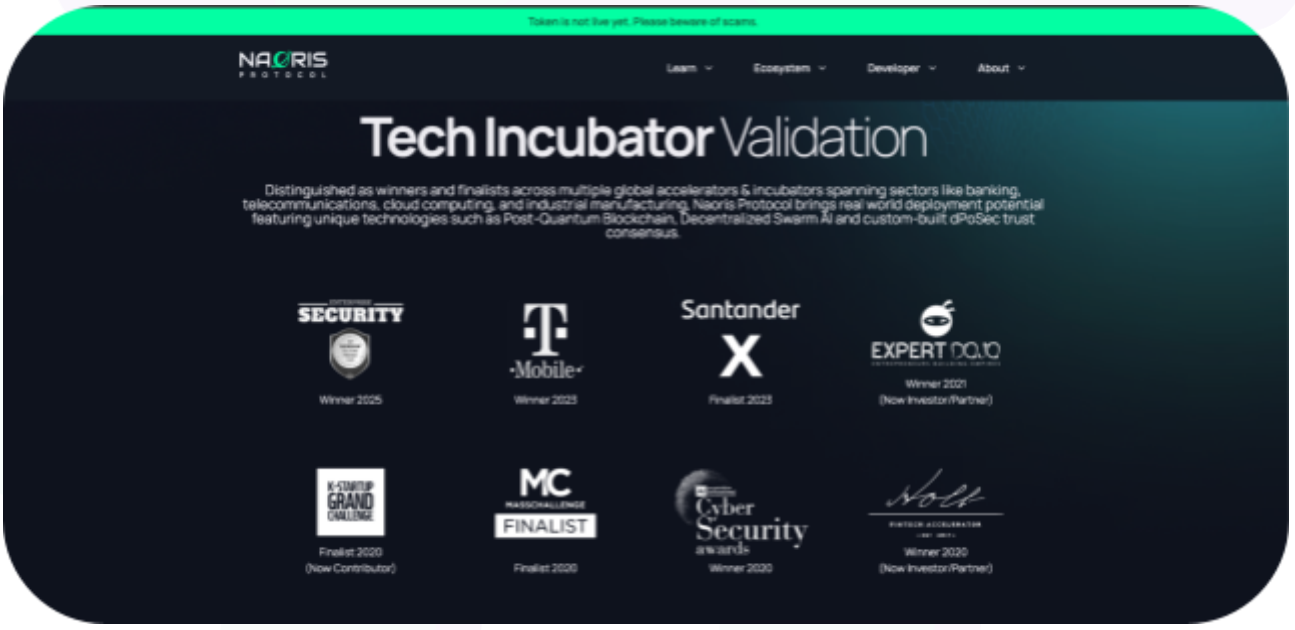
Project Name: Naoris Protocol
Project Type: DeFi
Compiler Version: ^0.8.22
Website: https://www.naorisprotocol.com/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Naoris Protocol project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Naoris Protocol Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**July, 2025**
Contract 1	Naoris.sol
Contract 1 MD5 Hash	b4ff218f60b91911b30d6d378eee2d0e
Contract 2	Governance.sol
Contract 2 MD5 Hash	27c7d66bd300f75b2c38b76908b8c41a

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section, and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

2 High severity vulnerabilities

9 Medium severity vulnerabilities

4 Low severity vulnerabilities

1 Gas Optimisation

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour

Governance.sol
Allows users to:
- Vote on active governance proposals with a weight determined by an external staking contract.
- Delegate their voting power to another address to vote on their behalf, either for a single proposal (delegateVoteForProposal) or for all proposals globally (delegateVoteGlobally).
- Revoke a previous delegation to reclaim their direct voting rights, for both proposal-specific and global delegations.
- Query and view a wide range of governance data, including:
  - The details, status, and options of any proposal.
  - The current list of active or cancelled proposals.
  - The vote count for each option in a proposal.
  - Whether they or another user has voted on a specific proposal.
  - The final winning option of a successful proposal.
- Clean Up Data by removing their voting record from a proposal that has been cancelled (Note: This function has access control flaws as audited, but this is its intended behavior).
	Contract achieves this functionality.

Allows admins to: The contract has two distinct administrative roles with different powers: The owner role can: - Manage the Multisig: Change the multisig address, effectively transferring all multisig powers to a new entity. The multisig role can: - Create Proposals: Create new governance proposals, defining their type, description, and voting options. - Cancel Proposals: Cancel a pending or active proposal before it has been finalized. - Execute Proposals: Trigger the finalization of a proposal after its voting and timelock periods have ended, setting its status to Succeeded, Defeated, or Tie. - Update Proposals: Modify the details of a proposal (e.g., description, IPFS link) before its voting period begins. - Extend Voting: Prolong the voting deadline for an active proposal. - Configure Governance Parameters: Adjust the core rules of the governance process, including the voteDelay (cool-down before voting starts), voteDuration (how long voting lasts), and timelockDuration (cool-down after voting ends).	
Naoris.sol This contract defines an upgradeable ERC20 token with a fixed total supply. It includes administrative controls for pausing token activity, managing roles, and	**Contract achieves this functionality.**

performing contract upgrades.

Allows users to:

- Transfer tokens they own to other addresses, as long as the contract is not paused.
- Check the token balance of any address.
- Approve other accounts (e.g., decentralized exchanges) to spend a specific amount of their tokens.
- Modify approvals by increasing or decreasing the allowance for a spender.
- Perform gas-less approvals by signing an off-chain message using the permit function, which another party can then submit on-chain.
- Query token metadata, including its name (name), symbol (symbol), decimal precision (decimals), total supply, and the fixed supply cap.

Allows admins & privileged roles to:

The contract defines several distinct administrative roles with different capabilities, primarily the owner, DEFAULT_ADMIN_ROLE holder, and PAUSER_ROLE holder.

The owner can:

- Authorize an upgrade of the contract's logic to a new implementation address, which is the most powerful administrative action.

Accounts with DEFAULT_ADMIN_ROLE can:

- Manage all roles, including granting the PAUSER_ROLE to new addresses or revoking it from existing ones.

Accounts with PAUSER_ROLE can:

- Pause the contract, which halts all major token

activities, including transfer, transferFrom, and approve. - Unpause the contract to resume normal token functionality. Additionally, the contract has a one-time initialization function (callable only once upon deployment) that: - Sets the token's name and symbol. - Mints the entire fixed supply of 4 billion tokens to the initialOwner. - Assigns the owner, DEFAULT_ADMIN_ROLE, and PAUSER_ROLE to the initialOwner. - Starts the contract in a paused state by default.	

Code Quality

This audit scope involves the smart contracts of the Naoris Protocol, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Naoris Protocol smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] Governance.sol - Indefinite Postponement of Proposal Finalization

Description

The `extendVoting` function grants the multisig address the power to prolong a proposal's voting period. While intended for flexibility, this function lacks any limitations on its use. A malicious or controlling multisig can repeatedly call this function to indefinitely push the proposal's deadline into the future, effectively preventing any proposal it disagrees with from ever being finalized.

Vulnerability Details

The core of this vulnerability is a flaw in the governance design that grants unchecked power to the multisig.

- Unrestricted Access: The `extendVoting` function is controlled only by the `onlyMultisig` modifier.
- Lack of Limits: There are no protocol enforced limits on how many times a proposal's voting period can be extended or on the total duration that can be added.

Proof of Concept

There are no protocol enforced limits on how many times `extendVoting` function can be called on a proposal:

```
function extendVoting(uint256 id, uint32 additionalTime) external onlyMultisig proposalExists(id) {
        ProposalDetails storage p = proposals[id];
        if (block.timestamp > p.voteEnd) revert VotingAlreadyEnded();
        p.voteEnd += additionalTime;
        p.timelockEnd += additionalTime;
```

```
                    emit VotingExtended(id, additionalTime, p.voteEnd, p.timelockEnd,
block.timestamp);
    }
```

The attack scenario is outlined as follows:

A proposal is created and is gathering significant community support. The multisig entity observes that the vote is trending towards an outcome they wish to avoid.

Just before the `voteEnd` timestamp is reached, the multisig calls `extendVoting`, pushing the deadline further into the future. They can repeat this action indefinitely, each time the new deadline approaches.

As a result, the proposal can never be finalized and `executeProposal` can never be called. The multisig achieves a powerful, unilateral veto, not by casting votes, but by endlessly stalling the process.

Impact

This is a High severity governance vulnerability that fundamentally undermines the principles of decentralized decision making.

- Centralized Censorship and Control: It grants the multisig the de facto power to censor any proposal. By refusing to let a vote conclude, they can effectively block any community decision, completely overriding the will of the token holders.
- Denial of Service for Governance: This mechanism can be used to mount a denial of service attack on the entire governance system. Any proposal can be stalled, rendering the voting process meaningless.

Recommendation

Introduce a counter to track the number of extensions for each proposal and enforce a hard limit.

Status

Resolved

[H-02] Governance.sol - Potential Denial of Service via Gas-Intensive Delegator Loop in castVote

Description

The `castVote` function is designed to aggregate a voter's total governance weight by combining their personal weight with that of all the users who have delegated their vote to them. It achieves this by looping through an EnumerableSet of delegators. However, this design does not impose any limit on the number of delegators a single address can have. If a user becomes a highly popular delegatee, attracting a very large number of delegators, the loop inside the `castVote` function can become so computationally expensive that the gas required to execute it exceeds the blockchain's block gas limit.

Vulnerability Details

The unbounded loop `for (uint256 i = 0; i < delegatorsCount; ++i)` iterates over every single delegator associated with the `voter` (`msg.sender`). Each loop performs multiple state-changing operations, all of which consume gas:

- Reading the delegator's address from storage (`EnumerableSet.at`).
- Checking if the delegator has already voted (`hasVoted[proposalId][delegator]`).
- Making an external call to get the delegator's weight (`getUserTotalGovernanceWeight`).
- Multiple state writes (`hasVoted`, `userVotes`, `proposalVoters`, `p.castedVotes`, `totalWeight`).

Proof of Concept

Attack Scenario (Griefing or Unintentional) is outlined as follow:

1. An address, "Delegatee Alpha," becomes a very popular and trusted delegatee.
2. Thousands of users delegate their votes to Delegatee Alpha. This can happen organically or could be orchestrated maliciously by creating many new accounts that delegate to the target.
3. When Delegatee Alpha attempts to call `castVote` on any proposal, the transaction begins executing the for loop.

4. As the loop iterates through thousands of delegators, the cumulative gas cost surpasses the block gas limit.
5. The transaction fails due to an "out of gas" error.
6. Because the number of delegators cannot be reduced by the delegatee, and every attempt to vote will trigger the same gas-intensive loop, Delegatee Alpha is permanently blocked from ever successfully casting a vote.

Impact

A popular delegatee and, by extension, all of the thousands of users who delegated their voting power to them, are completely prevented from participating in governance. Their collective voting weight is effectively nullified.

Malicious actors could also intentionally delegate a massive number of low-weight votes to a delegatee they oppose, specifically to trigger this DoS condition and remove that delegatee's influence from a close vote.

Recommendation

Limit on how many delegators a user can have, and enforce it during delegation.

Status

Resolved

Medium

[M-01] Naoris.sol - The whenNotPaused modifier on the approve function can be bypassed by invoking the permit function.

Description

The `Naoris.sol` contract intends to prevent new allowances from being created while it is paused by placing the `whenNotPaused` modifier on the `approve` function. However, the contract also inherits the `ERC20PermitUpgradeable` extension, which introduces a separate, un-paused mechanism for creating allowances via the `permit` function.

The `permit` function allows users to grant an allowance to a spender by signing an approval message off-chain. Anyone can then submit this signature to the `permit` function on chain, which validates the signature and then calls the internal `_approve` function. Because the `permit` function itself was not overridden and protected with the `whenNotPaused` modifier, it provides a loophole to bypass the contract's paused state for approvals.

Vulnerability Details

The vulnerability arises because the pause control is not applied consistently across all methods that can alter token allowances.

Below is the attack scenario:

1. Contract is Paused: An owner with the `PAUSER_ROLE` calls the `pause()` function. This action is intended to freeze all token-related activities, including transfers and new approvals, as a security measure.
2. User Signs a Permit: A token holder (Alice) wants to approve a spender (Bob) to access her tokens. A direct call to `approve(bob, 1000)` would fail due to the `whenNotPaused` modifier.
3. Bypassing the Pause: Alice instead signs an EIP-712 compliant message off-chain that specifies her intent to grant Bob an allowance of 1000 tokens.

4. Executing the Permit: Bob (or any other party) takes Alice's signature and calls the `permit()` function on the `Naoris.sol` contract, passing in the signature components and approval details.
5. Successful Approval: The `permit` function is not protected by a `whenNotPaused` modifier. It successfully verifies Alice's signature and executes `_approve(alice, bob, 1000)` internally.
6. Outcome: Bob now has an allowance of 1000 of Alice's tokens, created while the contract was paused. The security control on the `approve` function has been completely bypassed. While Bob cannot yet call transferFrom (as it is also paused), the allowance has been successfully created, undermining the integrity of the pause mechanism.

Impact

The primary impact is the failure of a security mechanism. The pause feature is designed to be a "big red button" to halt contract activity in an emergency. This vulnerability creates an incorrect assumption that the owner of the contract may believe all allowance changes are frozen, but unexpected state changes of the token allowance are still possible.

Recommendation

Override the `permit` function in the `Naoris.sol` contract and apply the `whenNotPaused` modifier.

Status

Resolved

[M-02] Naoris.sol - Redundant and Potentially Confusing Role Assignments

Description

The `Naoris.sol` contract concurrently implements two different access control patterns from OpenZeppelin: `OwnableUpgradeable` and `AccessControlUpgradeable`. `Ownable` provides a simple, single-owner model, while `AccessControl` offers a more flexible system based on granular roles. Using both simultaneously creates a redundant, complex, and potentially confusing administrative framework where different critical functions are protected by separate and overlapping permission systems.

Vulnerability Details

Scenario of Failure:

Imagine the administrators decide to transfer administrative duties.

The current owner calls `transferOwnership(newAdmin)`. This successfully transfers the owner role.

However, they forget to separately grant the necessary roles to the new admin using `grantRole(PAUSER_ROLE,newAdmin)` and `grantRole(DEFAULT_ADMIN_ROLE, newAdmin)`.

As a result, the `newAdmin` can now authorize contract upgrades (as they are the new owner), but they cannot pause the contract or manage other roles because they do not have the `PAUSER_ROLE` or `DEFAULT_ADMIN_ROLE`. The contract is now in an inconsistent state where administrative powers are split between two different addresses, leading to a partial loss of control.

Impact

The use of dual access control systems introduces significant risks that compromise the contract's manageability and security.

Increased Risk of Misconfiguration: The primary risk is human error. It is easy for administrators to update permissions in one system while forgetting the other, leading to a loss of access to critical functions, as demonstrated in the scenario above.

Complex Management Overhead: Administrators must understand and manage two separate systems. This increases the cognitive load and complicates routine tasks like transferring administrative privileges or auditing permissions.

Violates Principle of Least Astonishment: The behavior of the contract can be surprising. An administrator who is the owner would logically expect to have full control, yet they would be unable to pause the contract without the separate PAUSER_ROLE.

Recommendation

simplify the architecture by committing to a single, consistent access control pattern. Given the contract's needs (separate roles for pausing and potentially other functions), `AccessControl` is the more suitable and flexible choice.

1. Remove `OwnableUpgradeable`: Eliminate `OwnableUpgradeable` from the contract's inheritance list and remove `__Ownable_init()` from the `initialize` function.
2. Update `_authorizeUpgrade`: Upgrade the modifier from `onlyOwner` to `onlyRole(DEFAULT_ADMIN_ROLE)`.

Status

Resolved

[M-03] Governance.sol - Defeated Proposals Can Be Executed Multiple Times

Description

The `executeProposal` function is responsible for finalizing a proposal's outcome after its voting period and timelock have passed. The function contains logic to mark a proposal as `Defeated` if it fails to meet the `minimumVotes` threshold. However, the function lacks a crucial check to prevent this defeat logic from being triggered repeatedly on the same proposal, allowing key state variables such as `totalExecutedProposals` and `totalDefeatedProposals` to be manipulated.

Vulnerability Details

The vulnerability exists because the function does not properly check the proposal's status before evaluating the defeat condition. The control flow allows the defeat logic to be triggered repeatedly.

Let's trace the execution path for a defeated proposal:

1. First `executeProposal` Call:
    - A multisig calls `executeProposal` on a proposal where the number of casted votes is less than the required minimum.
    - The check `if (p.castedVotes < p.minimumVotes)` evaluates to true.
    - The proposal's status is set to `ProposalStatus.Defeated`.
    - The counters `totalExecutedProposals` and `totalDefeatedProposals` are both incremented.
    - The function emits an event and then executes a return, terminating the call.
2. Second `executeProposal` Call (on the same proposal):
    - The multisig calls `executeProposal` again for the same proposal ID.
    - The function's initial checks pass.
    - It reaches the same defeat condition: `if (p.castedVotes < p.minimumVotes)`. Since the votes have not changed, this condition is still true.

- Crucially, there is no check preventing this block from running again. The code re-executes the defeat logic.
- The status is redundantly set to `Defeated` again.
- The counters `totalExecutedProposals` and `totalDefeatedProposals` are incremented a second time.
- The `ProposalExecuted` event is emitted again.

This process can be repeated indefinitely, allowing the multisig to arbitrarily inflate the `totalExecutedProposals` and `totalDefeatedProposals` counters.

Impact

The `totalExecutedProposals` and `totalDefeatedProposals` counters are critical metrics for assessing the health, activity, and history of the governance system. By repeatedly executing a defeated proposal, the multisig can make these numbers meaningless, creating a false impression of high governance activity.

Recommendation

The `executeProposal` function must be modified to ensure that a proposal can only be processed once. This is best achieved by moving the status check to the beginning of the function, immediately after retrieving the proposal details. This ensures that any proposal that is no longer in an Active state cannot be processed again.

```
function executeProposal(uint256 proposalId) external proposalExists(proposalId)
onlyMultisig {
        ProposalDetails storage p = proposals[proposalId];
        if (block.timestamp < p.timelockEnd) revert TimelockNotOver();
++      if (p.status != ProposalStatus.Active) revert InvalidProposalStatus();

        if (p.castedVotes < p.minimumVotes)  {
            p.status = ProposalStatus.Defeated;
            totalExecutedProposals++;
            totalDefeatedProposals++;
            activeProposals.remove(proposalId);
            emit ProposalExecuted(proposalId, p.status, block.timestamp);
            return;
        }
```

```
--      if (p.status != ProposalStatus.Active) revert InvalidProposalStatus();

//REDACTED
```

Status

Resolved

[M-04] Governance.sol - Voting Period Can Be Extended on a Cancelled Proposal

Description

The `extendVoting` function is intended to give the multisig the flexibility to prolong the voting period for an ongoing or upcoming proposal. However, the function critically fails to check the proposal's current status before applying the extension. This omission allows the multisig to modify the `voteEnd` and `timelockEnd` of a proposal that has already been `Cancelled`, leading to an illogical and inconsistent contract state.

Vulnerability Details

The execution flow only validates that the original voting period has not yet ended (`block.timestamp < p.voteEnd`). It does not contain any checks to ensure the proposal is in a state that should logically be extendable (i.e., `Pending` or `Active`).

This creates a loophole that can be exploited in the following sequence:

1. A proposal is created and is in an `Active` state.
2. The multisig decides to cancel the proposal and calls `cancelProposal(proposalId)`. The proposal's status is now correctly set to `ProposalStatus.Cancelled`.
3. Before the original `voteEnd` timestamp passes, the multisig calls `extendVoting(proposalId, ...)`.
4. The `extendVoting` function executes successfully because the only condition it checks (`block.timestamp < p.voteEnd`) is met. It does not check that `p.status` is `Cancelled`.
5. As a result, the `voteEnd` and `timelockEnd` timestamps of an officially `Cancelled` proposal are updated.

Impact

A `Cancelled` proposal is terminal and should be inert. Modifying its properties makes it a "zombie" proposal, cancelled in name but with active looking timestamps. This violates the integrity of the proposal lifecycle.

Also, front-end applications, dashboards, and data analytics tools that read from the contract will be presented with contradictory information (e.g., a proposal is displayed as "`Cancelled`" but has a voting deadline in the future). This can break UI components and lead to profound user confusion.

Recommendation

The `extendVoting` function must be updated to validate that the proposal is in a valid state before applying an extension. The voting period should only be extendable if the proposal is `Pending` or `Active`.

```
function extendVoting(uint256 id, uint256 additionalTime) external onlyMultisig
proposalExists(id) {
    ProposalDetails storage p = proposals[id];
    if (block.timestamp >= p.voteEnd) revert VotingAlreadyEnded();


++  if (p.status != ProposalStatus.Active && p.status != ProposalStatus.Pending) {
++      revert InvalidProposalStatus();
++  }

    p.voteEnd += additionalTime;
    p.timelockEnd += additionalTime;

    emit VotingExtended(id, additionalTime, p.voteEnd, p.timelockEnd, block.timestamp);
}
```

Status

Resolved

[M-05] Governance.sol - Missing Access Control in removeCancelledProposalData

Description

The `removeCancelledProposalData` function is intended to allow a user to clean up their personal voting data from a proposal that has been cancelled. However, this function completely lacks access control. It is an external function that can be called by anyone, and it allows the caller to specify any `voter` address as a parameter, leading to a scenario where one user can manipulate the state of another user without their consent.

Vulnerability Details

The core of the vulnerability is that the function does not validate that the caller (`msg.sender`) is the same person as the `voter` whose data is being deleted.

The `voter` address is supplied as an external argument. There are no modifiers or require statements to check if `msg.sender == voter`.

Impact

The primary impact is that any user can arbitrarily modify the on-chain voting records of any other user for a cancelled proposal. This violates the principle of user sovereignty over their data and state.

Recommendation

The `removeCancelledProposalData` function must be refactored to ensure that only the user whose data is being cleared can initiate the action. This can be achieved by adding the check `require(msg.sender == voter,"Unauthorized!");`

Status

Resolved

[M-06] Governance.sol - Unbounded Proposal Options Can Cause Denial of Service

Description

The `createProposal` function allows the multisig to specify the options for a new proposal without imposing any upper limit on the number of options. The `executeProposal` function, which is essential for finalizing a proposal's outcome, must iterate through every single one of these options to determine the winner. If a proposal is created with an excessively large number of options, the gas cost required to execute this loop will exceed the block gas limit, making it impossible to ever finalize the proposal.

Vulnerability Details

This vulnerability creates a Denial of Service (DoS) condition on a per-proposal basis due to a lack of input validation.

- Unchecked Input: In `createProposal`, the options array can be of any size greater than two. A multisig could, either maliciously or accidentally, provide an array with thousands of elements.
- Gas-Intensive Loop: In `executeProposal`, the for loop `for (uint256 i = 0; i < optionsLength; i++)` directly depends on this unbounded input. Each iteration of the loop performs state reads `optionWeights[proposalId][i]`, memory writes, and comparisons, all of which consume gas.

The attack scenario is outlined as follows:

1. The multisig creates a proposal with, for example, 1000 options.
2. The community participates, casting votes and spending gas.
3. After the voting and timelock periods conclude, the function `executeProposal` is called to finalize the result.
4. The transaction attempts to execute the for loop 1000 times. The cumulative gas cost quickly surpasses the block gas limit, causing the transaction to fail and revert.
5. Because no transaction can ever supply enough gas to complete the loop, `executeProposal` can never succeed for this specific proposal.

6. The proposal becomes stuck in an `Active` state, unable to be resolved as `Succeeded`, `Defeated`, or `Tie`.

Impact

A proposal can be "bricked," making it impossible to ever finalize its outcome or act upon the community's decision. Other than that, all votes cast and gas fees paid by community members for such a proposal are completely wasted.

Recommendation

A sensible, fixed upper limit on the number of options a proposal can have must be enforced during creation of a new proposal, such as:

```
contract Governance is Ownable {
    uint256 constant MAX_PROPOSAL_OPTIONS = 25; // A reasonable limit
    // ...
}
```

Step 1: Define a constant for the maximum number of options.

```
function createProposal(
    // ... parameters
) external onlyMultisig {
    if (options.length < 2) revert AtLeastTwoOptionsRequired();
    // Add the new check here
    require(options.length <= MAX_PROPOSAL_OPTIONS, "Exceeds maximum allowed options");

    uint256 id = ++proposalCount;
    // ... rest of the function
}
```

Step 2: Add an input validation check in `createProposal`.

```
function updateProposalDetails(
        uint256 id,
        ProposalType proposalType,
        string memory description,
        string memory ipfsCID,
        string[] memory options
```

```
    ) external onlyMultisig proposalExists(id)  {
      ProposalDetails storage p = proposals[id];
      if (block.timestamp >= p.voteStart) revert VotingAlreadyStarted();
      if (p.status != ProposalStatus.Pending) revert InvalidProposalStatus();
      // Add the new check here
      require(options.length <= MAX_PROPOSAL_OPTIONS, "Exceeds maximum allowed options");
      // REDACTED
```

Step 3: Add an input validation check in `updateProposalDetails`.

Status

Resolved

[M-07] Governance.sol - getWinningOption Uses Incorrect Logic and Lacks Proper Status Check

Description

The `getWinningOption` function is intended to be a simple "getter" that retrieves the winning outcome of a completed proposal. However, its current implementation contains two significant flaws. First, instead of simply reading the stored result, it recalculates the winner by iterating through all options. Second, it fails to perform the most critical validation: checking if the proposal's status is Succeeded.

Vulnerability Details

The function's logic is both inefficient and incorrect. A view function for retrieving a result should not reimplement the complex business logic that was already handled by a state-changing function (`executeProposal`).

- Incorrect Checks: The function checks for what the status is not (`!= Defeated`,`!= Tie`) instead of what it is. This is a weak and incomplete validation. A proposal's winner is only officially determined and stored when its status is set to `Succeeded`. The function will incorrectly proceed for proposals that are Pending, Active, or Cancelled (though the timelock check may coincidentally prevent some of these cases).
- Redundant Calculation: The `executeProposal` function has already performed the logic to find the winning option and has saved the results in `p.winningOption` and `p.highestWeight`. The `getWinningOption` function wastefully reruns this same loop, which is inefficient and can potentially produce a different or misleading result.

Impact

Off-chain applications or users calling this function can receive unofficial or premature "winning" information. This could lead to confusion, as the result returned by this function may not match the final, official outcome once the proposal is correctly executed.

Recommendation

The `getWinningOption` function should be completely refactored to serve its true purpose: reading stored results from an officially succeeded proposal. It should not perform any calculations.

Recommended Code:

```
function getWinningOption(uint256 proposalId) external view proposalExists(proposalId)
returns (uint256, string memory, uint256) {
    ProposalDetails storage p = proposals[proposalId];
    if (block.timestamp <= p.timelockEnd) revert TimelockNotOver();
    // The single, correct check: ensure the proposal has officially succeeded.
    require(p.status == ProposalStatus.Succeeded, "Proposal has not succeeded");

    // Return the values that were already calculated and stored by executeProposal.
    uint256 winningIndex = p.winningOption;
    return (winningIndex, p.options[winningIndex], p.highestWeight);
}
```

Status

Resolved

[M-08] Governance.sol - Missing Vote Choice Recording in castVote for Delegators

Description

When a delegatee casts a vote, the `castVote` function correctly processes the voting weight of their delegators and marks those delegators as having participated in the vote. However, the function fails to record the specific option choice for the individual delegators in the `voterChoice` mapping. This results in a critical data integrity issue where the contract acknowledges that a delegator has voted but stores incorrect information about what they voted for.

Vulnerability Details

The flaw lies within the for loop of the castVote function that iterates through a delegatee's delegators. While the logic correctly updates `hasVoted` for the delegator, it omits the corresponding update to `voterChoice`.

Impact

The contract stores an inconsistent state. It claims a user has voted but records an incorrect choice for them, which is misleading and factually wrong.

Recommendation

The fix is straightforward and involves adding the missing line of code to record the vote choice for each delegator within the `castVote` function's loop:

```
function castVote(uint256 proposalId, uint256 optionIndex) external proposalExists(proposalId) {
        ProposalDetails storage p = proposals[proposalId];
        //REDACTED

            if (!hasVoted[proposalId][delegator]) {
                uint256 delegatorWeight = getUserTotalGovernanceWeight(delegator);
                if (delegatorWeight > 0) {
                    hasVoted[proposalId][delegator] = true;
                    userVotes[delegator].add(proposalId);
                    proposalVoters[proposalId].push(delegator);
```

```
++                  voterChoice[proposalId][delegator] = optionIndex;
                    totalWeight += delegatorWeight;
                    p.castedVotes++;
                }
            }
        }
        //REDACTED
```

Status

Resolved

[M-09] Governance.sol - Unsafe use of uint8 for userConsecutiveVotes

Description

The `userConsecutiveVotes` variable, intended to track how many times a user has voted consecutively, is declared as a `uint8` data type within the Governance contract.

```
mapping(address => uint8) public userConsecutiveVotes;
```

This variable is incremented in the `castVote` function whenever a user successfully casts a vote and has also voted on the immediately preceding proposal.

Vulnerability Details

A `uint8` variable can only store values ranging from 0 to 255. When the value of `userConsecutiveVotes` reaches 255 and is incremented again, it will revert due to overflow. In Solidity versions 0.8.0 and higher, arithmetic operations are checked by default, meaning that integer overflow (and underflow) will cause the transaction to revert rather than silently wrapping around.

Impact

The primary impact is that highly active users, once they reach 255 consecutive votes, will no longer be able to cast votes successfully through the `castVote` function. Their transactions will consistently fail and revert due to the integer overflow.

Recommendation

To prevent this critical issue and allow all users to participate indefinitely, the `userConsecutiveVotes` variable's data type should be upgraded to a larger unsigned integer type, ideally `uint256`:

```
mapping(address => uint256) public userConsecutiveVotes;
```

Status

Resolved

Low

[L-01] Governance.sol, Naoris.sol - Unrestricted `renounceOwnership()` Call

Description

The contract `Governance.sol` inherits from OpenZeppelin's `Ownable` contract, and the contract `Naoris.sol` inherits from OpenZeppelin's `OwnableUpgradeable` contract, which includes the `renounceOwnership()` function in both contracts. This function allows the current owner to irreversibly relinquish ownership, setting the `owner` address to `address(0)`.

Impact

In the current implementation, `renounceOwnership()` is not overridden or restricted, meaning any accidental or intentional call by the owner will permanently remove owner control over the contract.

Recommendation

If ownership should never be renounced (common for fixed-governance or permanently-administered tokens), override and disable It by adding the following function:

```
function renounceOwnership() public override onlyOwner {

    revert("Renouncing ownership is disabled.");

}
```

Status

Resolved

[L-02] Governance.sol, Naoris.sol - Use Ownable2Step/Ownable2StepUpgradeable contract rather than Ownable/OwnableUpgradeable contract

Description

`Ownable` and `OwnableUpgradeable` contracts from OpenZeppelin are used in the `Governance.sol` and `Naoris.sol` contracts, respectively. The primary concern with the `Ownable/OwnableUpgradeable` contract's `transferOwnership` function is its immediacy and lack of verification. Once called, ownership is instantly transferred to the specified address without any confirmation from the recipient.

Impact

If the new owner's address is entered incorrectly, the contract's control could be lost or handed over to an unintended entity, leading to potential security risks or loss of functionality.

Recommendation

To mitigate this risk, it's advisable to use OpenZeppelin's `Ownable2Step/Ownable2StepUpgradeable` contracts, these contracts enhance the ownership transfer process by implementing a two-step mechanism.

This two-step process ensures that ownership is only transferred if the new owner explicitly accepts it, thereby preventing accidental transfers to incorrect addresses.

Status

Resolved

[L-03] Governance.sol - Incorrect Comparison Operator in `cancelProposal` and `extendVoting`

Description

The `cancelProposal` function is designed to allow a multisig wallet to cancel a governance proposal. The function includes a check to prevent a proposal from being cancelled after the voting period has finished. The current implementation uses the condition `if (block.timestamp > p.voteEnd)` to determine if the voting period has ended. However, this logic is imprecise. It does not account for the scenario where a cancellation transaction is executed in the exact same block where the voteEnd timestamp is reached.

The `extendVoting` function is designed to allow a multisig wallet to extend the voting period for an active proposal. To ensure this action is only performed on proposals that have not yet concluded, the function contains a time checking condition. However, the specific logic used, `if (block.timestamp > p.voteEnd)` , is imprecise and fails to correctly handle the boundary case where the function is executed at the exact moment the voting period ends.

Impact

The impact of this vulnerability is classified as Low, but it represents a flaw in the contract's state management logic.

It allows the multisig to cancel a proposal at the very last moment, even as the final votes are being cast in the same block. This could be exploited to nullify a proposal's outcome that the multisig finds unfavorable, right at the deadline. While the window of opportunity is extremely narrow (it must be executed in that specific block), it undermines the fairness and predictability of the governance process by allowing an action that should be prohibited once the voting deadline is reached.
For `extendVoting` function, it compromises the finality and predictability of a proposal's deadline. It grants the multisig the ability to modify the rules of a vote at the very last second.

For example, if a vote is about to conclude with an outcome the multisig finds unfavorable, they can execute `extendVoting` in the final block to give themselves more time to influence the result. This reactive capability undermines the trust in the governance process, as it suggests that deadlines are not absolute but can be shifted based on the whims of the multisig at the last moment. While the window of opportunity is very small (a single block), it represents a significant flaw in the contract's state machine.

Recommendation

To fix this vulnerability, the comparison operator in the `cancelProposal` & `extendVoting` functions must be changed from `>` to `>=`:

```
function cancelProposal(uint256 proposalId) external onlyMultisig proposalExists(proposalId) {
        ProposalDetails storage p = proposals[proposalId];
--      if (block.timestamp > p.voteEnd) revert VotingAlreadyEnded();
++      if (block.timestamp >= p.voteEnd) revert VotingAlreadyEnded();

         if (p.status == ProposalStatus.Cancelled || p.status == ProposalStatus.Defeated || p.status == ProposalStatus.Succeeded)
            revert InvalidProposalStatus();

        p.status = ProposalStatus.Cancelled;

        activeProposals.remove(proposalId);
        cancelledProposals.add(proposalId);

        emit ProposalCancelled(proposalId, msg.sender, block.timestamp);
    }
```

```
function extendVoting(uint256 id, uint32 additionalTime) external onlyMultisig proposalExists(id) {
        ProposalDetails storage p = proposals[id];
--      if (block.timestamp > p.voteEnd) revert VotingAlreadyEnded();
++      if (block.timestamp >= p.voteEnd) revert VotingAlreadyEnded();
        p.voteEnd += additionalTime;
        p.timelockEnd += additionalTime;
```

```
                    emit VotingExtended(id, additionalTime, p.voteEnd, p.timelockEnd,
block.timestamp);
    }
```

Status

Resolved

[L-04] Governance.sol - Important functions lack event emissions

Description

In Solidity, events provide a critical logging mechanism that external observers (like dApps, users, or auditors) rely on to track what's happening on-chain. Important functions such as those that modify contract state, transfer funds, or change ownership should emit events to provide transparency.

The following critical functions were lacking event emissions:

```
function removeCancelledProposalData()
```

Impact

Critical actions (e.g., withdrawals, admin changes) occur silently, making monitoring via block explorers or off-chain tools impossible.

Recommendation

Emit appropriate events in all important state-changing functions.

Status

Resolved

Gas

[G-01] Governance.sol - Redundant Variable Initialization

Description

In Solidity, variables are initialized to a default zero-value (0, false, address(0), etc.). Explicitly initializing a variable to its default value is redundant and adds unnecessary bytecode.

In `executeProposal`, `uint256 highestWeight = 0;` , `uint256 tieCount = 0;` and `for (uint256 i = 0;` are not necessary.

In `getWinningOption`, `castVote` and `getProposalDelegators` functions, `for (uint256 i = 0;` are not necessary.

Recommendation

Remove initializations where the variable is being set to its default zero-value, such as `uint256 highestWeight = 0;` to `uint256 highestWeight;`

Status

Resolved

Centralisation

Naoris Protocol values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, Naoris Protocol seems to have a sound and well-tested code base; now that our vulnerability findings have been resolved. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the 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).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. 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 also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

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 their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au