Soroswap

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Andreas Mantzoutas andreas@osec.io

Contents

01 | Executive Summary

Overview

Soroswap Finance engaged OtterSec to assess the `soroswap-core` program. This assessment was conducted between December 11th and December 27th, 2023. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 6 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability, concerning the use of instance storage to store data about token pairs, resulting in a denial of service due to limited space (OS-SWP-ADV-00) and another issue, regarding the burning of liquidity pool tokens without appropriately adjusting the internal pool state causing an inconsistency between the liquidity pool token supply and the actual liquidity (OS-SWP-ADV-01). We further identified a case of round-down division during the fee calculation, resulting in a lower fee amount (OS-SWP-ADV-02).

We also made recommendations around the presence of an integer overflow (OS-SWP-SUG-00) and advised against the lack of an address check during pair creation, which may result in the creation of pairs with similar addresses (OS-SWP-SUG-01). Additionally, we highlighted the incorrect authorization logic utilized for token transfers enabling the use of phishing attacks to steal users funds (OS-SWP-SUG-02).

02 | Scope

The source code was delivered to us in a Git repository at github.com/soroswap/core. This audit was performed against commit d891832.

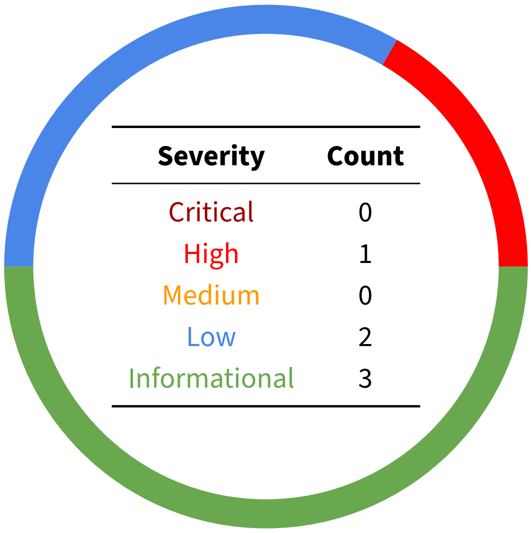
A brief description of the programs is as follows:

Name	Description
soroswap-core	An automated liquidity protocol employing the constant product formula implemented within a system of non-upgradable smart contracts on the Soroban blockchain, where each Soroswap pair smart contract oversees a liquidity pool composed of reserves consisting of two tokens. These tokens may be Soroban native tokens implementing the Soroban token interface or Stellar assets.

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	0
Low	2
Informational	3

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 A.

ID	Severity	Status	Description
OS-SWP-ADV-00	High	Resolved	Unbounded data stored in the Instance storage could lead to a denial of service
OS-SWP-ADV-01	Low	Resolved	Burning liquidity pool tokens without appropriately adjusting the internal pool state may result in an inconsistency between the liquidity pool token supply and the actual liquidity.
OS-SWP-ADV-02	Low	Resolved	The fee calculation in swap is prone to round-down division, rounding down the fee.

OS-SWP-ADV-00 [high]| Incorrect Use Of Instance Storage

Description

`SoroswapFactory` currently utilizes instance storage to store all token pairs. As the documentation describes, this instance storage is designed for small, frequently utilized data directly associated with the contract instance.

factory/src/lib.rs RUST

```rust
fn add_pair_to_all_pairs(e: &Env, pair_address: &Address) {
    // Get the current `allPairs` vector from storage
    let mut all_pairs = get_all_pairs(e);
    // Push the new `pair_address` onto the vector
    all_pairs.push_back(pair_address.clone());
    // Save the updated `allPairs` vector to storage
    e.storage().instance().set(&DataKey::AllPairs, &all_pairs);
}
```

Thus, storing vectors of pair addresses (`allPairs`) in the instance storage results in inadequate storage space, particularly as the data of pair tokens grows limitlessly in a permissionless manner, swiftly surpassing the constrained storage capacity. Malicious users may exploit this by flooding the instance storage with fake token pairs, resulting in a denial of service scenario as the storage space is exhausted.

Proof of Concept

1. An attacker repeatedly calls `create_pair`, creating a large number of pairs between arbitrary tokens.
2. `add_pair_to_all_pairs` appends each new pair address to the `allPairs` vector in the instance storage. Since the contract stores vectors, assuming the creation of a significant number of pairs, the storage size quickly approaches its limit.
3. As the instance storage reaches its capacity (64 KB), disruptions of legitimate operations of `SoroswapFactory` may occur since the contract will not be able to store additional data in the instance storage due to its limitations.

Remediation

Utilize persistent storage for large or potentially unbounded data sets, such as in the case above. However, do not use vectors, as storing vectors directly in persistent storage may also result in issues with the ledger entry size limit.

Patch

Fixed in cb7304f.

OS-SWP-ADV-01 [low] | Incorrect Implementation Of Burn Logic

Description

Liquidity pool tokens represent the ownership of liquidity provided to a liquidity pool. The contract mints them when a user provides liquidity to the pool and burns them when the user wants to withdraw their liquidity. While burning liquidity pool tokens of a Soroswap pair pool, the `burn` method in its current implementation burns tokens (`amount`) from an account (`from`), but it fails to adjust the number of tokens in that liquidity pool.

src/soroswap_pair_token/contract.rs RUST

```rust
fn burn(e: Env, from: Address, amount: i128) {
    from.require_auth();
    check_nonnegative_amount(amount);
    e.storage()
        .instance()
        .bump(INSTANCE_LIFETIME_THRESHOLD, INSTANCE_BUMP_AMOUNT);
    spend_balance(&e, from.clone(), amount);
    TokenUtils::new(&e).events().burn(from, amount);
}
```

Thus, when a liquidity provider calls `burn` to withdraw their liquidity and claim the fees, the contract only burns the tokens from the liquidity provider's account. Still, it does not adjust the pool's internal state (`total_shares`), resulting in an inconsistency where the LP token supply suggests the presence of more liquidity than what is contained in the pool. This disparity in the amount of liquidity pool tokens may impact the decentralized exchange's pricing mechanism and overall stability.

Remediation

Ensure when invoking `burn`, it also adjusts the relevant internal state of the liquidity pool by decrementing `total_shares`.

Patch

Fixed in ee9e40f.

OS-SWP-ADV-02 [low] | Rounding Error In Fee Calculation

Description

The protocol applies a 0.30% fee on the trade, which a user pays on invoking `swap`. The code calculates the fee by multiplying the input amount `amount_0_in or amount_1_in` by three and dividing the result by 1000, essentially calculating a 0.3% fee on the input amounts.

pair/src/lib.rs RUST

```rust
fn swap(e: Env, amount_0_out: i128, amount_1_out: i128, to: Address)->
    ↪  Result<(),SoroswapPairError> {
    [...]
    if amount_0_in == 0 && amount_1_in == 0 {
        return Err(SoroswapPairError::SwapInsufficientInputAmount);
    }
    if amount_0_in < 0 || amount_1_in < 0 {
        return Err(SoroswapPairError::SwapNegativesInNotSupported);
    }
    let fee_0 = (amount_0_in.checked_mul(3).unwrap()).checked_div(1000).unwrap();
    let fee_1 = (amount_1_in.checked_mul(3).unwrap()).checked_div(1000).unwrap();
    [...]
}
```

However, these operations involve integer arithmetic, giving rise to rounding errors, especially when dealing with fractional percentages. The result of `amount_0_in.checked_mul(3).unwrap()` may not be perfectly divisible by 1000, which may truncate decimal values, resulting in the actual fee value to be rounded down to a lower amount, enabling the user to pay less fees.

Remediation

Employ a ceiling division operation to ensure the user always pays the full fee amount, preventing under-payment.

Patch

Fixed in 4eca4c1 and 8460791.

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-SWP-SUG-00	Presence of integer overflow in `receive_balance`.
OS-SWP-SUG-01	`Pair::new` lacks a same address check allowing a Pair to be created with two identical addresses.
OS-SWP-SUG-02	After a user grants permission to a contract with `require_auth` at the top invocation, malicious contracts can insert harmful authorization objects into the authorization tree, potentially leading to phishing attacks on the user.

OS-SWP-SUG-00 | Integer Overflow

Description

`receive_balance` reads the existing balance of a given address, adds the provided amount to it, and then writes the updated balance back to the storage. There is a potential integer overflow in the addition operation `balance + amount`, where if the sum of balance and amount exceeds the maximum representable value for the `i128` data type, it will result in an overflow.

pair/src/soroswap_pair_token/balance.rs RUST

```rust
pub fn receive_balance(e: &Env, addr: Address, amount: i128) {
    let balance = read_balance(e, addr.clone());
    write_balance(e, addr, balance + amount);
}
```

Remediation

Implement explicit overflow checks utilizing `checked_add`.

Patch

Fixed in 716822e.

OS-SWP-SUG-01 | Missing Address Check

Description

`Pair::new` establishes a consistent ordering of token addresses within a pair. However, the current implementation lacks a check in `Pair::new` to ensure that when creating a new pair, the addresses a and b are distinct. Since `Pair` is intended to represent a unique pair of addresses, attempting to create a pair with identical addresses violates the expected semantics of a pair.

factory/src/lib.rs RUST

```rust
pub fn new(a: Address, b: Address) -> Self {
    if a < b {
        Pair(a, b)
    } else {
        Pair(b, a)
    }
}
```

Including a check for similar addresses during creation of a new pair will ensure an early failure in `Pair::new`, rather than casing an error in `get_pair` and `pair_exists`.

Remediation

Add an address check in `Pair::new` which validates the input parameters, a and b are not the same addresses.

factory/src/lib.rs RUST

```rust
pub fn new(a: Address, b: Address) -> Self {
    if a == b {
        panic!("Same addresses");
    }

    if a < b {
        Pair(a, b)
    } else {
        Pair(b, a)
    }
}
```

Patch

Fixed in 83c17c3.

OS-SWP-SUG-02 | Authorization Required For Token Transfer

Description

Soroswap pools necessitate user authorization to transfer tokens from the user to the pair. In `swap_exact_tokens_for_tokens`, `require_auth` is invoked to validate and authorize token transfers during a swap.

router/src/lib.rs RUST

```rust
fn swap_exact_tokens_for_tokens(
        [...]
    ) -> Result<Vec<i128>, CombinedRouterError> {
        check_initialized(&e)?;
        check_nonnegative_amount(amount_in)?;
        check_nonnegative_amount(amount_out_min)?;
        to.require_auth();
        ensure_deadline(&e, deadline)?;
        [...]
}
```

While this is the method proposed by the Stellar Development Foundation, an issue arises when the router uses `require_auth` during the initial smart contract call, as malicious contracts can exploit this by inserting fraudulent authorization objects into the authorization tree. This deceptive tactic can lead to phishing attacks, where attackers trick users into validating dangerous tokens. Gaining wrongful authorization for external calls through `transfer`, they can exploit this to execute unauthorized transfers, stealing the user's entire balance of any token.

Remediation

Implement frontend warnings and alerts to inform users about potential risks when dealing with unverified pairs, and exclude unverified tokens from your off-chain router.

A | 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.

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 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.