Reflector

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Andreas Mantzoutas andreas@osec.io

Contents

01 | Executive Summary

Overview

StellarExpert engaged OtterSec to assess the Reflector Oracle Protocol program. This assessment was conducted between January 18th and January 25th, 2024. For more information on our auditing methodology, refer to Appendix C.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a vulnerability concerning the unsafe truncation of a u32 asset index to a u8, resulting in inaccurate prices for assets with an index greater than or equal to 256 (OS-REF-ADV-00), and an additional issue regarding the absence of a sanity check on the timestamp, enabling the potential setting of prices with future timestamps (OS-REF-ADV-01). Furthermore, we highlighted the lack of checks for admin authorization on calling the configuration functionality (OS-REF-ADV-02) and the incorrect management of missing intermediate prices in time-weighted average price calculation (OS-REF-ADV-03).

We also made recommendations around the inadequate handling of division by zero during floor division (OS-REF-SUG-00) and suggested certain optimizations involving the elimination of the need for maintaining a vector of assets (OS-REF-SUG-01).

02 | Scope

The source code was delivered to us in a Git repository at github.com/reflector-network/reflector-contract. This audit was performed against commit 14b1bd3.

A brief description of the programs is as follows:

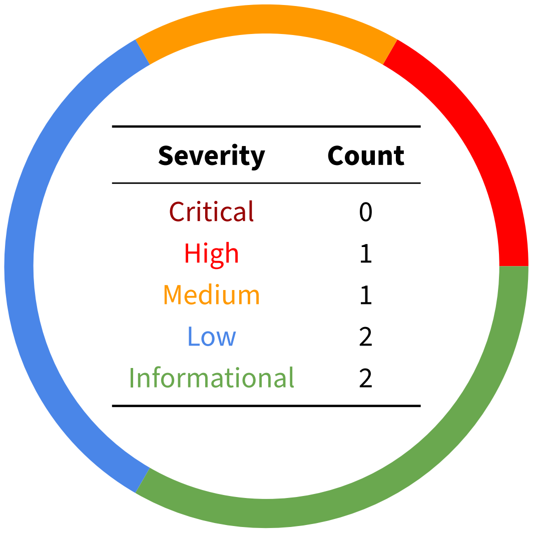
Name	Description
Reflector Oracle Protocol	An oracle protocol that combines specialized smart contracts and a peer-to-peer consensus among data provider nodes, maintained by trusted organizations within the Stellar ecosystem, serving as intermediaries connecting Stellar smart contracts with external price feed data sources. The protocol facilitates the integration of on-chain and off-chain asset prices, centralized and decentralized exchange rates, stock indices, financial market APIs, and more. Reflector nodes meticulously process, normalize, aggregate, and store trade information from the Stellar Classic decentralized exchange, Soroban decentralized exchange protocols, and external sources.

As part of this audit, we also provided a proof of concept to prove exploitability and enable simple regression testing.

03 | Findings

Overall, we reported 6 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 do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	0
High	1
Medium	1
Low	2
Informational	2

Proofs of Concept

We created a proof of concept for easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure may be found in Appendix A.

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 B.

ID	Severity	Status	Description
OS-REF-ADV-00	High	Resolved	The asset index is stored as a `u32` and unsafely truncated to a `u8` when retrieved. This results in reporting incorrect prices when attempting to access assets with an index greater than `u8::MAX`.
OS-REF-ADV-01	Medium	Resolved	`set_price` currently lacks a sanity check for the input timestamp, which may introduce invalid timestamps into the protocol.
OS-REF-ADV-02	Low	Resolved	`config` does not check whether `config.admin` authorized the call. This may result in a loss of control over the deployed contract.
OS-REF-ADV-03	Low	Resolved	`twap` inaccurately calculates averages when intermediate prices are absent.

OS-REF-ADV-00 [high] | Unsafe Asset Index Truncation

Description

The current implementation retrieves the asset index as a `u32` and truncates it to a `u8` without proper bounds checking. This unsafe truncation results in incorrect asset prices with an index greater than or equal to `256`.

Proof of Concept

An example scenario demonstrates incorrect price reporting due to unsafe truncation in Appendix A.

Remediation

Implement proper bounds checking when retrieving and handling asset indices.

Patch

Fixed in 0207362.

OS-REF-ADV-01 [med]| Lack of Timestamp Sanitization

Description

The absence of a sanity check in `set_price` exposes the protocol to potential issues by permitting the insertion of invalid timestamps.

Remediation

Implement a thorough sanity check for the input timestamp within the `set_price` entry point.

Patch

Fixed in 3716f44 by asserting that the input timestamp is both normalized and not set to a future date.

```diff
DIFF
    pub fn set_price(e: Env, admin: Address, updates: Vec<i128>, timestamp: u64)
    ...
+        let timeframe: u64 = e.get_resolution().into();
+        let ledger_timestamp = now(&e);
+        if timestamp == 0 || !timestamp.is_valid_timestamp(timeframe) || timestamp
     ↪  > ledger_timestamp {
+            panic_with_error!(&e, Error::InvalidTimestamp);
+        }
    ...
```

OS-REF-ADV-02 [low] | Missing Admin Verification

Description

The existing implementation of `config` lacks a crucial validation step to confirm that the caller's address aligns with the authorized admin address (`config.admin`). This oversight poses a notable security risk, as providing an incorrect address to `config.admin` may result in unauthorized control over the deployed contract.

RUST
```rust
pub fn config(e: Env, admin: Address, config: ConfigData) {
    admin.require_auth();
    if e.is_initialized() {
        e.panic_with_error(Error::AlreadyInitialized);
    }
    e.set_admin(&config.admin);
    ...
}
```

Remediation

Require the authorization from `config.admin`.

Patch

Fixed in b7855da.

OS-REF-ADV-03 [low] | Inconsistency Due To Incomplete Price Data

Description

The current implementation of `twap` uses `prices` to retrieve the latest price data necessary for calculating the time-weighted average price. It computes the average from these data points. This approach precisely calculates the time-weighted average price across N equally spaced values. However, it becomes inaccurate if a value is missing in the sequence, leading to unequal intervals between the data points and resulting in an average that does not accurately represent the time-weighted average price.

Remediation

Revise `twap` to accurately compute the time-weighted average price, even with missing intermediate prices, by applying the standard algorithm that weights each price by its corresponding time interval.

Patch

Fixed in 23f0e48 by returning `None` on encountering missing intermediate prices.

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 represent anti-patterns and may result in security issues in the future.

ID	Description
OS-REF-SUG-00	The current behavior of `div_floor`, wherein it returns zero when the divisor is zero, may not adequately handle exceptional cases and result in unintended outcomes.
OS-REF-SUG-01	Optimize `__add_assets` by checking if an asset already exists utilizing the existing `get_asset_index` method.

OS-REF-SUG-00 | Handling Division By Zero

Description

In the current implementation, when the divisor (`priceB`) is zero, `div_floor` returns zero. While it may be unlikely for the backend to push such a price, it is crucial to consider and handle such exceptional cases robustly. As an improvement, update `div_floor` to panic when the divisor is zero, ensuring that any attempt to perform division by zero results in a clear and explicit error.

```rust
fn div_floor(dividend: i128, divisor: i128, decimals: u32) -> i128 {
    if (dividend == 0) || (divisor == 0) {
        0_i128;
    }
    ...
```

Remediation

Modify `div_floor` to incorporate a check for a zero divisor (`priceB`). In such a scenario, configure the function to trigger a panic, preventing unintended division by zero. Accomplish this by asserting that the divisor is not equal to zero before executing the division operation.

Patch

Fixed in bf6fa94 by panicking instead of returning zero in such events.

OS-REF-SUG-01 | Asset Management Optimization

Description

`__add_assets` may be optimized to enhance efficiency and readability and minimize resource costs.

RUST

```rust
fn __add_assets(e: &Env, assets: Vec<Asset>) {
    let assets_len = assets.len();
    if assets_len == 0 || assets_len >= 256 {
        panic_with_error!(&e, Error::InvalidUpdateLength);
    }
    let mut presented_assets = e.get_assets();

    let mut assets_indexes: Vec<(Asset, u32)> = Vec::new(&e);
    for asset in assets.iter() {
        //check if the asset has been already added
        if e.get_asset_index(&asset).is_some() {
            panic_with_error!(&e, Error::AssetAlreadyExists);
        }
        presented_assets.push_back(asset.clone());
        assets_indexes.push_back((asset, presented_assets.len() - 1));
    }

    e.set_assets(presented_assets);
    for (asset, index) in assets_indexes.iter() {
        e.set_asset_index(&asset, index);
    }
}
```

Remediation

Eliminate the `asset_indexes` vector and conduct the `set_asset_index` operations directly within the initial `for` loop.

Patch

Fixed in 0207362.

A | Proofs of Concept

OS-REF-ADV-00

The following is the test case we prepared:

```diff
diff --git a/src/test.rs b/src/test.rs
index f141ce8..bcfc5aa 100644
--- a/src/test.rs
+++ b/src/test.rs
@@ -181,6 +181,34 @@ fn add_assets_test() {
     assert_eq!(result, expected_assets);
 }
 
+#[test]
+fn index_overflow_bug() {
+    let (env, client, init_data) = init_contract_with_admin();
+
+    env.mock_all_auths();
+
+    let admin = &init_data.admin;
+
+    let mut assets = vec![
+        &env,
+    ];
+    for i in 0..=0x100 {
+        assets.push_back(Asset::Other(Symbol::new(&env, &("Asset".to_string() +
    ↪  &i.to_string()))));
+    }
+
+    client.add_assets(&admin, &assets);
+    let mut updates = vec![&env];
+    for i in 0..=0x100 {
+        updates.push_back(normalize_price(i as i128 + 1));
+    }
+    client.set_price(&admin, &updates, &600_000);
+    let asset0 = assets.get(0).unwrap();
+    let asset256 = assets.get(0x100).unwrap();
+    let price_data = client.x_price(&asset256, &asset0, &600_000).unwrap();
+    std::println!("price: {:?}", price_data.price);
+    assert_eq!(price_data.price,
    ↪  updates.get_unchecked(0x100).fixed_div_floor(updates.get_unchecked(0),
    ↪  DECIMALS));
+}
+
 #[test]
 fn set_period_test() {
     let (env, client, init_data) = init_contract_with_admin();
```

B | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a 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 that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still 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.

C | 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 on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, 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 comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. 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.