Security Audit Report for BurrowLand

Date: Jan 30, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2024.01.31
15:01:13 +08'00'

Contents

Report Manifest

Item	Description
Client	Ref Labs
Target	BurrowLand

Version History

Version	Date	Description
1.0	January 30, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

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 burrowland [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
BurrowLand	Version 1	a1b9f4581dc4b80d0c85998455f38aa193b6afc1
	Version 2	3622051b676b65d637142f51339e0c2405d06b7d

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

- account.rs
- account_asset.rs
- account_farm.rs
- account_view.rs
- actions.rs
- asset.rs
- asset_config.rs
- asset_farm.rs
- asset_view.rs
- big_decimal.rs
- booster_staking.rs
- config.rs
- events.rs
- fungible_token.rs
- legacy.rs
- lib.rs
- pool.rs
- price_receiver.rs
- prices.rs
- storage.rs

[1] https://github.com/burrowHQ/burrowland/tree/pyth

- storage_tracker.rs
- upgrade.rs
- utils.rs
- shadow_actions.rs
- position.rs
- pyth.rs

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.

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

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

Table 1.1: Vulnerability Severity Classification

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

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

Chapter 2 Findings

In total, we find **two** potential issues. Besides, we have **three** recommendations and **one** note as follows:

- High Risk: 0
- Medium Risk: 2
- Low Risk: 0
- Recommendations: 3
- Notes: 1

ID	Severity	Description	Category	Status
1	Medium	Potential Unfairness Due to Inconsistent Prices from Different Oracles	Defi Security	Fixed
2	Medium	Lack of Check in remove_token_pyth_info()	DeFi Security	Fixed
3	-	Unnecessary Handling of fraction_digits	Recommendation	Confirmed
4	-	Lack of Check in add_token_pyth_info()	Recommendation	Fixed
5	-	Lack of Gas Check in internal_execute_with_pyth()	Recommendation	Fixed
6	-	Potential Risks Introduced by default_price	Note	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential Unfairness Due to Inconsistent Prices from Different Oracles

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description In the current implementation, users have the option to perform position adjustments using the price fed by the existing oracle through the function oracle_on_call() if enable_price_oracle is set True, as well as using the newly added Pyth oracle. However, the prices provided by these two oracles may differ, which allows users to adjust their positions based on price feeds that are more favorable to them. It creates an unfair advantage over others.

```
fn oracle_on_call(&mut self, sender_id: AccountId, data: PriceData, msg: String) {
    assert_eq!(env::predecessor_account_id(), self.get_oracle_account_id());

    assert!(self.get_config().enable_price_oracle, "Price oracle disabled");

    let actions = match serde_json::from_str(&msg).expect("Can't parse PriceReceiverMsg") {
        PriceReceiverMsg::Execute { actions } => actions,
    };

    let mut account = self.internal_unwrap_account(&sender_id);
    self.validate_price_data(&data);
    self.internal_execute(&sender_id, &mut account, actions, data.into());
    self.internal_set_account(&sender_id, account);
}
```

Listing 2.1: price_receiver.rs

```
#[payable]
pub fn execute_with_pyth(&mut self, actions: Vec<Action>) {
    assert_one_yocto();
    let account_id = env::predecessor_account_id();
    let mut account = self.internal_unwrap_account(&account_id);
    self.internal_execute_with_pyth(&account_id, &mut account, actions);
    self.internal_set_account(&account_id, account);
}
```

Listing 2.2: price_receiver.rs

Impact Inconsistent prices can create unfairness between users.

Suggestion Ensure only one oracle is available.

2.1.2 Lack of Check in remove_token_pyth_info()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `remove_token_pyth_info()` is used to remove the existing `Pyth` information associated with a specific `token_id`. However, it does not check whether the token being removed is still used in users' positions. If the token is still held in users' positions, users may be unable to adjust their positions.

```
#[payable]
pub fn remove_token_pyth_info(&mut self, token_id: TokenId) {
    assert_one_yocto();
    self.assert_owner_or_guardians();
    let is_success = self.token_pyth_info.remove(&token_id).is_some();
    assert!(is_success, "Invalid token id");
}
```

Listing 2.3: lib.rs

Impact Users' positions cannot be adjusted.

Suggestion Ensure the token to be removed will not be held in any users' positions.

2.2 Additional Recommendation

2.2.1 Unnecessary Handling of fraction_digits

Status Confirmed

Introduced by `Version 1`

Description The function pyth_price_to_price_oracle_price() calculates the price based on the token_info and the pyth_price received from the Pyth oracle. However, the handling of fraction_digits in token_info is redundant, as it does not affect the process of price calculations.

```
 pub fn pyth_price_to_price_oracle_price(token_info: &TokenPythInfo, pyth_price: &PythPrice)
     -> Price {
    require!(pyth_price.price.0 > 0, "Invalid Pyth Price");
    let mut multiplier = BigDecimal::from(pyth_price.price.0 as Balance);
    if pyth_price.expo > 0 {
        multiplier = multiplier * BigDecimal::from(10u128.pow(pyth_price.expo.abs() as u32));
    } else {
        multiplier = multiplier / BigDecimal::from(10u128.pow(pyth_price.expo.abs() as u32));
    }

    Price {
        multiplier: (multiplier * BigDecimal::from(10u128.pow(token_info.fraction_digits as u32))).
            round_down_u128(),
        decimals: token_info.decimals + token_info.fraction_digits
    }
}
```

Listing 2.4: pyth.rs

Suggestion I Remove the related logic of fraction_digits.

2.2.2 Lack of Check in add_token_pyth_info()

Status Fixed in Version 2

Introduced by Version 1

Description Function add_token_pyth_info() allows the admin to register a specific token_id for token_pyth_info. In addition, the admin can also update the token_pyth_info for already registered token_id. It is suggested to check that the token_id is not registered previously.

Meanwhile, modifying the configuration of tokens within token_pyth_info is required in rare cases. It is suggested to implement a separate function specifically for handling updates.

```
    #[payable]
    pub fn add_token_pyth_info(&mut self, token_id: TokenId, token_pyth_info: TokenPythInfo) {
        assert_one_yocto();
        self.assert_owner_or_guardians();
        self.token_pyth_info.insert(token_id, token_pyth_info);
    }
```

Listing 2.5: lib.rs

Suggestion I Add a check to ensure the token_id is registered before. Besides, if necessary, implement a new function for updating the token_pyth_info of registered token_id.

2.2.3 Lack of Gas Check in internal_execute_with_pyth()

Status Fixed in Version 2

Introduced by Version 1

Description Function `internal_execute_with_pyth()` executes several price queries as promises and, in the callback function `callback_execute_with_pyth()`, performs the next steps based on the results of the promises and input actions. However, the function does not check the gas required for promises, and insufficient gas can lead to waste of gas.

```rust
pub fn internal_execute_with_pyth(&mut self, account_id: &AccountId, account: &mut Account,
     actions: Vec<Action>) {
    let pyth_oracle_account_id = self.internal_config().pyth_oracle_account_id;
    let involved_tokens = self.involved_tokens(&account, &actions);
    if involved_tokens.len() > 0 {
        let (mut all_promise_flags, mut promise) = token_involved_promises(
                &pyth_oracle_account_id, &self.get_pyth_info_by_token(&involved_tokens[0]), &
                     involved_tokens[0]);
        for token in involved_tokens[1..].iter() {
            let (token_promise_flags, token_promise) = token_involved_promises(
                &pyth_oracle_account_id, &self.get_pyth_info_by_token(token), token);
            all_promise_flags.extend(token_promise_flags);
            promise = promise.and(token_promise);
        }
        promise.then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_CALLBACK_EXECUTE_WITH_PYTH)
                .callback_execute_with_pyth(account_id.clone(), all_promise_flags, actions)
        );
    } else {
        self.internal_execute(account_id, account, actions, Prices::new());
    }
}
```

Listing 2.6: pyth.rs

Suggestion I Add a check to ensure the gas is enough for all promises.

2.3 Notes

2.3.1 Potential Centralization Risks Introduced by default_prices

Status Confirmed

Introduced by version 2

Description The project adds a `default_price` configuration to the token's `token_pyth_info`, which can be manually updated through the privileged function `update_token_pyth_info()`. The purpose of this implementation is to ensure the operations (e.g., liquidate) that involved tokens whose price is not available in pyth can be conducted. However, it is important to note that this implementation introduces centralization problem. We recommend that the team carefully select tokens eligible for the market and appropriately set the `default_price` to mitigate the potential risks.