// Private Smart Contract Security Assessment

08.26.2024 - 09.10.2024

Swaylend Protocol

Reserve

HALBORN

Swaylend Protocol - Reserve

Prepared by: HALBORN

Last Updated 09/17/2024

Date of Engagement by: August 26th, 2024 - September 10th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	1	1	1	0	0

TABLE OF CONTENTS

1. Introduction

`Reserve Labs` engaged Halborn to conduct a security assessment on their lending project, beginning on August 26, 2024, and ending on September 10, 2024. The security assessment was scoped to cover their `swaylend-monorepo` GitHub repository, located at https://github.com/Swaylend/swaylend-monorepo with commit ID `5d7b294035c14e62980c2bdabf9ac51d394235c0`.

2. Assessment Summary

The team at Halborn was provided two weeks for the engagement and assigned one full-time security engineer to assess the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to achieve the following:

- Ensure that the system operates as intended.
- Identify potential security issues.
- Identify lack of best practices within the codebase.
- Identify systematic risks that may pose a threat in future releases.

In summary, Halborn identified some security issues that were successfully addressed by the `Reserve Labs team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Manual code review and walkthrough.
- Manual testing by custom scripts.

4. RISK METHODOLOGY

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

- **10** - CRITICAL
- **9 - 8** - HIGH
- **7 - 6** - MEDIUM
- **5 - 4** - LOW
- **3 - 1** - VERY LOW AND INFORMATIONAL

5. SCOPE

FILES AND REPOSITORY

(a) Repository: swaylend-monorepo

(b) Assessed Commit ID: 5d7b294

(c) Items in scope:

- contracts/token/src/main.sw
- contracts/market/src/main.sw

Out-of-Scope:

REMEDIATION COMMIT ID:

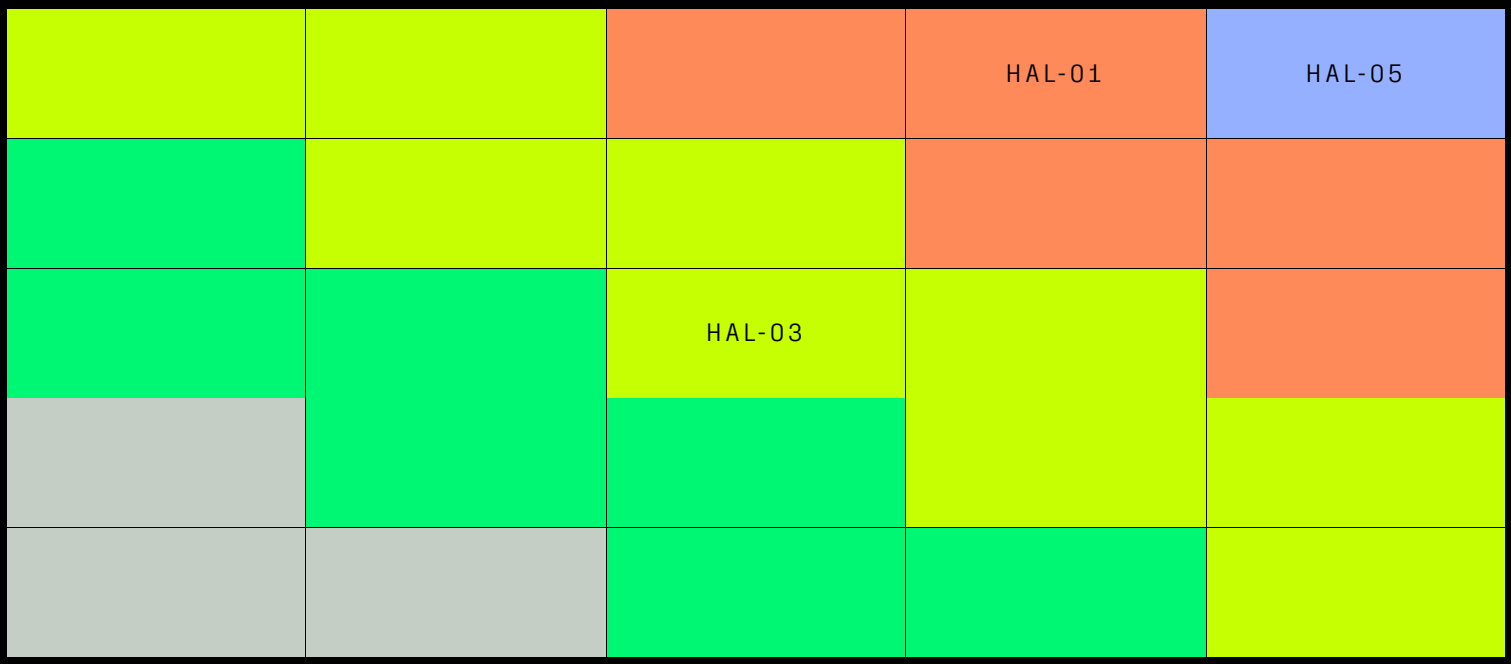
- 622bd07

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	1	0	0

IMPACT X LIKELIHOOD



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING FUNDS VALIDATION IN UPDATE_PRICE_FEEDS	CRITICAL	SOLVED - 09/13/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING STALENESS CHECKS IN ORACLE QUERIES	HIGH	SOLVED - 09/13/2024
MISSING PRICE FEED VALIDATION	MEDIUM	SOLVED - 09/13/2024

7. FINDINGS & TECH DETAILS

7.1 MISSING FUNDS VALIDATION IN UPDATE_PRICE_FEEDS

// CRITICAL

Description

The function `update_price_feeds` is responsible of updating the Pyth oracle by paying a given fee and pushing the most recent price (as Pyth oracles follow a push-like behavior where users are the ones which update the price feed at will). However, it is not checked that the funds sent by the user are indeed the passed `update_fee`, so such amount is retrieved from the internal balances of the market (which is a loss of funds for depositors).

Proof of Concept

The function `update_price_feeds` is responsible of updating the Pyth price feed from within the market contract. However, Pyth oracles need a fee being paid upfront, as seen in https://github.com/pyth-network/pyth-crosschain/blob/e3d8bfeff19906afc9640a52cadc165473560ff1/target_chains/fuel/contracts/pyth-contract/src/main.sw#L422

```
#[storage(read, write), payable]
fn update_price_feeds(update_data: Vec<Bytes>) {
    require(
        msg_asset_id() == AssetId::base(),
        PythError::FeesCanOnlyBePaidInTheBaseAsset,
    );

    ...

    let required_fee = total_fee(total_number_of_updates, storage.single_update_fee);
    require(msg_amount() >= required_fee, PythError::InsufficientFee);

    ...
}
```

For that, the function `update_price_feeds` and its internal version calls `update_price_feeds` in the oracle endpoint and passes as fee `update_fee`. However, it does not check that the `msg.value` is equal to `update_fee`, so it is possible to update the price feeds by sending funds from the market balance instead of the caller assets.

```
#[payable, storage(read)]
fn update_price_feeds_internal(update_fee: u64, update_data: Vec<Bytes>) {
    let contract_id = storage.pyth_contract_id.read();
    require(contract_id != ZERO_B256, Error::OracleContractIdNotSet);

    let oracle = abi(PythCore, contract_id);
    oracle.update_price_feeds {
        asset_id: FUEL_ETH_BASE_ASSET_ID, coins: update_fee
    } // @audit not checked update_fee == msg.value
    (update_data);
}
```

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:C/Y:N/R:N/S:C (8.4)

Recommendation

Require that the passed value is equal to the `update_fee` argument to avoid using market's funds.

Remediation

SOLVED: The code now checks for the passed amount to be higher or equal than `update_fee` and the token passed is ETH.

Remediation Hash

https://github.com/Swaylend/swaylend-monorepo/commit/622bd073b48cfc211e8b7d6d0474186c61903e6f#diff-17baa8be2b4713b0056e503d498adfbeaf1e214298728ef86d5ef6dc7e71a810

7.2 MISSING STALENESS CHECKS IN ORACLE QUERIES

// HIGH

Description

When using third-party oracles, it is recommended to check the timestamp of a price feed against a staleness factor to avoid using a stale price that did not reflect the real value of the underlying asset. However, this is not done in the lending market in scope, which triggers serious issues as stated in the next section.

Proof of Concept

In Fuel, Pyth price feeds follow the next structure:
https://github.com/pyth-network/pyth-crosschain/blob/e3d8bfeff19906afc9640a52cadc165473560ff1/target_chains/fuel/contracts/pyth-interface/src/data_structures/price.sw#L20C1-L31C2

```
pub struct Price {
    // Confidence interval around the price
    pub confidence: u64,
    // Price exponent
    // This value represents the absolute value of an i32 in the range -255 to 0. Values other than 0, should
be considered negative:
    // exponent of 5 means the Pyth Price exponent was -5
    pub exponent: u32,
    // Price
    pub price: u64,
    // The TAI64 timestamp describing when the price was published
    pub publish_time: u64,
}
```

to check the staleness of the price feed, the protocol using it must check for `publish_time` to not be too far in the past. However, it is not done in `get_price_internal`:

```
// # 10. Pyth Oracle management
#[storage(read)]
fn get_price_internal(price_feed_id: PriceFeedId) -> Price {
    let contract_id = storage.pyth_contract_id.read();
    require(contract_id != ZERO_B256, Error::OracleContractIdNotSet);

    let oracle = abi(PythCore, contract_id);
    let price = oracle.price(price_feed_id);
    price
}
```

As Pyth oracles follow a push behavior, where the price feed is not updated until a third actor updates it, then it is possible for users to "game" the market by interacting with it where the Pyth oracle returns a favorable price that does not reflect the real value of the asset, update it, and then gain advantage of it by interacting again /for example, borrowing more assets than the real value of its collateral.

BVSS

AO:A/AC:M/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (5.2)

Recommendation

Check `publish_time` to not be too far in the past against a staleness factor.

Remediation

SOLVED: The code now checks for the price publish time to not be too far in the future nor in the past.

Remediation Hash

https://github.com/Swaylend/swaylend-monorepo/commit/622bd073b48cfc211e8b7d6d0474186c61903e6f#diff-17baa8be2b4713b0056e503d498adfbeaf1e214298728ef86d5ef6dc7e71a810

7.3 MISSING PRICE FEED VALIDATION

// MEDIUM

Description

The `get_price_internal` function does not perform input validation on the `price`, `confidence`, and `exponent` values returned from the called price feed, which can lead to the contract accepting invalid or untrusted prices. Those values should be checked as clearly stated in the official documentation.

Proof of Concept

The function is as follows:

```
// # 10. Pyth Oracle management
#[storage(read)]
fn get_price_internal(price_feed_id: PriceFeedId) -> Price {
    let contract_id = storage.pyth_contract_id.read();
    require(contract_id != ZERO_B256, Error::OracleContractIdNotSet);

    let oracle = abi(PythCore, contract_id);
    let price = oracle.price(price_feed_id);
    price
}
```

which can be seen does not perform any type of input validation.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:N/R:N/S:C (4.7)

Recommendation

The function should revert the transaction if one of the following conditions is triggered:

- `price == 0`
- `confidence > 0 && (price / confidence) < MIN_CONF_RATIO` for a given `MIN_CONF_RATIO`

Remediation

SOLVED: The recommended checks are now done in function `get_price_internal`.

Remediation Hash

https://github.com/Swaylend/swaylend-monorepo/commit/622bd073b48cfc211e8b7d6d0474186c61903e6f#diff-17baa8be2b4713b0056e503d498adfbeaf1e214298728ef86d5ef6dc7e71a810

References

https://solodit.xyz/issues/m-01-pyth-oracle-price-is-not-validated-properly-pashov-audit-group-none-nabla-markdown

---