Security Review Report
NM-0286 Nodle

NETHERMIND
SECURITY

(Sep 03, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Nodle Network. This review focused on the smart contracts enabling the migration of Nodle Network Grants and rewards system from the Eden parachain on Polkadot to ZkSync on Ethereum.

The Grants system involves locked NODL tokens that are vested according to specific schedules for each user. Users are now able to migrate their vesting schedules to zkSync, which requires Nodle's off-chain system to capture and translate these schedules accurately. Nodle oracles will then submit the schedules to the zkSync `GrantsMigration` contract as proposals. The migration process is completed once the proposal receives approval from a predefined number of oracles.

Additionally, rewards distribution has been transitioned from the Eden parachain to zkSync via the `Rewards` contract. The new system introduces batched rewards submissions, allowing multiple rewards to be signed and submitted in a single transaction. Users are encouraged to participate in batch submissions due to the rewards percentage they earn from performing these operations. However, users who prefer not to wait for batched rewards can still request individual signatures from the backend and submit their rewards separately.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** three points of attention, where one is classified as `High`, one is classified as `Low`, and one is classified as `Informational` . The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

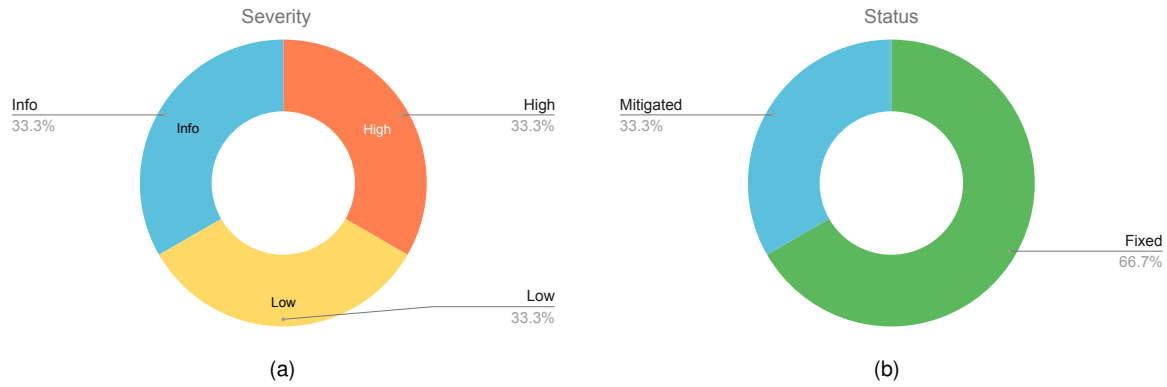

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (0), **Low** (1), **Undetermined** (0), **Informational** (1), **Best Practices** (0). **Distribution of status: Fixed** (2), **Acknowledged** (0), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug 19, 2024
Response from Client	Regular responses during audit engagement
Final Report	Sep 03, 2024
Repository	rollup
Commit (Audit)	c90ba673866defc2100c544278bad0e0d048098f
Commit (Final)	a76f669198cd09ba92bc52412f1a32e0df676ec3
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/Rewards.sol	157	182	115.9%	37	376
2	src/Grants.sol	145	38	26.2%	29	212
3	src/bridge/BridgeBase.sol	86	66	76.7%	39	191
4	src/bridge/GrantsMigration.sol	112	24	21.4%	23	159
	Total	**500**	**310**	**62%**	**128**	**938**

3 Summary of Issues

	Finding	Severity	Update
1	User grants migration can be blocked by exploiting the `MAX_SCHEDULES` limit	High	Fixed
2	Incorrect check of the `batchSubmitterRewardPercentage` parameter during contract construction	Low	Fixed
3	Lack of disabling mechanism in `GrantsMigration` contract	Info	Mitigated

4 System Overview

The reviewed smart contracts are part of Nodle Network's migration from the Eden parachain on Polkadot to zkSync on Ethereum. These contracts handle both the migration of user grants and the transition of the rewards distribution mechanism, which will be halted on Eden parachain and fully transitioned to zkSync.

The following subsections provide an overview of the grants migration process and the new rewards distribution system, detailing the main user flows and components associated with each.

4.1 Grants Migration

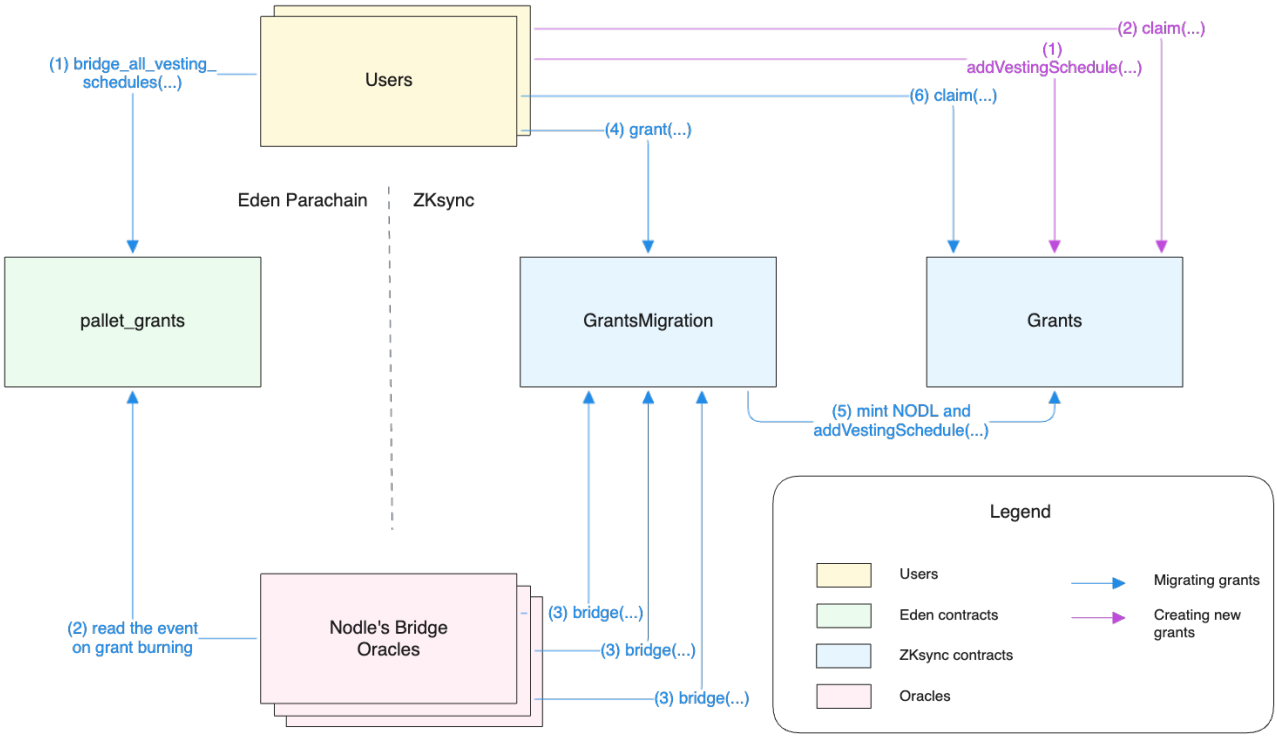

Fig. 2: Nodle's grants migration system overview. The diagram presents only the core flows, excluding standard configuration and admin-specific actions.

The grants migration goes through the following main steps:

1. Initiating Grant Migration: Users can initiate the migration of their vesting schedules to zkSync by invoking the `bridge_all_vesting_schedules(...)` function on the `pallet_grants` contract on the Eden parachain. This function claims the available tokens and transfers them to the user, then emits an event to initiate the migration of remaining locked tokens in the user's vesting schedules.

2. Proposal Submission and Voting: Nodle Bridge Oracles listen for events emitted by the `pallet_grants` contract and submit them as proposals to the `GrantsMigration` contract on zkSync. Each proposal includes the address of the grant recipient, the total amount of the grant, and the array of its schedules:

```
struct Proposal {
    address target; // Address of the grant recipient.
    uint256 amount; // Total amount of tokens to be vested.
    Grants.VestingSchedule[] schedules; // Array of vesting schedules.
}
```

Proposals are submitted on-chain through the `bridge(...)` function:

```
function bridge(bytes32 paraTxHash, address user, uint256 amount, Grants.VestingSchedule[] memory schedules) external
```

When the first oracle submits a proposal, other oracles can vote on it using the same function. Once a proposal gathers a predefined number of votes from trusted oracles (`threshold`), it can be executed after a `safetyDelay` period has passed since the last vote.

3. Proposal Execution and Grants Creation:

Any address can execute a proposal that has received enough votes and where the `safetyDelay` period has elapsed. This is accomplished using the `grant(...)` function:

```
function grant(bytes32 paraTxHash) external
```

This function creates vesting schedules for the target user by invoking the `addVestingSchedules(...)` function on the `Grants` contract. For that, it mints and then transfers the total `amount` to the `Grants` contract.

```
function addVestingSchedule(address to, uint256 start, uint256 period, uint32 periodCount, uint256 perPeriodAmount,
↪  address cancelAuthority) external
```

Note that the `Grants` contract introduces a threshold on the number of vesting schedules a user can have (`MAX_SCHEDULES`); this limit is checked with every addition of new schedules.

4. Claiming Tokens: Once the vesting schedules are migrated to the Grants contract on zkSync, users can call the `claim()` function to claim their tokens according to the predefined schedule.

```
function claim() external
```

4.2 Rewards Distribution

The `Nodle` reward distribution mechanism on zkSync is a hybrid system that integrates both off-chain and on-chain components. The off-chain system handles the computation of rewards for users and generates signed reward messages. Meanwhile, the on-chain component, consisting of the `Rewards` contract, allows the submission of these signed messages and the claiming of rewards by users. Nodle's system supports both individual and batch reward submissions.

The rewards off-chain system consists of the following main components:

- **Public API**: This is the primary interface for Nodle users. It enables users to request signed messages for their rewards or retrieve existing signatures for sequences not yet submitted on-chain.
- **Signer Server**: A private server accessible only via the API. Its role is to sign new reward messages for unused sequence numbers or return existing signatures from the draft database.
- **Draft Rewards Database**: This database maintains records of the issued signatures; it stores information such as the sequence number, the user address, and the reward amount for each signed message. A new entry is added to this database each time the Signer issues a signature.
- **Payout Database**: This database tracks all pending rewards for different users. When a new reward message is requested, the database is queried to retrieve and sum up all the pending rewards for the specified user. The calculated amount is then signed, and the used rows in the database are updated to have a completed status.

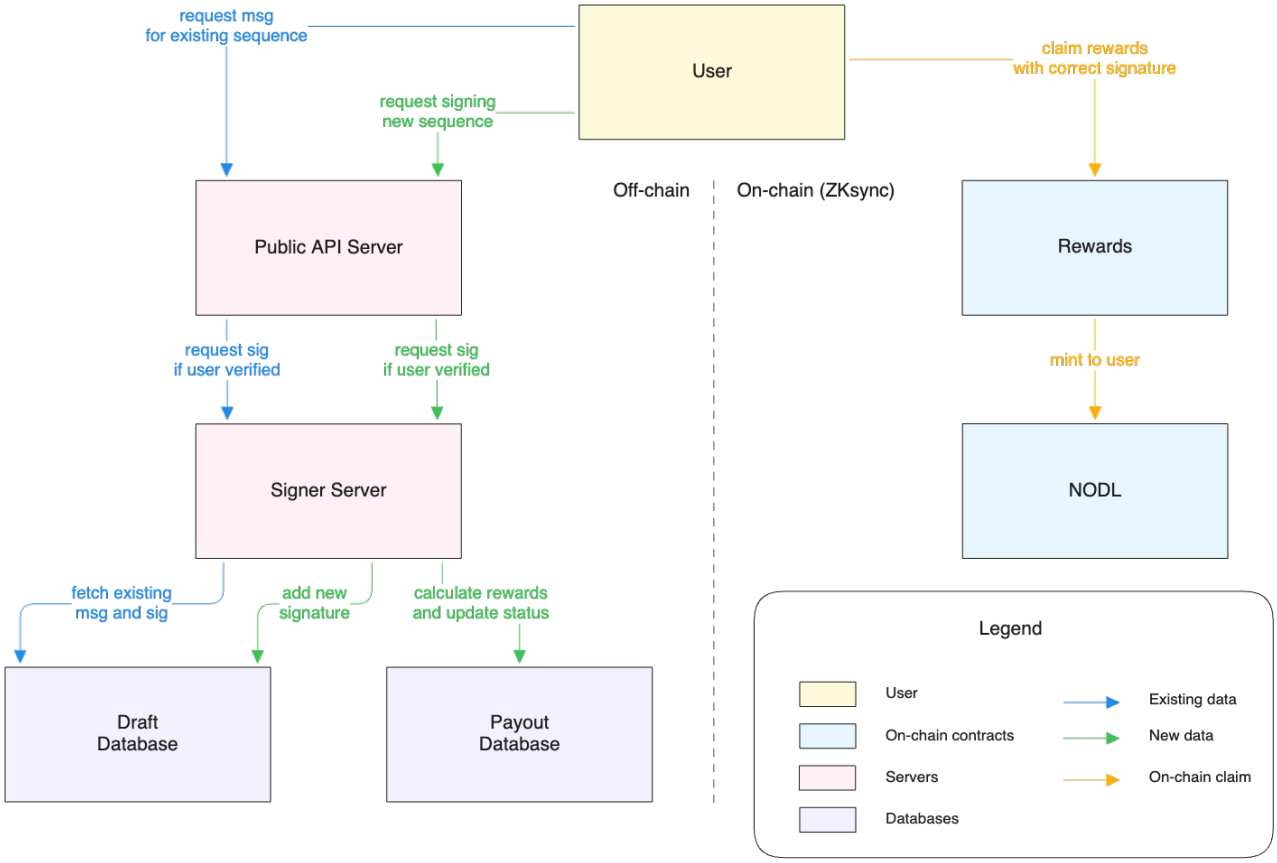

Fig. 2: NODL's rewards system overview. The diagram presents only the core flows, excluding standard configuration and admin-specific actions.

Each user has a unique sequence number that is used to sign and validate individual reward claims, incremented with each successful submission. Similarly, batch rewards have a distinct sequence number, `batchSequence`, which is also incremented with each successful batch submission.

a. **Individual Rewards**: Users who wish to claim rewards manually can request a signed message from the Nodle API. This message includes their address, the total reward amount, and a sequence number. The signed message, along with the relevant information, is then submitted to the `Rewards` contract via the `mintReward(...)` function.

```
function mintReward(Reward memory reward, bytes memory signature) external
```

This function verifies that the signature is valid (signed by the `authorizedOracle`) and that the sequence number matches the user's expected value. Upon validation, the reward amount is minted and transferred to the user, and the sequence number is incremented by 1.

b. **Batch Rewards**: Users can submit a batch of rewards in a single transaction using the `mintBatchReward(...)` function.

```
function mintBatchReward(BatchReward memory batch, bytes memory signature) external
```

To do this, users must request a signed message from the API for a batch of addresses, specifying the `batchSequence` number. This signature is then used to submit the batch, which mints and transfers the specified amounts to each user in the batch. The `batchSequence` number is then incremented by 1.

The Nodle team will automatically handle batch issuance for addresses with the highest rewards. Users are incentivized to participate in the submission process, as they receive a `batchSubmitterRewardPercentage` of the total rewards for their efforts.

Additionally, the contract imposes a `quota` on the number of tokens minted within a given period. This limit is enforced in both the individual and batch mint functions. Nodle governance has the ability to adjust the quota if necessary.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] User grants migration can be blocked by exploiting the MAX_SCHEDULES limit

File(s): Grants.sol

Description: The Grants contract manages users' vesting schedules through the vestingSchedules mapping. The addVestingSchedule(...) function, which is public, enables anyone to add a vesting schedule for any recipient by specifying parameters such as period, start, and perPeriodAmount. This function includes a safeguard to ensure that the number of schedules per user does not exceed the predefined MAX_SCHEDULES limit, set to 100.

However, because the function is public, a malicious actor could exploit it by creating multiple vesting schedules with zero perPeriodAmount and long period values. This abuse would fill the schedule limit with ineffective entries, potentially blocking the GrantsMigration contract from successfully adding new schedules and completing the user migration processes.

```
function addVestingSchedule(
        address to,
        uint256 start,
        uint256 period,
        uint32 periodCount,
        // @audit Amount can be `0`
        uint256 perPeriodAmount,
        address cancelAuthority
    ) external {
        // ...
        // @audit Cannot add vesting schedule if the limit is reached
        _mustNotExceedMaxSchedules(to);

        token.safeTransferFrom(msg.sender, address(this), perPeriodAmount * periodCount);

       VestingSchedule memory schedule = VestingSchedule(cancelAuthority, start, period, periodCount, perPeriodAmount);
        //@audit Function can be repeatedly called with 0 amount until the array reaches the limit
        // Moreover, `perPeriodAmount` can be zero
        vestingSchedules[to].push(schedule);

        emit VestingScheduleAdded(to, schedule);
}
```

Recommendation(s): To mitigate this issue, consider implementing the following recommendations:

1. Eliminate the MAX_SCHEDULES cap and instead implement pagination for functions that iterate through the vesting schedules, such as claim(...), renounce(...), and cancelVestingSchedules(...). This allows users to specify the indices of the schedules they want to interact with, addressing the mentioned issue while maintaining control over iteration limits;
2. Introduce constraints to prevent perPeriodAmount from being zero or excessively small, and set reasonable limits on the period to avoid invalid long durations. This will reduce the incentive for malicious actors, as such an exploit would become costlier and less effective;

Status: Fixed

Update from the client: We are considering to keep the MAX_SCHEDULES limit so all the api of this contract remains as it's on the parachain but taking the second recommendation here: 5adfcef.

Update from the Nethermind Security: The provided change make the attack more costly and so less possible to occur, but still possible. Therefore we recommend to set the limit considerably high. Additionally consider limiting upper range of the period to prevent adding infinitely long periods.

Update from the client: Fixed in 9f13844.

6.2 [Low] Incorrect check of the `batchSubmitterRewardPercentage` parameter during contract construction

File(s): `Rewards.sol`

Description: In the `Rewards` contract, the `constructor` includes a check using the `_mustBeLessThan100(...)` function to ensure that the `batchSubmitterRewardPercentage` is not set above 100. However, this check is ineffective because `batchSubmitterRewardPercentage` is initialized after the check with the value of `rewardPercentage`. At the time of the check, `batchSubmitterRewardPercentage` has not yet been assigned, so the check will always pass regardless of the value of `rewardPercentage`.

```
constructor(
        NODL token,
        uint256 initialQuota,
        uint256 initialPeriod,
        address oracleAddress,
        uint256 rewardPercentage
    ) EIP712(SIGNING_DOMAIN, SIGNATURE_VERSION) {
        // ...
        //@audit `batchSubmitterRewardPercentage` is not initialized
        _mustBeLessThan100(batchSubmitterRewardPercentage);
        // ...
        // @audit `rewardPercentage` value is not validated
        batchSubmitterRewardPercentage = rewardPercentage;
 }
```

As a result, the `batchSubmitterRewardPercentage` could be set to a value greater than `100`.

Recommendation(s): Update the constructor to perform the check on `rewardPercentage`: `_mustBeLessThan100(rewardPercentage);`

Status: Fixed

Update from the client: Fixed in 21b6df7.

6.3 [Info] Lack of disabling mechanism in `GrantsMigration` contract

File(s): `GrantsMigration.sol`

Description: The `GrantsMigration` contract facilitates the migration of users' vesting schedules from the Eden parachain to the Zksync network. It mints tokens for users to complete their migration process, with each execution requiring voting from a set number of trusted oracles. The contract will become deprecated once the migration period, as defined by the Nodle team, expires and all users have completed their migration.

However, the current implementation lacks a mechanism to disable the contract's functions after the migration period concludes. As a result, oracles will retain the ability to execute various functions, including minting new tokens, even after the migration is complete. This ongoing access poses a risk of unauthorized actions or misuse, particularly if the oracles' private keys are compromised or leaked in the future. Thus, without a disabling mechanism, the contract could be vulnerable to exploitation beyond its intended operational timeframe.

Recommendation(s): Implement a mechanism to disable the contract's functionality once the migration is complete.

Status: Mitigated

Update from the client: The mint permission of this contract is removable through the governance of the `NODL` token.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Nodle documentation

The Nodle team was actively engaged during meetings, addressing questions and concerns effectively. This interaction provided the auditing team with a thorough understanding of both the system architecture and the technical details of the project. Additionally, the Nodle team provided a comprehensive documentation, which proved invaluable for the code review process. This documentation detailed the entire migration process and rewards mechanism, including an explanation of the off-chain components and their main interactions.

8 Test Suite Evaluation

8.1 Compilation Output

```
forge build
[] Compiling...
[] Compiling 110 files with Solc 0.8.23
[] Solc 0.8.23 finished in 154.89s
Compiler run successful with warnings:
Warning (2519): This declaration shadows an existing declaration.
  --> lib/zksync-contracts/l2/system-contracts/libraries/Utils.sol:85:9:
   |
85 |         uint256 bytecodeLenInWords = _bytecode.length / 32;
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^
Note: The shadowed declaration is here:
  --> lib/zksync-contracts/l2/system-contracts/libraries/Utils.sol:43:5:
   |
43 |     function bytecodeLenInWords(bytes32 _bytecodeHash) internal pure returns (uint256 codeLengthInWords) {
   |     ^ (Relevant source part starts here and spans across multiple lines).
```

8.2 Test Suite Output

```
forge test
[] Compiling...
No files changed, compilation skipped

Ran 5 tests for test/NODL.t.sol:NODLTest
[PASS] test_defaultACLs() (gas: 15520)
[PASS] test_has18Decimals() (gas: 5771)
[PASS] test_isBurnable() (gas: 50076)
[PASS] test_isMintable() (gas: 64169)
[PASS] test_shouldDeployWithNoSupply() (gas: 7596)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 15.91ms (330.10µs CPU time)

Ran 2 tests for test/contentsign/EnterpriseContentSign.t.sol:EnterpriseContentSignTest
[PASS] test_nonWhitelistedCannotMint() (gas: 13174)
[PASS] test_whitelistedUserCanMint() (gas: 116148)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 15.90ms (509.81µs CPU time)

Ran 3 tests for test/contentsign/BaseContentSign.t.sol:BaseContentSignTest
[PASS] test_nonWhitelistedCannotMint() (gas: 16014)
[PASS] test_setsMetadata() (gas: 14625)
[PASS] test_whitelistedCanMint() (gas: 131529)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 15.91ms (872.77µs CPU time)

Ran 2 tests for test/contentsign/ClickContentSign.t.sol:ClickContentSignTest
[PASS] test_nonWhitelistedCannotMint() (gas: 18648)
[PASS] test_whitelistedUserCanMint() (gas: 119051)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 16.06ms (397.94µs CPU time)

Ran 5 tests for test/paymasters/WhitelistPaymaster.t.sol:WhitelistPaymasterTest
[PASS] test_allowsGeneralFlowOnlyIfWhitelistingPasses() (gas: 76084)
[PASS] test_defaultACLs() (gas: 21296)
[PASS] test_doesNotSupportApprovalBasedFlow() (gas: 13645)
[PASS] test_nonWhitelistAdminCannotUpdateWhitelists() (gas: 36991)
[PASS] test_whitelistAdminUpdatesWhitelists() (gas: 59032)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 16.25ms (740.46µs CPU time)

Ran 11 tests for test/bridge/NODLMigration.t.sol:NODLMigrationTest
[PASS] test_configuredProperToken() (gas: 7866)
[PASS] test_mayNotVoteIfParametersChanged() (gas: 149268)
[PASS] test_mayNotVoteOnceExecuted() (gas: 231895)
[PASS] test_mayNotVoteTwice() (gas: 134697)
[PASS] test_mayNotWithdrawBeforeDelay() (gas: 168987)
[PASS] test_mayNotWithdrawIfNotEnoughVotes() (gas: 135379)
[PASS] test_mayNotWithdrawTwice() (gas: 227225)
[PASS] test_mayWithdrawAfterDelay() (gas: 232830)
```

```
[PASS] test_nonOracleMayNotBridgeTokens() (gas: 13922)
[PASS] test_oraclesAreRegisteredProperly() (gas: 31217)
[PASS] test_recordsVotes() (gas: 171835)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 18.63ms (3.53ms CPU time)

Ran 9 tests for test/bridge/GrantsMigration.t.sol:GrantsMigrationTest
[PASS] test_executionFailsIfInsufficientVotes() (gas: 508373)
[PASS] test_executionFailsIfTooEarly() (gas: 550800)
[PASS] test_executionOfProposals() (gas: 992601)
[PASS] test_oraclesAreRegisteredProperly() (gas: 31317)
[PASS] test_proposalCreationAndVoting() (gas: 552744)
[PASS] test_proposalParameterChangesPreventVoting() (gas: 666862)
[PASS] test_registeringTooManyOraclesFails() (gas: 59026)
[PASS] test_rejectionOfDuplicateVotes() (gas: 511762)
[PASS] test_rejectionOfVoteOnAlreadyExecuted() (gas: 998778)
Suite result: ok. 9 passed; 0 failed; 0 skipped; finished in 21.90ms (4.50ms CPU time)

Ran 20 tests for test/Grants.t.sol:GrantsTest
[PASS] test_CancelVestingSchedulesRedeemsAllIfNoneVested() (gas: 281937)
[PASS] test_CancelVestingSchedulesRedeemsPartiallyVested() (gas: 299668)
[PASS] test_CancelVestingSchedulesRedeemsZeroIfFullyVested() (gas: 296289)
[PASS] test_addVestingScheduleFailsIfNotEnoughToken() (gas: 50292)
[PASS] test_addingTooManyScheduleReverts() (gas: 12159449)
[PASS] test_aliceCanOnlyCancelHerOwnGivenGrants() (gas: 400860)
[PASS] test_cancelAuthorityCouldbeADifferentAccount() (gas: 401396)
[PASS] test_claimAfterOnePeriod() (gas: 231590)
[PASS] test_claimRemovesAllSchedules() (gas: 311463)
[PASS] test_claimRemovesFullyVestedSchedules() (gas: 311373)
[PASS] test_claimSeveralGrants() (gas: 491624)
[PASS] test_nonCancellabelSchedule() (gas: 176071)
[PASS] test_nothingToClaimBeforeOnePeriod() (gas: 204823)
[PASS] test_nothingToClaimBeforeStart() (gas: 203734)
[PASS] test_renounceRevertsIfNoSchedules() (gas: 175152)
[PASS] test_renouncedCannotBeCanceled() (gas: 406278)
[PASS] test_vestingToSelfReverts() (gas: 41298)
[PASS] test_vestingToZeroAddressReverts() (gas: 41836)
[PASS] test_zeroCountReverts() (gas: 43614)
[PASS] test_zeroVestingPeriodReverts() (gas: 43527)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 22.63ms (14.43ms CPU time)

Ran 2 tests for test/paymasters/BasePaymaster.t.sol:BasePaymasterTest
[PASS] test_defaultACLs() (gas: 16957)
[PASS] test_withdrawExcessETH() (gas: 47881)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 22.67ms (136.78µs CPU time)

Ran 14 tests for test/bridge/MigrationNFT.t.sol:MigrationNFTTest
[PASS] test_canLevelUpEvenThoughMaxHoldersWasReached() (gas: 2998493)
[PASS] test_enforceEqualLength() (gas: 351564)
[PASS] test_enforceSorting() (gas: 375229)
[PASS] test_initialState() (gas: 47741)
[PASS] test_isSoulbound() (gas: 378805)
[PASS] test_mint() (gas: 394958)
[PASS] test_mintFailsIfAlreadyClaimed() (gas: 377592)
[PASS] test_mintFailsIfNoMoreLevels() (gas: 698819)
[PASS] test_mintFailsIfNotExecuted() (gas: 143274)
[PASS] test_mintFailsIfProposalDoesNotExist() (gas: 23068)
[PASS] test_mintFailsIfTooManyHolders() (gas: 2947102)
[PASS] test_mintFailsIfUnderMinimumAmount() (gas: 223602)
[PASS] test_mintFullLevelUp() (gas: 634261)
[PASS] test_mintLevelUpOneByOne() (gas: 1254979)
Suite result: ok. 14 passed; 0 failed; 0 skipped; finished in 29.66ms (13.56ms CPU time)

Ran 23 tests for test/Rewards.t.sol:RewardsTest
[PASS] test_changingSubmitterRewardPercentageIsEffective() (gas: 222742)
[PASS] test_digestReward() (gas: 11040)
[PASS] test_gasUsed() (gas: 13843964)
[PASS] test_mintBatchReward() (gas: 197682)
[PASS] test_mintBatchRewardInvalidDigest() (gas: 28555)
[PASS] test_mintBatchRewardInvalidSequence() (gas: 22465)
[PASS] test_mintBatchRewardInvalidStruct() (gas: 19695)
[PASS] test_mintBatchRewardOverflowsOnBatchSum() (gas: 30665)
```

```
[PASS] test_mintBatchRewardOverflowsOnSubmitterReward() (gas: 212002)
[PASS] test_mintBatchRewardQuotaExceeded() (gas: 36276)
[PASS] test_mintBatchRewardUnauthorizedOracle() (gas: 28418)
[PASS] test_mintReward() (gas: 128828)
[PASS] test_mintRewardInvalidDigest() (gas: 27272)
[PASS] test_mintRewardInvalidsequence() (gas: 19099)
[PASS] test_mintRewardQuotaExceeded() (gas: 29593)
[PASS] test_mintRewardUnauthorizedOracle() (gas: 24715)
[PASS] test_rewardsClaimedAccumulates() (gas: 290118)
[PASS] test_rewardsClaimedResetsOnNewPeriod() (gas: 270137)
[PASS] test_setBatchSubmitterRewardPercentage() (gas: 46561)
[PASS] test_setBatchSubmitterRewardPercentageOutOfRange() (gas: 40219)
[PASS] test_setBatchSubmitterRewardPercentageUnauthorized() (gas: 13016)
[PASS] test_setQuota() (gas: 48704)
[PASS] test_setQuotaUnauthorized() (gas: 13531)
Suite result: ok. 23 passed; 0 failed; 0 skipped; finished in 29.68ms (48.02ms CPU time)

Ran 11 test suites in 320.92ms (225.20ms CPU time): 96 tests passed, 0 failed, 0 skipped (96 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology 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, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.