Reserve

Security Assessment

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 and Felipe Manzano**

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.

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 timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and relevant documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are 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

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-RES-4**
  The insufficient token allowance provided to the `EasyAuction` contract could prevent the initiation of any token auctions.

- **TOB-RES-6**
  Incorrect data manipulation prevents the initiation of auctions of defaulted collateral tokens.

- **TOB-RES-11**
  If `stakeRSR` is set to zero after a call to the `StRSR` contract's `seizeRSR` function, stakers who wish to exit the system may be unable to do so.

- **TOB-RES-12**
  The owner of the `Main` contract has excessive privileges, which creates a single point of failure within the system.

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 engineers were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Felipe Manzano, Consultant
felipe@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
August 11, 2022	Delivery of fix review report

Project Goals

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

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place for user and admin operations?
- Are the Reserve protocol state machine transitions correct?
- Are the data validation, arithmetic, and unit conversion operations performed on token balances, ratios, percentages, and arrays handled correctly?
- Does the recapitalization mechanism introduce edge cases or increase the likelihood of undefined behavior?
- Does the ability to provide certain parameters enable users to engage in malicious behavior?
- Do the external contract interactions introduce reentrancy attack risks?
- Is the Gnosis Auction platform used correctly, and does it introduce any additional risks?

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

Project Coverage

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

- **Recapitalization strategy.** We reviewed the recapitalization strategy used by the Reserve protocol when a collateral token defaults or the underlying reference basket is switched. This analysis led to the discovery of the issue in TOB-RES-6, which prevents the initiation of auctions of defaulted collateral tokens. Additionally, a malicious user could send RSR tokens to the `BackingManager` contract to prevent fallen-target auctions from occurring (TOB-RES-7). Finally, we discovered that because of the thresholds set by the system's owner, a single token auction may not be sufficient to completely clear a collateral token deficit (TOB-RES-9).

- **`seizeRSR()` arithmetic.** We reviewed the arithmetic involved in the seizure of RSR tokens and looked for related edge cases. In this review, we found that if `stakeRSR` is set to zero, users may be unable to exit the system (TOB-RES-11).

- **Gnosis Auction platform.** We reviewed the use of the `EasyAuction` contract to set up token auctions. In this review, we found that the token approval granted to the contract may be insufficient, which could prevent it from initiating any auctions (TOB-RES-4).

- **`RToken` arithmetic and data validation.** We analyzed the arithmetic, data validation, and array manipulations performed in the `RToken` contract. This led us to discover that RTokens can be purchased for a low cost if a collateral token enters the `IFFY` state (TOB-RES-14). Additionally, because of the incorrect use of function parameters, users' attempts to cancel their issuance requests may lead to faulty behavior (TOB-RES-8).

- **`StRSR` arithmetic and data validation.** We analyzed the arithmetic, data validation, and array manipulations performed in the `StRSR` abstract contract. Analysis of the changes made to balances, ratios, and internal state variables did not yield any findings.

- **Access controls.** In reviewing the system access controls, we found that excessive privileges are provided to the owner of the system (TOB-RES-12). However, we did not identify any ways to bypass the access controls.

- **Privileged user–related data validation.** We also reviewed the validation of parameters provided by privileged users and identified insufficient zero address and system parameter checks (TOB-RES-13 and TOB-RES-3).

- **Oracle systems.** Analysis of the interactions with the Aave and Compound oracle systems led to the discovery of TOB-RES-10, which concerns the risks associated with relying on data provided by those systems.
- **Arithmetic bit shifts.** A review of the arithmetic conversions between whole and quantum tokens did not yield any findings.
- **Token arithmetic and conversions.** A review of the unit conversions and arithmetic operations, including those involving whole tokens, quantum tokens, unit-of-account (UoA) values, token prices, and basket units, did not yield any findings.
- **Front-running resistance.** We reviewed the system for any flaws that could enable front-running of an oracle update. While we found that an attacker could back-run an oracle update to purchase RTokens at a discount (TOB-RES-14), we did not identify any ways to front-run contract initializations, RSR seizures, or basket switches.
- **Compliance with the Universal Upgradeable Proxy Standard (UUPS).** A review of the system's compliance with the OpenZeppelin UUPS did not yield any findings.
- **External contract interactions.** A review of the interactions with external contracts, including token transfers, did not identify any reentrancy attack risks or other flaws.
- **Internal token transfers.** A review of the internal token transfers and the reward-claiming mechanism did not yield any findings.
- **Use of micro-percentages in `Governance`.** Analysis of the use of micro-percentages instead of token quantities in the `Governance` contract did not yield any findings.
- **Use of eras in `StRSRVotes`.** We analyzed the `StRSRVotes` contract's deviations from the OpenZeppelin `ERC20VotesUpgradeable` contract to identify any edge cases or undefined behavior; no such issues were detected.
- **`Pausable`.** We reviewed the `Pausable` contract for any state transitions that could lead to vulnerabilities or a bypass of the access controls; this review did not yield any findings.
- **C3 linearization.** A *best-effort* review of the system's use of C3 linearization did not identify any issues.

- **Dynamic testing of `FixLib`.** A *best-effort* automated review of the `FixLib` library's system properties did not yield any findings. The system properties that were tested are outlined in the Automated Testing section.

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:

- **Manual review of `FixLib`.** The `FixLib` library is the Reserve protocol's implementation of a fixed-point library for `uint192x18` values. Fixed-point arithmetic is used across all of the contracts and is crucial to the correctness of the system behavior. We did not perform a manual review of this library.

- **Facade and FacadeP1.** The `Facade` and `FacadeP1` contracts enable permissionless non-governance interactions with the Reserve protocol. Because these contracts do not perform any state changes and are mainly used by the user interface to retrieve information about the current state of the system, the contracts were not reviewed in this audit.

- **`p0` contracts.** The `contracts/` folder in the repository contains two release candidates: `p0`, which is the reference implementation, and `p1`, which is the candidate for production release. We did not review any of the contracts in the `p0/` folder.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Test Results

The results of this testing are detailed below.

`Fixed.sol`. The `Fixed.sol` file holds the `FixLib` library, which is used to perform arithmetic operations on `uint192x18` values. We used Echidna to test the library's system properties.

Property	Tool	Result
`test_fullMul`: The result of a call to `fullMul()` should match that of a custom reference implementation of the same function.	Echidna	**Passed**
`test_mul_commutative`: The `mul()` function maintains the commutative property of multiplication.	Echidna	**Passed**
`test_minus_plus_symmetric`: If one value is subtracted from another, re-adding it should result in the original value.	Echidna	**Passed**
`test_powu_pow0`: If `powu()` is called with any value raised to the power of zero, it should return `FIX_ONE` and should never revert.	Echidna	**Passed**
`test_powu_pow1`: If `powu()` is called with any value raised to the power of `uint256(1)`, it should return 0 and should never revert.	Echidna	**Passed**

`test_powu_pow_neutral`: If `powu()` is called with `FIX_ONE` raised to the power of any value, it should return `FIX_ONE` and should never revert.	Echidna	**Passed**
`test_shiftl_rnd`: When called, the `shiftlRnd` function should return `x` shifted `decimals` places in the correct direction and appropriately rounded.	Echidna	**Passed**
`test_shiftl_toFix_rnd`: Similarly to `test_shiftl_rnd`, `test_shiftl_toFix_rnd` should return `x` shifted `decimals` places in the correct direction and appropriately rounded; however, `x` must also be then converted to a `uint192` fixed-point representation.	Echidna	**Passed**
`test_plus_commutative`: The `plus()` function maintains the commutative property of addition.	Echidna	**Passed**
`test_plus_associative`: The `plus()` function maintains the associative property of addition.	Echidna	**Passed**
`test_mulDiv256`: `((x*y)/z) * z == (x*y)`	Echidna	**Passed**
`test_mulDiv256Rnd`: The result of a call to `mulDiv256Rnd` should be the rounded equivalent of the output of `mulDiv256`.	Echidna	**Passed**
`test_toFix`: Conversion from a `uint256` to a `uint192` should fail only when the result will not fit in a `uint192`.	Echidna	**Passed**

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 system uses Solidity v0.8.0 arithmetic operations, and there are thorough unit tests of the arithmetic performed in the contracts. However, the expected behavior of the system arithmetic is not well documented, and we identified rounding errors (TOB-RES-5) and an incorrect use of function parameters (TOB-RES-8). Further investigation of the Reserve protocol's arithmetic operations through dynamic testing would be beneficial.	**Further Investigation Required**
Auditing	All critical state changes trigger events. However, the documentation we were provided lacks information on the use of off-chain components in behavior monitoring as well as a formal incident response plan.	**Satisfactory**
Authentication / Access Controls	The system does not follow the principle of least privilege. The owner of the system has the ability to manipulate numerous critical system parameters (TOB-RES-12, appendix D). The extensiveness of the owner's privileges creates a single point of failure within the system.	**Weak**
Complexity Management	Most of the logic is separated into functions that have clear purposes. However, many functions lack clear documentation. Additionally, some functions have high cyclomatic complexity, which makes them difficult to test.	**Moderate**

Decentralization	The Reserve protocol team indicated that the RToken(s) it deploys will be owned by either a multisig or by on-chain governance, which will manage critical state changes. Additionally, the Reserve protocol is a permissionless system in which all users can deploy their own RTokens. If the system is not paused, users can enter and exit the system at will. However, there is a lack of internal and public documentation on the risks related to the numerous owner privileges (TOB-RES-12, appendix D), the risks related to interactions with arbitrary RTokens, and the ways in which users can check the correctness of RToken deployments.	**Moderate**
Documentation	We received limited documentation that lacks details on various process flows and were not provided with a thorough specification. Moreover, the code has a limited number of comments, some of which are contradictory to the system behavior.	**Weak**
Front-Running Resistance	The protocol is resistant to front-running of contract initializations, RSR seizures, and basket switches. However, back-running of an oracle update could enable a user to purchase RTokens at a discount (TOB-RES-14). Additionally, since arbitrageurs are key actors in the system, users should be provided documentation on the system's use of arbitrage and the risk of front-running.	**Moderate**
Testing and Verification	The Reserve protocol has high unit test coverage and uses Slither for static analysis. However, many system properties are not detailed, and more extensive unit testing would have caught issues like that in TOB-RES-8. Finally, dynamic testing of important system invariants would help catch additional issues. (See appendix C for recommendations on dynamic fuzz testing.)	**Moderate**

Summary of Findings

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

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

13	Lack of zero address checks in Deployer constructor	Data Validation	**Low**
14	RTokens can be purchased at a discount	Data Validation	**Medium**
15	Inconsistent use of the FixLib library	Undefined Behavior	**Informational**

Detailed Findings

1. Solidity compiler optimizations can be problematic	
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.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Reserve protocol contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

2. Lack of a two-step process for contract ownership changes	
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#L76-80`

Exploit Scenario

Alice, the owner of the Reserve protocol `Main` contract, calls `transferOwnership()` but accidentally enters the wrong address as the new owner address. As a result, she permanently loses access to the contract.

Recommendations

Short term, perform ownership transfers through a two-step process in which the owner proposes a new address and the transfer is completed once the new address has executed a call to accept the role.

Long term, identify and document all possible actions that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

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

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()`

Exploit Scenario

Alice uses the `Deployer` contract to create an RToken. However, she incorrectly sets the revenue share value for the `Furnace` contract to 0. All underlying collateral tokens appreciate in value, which triggers the minting and subsequent melting of RTokens.

Because the revenue share value is set to 0, no RTokens are transferred to the `Furnace` contract; this inflates the total supply of RTokens.

Recommendations

Short term, create a lower and upper bound for all governance- / owner-controlled state variables, and validate each state variable's bounds when performing updates.

Long term, use dynamic testing to check whether the bounds of each parameter hold and are sensible in various system states.

4. All auction initiation attempts may fail

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

Exploit Scenario

The protocol initiates an auction to sell COMP tokens in exchange for RTokens. However, because the `EasyAuction` contract's token allowance is insufficient, the call to `initiateAuction` reverts, and the token auction does not occur.

Recommendations

Short term, create a `uint256` value called `actualSellAmount` (or a similar name) that is equal to `sellAmount` divided by 1 plus the `EasyAuction` fee. Have the `GnosisTrade`

contract call `initiateAuction` with `actualSellAmount` instead of `sellAmount` but continue to approve the `EasyAuction` contract to transfer `sellAmount`.

Long term, ensure that all mock contracts that are used for testing closely resemble the corresponding original contracts. Additionally, ensure that all upstream computations that rely on `sellAmount` are updated to account for the fee.

5. Per-block issuance limit can be bypassed

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`.

Recommendations

Short term, consider implementing a one-block delay for all token issuances.

Long term, use dynamic fuzz testing to identify any edge cases that could invalidate system properties.

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

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.

Exploit Scenario

A collateral token backing the RToken depreciates in value and consequently enters a default state. The owner of the RToken switches the underlying basket and calls the BackingManager contract to begin the process of auctioning off all defaulted collateral tokens and buying backup collateral tokens. However, because the EasyAuction.initiateAuction function requires that req.minBuyAmount be greater than zero, the initiation of the auction fails. Moreover, because the system cannot attempt to execute any other recapitalization mechanisms, the RToken remains undercapitalized.

Recommendations

Short term, consider either removing the line of code in which req.minBuyAmount is set to zero or setting req.minBuyAmount to one.

Long term, ensure that all mock contracts that are used for testing closely resemble the corresponding original contracts. The GnosisMock contract does not include the check of _minBuyAmount and thus allows all associated unit tests to pass.

7. Fallen-target auctions can be prevented from occurring

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.

Exploit Scenario

Eve notices that there is an RToken with a deficit that is large enough to warrant a fallen-target token auction. She sends a large quantity of RSR tokens to the `BackingManager` contract. The transaction inflates the `basketTop` value in the `largestSurplusAndDeficit` function such that no other token in the `BackingManager` contract can be in surplus. As a result, the `BackingManager` contract must call `compromiseBasketsNeeded` to attempt to recapitalize the system.

Recommendations

Short term, omit the `BackingManager` contract's balance of RSR tokens from the `totalValue` calculation.

Long term, ensure that token airdrops can only benefit (not harm) the system and that they do not lead to unexpected system behavior.

8. Faulty RToken issuance-cancellation process

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`.

Exploit Scenario

Alice has five outstanding issuances and wishes to cancel the issuances in the range of indexes [3, 5). However, because of the incorrect use of the `left` input variable in `refundSpan`, Alice is refunded for the issuance requests in the range of indexes [0, 5)—that is, all of her issuance requests. Because none of the RToken issuances successfully finished, Alice must make an additional gas payment to restart the process.

Recommendations

Short term, replace all uses of `queue.left` in `refundSpan` with `left`.

Long term, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

9. Token auctions may not cover entire collateral token deficits

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`

Exploit Scenario

Alice calls the `BackingManager` contract to make a trade to cover a deficit in the system. The arithmetic performed in `prepareTradeSell` causes the `sellAmount` to decrease. Thus, the value of `minBuyAmount` also decreases, falling below `deficitAmount`. As a result, the auction does not cover the entire deficit, and additional auctions will have to be held.

Recommendations

Short term, have `prepareTradeToCoverDeficit` revert if `minBuyAmount` is less than `deficitAmount` after the call to `prepareTradeSell` .

Long term, use dynamic fuzz testing to identify any edge cases that could invalidate system properties.

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

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.

Exploit Scenario
The Chainlink system experiences an outage that prevents price feeds from being updated for an extended period of time. The Aave and Compound oracle systems continue reporting pricing data from the outdated feeds. During the outage, the status of a collateral token changes from `SOUND` to `IFFY`, but the change is not reflected on-chain. Eve is monitoring the prices off-chain and purchases RTokens during the outage, before an update causes the price to increase.

Recommendations
Short term, consider obtaining pricing data from Chainlink, which enables a protocol to validate the recency of its data.

Long term, consider using an off-monitoring solution to track extreme market conditions and to ensure that the Chainlink oracle system is live.

References
- Chainlink: Risk Mitigation
- Chainlink: Monitoring Data Feeds

11. An RSR seizure could leave the StRSR contract unusable

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.

Exploit Scenario

Eve sends RSR tokens to the `StRSR` contract, causing `rsrBalance` to be greater than the sum of `rsrAmount` and `dustRSRAmt` but approximately equal to `stakeRSR` and `rsrAmount`. Her transaction is followed by a call to `seizeRSR`, in which `stakeRSRToTake` ends up being equal to `stakeRSR`. The `stakeRSR` value is then set to zero, but a new era is not started. Alice and Bob try to unstake their tokens from the system but are unable to do so because there is no `stakeRSR` to use as their exit liquidity.

Recommendations

Short term, have the `StRSR` contract call the `beginEra` function if `stakeRSR` is set to zero.

Long term, use dynamic fuzz testing to identify any edge cases that could invalidate system properties.

12. System owner has excessive privileges

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.

Exploit Scenario

Eve is able to take ownership of a newly deployed RToken by gaining access to Alice's private key, which controls the system. Eve changes the address of the `RToken` contract to that of a malicious contract (via a call to `setRToken`) and uses the `BackingManager.grantRTokenAllowance` function to grant the malicious contract infinite approval for all collateral tokens. She then proceeds to steal all collateral from the protocol.

Recommendations

Short term, set all addresses in the `__ComponentRegistry_init` function, and remove all address-setter functions from the `ComponentRegistry` contract. Alternatively, add the `onlyInitializing` modifier to all address-setter functions in the `ComponentRegistry` contract.

Long term, develop user documentation on all risks associated with the system, including those associated with privileged users and the existence of a single point of failure.

13. Lack of zero address checks in Deployer constructor

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`

Exploit Scenario

Alice, a developer on the Reserve protocol team, sets the `gnosis` address to the zero address. Her mistake prevents the protocol from holding any token auctions and has severe consequences on the behavior of deployed RTokens.

Recommendations

Short term, have the `Deployer` contract verify that contract addresses are not set to the zero address.

Long term, ensure that all parameters provided by users (whether privileged or unprivileged) are validated on-chain. Additionally, expand the unit test suite to cover additional edge cases and to ensure that the system behaves as expected.

14. RTokens can be purchased at a discount

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`.

Exploit Scenario

Eve notices that a price update has caused the price of a collateral token to plummet and its status to change to `IFFY`. Eve back-runs the price update with a call to the `RToken.issue` function, requesting a large issuance that will need to be vested over some number of blocks. Eve waits for the collateral to return to the `SOUND` state and then immediately calls the `RToken.vestUpTo` function to vest her RTokens.

Recommendations

Short term, in the `issue` function, change `require(status != CollateralStatus.DISABLED)` to `require(status == CollateralStatus.SOUND)`.

Long term, use dynamic fuzz testing to identify any edge cases that could invalidate system properties.

15. Inconsistent use of the FixLib library

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.

Recommendations

Short term, update all parts of the codebase that are not compliant with the protocol specification and include additional comments explaining any deviations from the specification.

Long term, use differential testing to determine whether the use of standard arithmetic operations on `uint192` values introduces any edge cases that would not be an issue if the `FixLib` library were used instead.

Summary of Recommendations

The Reserve protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Reserve address the findings detailed in this report and take the following additional steps prior to deployment:

- Document the expected behavior of all functions in the system.
- Document the risks that users face and the responsibilities they have when interacting with an arbitrary RToken.
- Identify and analyze all system properties that are expected to hold.
- Use Echidna to test the system properties, and use the recommendations listed in appendix C as a starting point for assertion testing. Additionally, perform differential fuzz testing of the `p0` and `p1` release candidates to verify that the candidates' system behavior is consistent.
- Develop a detailed incident response plan to ensure that any issues that arise can be addressed promptly and without confusion (appendix F).

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
Front-Running Resistance	The system's resistance to front-running attacks
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

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. Recommendations on Fuzz Testing with Echidna

This appendix outlines a list of recommendations on creating a robust and mature suite of Echidna fuzz tests. We recommend that the Reserve team first review the documentation, complete the exercises provided for Echidna in the `building-secure-contracts` repository, and review Echidna's configuration options and their functionalities.

- **Use an external testing approach.** An external testing approach involves testing system properties by making external calls to a different contract. The only accessible functions / state variables in the external contract will be those marked as `public` / `external`. This testing approach is useful for systems that have complex initialization flows.

  Figure C.1 shows an example of an external testing approach. The `EchidnaContract.testRTokenIssuance` function calls into the `RToken.issue` function. The call enables Echidna to compare the state of the system after the call to `issue` to the state before the call to identify whether any system properties were violated. Note that in the context of these tests, `msg.sender` will be the `EchidnaContract` and not a `sender`. A subsequent recommendation in this appendix explains how to emulate multiple senders while still using a single `EchidnaContract`.

```
contract EchidnaContract {
    constructor() public {
        rTokenContract = ..;
    }
    function testRTokenIssuance(uint256 amount) public returns (bool) {
        // Precondition checks
        // Action: Call RToken.issue()
        try rTokenContract.issue(amount){
            // Postcondition checks (happy path)
        } catch (bytes memory) {
            // Postcondition checks (not-so-happy path)
        };
    }
}
```

Figure C.1: An example contract for external testing of the `RToken.issue` function

- **Use assertion testing to validate more complex and intricate system invariants.** Using Echidna in assertion testing mode makes it possible to test more complex system invariants. Although property-based testing is faster and easier to use than assertion testing, the functions being tested cannot take explicit input parameters, and no coverage results are provided at the end of the testing cycle. Assertion testing solves both of those problems in addition to identifying system invariants that could be broken in the middle of a transaction. Assertion testing can

be enabled by setting the `testMode` configuration variable to `assertion`. In general, we recommend using the following workflow when testing a system property through assertion testing:

  - **Precondition checks:** These checks act as barriers to entry for the fuzzer and effectively require that the system meet certain conditions before performing the "action." If the system does not pass these checks, the property should not be tested. In the context of `RToken.issue`, the preconditions could include a requirement that the sender own enough collateral tokens to mint the amount of tokens specified by `amtRToken`.

  - **Execution of the action:** The action is the code or function call that is run in order to test the property. Calling `RToken.issue()` is the action.

  - **Postcondition checks:** These checks ensure that the state of the system after the execution of the action maintains the system property. An example postcondition check would be ensuring that the sender's RToken balance has increased by `amtRToken`. The correctness of the testing is directly related to the efficacy of the postcondition checks in validating that property.

- **Use mock oracles and auctions.** Ideally, a fuzz testing environment should mimic the code that will actually be deployed. However, in many cases, that is not possible. For example, it is not possible to deploy an off-chain oracle, though a mock oracle can be used to obtain prices within a certain range and to test both happy and "not-so-happy" paths. For instance, the team could create a `MockOracleMixin` contract that overrides the `consultOracle` function and returns a value within a target range. Overriding a function to create custom behavior is a powerful technique for changing the functionality of code without affecting the rest of the system. The team could take a similar approach to simulating auctions, creating a `GnosisMock.initiateAuction` function that can be called to immediately settle an auction.

- **Use mintable ERC20 contracts.** Creating mock ERC20 contracts that have public `mint` functions facilitates complex testing. For example, a `MockCollateral` contract with a public `mint` function would allow a sender to mint enough collateral tokens to pass the example precondition check for `RToken.issue` mentioned above. Using a `MockCollateral` contract with only an internal `_mint` function would significantly limit the variety of system properties that could be tested through an external testing mode.

- **Use the `Account` contract proxy pattern to emulate multiple system actors.** Figure C.2 shows an example of an `Account` contract. An `Account` contract can hold tokens / ether and perform function calls. Thus, an `Account` contract is *functionally* equivalent to an externally owned account.

The contract shown in figure C.2 contains a `proxy` function through which arbitrary function calls can be made. In the `RToken.issue` example, the `target` argument of `Account.proxy` would be the address of the `RToken` contract, and `_calldata` would be the encoded calldata passed to `RToken.issue`. Note, though, that the `msg.sender` of a call to `proxy` would be the `Account` contract, not the main Echidna contract. Creating multiple `Account` contracts would enable the team to emulate multiple senders.

```
contract Account {
    /* An account used to interact with underlying contracts
    account = new Account()
    */
    function proxy(address target, bytes memory _calldata)
        public
        returns (bytes memory)
    {
        (bool success, bytes memory returnData) = address(target).call(
            _calldata
        );
        require(success);
        return returnData;
    }
}
```

Figure C.2: The `Account` contract can be used to emulate multiple senders.

- **Use `try-catch` closures to test system properties**. A common mistake in the execution of fuzz testing is testing only the happy execution paths. However, testing unhappy paths is just as important for debugging and completeness of a testing effort as testing happy paths. Implementing `try-catch` closures allows the tester to specify different postconditions based on the success or failure of a call. Additionally, `try-catch` closures can be useful in identifying errors in fuzz test code, such as unexpected overflows.

- **Use stateful testing.** Echidna can perform both stateful and stateless testing. In stateful testing, the fuzzer is capable of threading together multiple transactions before wiping the state of the EVM. (The `seqLen` configuration variable specifies the number of transactions executed in the sequence.) This makes stateful testing more powerful, as it facilitates testing of more complex system properties.

  For example, say that the team has created a fuzz test called `testRTokenIssuance` that tests the `RToken.issue` function. Then the team writes a new fuzz test for the `RToken.redeem` function, `testRTokenRedeem`, which requires an existing balance of RTokens as a precondition. Through stateful testing, the fuzzer can call `testRTokenIssuance` and `testRTokenRedeem` sequentially. Since a call to `testRTokenIssuance` will create a balance of RTokens for the sender, the precondition check in `testRTokenRedeem` will pass. (Note that an explicit call to

RToken.issue does *not* have to be made in testRTokenRedeem.) Writing strong precondition checks and leveraging stateful testing makes it possible to test a large variety of code paths and sequences without having to explicitly call a large number of functions.

- **Create a separate Setup contract that is inherited by the main Echidna contract.** The Setup contract should be responsible for deploying the system and maintaining state variables. The main Echidna contract should inherit from the Setup contract and hold all of the fuzz tests. With this modular design, fuzz testing can be performed in one place, and the general state can be maintained / setup can be performed in another. For very large deployments, it may be necessary to increase the value of the codeSize configuration variable. Additionally, if the initialization is too complex for a constructor, Etheno can be used to help set up the environment.

- **Use the coverage file for debugging.** The coverage file provided by Echidna is essential to the debugging process. When a team is building a fuzz test, the coverage file can help it determine whether the fuzz test is performing as expected and whether there are requirements or conditions that Echidna will be unable to pass. A common mistake is assuming that a fuzz test is running properly even if Echidna is unable to pass a specific precondition. However, if Echidna cannot pass a precondition required to test a system property, it cannot actually check the system property. The coverage file also aids in determining whether any preconditions required to test a system property are missing. To enable coverage, set the corpusDir configuration variable. For more information about Echidna's coverage capabilities, consult its README.

- **Use the between function to bound values between a lower and upper limit.** Figure C.3 shows an example of the between function, which we recommend using whenever a variable should be bounded. The function is more powerful than a require statement, because a require statement that fails will cause the transaction to revert. Using between prevents such a revert and optimizes the computation. For example, a require statement like require(x < 10), where x is of type uint256, would almost always fail. If x were bound in the range [0, 9], each transaction would be able to pass the require(x < 10) statement, allowing the fuzzer to continue its execution.

```
function between(
    uint256 val,
    uint256 lower,
    uint256 upper
) internal pure returns (uint256) {
    return lower + (val % (upper - lower + 1));
}
```

Figure C.3: The `between` function can be used to bound variables.

- **Add comprehensive event logging mechanisms to all fuzz tests to aid in debugging.** Logging events during smart contract fuzzing is crucial for understanding the state of the system when a system property is broken. Without logging, it is difficult to identify the arithmetic operation or computation that caused the failure. Insufficient logging may also increase the rate of false positives. In general, any action that changes a value should trigger an event.
- **Enrich each fuzz test with comments explaining the pre- and postconditions of the test.** Strong fuzz testing requires well-defined preconditions (for guiding the fuzzer) and postconditions (for properly testing the invariant(s) in question). Comments explaining the bounds on certain values and the importance of the system properties being tested aid in test suite maintenance and debugging efforts.
- **Integrate fuzz testing into the CI / CD workflow.** Continuous fuzz testing can help quickly identify any code changes that will result in a violation of a system property and forces developers to update the fuzz test suite in parallel with the code. Running fuzz campaigns stochastically may cause a divergence between the operations in the code and the fuzz tests. The duration of a fuzzer's execution is dictated by the `testLimit` configuration variable.

D. System Owner's Privileges

This appendix provides a list of the privileges granted to the owner of the system, with those privileges grouped by contract.

Contract	Owner Privileges
`Main` (TOB-RES-12)	• Pausing the system • Upgrading the `Main` implementation contract • Updating the `RToken` address • Updating the `StRSR` address • Updating the `AssetRegistry` address • Updating the `BasketHandler` address • Updating the `BackingManager` address • Updating the `Distributor` address • Updating the `RevenueTrader` address of the RToken and RSR token • Updating the `Furnace` address • Updating the `Broker` address
`AssetRegistry`	• Upgrading the `AssetRegistry` implementation contract • Registering / unregistering an `Asset` or `Collateral`
`BackingManager`	• Upgrading the `BackingManager` implementation contract • Updating the maximum slippage allowed for a trade • Updating the dust amount • Updating the delay between trades / transfers of surplus assets • Updating the buffer of collateral tokens held by the `BackingManager`
`BasketHandler`	• Upgrading the `BasketHandler` implementation contract • Updating the prime basket • Updating the reference basket • Updating the backup collateral tokens

`Broker`	• Upgrading the `Broker` implementation contract • Disabling the `Broker` • Updating the minimum bid size for a trade • Updating the duration of token auctions
`Distributor`	• Upgrading the `Distributor` implementation contract • Updating the weighted distribution of RSR tokens and RTokens to various addresses
`Furnace`	• Upgrading the `Furnace` implementation contract • Updating the frequency with which RTokens are melted • Updating the rate at which RTokens are melted
`RevenueTrader`	• Upgrading the `RevenueTrader` implementation contract • Updating the maximum slippage allowed for a trade • Updating the dust amount
`RToken`	• Upgrading the `RToken` implementation contract • Updating the rate at which the `totalSupply` of the RToken increases
`StRSR`	• Upgrading the `StRSR` implementation contract • Updating the name of the token • Updating the symbol of the token • Updating the delay between the unstaking and withdrawal of RSR tokens • Updating the frequency with which RSR token rewards are paid out to stakers • Updating the amount of the RSR token rewards paid to stakers
`Collateral`	• Setting the duration of the delay before a collateral token is considered to be in default

	• Setting the delta that determines the extent of a deviation in a collateral token price
`Asset`	• Setting the maximum trading volume allowed for an `Asset`

E. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Use the `interaction` and `governance` modifiers instead of a `require` statement in the `refreshBasket` function (figure E.1).**

```
function refreshBasket() external {
    require(!main.paused() || main.owner() == _msgSender(), "unpaused or by owner");
    [...]
}
```

Figure E.1: A snippet of the `refreshBasket` function in BasketHandler.sol#L117-118

- **Remove the unnecessary check (that is highlighted in figure E.2) from the `StRSR.seizeRSR` function**.

```
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;
    [...]
}
```

Figure E.2: A snippet of the `seizeRSR` function in StRSR.sol#L244-251

- **Remove line 104 from `GnosisTrade.settle`**. The `stateTransition` modifier executes the necessary state change (figure E.3).

```
 98    function settle()
 99       external
100       stateTransition(TradeStatus.OPEN, TradeStatus.CLOSED)
101       returns (uint256 soldAmt, uint256 boughtAmt)
102    {
103       require(msg.sender == origin, "only origin can settle");
104       status = TradeStatus.PENDING;
```

Figure E.3: A snippet of the `settle` function in GnosisTrade.sol#L98-104

- **Remove the check highlighted in figure E.4 from `StRSR.unstake`**. The `_burn` function performs that same check.

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

Figure E.4: A snippet of the `unstake` function in StRSR.sol#L178-181

- **In `RToken.vestUpTo`, change `queue.left == endId` to `queue.left < endId` and remove line 452.** The inclusion of both checks is redundant (figure E.5).

```
 450    function vestUpTo(address account, uint256 endId) private {
 451        IssueQueue storage queue = issueQueues[account];
 452        if (queue.left == endId) return;
 453
 454        require(queue.left <= endId && endId <= queue.right, "'endId' is out of
range");
```

Figure E.5: A snippet of the `vestUpTo` function in RToken.sol#L450-454

- **Remove the `Asset` abstract contract's inheritance of `Initializable`.** This inheritance structure is unnecessary, as there will not be any `Asset` contract proxies (figure E.6).

```
abstract contract Asset is Initializable, IAsset {[...]}
```

Figure E.6: The `Asset` contract's inheritance structure in Asset.sol#L9

- **Have the `Governor.state` function return `super.state(proposalId)` rather than `GovernorTimelockControl.state(proposalId)`.** Because of the use of C3 linearization, the `GovernorTimelockControl` contract's `state` function is called (figure E.7).

```
function state(uint256 proposalId)
    public
    view
    override(Governor, GovernorTimelockControl)
    returns (ProposalState)
{
    return GovernorTimelockControl.state(proposalId);
}
```

Figure E.7: The `state` function in Governor.sol#L85-92

- **Use either the `notPaused` or `interaction` modifier.** The modifiers both perform the same check and should not be used interchangeably (figure E.8).

```
modifier interaction() {
```

```
    require(!main.paused(), "paused");
    _;
}
[...]
modifier notPaused() {
    require(!main.paused(), "paused");
    _;
}
```

Figure E.8: The `interaction` and `notPaused` modifiers in `Component.sol#L28-41`

F. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Reserve will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.