Audit Report

Rusk Consensus

v1.0

September 20, 2024

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Dusk to perform a security audit of the consensus implementation of Rusk.

The objectives of the audit are as follows:

1. Determine the correct functioning of the system, in accordance with the system specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/dusk-network/rusk
Commit	`2b2b8f5c9efb117fd68ee071cd04de1c998f1620`
Scope	`consensus/`
Fixes verified at commit	`253ff91ce8becc1384bccd6393a5f717f5e67f0a` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Rusk is the official Dusk protocol node client and smart contract platform. It is written in Rust and adheres to the Dusk protocol specifications.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	High	-
Level of documentation	Medium-High	-
Test coverage	Medium	-

Summary of Findings

No	Description	Severity	Status
1	Unlimited transactions in proposal blocks can lead to denial of service	Critical	Resolved
2	Committee members can vote multiple times	Major	Resolved
3	Committee members can submit conflicting votes	Major	Resolved
4	Emergency block is unimplemented	Major	Acknowledged
5	Slashing for block generator is disabled after 10 iterations	Major	Acknowledged
6	Inefficient order of checks in `verify_new_block` can be exploited by attackers to overload the system.	Major	Resolved
7	Different calculation logic for majority and supermajority	Minor	Resolved
8	Iteration context is not reset when new iteration begins	Minor	Acknowledged
9	Suboptimal usage of `BTreeMap`	Minor	Acknowledged
10	Unclear purpose of `RUSK_CONSENSUS_SPIN_TIME`	Informational	Acknowledged
11	Hard coded block gas limit	Informational	Resolved
12	Invariants are not enforced	Informational	Acknowledged
13	Suboptimal construction of the bitset	Informational	Resolved
14	Suboptimal filtering of messages	Informational	Acknowledged
15	Unresolved `FIXME` commentary	Informational	Acknowledged
16	Potential vulnerabilities in consensus incentives due to incomplete slashing implementation	Informational	Resolved
17	Unused definitions	Informational	Resolved
18	Ungraceful error handling	Informational	Acknowledged

Detailed Findings

1. Unlimited transactions in proposal blocks can lead to denial of service

Severity: Critical

In `rusk/consensus/src/proposal/handler.rs:80-112` the `verify_new_block` function of the `ProposalHandler`, there is no limit on the number of transactions in a candidate block. The operation `p.candidate.txs().iter().map(|t| t.hash()).collect()` iterates over all transactions to calculate their hashes before computing the merkle root. An attacker could exploit this by submitting a candidate block with an extremely large number of transactions, causing excessive computation time and potential denial of service.

Recommendation

We recommend implementing an adequate limit on the number of transactions allowed in a candidate block, enforced before calculating transaction hashes and the merkle root.

Status: Resolved

2. Committee members can vote multiple times

Severity: Major

The function `collect_vote` in `consensus/src/aggregator.rs` should ensure that each committee member can only vote once per round. However, it only contains a `debug_assert` in line 96 which will panic when a committee member has already voted. But because `debug_assert` instead of `assert` is used, this will only be done for non optimized builds, i.e. not for production builds.

Recommendation

We recommend returning an error when a member tries to vote a second time.

Status: Resolved

3. Committee members can submit conflicting votes

Severity: Major

A committee member that has voted `valid` during the validation step could vote `invalid` during the ratification step. This could be used by committee members to obfuscate malicious behavior, while sabotaging the consensus.

Recommendation

We recommend returning an error when a member votes with a conflicting vote in the first step.

Status: Resolved

4. Emergency block is unimplemented

Severity: Major

According to the documentation an emergency empty emergency block is produced after 255 iterations. This is not the case. Unimplemented production relevant features are a major issue.

Recommendation

We recommend implementing the emergency block.

Status: Acknowledged

Team response: While the introduction of the Emergency Block is still planned (see Issue#1234), the introduction of the so-called "open consensus" mode mitigates the risk of having the network stalled due to missing Provisioners. Open consensus mode is enabled by nodes upon reaching the timeout on the very last iteration, and consists in keeping Emergency Iterations open indefinitely.

In other words, if the last iteration is reached, nodes will keep collecting candidates and votes for all previous emergency iterations. This way, if missing provisioners come back online, they can participate to pending iterations by generating candidates and voting on it. If any of such iterations reaches a quorum from both committees, the candidate is accepted and a new round is started.

The Emergency Block can still help in case missing provisioners do not ever come back online.

5. Slashing for block generator is disabled after 10 iterations

Severity: Major

In `consensus/src/proposal/step.rs:62`, the constant `config::RELAX_ITERATION_THRESHOLD` is used to limit the number of iterations from which we extract failed attestations. Currently, this constant is declared to be the number 10.

After the number of iterations in a single round reaches 10, only the 10th iteration is used to extract failed attestations, even if the 50th iteration is in progress.

Failed attestations are used in `rusk/src/lib/chain/rusk.rs` to extract generators to slash them for inactivity, see the function `reward_slash_and_update_root` (line `538`).

As a consequence of using this limit, generators from iterations `11`, `12`, `13` etc. are not slashed for neglecting their duties. Malicious generators can safely idle till the end of round if the block has not been confirmed during the first `10` iterations.

Recommendation

We recommend removing the `RELAX_ITERATION_THRESHOLD` constant and extracting failed attestations from all iterations.

Status: Acknowledged

Team response: This is a design choice based on the assumption that reaching the `RELAX_ITERATION_THRESHOLD` is a sign of a network-wide issue, and not due to a specific inactive generator. While it's true that generators after iteration 10 can safely skip their turn, it is also true that there is no interest nor gain for them in doing so, as they would only lose the potential profit of the block reward. In this respect, it is important to remark that the primary goal of soft-slashing (to which Relaxed Iterations refer) is to help the network deal with offline provisioners, rather than punish them. In fact, soft-slashed provisioners do not lose their money, as the slashed amount is simply locked, and can be retrieved by unstaking).

Also note that, in Emergency Mode, which is activated 6 rounds after Relaxed Mode, Fail Attestations are not produced at all, since iterations are kept "open" indefinitely.

Given the above, there currently is no plan to change this behavior. Nonetheless, we might consider increasing `RELAX_ITERATION_THRESHOLD` to coincide with `EMERGENCY_MODE_ITERATION_THRESHOLD` in the future.

6. Inefficient order of checks in `verify_new_block` can be exploited by attackers to overload the system.

Severity: Major

In `rusk/consensus/src/proposal/handler.rs:80-112` the function `verify_new_block`, executes the check `expected_generator!= p.sign_info.signer.bytes()` at the end of the function. This means computationally intensive operations like signature verification, previous block hash checking, and merkle root calculation are performed before confirming the block came from the expected generator. Attackers can exploit this even if the issue Unlimited transactions in proposal blocks can lead

to denial of service is fixed to cause an excessive number of unnecessary calculations and potentially overload the system.

Recommendation

We recommend moving the generator check to the beginning of `verify_new_block`, immediately after unwrapping the message, to ensure only the expected generator can trigger resource-intensive operations, and can be slashed adequately in the case of malicious behavior.

Status: Resolved

7. Different calculation logic for majority and supermajority

Severity: Minor

In the file `consensus/src/user/committee.rs`, the required credits for the majority and supermajority are calculated in lines `41-45`. For the supermajority, the credits are determined by multiplying the committee credits with the threshold and taking the ceiling of this result. For the majority, the committee credits are also multiplied with the majority threshold, but this is then casted to a `usize` and 1 is added to the result. In most cases, these two approaches are equivalent. However, when the committee credits are an exact multiple of `MAJORITY_THRESHOLD`, the required credits are higher by one. For instance if there are 4 committee credits with the `MAJORITY_THRESHOLD` set to 0.5 and the `SUPERMAJORITY_THRESHOLD` set to 0.67, the required credits will be 3 for both.

If the `MAJORITY_THRESHOLD` and `SUPERMAJORITY_THRESHOLD` were set to the same value, this would also mean that the required credits for the majority would be higher than for the supermajority, which does not make sense. Moreover, if it was set to 1.0, it would never be reachable.

Recommendation

We recommend using `ceil` for both values.

Status: Resolved

8. Iteration context is not reset when new iteration begins

Severity: Minor

Within `consensus/src/iteration_ctx.rs:105-108` the function `on_begin` is defined. This function is called when a new iteration begins, and it updates the iteration counter inside the `IterationCtx` structure. However, this increment is not consistent with the other fields of the structure, namely `join_set` and `timestamps`.

The `join_set` variable tracks all asynchronous tasks started during an iteration. If it's not reset when the new iteration begins, computation from the previous iteration might still be running and cause inconsistent state changes. The `timeouts` variable is involved in the adaptive timeout algorithm and contains current values of delays. New iteration involves a different committee with different availability, so `timeouts` should be reset to its default value.

Recommendation

We recommend aborting all ongoing asynchronous tasks tracked in `join_set`, and reset it together with `timeouts` to their default values.

Status: Acknowledged

Team response: With respect to `join_set`, all tasks are aborted by the `on_close` method. So there is no need to reset this variable in `on_begin`.

For what concerns timeouts, they are set at the beginning of the round and updated in case of step timeout event. So they should not be reset.

The actual timeout reset for the iteration is done in the event_loop with the deadline timeout.

9. Suboptimal usage of `BTreeMap`

Severity: Minor

The file `consensus/src/user/provisioners.rs` demonstrates a wise choice of data structure to track all available provisioners, namely `BTreeMap`. However, the structure is not utilized to the maximum of its capabilities.

The set of provisioners eligible for participation in the current round committee is computed by iterating all provisioners (line `154`), filtering them by round and minimum stake (line `155`), cloning their stakes (line `245`), comparing each public key with an excluded one (line 251) and, finally, constructing a new map from the iterator (line `252`).

These iterations are excessive and reduce tolerance to DoS attacks against the network.

Recommendation

We recommend utilizing standard functions `retain` and `remove` provided by the `BTreeMap` structure. The former can be used to filter the map in-place, while the latter will help remove the public key with asymptotic complexity `O(log(N))` instead of `O(N)`.

Status: Acknowledged

Team response: The team agrees on the issue and the recommended solution, whose implementation is currently planned (Issue #2146) for version 1.2.0 of the protocol.

10. Unclear purpose of `RUSK_CONSENSUS_SPIN_TIME`

Severity: Informational

The function `consensus_delay` within `consensus/src/consensus.rs` reads the environment variable `RUSK_CONSENSUS_SPIN_TIME`. This is then added to `UNIX_EPOCH` (the start of the year 1970 in UTC) and the system waits until this timestamp. After that, the environment variable is unset. This function is called every time when a new iteration is being started. Although after reaching the specified date and time the delay is always zero, the environment variable is still queried every time.

It is unclear what the purpose of this variable is and the usage of it can be confusing. The user would need to set the environment variable to a unix timestamp and the waiting will therefore only happen once in practice, as the time will be larger in any consecutive calls. Additionally, during this waiting period, the consensus engine still runs iterations despite the network not being launched yet.

Recommendation

We recommend removing this variable. If it is desired that the user can configure a delay before every consensus invocation, the variable should contain a time in seconds that is added to the current timestamp instead.

Status: Acknowledged

Team response: This variable is used for testing purposes, with the goal of having all nodes spin up at the same time. As such, we do not plan to remove this variable. However, the purpose of the constant will be documented in version 1.1.0 of the protocol (see Issue #2147).

11. Hard coded block gas limit

Severity: Informational

The file `consensus/src/proposal/block_generator.rs` sets the block limit to the constant `DEFAULT_BLOCK_GAS_LIMIT`. For various blockchains, the block gas limit varies over time, in which case the software would need to be recompiled.

Recommendation

We recommend considering adding a configuration option for the block gas limit with the default value `DEFAULT_BLOCK_GAS_LIMIT`.

Status: Resolved

12. Invariants are not enforced

Severity: Informational

The following locations in the codebase demonstrate the pattern:

- `consensus/src/iteration_ctx.rs:45`
- `consensus/src/iteration_ctx.rs:142`
- `consensus/src/user/provisioners.rs:224-225`

The pattern consists of creating an iterator and taking the very first element by the iterator. After that, the other elements are discarded. This pattern is used to retrieve the only member of a committee, which is interpreted as a block generator. While there is no evidence that this pattern can lead to an inconsistent state, it is a good practice to enforce assumed invariants.

Another invariant that could be enforced is hidden within `consensus/src/user/provisioners.rs:190-199`. The variable `total_weight` is checked against `subtracted_stake`, and if it is lower than that the iteration is stopped using the `break` expression. However, `total_weight` is initialized with the sum of all stakes, and it is reduced by the same amount that was subtracted from a stake. It cannot be exceeded, so instead of branching the control flow it is better to enforce this invariant.

Recommendation

We recommend using `debug_assert!` with conditions like `committee.size() == 1` and `total_weight > subtracted_stake`.

Status: Acknowledged

Team response: The team agrees with the issue and the recommended solution, which is planned for version 1.1.0 of the protocol (Issue #2186).

13. Suboptimal construction of the bitset

Severity: Informational

In file `consensus/src/user/committee.rs` within lines `101-108` a nested iteration implemented, with the purpose of constructing a membership bitset. During this nested iteration, elements of collections `voters` and `members` are compared against each other to discover positions of those bits which must be set to `1`. Asymptotic complexity of this nested iteration is $O(n^2)$, and each iteration compares 96 byte long public keys.

`BTreeMap` provides developers with a function `contains_key` implementing membership check with asymptotic complexity $O(\log(n))$.

This issue is reported with informational severity, because the overall number of members and voters is not high.

Recommendation

We recommend using standard `contains_key` function and implement the iteration with complexity `O(n*log(n))`.

Status: Resolved

14. Suboptimal filtering of messages

Severity: Informational

In file `consensus/src/user/queue.rs` within lines `61-67`, filtering of messages is implemented. Only messages belonging to specific rounds range are preserved. The range is specified using parameters `start_round` and `end_round`, the former is used in a call to `split_off` function allowing to move all elements after the specified position into a new map, while the latter is used to filter the moved elements.

The collection messages can be large, containing messages from many rounds and iterations. This approach in message filtering results in redundant iteration through messages belonging to rounds higher than `end_round`.

Recommendation

We recommend using the standard function `range` and iterating only through messages of those rounds which should be preserved.

Status: Acknowledged

Team response: The team agrees with the issue and the recommended solution, which is planned for version 1.2.0 of the protocol (Issue #2187).

15. Unresolved `FIXME` commentary

Severity: Informational

In `consensus/src/quorum/verifiers.rs:148-152` a comprehensive commentary is documented covering an edge case of handling the membership bitset. However, there is also a `FIXME` commentary highlighting that the correct handling is not implemented.

All `FIXME` and `TODO` comments should be resolved before using the codebase in production. Since the source code is publicly available, an attacker can study it and navigate the attack surface using `FIXME` and `TODO` comments as a map.

Additionally, from a development perspective, `FIXME` and `TODO` comments become stale easier than proper tasks tracked in an issue tracker.

Recommendation

We recommend removing the task from comments and registering it in an issue tracking system.

Status: Acknowledged

Team response: The team agrees with the issue and the recommended solution. A complete review of TODOs and FIXMEs is due (Issue #2187). We plan to convert remove all such items from the code and convert them into Issues, if necessary.

16. Potential vulnerabilities in consensus incentives due to incomplete slashing implementation

Severity: Informational

While slashing is an integral part of the consensus, it was out of scope for this audit. The following considerations are hypothetical but could have major or critical implications if slashing is not properly implemented:

1. There is a strong incentive per iteration for validators to vote against a block or delay voting, particularly for the generator of the next iteration. This issue arises from the predictability of the next generator, as highlighted in the theoretical review.
2. The large number of iterations (up to 255) may require a complex slashing design to ensure early iterations are preferred over later ones by generators. Otherwise they might try
3. After 50 iterations, voting for previously suggested blocks becomes possible. This feature could be exploited by a blocking minority to delay voting until they can back-vote over several past blocks, potentially choosing a block that benefits minority members. Such a benefitting block could be a block that has the shortest route until the next generator.

Recommendation

We recommend

1. Considering reducing the number of iterations, possibly to only 1x3 Steps, before an empty block is produced.
2. Considering the exclusion of future generators from voting, or implementing harsher slashing for them.
3. Reviewing the necessity and implications of allowing back-voting after 50 iterations.
4. Implementing slashing and considering at least the following slashable events
    a. Proposing and signing two different blocks for the same slot
    b. Double voting
    c. Conflicting votes
    d. Downtime, particularly during block production
    e. Manipulation of timestamps

Status: Resolved

17. Unused definitions

Severity: Informational

The following locations contain function or constant definitions which are not used in the codebase and can be safely removed:

Location in the codebase	Name of the definition
`consensus/src/queue.rs:51-54`	`remove_msgs_greater_than`
`consensus/src/commons.rs:134`	`delete_candidate_blocks`
`consensus/src/config.rs:21-22`	`CONSENSUS_DELAY_MS`
`consensus/src/user/cluster.rs:40-41`	`add`
`consensus/src/proposal/handler.rs:60-67`	`collect_from_past`

The last of aforementioned definitions, function `collect_from_past`, is called in `consensus/src/iteration_ctx.rs:155`. However, the call can be safely removed since the function `collect_from_past` has no side effect and simply returns a value, which is then ignored using the _ pattern.

Recommendation

We recommend pruning unused definitions from the codebase.

Status: Resolved

18. Ungraceful error handling

Severity: Informational

Throughout the codebase `anyhow` and `.expect()` are used to handle errors. There were no vulnerabilities found related to this. However, the use of these patterns might lead to obfuscated errors or panics in production. This might become particularly relevant if the codebase is upgraded at a later stage.

Recommendation

We recommend graceful error handling that addresses errors individually and avoiding panics in production.

Status: Acknowledged

Team response: The recommended measure is planned for version 1.2.0 of the protocol (Issue #2189)