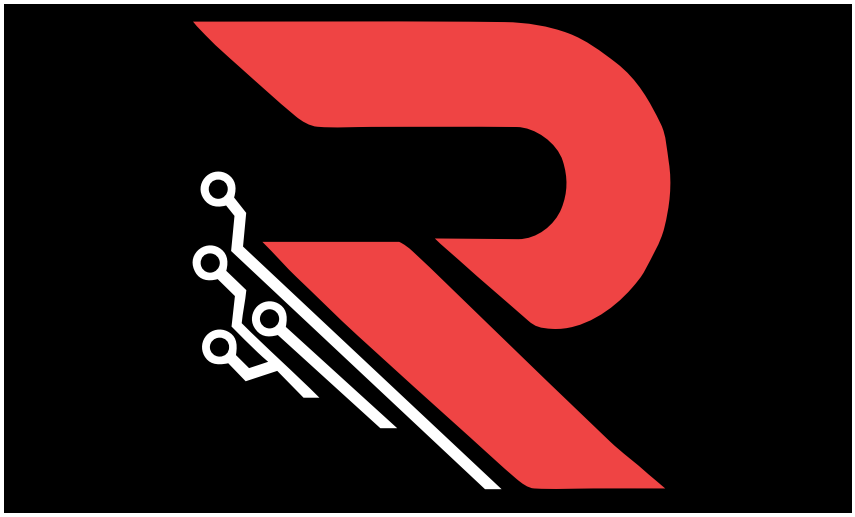
Spectra Security Review



Version 1.0

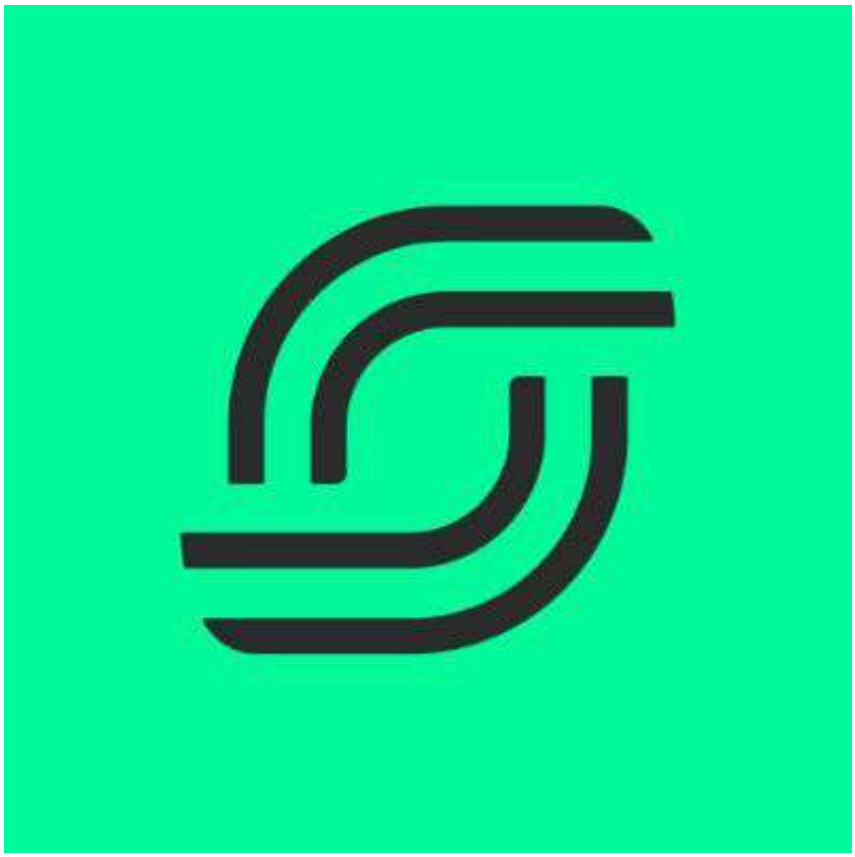
14.02.2025

Conducted by:

MaslarovK, Lead Security Researcher

radev-eth, Lead Security Researcher

Table of Contents



1 About MaslarovK

MaslarovK a Security Reseacher and Co-Founder of Rezolv Solutions.

2 About radev.eth

radev_eth a Security Reseacher and Co-Founder of Rezolv Solutions.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

4 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	MaxAPY
Repository	https://github.com/perspectivefi/Spectra-price-oracle
Commit hash	7845d5515568b86a19851bd477823bf633685e3a
Resolution	N/A
Documentation	N/A
Methods	Manual review

Scope

src/SpectraPriceOracle.sol
src/models/OracleLinearModel.sol
src/models/OracleZCBModel.sol

Issues Found

Critical risk	0
High risk	0
Medium risk	0
Low risk	1
Informational	1

6 Findings

6.1 Low 1: Incorrect SECONDS_PER_YEAR constant in SpectraPriceOracle.sol and LinearDiscountModel.sol (OracleLinearModel) contracts

6.1.1 Description

The `SECONDS_PER_YEAR` constant in `SpectraPriceOracle` and `LinearDiscountModel` is hardcoded as **365 days (31,536,000 seconds)**. However, this approach introduces two major issues:

1. Leap Year Inaccuracy
    - A year is not always 365 days; leap years have **366 days (31,622,400 seconds)**.
    - During leap years, the discount calculation will return incorrect values, leading to inconsistencies in Principal Token (PT) pricing.
2. Incorrect Approximation of the True Year Length
    - The precise average year length, accounting for leap years, is **365.2425 days (31,556,952 seconds)**.
    - The incorrect `365 days` assumption **slightly overestimates discounts**, leading to **minor miscalculations that accumulate over long-term maturities**.

This issues leads to minor errors in discount calculations that will **compound over multi-year maturities**. The further into the future a PT matures, the larger the accumulated error.

6.1.2 Recommendation

You have two solutons here: 1. Replace the hardcoded `SECONDS_PER_YEAR` with a dynamic function that adjusts for leap years:
“‘solidity function getSecondsPerYear() public view returns (uint256) { uint16 year = getCurrentYear(); // Implement a function to fetch the current year return (isLeapYear(year) ? 31_622_400 : 31_536_000); }

```
function isLeapYear(uint16 year) internal pure returns (bool) {
    return (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0));
}
```

‘‘‘

2. Alternatively, for a simpler fix, update `SECONDS_PER_YEAR` to the more accurate average year length: `solidity uint256 private constant SECONDS_PER_YEAR = 31_556_952; // 365.2425 days`

We recommend the second fix.

6.2 Informational/Recommendation 1: Adding `getCurrentDiscount()` for improved usability

6.2.1 Description

Currently, `getDiscount()` requires **manual input of `timeLeft` and `futurePTValue`**, which external contracts and users must fetch separately. This adds complexity and potential inconsistencies in discount calculations.

Here are some small problems with the current approach:
- Redundant external calls increase gas costs**.
- Inconsistent data risks if `timeLeft` and `futurePTValue` are fetched at different block timestamps. (really low likelihood)
- Maybe, higher integration difficulty for external DeFi protocols.

6.2.2 Recommendation

Implement a new `getCurrentDiscount()` function that automatically fetches `timeLeft` and `futurePTValue` internally:

```
function getCurrentDiscount() public view returns (uint256) {
    uint256 timeLeft = ptTimeLeft();
    uint256 futurePTValue = IPrincipalToken(PT).convertToUnderlying(UNIT);
    return getDiscount(timeLeft, futurePTValue);
}
```

This simplifies integrations**, reduces redundant calls, and ensures synchronized pricing data.