Centrifuge Baseline Security Assurance

Threat model and hacking assessment report

V2.3, Sep 5, 2022

Regina Bíró	regina@srlabs.de
Christopher Talib	christopher@srlabs.de
Mostafa Sattari	mostafa@srlabs.de

Abstract. This work describes the result of the thorough and independent security assurance audit of the Centrifuge blockchain platform performed by Security Research Labs. During this study, Centrifuge provided access to relevant documentation and supported the research team effectively.

The code of Centrifuge was verified to assure that the business logic of the product is resilient to hacking and abuse risks and that the users have their investments secured.

The research team identified several issues ranging from informational to high severity. Centrifuge, in cooperation with the auditors already remediated to a subset of the identified issues.

In addition to the implementation to the remaining open issues, Security Research Labs recommends applying the business logic recommendations as well as improving documentation processes and implement extended testing to catch inconsistencies early in the development process.

Content

1 Motivation and scope

Centrifuge is a blockchain network that will be deployed on the Polkadot relay chain as a parachain, built on top of Substrate. It allows users to invest in loans based on real world assets. Those assets are represented as tokens and set up in investment pools from which users can invest. Centrifuge allows users to invest from different risk models ("tranches") enabling more fine grain investment models.

This report details the baseline security assurance results with the aim of creating transparency in three steps:

Threat Model. The threat model is considered in terms of *hacking incentives*, i.e., the motivations to achieve the goals of breaching the integrity, confidentiality, or availability of nodes in future Centrifuge systems. For each hacking incentive, hacking *scenarios* were postulated, by which these goals could be reached. The threat model provides guidance for the design, implementation, and security testing of Centrifuge.

Security design coverage check. Next, the Centrifuge design was reviewed for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

Implementation baseline check. As a third step, the current Centrifuge implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

Centrifuge is built upon Substrate, a blockchain development framework. Both Centrifuge and Substrate are written in Rust, a memory safe programming language. Mainly, Substrate works with three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, GRANDPA finality gadget and the BABE block production engine.

The Centrifuge runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute the runtime code either natively or will execute the on-chain WASM blob.

With the help of Centrifuge, SRLabs created an overview [1] containing the current state of the runtime modules used by Centrifuge and its audit priority. The in-scope components and the assigned priorities are reflected in Table 1. SRLabs conducted the audit fully focusing on the components indicated as in scope by Centrifuge. Centrifuge's online documentation [2] provided the testers a good runtime module design and implementation overview.

Repository	Priority	Component(s)
Centrifuge-chain	High	pools, loan, permissions
	Medium	anchors, fees, registry, restricted-tokens
	Low	claims, crowdloan-claim

Table 1. In-scope Centrifuge components with audit priority

2 Methodology

To be able to effectively review the Centrifuge codebase, a threat-model driven code review strategy was employed. For each identified threat, hypothetical attacks that can be used to realize the threat were developed and mapped to their respective threat category as outlined in Chapter 3.

Prioritizing by risk, the codebase was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example, the relevant pallets.
2. Identified viable strategies for the code review. Manual code audits, fuzz testing, and manual tests were performed where appropriate.
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, otherwise, ensured that sufficient protection measures against specific attacks were present.
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations.

During the audit, we carried out a hybrid strategy utilizing a combination of code review and dynamic tests (e.g., fuzz testing) to assess the security of the Centrifuge codebase.

While fuzz testing and dynamic tests establish a baseline assurance, the focus of this audit was a manual code review of the Centrifuge codebase to identify logic bugs, design flaws, and best practice deviations. We used the *parachain* branch of the Centrifuge repository as the basis for the review. The approach of the review was to trace the intended functionality of the runtime modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Centrifuge codebase is entirely open source, it is realistic that a malicious actor would analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code that handles untrusted input, which in Centrifuge's case is mostly the functions implementing the extrinsics. (Note that the network part is handled by Substrate, which was not in scope for this review, but is built with a strong emphasis on security and where fuzz testing is also used). Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz testing methods written for this assessment utilized the test runtime Genesis configuration as well as mocked externalities to execute the fuzz test effectively against the extrinsics in scope.

During the audit, findings were shared via a private Github repository [3]. This repository was also utilized to document status updates and the overall review progress – in addition, weekly jour fixe meetings were held to provide detailed updates and address open questions.

3 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in Centrifuge's blockchain system. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized into *low*, *medium*, and *high*, considers the incentive of an attacker, as well as the effort required by an attacker to successfully execute the attack. The hacking value is calculated as:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While *incentive* describes what an attacker might gain from performing an attack successfully, *effort* estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat.
- Medium: Attacks offer the hacker considerable gains from executing the threat.
- High: Attacks offer the hacker high gains by executing this threat.

Effort:

- Low: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources.
- Medium: Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge.
- High: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors.

A more detailed description of the threat modelling framework is summarized in the shared Github repository [4].

After applying the framework to the Centrifuge system, different threat scenarios according to the CIA triad (Confidentiality, Integrity and Availability) were identified. Table 2 provides a high-level overview of the threat model with identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them are described in the threat model deliverable.

For Centrifuge's assessment, Confidentiality issues were deemed to be out of scope.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Integrity	High	- Compromise investment pools - Manipulate loan pricing and interests - Submit malicious or biased LP solution for an epoch - Gain unfair advantage from epoch manipulation	Medium	- Submit a malicious LP solution to crash nodes - Exploit logic bug to lock up anchors for other users - Cheaply fill up blockchain storage
Availability	High	- DoS collators (either directly or by corrupting their reputation) - Tamper permissions - Lock up anchors for legitimate owners	Low	- Submit a malicious LP solution to crash nodes - Exploit logic bug to lock up anchors for other users - Cheaply fill up blockchain storage

Table 2. Threat scenario overview. The threats for Centrifuge's blockchain were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

4 Findings summary

We identified **12** issues - summarized in Table 3 - during our analysis of the runtime modules in scope in the Centrifuge codebase that enable the attacks outlined above. In summary, 7 high severity, 2 medium, 2 low severity and 1 information-level issues were found.

Type	Issue	Severity	References	Status
Business logic	Out of bounds access due to missing pre-check in *update_invest_order* / *update_redeem_order* causes panic	High	[5]	Mitigated [6]
Best practices	Insufficient extrinsics weights	High	[7]	Mitigated [8]
Best practices	Multiple missing storage deposits	High	[9] , [10] , [11]	Mitigated [12] [13] [14]
Business logic	Integer overflow via *WriteOffGroup overdue_days*	High	[15]	Mitigated [16]
Miscellaneous	An attacker can congest transaction processing when spamming a lot of calls to the *claim*	High	[17]	Mitigated [18]

	function, because the calls are not charged			
Business logic	Potential deadlock in pool if no solution better than *no_execution_solution* exists	Medium	[19]	Mitigated [20]
Business logic	Potential spamming preventing honest solutions to be submitted	Medium	[21]	Mitigated [22]
Business logic	The weight calculation underestimates the computational complexity of the *write_off* and *admin_write_off* extrinsics	Low	[23]	Mitigated [24]
Business logic	Fixed minimum epoch duration may bring unfair advantages to attackers in case of a "bank run" after bad news get released	Low	[25]	Open
Business logic	Interest rates for tranches can be changed retroactively	Low	[26]	Mitigated [22]
Business logic	The update extrinsic does not perform any sanity checks on pool settings items	Info	[27]	Mitigated [28]

Table 3 Issue summary

5 Detailed findings

5.1 Issues affecting the business logic

In this section, we summarize the issues affecting the business logic of Centrifuge. The vulnerabilities have been discovered by following the threats identified from the threat model SRLabs has prepared during the assessment.

Centrifuge's core business logic is implemented in the pools and loans pallets. Tampering with these components could lead to integrity or availability issues for the users, and in most cases, direct financial loss. Loans and investments must be protected against logic errors or malicious behavior.

5.1.1 Out of bounds access could lead to DoS

We discovered one case of out of bounds access in the *pools* pallet [5]. As a result of missing sanity checks on parameters in two extrinsics (*update_redeem_order* and *update_invest_order*), a malicious user could inject a malicious payload while calling those extrinsics. As an attacker could crash collator nodes by a crafted extrinsic call that leverages this vulnerability, this could lead to a Denial of Service on the whole chain undermining the chain's availability.

The issue was mitigated by Centrifuge in PR 595 [6] by using functions that perform bounds checking.

5.1.2 Integer overflows could invert state of a loan

Integer overflows are common in Rust, they are also easy to prevent. Nonetheless, their impact could gravely affect the state of chain, apart from crashing nodes that are compiled in debug mode or with overflow checks enabled. Since Centrifuge is implementing a financial service, it is crucial for users that the calculations are correct and that an unexpected numerical input triggers an error state instead of an unknown erroneous state. We recommend using safe math functions whenever sensible.

We found a possible integer overflow that could lead to healthy loans being flagged as unhealthy [15]. This would prevent other investor to invest in that loan and would spread erroneous information on the state of the loan.

This issue was mitigated by using checked math functions via PR 597 [16].

5.1.3 Lack of sanity checks could prevent investments

Numerical values need to be checked for unexpected or unwanted values. The pool settings, defined by a *PoolAdmin*, weren't subject to such checks which could lead to unwanted consequences for the users and their investments in case one of the *min_epoch_time, challenge_time* or *max_nav_age* were set to 0 [27].

Setting *challenge_time* to 0 has the consequence that it will disable the challenge time for solutions which enable users to close and epoch with their (maybe biased) solutions without giving others a chance to submit a fairer solution. The two other parameters set to 0 would create an impossible investment period for the investors.

This issue was fixed by Centrifuge [28].

5.1.4 Potential deadlocks leading to DoS

SRLabs discovered a potential deadlock condition in the epoch execution. In case of a situation where no orders can be executed without making the pool state *Unhealthy*, the current logic will not presume the epoch execution with the *no_execution_solution* score [29].

This would prevent the epoch from being executed and closed, leaving it in a perpetual unstable stage. Investors wouldn't be capable of retrieving their tokens if this happens.

This issue was fixed by Centrifuge [20].

5.1.5 Interest manipulations

Considering the financial nature of Centrifuge, it is of utmost importance that interest rates shouldn't be manipulated in order to prevent some investors gaining unfair advantage. We found that it is possible for a *PoolAdmin* to change the interest rates of tranches through the *update_tranches* extrinsic retroactively, as the result is effective from the moment when *update_tranche_debt* has been called last. Changing the interest rate may undermine the trust in the ecosystem.

This issue was fixed by Centrifuge [22].

5.1.6 Unfair advantages in epochs

Investors might be very sensitive to rumors and in case of a bank run, it must be prevented that the pool runs low on liquidity and prevents investors who want to redeem their tokens later to do so. We discovered a possible issue of running low on liquidity in case of a bank run at the end of an epoch. The issue could lead some investors to suffer serious losses from not being able to redeem their invested tokens. This would create an unfair advantage to investors who react more quickly.

At the time of writing, this issue [25] is still open – as the logic is identical to Tinlake's, Centrifuge is accepting the risk until redesigning both protocols in the future. After discussing the risk with the Centrifuge team, we decided to lower the risk from "high" to "low", as the pool reserve is limited in the current implementation to reduce cash drag.

5.1.7 Honest solutions could be prevented from being submitted

Users can submit a solution calculated off-chain to close an epoch. Those transactions could be tipped to be given priority treatment. However, one or more malicious users could instrument the tipping system to submit a bad or biased solution by issuing some bogus transactions with tips in order to block other submitted solutions from being processed. In this case, honest solutions could be prevented from being submitted within the challenge period.

This issue was fixed by Centrifuge [22].

5.2 Missing best practices

The following issues are not directly affecting the business logic of Centrifuge, as they point to (partially) missing Substrate best practices that are recommended to make the project more sustainable and mature.

5.2.1 Proper weight calculation

When implementing weights for extrinsics, two things should be kept in mind. Firstly, making sure that all extrinsics are benchmarked before going into production. Secondly, not underestimating the computational complexity of the extrinsics. A best practice Parity Tech is implementing, is annotating the extrinsics with their computational complexity in the code comments, which makes the weight function implementation and benchmarking more straightforward. In both two cases the weight will be underestimated and thus it will enable an attacker to produce blocks that take longer to execute than their assigned weight indicators. This will cause validators/collators miss their block production slot and thus stalling the chain.

We identified several issues linked to default [7] or underestimated weights [23] which were mitigated by Centrifuge respectively in PR 609 [8] and PR 621 [24].

5.2.2 Missing storage deposits

Missing storage deposits can allow a malicious user to fill up the storage cheaply by calling affected extrinsics multiple times within a short time period. This leads the node to have a large storage and makes it much harder and more expensive be part of the network. This might prevent some users to get involved with Centrifuge's network. SRLabs detected such issues for the *create* [9] extrinsic, the *approve_role_for* [10] extrinsic in the Pools pallet and the *pre_commit* extrinsic in the Anchor pallet [11].

All missing storage deposit issues were fixed by Centrifuge [12], [14].

5.3 Miscellaneous

This section gathers the issues that do not fit in the previous categories.

5.3.1 Spamming attacks

Spamming attacks come from unsigned extrinsics as they don't require a fee to be called. The lack of fee will allow users to call repetitively and thus congesting the block and could lead to a Denial of Service. We discovered such an issue in the *claims* pallet [17].

This issue was fixed by Centrifuge by changing the unsigned extrinsic in the claims pallet to a signed extrinsic [18].

6 Evolution suggestions

6.1 Business logic suggestions

Tracking epoch time in blocks instead of seconds. In the current implementation, the *ChallengePeriod* of 2 minutes for epochs start after someone submits the first epoch solution. To ensure that other users can submit their solutions for an epoch, we recommend measuring the time passed for epoch state transitions with blocks instead of seconds. The reason for this is that there may be situations where block production is not steady, and some slots are missed. This would create an unfair advantage to the first submitter for the epoch solution. This mitigation was implemented in PR 759 [22].

Implementing an on-chain LP solver for epochs. As discussed with Centrifuge team, the LP problem in Centrifuge's case is not highly complex and finding a solution to LP problems of this size can be done on-chain via an LP solver. This is important for the security of the chain since currently the off-chain approach allows attackers to intervene with the solution submission, for example, by submitting a biased/bad solution and spamming the chain for a short period of time and preventing better solution being submitted to the chain. The current challenge according to the team is lack of a mature open-source LP-solver implementation in Rust and also size limitations in the case of using a web-assembly version (compiled from a C implementation) that is compiled and integrated into the runtime. We recommend implementing the on-chain epoch solver as soon as a mature Rust library for solving LP problems is available.

The Centrifuge team decided to keep the solution calculation off-chain to enable better scalability. As the challenge period time is now a runtime configurable parameter and set in blocks, the risk of spamming attacks by solution submission is reduced.

Fairness measures for *PoolAdmins* adjusting the interest rates. As per the current state of the business logic of the Centrifuge substrate implementation, there are no limiting factors for a *PoolAdmin* to change interest rates on command. This could cause some losses to pool investors; in case they are not aware of the changes and are not able to redeem/invest in advance. The Centrifuge team mitigated this issue by requiring a minimum delay to pass and preventing interest rate updates from being applied if there are any outstanding redemption orders from investors. This mitigation was implemented in PR 759 [22].

6.2 General suggestions

Up-to-date documentation. We recommend keeping an up-to-date documentation about the project and the processes on how the users should interact with it. Considering the financial nature of the Centrifuge core service, it is of utmost importance that a clear path of execution should be documented. Such a documentation should become a source of truth for the present and future and allows developers or auditors to compare the factual execution of the chain with the outlined documentation. Implementing a continuous documentation is often seen as cumbersome when it isn't entirely included within the development process.

We recommend including documentation steps within coding processes or while submitting pull requests which forces developers to keep a documented code. Experience shows that adding documentation step by step is a much more sustainable and viable model for high-paced environments than writing large chunks of documentation once every now and then.

Moreover, keeping a good documentation on the implemented process can shed lights on potential corner cases that are business relevant. When in doubt for a specific process or code section, there should always be one source of truth.

Structured and factorized code. After documentation, code is the source of truth. Keeping a styled and clean code allows new and more veteran developers alike understanding the intrinsic knowledge of the product. Some of the Centrifuge's pallets (e.g., loans) are well structured and factorized; we recommend doing this for all the other pallets implemented in the system as well as the future ones.

Implement integration testing. The current state of the project already implements unit tests which allows checking if a function is working properly. Integration tests enable developers to see if different units of the code are working according to the expectations. Considering the complexity of some of Centrifuge's components such as tranches and epochs, we recommend investing time in developing integration tests that could test throughout processes.

For instance, considering how users will behave while using the platform, how and why a user would close an epoch, investing in loans or redeeming tokens. Keeping the user in mind is at the core of recommended best practices.

Leverage internal threat modeling to spread awareness between developers. Since developers have a deep understanding of the codebase, identifying and prioritizing potential threats as part of the development process would enhance the security of the Centrifuge repositories.

Regular code review and continuous fuzz testing. We recommend regularly reviewing the codebase (for example, internally) to avoid introducing new logic or arithmetic bugs. We also recommend utilizing fuzz testing to identify potential vulnerabilities early in the development process. Ideally, Centrifuge would continuously fuzz their code on each commit made to the codebase. Centrifuge can draw some inspiration from the Substrate codebase, which includes multiple fuzzing harnesses based on *honggfuzz*.

Regular updates. New releases of Substrate may contain fixes of critical security issues. Since Centrifuge is a product that heavily relies on Substrate, we recommend updating to the latest version as soon as possible, when there is a new available release.

7 Bibliography

[1] [Online]. Available: https://centrifuge.hackmd.io/s8eNOLupS72hxIYQDg24Gw?view.

[2] [Online]. Available: https://docs.centrifuge.io/.

[3] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/.

[4] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/tree/parachain/SRL-audit.

[5] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/2.

[6] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/595 .

[7] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/3.

[8] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/609.

[9] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/6.

[10] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/7.

[11] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/14.

[12] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/685.

[13] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/686.

[14] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/879.

[15] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/4.

[16] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/597.

[17] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/9.

[18] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/744.

[19] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/11.

[20] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/719.

[21] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/15.

[22] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/759.

[23] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/5.

[24] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/621.

[25] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/13.

[26] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/12.

[27] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/8.

[28] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/issues/616.

[29] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/11.

[30] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/pull/622.

[31] [Online]. Available: https://github.com/centrifuge/centrifuge-chain/issues/616.

[32] [Online]. Available: https://github.com/centrifuge/centrifuge-chain-internal/issues/12.