Security Review Report
NM-0097 zkLend

NETHERMIND SECURITY

(Sep 29, 2023)

Contents

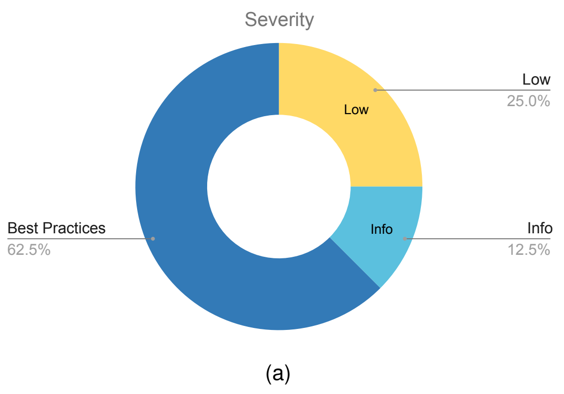
1 Executive Summary

This document presents the security review performed by Nethermind on the zkLend protocol. ZkLend is a Starknet native lending protocol where users can deposit specified assets in exchange for ZTokens, representing a user's portion of liquidity owned in a pool. Deposited assets are also treated as collateral, allowing for overcollateralized borrowing of assets in the same or other liquidity pools. Borrowing from a pool accrues interest, and when funds are paid after the borrow, this increases the value of each liquidity provider's position in the pool. The liquidity pools can also be used for flashloans, which cost some predetermined fee amount. If an account becomes undercollateralized during a borrow, its position can be eligible for liquidation to ensure the protocol remains solvent.

Before this engagement, Nethermind reviewed the zkLend protocol in September 2022. The only significant change between the previous review and this report was reimplementing the protocol in Cairo1 compared to Cairo0. No changes in the overall protocol logic have been made since the previous fix review. Each line had been manually rewritten from Cairo0 to Cairo1 to ensure that the new implementation behaves exactly the same as the original.

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** 8 points of attention, where 2 are classified as `Low`, 1 are classified as `Info` and 5 are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. 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.



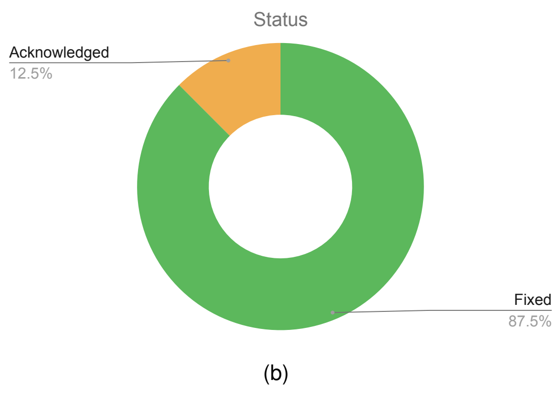


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (1), **Best Practices** (5).
Distribution of status: Fixed (7), **Acknowledged** (1), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug. 28, 2023
Response from Client	Sep. 20, 2023
Final Report	Sep. 29, 2023
Methods	Manual Review, Automated Analysis
Repository	https://github.com/zkLend/zklend-v1-core/
Commit Hash (Audit)	482d14fd6f767fe3f7bf67db35adae1474327d7b
Documentation	README.md
Documentation Assessment	High
Test Suite Assessment	High

2 Contracts

	Contract	Lines of Code	Lines of Comments	Comments Ratio	Blank Lines	Total Lines
1	src/lib.cairo	7	0	0.0%	6	13
2	src/oracles.cairo	1	0	0.0%	0	1
3	src/irms.cairo	1	0	0.0%	0	1
4	src/z_token.cairo	175	2	1.1%	37	214
5	src/market.cairo	273	17	6.2%	48	338
6	src/default_price_oracle.cairo	74	6	8.1%	16	96
7	src/interfaces.cairo	186	50	26.9%	80	316
8	src/libraries.cairo	6	0	0.0%	5	11
9	src/oracles/empiric_oracle_adapter.cairo	57	2	3.5%	12	71
10	src/libraries/math.cairo	66	5	7.6%	10	81
11	src/libraries/ownable.cairo	50	17	34.0%	14	81
12	src/libraries/safe_decimal_math.cairo	58	7	12.1%	14	79
13	src/libraries/safe_math.cairo	128	1	0.8%	25	154
14	src/libraries/reentrancy_guard.cairo	17	12	70.6%	7	36
15	src/libraries/pow.cairo	714	2	0.3%	8	724
16	src/market/traits.cairo	34	3	8.8%	10	47
17	src/market/view.cairo	117	18	15.4%	34	169
18	src/market/internal.cairo	722	116	16.1%	174	1012
19	src/market/errors.cairo	22	0	0.0%	0	22
20	src/market/storage.cairo	219	8	3.7%	80	307
21	src/market/external.cairo	215	13	6.0%	42	270
22	src/z_token/traits.cairo	25	3	12.0%	8	36
23	src/z_token/view.cairo	59	3	5.1%	24	86
24	src/z_token/internal.cairo	87	6	6.9%	20	113
25	src/z_token/errors.cairo	8	0	0.0%	0	8
26	src/z_token/external.cairo	213	18	8.5%	60	291
27	src/irms/default_interest_rate_model.cairo	71	4	5.6%	15	90
	Total	**3605**	**313**	**8.7%**	**749**	**4667**

3 Summary of Findings

Finding	Severity	Update
Oracle adapter does not validate price timestamp	Low	Unresolved
Oracle adapter should handle unexpected cases from oracle	Low	Unresolved
Oracle privilege risks	Info	Unresolved
Function code can be shorter	Best Practices	Unresolved
Unnecessary return of tuple	Best Practices	Unresolved
Missing input validation in interest rate model constructor	Best Practices	Unresolved
Typo corrections	Best Practices	Unresolved
Update oracle naming to refer to "Pragma" instead of "Empiric Network"	Best Practices	Unresolved

4 Protocol Overview

The zkLend protocol stores all liquidity on one contract, where each asset has its own "reserve", each having its own oracle, ZToken, and interest rate model. Up to 125 reserves can exist on the protocol at any given time. All user interactions happen through the market protocol, such as deposits, withdrawals, borrows, repayments, flashloans, and liquidations. The contract structure and a description of each zkLend contract are shown below.

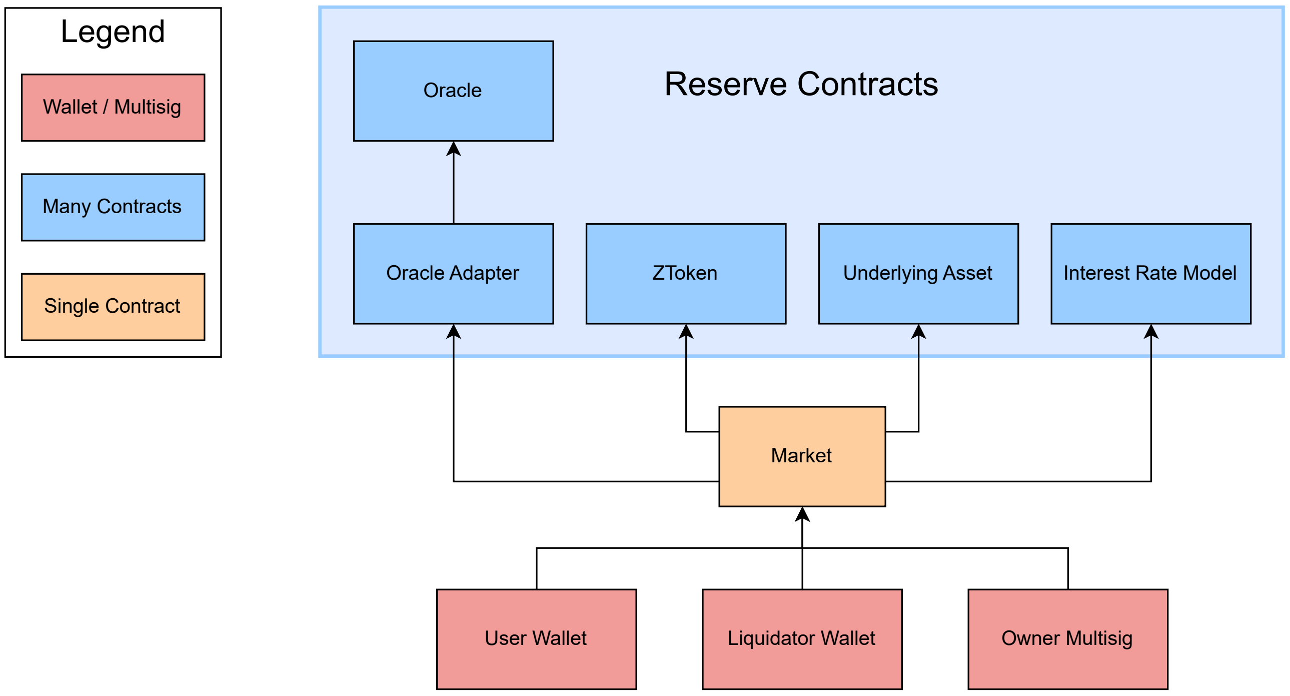


Fig. 2: Structural Diagram of Contracts

Market: Contains all market logic and handles accumulation of ZToken value compared to the underlying asset. All liquidity is stored on this contract, along with all reserve data. The market contract makes calls to other related contracts for each reserve, such as oracles, ZTokens, underlying assets, and the interest rate models. All user interactions and liquidations occur through this contract.

ZToken: Represents underlying assets that have been deposited to the protocol. When depositing, ZTokens are minted to the caller's address. These represent collateral that can be used to borrow funds from the same or other reserves. ZTokens operate in a similar way to a typical ERC20 token; however, when transferring tokens, collateral checks are done to ensure that after the transfer, the sender would not be in a position that leaves them undercollateralized on their existing loans.

Underlying Asset: The underlying value token which is deposited or borrowed. Examples of underlying assets can be USDC or Ether.

Interest Rate Model: This is called by the market contract to determine the borrowing and lending rate. The input for this calculation is the total liquidity compared to the borrowed liquidity, which can be used to find a utilization rate. This rate can then be used to find the borrowing and lending rates.

Oracle Adapter: Exists between the protocol and the price data source as a way to allow for multiple different price feeds. At the time of this report, the only oracle adapter that exists is designed to support Pragma.

Oracle: The source of price data that is used for pricing calculations. Oracle adapters exist between the oracle to allow multiple unique oracles to be supported. At the time of this report, the only supported oracle is Pragma.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 an attacker will uncover and exploit the finding. 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 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 Findings

6.1 [Low] Oracle adapter does not validate price timestamp

File(s): src/oracles/empiric_oracle_adapter.cairo

Description: When requesting price data from a Pragma oracle through the function get_spot_median(...), along with the price, a timestamp is returned, which indicates when the price has last been updated. The oracle adapter does not validate this data in get_-data(...). The function is shown below:

```
fn get_data(self: @ContractState) -> PriceWithUpdateTime {
    let oracle_addr = self.oracle.read();
    let pair_key = self.pair.read();

    let median = IEmpiricOracleDispatcher {
        contract_address: oracle_addr
    }.get_spot_median(pair_key);

    let scaled_price = scale_price(median.price, median.decimals);
    PriceWithUpdateTime { price: scaled_price, update_time: median.last_updated_timestamp }
}
```

It should be noted that calling the oracle adapter function get_price_with_time(...) does return the price with the last update timestamp, so a contract calling this function could validate the timestamp itself. However, if timestamp validation is moved outside of the oracle adapter, this will pose a risk to any contracts calling the standard get_price(...) function, since the price data may be late and there is no way to verify this.

Recommendation(s): Consider adding a validation for the price update timestamp in the oracle adapter to ensure that the price has been updated within a reasonable amount of time.

Status: Fixed

Update from the client: Fixed in PR #64.

6.2 [Low] Oracle adapter should handle unexpected cases from oracle

File(s): src/oracles/empiric_oracle_adapter.cairo

Description: The oracle adapter does not handle unexpected cases when retrieving price data from an oracle. The function used to call the oracle for price data is shown below:

```
fn get_data(self: @ContractState) -> PriceWithUpdateTime {
    let oracle_addr = self.oracle.read();
    let pair_key = self.pair.read();

    let median = IEmpiricOracleDispatcher {
        contract_address: oracle_addr
    }.get_spot_median(pair_key);

    let scaled_price = scale_price(median.price, median.decimals);
    PriceWithUpdateTime { price: scaled_price, update_time: median.last_updated_timestamp }
}
```

When using Pragma oracles for price data, we highlight two unexpected outcomes when calling get_spot_median(...), which should be considered:

1) A Pragma oracle will return a price of zero if the price update has not occurred in the past 2 hours and 10 minutes.

2) Pragma oracles are upgradeable; an incorrect implementation upgrade or an implementation upgrade that deprecates the currently used price function may cause calls to the oracle to revert.

Recommendation(s): Consider using a fallback price oracle (such as an on-chain AMM) if one of these unexpected cases occur. Alternatively, consider reverting (in the case of a revert on call to get_spot_median(...) the revert can be left to bubble up).

Status: Fixed

Update from the client: Fixed in PR #64.

6.3 [Info] Oracle privilege risks

File(s): `src/oracles/empiric_oracle_adapter.cairo`

Description: ZkLend uses the oracle provider Pragma to access price data on supported assets. The Pragma protocol has permissioned features accessible to privileged addresses, which should be noted. We present the specific features which may pose a risk to the ZkLend protocol:

- Add a price data source address
- Update a price data source address
- Remove a price data source address
- Update a currency
- Upgrade oracle implementation
- Allow an oracle to accept price data from a source
- Prevent an oracle from accepting price data from a source

Recommendation(s): This finding highlights permissioned actions which may pose a risk to the ZkLend protocol in the unlikely case that the Pragma privileged addresses behave incorrectly. No action is required. However, these risks should be acknowledged.

Status: Acknowledged

Update from the client: Acknowledged

6.4 [Best Practices] Function code can be shorter

File(s): `z_token/view.cairo`

Description: The functions `totalSupply(...)` and `balanceOf(...)` both call their respective `felt` returning functions and save that return to a variable. Then, that variable is changed to a `u256` type on a separate line and returned. The Cairo 1 compiler can interpret the return type, and `into(...)` can be used instead to simplify these functions. The function `totalSupply(...)` is shown below as an example:

```
fn totalSupply(self: @ContractState) -> u256 {
    /////////////////////////////////////////////////////////////////////////////////////
    // @audit-issue: Can call function, convert to u256 and return in single line instead
    /////////////////////////////////////////////////////////////////////////////////////

    let scaled_up_supply = felt_total_supply(self);
    let scaled_up_supply: u256 = scaled_up_supply.into();

    scaled_up_supply
}
```

Recommendation(s): Consider changing the structure of `totalSupply(...)` and `balanceOf(...)` to the following:

```
fn totalSupply(self: @ContractState) -> u256 {
    felt_total_supply(self).into()
}
```

Status: Fixed

Update from the client: Agreed. The original verbose code was simply the by-product of applying the strict line-by-line translation from Cairo 0 to minimize human errors as much as possible. Now is indeed a good timing to simplify them. Fixed in PR #62.

6.5 [Best Practices] Unnecessary return of tuple

File(s): `libraries/ownable.cairo`

Description: The function `renounce_ownership(...)` does not return a value, therefore there is no need to return an empty tuple at the end of the function.

```
fn renounce_ownership<T, impl TOwnable: Ownable<T>, impl TDrop: Drop<T>>(ref self: T) {
    ...
    return (); //@audit-issue : Unnecessary return empty tuple.
}
```

Recommendation(s): Consider removing the `return();` line in `renounce_ownership(...)`.

Status: Fixed

Update from the client: Fixed in PR #62.

6.6 [Best Practices] Missing input validation in interest rate model constructor

File(s): src/irms/default_interest_rate_model.cairo

Description: The default interest rate model contract is missing an input validation for the argument optimal_rate, which is used to indicate which slope should be used for calculating the borrow rate. This has already been acknowledged with a TODO comment by the team, however, this should still be addressed. The code is shown below:

```
#[constructor]
fn constructor(
    ref self: ContractState,
    slope_0: felt252,
    slope_1: felt252,
    y_intercept: felt252,
    optimal_rate: felt252
) {
    // TODO: check `optimal_rate` range
    self.curve_params.write(CurveParams { slope_0, slope_1, y_intercept, optimal_rate });
}
```

It should not be possible for the optimal rate to be higher than 100%. A check to ensure that the value is not higher than safe_decimal_-math::SCALE can be done to verify this.

Recommendation(s): Consider adding input validation for optimal_rate and removing the TODO comment.

Status: Fixed

Update from the client: Fixed in PR #60.

6.7 [Best Practices] Typo corrections

File(s): src/*

Description: The following is a list of spelling corrections in the source code:

```
In function market::internal::settle_extra_reserve_balance
    "impilcit_total_balance" -> "implicit_total_balance"

In function market::view::get_pending_treasury_amount
    "Tresury" -> "Treasury"
```

Recommendation(s): Correct the typos highlighted above.

Status: Fixed

Update from the client: Fixed in PR #63.

6.8 [Best Practices] Update oracle naming to refer to "Pragma" instead of "Empiric Network"

File(s): src/*

Description: The oracle provider used by the ZkLend protocol has changed its name from "Empiric Network" to "Pragma". However, all oracle-related traits, structs, modules, and interfaces refer to the old oracle naming. Ensuring that all references to external dependencies are accurate helps to improve code readability and maintainability.

Recommendation(s): Consider updating all references to the chosen oracle provider from "Empiric Network" to "Pragma".

Status: Fixed

Update from the client: Fixed in PR #61.

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 plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. 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 that describes the design and technical details of the smart contract. 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.

The provided `README` contains a helpful summary of the protocol design and upgradeability, as well as instructions on how to set up the environment, compile, and run tests. The protocol design summary shows a detailed graph presenting the entire protocol, including both the smart contract layer as well as the browser, wallet, and other infrastructure.

The zkLend contracts feature high-quality comments. Each function has a comment describing the expected functionality and any specified edge cases or prerequisites. There are detailed explanations for areas of the code that require extra clarity, which helps developers to understand the codebase quickly.

8 Test Suite Evaluation

8.1 Contracts Compilation

8.1.1 Cairo

```
>  ./scripts/compile.sh
Compiling zklend::market::Market
Compiling zklend::z_token::ZToken
Compiling zklend::default_price_oracle::DefaultPriceOracle
Compiling zklend::irms::default_interest_rate_model::DefaultInterestRateModel
Compiling zklend::oracles::empiric_oracle_adapter::EmpiricOracleAdapter
```

8.2 Tests Output

8.2.1 Cairo

```
> cairo-test --starknet .
running 69 tests
test zklend::libraries::safe_math::tests::test_add_2 ... ok
test zklend::libraries::safe_math::tests::test_mul_uint256_overflow ... ok
test zklend::libraries::pow::tests::test_ten_pow_overflow ... ok
test zklend::libraries::safe_math::tests::test_add_overflow_1 ... ok
test zklend::libraries::safe_math::tests::test_add_1 ... ok
test zklend::libraries::pow::tests::test_two_pow ... ok
test zklend::libraries::pow::tests::test_ten_pow ... ok
test zklend::libraries::safe_decimal_math::tests::test_mul_decimals_overflow ... ok
test zklend::libraries::pow::tests::test_two_pow_overflow ... ok
test zklend::libraries::math::tests::test_shl ... ok
test zklend::libraries::safe_math::tests::test_div_1 ... ok
test zklend::libraries::safe_math::tests::test_add_overflow_2 ... ok
test zklend::libraries::safe_decimal_math::tests::test_div ... ok
test zklend::libraries::safe_math::tests::test_sub_underflow_1 ... ok
test zklend::libraries::safe_math::tests::test_div_2 ... ok
test zklend::libraries::safe_math::tests::test_sub_1 ... ok
test zklend::libraries::safe_math::tests::test_sub_underflow_2 ... ok
test zklend::libraries::safe_decimal_math::tests::test_div_decimals ... ok
test zklend::libraries::math::tests::test_shr ... ok
test zklend::libraries::safe_math::tests::test_div_3 ... ok
test zklend::libraries::safe_math::tests::test_sub_2 ... ok
test zklend::libraries::safe_math::tests::test_mul_1 ... ok
test zklend::libraries::safe_math::tests::test_mul_2 ... ok
test zklend::libraries::safe_math::tests::test_div_division_by_zero ... ok
test zklend::libraries::safe_decimal_math::tests::test_mul ... ok
test zklend::libraries::safe_math::tests::test_mul_felt_overflow ... ok
test zklend::libraries::safe_decimal_math::tests::test_mul_decimals ... ok
test zklend::libraries::safe_decimal_math::tests::test_mul_overflow ... ok
test tests::default_interest_rate_model::test_borrow_rates ... ok
test tests::z_token::test_transfer_should_emit_events ... ok
test tests::z_token::test_balance_should_scale_with_accumulator ... ok
test tests::z_token::test_transfer_all_should_emit_events ... ok
test tests::z_token::test_meta ... ok
test tests::z_token::test_approve_should_change_allowance ... ok
test tests::z_token::test_transfer_from ... ok
test tests::market::test_borrow_cannot_exceed_debt_limit ... ok
test tests::z_token::test_transfer_all ... ok
test tests::market::test_token_burnt_on_withdrawal ... ok
test tests::market::test_rate_changes_on_withdrawal ... ok
test tests::market::test_cannot_liquidate_too_much ... ok
test tests::market::test_debt_repayment ... ok
test tests::z_token::test_burn_all ... ok
test tests::market::test_debt_limit_is_global ... ok
test tests::market::test_cannot_liquidate_healthy_positions ... ok
test tests::market::test_debt_accumulation ... ok
test tests::market::test_liquidation ... ok
test tests::market::test_new_reserve_event ... ok
test tests::market::test_event_emission ... ok
test tests::market::test_repay_all_with_interest ... ok
test tests::market::test_can_borrow_till_debt_limit ... ok
test tests::market::test_no_debt_accumulation_without_loan ... ok
test tests::market::test_disabling_already_disabled_collateral ... ok
test tests::market::test_has_debt_flag_changed_on_borrow ... ok
test tests::market::test_interest_accumulation ... ok
test tests::market::test_rates_changed_on_borrow ... ok
test tests::market::test_deposit_transfer_failed ... ok
test tests::market::test_cannot_borrow_more_than_capacity ... ok
test tests::market::test_cannot_withdraw_with_zero_amount ... ok
test tests::market::test_rate_changes_on_deposit ... ok
test tests::market::test_flashloan_fails_without_enough_fees ... ok
test tests::market::test_token_transferred_on_deposit ... ok
test tests::market::test_token_received_on_borrow ... ok
test tests::market::test_flashloan_succeeds_with_enough_fees ... ok
test tests::market::test_cannot_withdraw_collateral_used_by_loan ... ok
test tests::market::test_cannot_transfer_collateral_used_by_loan ... ok
test tests::market::test_flashloan_fee_distribution ... ok
test tests::market::test_collateral_used_by_existing_loan ... ok
test tests::audit::test_leveraged_borrow ... ok
test tests::market::test_can_borrow_again_with_more_collateral ... ok
test result: ok. 69 passed; 0 failed; 0 ignored; 0 filtered out;
```

8.3 Tests Summary

The test suite for smart contracts are well implemented and cover edge cases, exploring function calls with different inputs, both valid and invalid. Integration tests between smart contracts are well implemented with native cairo tests without including any external dependencies.

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