Security Audit Report for Phoenix Bonds

Date: Jan 12, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.01.12
20:01:32 +08'00'

Contents

Report Manifest

Item	Description
Client	LiNEAR
Target	Phoenix Bonds

Version History

Version	Date	Description
1.0	January 12, 2023	First Version

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The repository that has been audited includes Phoenix Bonds [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Phoenix Bonds	`Version 1`	`0e4308ea83ee73ef85e818b78bee4e4816857c0f`
	`Version 2`	`7a316469ef2e8d9b77aed47aab2d636e061b3653`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **phoenix-bonds/contracts/phoenix-bonds/src** folder contract only. Specifically, the files covered in this audit include:

- fungible_token/core.rs
- fungible_token/meta.rs
- fungible_token/mod.rs
- interfaces/ft.rs
- interfaces/linear.rs
- interfaces/mod.rs
- accrual.rs
- active_vector.rs
- bond_note.rs
- events.rs
- legacy.rs
- lib.rs
- lost_found.rs
- math.rs
- owner.rs
- types.rs
- upgrade.rs
- utils.rs
- view.rs

[1]https://github.com/linear-protocol/phoenix-bonds

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

∗ Reentrancy
∗ DoS
∗ Access control
∗ Data handling and data flow
∗ Exception handling
∗ Untrusted external call and control flow
∗ Initialization consistency
∗ Events operation
∗ Error-prone randomness
∗ Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

		Likelihood	
Impact	*High*	High	Medium
	Low	Medium	Low
		High	*Low*

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we do not find any potential issues. We have **five** recommendations and **three** notes as follows:

- High Risk: 0
- Medium Risk: 0
- Low Risk: 0
- Recommendations: 5
- Notes: 3

ID	Severity	Description	Category	Status
1	-	Two-Step Transfer of Privileged Account Ownership	Recommendation	Confirmed
2	-	Potential Centralization Problem	Recommendation	Confirmed
3	-	Lack of Check of Price for LiNEAR	Recommendation	Confirmed
4	-	Lack of Validation for Owner when Upgrading	Recommendation	Fixed
5	-	Code Optimization	Recommendation	Fixed
6	-	Assumption on the Secure Implementation of Dependencies	Note	Confirmed
7	-	Delayed Price of LiNEAR	Note	Confirmed
8	-	No Restriction of Redeem Time	Note	Confirmed

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Two-Step Transfer of Privileged Account Ownership

Status Confirmed

Introduced by Version 1

Description The contract uses change_owner() to configure the privileged account, which can conduct many sensitive operations (e.g., withdraw LiNEAR tokens from the treasury). In this case, when an incorrect new owner is provided, the contract is at risk of attack and the privileged function can be invoked.

```
#[payable]
pub fn change_owner(&mut self, new_owner_id: AccountId) {
    self.assert_owner();
    self.owner_id = new_owner_id;
}
```

Listing 2.1: owner.rs

Suggestion I Implement a two-step approach for the owner update (e.g., change_owner() and commit_owner()).

Feedback from the Project The owner will be set to DAO once the contract is initialized and is not likely to change after that. If we do want to change the owner, after configuring owner to DAO, every time when we do the change, the proposal will be reviewed by multiple people to ensure it's correct. The effect is kind of similar to the two-step approach proposed here, so a two-step approach is not very necessary for this.

2.1.2 Potential Centralization Problem

Status Confirmed

Introduced by Version 1

Description This project has potential centralization problems. The owner has the privilege to configure several system parameters (e.g., PhoenixBonds.tau), pause/unpause the contract, and withdraw LiNEAR tokens from the treasury.

Suggestion I Introducing a decentralization design in the contract is recommended, such as a multi-signature or a public DAO. In addition, it's suggested to delete the full access keys of the contract so that the sensitive operations (e.g., NEAR transfer) can only be conducted within the logic of the smart contract.

Feedback from the Project Yeah. We'll update here after the owner DAO is configured and contract is locked for the mainnet version.

2.1.3 Lack of Check of Price for LiNEAR

Status Confirmed

Introduced by Version 1

Description Function get_linear_price() will invoke a cross-contract call to LiNEAR protocol for the LiNEAR price. The price should always be larger than one according to the implementation of the LiNEAR protocol, which is suggested to be checked.

```
fn get_linear_price(&self) -> Promise {
        linear_contract::ext(self.linear_address.clone())
            .with_static_gas(GAS_GET_LINEAR_PRICE)
            .ft_price()
    }
```

Listing 2.2: lib.rs

Suggestion I Add a check to make sure the price of LiNEAR is larger than 1.

Feedback from the Project As long as the LiNEAR contract address is correct, and LiNEAR contract works properly, the LiNEAR price should be >= 1 and no need to check.

2.1.4 Lack of Validation for Owner when Upgrading

Status Fixed in Version 2

Introduced by Version 1

Description In the process of upgrading, function get_summary() is invoked to validate the state of the contract. However, the contract owner is not checked. In this case, the owner may not be able to execute the upgrade again if the previous upgrade/migration went badly.

```
#[no_mangle]
 pub fn upgrade() {
     env::setup_panic_hook();
     let contract: PhoenixBonds = env::state_read().expect("ERR_CONTRACT_IS_NOT_INITIALIZED");
     contract.assert_owner();
     let current_id = env::current_account_id().as_bytes().to_vec();
```

```rust
        let migrate_method_name = b"migrate".to_vec();
        let get_summary_method_name = b"get_summary".to_vec();
        let get_summary_args = b"{\"linear_price\":\"1000000000000000000000000\"}".to_vec();
        unsafe {
            // Load input (wasm code) into register 0.
            sys::input(0);
            // Create batch action promise for the current contract ID
            let promise_id =
                sys::promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
            // 1st batch action in the Tx: "deploy contract" (code is taken from register 0)
            sys::promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
            // 2nd batch action in the Tx: call migrate() in the contract with sufficient gas
            let required_gas =
                env::used_gas() + GAS_FOR_COMPLETING_UPGRADE_CALL + GAS_FOR_GET_SUMMARY_CALL;
            require!(
                env::prepaid_gas() >= required_gas + MIN_GAS_FOR_MIGRATE_CALL,
                "Not enough gas to complete contract state migration"
            );
            let migrate_attached_gas = env::prepaid_gas() - required_gas;
            sys::promise_batch_action_function_call(
                promise_id,
                migrate_method_name.len() as _,
                migrate_method_name.as_ptr() as _,
                0_u64,
                0_u64,
                0_u64,
                migrate_attached_gas.0,
            );
            // 3rd batch action in the Tx: call get_summary() in the contract to validate
            // the contract state. If the validation failed, the entire upgrade() method
            // will be rolled back. The get_summary() view call will access most of the
            // states in the contract, so should guarantee the contract is working as expected
            sys::promise_batch_action_function_call(
                promise_id,
                get_summary_method_name.len() as _,
                get_summary_method_name.as_ptr() as _,
                get_summary_args.len() as _,
                get_summary_args.as_ptr() as _,
                0_u64,
                GAS_FOR_GET_SUMMARY_CALL.0,
            );
            sys::promise_return(promise_id);
        }
    }
```

Listing 2.3: upgrade.rs

```rust
pub fn get_summary(&self, linear_price: U128) -> Summary {
    Summary {
        linear_balance: self.linear_balance.into(),
        reserve_pool_near_amount: self.reserve_pool_near_amount(linear_price.0).into(),
        pending_pool_near_amount: self.pending_pool_near_amount.into(),
        permanent_pool_near_amount: self.permanent_pool_near_amount.into(),
            treasury_pool_near_amount: self.treasury_pool_near_amount.into(),
            bootstrap_ends_at: self.bootstrap_ends_at,
            tau: self.tau,
            alpha: self.accrual_param.current_alpha(current_timestamp_ms()),
        }
    }
```

Listing 2.4: view.rs

Suggestion I Return the owner of the contract in function `get_summary()`.

2.1.5 Code Optimization

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the callback function `on_get_linear_price_for_redeem()`, the amount of `LiNEAR` tokens that the user could redeem is calculated based on the amount of `pNEAR` tokens. The contract first calculates the amount of `NEAR` with the given `pNEAR` tokens (i.e., `equivalent_near_amount`), then converts them to `LiNEAR` according to the price of `LiNEAR`. Since `equivalent_near_amount` is an intermediate value, the calculation mentioned above can be optimized in one step.

```
 #[private]
 pub fn on_get_linear_price_for_redeem(
     &mut self,
     user_id: AccountId,
     pnear_amount: U128,
     #[callback_result] linear_price: Result<U128, PromiseError>,
 ) -> Promise {
     let linear_price = linear_price.expect(ERR_GET_LINEAR_PRICE);
     require!(
         self.ft.internal_unwrap_balance_of(&user_id) >= pnear_amount.0,
         ERR_NOT_ENOUGH_PNEAR_BALANCE
     );
     require!(
         self.pnear_total_supply() - pnear_amount.0 > ONE_PNEAR,
         ERR_BURN_TOO_MANY
     );

     // equivalent amount of NEAR that given pNEAR worth
     let equivalent_near_amount = pnear2near(pnear_amount.0, self.pnear_price(linear_price.0));
     let redeemed_linear = near2linear(equivalent_near_amount, linear_price.0);

     self.linear_balance -= redeemed_linear;
     self.burn_pnear(&user_id, pnear_amount.0, Some("Redeem pNEAR"));

     Event::Redeem {
         account_id: user_id.clone(),
         pnear_amount,
         redeemed_linear: redeemed_linear.into(),
     }
     .emit();
```

```
        self.transfer_linear(&user_id, redeemed_linear, "pNEAR Redeem")
    }
```

Listing 2.5: lib.rs

Suggestion I Calculate the `redeemed_linear` in one step.

2.2 Notes

2.2.1 Assumption on the Secure Implementation of Dependencies

Status Confirmed

Introduced by `version 1`

Description The `Phoenix Bonds` contract is built based on the crates `near-sdk` (version 4.1.1), `near-contract-standards` (version 4.1.1), and `near_bigdecimal` (version 0.1.1).

The required interfaces and the basic functionality listed below are provided in the contract:

- `NEP-141` (Fungible Token Standard)
- `NEP-145` (Storage Management Standard)
- `NEP-148` (Fungible Token Metadata Standard)
- `NEP-297` (Events Standard)

In this audit, we assume the standard library provided by `NEAR-SDK-RS` [1] (i.e., `near_contract_standards`) and `near-bigdecimal` [2] has no security issues.

Feedback from the Project Yes. We assume the `near-sdk-rs` standard library is secure as built by `NEAR` team, and the source code of the `near-bigdecimal` is used by multiple projects such as `Burrow`, `Ref` and `Cornerstone`, and is already audited by `BlockSec` in previous auditing.

2.2.2 Delayed Price of LiNEAR

Status Confirmed

Introduced by `version 1`

Description Given the asynchronous nature of the `NEAR` protocol, one transaction on the `NEAR` protocol may be executed in several blocks. The price of `LiNEAR` tokens may not be the latest, which may make the user earn more profits than expected.

Feedback from the Project This is very hard to totally avoid due to the nature of `NEAR`, however this could only happen when `LiNEAR` is updating its staking rewards, which typically takes just several minutes in each epoch.

2.2.3 No Restriction of Redeem Time

Status Confirmed

Introduced by `version 1`

[1]https://github.com/near/near-sdk-rs

[2]https://github.com/linear-protocol/near-bigdecimal

Description The user can choose to burn their `pNEAR` tokens to redeem `LiNEAR` tokens without any restriction. This may result in a potential risk of loss if the users redeem their claimed `pNEAR` immediately. Besides, the contract will ensure the total supply of `pNEAR` tokens will be no less than 10^{24} yocto during the redemption.

Feedback I from the Project The `redeem()` function is not available on Phoenix Bonds UI, so should not directly impact end users. For rational users who'd like to maximize their return, they should take into account both the market price and the redemption price of pNEAR.

Feedback II from the Project This is to prevent `pNEAR` total supply being 0 which makes it impossible to calculate `pNEAR` price. In practice our team will lock some `NEAR` in the contract and never redeem them.