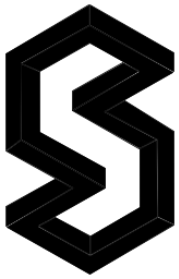


NEAR Liquid Token

Audit

Presented by:

OtterSec contact@osec.io

Adrian Self adnan@osec.io

William Wang defund@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Stader Labs engaged OtterSec to perform an assessment of the `near-x` contract. This assessment was conducted between August 29th and September 9th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation. We delivered final confirmation of the patches September 9th, 2022.

After our initial audit, we also performed additional incremental reviews.

Key Findings

Over the course of this audit engagement, we produced 5 findings total.

In particular, we accounting issues related to asynchronous execution of callbacks (OS-NLT-ADV-00).

We also made a number of recommendations around tighter function access control (OS-NLT-SUG-00), stricter standard conformance (OS-NLT-SUG-02), minor rounding issues (OS-NLT-SUG-03), and more.

Overall, we commend the Stader team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/stader-labs/near-liquid-token. This audit was performed against commit edbbdcb. We also reviewed PR #71

There was a total of 1 program included in this audit. A brief description of the program is as follows. A full list of program files and hashes can be found in Appendix A.

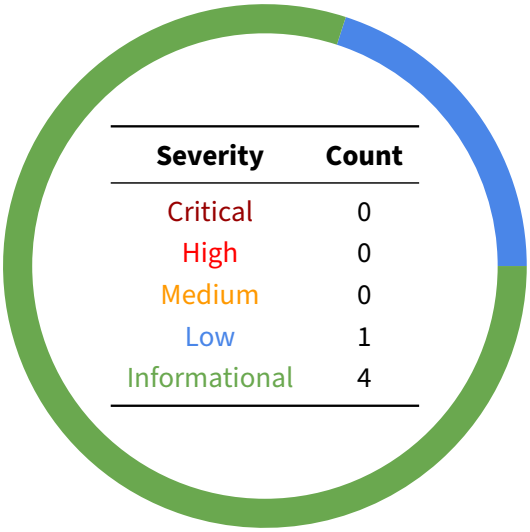
Name	Description
near-x	Liquid staking protocol where staked NEAR is represented by NearX tokens.

03 | Findings

Overall, we report 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4

04 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix D.

ID	Severity	Status	Description
OS-NLT-ADV-00	Low	Resolved	Improper handling of asynchronous behavior during direct deposit

OS-NLT-ADV-00 [low] [resolved] | Improper Direct Deposit Async Handling

Description

When invoking `internal_direct_deposit_and_stake`, a copy of the validator info is passed into the asynchronous callback. Unfortunately due to Near's asynchronous nature, this data can become out of date, leading to a potential race condition.

If the deposit function is invoked multiple times, this might cause inaccurate accounting due to use of a cached `ValidatorInfo`.

A similar issue affects on`_get_sp_staked_balance_for_rewards`.

RUST

```
if is_promise_success() {
    validator_info.staked += amount;
    acc.stake_shares += num_shares;
    self.total_stake_shares += num_shares;
    self.total_staked += amount;
    log!(
        "Successfully staked {} into {}",
        amount,
        validator_info.account_id
    );
    self.internal_update_validator(&validator_info.account_id,
    ↪  &validator_info);
```

Remediation

Pass in the validator account id and reload the validator info in the callback handler.

RUST

```
    pub fn on_stake_pool_direct_deposit_and_stake(
        &mut self,
        validator_account_id: AccountId,
        // ...
    ) -> PromiseOrValue<bool> {
```

Patch

Resolved in ac317d0 and 3912ee1.

05 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-NLT-SUG-00	Functions are marked as private when they should be internal.
OS-NLT-SUG-01	NEP-330 Contract Source Metadata is implemented, but contract source is not publicly available at the given URL.
OS-NLT-SUG-02	The codebase implements a helper function which is already provided by NEAR's Rust SDK.
OS-NLT-SUG-03	The contract rounds against the protocol when unstaking NEAR.

OS-NLT-SUG-00 | Internal Functions Marked Private

Description

Functions in NEAR can be public, private, or internal. A private function is exposed on the blockchain, but attempts to call the private function will fail unless the caller is the contract itself; either through a callback or using the contract's account directly. Internal functions, on the other hand, cannot be directly called at all; they only perform functionality internal to the contract.

The following functions were observed to be private:

- `get_validator_to_stake`
- `get_validator_to_unstake`
- `get_unstake_release_epoch`
- `epoch_reconcilation`

However, these functions are not used for callbacks, and there is need to call them directly using the contract's account keys.

Remediation

Change these functions from private to internal.

OS-NLT-SUG-01 | Contract Source Metadata

Description

The purpose of ContractSourceMetadata, as per NEP-330, is to allow "auditing and viewing source code for a deployed smart contract." This NEP is generally suitable only for open-source contracts, as a way for users to easily locate the source code for the deployed contract.

The contract implements `ContractSourceMetadataTrait` in `contracts/near-x/src/contract/metadata.rs`. When this data is queried using the NEAR CLI, as shown below, the contract provides the URL https://github.com/stader-labs/near-liquid-token and the cargo package version. However, this is not a public repository, and accessing this URL results in an error 404.

```sh
$ NEAR_ENV=mainnet near view nearx.stader-labs.near
    ↪  contract_source_metadata
View call: nearx.stader-labs.near.contract_source_metadata()
{
  version: '0.1.0',
  link: 'https://github.com/stader-labs/near-liquid-token'
}
```

Remediation

Do not implement `ContractSourceMetadata` unless the contract is open-source and the repository is available to the community. If this contract becomes open-source in the future, consider implementing NEP-330 again.

OS-NLT-SUG-02 | Redundant Helper Function

Description

In `operator.rs`, several callback functions use `is_promise_success`, a helper function which is implemented in the codebase. However, this functionality is already provided by NEAR's Rust SDK as `near_sdk::is_promise_success`. Note that this is already used elsewhere in the codebase.

src/utils.rs RUST

```rust
pub fn is_promise_success() -> bool {
    require!(
        env::promise_results_count() == 1,
        ERROR_EXPECT_RESULT_ON_CALLBACK
    );

    matches!(env::promise_result(0), PromiseResult::Successful(_))
}
```

The current behavior of the custom implementation seems to be consistent with the behavior of `near_sdk::is_promise_success`. However, it is recommended to use the NEAR Rust SDK's implementation to benefit from any updates made to the functionality or efficiency of the function in future.

Remediation

Remove the custom implementation of `is_promise_success`, and use `near_sdk::is_promise_success` everywhere instead.

OS-NLT-SUG-03 | Rounding Direction in Stake/Unstake

Description

When a user stakes NEAR, the receive a proportional amount of shares in return. This calculation is rounded against the user, in that they may receive less than what is fair.

src/contract/internal.rs RUST

```rust
// Calculate the number of "stake" shares that the account will receive for staking the
// given amount.
let num_shares = self.num_shares_from_staked_amount_rounded_down(amount);
require!(num_shares > 0, ERROR_NON_POSITIVE_STAKE_SHARES);
```

When a user unstakes shares, they receive NEAR in return. However, in this case the calculation is rounded against the protocol, in that the user may receive more than what is fair. Interestingly, the program later rounds against the user while calculating their remaining amount.

src/contract/internal.rs RUST

```rust
let mut receive_amount = self.staked_amount_from_num_shares_rounded_up(num_shares);
require!(
    receive_amount > 0,
    ERROR_NON_POSITIVE_UNSTAKE_RECEVIE_AMOUNT
);
```

It is worth noting that rounding errors are fairly inconsequential on NEAR, as the discrepancy (1 yoctoNEAR) is negligible. Additionally, NEAR's staking pool core contract rounds against the protocol in both stake and unstake.

Remediation

The sound approach would be to always round against the user, in favor of the protocol. In this case, use the `staked_amount_from_num_shares_rounded_down` method to calculate `receive_amount`.

If the goal is to be consistent with NEAR's core contracts, consider rounding against the protocol in all cases. Additionally, document this behavior and ensure there is a "cushion fund" to handle rounding losses.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
Cargo.toml                    64a152bc7831e6070d8a8f30693538a89802efbe839fe3f806d794b1de663ad1
src
  constants.rs                50c53929f3083c2d4fe1cda4a0feb701db5c65398fd4654d13c2214e49cd9328
  contract.rs                 452c9b9433eb283e78f609c1f011d7a76c15cbc0b484e01d1cdbfbc1933a2fb8
  errors.rs                   8af8eb04547576aada86df080e23b85a9322d93b5656bad80e97e8df8266b25f
  events.rs                   56ef0627a79d9bfac632fb0ce56ae286a2cf2427df8f2c37388d6acec9de6116
  fungible_token.rs           e76e68cd007a050ffb6ef7cec3441405264775f2a0e04ebcf6e79ca565661e7f
  lib.rs                      4d2921421b084175652bf948a24fb138c43132c1b3f63953138d11ab5dad83ee
  state.rs                    2e4bc931008b81137468575efb569d94b39092ba457da94436c1d77e09edd4cd
  utils.rs                    c97a01d029759486062e693aae0b11e91c5387d8c522258dc7bcd96d525dfb67
  contract
    internal.rs               5f30549619aab6b1567c5fef5e6d426d2f8cf13ac3783afc20e8baaa430ce3c0
    metadata.rs               0dd01b05bdacadaafab6f4c4c1dc3abf67cd420eb0b4f1b8e2dd05ad46c636c2
    operator.rs               cc9c1a72acb54389bdadc9a35315f54cc9b393775790deaab018ead3a809e381
    public.rs                 d5c829b1da4fdef92d77062894efc9a1a2ea509ac16d4d92862d5a241bf756e1
    storage_spec.rs           4bf8c2ebfacda16979f4cb36beb21e7b23e6a457d8d8195e178dc24a604682d8
    upgrade.rs                9ae925313f842f1a407b88dc41887f365b353e8db0d7508f019fb770624c3d84
    util.rs                   b7337f4a7e3fe32329909cac3a45667741664197fbe63c593a3f97cc36939b89
  fungible_token
    metadata.rs               051800e4ad991961b85e2f741062516456dd61d02deea1befab1ed92e3c886f1
    nearx_internal.rs         441dbf9e2799d493a0bddafa6e7663aa019a21f44f1a59269c8719afae2e83e4
    nearx_token.rs            7bce195f5c1997b207622dfd13c2fb0f09ed41296c2ab74f5091474bbbba9399
tests
  unit_tests.rs               c4728ead3021c5337ea24a12076a553a2650e465ea3b4e592a531c492662d2fd
  helpers
    mod.rs                    a447ff7753c5658e79d3af169f08adc003df9f10f81bdf6d233ccb25e82bde8b
```

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an onchain program. In other words, there is no way to steal tokens or deny service. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of NEAR's execution model. For a non-exhaustive list of security issues we check for, see Appendix C.

Implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

C | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

D | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities which immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority/token account validation • Rounding errors on token transfers
High	Vulnerabilities which could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities which could lead to denial of service scenarios or degraded usability. Examples: • Malicious input cause computation limit exhaustion • Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation • Uncaught Rust errors (vector out of bounds indexing)