Parabol Labs - Protocol Contracts

Security Assessment

May 14, 2024

Prepared for:
Emre Colakoglu
Parabol Labs

Prepared by: **Kurt Willis**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 12, 2024	Technical onboarding call
March 15, 2024	Project kickoff call
March 25, 2024	Week 1 report readout meeting
April 1, 2024	Final report readout meeting
May 14, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Parabol Labs engaged Trail of Bits to review the security of the Parabol Labs Protocol smart contracts.

A team of one consultant conducted the review from March 18 to March 29, 2024, for a total of two engineer-weeks of effort. Our testing efforts focused on the `ReserveStabilityPool` and the `NonFungibleNotePosition` contracts. With full access to source code and documentation, we performed static and dynamic testing of the code base, using automated and manual processes.

Observations and Impact

Aside from one high-severity issue (TOB-PRBL-4) regarding an accounting error that should be remedied, the protocol appears to be in a relatively mature state. The smart contracts are well-architected and simple, and they do not include unnecessary features.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Parabol Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Devise a written incident response plan.** A written and rehearsed incident response plan can be a deciding factor when it is necessary to respond to a security incident in a swift and coordinated manner.
- **Include public, user-facing documentation about the multisig setup.** Expanding the public documentation will help users gain an understanding of the degree of decentralization and the risks associated with interacting with the protocol.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	0
Low	5
Informational	7
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	11
Undefined Behavior	2

Project Goals

The engagement was scoped to provide a security assessment of the Parabol Labs Protocol Contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are all initialization functions secured?
- Are access controls in place for privileged functions?
- Does the possibility exist of users not being able to retrieve their funds?
- Can a note be claimed or closed early without proper permission?
- Are price feeds valid only for the given time range?
- Are a note's fixed and floating income calculated correctly?
- Will the protocol correctly handle a user reclaiming their principal when a floating income value is not present?
- Is the accumulated floating income invariant of non-descending values always guaranteed?

Project Targets

The engagement involved a review and testing of the following target.

parabol-protocol-contracts

Repository	https://github.com/Parabol-Finance/parabol-protocol-contracts
Version	commit `7537ffb07528975985a70ac53cfbc5640a53b90d`
Type	EVM
Platform	Solidity

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- A manual review of the codebase
- A review of the documentation
- A review of access controls
- A manual and automatic review of the protocol's arithmetic
- Running tests and measuring coverage
- Running our static analysis tool, Slither, and triaging results
- Writing proof-of-concept tests

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- `contracts/libraries/DateTime.sol`
- `contracts/libraries/ParabolNoteDescriptor.sol`
- `contracts/libraries/ParabolNoteSVG.sol`

Less focus was given to these contracts, as the resulting NFTs are displayed on NFT marketplaces as intended. These contracts pertain to cosmetic aspects, and issues in these would not imply security-related issues. In the case that they do not display as intended, they can be upgraded easily.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The protocol contains simple calculations for the generated income and fee values. These are generally handled well; however, one high-severity issue (TOB-PRBL-4) highlights an oversight pertaining to the income calculation.	**Moderate**
Auditing	Sufficient events are emitted for important functions and state updates. Parabol Labs intends to build off-chain monitoring systems. An incident response plan does not exist, but one is being revised. See here for our guidelines.	**Moderate**
Authentication / Access Controls	All relevant contracts contain well-defined roles and access controls for privileged functions. The principle of least privilege is followed.	**Satisfactory**
Complexity Management	The individual contracts are abstracted and compartmentalized well. The protocol does not contain unnecessary features, and internal functions are not used excessively and only whenever logic can be repurposed. `ERC20BaseUpgradeable` and `ERC20PermitUpgradeable` could be merged for simplicity.	**Satisfactory**
Decentralization	The protocol incorporates mild security checks in order to not lock users out. However, the entire system holding user funds is upgradeable and controlled by a single admin address. System parameters can be changed at any time, and users are not given the chance to opt out.	**Weak**
Documentation	Contracts, functions, parameters, and events are sufficiently documented. Inline comments are in place.	**Moderate**

	Public user-facing documentation exists that describes the protocol in broad terms. However, more technical documentation, a glossary, and an overview of the protocol architecture, the multisig considerations, and user flows could be beneficial.	
Low-Level Manipulation	The contracts avoid any unnecessary use of assembly or unsafe code. Unchecked arithmetic is used sparingly.	**Satisfactory**
Testing and Verification	The unit tests show a high level of function and branch coverage. One high-severity issue (TOB-PRBL-4) highlights the need for testing under further edge-cases and with previously initialized data.	**Moderate**
Transaction Ordering	The protocol does not exhibit any behavior relevant to transaction ordering.	**Not Applicable**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Incorrect ERC-7201 storage location in Denylister	Data Validation	Low
2	ERC20BaseStorage._version is not initialized	Data Validation	Informational
3	Incorrect comparison when setting the minimum lending limit	Data Validation	Low
4	Income is erroneously calculated using accumulated income	Data Validation	High
5	ERC20BaseUpgradeable and ERC721PermitUpgradeable should not signal support of IERC1271	Undefined Behavior	Low
6	Updating the previous floating income can break an invariant	Undefined Behavior	Low
7	Incorrect parameter and event description	Data Validation	Low
8	Inconsistent use of day parameter in event emission	Data Validation	Informational
9	Protocol reports zero floating income for unset values	Data Validation	Informational
10	Floating income is denominated in unclear units	Data Validation	Informational
11	Unusual denominator values	Data Validation	Informational
12	ERC20BaseUpgradeable initializer does not initialize ERC20Upgradeable	Data Validation	Informational
13	Unnecessarily restrictive data type for timestamps	Data Validation	Informational

Detailed Findings

1. Incorrect ERC-7201 storage location in Denylister

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PRBL-1
Target: `contracts/Denylister.sol`	

Description
The ERC-7201 storage location is calculated incorrectly, potentially causing storage collisions.

The ERC-7201 Namespaced Storage Layout location is defined by `erc7201(id: string) = keccak256(keccak256(id) - 1) & ~0xff`. The `Denylister` contract includes the storage location as a constant.

```
/**
 * @dev The storage slot location used for the DenylisterStorage struct.
 * This location is calculated to ensure no storage collisions with inherited
contracts, utilizing a unique identifier.
 * The calculation method
keccak256(abi.encode(uint256(keccak256("parabol.storage.Denylister")) - 1)) &
~bytes32(uint256(0xff))
 */
bytes32 private constant DenylisterStorageLocation =
    0x86307cd2a0b90ef396d609e421d149707f3c882bc4511db533172ea45bcbbf00;
```

Figure 1.1: The ERC-7201 storage location is given as a constant value. (contracts/Denylister.sol#33–39)

However, the shown storage location constant does not equal the result from the computation:

`keccak256(abi.encode(uint256(keccak256("parabol.storage.Denylister")) - 1)) & ~bytes32(uint256(0xff)) = 0x0b7f09e080729a6f01210a857b22417bb37e71acad7810ed3f0850eb44452c00`.

If the storage location is unique to the `Denylister` contract, the impact is limited. However, if this constant equals the ERC-7201 storage location of another contract, then a storage collision could have disastrous effects.

Exploit Scenario

The storage location results in a collision with another mapping that is used for access control. Due to the storage collision, as soon as a user is denylisted, they are also given admin access.

Recommendations

Short term, use the correct storage location: `0x0b7f09e080729a6f01210a857b22417bb37e71acad7810ed3f0850eb44452c00`.

Long term, consider using the compiler's ability for constant evaluation of expressions at compile time instead of using a constant value. Alternatively, include an assert statement that will revert if the storage location is not as expected, and include tests that are able to verify that these storage locations are computed correctly.

2. ERC20BaseStorage._version is not initialized

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PRBL-2
Target: `contracts/base/ERC20BaseUpgradeable.sol`	

Description

The version value in the `ERC20Base` contract is not initialized, leading to the contract reporting an empty version string.

`ERC20Base`'s ERC-7201 storage struct is defined with the following values.

```
struct ERC20BaseStorage {
    bytes32 _nameHash;
    bytes32 _versionHash;
    string _version;
}
```

Figure 2.1: The ERC-7201 storage struct for ERC20Base is defined. (`contracts/base/ERC20BaseUpgradeable.sol#29–33`)

The `_version` parameter is read when retrieving the token's version.

```
/**
 * @notice Returns the version of the token as a plain text string.
 * @dev This function allows external entities to retrieve the version of the token.
 * @return The version of the token.
 */
function version() external view returns (string memory) {
    return _getERC20BaseStorage()._version;
}
```

Figure 2.2: The version function (`contracts/base/ERC20BaseUpgradeable.sol#59–66`)

The parameter itself is, however, never initialized, and it is not configurable.

```
function __ERC20BaseUpgradeable_init_unchained(
    string memory name_,
    string memory version_
) internal onlyInitializing {
    ERC20BaseStorage storage $ = _getERC20BaseStorage();
    $._nameHash = keccak256(bytes(name_));
    $._versionHash = keccak256(bytes(version_));
```

```
}
```

Figure 2.3: The ERC20Base contract's initialization function (`contracts/base/ERC20BaseUpgradeable.sol#89–96`)

This leads to the contract reporting an empty string as the version identifier.

Recommendations

Short term, initialize the version parameter.

Long term, expand the test suite to ensure 100% function coverage.

3. Incorrect comparison when setting the minimum lending limit

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRBL-3
Target: `contracts/ReserveStabilityPool.sol`	

Description
An incorrect comparison could result in a contract reversion when setting the minimum lending limit.

The `setMinLendLimit` function includes a check that aims to prevent the admin from setting the minimum lending limit to the already existing value.

```
if ($._withdrawalFee == minLendLimit_)
    revert ReserveStabilityPool__SameMinLendLimit();
```

Figure 3.1: The `minLendLimit` is being compared to `withdrawalFee` (`contracts/ReserveStabilityPool.sol#300–301`)

The check is performed incorrectly as the new minimum lending limit value is being compared against the withdrawal fee value stored in the contract storage.

Exploit Scenario
The admin wants to reduce the minimum lending limit. As the value coincides with the withdrawal fee, the function reverts.

Recommendations
Short term, correct the compared values.

```
if ($._minLendLimit == minLendLimit_)
```

Figure 3.2: A correct comparison using the stored `minLendLimit` value

Long term, double-check all values when copying and modifying functions.

4. Income is erroneously calculated using accumulated income

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PRBL-4
Target: `contracts/ReserveStabilityPool.sol`	

Description

When calculating a note's floating income value, an accounting error can lead to a user receiving too much income.

When a user redeems their non-fungible note position, the note's total income is calculated. The total income consists of the note's fixed income and its floating income.

```
function _calculateTotalIncome(
    INonFungibleNotePosition.Note memory lendInfo
) internal view returns (uint256) {
    return
        _calculateFixedIncome(
            lendInfo.principal,
            lendInfo.coupon,
            lendInfo.maturityTimestamp,
            lendInfo.lendTimestamp
        ) +
        _calculateFloatingIncome(
            lendInfo.principal,
            lendInfo.maturityTimestamp,
            lendInfo.lendTimestamp
        );
}
```

Figure 4.1: The total income consists of the note's fixed income and its floating income. (contracts/ReserveStabilityPool.sol#652–667)

The floating income is calculated from the daily accumulated floating income values.

```
function _calculateFloatingIncome(
    uint256 principal,
    uint32 lastTimestamp,
    uint32 lendTimestamp
) internal view returns (uint256) {
    ReserveStabilityPoolStorage
        storage $ = _getReserveStabilityPoolStorage();

    if ($._lastFloatingIncomeUpdateDay == 0) return 0;
```

```
    uint32 lendDay = getDay(lendTimestamp);
    if ($._lastFloatingIncomeUpdateDay < lendDay) return 0;

    uint256 lendDayIncome = $._accFloatingIncome[lendDay];

    uint256 lendTimestampInSecond = ((lendDay + 1) * 1 days) -
        lendTimestamp;

    uint256 lendDayUserIncome = (lendTimestampInSecond *
        lendDayIncome *
        principal) / 1e25;

    uint32 maturityDay = getDay(lastTimestamp) - 1;

    uint256 lastAccumulatedFloatingIncome;

    if ($._lastFloatingIncomeUpdateDay < maturityDay) {
        lastAccumulatedFloatingIncome = $._accFloatingIncome[
            $._lastFloatingIncomeUpdateDay
        ];
    } else {
        lastAccumulatedFloatingIncome = $._accFloatingIncome[maturityDay];
    }

    uint256 remainingDaysUserIncome = ((lastAccumulatedFloatingIncome -
        $._accFloatingIncome[lendDay]) *
        principal *
        24 hours) / 1e25;

    uint256 totalIncome = lendDayUserIncome + remainingDaysUserIncome;
    return totalIncome;
}
```

Figure 4.2: The floating income is calculated.
(contracts/ReserveStabilityPool.sol#700–741)

The calculation is split in two—the floating income from the day the loan started (`lendDayIncome`) and the floating income that accumulated over the loan duration counted in full days (`remainingDaysUserIncome`).

In order to retrieve the amount that was accumulated over the days of the loan, `remainingDaysUserIncome` correctly uses `accFloatingIncome[lastDay] - accFloatingIncome[lendDay]`.

When determining the fraction of the income for the day where the loan started, the entire accumulated income up until that day is used instead of the amount that just accumulated for that day.

```
uint256 lendDayIncome = $._accFloatingIncome[lendDay];
```

Figure 4.3: The income for the lend day is read using the accumulated floating income values. (contracts/ReserveStabilityPool.sol#713)

The correct calculation should take the difference of the accumulated floating income values to the previous day: `accFloatingIncome[lendDay] - accFloatingIncome[lendDay - 1]`.

This leads to an accounting error, where the users will be paid out too-high amounts.

Exploit Scenario

Bob loans 1M ParaUSD to ReserveStabilityPool. His income for the loan day is reported too high by $100K, since he deposited early in the day. Too many unbacked ParaUSD tokens are minted.

Recommendations

Short term, fix the accounting error for the income of the loan day: `lendDayIncome = accFloatingIncome[lendDay] - accFloatingIncome[lendDay - 1]`.

Long term, expand the test suite to ensure that edge case scenarios such as these are captured.

5. ERC20BaseUpgradeable and ERC721PermitUpgradeable should not signal support of IERC1271

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PRBL-5
Target: `contracts/ReserveStabilityPool.sol`	

Description
Both `ERC20BaseUpgradeable` and `ERC721PermitUpgradeable` signal support of IERC1271 while not supporting the interface, which could lead to composability issues.

Both `ERC20BaseUpgradeable` and `ERC721PermitUpgradeable` report their support of the IERC1271 interface.

```
function supportsInterface(
    bytes4 interfaceId
) public view virtual override returns (bool) {
    return
        interfaceId == type(IERC1271).interfaceId ||
        super.supportsInterface(interfaceId);
}
```

Figure 5.1: ERC20BaseUpgradeable's `supportsInterface` function (`contracts/base/ERC20BaseUpgradeable.sol#180–186`)

The contracts use ERC-1271's signature validation standard for contracts, yet the contracts that perform the validation themselves usually do not support the interface function `isValidSignature`, which is required by the signing contracts.

Exploit Scenario
A contract checks if the Parabol contracts support IERC-1271 in order to then perform a signature check. The signature validation reverts, because the contracts do not implement IERC-1271.

Recommendations
Short term, remove the check that signals support of IERC-1271.

Long term, be aware that implementing the logic for an ERC does not automatically mean that the contract itself supports the ERC interface.

6. Updating the previous floating income can break an invariant

Severity: Low	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PRBL-6
Target: `contracts/ReserveStabilityPool.sol`	

Description

A system invariant could be broken, which could cause accumulated floating income values to become out of sync and potentially block claims.

An unwritten invariant of the accounting system is that all accumulated non-negative floating income values should always be non-decreasing. This is because they represent sequential accumulations of non-negative values. In case this invariant is broken, accounting errors or denial-of-service scenarios are possible. The invariant should hold for all days up until the last day on which the incomes were updated. It is implicitly enforced when updating the values on a particular day or when updating multiple previous floating income values.

Updating the daily floating income is subject to a requirement: that the income can be updated only once per day and only on subsequent days.

```
function updateDailyFloatingIncome(
    uint256 income
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    ReserveStabilityPoolStorage
        storage $ = _getReserveStabilityPoolStorage();

    uint256 day = getDay(block.timestamp) - 1;

    if ($._accFloatingIncome[day] != 0)
        revert ReserveStabilityPool__DailyFloatingIncomeAlreadyUpdated();

    if ($._lastFloatingIncomeUpdateDay != day - 1)
        if ($._lastFloatingIncomeUpdateDay != 0)
            // if not first update
            revert ReserveStabilityPool__PreviousDayNotUpdated();

    uint256 newAccFloatingIncome = $._accFloatingIncome[day - 1] + income;
    $._accFloatingIncome[day] = newAccFloatingIncome;

    $._lastFloatingIncomeUpdateDay = day;

    emit UpdateDailyFloatingIncome(day + 1, income, newAccFloatingIncome);
```

```
}
```

Figure 6.1: The daily floating income values are updated (contracts/ReserveStabilityPool.sol#519–541)

When updating floating income values of previous days, additional checks are required to enforce the system invariant.

```
function updatePreviousFloatingIncome(
    uint256 firstDay,
    uint256 dayCount,
    uint256[] calldata prevAccFloatingIncome
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    if (firstDay == 0 || dayCount == 0)
        revert ReserveStabilityPool__InvalidInputParameter();

    if (dayCount != prevAccFloatingIncome.length)
        revert ReserveStabilityPool__ArrayLengthMismatch();

    uint256 currentDay = getDay(block.timestamp);
    uint256 lastUpdateDay = firstDay + dayCount - 1;

    if (lastUpdateDay >= currentDay)
        revert ReserveStabilityPool__UpdateDayIsNotValid(lastUpdateDay);

    ReserveStabilityPoolStorage
        storage $ = _getReserveStabilityPoolStorage();

    for (uint256 i; i < dayCount; ) {
        $._accFloatingIncome[firstDay + i] =
            $._accFloatingIncome[firstDay + i - 1] +
            prevAccFloatingIncome[i];
        unchecked {
            i++;
        }
    }

    if ($._lastFloatingIncomeUpdateDay < lastUpdateDay)
        $._lastFloatingIncomeUpdateDay = lastUpdateDay;
    else if (
        $._accFloatingIncome[lastUpdateDay + 1] <
        $._accFloatingIncome[lastUpdateDay]
    ) {
        revert ReserveStabilityPool__FloatingIncomeNotSynced();
    }

    // ...
}
```

Figure 6.2: Previous daily floating income values are updated (contracts/ReserveStabilityPool.sol#470–513)

In particular, the last two checks in figure 6.1 aim to enforce this invariant by directly ensuring that the day after the last day being updated requires an accumulated floating income value that is greater than that of the day before.

Through an inappropriate use of the `updatePreviousFloatingIncome` function, this invariant can be broken. This is because the function correctly ensures that the invariant is upheld for the last day, but not for the first day.

Exploit Scenario

An example scenario showcases how the floating income values can become out of sync, causing users to be unable to reclaim their funds.

```
function test_updateDailyFloatingIncome() public {
    uint256 startDay = block.timestamp / 1 days - 1;

    pool.setMinMaturityLimit(1 days);

    // Set some initial floating income values.
    pool.updateDailyFloatingIncome(1);
    skip(1 days);
    pool.updateDailyFloatingIncome(2);
    skip(1 days);

    //   V
    // 1 3 0 0 0 0 0

    // Alice agrees to lend capital for 4 days.
    uint256 principal = 1_000 ether;
    uint256 coupon = 2_000;
    uint256 maturityDuration = 4 days;
    uint256 interest = principal * coupon * (maturityDuration) / (1_000_000 * 360
days);

    (IFeedSignatureVerifier.PriceFeed memory priceFeed,
IFeedSignatureVerifier.Signature memory feedSignature) =
    signPriceFeed({
        maturityTimestamp: block.timestamp + maturityDuration,
        coupon: coupon,
        validAfter: block.timestamp - 1,
        validBefore: block.timestamp + 1
    });

    vm.prank(alice);
    pool.lend(priceFeed, feedSignature, principal);

    pool.updateDailyFloatingIncome(1);
    skip(1 days);
    pool.updateDailyFloatingIncome(4);
    skip(3 days);

    //       V
```

```
    // 1 3 4 8 0 0 0
    assertEq(pool.accFloatingIncome(startDay + 0), 1);
    assertEq(pool.accFloatingIncome(startDay + 1), 3);
    assertEq(pool.accFloatingIncome(startDay + 2), 4);
    assertEq(pool.accFloatingIncome(startDay + 3), 8);
    assertEq(pool.accFloatingIncome(startDay + 4), 0);
    assertEq(pool.accFloatingIncome(startDay + 5), 0);

    uint256[] memory accFloatingIncome = new uint256[](2);
    accFloatingIncome[0] = 1;
    accFloatingIncome[1] = 2;

    pool.updatePreviousFloatingIncome(startDay + 5, 2, accFloatingIncome);

    //             V
    // 1 3 4 8 0 1 3
    assertEq(pool.accFloatingIncome(startDay + 0), 1);
    assertEq(pool.accFloatingIncome(startDay + 1), 3);
    assertEq(pool.accFloatingIncome(startDay + 2), 4);
    assertEq(pool.accFloatingIncome(startDay + 3), 8);
    assertEq(pool.accFloatingIncome(startDay + 4), 0);
    assertEq(pool.accFloatingIncome(startDay + 5), 1);
    assertEq(pool.accFloatingIncome(startDay + 6), 3);

    // Alice is unable to reclaim her funds.
    vm.prank(alice);
    vm.expectRevert(stdError.arithmeticError);

    pool.claim(1);
}
```

Figure 6.3: Proof-of-concept scenario of an out-of-sync `accFloatingIncome`.

Recommendations

Short term, enforce the system invariant through disallowing gaps in the updates—by requiring that the first update day cannot lie past the last update day.

```
if ($._lastFloatingIncomeUpdateDay != 0 && $._lastFloatingIncomeUpdateDay <
firstDay) revert ReserveStabilityPool__FloatingIncomeNotSynced;
```

Figure 6.4: A check that enforces the system invariant

Long term, take note of any sensible system invariants; explicitly state these, and ensure that they are enforced.

7. Incorrect parameter and event description

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRBL-7
Target: `contracts/ReserveStabilityPool.sol`	

Description

The function and event parameters' description for updating previous floating income values does not match up with their usage.

When updating previous floating income values via the `updatePreviousFloatingIncome` function, the `prevAccFloatingIncome` parameter is supplied.

```
/**
 * @dev Updates the previous floating income data for previous days.
 * @param prevAccFloatingIncome An array of previous floating income values.
 */
function updatePreviousFloatingIncome(
    uint256 firstDay,
    uint256 dayCount,
    uint256[] calldata prevAccFloatingIncome
) external onlyRole(DEFAULT_ADMIN_ROLE) {

    // ...

    for (uint256 i; i < dayCount; ) {
        $._accFloatingIncome[firstDay + i] =
            $._accFloatingIncome[firstDay + i - 1] +
            prevAccFloatingIncome[i];
        unchecked {
            i++;
        }
    }

    // ...

    emit UpdatePreviousFloatingIncome(
        firstDay,
        dayCount,
        prevAccFloatingIncome
    );
}
```

Figure 7.1: The `UpdatePreviousFloatingIncome` event is emitted in `updatePreviousFloatingIncome`. (`contracts/ReserveStabilityPool.sol#466–513`)

The NatSpec documentation for `prevAccFloatingIncome` describes it as "An array of previous floating income values." The parameter naming—which includes "Acc"—suggests that the values are accumulated (absolute) values. By looking at the code, however, it is clear that the values are daily income (relative) values.

Further, the event documentation describes the last parameter (`accFloatingIncomes`) as "the accumulated floating income for the day."

```
/**
 * @dev Emitted when previous floating income data is updated.
 * @param accFloatingIncomes The accumulated floating income for the day.
 */
event UpdatePreviousFloatingIncome(
    uint256 firstDay,
    uint256 dayCount,
    uint256[] accFloatingIncomes
);
```

Figure 7.2: The `UpdatePreviousFloatingIncome` event is declared. (`contracts/interfaces/IReserveStabilityPool.sol#66–74`)

The actual values that are emitted in the event represent daily, relative values as opposed to accumulated values.

Exploit Scenario

Due to an accounting error, previous daily income values must be updated. By mistake, the accumulated daily income values are input to the function call, as the parameter name suggests that these are expected. This leads to the protocol paying out users in too-high amounts.

Recommendations

Short term, decide whether the parameter should actually represent accumulated values or daily values, and update the parameter name, documentation, and the related event accordingly. If the parameter represents accumulated values, ensure that the additional check mentioned in issue TOB-PRBL-6 is implemented in order to uphold the invariant of non-decreasing values.

Long term, make sure to conform parameter inputs throughout the protocol to avoid creating any unintended errors.

8. Inconsistent use of day parameter in event emission

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PRBL-8
Target: `contracts/ReserveStabilityPool.sol`	

Description

The parameters for events that are emitted when floating income values are updated are used inconsistently, which could cause confusion.

When updating previous floating income values via `updatePreviousFloatingIncome`, the event emission reflects the *first* day of the values that are updated. When updating the daily floating income value via `updateDailyFloatingIncome`, the event emission reflects the *current* day—the day that the values are updated.

```
/**
 * @dev Emitted when daily data is updated.
 * @param currentDay The day for which data is updated.
 * @param floatingIncome The daily interest rate for the day.
 * @param accFloatingIncome The accumulated floating income for the day.
 */
event UpdateDailyFloatingIncome(
    uint256 currentDay,
    uint256 floatingIncome,
    uint256 accFloatingIncome
);

/**
 * @dev Emitted when previous floating income data is updated.
 * @param accFloatingIncomes The accumulated floating income for the day.
 */
event UpdatePreviousFloatingIncome(
    uint256 firstDay,
    uint256 dayCount,
    uint256[] accFloatingIncomes
);
```

Figure 8.1: The event definitions in `IReserveStabilityPool` (`contracts/interfaces/IReserveStabilityPool.sol#54-74`)

Recommendations

Short term, change the day parameter in the `UpdateDailyFloatingIncome` event to correspond to the day tied to the income value.

Long term, when emitting events, focus on the values that matter to an off-chain indexer in order to prevent the possibility of an error.

9. Protocol reports zero floating income for unset values

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRBL-9
Target: `contracts/ReserveStabilityPool.sol`	

Description

If the floating income values have not been updated until the day of the loan, the function reports zero floating income.

```
uint32 lendDay = getDay(lendTimestamp);
if ($._lastFloatingIncomeUpdateDay < lendDay) return 0;
```

Figure 9.1: A value of zero is returned if the last update day occurs before the lending day. (contracts/ReserveStabilityPool.sol#710–711)

Further, if the values have not been updated on the day of the note's maturity, the closest date will be taken instead.

```
if ($._lastFloatingIncomeUpdateDay < maturityDay) {
    lastAccumulatedFloatingIncome = $._accFloatingIncome[
        $._lastFloatingIncomeUpdateDay
    ];
} else {
    lastAccumulatedFloatingIncome = $._accFloatingIncome[maturityDay];
}

uint256 remainingDaysUserIncome = ((lastAccumulatedFloatingIncome -
    $._accFloatingIncome[lendDay]) *
    principal *
    24 hours) / 1e25;
```

Figure 9.2: The floating income value of the last available date is used (contracts/ReserveStabilityPool.sol#726–737)

This could lead to a user prematurely receiving too-little income and funds being unaccounted for if the values have not been updated in time.

Recommendations

Short term, document this behavior and the expected floating income update frequency. Further, consider including a function that shows a clear demarcation of the normal behavior of the protocol, as shown below:

```
bool isProtocolSynced = $._lastFloatingIncomeUpdateDay >= maturityDay;
```

Figure 9.3: Defining a Boolean that signals an unexpected scenario

The protocol could further enforce that a user must at least leave a margin of X days before the accumulated floating income values are truly reported as zero. This should also be clearly checked on a front-end interface and properly handled.

Long term, improve documentation on what is expected and unexpected behavior in general and reflect this in the code and front end.

10. Floating income is denominated in unclear units

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PRBL-10
Target: `contracts/ReserveStabilityPool.sol`	

Description

The NatSpec comments describe the `income` parameter in `updateDailyFloatingIncome` as *"the calculated per token per wei income generated in the protocol since the last update."*

```
/**
 * @dev Updates the daily floating income data.
 * @param income The calculated per token per wei income generated in the protocol
since the last update.
 */
function updateDailyFloatingIncome(
    uint256 income
) external onlyRole(DEFAULT_ADMIN_ROLE) {
```

Figure 10.1: The `updateDailyFloatingIncome` function (contracts/ReserveStabilityPool.sol#515–521)

The actual usage of the accumulated values suggests that the income parameter should be *"the calculated per token per wei per second income generated in the protocol since the last update multiplied by 1e25."*

```
uint256 remainingDaysUserIncome = ((lastAccumulatedFloatingIncome -
    $._accFloatingIncome[lendDay]) *
    principal *
    24 hours) / 1e25;
```

Figure 10.2: Use of the accumulated floating income values (contracts/ReserveStabilityPool.sol#734–737)

This discrepancy in the usage and the NatSpec description of the parameter could lead to submission of invalid values.

Recommendations

Short term, update the NatSpec documentation to clarify units of the income parameter.

Long term, identify and improve areas of ambiguity.

11. Unusual denominator values

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PRBL-11
Target: `contracts/ReserveStabilityPool.sol`	

Description

A global denominator value is defined; however, it is always multiplied by 100, leading to ambiguity for input units.

The retrievable `DENOMINATOR` value is defined as a constant in the `ReserveStabilityPool` contract.

```
/// @dev A constant used as the denominator in certain mathematical operations.
uint256 public constant DENOMINATOR = 10000;
```

Figure 11.1: The public constant `DENOMINATOR` is defined. (contracts/ReserveStabilityPool.sol#50–51)

As a result, one would expect a fee of 5%, for example, to be defined as 500—as is a common convention for protocols that use Basis Points (a denominator of `10_000`).

The actual usage, however, suggests that a 5% fee value should be defined as `50_000` instead.

```
uint256 fee = (totalIncome * $._protocolFee) / (DENOMINATOR * 100);
```

Figure 11.2: Use of the denominator for calculating the protocol fee (contracts/ReserveStabilityPool.sol#802)

```
uint256 fee = (lendInfo.principal * $._withdrawalFee) /
    (DENOMINATOR * 100);
```

Figure 11.3: Use of the denominator for calculating the withdrawal fee (contracts/ReserveStabilityPool.sol#844–845)

This is uncommon and could be misleading, opening up the possibility of errors.

Recommendations

Short term, remove the extra multiplication by 100 in the denominator to reduce ambiguity. If the same precision is required, increase the denominator constant from `10_000` to `1_000_000`.

Long term, identify areas in the code that could seem ambiguous and lead to human errors, and clarify these.

12. ERC20BaseUpgradeable initializer does not initialize ERC20Upgradeable

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PRBL-12
Target: `contracts/base/ERC20BaseUpgradeable.sol`	

Description

The initialization function of the `ERC20BaseUpgradeable` contract does not initialize its parent ERC20 contract.

```
/**
 * @notice Initializes the ERC20Base contract with a name and version.
 * @dev Sets the initial name and version of the token, hashing both for use in
EIP-712 domain separation.
 * This initializer should be called from an inheriting contract's initializer
function.
 * @param name_ The name of the token.
 * @param version_ The version of the token.
 */
function __ERC20BaseUpgradeable_init(
    string memory name_,
    string memory version_
) internal onlyInitializing {
    __ERC20BaseUpgradeable_init_unchained(name_, version_);
}
```

Figure 12.1: The initialization function of `ERC20BaseUpgradeable` (`contracts/base/ERC20BaseUpgradeable.sol#68–80`)

The initialization function `__ERC20BaseUpgradeable_init` should include a call to `__ERC20_init`, as these calls should be chained.

Currently, this is not an issue as the only contract that uses `ERC20BaseUpgradeable` is `ParabolUSD` and it calls all unchained initializers.

```
__ERC20_init_unchained(name_, symbol_);
__ERC20BaseUpgradeable_init_unchained(name_, version_);
__Pausable_init_unchained();
```

Figure 12.2: ParabolUSD's `initialize` function (`contracts/ParabolUSD.sol#139–141`)

Exploit Scenario
ParabolUSD is adapted to call the default initialization function. Because the call does not initialize the ERC20 contract, the name and symbol fields are left uninitialized.

Recommendations
Short term, include a call to `__ERC20__init`, passing in the name and symbol variables.

Long term, ensure that all base contracts are initialized when using the unchained version.

13. Unnecessarily restrictive data type for timestamps

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PRBL-13
Target: `contracts/base/ERC20BaseUpgradeable.sol`	

Description

The protocol handles all timestamp data as `uint32` types, an unnecessarily small data type.

All timestamps in the protocol are handled as `uint32` types.

```
function _calculateFixedIncome(
    uint256 principal,
    uint256 coupon,
    uint32 maturityTimestamp,
    uint32 lendTimestamp
) internal view returns (uint256) {
    return
        (principal *
            coupon *
            (
                block.timestamp >= maturityTimestamp
                    ? maturityTimestamp - lendTimestamp
                    : block.timestamp - lendTimestamp
            )) / (DENOMINATOR * 360 days * 100);
```

Figure 13.1: The fixed income is calculated using `uint32` timestamps (`contracts/ReserveStabilityPool.sol#677–690`)

`uint32` timestamps are to become invalid in the year 2106, which means that `uint32` is unnecessarily restrictive. Smaller data types (sub `uint256`) are handled by the EVM through additional operations, increasing gas costs. There can be a benefit when packing multiple timestamps in the same storage slot. Only the `Note` struct stores multiple timestamps in storage.

```
struct Note {
    uint32 maturityTimestamp;
    uint256 coupon;
    uint256 principal;
    uint32 lendTimestamp;
}
```

Figure 13.2: The Note struct is declared.
(contracts/interfaces/INonFungibleNotePosition.sol#28–33)

However, the optimizer cannot pack these values into a single slot efficiently due to the storage layout. Both `uint32` data types can be fit into a single slot if they are moved to the start or the end of the struct.

Recommendations

Short term, use a bigger data type to hold the timestamps. A `uint128`, which allows for efficient data packing in storage, can be used.

```
struct Note {
    uint128 lendTimestamp;
    uint128 maturityTimestamp;
    uint256 coupon;
    uint256 principal;
}
```

Figure 13.3: Recommended declaration of the `Note` struct

Long term, identify areas where the protocol might be overly restrictive, especially when this design has no added benefit.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Issues

The following items highlight areas in the code that could improve in code quality.

- **Contracts could be merged.** `ERC20BaseUpgradeable` and `ERC20PermitUpgradeable` could be merged for simplicity.
- **Missing named mapping parameters.**

```
mapping(address => mapping(bytes32 => bool)) _authorizationStates;
```

Figure C.1: Missing named mapping parameters (contracts/base/ERC20AuthUpgradeable.sol#55)

- **Missing ERC-7201 storage annotations.** The following locations are missing these annotations:

```
struct DenylisterStorage {
    /// @notice Mapping of addresses to their denylist status.
    mapping(address => bool) _denylist;
}
```

Figure C.2: contracts/Denylister.sol#28–31

```
struct ERC20AuthStorage {
    mapping(address => mapping(bytes32 => bool)) _authorizationStates;
}
```

Figure C.3: contracts/base/ERC20AuthUpgradeable.sol#54–56

```
struct ERC20BaseStorage {
    bytes32 _nameHash;
    bytes32 _versionHash;
    string _version;
}
```

Figure C.4: contracts/base/ERC20BaseUpgradeable.sol#29–33

```
struct ERC721PermitStorage {
    //@notice The keccak256 hash of the token's name used in the permit signature
verification
    bytes32 _nameHash;
    //@notice The keccak256 hash of the token's version used in the permit signature
verification
    bytes32 _versionHash;
    mapping(uint256 => uint256) _nonces;
}
```

Figure C.5: contracts/base/ERC721PermitUpgradeable.sol#25–31

```
struct FeedSignatureVerifierStorage {
    address _signer;
    bytes32 _nameHash;
    bytes32 _versionHash;
}
```

Figure C.6: contracts/base/FeedSignatureVerifierUpgradeable.sol#29–33

```
struct ParabolUSDStorage {
    IDenylister _denylister;
    uint256 _maxMintLimit;
}
```

Figure C.7: contracts/ParabolUSD.sol#65–68

```
struct ReserveStabilityPoolStorage {
    IParabolUSD _paraUSD;
    INonFungibleNotePosition _nonFungibleNotePosition;
    address _parabolVault;
    uint256 _protocolFee;
    uint256 _withdrawalFee;
    uint256 _minLendLimit;
    uint256 _maxCouponLimit;
    uint256 _lastFloatingIncomeUpdateDay;
    uint32 _minMaturityLimit;
    mapping(uint256 => uint256) _accFloatingIncome;
}
```

Figure C.8: contracts/ReserveStabilityPool.sol#66–77

```
struct NonFungibleNotePositionStorage {
    /// @dev Incremental ID tracker for note positions.
    Counter _tokenIdTracker;
    /// @dev Manages denylisted addresses, blocking their interaction.
    IDenylister _denylister;
    /// @dev Interface to the Reserve Stability Pool for managing lending aspects.
    IReserveStabilityPool _reserveStabilityPool;
    /// @dev Maps token IDs to their corresponding lending information.
    mapping(uint256 => Note) _lendInfos;
}
```

Figure C.9: contracts/NonFungibleNotePosition.sol#46–55

- **Interface support signaling**

```
function supportsInterface(
    bytes4 interfaceId
)
    public
    view
    override(ERC20BaseUpgradeable, AccessControlUpgradeable)
    returns (bool)
```

```
{
    return
        interfaceId == type(IERC20Permit).interfaceId ||
        super.supportsInterface(interfaceId);
}
```

Figure C.10: `ERC20PermitUpgradeable` does support `IERC20Permit`, but `ParabolUSD` does. (contracts/ParabolUSD.sol#286–297)

```
/// @inheritdoc IERC165
function supportsInterface(
    bytes4 interfaceId
)
    public
    view
    override(ERC721PermitUpgradeable, AccessControlUpgradeable, IERC165)
    returns (bool)
{
    return super.supportsInterface(interfaceId);
}
```

Figure C.11: `NonFungibleNotePosition` does not signal support for `IAccessControl`. (contracts/NonFungibleNotePosition.sol#248–258)

- **Missing comment about hash derivation**

```
/// @dev Role identifier for the Reserve Stability Pool.
bytes32 public constant RSP_ROLE =
    0x6a32f3ff343ae66b7d86c9069c431fcd393953a0c476bad3219a146353e0a088;

/// @dev Role identifier for pausing and unpausing contract operations.
bytes32 public constant PAUSER_ROLE =
    0x539440820030c4994db4e31b6b800deafd503688728f932addfe7a410515c14c;
```

Figure C.12: Missing comment about hash derivation (contracts/NonFungibleNotePosition.sol#38–44)

D. Code Optimizations

The following items highlight areas in the code that could improve through optimization.

- **No-ops.** The admin role of any role is already the `DEFAULT_ADMIN_ROLE`, hence the name.

```
_setRoleAdmin(PAUSER_ROLE, DEFAULT_ADMIN_ROLE);
```

Figure D.1: No-ops (contracts/ReserveStabilityPool.sol#217)

```
_setRoleAdmin(MINTER_ROLE, DEFAULT_ADMIN_ROLE);
_setRoleAdmin(BURNER_ROLE, DEFAULT_ADMIN_ROLE);
_setRoleAdmin(PAUSER_ROLE, DEFAULT_ADMIN_ROLE);
```

Figure D.2: No-ops (contracts/ParabolUSD.sol#148–150)

- **Redundant equality comparison.**

```
if (
    _getERC20AuthStorage()._authorizationStates[authorizer][nonce] ==
    true
) revert ERC20Auth__AuthUsedOrCanceled(authorizer, nonce);
```

Figure D.3: Redundant equality comparison (contracts/base/ERC20AuthUpgradeable.sol#280–283)

- **Duplicate operations.**

```
if (
    ecrecover(digest, signature.v, signature.r, signature.s) !=
    _getFeedSignatureVerifierStorage()._signer
) {
    address recoveredAddress = ecrecover(
        digest,
        signature.v,
        signature.r,
        signature.s
    );
```

Figure D.4: Duplicate operations: `ecrecover` is called twice (contracts/base/FeedSignatureVerifierUpgradeable.sol#164–173)

- **Duplicate checks.**

```
function burn(address from, uint256 amount) external onlyRole(BURNER_ROLE) {
    if (amount == 0) {
```

```
        revert ParabolUSD__ZeroAmount();
    }
    if (balanceOf(from) < amount) {
        revert ParabolUSD__InsufficientBalance();
    }
    if (from != msg.sender) {
        if (allowance(from, msg.sender) < amount) {
            revert ParabolUSD__TokenNotApproved(from, msg.sender, amount);
        }
        _spendAllowance(from, msg.sender, amount);
    }

    _burn(from, amount);
    emit Burn(from, amount);
}
```

Figure D.5: A user's balance is already checked in `_burn`.(contracts/ParabolUSD.sol#224–240)

```
function permitLend(
    PriceFeed calldata priceFeed,
    Signature calldata feedSignature,
    uint256 principal,
    uint256 permitDeadline,
    Signature calldata permitSignature
) external whenNotPaused {
    try
        _getReserveStabilityPoolStorage()._paraUSD.permit(
            msg.sender,
            address(this),
            principal,
            permitDeadline,
            permitSignature.v,
            permitSignature.r,
            permitSignature.s
        )
    {} catch {
        if (
            _getReserveStabilityPoolStorage()._paraUSD.allowance(
                msg.sender,
                address(this)
            ) < principal
        ) revert ReserveStabilityPool__InsufficientAllowance();
    }

    _lend(priceFeed, feedSignature, principal);
}
```

Figure D.6: A user's allowance is already checked in `_lend`'s `transferFrom`. (contracts/ReserveStabilityPool.sol#367–394)

```
function permitClaim(
```

```
    uint256 tokenId,
    uint256 permitDeadline,
    Signature calldata permitSignature
) external whenNotPaused {
    try
        _getReserveStabilityPoolStorage()._nonFungibleNotePosition.permit(
            address(this),
            tokenId,
            permitDeadline,
            permitSignature.v,
            permitSignature.r,
            permitSignature.s
        )
    {} catch {
        if (
            _getReserveStabilityPoolStorage()
                ._nonFungibleNotePosition
                .getApproved(tokenId) != address(this)
        ) revert ReserveStabilityPool__TokenNotApproved();
    }

    _claim(tokenId);
}
```

Figure D.7: A user's allowance is already checked in `_claim`'s `transferFrom`. (contracts/ReserveStabilityPool.sol#401–424)

```
function setVerifierSigner(
    address newSigner_
) external override onlyRole(DEFAULT_ADMIN_ROLE) {
    if (newSigner_ == address(0))
        revert ReserveStabilityPool__ZeroAddress();

    _setVerifierSigner(newSigner_);
}
```

Figure D.8: `_setVerifierSigner` already contains a check for the zero address. (contracts/ReserveStabilityPool.sol#317–324)

- **Non-configurable variables should be immutable.**

```
struct ERC20BaseStorage {
    bytes32 _nameHash;
    bytes32 _versionHash;
    string _version;
}
```

Figure D.9: contracts/base/ERC20BaseUpgradeable.sol#29–33

```
//@notice The keccak256 hash of the token's name used in the permit signature
verification
bytes32 _nameHash;
```

```
//@notice The keccak256 hash of the token's version used in the permit signature
verification
bytes32 _versionHash;
```

Figure D.10: contracts/base/ERC721PermitUpgradeable.sol#26–29

```
bytes32 _nameHash;
bytes32 _versionHash;
```

Figure D.11: contracts/base/FeedSignatureVerifierUpgradeable.sol#31–32

- **Duplicate storage reads**

```
_requireOwned(tokenId);
if (block.timestamp > deadline) revert ERC721Permit__DeadlineExpired();

address owner = ownerOf(tokenId);
```

Figure D.12: Duplicate storage reads (contracts/base/ERC721PermitUpgradeable.sol#102–105)

The owner can directly be retrieved from `_requireOwned`:

```
address owner = _requireOwned(tokenId);
```

Figure D.13: Recommended use: the owner can directly be retrieved from `_requireOwned`.

- **Storage variables can be cached.** The new accumulated floating income values from the previous day can be cached.

```
for (uint256 i; i < dayCount; ) {
    $._accFloatingIncome[firstDay + i] =
        $._accFloatingIncome[firstDay + i - 1] +
        prevAccFloatingIncome[i];
    unchecked {
        i++;
    }
}
```

Figure D.14: Storage variables can be cached (contracts/ReserveStabilityPool.sol#490–497)

A local variable can keep track of the accumulated floating income:

```
uint256 accFloatingIncome = $._accFloatingIncome[firstDay - 1];
for (uint256 i; i < dayCount; ) {
    $._accFloatingIncome[firstDay + i] =
        accFloatingIncome += prevAccFloatingIncome[i];
    unchecked {
        i++;
    }
```

```
}
```

Figure D.15: Recommended use: a variable on the stack keeps track of the accumulated floating income.

```
if ($._lastFloatingIncomeUpdateDay != day - 1)
    if ($._lastFloatingIncomeUpdateDay != 0)
        // if not first update
        revert ReserveStabilityPool__PreviousDayNotUpdated();
```

Figure D.16: The last floating income update day (`$._lastFloatingIncomeUpdateDay`) can be cached from storage. (contracts/ReserveStabilityPool.sol#530–533)

A local variable can cache `_lastFloatingIncomeUpdateDay`:

```
uint256 lastUpdate = $._lastFloatingIncomeUpdateDay;
if (lastUpdate != day - 1)
    if (lastUpdate != 0)
        // if not first update
        revert ReserveStabilityPool__PreviousDayNotUpdated();
```

Figure D.17: Recommended use: _lastFloatingIncomeUpdateDay is cached on the stack.

```
if ($._lastFloatingIncomeUpdateDay == 0) return 0;

uint32 lendDay = getDay(lendTimestamp);
if ($._lastFloatingIncomeUpdateDay < lendDay) return 0;

uint256 lendDayIncome = $._accFloatingIncome[lendDay];

uint256 lendTimestampInSecond = ((lendDay + 1) * 1 days) -
    lendTimestamp;

uint256 lendDayUserIncome = (lendTimestampInSecond *
    lendDayIncome *
    principal) / 1e25;

uint32 maturityDay = getDay(lastTimestamp) - 1;

uint256 lastAccumulatedFloatingIncome;

if ($._lastFloatingIncomeUpdateDay < maturityDay) {
    lastAccumulatedFloatingIncome = $._accFloatingIncome[
        $._lastFloatingIncomeUpdateDay
    ];
} else {
    lastAccumulatedFloatingIncome = $._accFloatingIncome[maturityDay];
}
```

Figure D.18: The last floating income update day (`$._lastFloatingIncomeUpdateDay`) can be cached from storage. (contracts/ReserveStabilityPool.sol#708–732)

```
if (msg.sender != $._nonFungibleNotePosition.ownerOf(tokenId))
    revert ReserveStabilityPool__NotTokenOwner();

INonFungibleNotePosition.Note memory lendInfo = $
    ._nonFungibleNotePosition
    .getLendInfo(tokenId);

if (!_isMaturityPassed(lendInfo.maturityTimestamp))
    revert ReserveStabilityPool__MaturityNotPassed();

uint256 totalIncome = _calculateTotalIncome(lendInfo);
uint256 fee = (totalIncome * $._protocolFee) / (DENOMINATOR * 100);
uint256 userIncome = totalIncome - fee;

$._nonFungibleNotePosition.burn(tokenId);

if (fee > 0) $._paraUSD.mint($._parabolVault, fee);

if (userIncome > 0) $._paraUSD.mint(msg.sender, userIncome);

$._paraUSD.transfer(msg.sender, lendInfo.principal);
```

Figure D.19: The addresses stored in `$._nonFungibleNotePosition` and `$._paraUSD` can be cached from storage. (contracts/ReserveStabilityPool.sol#791–811)

```
if (msg.sender != $._nonFungibleNotePosition.ownerOf(tokenId))
    revert ReserveStabilityPool__NotTokenOwner();

INonFungibleNotePosition.Note memory lendInfo = $
    ._nonFungibleNotePosition
    .getLendInfo(tokenId);

if (_isMaturityPassed(lendInfo.maturityTimestamp))
    revert ReserveStabilityPool__MaturityPassed();

uint256 fee = (lendInfo.principal * $._withdrawalFee) /
    (DENOMINATOR * 100);

uint256 floatingIncome = _calculateFloatingIncome(
    lendInfo.principal,
    lendInfo.maturityTimestamp,
    lendInfo.lendTimestamp
);

$._nonFungibleNotePosition.burn(tokenId);

if (floatingIncome > 0)
    $._paraUSD.mint($._parabolVault, floatingIncome);

if (fee > 0) $._paraUSD.transfer($._parabolVault, fee);

uint256 refundAmount = lendInfo.principal - fee;
if (refundAmount > 0) $._paraUSD.transfer(msg.sender, refundAmount);
```

Figure D.20: The addresses stored in `$._nonFungibleNotePosition` and `$._paraUSD` can be cached from storage. (`contracts/ReserveStabilityPool.sol#834–861`)

Inefficient storage layout

```
struct Note {
    uint32 maturityTimestamp;
    uint256 coupon;
    uint256 principal;
    uint32 lendTimestamp;
}
```

Figure D.21: Note's storage layout does not pack the variables `maturityTimestamp` and `lendTimestamp` into one slot. (`contracts/interfaces/INonFungibleNotePosition.sol#28–33`)

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On April 22, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Parabol Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 13 issues described in this report, Parabol Labs has resolved all 13 issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Incorrect ERC-7201 storage location in Denylister	Resolved
2	ERC20BaseStorage._version is not initialized	Resolved
3	Incorrect comparison when setting the minimum lending limit	Resolved
4	Income is erroneously calculated using accumulated income	Resolved
5	ERC20BaseUpgradeable and ERC721PermitUpgradeable should not signal support of IERC1271	Resolved
6	Updating the previous floating income can break an invariant	Resolved
7	Incorrect parameter and event description	Resolved
8	Inconsistent use of day parameter in event emission	Resolved
9	Protocol reports zero floating income for unset values	Resolved
10	Floating income is denominated in unclear units	Resolved
11	Unusual denominator values	Resolved

12	ERC20BaseUpgradeable initializer does not initialize ERC20Upgradeable	**Resolved**
13	Unnecessarily restrictive data type for timestamps	**Resolved**

Detailed Fix Review Results

TOB-PRBL-1: Incorrect ERC-7201 storage location in Denylister
Resolved in commit 26f7839. The Denylister's pre-computed storage location was updated correctly.

TOB-PRBL-2: ERC20BaseStorage._version is not initialized
Resolved in commit 12d51ed. The version parameter is now initialized.

TOB-PRBL-3: Incorrect comparison when setting the minimum lending limit
Resolved in commit 32503f4. The comparison now checks the correct variable.

TOB-PRBL-4: Income is erroneously calculated using accumulated income
Resolved in commit 435b334. The income calculation now correctly takes into account only the income of the single lending day.

TOB-PRBL-5: ERC20BaseUpgradeable and ERC721PermitUpgradeable should not signal support of IERC1271
Resolved in commit 86e4364. The ERC1271 supported interface is now removed.

TOB-PRBL-6: Updating the previous floating income can break an invariant
Resolved in commit 1b2384c. A check now prevents the first day of the update from occurring after the last update day of the floating income value, correctly preserving the invariant.

TOB-PRBL-7: Incorrect parameter and event description
Resolved in commit 647f046. The parameter and event descriptions have been updated to clarify whether accumulated or non-accumulated values are expected. The event now emits both parameter versions.

TOB-PRBL-8: Inconsistent use of day parameter in event emission
Resolved in commit 1c189cf. The day parameter was updated to match the usage in the `updatePreviousFloatingIncome` function and reflects the day that is linked to the new value.

TOB-PRBL-9: Protocol reports zero floating income for unset values
Resolved in commit 26a81fe. An external function has been included that reports whether the floating income values are synchronized or not. The client has further provided the following comment:

> *"we will show on the front-end whether the contract is synchronized and indicate that it updates daily at the same time"*

TOB-PRBL-10: Floating income is denominated in unclear units
Resolved in commit a421c77. The NatSpec has been updated to clarify the denominated units of the floating income parameter.

TOB-PRBL-11: Unusual denominator values
Resolved in commit 7aad9d4. The undocumented multiplication x100 has been removed.

TOB-PRBL-12: ERC20BaseUpgradeable initializer does not initialize ERC20Upgradeable
Resolved in commit b7fde77. The initialization function now also initializes the ERC20 base contract.

TOB-PRBL-13: Unnecessarily restrictive data type for timestamps
Resolved in commit 3ff9441. The maturity and lend timestamps have been updated to a `uint128` type. The time parameters in the Note struct have been reordered, allowing for a packed encoding in storage.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.