Zellic

Prepared for
Edi Sinovcic
SpaceShard

Prepared by
Jisub Kim
Daniel Lu
Zellic

November 26, 2024

Nimbora

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for SpaceShard from November 11th to November 18th, 2024. During this engagement, Zellic reviewed Nimbora's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a malicious admin or pauser disrupt the system or negatively impact users by misusing their permissions?
- Could improper handling or malicious manipulation of bucket states result in locked or lost funds?
- Could an on-chain attacker manipulate the `report` function to mismanage liquidity or drain funds from the system?
- Could `RedeemRequest` be exploited to withdraw more assets than entitled or to manipulate the withdrawal process?
- Are there vulnerabilities in the interaction between the liquid staking contract and bucket contracts that could be exploited to disrupt the system or drain funds?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Packages code (including Starknet-staking)
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

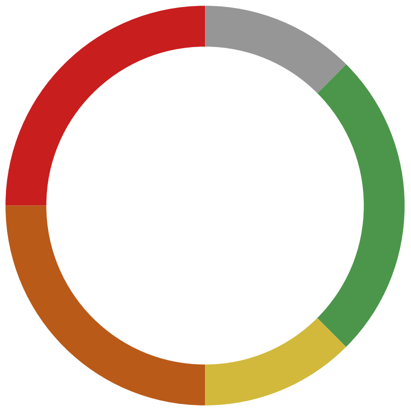
1.4. Results

During our assessment on the scoped Nimbora contracts, we discovered eight findings. Two critical issues were found. Two were of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of SpaceShard in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	2
🟧 High	2
🟨 Medium	1
🟩 Low	2
⬜ Informational	1



2. Introduction

2.1. About Nimbora

SpaceShard contributed the following description of Nimbora:

> Nimbora introduces a liquid staking solution that allows users to stake their STRK and earn rewards, while maintaining the liquidity and fungibility of their staked tokens. The platform provides a seamless staking experience, enabling users to participate in network validation and governance while having access to their staked assets.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Nimbora Contracts

Type	Cairo
Platform	Starknet

Target	Nimbora-lst
Repository	https://github.com/0xSpaceShard/nimbora-lst ↗
Version	`2e1c4a12ba163715f50e4362183727db203774ea`

Programs	`src/liquidstaking/*.cairo`
	`src/redeem_request/*.cairo`
	`src/bucket/*.cairo`
	`src/reserve/*.cairo`
	`src/rewards_accumulator/*.cairo`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.8 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jisub Kim
Engineer
jisub@zellic.io ↗

Daniel Lu
Engineer
daniel@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 11, 2024	Kick-off call
November 11, 2024	Start of primary review period
November 18, 2024	End of primary review period

3. Detailed Findings

3.1. Redemption requests do not consume allowance

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The liquid-staking contract has a `request_redeem` function allowing users to remove staked tokens and rewards. This function consumes the user's liquid-staking token and mints them an NFT representing the right to withdraw assets once they are available.

The function takes in an arbitrary owner and recipient address. It consumes its allowance of the owner's token to the protocol and mints an NFT to the recipient.

```
fn request_redeem(
    ref self: ContractState,
    shares: u256,
    recipient: ContractAddress,
    owner: ContractAddress
) -> u256 {
    self.pausable.assert_not_paused();
    let assets = self._convert_to_assets(shares, Rounding::Down(()));
    assert(assets.is_non_zero(), Errors::ZERO_ASSETS);
    self._burn_lst(owner, shares);
```

This means that an attacker can initiate redemptions on behalf of other users for themselves, without authorization.

Impact

An attacker can steal any staked funds from any user who has prepared to redeem funds.

Recommendations

We recommend retrieving the tokens from the *caller's* allowance to the protocol, instead of taking the owner as an argument.

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit `a1857e4c` ↗.

3.2. Redemption NFTs can be double-spent

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Once a redemption is requested, the recipient is minted an NFT. This NFT can be permissionlessly used to redeem the underlying assets. However, the `claim_redeem` function does not actually burn the NFT, allowing it to be reused.

```
fn claim_redeem(ref self: ContractState, id: u256) -> u256 {
    self.pausable.assert_not_paused();
    let redeem_request = self.redeem_request.read();
    let erc721_disp = ERC721ABIDispatcher { contract_address:
    redeem_request };
    let owner_of_id = erc721_disp.owner_of(id);
    let redeem_request_disp = IRedeemRequestDispatcher { contract_address:
    redeem_request };
    let redeem_info = redeem_request_disp.id_to_info(id);
    let epoch = redeem_info.epoch;
    assert(
        self._get_withdrawal_pool_state(epoch)
    == WithdrawalPoolStatus::Claimable(()),
        Errors::NOT_CLAIMABLE
    );
    let withdrawal_pool = self.withdrawal_pool.read(epoch);
    let withdrawal_share = self.withdrawal_share.read(epoch);
    let pool_shares = redeem_info.pool_shares;
    let assets = self
        ._calc_convert_to_assets(
            withdrawal_share, withdrawal_pool, pool_shares, Rounding::Down(())
        );
    let erc20_disp = ERC20ABIDispatcher { contract_address: self.asset.read()
    };
    erc20_disp.transfer(owner_of_id, assets);
    self
        .emit(
            RedeemRequestClaimed {
                recipient: owner_of_id, pool_shares: pool_shares, assets:
    assets, id: id,
            }
        );
    assets
}
```

Impact

An attacker can repeatedly redeem more assets using the same NFT and steal all protocol funds.

Recommendations

We recommend consuming the NFT when it is used.

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit `a1857e4c` ↗.

3.3. Stray buckets in past withdrawals can block epoch progression

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

User withdrawals are grouped together by the epochs in which they are requested. Each time `report` runs, the protocol will check if there are enough buckets to initiate withdrawals for past epochs. If so, it will assign them buckets and begin the withdrawal process.

Nimbora performs some optimizations for minimizing bucket usage. First, during this assignment process, it will attempt to reuse buckets across multiple epochs.

Second, it will attempt to cancel buckets involved in ongoing withdrawals in order to fulfill new deposit requests. However, the implementation of this second optimization does not account for the first — namely the fact that buckets may be reused. When a bucket's unpool state is canceled, it is removed from the epoch it is found in but not from every epoch it has been assigned to.

Impact

This means that an epoch's withdrawals may contain buckets that no longer align with its state. This can cause unexpected behavior and delays in the protocol.

Recommendations

We recommend either assigning buckets to only one epoch at a time or improving the bucket-removal process to account for reuse.

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit e944a478 ↗.

3.4. Uncaught errors in `claim_rewards` can halt protocol

Target	bucket/bucket.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Note: SpaceShard brought this finding to our attention during the engagement and issued a proactive fix. We document it here for completeness.

Description

In the `report` function, the contract calls `_claim_rewards_and_pay_fees`, which iterates over all buckets and calls `claim_rewards` on each without checking if they are still pool members:

```
fn _claim_rewards_and_pay_fees(
    ref self: ContractState, asset: ERC20ABIDispatcher
) -> (u256, u256) {
    let mut i = 0;
    let bucket_len = self.bucket_len.read();
    loop {
        if i == bucket_len {
            break;
        }
        IBucketDispatcher { contract_address: self.bucket.read(i)
    }.claim_rewards();
        i += 1;
    };
    // [...]
```

If `claim_rewards` fails (e.g., because a bucket is no longer a pool member), it causes the entire `report` function to fail, preventing the protocol from advancing epochs and processing essential operations.

Impact

A single failing `claim_rewards` call can halt the `report` function, stopping epoch progression and disrupting the protocol.

Recommendations

We recommend checking pool members before calling `claim_rewards`:

```
fn claim_rewards(ref self: ContractState) {
    self._assert_liquid_staking();
    IPoolDispatcher { contract_address: self.delegation_pool.read() }
        .claim_rewards(get_contract_address());
    let pool = IPoolDispatcher { contract_address: self.delegation_pool.read(
        ) };
    if (self._is_pool_member(pool)) {
        pool.claim_rewards(get_contract_address());
    }
}
```

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit 363dc8b6 ↗.

3.5. Incorrect role permission in `unpause`

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The current implementation allows the pauser role to both pause and unpause the contract. However, the intended behavior is for only the pauser role to be able to pause the contract, while unpausing should be restricted to the admin role.

```
fn pause(ref self: ContractState) {
    self.accesscontrol.assert_only_role(ROLE::PAUSE_ROLE);

fn unpause(ref self: ContractState) {
    self.accesscontrol.assert_only_role(ROLE::PAUSE_ROLE);
```

Impact

This incorrect permission assignment allows a pauser to resume the contract, which is not in line with the desired access-control policy.

Recommendations

Consider changing the `unpause` function to require the admin role instead of the pauser role:

```
fn unpause(ref self: ContractState) {
    self.accesscontrol.assert_only_role(ROLE::PAUSE_ROLE);
    self.accesscontrol.assert_only_role(ROLE::ADMIN_ROLE);
}
```

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit `0fa79abc` ↗.

3.6. Missing bucket-removal functionality

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The protocol utilizes a collection of bucket contracts that have funds staked with Starknet. While the protocol admin has the ability to add more buckets (say, if the protocol volume increases), there is no functionality to remove them.

Impact

This means that adding buckets is irreversible. But the presence of extraneous buckets can lead to issues with the protocol; many actions involve iterating over all buckets. If the admin mistakenly adds too many, this can cause funds to become stuck in the protocol.

We note further that protocol documentation indicates that this functionality should exist, which may cause confusion for users and maintainers.

Recommendations

We recommend adding an upper limit to the total number of buckets or introducing a mechanism to remove buckets. Additionally, we recommend that protocol documentation be updated to reflect any changes.

Remediation

SpaceShard acknowledges that this functionality is missing and has potential risks, commenting that the total number of buckets will be capped to 25.

3.7. Inefficient recursive implementation in _pow function

Target	liquid_staking/liquid_staking.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The _pow function is currently implemented recursively with a time complexity of $\mathcal{O}(n)$, which is inefficient for large exponents. When compiled, the function executes actual recursive calls rather than being optimized into an iterative loop.

Impact

The slow time complexity of this implementation may impact performance, particularly for operations involving large numbers. This could result in higher execution costs.

Recommendations

Consider updating the _pow function to use exponentiation by squaring ↗, which achieves $\mathcal{O}(\log n)$ complexity iteratively.

```
fn pow2(self: @ContractState, x: u256, n: u256) -> u256 {
    if n == 0 {
        return 1;
    }
    if n == 1 {
        return x;
    }
    let tmp = self.pow2(x * x, n / 2);
    if (n % 2) == 1 {
        return x * tmp;
    }
    return tmp;
}
```

The improvement can be verified via the following command.

```
scarb test test_pow --save-trace-data
```

Comparing the execution traces, we see the following results:

- `pow()`: `"vm_resources": "n_steps":23626, "n_memory_holes": 1907`
- `pow2()`: `"vm_resources": "n_steps":2999, "n_memory_holes": 115`

This demonstrates a substantial reduction in execution steps and memory usage.

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit `6bf47e1b` ↗.

3.8. Incorrect `unpause()` test

Target	test_permission.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The test cases for `test_pause` and `test_unpause` are the same, and both call `pause()`.

Impact

Incorrect or redundant test coverage reduces the effectiveness of testing, which may allow potential issues with the `unpause()` functionality to go unnoticed.

Recommendations

Consider updating `test_unpause` to correctly test the `unpause()` functionality.

Remediation

This issue has been acknowledged by SpaceShard, and a fix was implemented in commit `0fa79abc ↗`.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Admin responsibilities and centralization risk

We note that the protocol requires an amount of admin upkeep. Namely, an essential parameter is the number of buckets used. Whether the protocol has enough buckets to keep up with its volume can significantly impact its efficiency.

Additionally, the protocol gives the admin (as well as a separate role) the ability to pause not only deposits but also withdrawals. This can lead to user funds being locked in the protocol. More importantly, many components of the protocol are operator upgradable, which can allow funds to be stolen if the owner is compromised or malicious. In general, we recommend clarifying to users 1) the conditions under which these actions might be taken and 2) how admin and pauser actions are governed, as these can impact user trust in the protocol.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

Like other liquid-staking protocols, Nimbora stakes deposits on behalf of users, issuing a more liquid asset in return. Specifically, when a user deposits STRK, the protocol mints an ERC-20 token representing the user's share of total deposits. At any time, a user can initiate a withdrawal of their STRK and receive an NFT representing their redemption request. Once the funds are unlocked, this NFT can be burned to redeem associated assets and rewards.

5.1. Bucketing

The protocol owns a collection of bucket contracts that each deposit a portion of funds into the Starknet delegation pools. These are used because Starknet staking does not let users have concurrent withdrawals; any additional undelegation will reset the withdrawal timelock.

The bucket contracts simply track the identities of Nimbora's main liquid-staking contract and the chosen Starknet delegation pool. The liquid-staking contract has permission to

- control how it adds to and exits from the delegation pool,
- update what pool the bucket is using, and
- trigger a withdrawal into the reward accumulator.

5.2. Reward accumulator

The reward accumulator is a contract that holds rewarded tokens. It receives these because each bucket sets the contract as the recipient while entering the pool. The liquid-staking contract has permission to harvest these rewards into its own balance.

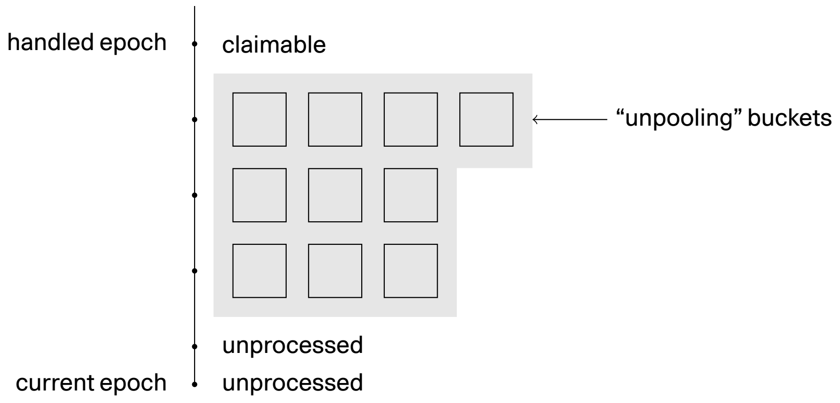
5.3. Reserve

The reserve is a contract responsible for holding liquid-staking tokens that have not been distributed during deposit.

5.4. Liquid staking

The core liquid-staking contract manages the bucket state, given user deposit and withdrawal intentions. Staking and unstaking actions are driven by the `report` function and can be called once per epoch (daily).

Logically, the contract controls two kinds of buckets: some will have an ongoing "undelegate" intent, while others will not. These undelegating buckets are associated with withdrawal requests, which are aggregated by epoch. The protocol refers to these as "unpooling" buckets and the others as "regular" buckets.



The above diagram implies that an unpooling bucket can be associated with only one withdrawal epoch at a time. While this is now, at the time of writing, the intended behavior, the protocol originally allowed buckets to participate in simultaneous withdrawals. See Finding 3.3. ↗ for one reason this behavior was changed.

Notice that an epoch can be in a few states. It may be "claimable", which means that the funds in its withdrawals have been unlocked. It may be "pending", which means that it has been assigned buckets that are awaiting withdrawal. Or it might be "unprocessed", which means that it has not yet been assigned buckets.

If the contract receives a deposit request, it immediately collects the funds and mints the corresponding liquid-staking tokens. The deposit amount is added to a deposit buffer, which represents the balance yet to be staked.

Similarly, if the contract receives a request to redeem, it immediately burns the user's liquid-staking tokens and issues an NFT representing the redemption shares. These assets and shares are then tracked in the current epoch's withdrawal bucket. The NFT can be burned for the underlying asset once the corresponding epoch is claimable.

Then, the contract's `report` function can be called once per epoch. It works as follows.

1. First, we iterate over the withdrawal epochs that are not yet claimable. If any of these have

reached the unlock time, we iterate over the underlying buckets and withdraw the unlocked funds. We then mark the unlocked epochs as claimable.

2. Then, we iterate over all buckets to harvest rewards and pay protocol fees.
3. Next, recall that the current epoch's deposits have yet to be staked. We first attempt to stake these funds by updating existing withdrawals. Specifically, we iterate over epochs that cannot be claimed.
   a. If an epoch is pending, we iterate over the assigned buckets. If the pending withdrawal on a bucket is less than the deposit buffer, we cancel the withdrawal and remove that bucket from the epoch. Effectively, we stake some of the deposit buffer by freeing up the bucket.
   b. If an epoch is unprocessed, we check if its total withdrawal amount is less than the deposit buffer. If so, we mark it as pending while assigning no buckets to it. This means that we use the deposit buffer to fulfill the entire epoch's withdrawal, but we still subject its users to the unstaking period.
4. Now, there still remains some amount of the deposit buffer that has not been staked. We distribute these among the regular buckets if possible. If there are no regular buckets remaining (i.e., if every bucket is used in a pending withdrawal), we deposit them among the unpooling buckets.
5. Finally, we begin the next epoch.

SpaceShard includes unit tests for the liquid-staking contract that cover deposit, withdrawal, and epoch progression. For ensuring the correctness of bucket management with respect to Starknet staking, we recommend thoroughly testing the *end-to-end* mechanics of the protocol. This is particularly important because the bucket management has

- complex functionality and large state space on its own, as well as
- close dependencies on the logic of the Starknet staking contracts.

Improved end-to-end coverage will particularly help ensure that the protocol behaves as expected as Starknet staking evolves.

Shares

In order to distribute rewards fairly, the protocol interprets a user's balance in liquid-staking tokens as their share of the deposited funds. If a user performs a deposit, they receive shares; if they perform a withdrawal, their shares are burned in exchange for the NFT.

When computing the value of shares for withdrawal, the contract needs 1) the total existing shares and 2) the total deposited funds. The total existing shares is maintained entirely internally by the contract, simply updated when shares are drawn from or returned to the reserve contract.

The contract recovers the total deposited funds from three sources:

1. The funds deposited that have yet to be staked
2. The total funds that are staked

3. The funds that are still staked but have been claimed by burning liquid-staking tokens

The first source is maintained completely internally by the contract; it is simply the deposit buffer. The second source is derived directly from the underlying Starknet staking: during each epoch, the contract iterates over all buckets and computes the total staked funds. The third source is also maintained by the contract, by taking the sum over withdrawal epochs that have not been matched with buckets.

A key point is that the contract should not issue rewards for funds when they are in the deposit buffer. This is accomplished by collecting rewards only after processing the withdrawals.

Withdrawal shares

To recap, the shares discussed previously are used to track what users are owed when redemption is initiated. But the protocol also tracks assets owed to users *during the withdrawal period*. Namely, the NFT issued is associated with the following data.

```
#[derive(Debug, PartialEq, Drop, Serde, Copy, starknet::Store)]
pub struct RedeemRequestInfo {
    pub epoch: u256,
    pub pool_shares: u256,
}
```

Currently, a share of a given withdrawal epoch corresponds exactly to the concrete amount of funds that the user will be able to withdraw. SpaceShard commented that the bookkeeping is structured to facilitate future functionality, like handling cases where stake is slashed.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to Starknet Sepolia.

During our assessment on the scoped Nimbora contracts, we discovered eight findings. Two critical issues were found. Two were of high impact, one was of medium impact, two were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.