Swaylend

Security Assessment

October 28th, 2024 — Prepared by OtterSec

Renato Eugenio Maria Marziano renato@osec.io

James Wang james.wang@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Swaylend engaged OtterSec to assess the `swaylend-monorepo` program. This assessment was conducted between October 9th and October 18th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we highlighted the possibility of specific functions aborting due to calculations reverting on certain combinations of token decimals (OS-SWL-ADV-00).

We also made a recommendation to remove redundant code within the system for increased readability (OS-SWL-SUG-03) and advised incorporating additional checks within the codebase for improved robustness and security (OS-SWL-SUG-01). We further suggested modifying the codebase for enhanced functionality, efficiency, and maintainability (OS-SWL-SUG-02).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/Swaylend/swaylend-monorepo. This audit was performed against 14934ae.

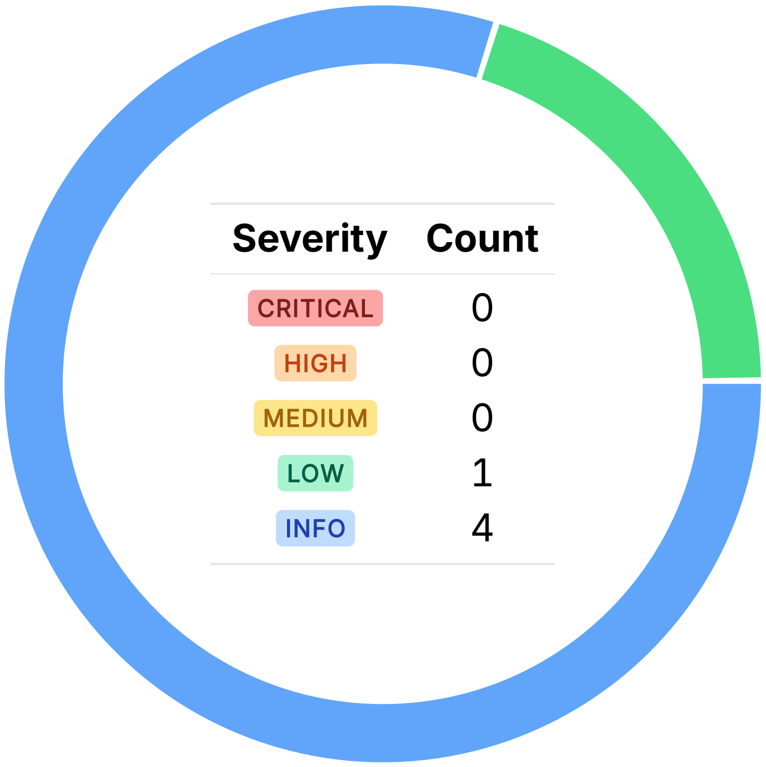
A brief description of the program is as follows:

Name	Description
swaylend-monorepo	A decentralized lending platform operating on the Fuel Network, which utilizes an Ethereum consensus layer.

03 — Findings

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



Severity	Count
CRITICAL	0
HIGH	0
MEDIUM	0
LOW	1
INFO	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 A.

ID	Severity	Status	Description
OS-SWL-ADV-00	LOW	RESOLVED ✓	The calculations in `available_to_borrow` and `collateral_value_to_sell` may revert due to potential mismatches in token decimal configurations.

Possible Revert Due to Improper Values LOW OS-SWL-ADV-00

Description

In `available_to_borrow` and `collateral_value_to_sell`, the calculations for scale have the potential to revert under specific conditions related to token decimals.

Remediation

Check the token decimal values to ensure they do not result in a revert.

Patch

Resolved in 485d442.

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-SWL-SUG-00	Calculating `market_basic.total_supply_base` and `market_basic.total_borrow_base` as present values rather than their expected principal amounts may confuse users.
OS-SWL-SUG-01	There are several instances where proper validation is not done, resulting in potential security issues.
OS-SWL-SUG-02	Recommendation for modifying the codebase for improved functionality, efficiency, and maintainability.
OS-SWL-SUG-03	The codebase contains multiple cases of redundant code that should be removed for better maintainability and clarity.

Dual purpose of fields OS-SWL-SUG-00

Description

There is a potential confusion in how `get_market_basics_with_interest` handles the `total_supply_base` and `total_borrow_base` fields. Normally, these fields are expected to represent principal values, which are the amounts initially supplied or borrowed without accounting for accrued interest. However, within this function, the fields are updated to hold present values (the principal adjusted for accrued interest). This design may result in problems for users of the function who expect the original principal values to be returned instead of present values.

>_ *contracts/market/src/main.sw* RUST

```rust
fn get_market_basics_with_interest() -> MarketBasics {
    [...]
    market_basic.total_supply_base = present_value_supply(
        market_basic
            .base_supply_index,
        market_basic
            .total_supply_base,
    );
    market_basic.total_borrow_base = present_value_borrow(
        market_basic
            .base_borrow_index,
        market_basic
            .total_borrow_base,
    );
    market_basic
}
```

Remediation

Introduce separate fields for principal and present values, or clearly state in the documentation that `total_supply_base` and `total_borrow_base` reflect present values after the function is called.

Missing Validation Logic OS-SWL-SUG-01

Description

1. The per-collateral pausing should be checked more often. In `withdraw_collateral`, for instance, it is important to ensure that the flag for per-collateral pausing is checked before allowing the withdrawal.
2. In the current implementation of `src-20`, anyone may call `emit_src20_events`, resulting in arbitrary event emissions that may not reflect the actual state of the contract. This affects the integrity and transparency of the contract, especially since events are meant to log significant state changes. Implementing proper access control is necessary to restrict event emissions.

   >_ *contracts/src-20/src/main.sw* RUST

   ```rust
   fn emit_src20_events() {
       // Metadata that is stored as a configurable should only be emitted once.
       let asset = AssetId::default();
       let sender = msg_sender().unwrap();
       let name = Some(String::from_ascii_str(from_str_array(NAME)));
       let symbol = Some(String::from_ascii_str(from_str_array(SYMBOL)));
       SetNameEvent::new(asset, name, sender).log();
       SetSymbolEvent::new(asset, symbol, sender).log();
       SetDecimalsEvent::new(asset, DECIMALS, sender).log();
       TotalSupplyEvent::new(asset, storage.total_supply.read(), sender).log();
   }
   ```

3. The current design of `src-20::constructor` and `market::activate_contract` carries the risk of front-running, where anyone may call these functions before the intended user to set themselves as the owner and modify configuration parameters. Ensure that only a predefined address (such as the deployer's address) may call the initialization functions.

Remediation

Add the above-stated validations.

Code Refactoring OS-SWL-SUG-02

Description

1. Within `available_to_borrow`, include the supply in the borrowable amount calculation to allow users to borrow more effectively, reflecting their total liquidity (collateral + supply). This adjustment will provide a more accurate representation of available borrowing power.
2. The current approach of rounding down in `present_value_borrow` may result in underestimating the effective debt that users owe. Rounding up will provide a more conservative estimate of the user's debt, ensuring that they account for the maximum possible obligation.

>_ *contracts/market/src/main.sw* RUST

```rust
pub fn present_value_borrow(base_borrow_index: u256, principal: u256) -> u256 {
    principal * base_borrow_index / BASE_INDEX_SCALE_15
}
```

3. It would be appropriate to opt for a more prudent approach by utilizing the upper or lower boundary for price estimations in the market module instead of `PricePosition::Middle`. Given that oracle prices are inherently estimates, relying on the middle value may introduce unnecessary risk, especially if the protocol's intent is to create an additional buffer or margin for safety.

Remediation

Implement the above-mentioned suggestions.

Redundant Code OS-SWL-SUG-03

Description

1. In `market`, the `price` check in `get_price_internal` can be rewritten as `price.price != 0` because the value is of type `u64`, which will always be positive. Similarly, the `reserves < I256::zero()` check in `buy_collateral` is unnecessary because `market_configuration.target_reserves` is always non-negative.
2. In the current protocol implementation, there are no external calls to untrusted contracts, eliminating the risk of control flow manipulation and reentrancy attacks. As a result, the reentrancy checks are redundant and may be safely removed.

Remediation

Remove the above instances of redundant code.

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.

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.

INFO

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 others may have 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.