Reserve

Fix Review

August 11, 2022

Prepared for:

Nevin Freeman
Reserve

Matt Elder
Reserve

Taylor Brent
Reserve

Julian Rodriguez
Reserve

Luis Camargo
Reserve

Thomas Mattimore
Reserve

Prepared by: **Anish Naik**

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 80+ 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.
228 Park Ave S #80688
New York, NY 10003
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.

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.

Table of Contents

Executive Summary

Engagement Overview

Reserve engaged Trail of Bits to review the security of its Reserve protocol. From May 30 to June 24, 2022, a team of two consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's scope, timeline, test targets, and coverage are provided in the original audit report.

Reserve contracted Trail of Bits to review the fixes implemented for issues identified in the original report. On August 4, 2022, one consultant conducted a review of the client-provided source code, with eight person-hours of effort.

Summary of Findings

The original audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the original findings is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	2
Low	3
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	12
Undefined Behavior	2

Overview of Fix Review Results

Reserve has sufficiently addressed most of the issues described in the original audit report. We reviewed only the fixes related to the `p1` release candidate and the associated contracts; we did not review changes made to the `p0` release candidate or to any of the other components that were not part of the original audit's scope.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineer was associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Project Timeline

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

Date	Event
May 26, 2022	Pre-project kickoff call
June 6, 2022	Status update meeting #1
June 10, 2022	Status update meeting #2
June 17, 2022	Status update meeting #3
June 27, 2022	Delivery of report draft and report readout meeting
August 11, 2022	Delivery of final report and fix review

Project Methodology

Our work in the fix review included the following:

- A review of the findings in the original audit report
- A manual review of the client-provided source code and configuration material
- A review of the documentation provided alongside the codebase

Project Targets

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

Reserve protocol

Repository	https://github.com/reserve-protocol/protocol
Version	`3f1d16f0b62869c7fc932f8e9887826f5c93ce56`
Type	Solidity
Platform	Ethereum

Summary of Fix Review Results

The table below summarizes each of the original findings and indicates whether the issue has been sufficiently resolved.

ID	Title	Status
1	Solidity compiler optimizations can be problematic	Unresolved
2	Lack of a two-step process for contract ownership changes	Partially Resolved
3	Unbounded and invalidly bounded system parameters may cause undefined behavior	Resolved
4	All auction initiation attempts may fail	Resolved
5	Per-block issuance limit can be bypassed	Resolved
6	All attempts to initiate auctions of defaulted collateral tokens will fail	Resolved
7	Fallen-target auctions can be prevented from occurring	Resolved
8	Faulty RToken issuance-cancellation process	Resolved
9	Token auctions may not cover entire collateral token deficits	Resolved
10	Inability to validate the recency of Aave and Compound oracle data	Resolved
11	An RSR seizure could leave the StRSR contract unusable	Resolved
12	System owner has excessive privileges	Resolved

13	Lack of zero address checks in Deployer constructor	**Resolved**
14	RTokens can be purchased at a discount	**Resolved**
15	Inconsistent use of the FixLib library	**Resolved**

Detailed Fix Review Results

1. Solidity compiler optimizations can be problematic	
Status: **Unresolved**	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-RES-1
Target: `hardhat.config.js`	

Description
The Reserve protocol contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Fix Analysis
This issue has not been resolved. The Reserve team has accepted the additional technical risk that results from using `solc`'s optimization features. Additional context provided by the team is included verbatim below.

> We have measured the gas savings from optimizations, and have concluded that:

- Many of our contracts exceed the contract size limit without optimization. We prefer the risks associated with using the optimizer to the risks associated with the increasing complexity that would arise from breaking single, coherent contracts into multiple deployments.

- The gas savings on our three major user functions (RToken `issue`, `vest`, and `redeem`) are 14%, 11%, and 18%. We expect that our users will benefit more from these gas savings than they are likely to lose from the risk of optimizer failure.

We mitigate (but do not eliminate) this risk in three main ways:

1. We compile our contracts using the default runs value of 200 in all of our core system contracts. This ensures that the optimizations that we are using are the most commonly-used optimizations, and thus optimizer bugs are marginally less likely.

2. We are not using the very newest version of `solc`. Instead, we've chosen our compiler version to be as old as possible, while still including the language features on which we rely and any known security patches that may affect the correctness of our code. Thus, it's marginally less likely for serious compiler bugs to exist in that version.

3. Our system is upgradable and contains an emergency-stop mechanism. While this cannot eliminate the possibility of zero-day attacks due to as-yet unknown optimizer bugs, if the community learns about optimizer bugs that affect the code we've already deployed, it can likely pause the whole system more quickly than the system can be attacked, and then it should be relatively straightforward to upgrade the whole system using contracts compiled differently.

2. Lack of a two-step process for contract ownership changes	
Status: **Partially Resolved**	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-2
Target: `contracts/p1/Main.sol`	

Description

The `Main` contract inherits from OpenZeppelin's `OwnableUpgradeable` contract, which provides a basic access control mechanism. However, to perform contract ownership changes, the `OwnableUpgradeable` contract internally calls the `_transferOwnership()` function, which immediately sets the new owner of the contract. Making such a critical change in a single step is error-prone and can lead to irrevocable mistakes.

```
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Figure 2.1: The `_transferOwnership` function in `OwnableUpgradeable.sol#L7680`

Fix Analysis

This issue has been partially resolved. The Reserve team implemented OpenZeppelin's `AccessControlUpgradeable` contract to allow the existing owner of the `Main` contract to grant the owner role to a new address and the new owner to revoke the role from the previous owner. However, if the new owner chooses not to call `revokeRole` on the previous owner, there will be two owners of the system, which could introduce additional undefined behavior.

3. Unbounded and invalidly bounded system parameters may cause undefined behavior

Status: **Resolved**	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-3
Target: `contracts/p1/BasketHandler.sol`, `contracts/p1/Distributor.sol`	

Description

Many system parameters are unbounded or are bounded incorrectly; this increases the risk of undefined system behavior.

For example, when the `BasketHandler` contract creates the target basket configuration, it does not check that the target weight for each collateral token is bounded between 0 and 1,000 (figure 3.1). According to the documentation, a target weight that is not in this range can cause unexpected reverts.

```
function setPrimeBasket(IERC20[] calldata erc20s, uint192[] calldata targetAmts)
    external
    governance
{
    // withLockable not required: no external calls
    require(erc20s.length == targetAmts.length, "must be same length");
    delete config.erc20s;
    IAssetRegistry reg = main.assetRegistry();
    bytes32[] memory names = new bytes32[](erc20s.length);

    for (uint256 i = 0; i < erc20s.length; ++i) {
        // This is a nice catch to have, but in general it is possible for
        // an ERC20 in the prime basket to have its asset unregistered.
        // In that case the basket is set to disabled.
        require(reg.toAsset(erc20s[i]).isCollateral(), "token is not collateral");

        config.erc20s.push(erc20s[i]);
        config.targetAmts[erc20s[i]] = targetAmts[i];
        names[i] = reg.toColl(erc20s[i]).targetName();
        config.targetNames[erc20s[i]] = names[i];
    }

    emit PrimeBasketSet(erc20s, targetAmts, names);
}
```

Figure 3.1: The `setPrimeBasket` function in `BasketHander.sol#L128-151`

Similarly, the `Distributor` contract expects the revenue share values of the `StRSR` contract's reward pool and the `Furnace` contract to be between 0 and 10,000 (figure 3.2). However, a revenue share value of 0 for the `StRSR` or `Furnace` contract would prevent the payout of rewards to RSR stakers or result in inflation of the RToken supply, respectively.

```
function _setDistribution(address dest, RevenueShare memory share) internal {
    if (dest == FURNACE) require(share.rsrDist == 0, "Furnace must get 0% of RSR");
    if (dest == ST_RSR) require(share.rTokenDist == 0, "StRSR must get 0% of
RToken");
    require(share.rsrDist <= 10000, "RSR distribution too high");
    require(share.rTokenDist <= 10000, "RToken distribution too high");

    destinations.add(dest);
    distribution[dest] = share;
    emit DistributionSet(dest, share.rTokenDist, share.rsrDist);
}
```

Figure 3.2: The `_setDistribution` function in `Distributor.sol#L114-123`

There are many other governance- / owner-set parameters that have undefined bounds:

- `BackingManager.backingBuffer()`
- `BackingManager.tradingDelay()`
- `BackingManager.dustAmount()`
- `BackingManager.maxTradeSlippage()`
- `Broker.auctionLength()`
- `Broker.minBidSize()`
- `Furnace.ratio()`
- `Furnace.period()`
- `RevenueTrader.dustAmount()`
- `RevenueTrader.maxTradeSlippage()`
- `RToken.issuanceRate()`
- `StRSR.rewardRatio()`

Fix Analysis

This issue has been resolved. The Reserve protocol now checks the maximum and / or minimum values of the incorrectly bounded / unbounded system parameters mentioned in

this finding (among others). Any additional system parameters introduced in the future should also be bounded.

4. All auction initiation attempts may fail

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RES-4
Target: `contracts/plugins/trading/GnosisTrade.sol`	

Description

The token auction platform used by the Reserve protocol, the Gnosis Auction platform, may not be provided with a large enough allowance of the token to be auctioned. As of this writing, the fee imposed by the `EasyAuction` contract is set to zero. However, if the owner of the `EasyAuction` contract changes the fee, this issue will immediately pose a severe risk, as it will prevent any auctions from occurring.

The Reserve protocol relies on token auctions to increase the value of an RToken, to increase the amount of rewards paid to RSR stakers, and to recapitalize the system if one or more collateral tokens have defaulted.

The `GnosisTrade` contract initiates auctions by calling the `initiateAuction` function in the `EasyAuction` contract (an external Gnosis-created contract). Before calling `initiateAuction`, the `GnosisTrade` contract approves the `EasyAuction` contract to transfer `sellAmount` of `sell` tokens (figure 4.1). This allows the `initiateAuction` function to transfer the number of `sell` tokens necessary for it to start the auction.

```
function init(
    IBroker broker_,
    address origin_,
    IGnosis gnosis_,
    uint32 auctionLength,
    uint256 minBidSize,
    TradeRequest memory req
) external stateTransition(TradeStatus.NOT_STARTED, TradeStatus.OPEN) {
    [...]
    // == Interactions ==
    IERC20Upgradeable(address(sell)).safeIncreaseAllowance(address(gnosis),
sellAmount);
    auctionId = gnosis.initiateAuction(
        sell,
        buy,
        endTime,
        endTime,
```

```
        uint96(sellAmount),
        uint96(req.minBuyAmount),
        minBidSize,
        req.minBuyAmount, // TODO to double-check this usage of gnosis later
        false,
        address(0),
        new bytes(0)
    );
}
```

Figure 4.1: Part of the `init` function in GnosisTrade.sol#L53-94

However, the number of `sell` tokens that need to be transferred from the `GnosisTrade` contract to the `EasyAuction` contract is actually more than the `sellAmount`. This is because the `EasyAuction` contract takes a fee, which is capped at 1.5%, in exchange for running the auction (figure 4.2). Thus, the token allowance may be insufficient and cause the `initiateAuction` call to revert.

```
function initiateAuction(
    IERC20 _auctioningToken,
    IERC20 _biddingToken,
    uint256 orderCancellationEndDate,
    uint256 auctionEndDate,
    uint96 _auctionedSellAmount,
    uint96 _minBuyAmount,
    uint256 minimumBiddingAmountPerOrder,
    uint256 minFundingThreshold,
    bool isAtomicClosureAllowed,
    address accessManagerContract,
    bytes memory accessManagerContractData
) public returns (uint256) {
    // withdraws sellAmount + fees
    _auctioningToken.safeTransferFrom(
        msg.sender,
        address(this),
        _auctionedSellAmount.mul(FEE_DENOMINATOR.add(feeNumerator)).div(
            FEE_DENOMINATOR
        ) //[0]
    );
    [...]
}
```

Figure 4.2: Part of the `initiateAuction` function in EasyAuction.sol#L152-227

Fix Analysis

This issue has been resolved. The Reserve team updated the arithmetic such that the allowance granted to the `EasyAuction` contract includes the fee required by the Gnosis Auction platform.

5. Per-block issuance limit can be bypassed

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-5
Target: `contracts/p1/RToken.sol`	

Description

Because of a rounding error in a division operation, the per-block issuance limit can be bypassed.

The protocol imposes a delay on large RToken issuances to prevent rapid inflation of the RToken supply and to prevent attacks that can be executed when a collateral token defaults. For example, if a user wishes to mint 1 million RTokens and the maximum issuance per block is 25,000 RTokens, it will take 40 blocks for the issuance to finish.

The `whenFinished` function in the `RToken` contract determines the number of blocks it will take for an issuance to finish. The number of RTokens that need to be minted, `amtRToken`, is divided by the per-block issuance rate, `lastIssRate`. As long as the total of all `amtRToken` values across all issuances in a block is less than `lastIssRate`, the issuance will happen in the same block as the request.

```
function whenFinished(uint256 amtRToken) private returns (uint192 finished) {
    // Calculate the issuance rate (if this is the first issuance in the block)
    if (lastIssRateBlock < block.number) {
        lastIssRateBlock = block.number;
        lastIssRate = uint192((issuanceRate * totalSupply()) / FIX_ONE);
        if (lastIssRate < MIN_ISS_RATE) lastIssRate = MIN_ISS_RATE;
    }

    // Add amtRToken's worth of issuance delay to allVestAt
    uint192 before = allVestAt; // D18{block number}
    uint192 worst = uint192(FIX_ONE * (block.number - 1)); // D18{block number}
    if (worst > before) before = worst;
    finished = before + uint192((FIX_ONE_256 * amtRToken) / lastIssRate);
    allVestAt = finished;
}
```

Figure 5.1: The `whenFinished` function in `RToken.sol#L202-216`

However, because of a rounding error, the total `amtRToken` value can be greater than `lastIssRate`. If `lastIssRate` is 10,000 *whole* tokens (1 whole token is 10^{18} tokens) and the current total is 1 token *less* than 10,000 whole tokens, a user will be able to mint 9,999 tokens without causing `finished` to increase. Then, because `finished` has not increased, the issuance will be atomic, but the total number of minted RTokens will be greater than `lastIssRate`.

Fix Analysis

This issue has been resolved. The Reserve team updated the arithmetic to calculate `finished` to round up instead of down. Thus, the per-block issuance can no longer be greater than `lastIssRate`.

6. All attempts to initiate auctions of defaulted collateral tokens will fail

Status: **Resolved**	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-RES-6
Target: `contracts/p1/mixins/TradingLib.sol`	

Description

When collateral tokens default, they can be sold through an auction in exchange for backup collateral tokens. However, any attempt to initiate such an auction will fail, causing the protocol to remain undercapitalized.

A collateral token defaults when its price falls below a certain target for a set period of time. To make the RToken whole again, the owner of the RToken will "switch" the defaulted collateral token for a backup collateral token by calling the `BasketHandler.refreshBasket` function. The `BackingManager.manageTokens` function will then execute the recapitalization strategy (figure 6.1).

```
function manageTokens(IERC20[] calldata erc20s) external interaction {
    // == Refresh ==
    main.assetRegistry().refresh();

    if (tradesOpen > 0) return;
    // Do not trade when DISABLED or IFFY
    require(main.basketHandler().status() == CollateralStatus.SOUND, "basket not
sound");

    (, uint256 basketTimestamp) = main.basketHandler().lastSet();
    if (block.timestamp < basketTimestamp + tradingDelay) return;

    if (main.basketHandler().fullyCapitalized()) {
        // == Interaction (then return) ==
        handoutExcessAssets(erc20s);
        return;
    } else {
        bool doTrade;
        TradeRequest memory req;
        // 1a
        (doTrade, req) = TradingLibP1.nonRSRTrade(false);
        // 1b
        if (!doTrade) (doTrade, req) = TradingLibP1.rsrTrade();
        // 2
```

```
        if (!doTrade) (doTrade, req) = TradingLibP1.nonRSRTrade(true);
        // 3
        if (!doTrade) {
            compromiseBasketsNeeded();
            return;
        }
        // == Interaction ==
        if (doTrade) tryTrade(req);
    }
}
```

Figure 6.1: The manageTokens function in BackingManager.sol#L49-105

The first recapitalization strategy involves a call to the nonRSRTrade function in the TradingLib library. The function will create a TradeRequest object and use it to initiate an auction in which the defaulted collateral will be sold in exchange for as much backup collateral as possible on the open market (figure 6.2).

```
function nonRSRTrade(bool useFallenTarget)
    external
    view
    returns (bool doTrade, TradeRequest memory req)
{
    (
        IAsset surplus,
        ICollateral deficit,
        uint192 surplusAmount,
        uint192 deficitAmount
    ) = largestSurplusAndDeficit(useFallenTarget);

    if (address(surplus) == address(0) || address(deficit) == address(0)) return
(false, req);

    // Of primary concern here is whether we can trust the prices for the assets
    // we are selling. If we cannot, then we should ignore `maxTradeSlippage`.

    if (
        surplus.isCollateral() &&
        assetRegistry().toColl(surplus.erc20()).status() ==
CollateralStatus.DISABLED
    ) {
        (doTrade, req) = prepareTradeSell(surplus, deficit, surplusAmount);
        req.minBuyAmount = 0;
    } else {
        [...]
    }
    [...]
    return (doTrade, req);
}
```

Figure 6.2: Part of the nonRSRTrade function in TradingLib.sol#L229-264

However, the `EasyAuction` contract, which is used downstream to manage auctions, will prevent the creation of the auction. This is because `req.minBuyAmount` is set to zero in `nonRSRTrade`, while `EasyAuction` requires that `_minBuyAmount` be greater than zero (figure 6.3).

```
function initiateAuction(
    IERC20 _auctioningToken,
    IERC20 _biddingToken,
    uint256 orderCancellationEndDate,
    uint256 auctionEndDate,
    uint96 _auctionedSellAmount,
    uint96 _minBuyAmount,
    uint256 minimumBiddingAmountPerOrder,
    uint256 minFundingThreshold,
    bool isAtomicClosureAllowed,
    address accessManagerContract,
    bytes memory accessManagerContractData
) public returns (uint256) {
    [...]
    require(_minBuyAmount > 0, "tokens cannot be auctioned for free");
    [...]
}
```

Figure 6.3: Part of the `initiateAuction` function in `EasyAuction.sol#L152-227`

While there exist other options for recapitalizing the system, such as seizing RSR tokens and compromising the number of basket units (BUs) backing the RToken, the system will always try the above approach first. However, the transaction will always revert, and the system will not try any of the other options.

Fix Analysis

This issue has been resolved. The Reserve team updated the `GnosisTrade` contract to ensure that at least 1 quantum token will be bought back as part of each defaulted collateral auction. This minimum purchase will satisfy the requirement in the `EasyAuction.initiateAuction` function.

7. Fallen-target auctions can be prevented from occurring

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-7
Target: `contracts/p1/mixins/TradingLib.sol`	

Description

By sending enough RSR tokens to the `BackingManager` contract, a user could prevent the contract from considering any other asset to be in surplus; when there is no asset in surplus, the protocol is unable to hold a fallen-target token auction. However, this would be possible only if an attempt to seize RSR failed or if the Reserve team changed the recapitalization strategy to attempt a fallen-target auction before an RSR seizure.

A fallen-target token auction is the protocol's second-to-last resort when it is attempting to recapitalize the system without directly compromising the number of underlying BUs. The `TradingLib` library's `largestSurplusAndDeficit` function determines which asset should be sold (i.e., which asset is the most in surplus) and which asset should be bought (i.e., which has the largest deficit). In this function, the `basketTop` value represents the BU threshold that determines whether an asset is in surplus. If a token has a balance sufficient to represent at least `basketTop` BUs, it is in surplus (figure 7.1).

```
function largestSurplusAndDeficit(bool useFallenTarget)
    public
    view
    returns (
        IAsset surplus,
        ICollateral deficit,
        uint192 sellAmount,
        uint192 buyAmount
    )
{
    IERC20[] memory erc20s = assetRegistry().erc20s();

    // Compute basketTop and basketBottom
    // basketTop is the lowest number of BUs to which we'll try to sell surplus
assets
    // basketBottom is the greatest number of BUs to which we'll try to buy deficit
assets
    uint192 basketTop = rToken().basketsNeeded(); // {BU}
    uint192 basketBottom = basketTop; // {BU}
```

```
    if (useFallenTarget) {
        uint192 tradeVolume; // {UoA}
        uint192 totalValue; // {UoA}
        for (uint256 i = 0; i < erc20s.length; ++i) {
            IAsset asset = assetRegistry().toAsset(erc20s[i]);

            // Ignore dust amounts for assets not in the basket
            uint192 bal = asset.bal(address(this)); // {tok}
            if (basket().quantity(erc20s[i]).gt(FIX_ZERO) ||
bal.gt(dustThreshold(asset))) {
                // {UoA} = {UoA} + {UoA/tok} * {tok}
                totalValue = totalValue.plus(asset.price().mul(bal, FLOOR));
            }
        }
        basketTop = totalValue.div(basket().price(), CEIL);
        [...]
    }

    uint192 maxSurplus; // {UoA}
    uint192 maxDeficit; // {UoA}

    for (uint256 i = 0; i < erc20s.length; ++i) {
        if (erc20s[i] == rsr()) continue; // do not consider RSR

        IAsset asset = assetRegistry().toAsset(erc20s[i]);
        uint192 bal = asset.bal(address(this));

        // Token Threshold - top
        uint192 tokenThreshold = basketTop.mul(basket().quantity(erc20s[i]), CEIL);
// {tok};
        if (bal.gt(tokenThreshold)) {
            // {UoA} = ({tok} - {tok}) * {UoA/tok}
            uint192 deltaTop = bal.minus(tokenThreshold).mul(asset.price(), FLOOR);
            if (deltaTop.gt(maxSurplus)) {
                surplus = asset;
                maxSurplus = deltaTop;

                // {tok} = {UoA} / {UoA/tok}
                sellAmount = maxSurplus.div(surplus.price());
                if (bal.lt(sellAmount)) sellAmount = bal;
            }
        } else {
            [...]
                }
            }
        }
    }
}
```

Figure 7.1: The `largestSurplusAndDeficit` function in `TradingLib.sol#L102-192`

However, the `basketTop` value can be artificially manipulated because it incorporates the value of *all* assets, including the RSR token, but there is no check of whether the RSR token is the one most in surplus. Thus, if the RSR balance is large enough, the value of `basketTop` can increase such that none of the other assets or collateral held by the `BackingManager` contract (including the RToken) will have a balance large enough to represent `basketTop` BUs. Then, without an asset in surplus, it will be impossible to launch a fallen-target token auction; instead, the `BackingManager` contract will call the `compromiseBasketsNeeded` function to compromise the number of BUs held by the contract, effectively decreasing the value of the RToken.

Fix Analysis

This issue has been resolved. The Reserve team has updated the recapitalization strategy, including by removing the fallen-target auctions, allowing the RSR token to be the asset most in surplus, and allowing the RSR token to be sold in token auctions. Thus, airdrops of the RSR token to either the `BackingManager` or `StRSR` contract are now beneficial to the protocol. The other significant changes made to the recapitalization strategy were not assessed during the fix review.

8. Faulty RToken issuance-cancellation process

Status: **Resolved**	
Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-RES-8
Target: `contracts/p1/RToken.sol`	

Description

Because of incorrect data validation in the issuance-cancellation process, users who wish to cancel *some* of their issuance requests may have *all* of their issuances canceled.

To cancel issuance requests, users call the `cancel` function in the `RToken` contract. The `cancel` function takes two arguments: `endId` and `earliest`. The `earliest` variable indicates whether the user wants to cancel early issuances or later issuances. If `earliest` is `true`, the `endId` variable indicates the index of the *last* issuance to be canceled. If `earliest` is `false`, `endId` indicates the index of the *first* issuance to be canceled (figure 8.1).

```
function cancel(uint256 endId, bool earliest) external interaction {
    address account = _msgSender();
    IssueQueue storage queue = issueQueues[account];

    require(queue.left <= endId && endId <= queue.right, "'endId' is out of range");

    // == Interactions ==
    if (earliest) {
        refundSpan(account, queue.left, endId);
    } else {
        refundSpan(account, endId, queue.right);
    }
}
```

Figure 8.1: The `cancel` function in RToken.sol#L273-285

The `cancel` function internally calls the `refundSpan` function. The `refundSpan` function refunds the user for all issuances in the range of indexes `[left, right)` (figure 8.2).

```
function refundSpan(
    address account,
    uint256 left,
    uint256 right
```

```
) private {
    if (left >= right) return; // refund an empty span

    IssueQueue storage queue = issueQueues[account];

    // compute total deposits to refund
    uint256 tokensLen = queue.tokens.length;
    uint256[] memory amt = new uint256[](tokensLen);
    IssueItem storage rightItem = queue.items[right - 1];

    // we could dedup this logic but it would take more SLOADS, so I think this is
best
    if (queue.left == 0) {
        for (uint256 i = 0; i < tokensLen; ++i) {
            amt[i] = rightItem.deposits[i];
        }
    } else {
        IssueItem storage leftItem = queue.items[queue.left - 1];
        for (uint256 i = 0; i < tokensLen; ++i) {
            amt[i] = rightItem.deposits[i] - leftItem.deposits[i];
        }
    }
    [...]
}
```

Figure 8.2: Part of the `refundSpan` function in RToken.sol#L401-445

However, instead of using the `left` input variable, `refundSpan` uses `queue.left`, which is the first issuance in the `queue` array that *can be* canceled. On the other hand, when `earliest` is `false`, `endId` (i.e., `left`) points to the first issuance that the user *wants to* cancel. Note that `left` does *not* have to be equal to `queue.left`; `left` could instead be greater than `queue.left`. Thus, a user who wishes to cancel issuance requests in the range of `[left, right)` may actually be refunded for those in the range of `[queue.left, right)`, where `left` is greater than `queue.left`.

Fix Analysis

This issue has been resolved. All uses of `queue.left` in the `refundSpan` function have been replaced with `left`, which ensures that only the pending issuances in the range of `[left, right)` will be canceled.

9. Token auctions may not cover entire collateral token deficits

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-9
Target: `contracts/p1/mixins/TradingLib.sol`	

Description

Because of incorrect data validation, token auctions initiated to cover a specific deficit may not always cover the entire deficit amount.

When the system is not fully capitalized (i.e., the number of BUs held by the `BackingManager` contract is less than the number required by the `RToken` contract), the `BackingManager` will try to identify the token most in surplus and the collateral token with the largest deficit. If the former token is not a collateral token or is a collateral token that has not defaulted, the `prepareTradeToCoverDeficit` function will be called (figure 9.1). This function tries to compute the number of `sell` tokens that need to be sold to cover a fixed number of `buy` tokens (i.e., the deficit). By contrast, the `prepareTradeSell` function tries to compute the number of `buy` tokens that can be bought given a fixed number of `sell` tokens.

```
function nonRSRTrade(bool useFallenTarget)
    external
    view
    returns (bool doTrade, TradeRequest memory req)
{
    (
        IAsset surplus,
        ICollateral deficit,
        uint192 surplusAmount,
        uint192 deficitAmount
    ) = largestSurplusAndDeficit(useFallenTarget);

    if (address(surplus) == address(0) || address(deficit) == address(0)) return
(false, req);

    // Of primary concern here is whether we can trust the prices for the assets
    // we are selling. If we cannot, then we should ignore `maxTradeSlippage`.

    if (
        surplus.isCollateral() &&
```

```
        assetRegistry().toColl(surplus.erc20()).status() ==
CollateralStatus.DISABLED
    ) {
        (doTrade, req) = prepareTradeSell(surplus, deficit, surplusAmount);
        req.minBuyAmount = 0;
    } else {
        (doTrade, req) = prepareTradeToCoverDeficit(
            surplus,
            deficit,
            surplusAmount,
            deficitAmount
        );
    }

    if (req.sellAmount == 0) return (false, req);

    return (doTrade, req);
}
```

Figure 9.1: The nonRSRTrade function in TradingLib.sol#L229-264

After identifying the `sellAmount` value that should be used to cover the fixed `deficitAmount`, the `prepareTradeToCoverDeficit` function calls `prepareTradeSell`. However, the `minBuyAmount` value returned by the `prepareTradeSell` function may be less than `deficitAmount`. This can occur if `sellAmount` is greater than the maximum trading volume of the asset, an owner-set parameter. In such a case, `sellAmount` will decrease, as will `minBuyAmount` (figure 9.2). If `minBuyAmount` is less than `deficitAmount`, another auction will have to take place to cover the deficit.

```
function prepareTradeSell(
    IAsset sell,
    IAsset buy,
    uint192 sellAmount
) public view returns (bool notDust, TradeRequest memory trade) {
    trade.sell = sell;
    trade.buy = buy;
    [...]
    uint192 s = fixMin(sellAmount, sell.maxTradeVolume().div(sell.price(), FLOOR));
    trade.sellAmount = s.shiftl_toUint(int8(sell.erc20().decimals()), FLOOR);
    [...]
    // {buyTok} = {sellTok} * {UoA/sellTok} / {UoA/buyTok}
    uint192 b = s.mul(FIX_ONE.minus(maxTradeSlippage())).mulDiv(
        sell.price(),
        buy.price(),
        CEIL
    );
    trade.minBuyAmount = b.shiftl_toUint(int8(buy.erc20().decimals()), CEIL);
    [...]
    return (true, trade);
}
function prepareTradeToCoverDeficit(
```

```
    IAsset sell,
    IAsset buy,
    uint192 maxSellAmount,
    uint192 deficitAmount
) public view returns (bool notDust, TradeRequest memory trade) {
    [...]
    deficitAmount = fixMax(deficitAmount, dustThreshold(buy));

    // {sellTok} = {buyTok} * {UoA/buyTok} / {UoA/sellTok}
    uint192 exactSellAmount = deficitAmount.mulDiv(buy.price(), sell.price(), CEIL);
    [...]
    uint192 slippedSellAmount =
exactSellAmount.div(FIX_ONE.minus(maxTradeSlippage()), CEIL);

    uint192 sellAmount = fixMin(slippedSellAmount, maxSellAmount);
    return prepareTradeSell(sell, buy, sellAmount);
}
```

Figure 9.2: Parts of the `prepareTradeSell` and `prepareTradeToCoverDeficit` functions in TradingLib.sol#L26-94

Fix Analysis

This issue has been resolved. The Reserve team indicated that this edge case is the system's expected behavior. Additional context provided by the team is included verbatim below.

> This issue reports that `prepareTradeToCoverDeficit()` can return a `minBuyAmount` that is less than the requested `deficitAmount`, if the `sellAmount` is greater than the configured maximum trade volume for that asset.
>
> This is the intended behavior of these functions. If the deficit to be covered is larger than the maximum trade volume, the auction should be run at the maximum trade volume. After that auction completes, another trade to cover the reduced deficit may begin on the next call to `BackingManager.manageTokens`, and this is expected to repeat until the deficit is fully covered.

10. Inability to validate the recency of Aave and Compound oracle data

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-10
Target: `contracts/plugins/assets/abstract/AaveOracleMixin.sol`, `contracts/plugins/assets/abstract/CompoundOracleMixin.sol`	

Description

The Aave and Compound oracle systems do not provide timestamps or round data. Thus, the Reserve protocol cannot validate the recency of the pricing data they provide.

The Reserve protocol obtains pricing data for collateral and RSR tokens from the Aave and Compound oracle systems. Each oracle system relies on a Chainlink price feed as its underlying data feed. However, unlike Chainlink, Aave and Compound do not provide information on when a price was last updated. Thus, the pricing data reported to the Reserve protocol could be stale or invalid, exposing the protocol to risk. Additionally, in extreme market conditions, Chainlink may pause its oracle systems, which can increase the risk of undefined behavior.

Fix Analysis

This issue has been resolved. The Reserve protocol now uses Chainlink's `AggregatorV3Interface` as its primary source of pricing data and includes checks of the recency of that data. We did not evaluate the newly introduced asset types or the use of the Chainlink oracle to obtain pricing data for those new types.

11. An RSR seizure could leave the StRSR contract unusable

Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-11
Target: `contracts/p1/StRSR.sol`	

Description

The seizure of all of the RSR in the staking pool could leave the system in a state that prevents stakers from unstaking.

RSR seizures occur when the system is undercapitalized. RSR tokens are seized from the `StRSR` contract and sent to the `BackingManager` contract. The contract initiates an auction of the tokens, which enables the protocol to buy back a portion of the collateral token that is causing the deficit.

This action is triggered via the `seizeRSR` function in the `StRSR` contract. The tokens that need to be seized (the quantity represented by `rsrAmount`) are taken evenly from the stake pool, the draft pool, and the reward pool. The `stakeRSR` value is the amount of RSR tokens backing current stakes (i.e., the stake pool), `draftRSR` is the amount of RSR tokens reserved for withdrawals (i.e., the draft pool), and `rewards` is the balance of the RSR tokens that are in neither the stake nor draft pool (figure 11.1).

```
function seizeRSR(uint256 rsrAmount) external notPaused {
    require(_msgSender() == address(main.backingManager()), "not backing manager");
    require(rsrAmount > 0, "Amount cannot be zero");
    uint192 initRate = stakeRate;

    uint256 rsrBalance = main.rsr().balanceOf(address(this));
    require(rsrAmount <= rsrBalance, "Cannot seize more RSR than we hold");
    if (rsrBalance == 0) return;

    // Calculate dust RSR threshold, the point at which we might as well call it a
wipeout
    uint256 dustRSRAmt = (MIN_EXCHANGE_RATE * (totalDrafts + totalStakes)) /
FIX_ONE; // {qRSR}
    uint256 seizedRSR;
    if (rsrBalance <= rsrAmount + dustRSRAmt) {
        // Rebase event: total RSR stake wipeout
        seizedRSR = rsrBalance;
        beginEra();
```

```
    } else {
        uint256 rewards = rsrRewards();

        // Remove RSR evenly from stakeRSR, draftRSR, and the reward pool
        uint256 stakeRSRToTake = (stakeRSR * rsrAmount + (rsrBalance - 1)) /
rsrBalance;
        stakeRSR -= stakeRSRToTake;
        seizedRSR = stakeRSRToTake;
        stakeRate = stakeRSR == 0 ? FIX_ONE : uint192((FIX_ONE_256 * totalStakes) /
stakeRSR);

        uint256 draftRSRToTake = (draftRSR * rsrAmount + (rsrBalance - 1)) /
rsrBalance;
        draftRSR -= draftRSRToTake;
        seizedRSR += draftRSRToTake;
        draftRate = draftRSR == 0 ? FIX_ONE : uint192((FIX_ONE_256 * totalDrafts) /
draftRSR);

        // Removing from unpaid rewards is implicit
        seizedRSR += (rewards * rsrAmount + (rsrBalance - 1)) / rsrBalance;
    }

    // Transfer RSR to caller
    emit ExchangeRateSet(initRate, stakeRate);
    exchangeRateHistory.push(HistoricalExchangeRate(uint32(block.number),
stakeRate));
    IERC20Upgradeable(address(main.rsr())).safeTransfer(_msgSender(), seizedRSR);
}
```

Figure 11.1: The `seizeRSR` function in StRSR.sol#L244-282

If `stakeRSR` is set to zero in the `else` closure highlighted in figure 11.1, the `beginEra` function should be called. The `beginEra` function allows the system to reset a staking pool that has experienced a significant token seizure. However, `beginEra` is not called; thus, `totalStakes` will be a non-zero value and `stakeRate` will be `FIX_ONE`, but `stakeRSR` will be zero. When users try to unstake their tokens by calling the `unstake` function, the calls will revert (figure 11.2).

```
function unstake(uint256 stakeAmount) external notPaused {
    address account = _msgSender();
    require(stakeAmount > 0, "Cannot withdraw zero");
    require(stakes[era][account] >= stakeAmount, "Not enough balance");

    _payoutRewards();

    // ==== Compute changes to stakes and RSR accounting
    // rsrAmount: how many RSR to move from the stake pool to the draft pool
    // pick rsrAmount as big as we can such that (newTotalStakes <= newStakeRSR *
stakeRate)
    _burn(account, stakeAmount);
```

```
    // {qRSR} = D18 * {qStRSR} / D18{qStRSR/qRSR}
    uint256 newStakeRSR = (FIX_ONE_256 * totalStakes) / stakeRate;
    uint256 rsrAmount = stakeRSR - newStakeRSR;
    stakeRSR = newStakeRSR;

    // Create draft
    (uint256 index, uint64 availableAt) = pushDraft(account, rsrAmount);
    emit UnstakingStarted(index, era, account, rsrAmount, stakeAmount, availableAt);
}
```

Figure 11.2: The `unstake` function in StRSR.sol#L178-198

This edge case can occur only if `rsrBalance`, `stakeRSR`, and `rsrAmount` are approximately equal, but `rsrBalance` is just large enough to be greater than the sum of `rsrAmount` and `dustRSRAmt`, causing `seizeRSR` to enter the `else` closure. At that point, the only way to recover the system is for someone to stake enough RSR to render the `stakeRSR` value equal to `totalStakes`. Then everyone who wishes to exit the system will be able to do so.

Fix Analysis

This issue has been resolved. The `StRSR` contract now begins a new era if `stakeRSR` is set to zero in the `else` closure, which mitigates the edge case. Additionally, if `draftRSR` is set to zero in the `else` closure, a new *draft* era begins. The creation of these discrete eras enables stakers to use any remaining liquidity to exit the system.

12. System owner has excessive privileges

Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-RES-12
Target: `contracts/mixins/ComponentRegistry.sol`	

Description

The owner of the `Main` contract has excessive privileges, which puts the entire system at risk. The owner of the `Main` contract is effectively the owner of all periphery contracts and thus the entire system.

An exhaustive list of the system owner's privileges is provided in appendix D. These privileges include changing numerous system parameters. For example, at any time, the owner can change the address of any of the following contracts by calling the functions shown in figure 12.1 (which are defined in the `ComponentRegistry` abstract contract from which `Main` inherits):

- `RToken`
- `StRSR`
- `AssetRegistry`
- `BasketHandler`
- `BackingManager`
- `Distributor`
- The `RevenueTrader` of the `RToken` and RSR token
- `Furnace`
- `Broker`

```
IRToken public rToken;

function setRToken(IRToken val) public onlyOwner {
    emit RTokenSet(rToken, val);
```

```
    rToken = val;
}

IStRSR public stRSR;

function setStRSR(IStRSR val) public onlyOwner {
    emit StRSRSet(stRSR, val);
    stRSR = val;
}

[...]

IBackingManager public backingManager;

function setBackingManager(IBackingManager val) public onlyOwner {
    emit BackingManagerSet(backingManager, val);
    backingManager = val;
}
[...]
```

Figure 12.1: Critical address–changing functions in `ComponentRegistry.sol#L30-98`

The owner's ability to change so many critical components of the system architecture creates a single point of failure. It increases the likelihood that the `Main` contract's owner will be targeted by an attacker and increases the incentives for the owner to act maliciously.

The RToken(s) deployed by the Reserve team will be owned by on-chain governance. However, malicious parties with significant stakes may be able to swing proposals in their favor. Alternatively, because the process of creating an RToken is permissionless, the owner of an RToken could also be a single private key.

Fix Analysis

This issue has been resolved. The owner of the system is no longer able to update critical contract addresses after the system's initialization.

13. Lack of zero address checks in Deployer constructor

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-RES-13
Target: `contracts/p1/Deployer.sol`	

Description

The `Deployer` contract's constructor initializes 19 contract addresses but does not check any of those addresses against the zero address. (Note that 13 of the addresses are contained in the `implementations_` input variable object shown in figure 13.1.) If the deployer of the `Deployer` contract accidentally set any of those addresses to the zero address, any RToken deployed through the contract would exhibit undefined behavior.

```
constructor(
    IERC20Metadata rsr_,
    IERC20Metadata comp_,
    IERC20Metadata aave_,
    IGnosis gnosis_,
    IComptroller comptroller_,
    IAaveLendingPool aaveLendingPool_,
    Implementations memory implementations_
) {
    rsr = rsr_;
    comp = comp_;
    aave = aave_;
    gnosis = gnosis_;
    comptroller = comptroller_;
    aaveLendingPool = aaveLendingPool_;
    implementations = implementations_;
}
```

Figure 13.1: The constructor in `Deployer.sol#L41-57`

Fix Analysis

This issue has been resolved. Of the 19 addresses mentioned in this finding, 16 addresses are now checked against the zero address; the other three (the `comp_`, `aave_`, and `comptroller_` addresses) are no longer provided to the constructor.

14. RTokens can be purchased at a discount

Status: **Resolved**	
Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-RES-14
Target: `contracts/p1/RToken.sol`	

Description

If the collateral in a basket has decreased in price but has not defaulted, users may be able to purchase RTokens at a discount by creating large issuance requests.

The status of a collateral token can be either `SOUND`, `IFFY`, or `DISABLED`. If the status of a collateral token is `SOUND`, the price is within a stable range; if it is `IFFY`, the price has deviated from the range; and if it is `DISABLED`, the price has deviated from the range for an extended period of time.

When a user makes an issuance request, the `RToken.issue` function checks whether any collateral in the basket is `DISABLED`. If none is, the user will be issued RTokens atomically or over some number of blocks (figure 14.1). If the issuance is non-atomic, all collateral must be `SOUND` for the vesting to finish.

```
function issue(uint256 amtRToken) external interaction {
    require(amtRToken > 0, "Cannot issue zero");

    // == Refresh ==
    main.assetRegistry().refresh();

    address issuer = _msgSender(); // OK to save: it can't be changed in reentrant
runs
    IBasketHandler bh = main.basketHandler(); // OK to save: can only be changed by
gov

    (uint256 basketNonce, ) = bh.lastSet();
    IssueQueue storage queue = issueQueues[issuer];

    [...]

    // == Checks-effects block ==
    CollateralStatus status = bh.status();
    require(status != CollateralStatus.DISABLED, "basket disabled");
```

```
    main.furnace().melt();

    // ==== Compute and accept collateral ====
    // D18{BU} = D18{BU} * {qRTok} / {qRTok}
    uint192 amtBaskets = uint192(
        totalSupply() > 0 ? mulDiv256(basketsNeeded, amtRToken, totalSupply()) :
amtRToken
    );

    (address[] memory erc20s, uint256[] memory deposits) = bh.quote(amtBaskets,
CEIL);

    // Add amtRToken's worth of issuance delay to allVestAt
    uint192 vestingEnd = whenFinished(amtRToken); // D18{block number}

    // Bypass queue entirely if the issuance can fit in this block and nothing
blocking
    if (
        vestingEnd <= FIX_ONE_256 * block.number &&
        queue.left == queue.right &&
        status == CollateralStatus.SOUND
    ) {
        // Complete issuance
        _mint(issuer, amtRToken);
        uint192 newBasketsNeeded = basketsNeeded + amtBaskets;
        emit BasketsNeededChanged(basketsNeeded, newBasketsNeeded);
        basketsNeeded = newBasketsNeeded;

        // Note: We don't need to update the prev queue entry because queue.left =
queue.right
        emit Issuance(issuer, amtRToken, amtBaskets);

        address backingMgr = address(main.backingManager());

        // == Interactions then return: transfer tokens ==
        for (uint256 i = 0; i < erc20s.length; ++i) {
            IERC20Upgradeable(erc20s[i]).safeTransferFrom(issuer, backingMgr,
deposits[i]);
        }
        return;
    }
    [...]
    // == Interactions: accept collateral ==
    for (uint256 i = 0; i < erc20s.length; ++i) {
        IERC20Upgradeable(erc20s[i]).safeTransferFrom(issuer, address(this),
deposits[i]);
    }
}
```

Figure 14.1: The `issue` function in RToken.sol#L95-198

However, if a user makes an issuance request when the status of a collateral token is IFFY (because of a price decrease), the user will pay a discounted price for the RTokens. This is

because the user sends the cheap collateral immediately, during the call to `RToken.issue`, but the RToken vesting will occur later, after the price of the collateral has stabilized and its status has changed back to `SOUND`.

Fix Analysis

This issue has been resolved. The Reserve team has updated the `RToken` contract to allow issuances only when the status of the basket is `SOUND`.

15. Inconsistent use of the FixLib library

Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-RES-15
Target: `contracts/`	

Description

According to the protocol specification for arithmetic operations, all computations performed on `uint192` values, which represent `uint192x18` variables, should use the `FixLib` library. However, numerous computations in the codebase do not comply with the guidance in the specification.

The protocol specification has the following general guidelines on using the `FixLib` library for `uint192` values:

- Never allow a `uint192` to be implicitly upcast to a `uint256` without a comment explaining what is happening and why.
- Never explicitly cast between `uint192` and `uint256` values without doing the appropriate numeric conversion (e.g., `toUint()` or `toFix()`).
- Use standard arithmetic operations on `uint192` values only if you are gas-optimizing a hotspot in `p1` and need to remove `FixLib` calls (and leave inline comments explaining what you are doing and why).

The codebase does not consistently follow these guidelines. Figure 15.1 shows an instance in which a `uint256` is downcast to a `uint192` without an explicit comment.

```
// ==== Compute and accept collateral ====
// D18{BU} = D18{BU} * {qRTok} / {qRTok}
uint192 amtBaskets = uint192(
    totalSupply() > 0 ? mulDiv256(basketsNeeded, amtRToken, totalSupply()) :
amtRToken
);
```

Figure 15.1: Part of the `issue` function in RToken.sol#L127-131

In the code in figure 15.2, standard arithmetic operations are performed on `uint192` values, but there are no inline comments indicating whether the operations are meant to optimize gas.

```
// ==== Compute and accept collateral ====
// Paying out the ratio r, N times, equals paying out the ratio (1 - (1-r)^N) 1
time.
// Apply payout to RSR backing
uint192 payoutRatio = FIX_ONE - FixLib.powu(FIX_ONE - rewardRatio, numPeriods);
```

Figure 15.2: Part of the `_payoutRewards` function in `StRSR.sol#L334-336`

Failure to use the `FixLib` library for operations on `uint192` values and to comply with the protocol specification can lead to undefined system behavior.

Fix Analysis

This issue has been resolved. The inline documentation now justifies the cases in which the `FixLib` library is not used for arithmetic operations on `uint192` values.

A. 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.

B. 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.