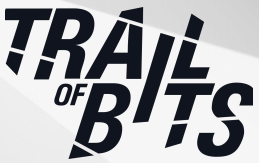

Bunni v2

Security Assessment

January 28, 2025

Prepared for:
Zefram
Bacon Labs

Prepared by: **Elvis Skoždopolj, Priyanka Bose, and Michael Colburn**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@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

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Project Timeline

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

Date	Event
September 20, 2024	Pre-project kickoff call
September 30, 2024	Status update meeting #1
October 9, 2024	Status update meeting #2
October 15, 2024	Status update meeting #3
October 23, 2024	Delivery of report draft
October 23, 2024	Report readout meeting
January 28, 2025	Delivery of comprehensive report

Executive Summary

Engagement Overview

Bacon Labs engaged Trail of Bits to review the security of its Bunni v2 smart contracts. Bunni v2 is an automated market maker (AMM) built on top of Uniswap v4 that allows users to implement custom behavior and liquidity curves. It introduces multiple new features: an auction-managed AMM (am-AMM) to optimize swap fees and recapture MEV, liquidity density functions (LDFs) to allow liquidity providers to provide liquidity in complex shapes and dynamically switch between liquidity shapes, autonomous rebalancing, rehypothecation into yield bearing vaults, and surge fees to protect against sandwich attacks.

A team of three consultants conducted the review from September 23 to October 22, 2024, for a total of eight engineer-weeks of effort. Our testing efforts focused on reviewing the contracts for common Solidity pitfalls; reviewing the internal accounting; reviewing interactions with external contracts, such as the Uniswap v4 `PoolManager`, `FloodPlain`, and `Permit2` contracts; reviewing access controls; and assessing risks related to interacting with arbitrary hooks, hooklets, vaults, and tokens. Additionally, we reviewed the rounding directions and the magnitude of arithmetic errors in the LDFs and `BunniSwapMath`, and we looked into issues related to token approvals, round-trip swaps, swaps happening on edge ticks, and equivalency and symmetricity of various calculations. With full access to source code and documentation, we performed static and dynamic testing of the Bunni v2 codebase, using automated and manual processes.

The commit under review was previously reviewed, and a list of known issues was shared with us prior to starting this security review. Any vulnerabilities present in this list, which includes high-severity issues in multiple components, were considered known and have been omitted from this report.

Observations and Impact

Bunni v2 uses a singleton design where the ERC-6909 tokens minted via the Uniswap v4 `PoolManager` contract are kept together in the `BunniHub` contract. This increases the flexibility and extensibility of the protocol; however, it comes with inherent risks that a single protocol vulnerability can drain the entirety of the protocol total value locked (TVL). This is mitigated by reducing trust assumptions when interacting with external contracts and keeping separate internal accounting values for each pool.

The codebase allows arbitrary hooks, hooklets, LDFs, tokens, and vaults to be used with the protocol. A pool can have up to two ERC-4626 vaults, one for each token of the pool, which can be used for rehypothecation. Since these vaults are arbitrary contracts, the protocol guards against misuse by not trusting the return values of the vault functions and instead

verifying the actual values by using the `PoolManager` contract. However, token approvals to vaults are not cleared (TOB-BUNNI-2) after use, which increases the risks of token theft.

The rebalance mechanism allows pools to be automatically rebalanced by using an external protocol `FloodPlain` for the settling of the created rebalance orders. However, this rebalancing mechanism contains multiple flaws: the access controls are insufficient to protect against token theft (TOB-BUNNI-6), the sequential execution of the rebalance pre- and post-hooks is not enforced (TOB-BUNNI-7), and the excess amounts calculation can be inflated to drain the pool (TOB-BUNNI-9). Furthermore, the `FloodPlain` codebase itself contains high-severity vulnerabilities and does not appear to have gone through a security review.

The codebase contains complex mathematical operations, particularly around the swap functionality and liquidity density functions. During our invariant testing of these components, we identified several issues, including a few high-severity ones. Noteworthy issues include arithmetic and rounding errors (TOB-BUNNI-13), users obtaining free tokens during swaps (TOB-BUNNI-15), the potential for round-trip swaps to yield profit (TOB-BUNNI-16), and different swap configurations resulting in varying input/output token amounts, allowing users to provide fewer tokens for the same output (TOB-BUNNI-17).

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Bacon Labs take the following steps prior to deploying the system:

- **Remediate the findings disclosed in this report and the prior security review.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations. Consider an additional security review due to the extent of the changes necessary to remediate all of the issues.
- **Create a thorough stateful fuzzing suite and triage the pending property failures included in this report.** Due to the complexity of the protocol and lack of a systematic approach to rounding and arithmetic errors, there is a high likelihood that the LDFs and `BunniSwapMath` contain multiple issues that can be abused for a profit. System- and function-level invariants should be created and tested with both stateful and stateless fuzzing, and the failing properties included in this report should be investigated further.
- **Improve the testing suite.** While the testing suite covers most normal system behavior and uses stateless fuzzing for a large number of library functions, there is a clear lack of coverage of common adversarial situations. The coverage of the testing suite should be improved, testing for both normal system behavior and adversarial situations.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	1
Low	2
Informational	7
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	4
Auditing and Logging	1
Data Validation	14

Project Goals

The engagement was scoped to provide a security assessment of the Bacon Labs Bunni v2 smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can a malicious `AmAmm` manager exploit honest pools?
- Are rebalancing orders done correctly?
- Can rebalancing be manipulated?
- Can the rebalance validation be bypassed to drain honest pools?
- Does the system implement reasonable bounds for configurable parameters?
- Can a malicious token or hook be used to steal assets from honest pools?
- Can token approvals be used to steal assets?
- Can `permit2` approvals be used to steal assets?
- Do calculations consistently round in favor of the protocol?
- Are the arithmetic errors properly bound?
- Are native asset and token deposits handled securely?
- Can users get free tokens by performing round-trip swaps?

Project Targets

The engagement involved a review and testing of the targets listed below.

Bunni v2

Repository	https://github.com/timeless-fi/bunni-v2
Version	`7faae4718eecda1b33dc3abd894431ed2d16c929`
Type	Solidity
Platform	Ethereum

AmAmm

Repository	https://github.com/Bunniapp/biddog
Version	`95f4270ad4447e96044973580afda9176730e7c8`
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:

- **`BunniHub` and `BunniHubLogic`.** We reviewed the `BunniHub` and `BunniHubLogic` contracts for transaction reordering risks, such as the ability to front run deposits and the initialization and deployment of a `BunniToken` and pool, and the impact of interactions with arbitrary hooks, hooklets, tokens, and vaults. We looked into the access controls to identify if there is a way to bypass access controls or fake the caller via the multicall contracts. We looked at how liquidity is deposited and withdrawn, looking for accounting errors and missing validation. Additionally, we reviewed how native assets are handled and refunded, how swaps are handled through the hook callback, and the trust assumptions around how data returned from calls to the ERC-4626 vaults is handled.

- **`BunniHook` and `BunniHookLogic`.** We reviewed the `BunniHook` and `BunniHookLogic` contracts to identify missing or insufficient access controls, whether privileged role actions can have a negative impact on users, and whether configurable parameters have reasonable limits (TOB-BUNNI-8). We looked into the trust assumptions made when interacting with the `FloodPlain` contracts to determine if rebalance orders can be faked, executed multiple times, or in any way used to steal assets for legitimate liquidity pools (TOB-BUNNI-6, TOB-BUNNI-7, TOB-BUNNI-9). We investigated how swaps and initialization are handled, how fees are accrued and claimed, how surge fees are activated and calculated, and how interactions with the Uniswap v4 `PoolManager` contract are handled. We reviewed the calculations, looking for incorrect rounding directions or unhandled arithmetic errors that would result in tokens being extracted from a pool.

- **`BunniToken`.** We reviewed if the calculation, distribution, and claiming of scores and rewards can be abused to grief users, claim rewards multiple times, or extract value from the `BunniToken`. We checked if the implementation contains common ERC-20 vulnerabilities such as minting via self-transfers. This component contained a large number of known issues that are not included in this report.

- **`BunniQuoter`.** We reviewed the function of the `BunniQuoter` contract for equivalence to the non-view functions of the `BunniHook` and `BunniHub` contracts.

- **`Oracle`.** We reviewed the `Oracle` implementation against the Uniswap v3 implementation and against the proposed truncated oracle implementation of Uniswap v4.

- **AmAmm.** We reviewed the `AmAmm` contract to try to identify any way to interfere with the bidding process, scenarios that could result in rent being over- or undercharged, unexpected state transitions, as well as how the `amAMM` manager could use their privileged status to influence Bunni v2 pools.

- **Liquidity density functions, their libraries, and `BunniSwapMath`.** We created a stateless fuzzing suite testing various invariants detailed in the Automated Testing section of the report. We reviewed the calculations for rounding and arithmetic errors and investigated whether a zero-amount swap can lead to nonzero tokens out, if round-trip swaps can be profitable, if the arithmetic error bounds for the calculations are correct, and if functions that are supposed to be equivalent or symmetrical return the correct results.

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:

- The commit under review was previously reviewed, and a list of known issues was presented to us prior to starting the security review. Any vulnerabilities contained in this list were considered known and are not included in this report. We applied less scrutiny when reviewing parts of the codebase that contain a large number of known security vulnerabilities, such as the `BunniToken` contract.

- We did not review the library contracts forked from Uniswap and contracts indicated as out of scope, including the `SqrtPriceMath`, `SwapMath`, `OrderHashMemory`, `LiquidityAmounts`, `ERC20`, and `Ownable` contracts.

- While we reviewed and tested the LDFs and the `BunniSwapMath` contract using fuzzing, we did not fully verify the correctness of the calculations due to their complexity and the time-boxed nature of the review. We discovered multiple invariant failures that indicate the presence of incorrect rounding directions, arithmetic errors, or logical issues in these contracts, and these should be investigated further.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	No explicit policy, as the rules created run under a few seconds
Medusa	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Timeout of 24 hours

Summary of Invariants

Our automated testing and verification work focused on the following system properties:

Component	Invariant Type	Total Number
`BunniSwapMath` library	Stateless invariants	5
`UniformDistribution`	Stateless invariants	2
`GeometricDistribution`	Stateless invariants	2
`CarpetedGeometricDistribution`	Stateless invariants	2
`DoubleGeometricDistribution`	Stateless invariants	2
`CarpetedDoubleGeometricDistribution`	Stateless invariants	2

Test Results

The results of this focused testing are detailed below.

`BunniSwapMath` library: Implements the swap functionality for `BunniSwap`

Property	Tool	Result
`computeSwap` in `BunniSwapMath` should output the same amount of input and output tokens both in `ExactIn` and `ExactOut` configurations given the same pool state.	Medusa	TOB-BUNNI-17
Users should not be able to gain any tokens through round-trip swaps.	Medusa	TOB-BUNNI-16
Users should not be able to get free output tokens for zero input tokens when `amountSpecified` is nonzero for a given valid pool state.	Medusa	TOB-BUNNI-15
`computeSwap` in `BunniSwapMath` should not raise any panics during a swap on a valid pool state.	Medusa	TOB-BUNNI-18
`computeSwap` in `BunniSwapMath` should output a valid `sqrtPrice` of the pool after the swap.	Medusa	**Passed**

LDFs: Implement various liquidity curves for `BunniSwap`

Property	Tool	Result
Given a valid cumulative amount of `token0`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount0` should be less than or equal to the specified cumulative amount for `UniformDistribution` in rounded ticks [`roundedTick`, `tickUpper`).	Medusa	TOB-BUNNI-19
Given a valid cumulative amount of `token1`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount1` should be greater than or equal to the specified cumulative amount for `UniformDistribution` in rounded ticks [`tickLower`, `roundedTick`).	Medusa	**Passed**

Given a valid cumulative amount of `token0`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount0` should be less than or equal to the specified cumulative amount for `GeometricDistribution` in rounded ticks [`roundedTick`, `tickUpper`).	Medusa	**Passed**
Given a valid cumulative amount of `token1`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount1` should be greater than or equal to the specified cumulative amount for `GeometricDistribution` in rounded ticks [`tickLower`, `roundedTick`].	Medusa	**Passed**
Given a valid cumulative amount of `token0`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount0` should be less than or equal to the specified cumulative amount for `DoubleGeometricDistribution` in rounded ticks [`roundedTick`, `tickUpper`).	Medusa	**Passed**
Given a valid cumulative amount of `token1`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount1` should be greater than or equal to the specified cumulative amount for `DoubleGeometricDistribution` in rounded ticks [`tickLower`, `roundedTick`].	Medusa	
Given a valid cumulative amount of `token0`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount0` should be less than or equal to the specified cumulative amount for `CarpetedGeometricDistribution` in rounded ticks [`roundedTick`, `tickUpper`).	Medusa	TOB-BUNNI-19
Given a valid cumulative amount of `token1`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount1` should be greater than or equal to the specified cumulative amount for `CarpetedGeometricDistribution` in rounded ticks [`tickLower`, `roundedTick`].	Medusa	**Passed**
Given a valid cumulative amount of `token0`, the cumulative	Medusa	**Passed**

amount calculated using the rounded tick from `inverseCommulativeAmount0` should be less than or equal to the specified cumulative amount for `CarpetedDoubleGeometricDistribution` in rounded ticks [`roundedTick`, `tickUpper`).		
Given a valid cumulative amount of `token1`, the cumulative amount calculated using the rounded tick from `inverseCommulativeAmount1` should be greater than or equal to the specified cumulative amount for `CarpetedDoubleGeometricDistribution` in rounded ticks [`tickLower`, `roundedTick`].	Medusa	**Passed**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	Most instances of unchecked arithmetic are clearly documented and contain validation to prevent underflows. Much of the math is abstracted away through the LDFs, `BunniSwapMath`, or Uniswap libraries. However, the math is very complex and the system lacks a systematic approach to rounding and arithmetic errors. There are multiple instances of code paths that check for errors and either truncate the value or rerun the calculation multiple times. This indicates that the system contains large relative arithmetic errors that could potentially be abused to extract value from the system. The properties defined in the Automated Testing section indicate that the math is insufficiently robust. It would be beneficial to fuzz the math libraries more thoroughly and investigate the failing properties further. Operations should be fuzzed for equivalence and symmetricity, and rounding direction should be enforced such that they always favor the protocol.	**Weak**
Auditing	Most of the state-changing functions emit events; however, we found a few instances (TOB-BUNNI-10) where additional event emission would be beneficial. We did not evaluate a potential monitoring strategy, nor an incident response plan, and these areas would require additional investigation.	**Further Investigation Required**
Authentication / Access Controls	Overall, the protocol is permissionless and non-upgradeable, and it contains a limited number of privileged actions. However, users should be made aware of the risks when interacting with arbitrary pools since the deployer of the pool could use malicious contracts to	**Moderate**

	drain user assets. The interactions between the `BunniHub`, `BunniHook`, the Uniswap v4 `PoolManager` contract, and `FloodPlain` each have strict access controls. However, a vulnerability in any of the components could lead to a legitimate pool being drained, as demonstrated by TOB-BUNNI-6. Additionally, the `AmAmm` manager has a large amount of power over the pool that they manage, being able to arbitrarily set fees. Since anyone can become the `AmAmm` manager of a pool, this power should be limited by choosing reasonable swap fee limits for each pool. The protocol uses `Permit2` to handle most token transfers and uses the `isValidSignature` function as an access control for `Permit2` signatures provided for the rebalancing mechanism. This function, along with the access control between `BunniHook` and `FloodPlain`, is insufficient and should be redesigned (TOB-BUNNI-6).	
Complexity Management	Multiple vendored libraries from Uniswap and Solady have modifications that are insufficiently documented. The system is incredibly configurable with many different and optional features that can interact. While this increases the flexibility of the system, it also increases the attack surface. From a security perspective, implementing the protocol through Uniswap v4 hooks provides a solid foundation instead of reinventing the wheel, but it also introduces a second codebase that must be understood at a deep level, with its own coding style and underlying assumptions. The system would benefit from reducing code duplication outlined in the Code Quality Issues section. The LDFs and `BunniSwapMath` contain very complex arithmetic that is difficult to fully verify. We discovered multiple invariant failures that should be investigated further; for more information, see the Automated Testing section. The system lacks a systematic approach to rounding directions and handling of arithmetic errors.	**Weak**
Decentralization	The protocol is decentralized and aims to be permissionless. The contracts are not upgradeable, though the address of the `BunniZone` contract that manages the allowlist for rebalance orders can be	**Moderate**

	updated. However, some privileged actions, such as updating the `globalAmAmmEnabledOverride` and `amAmmEnabledOverride` variables, could impact users. The protocol supports arbitrary hooks, hooklets, tokens, and vaults. Any of these components could be misused by a malicious deployer to drain users' assets. Additionally, the `AmAmm` manager has the power to update fees at any point. Users should be made aware of the risks of interacting with arbitrary pools.	
Documentation	The system contains extensive public documentation along with white papers. The public documentation provides a good high-level overview of the protocol, and the inline code documentation explains what the code is doing during more complex flows. Given how highly configurable pools are, providing users with sensible defaults and how to set parameters in a safe way will be crucial. Currently, the Guides section of the public documentation is an empty placeholder.	**Moderate**
Low-Level Manipulation	The Bunni v2 contracts include some low-level calls. These are generally handled by libraries that perform appropriate checks. The codebase includes a fair amount of assembly, but this is mostly in the form of simple statements and packing variables, and most of the more complex assembly blocks have comments with either the Solidity equivalent or line-by-line explanations.	**Satisfactory**
Testing and Verification	As a result of the way the protocol is designed, the test suite leans more on integration tests. The LDFs are covered with unit tests, with the exception of the `BuyTheDipGeometricDistribution` contract, which only has limited coverage. The tests are generally parameterized to allow for fuzzing. Given the high level of configurability, the test suite relies too heavily on hard-coded parameters. For example, the tests in `BunniHub.t.sol` all use a `GeometricDistribution` LDF with the same parameters.	**Weak**
Transaction Ordering	The protocol allows users to set price limits and specify input or output amounts depending on the type of swap to protect against slippage. Additionally, the am-AMM and dynamic fee mechanisms are both designed to help	**Moderate**

	minimize the impact of MEV on swaps. However, we did identify one scenario that would allow a malicious fulfiller to front run the creation of rebalance orders in order to steal liquidity from the pool.	

Summary of Findings

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

ID	Title	Type	Severity
1	BunniToken permit cannot be revoked	Access Controls	Informational
2	Token approvals to ERC-4626 vaults are never revoked	Access Controls	Informational
3	Overly strict bid withdrawal validation reduces am-AMM efficiency by enabling griefing	Data Validation	Low
4	Users can bid arbitrarily low rent during the bidding process	Data Validation	Undetermined
5	Dirty bits of narrow types are not cleaned	Data Validation	Informational
6	Rebalance mechanism access control can be bypassed	Access Controls	High
7	Pools can be drained via the rebalance mechanism by selectively executing the rebalanceOrderPreHook and the rebalanceOrderPostHook	Data Validation	High
8	Missing maximum bounds for rebalance parameters	Data Validation	Informational
9	Excess liquidity can be inflated to create arbitrarily large rebalance orders	Data Validation	High
10	Insufficient event generation	Auditing and Logging	Informational
11	AmAmm manager can manipulate TWAP prices without risk	Access Controls	Medium

12	Lack of zero-value checks	Data Validation	**Informational**
13	Lack of systematic approach to rounding and arithmetic errors	Data Validation	**Undetermined**
14	Native assets deposited to pools with no native currencies are lost	Data Validation	**Informational**
15	Users can gain free tokens through the BunniSwap swap functionality	Data Validation	**High**
16	Users can gain tokens during round-trip swaps	Data Validation	**High**
17	Different amount of input/output tokens can be returned in ExactIn and ExactOut configurations during the swap	Data Validation	**Low**
18	BunniSwap swap functionality can cause panics during the swap	Data Validation	**Undetermined**
19	cumulativeAmount0 can be greater than the cumulative amount computed through inverse functionality for certain LDFs	Data Validation	**Undetermined**

Detailed Findings

1. BunniToken permit cannot be revoked

Severity: **Informational**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BUNNI-1
Target: `src/BunniToken.sol`	

Description

The `BunniToken` contract does not implement a way for permit signatures to be revoked, resulting in these signatures being valid until the deadline.

The `BunniToken` contract inherits the `permit` function from the `ERC20` contract, allowing users to sign a payload in order to allow another user or contract to spend their tokens.

```
function permit(address owner, address spender, uint256 value, uint256 deadline,
uint8 v, bytes32 r, bytes32 s)
    public
    virtual
{
    bytes32 nameHash = _constantNameHash();
    //  We simply calculate it on-the-fly to allow for cases where the `name` may
change.
    if (nameHash == bytes32(0)) nameHash = keccak256(bytes(name()));
    /// @solidity memory-safe-assembly
    assembly {
        // Revert if the block timestamp is greater than `deadline`.
        if gt(timestamp(), deadline) {
            mstore(0x00, 0x1a15a3cc) // `PermitExpired()`.
            revert(0x1c, 0x04)
        }
        let m := mload(0x40) // Grab the free memory pointer.
        // Clean the upper 96 bits.
        owner := shr(96, shl(96, owner))
        spender := shr(96, shl(96, spender))
        // Compute the nonce slot and load its value.
        mstore(0x0e, _NONCES_SLOT_SEED_WITH_SIGNATURE_PREFIX)
        mstore(0x00, owner)
        let nonceSlot := keccak256(0x0c, 0x20)
        let nonceValue := sload(nonceSlot)
        // Prepare the domain separator.
        mstore(m, _DOMAIN_TYPEHASH)
        mstore(add(m, 0x20), nameHash)
        mstore(add(m, 0x40), _VERSION_HASH)
```

```
        mstore(add(m, 0x60), chainid())
        mstore(add(m, 0x80), address())
        mstore(0x2e, keccak256(m, 0xa0))
        // Prepare the struct hash.
        mstore(m, _PERMIT_TYPEHASH)
        mstore(add(m, 0x20), owner)
        mstore(add(m, 0x40), spender)
        mstore(add(m, 0x60), value)
        mstore(add(m, 0x80), nonceValue)
        mstore(add(m, 0xa0), deadline)
        mstore(0x4e, keccak256(m, 0xc0))
        // Prepare the ecrecover calldata.
        mstore(0x00, keccak256(0x2c, 0x42))
        mstore(0x20, and(0xff, v))
        mstore(0x40, r)
        mstore(0x60, s)
        let t := staticcall(gas(), 1, 0, 0x80, 0x20, 0x20)
        // If the ecrecover fails, the returndatasize will be 0x00,
        // `owner` will be checked if it equals the hash at 0x00,
        // which evaluates to false (i.e. 0), and we will revert.
        // If the ecrecover succeeds, the returndatasize will be 0x20,
        // `owner` will be compared against the returned address at 0x20.
        if iszero(eq(mload(returndatasize()), owner)) {
            mstore(0x00, 0xddafbaef) // `InvalidPermit()`.
            revert(0x1c, 0x04)
        }
        // Increment and store the updated nonce.
        sstore(nonceSlot, add(nonceValue, t)) // `t` is 1 if ecrecover succeeds.
        // Compute the allowance slot and store the value.
        // The `owner` is already at slot 0x20.
        mstore(0x40, or(shl(160, _ALLOWANCE_SLOT_SEED), spender))
        sstore(keccak256(0x2c, 0x34), value)
        // Emit the {Approval} event.
        log3(add(m, 0x60), 0x20, _APPROVAL_EVENT_SIGNATURE, owner, spender)
        mstore(0x40, m) // Restore the free memory pointer.
        mstore(0x60, 0) // Restore the zero pointer.
    }
}
```

Figure 1.1: The `permit` function (src/base/ERC20.sol#L291–L355)

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.

2. Token approvals to ERC-4626 vaults are never revoked

Severity: **Informational**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BUNNI-2
Target: `src/BunniHub.sol, src/lib/BunniHubLogic.sol`	

Description

The `BunniHub` contract can approve an arbitrary ERC-4626 vault to handle its tokens when calling the `deposit` function of the vault; however, these approvals are never revoked.

The `BunniHub` contract can give approvals to spend a certain amount of its tokens to an ERC-4626 vault in order to deposit these tokens into the vault. For instance, this is done when updating the reserves during a liquidity deposit (figure 2.1) or when a pool does not have a sufficient amount of output tokens (figure 2.2).

```
// ...
// do vault deposit
address(token).safeApproveWithRetry(address(vault), amount);
reserveChange = vault.deposit(amount, address(this));
reserveChangeInUnderlying = vault.previewRedeem(reserveChange);
// ...
```

Figure 2.1: The `_depositVaultReserve` function (`src/lib/BunniHubLogic.sol#L665–L668`)

```
// ...
function _updateVaultReserveViaClaimTokens(int256 rawBalanceChange, Currency
currency, ERC4626 vault)
    internal
    returns (int256 reserveChange, int256 actualRawBalanceChange)
{
    // ...
        address(token).safeApproveWithRetry(address(vault), absAmount);
        reserveChange = vault.deposit(absAmount, address(this)).toInt256();
    // ...
```

Figure 2.2: The `_updateVaultReserveViaClaimTokens` function (`src/BunniHub.sol#L432–L433`)

However, since the vault implementation could be malicious, the `deposit` function is not guaranteed to perform the operation correctly and consume the token approvals. While the balance updates should prevent the tokens from being stolen, there could be a vulnerability in some other piece of the codebase that would allow this to be abused.

Recommendations

Short term, reset the approvals to zero after the call to the `deposit` function.

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

3. Overly strict bid withdrawal validation reduces am-AMM efficiency by enabling griefing

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-3
Target: `src/AmAmm.sol`	

Description

The `AmAmm` contract implements an auction mechanism that sells the rights to set and collect swap fees to the highest bidder. The proceeds of the auction are deducted over time as rent and distributed to liquidity providers in lieu of the swap fees that they have forfeited. The bidding process is defined in such a way that the top bidder and rent are determined K units of time (for Bunni v2, this is 24 hours) in advance of them taking effect.

Per the written description, the next bidder should be able to withdraw their deposit as long as they leave enough deposit to cover any difference, if any, between the remaining time left on the current top bidder's deposit and 24 hours. Therefore, if the current top bidder has at least 24 hours of deposit left, the next highest bidder should be able to withdraw their full deposit. The implementation used by Bunni v2, however, requires that the next highest bidder always leaves at least 24 hours of deposit regardless of the state of the current top bidder's deposit.

```
273     // require D_next / R_next >= K
274     if ((nextBid.deposit - amount) / nextBid.rent < K(id)) {
275         revert AmAmm__BidLocked();
276     }
```

Figure 3.1: A snippet of the `withdrawNextBid` function (`src/AmAmm.sol#L273–L276`)

While there is a `cancelNextBid` function that the next highest bidder can use to cancel their pending bid, if there is currently an active top bidder, this is only available as long as the top bid's deposit can still cover at least 24 hours.

```
319     // require D_top / R_top >= K
320     if (topBid.manager != address(0) && topBid.deposit / topBid.rent < K(id)) {
321         revert AmAmm__BidLocked();
322     }
323
324     // delete next bid from storage
325     delete _nextBids[id];
```

Figure 3.2: A snippet of the `cancelNextBid` function (`src/AmAmm.sol#L319–L325`)

This difference in behavior introduces more risk to bidders who wish to act as the fee manager but do not wish to exceed the current top bidder, as some portion of their funds may remain locked in the contract until the current top bidder exhausts their deposit completely or they are outbid by somebody else. This may also unintentionally incentivize top bidders to extend their reign longer than strictly economically necessary in order to spite competitors and also incentivize bidders to wait until there is no active top bidder to begin submitting bids, creating more gaps where the pool falls back to the default swap fee mechanism that may expose LPs to more arbitrage opportunities and reduce their overall earnings.

Exploit Scenario

Alice is the current swap fee manager, pays 10 `BunniToken` in rent per epoch, and has 40 tokens deposited. Bob notices that Alice only has a small deposit remaining and submits a bid at eight tokens per epoch and includes a 192-token deposit. Alice also notices her deposit was running low and deposits tokens to top up her deposit to 23 hours total. Alice can repeat this top up every day. Bob cannot recover any of his 192 tokens until Alice's deposit is exhausted or someone outbids him.

Recommendations

Short term, update the user- and developer-facing documentation so that users bidding through the am-AMM are aware of the risks.

Long term, carefully consider the economic (dis)incentives created when deviating from written specifications.

References

- am-AMM: An Auction-Managed Automated Market Maker

4. Users can bid arbitrarily low rent during the bidding process

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-4
Target: `lib/biddog/AmAMM.sol`	

Description

During the am-AMM auction bidding process, a user vying to be a future manager can bid extremely low rent, even zero, in the absence of a current am-AMM manager or next bids, distorting the bidding process.

As depicted in figure 4.1, in function `bid` in the `AmAmm` contract, when a user places a new bid, the rent is compared with the existing next bid rent and is set as the next bid if the rent is 10% more than the existing next bid rent. In the absence of an existing next bid, the user's rent can be zero and still pass the sanity checks. As a result, future bidders may also opt for bidding low rent, aiming to be just above 10% of the previous rent.

```
// update state machine
_updateAmAmmWrite(id);

// ensure bid is valid
// - manager can't be zero address
// - bid needs to be greater than the next bid by >10%
// - deposit needs to cover the rent for K hours
// - deposit needs to be a multiple of rent
// - payload needs to be valid
if (
    manager == address(0) || rent <=
_nextBids[id].rent.mulWad(MIN_BID_MULTIPLIER(id)) || deposit < rent * K(id)
        || deposit % rent != 0 || !_payloadIsValid(id, payload)
) {
    revert AmAmm__InvalidBid();
}
```

Figure 4.1: A snippet of the bid function in the AmAmm contract (src/AmAmm.sol#L75–L86)

Moreover, as illustrated in figure 4.2, in the function `_stateTransitionWrite` in the contract `AmAmm` where the new manager is set, a remarkably low next bid rent could still become the manager if there is no existing top bid, given that the rent must be greater than 10% of the existing top bid rent. This scenario allows the winning bidder to attain the manager's role at a substantially low rent and earn significantly more in swap fees.

```
// State D
// we charge rent from the top bid only until K epochs after the next bid was
submitted
// assuming the next bid's rent is greater than the top bid's rent + 10%, otherwise
we don't care about
// the next bid
bool nextBidIsBetter = nextBid.rent > topBid.rent.mulWad(MIN_BID_MULTIPLIER(id));
uint40 epochsPassed;
```

Figure 4.2: A snippet of the `_stateTransitionWrite` function in the AmAmm contract (src/AmAmm.sol#L676–L680)

Recommendations

Short term, implement a minimum rent requirement to ensure that there is a lowest acceptable bid that a bidder must provide to qualify for consideration as the next bid.

5. Dirty bits of narrow types are not cleaned

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BUNNI-5
Target: `src/lib/SwapMath.sol`	

Description

The `SwapMath` library used by Bunni v2 is based on the library of the same name from Uniswap v4. However, the `getSqrtPriceTarget` function in the Uniswap library has been updated to properly clean the higher bits of the `sqrtPrice*` parameters. Solidity makes no guarantees about the contents of these unused bits in the case of types smaller than one word (e.g., `uint24`, `uint160`, `address`).

```
function getSqrtPriceTarget(bool zeroForOne, uint160 sqrtPriceNextX96, uint160
sqrtPriceLimitX96)
    internal
    pure
    returns (uint160 sqrtPriceTargetX96)
{
    assembly {
        // a flag to toggle between sqrtPriceNextX96 and sqrtPriceLimitX96
        // when zeroForOne == true, nextOrLimit reduces to sqrtPriceNextX96 >=
sqrtPriceLimitX96
        // sqrtPriceTargetX96 = max(sqrtPriceNextX96, sqrtPriceLimitX96)
        // when zeroForOne == false, nextOrLimit reduces to sqrtPriceNextX96 <
sqrtPriceLimitX96
        // sqrtPriceTargetX96 = min(sqrtPriceNextX96, sqrtPriceLimitX96)
        let nextOrLimit := xor(lt(sqrtPriceNextX96, sqrtPriceLimitX96), zeroForOne)
        let symDiff := xor(sqrtPriceNextX96, sqrtPriceLimitX96)
        sqrtPriceTargetX96 := xor(sqrtPriceLimitX96, mul(symDiff, nextOrLimit))
    }
}
```

Figure 5.1: The body of the `getSqrtPriceTarget` function in the Bunni v2 `SwapMath` contract, showing the 160-bit values being used directly (`src/lib/SwapMath.sol#L19–L34`)

```
function getSqrtPriceTarget(bool zeroForOne, uint160 sqrtPriceNextX96, uint160
sqrtPriceLimitX96)
    internal
    pure
    returns (uint160 sqrtPriceTargetX96)
{
    assembly ("memory-safe") {
        // a flag to toggle between sqrtPriceNextX96 and sqrtPriceLimitX96
```

```
        // when zeroForOne == true, nextOrLimit reduces to sqrtPriceNextX96 >=
sqrtPriceLimitX96
        // sqrtPriceTargetX96 = max(sqrtPriceNextX96, sqrtPriceLimitX96)
        // when zeroForOne == false, nextOrLimit reduces to sqrtPriceNextX96 <
sqrtPriceLimitX96
        // sqrtPriceTargetX96 = min(sqrtPriceNextX96, sqrtPriceLimitX96)
        sqrtPriceNextX96 := and(sqrtPriceNextX96,
0xffffffffffffffffffffffffffffffffffffffff)
        sqrtPriceLimitX96 := and(sqrtPriceLimitX96,
0xffffffffffffffffffffffffffffffffffffffff)
        let nextOrLimit := xor(lt(sqrtPriceNextX96, sqrtPriceLimitX96),
and(zeroForOne, 0x1))
        let symDiff := xor(sqrtPriceNextX96, sqrtPriceLimitX96)
        sqrtPriceTargetX96 := xor(sqrtPriceLimitX96, mul(symDiff, nextOrLimit))
    }
}
```

Figure 5.2: The body of the `getSqrtPriceTarget` function in the Uniswap v4 `SwapMath` contract, showing the high bits of the values being cleared before being used (`src/libraries/SwapMath.sol#L20–L37`)

Recommendations

Short term, update the `getSqrtPriceTarget` function to clear the higher-order bits before using them.

Long term, regularly review the sources of any modified third-party code to ensure you use the most up-to-date version and benefit from any security improvements.

6. Rebalance mechanism access control can be bypassed

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BUNNI-6
Target: `src/BunniHook.sol`, `src/lib/BunniHookLogic.sol`	

Description

The access controls of the rebalance mechanism are ineffective, allowing anyone to execute the pre- and post-hooks of the mechanism.

Once a rebalance order is created, users can fulfill this order by calling one of the `fulfillOrder` functions of the `FloodPlain` contract. The functions will make a call to the `BunniZone`, if it is defined, in order to validate the caller, as shown in figure 6.1:

```
function fulfillOrder(SignedOrder calldata package) external payable nonReentrant {
    Order calldata order = package.order;

    bytes32 orderHash = order.hash();

    // Check zone accepts the fulfiller. Fulfiller is msg.sender in direct fills.
    if (order.zone != address(0)) if (!(IZone(order.zone).validate(order,
msg.sender))) revert ZoneDenied();

    // Execute pre hooks.
    order.preHooks.execute();

    // Transfer each offer item to msg.sender using Permit2.
    _permitTransferOffer(order, package.signature, orderHash, msg.sender);

    // Transfer consideration item from msg.sender to offerer.
    uint256 amount = order.consideration.amount;
    IERC20(order.consideration.token).safeTransferFrom(msg.sender, order.recipient,
amount);

    // Execute post hooks.
    order.postHooks.execute();

    // Emit an event signifying that the order has been fulfilled.
    emit OrderFulfilled(orderHash, order.zone, msg.sender, amount);
}
```

Figure 6.1: One of the `fulfillOrder` functions (`flood-contracts/master/src/FloodPlain.sol#L55-L78`)

However, as figure 6.2 shows, the BunniZone does not perform any validation on the contents of the order, but only on the fulfiller (`msg.sender` in the highlighted line of figure 6.1):

```
function validate(IFloodPlain.Order calldata order, address fulfiller) external view
returns (bool) {
    // extract PoolKey from order's preHooks
    IBunniHook.RebalanceOrderHookArgs memory hookArgs =
        abi.decode(order.preHooks[0].data[4:], (IBunniHook.RebalanceOrderHookArgs));
    PoolKey memory key = hookArgs.key;
    PoolId id = key.toId();

    // query the hook for the am-AMM manager
    IAmAmm amAmm = IAmAmm(address(key.hooks));
    IAmAmm.Bid memory topBid = amAmm.getTopBid(id);

    // allow fulfiller if they are whitelisted or if they are the am-AMM manager
    return isWhitelisted[fulfiller] || topBid.manager == fulfiller;
}
```

Figure 6.2: The `validate` function (src/BunniZone.sol#L41–L54)

This allows anyone to execute the pre- and post-hooks by creating a fake order and then inserting a real order into the `preHooks` and `postHooks` arrays.

Additionally, the BunniHook contract considers the `poolId` of the created order combined with the order hash to be a valid permit signature, as visible in figures 6.3 and 6.4:

```
function _createRebalanceOrder(
    HookStorage storage s,
    Env calldata env,
    PoolId id,
    PoolKey memory key,
    uint16 rebalanceOrderTTL,
    Currency inputToken,
    Currency outputToken,
    uint256 inputAmount,
    uint256 outputAmount
) internal {
    // ...

    // approve input token to permit2
    if (inputERC20Token.allowance(address(this), env.permit2) < inputAmount) {
        address(inputERC20Token).safeApproveWithRetry(env.permit2,
type(uint256).max);
    }

    // etch order so fillers can pick it up
    // use PoolId as signature to enable isValidSignature() to find the correct
order hash
    IOnChainOrders(address(env.floodPlain)).etchOrder(
```

```
        IFloodPlain.SignedOrder({order: order, signature: abi.encode(id)})
    );
}
```

Figure 6.3: The rebalance order signature creation in the `_createRebalanceOrder` function (src/lib/BunniHookLogic.sol#L733–L743)

```
function isValidSignature(bytes32 hash, bytes memory signature)
    external
    view
    override
    returns (bytes4 magicValue)
{
    // verify rebalance order
    PoolId id = abi.decode(signature, (PoolId)); // we use the signature field to
store the pool id
    if (s.rebalanceOrderHash[id] == hash) {
        return this.isValidSignature.selector;
    }
}
```

Figure 6.4: The rebalance order signature verification in the `isValidSignature` function (src/BunniHook.sol#L120–L131)

Since the `FloodPlain` contract allows arbitrary functions to be executed on arbitrary targets, this contract has a high-severity vulnerability where a call to `permit2` can be inserted into the `preHooks` or `postHooks` array in order to consume the `permit2` coupon for other users and steal their tokens.

Due to this, anyone can drain the balance of the `BunniHook` contract for any token that has an outstanding rebalance order by inserting a call to `permit2` into the `preHooks` or `postHooks` arrays with the correct `witness`, `witnessTypeString`, and `poolId` variables but a changed receiver address.

Exploit Scenario

A pool contains 10,000 USDC and 20,000 USDT, and a rebalance order is created to sell 5,000 USDT and get 5,000 USDC. Eve creates a fake order that contains the actual rebalance pre- and post-hook in the `preHooks` array, but leaves the `postHooks` array empty. The output amount is pulled from the `BunniHub` contract and stays in the `BunniHook` contract. Eve then creates another fake order and inserts a call to the `permit2` contract's `permitWitnessTransferFrom` function in the `preHooks` array to transfer the amount to herself. This passes since the signature is considered valid whenever the correct hash and `PoolId` are supplied to the `isValidSignature` function and since the `permit2` call was made from the `FloodPlain` contract.

Recommendations

Short term, do not deploy any pools with rebalancing enabled until Flood addresses this issue and undergoes a security review. Otherwise, consider either creating custom

wrappers for interaction with the `FloodPlain` contract that would validate the order, caller, and the hooks at once, or using a different provider for order fulfillment.

Long term, redesign the rebalance flow in order to connect the validation and execution of the rebalance actions. Ensure that this feature is thoroughly tested for common adversarial situations.

7. Pools can be drained via the rebalance mechanism by selectively executing the rebalanceOrderPreHook and the rebalanceOrderPostHook

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-7
Target: `src/BunniHook.sol`, `src/lib/BunniHookLogic.sol`	

Description

Any pool that has an active rebalance order can be drained because the `BunniHook` does not enforce that a `rebalanceOrderPreHook` and `rebalanceOrderPostHook` need to be executed together.

When a rebalance order is fulfilled by using the `fulfillOrder` function of the `FloodPlain` contract, the actual data passed to the `rebalanceOrderPreHook` and `rebalanceOrderPostHook` functions is arbitrary, as shown in figures 7.1 and 7.2:

```
function fulfillOrder(SignedOrder calldata package) external payable nonReentrant {
    Order calldata order = package.order;

    bytes32 orderHash = order.hash();

    // Check zone accepts the fulfiller. Fulfiller is msg.sender in direct fills.
    if (order.zone != address(0)) if (!(IZone(order.zone).validate(order,
msg.sender))) revert ZoneDenied();

    // Execute pre hooks.
    order.preHooks.execute();

    // Transfer each offer item to msg.sender using Permit2.
    _permitTransferOffer(order, package.signature, orderHash, msg.sender);

    // Transfer consideration item from msg.sender to offerer.
    uint256 amount = order.consideration.amount;
    IERC20(order.consideration.token).safeTransferFrom(msg.sender, order.recipient,
amount);

    // Execute post hooks.
    order.postHooks.execute();

    // Emit an event signifying that the order has been fulfilled.
    emit OrderFulfilled(orderHash, order.zone, msg.sender, amount);
}
```

Figure 7.1: The `fulfillOrder` function (flood-contracts/master/src/FloodPlain.sol#L55–L78)

```
function execute(IFloodPlain.Hook calldata hook) internal {
    address target = hook.target;
    bytes calldata data = hook.data;

    bytes28 extension;
    assembly ("memory-safe") {
        extension := shl(32, calldataload(data.offset))
    }
    require(extension != SELECTOR_EXTENSION, "MALICIOUS_CALL");

    assembly ("memory-safe") {
        let fmp := mload(0x40)
        calldatacopy(fmp, data.offset, data.length)
        if iszero(call(gas(), target, 0, fmp, data.length, 0, 0)) {
            returndatacopy(0, 0, returndatasize())
            revert(0, returndatasize())
        }
    }
}
```

Figure 7.2: The `execute` function called on the `preHooks` and `postHooks` array elements (flood-contracts/master/src/lib/Hooks.sol#L9–L27)

The `rebalanceOrderPreHook` and `rebalanceOrderPostHook` functions do check that the data provided matches the hash of both order hooks in the respective functions, as shown in figures 7.3 and 7.4:

```
IBunniHook.RebalanceOrderHookArgs memory hookArgs =
IBunniHook.RebalanceOrderHookArgs({
    key: key,
    preHookArgs: IBunniHook.RebalanceOrderPreHookArgs({currency: inputToken, amount:
inputAmount}),
    postHookArgs: IBunniHook.RebalanceOrderPostHookArgs({currency: outputToken})
});

// prehook should pull input tokens from BunniHub to BunniHook and update pool
balances
IFloodPlain.Hook[] memory preHooks = new IFloodPlain.Hook[](1);
preHooks[0] = IFloodPlain.Hook({
    target: address(this),
    data: abi.encodeCall(IBunniHook.rebalanceOrderPreHook, (hookArgs))
});

// posthook should push output tokens from BunniHook to BunniHub and update pool
balances
IFloodPlain.Hook[] memory postHooks = new IFloodPlain.Hook[](1);
postHooks[0] = IFloodPlain.Hook({
    target: address(this),
    data: abi.encodeCall(IBunniHook.rebalanceOrderPostHook, (hookArgs))
});
```

Figure 7.3: The hook data creation in the `_createRebalanceOrder` function (src/lib/BunniHookLogic.sol#L696–L714)

```
    // verify call came from Flood
    if (msg.sender != address(floodPlain)) {
        revert BunniHook__Unauthorized();
    }

    // ensure args can be trusted
    if (keccak256(abi.encode(hookArgs)) !=
s.rebalanceOrderHookArgsHash[hookArgs.key.toId()]) {
        revert BunniHook__InvalidRebalanceOrderHookArgs();
    }
```

Figure 7.4: The hook data validation in the `rebalanceOrderPreHook` and `rebalanceOrderPostHook` functions (src/BunniHook.sol#L429–L438 and src/BunniHook.sol#L465–L474)

However, as described in TOB-BUNNI-6, we can provide an arbitrary order and then execute any arbitrary valid pre- and post-hook. Additionally, the functions have no way of guaranteeing that the pre- and post-hook for a particular order are executed sequentially and in the same block.

Due to this, an attacker can execute a valid order's pre-hook to increase the balance of the `BunniHook`, and then drain the balance by executing a malicious pool's post-hook.

Exploit Scenario

A pool contains 10,000 USDC and 20,000 USDT, and a rebalance order is created to sell 5,000 USDT and get 5,000 USDC. Eve creates a pool using the same `BunniHook`, one malicious token, and USDT. She then creates a fake order, signs it, and submits it to the `fulfillOrder` function of the `FloodPlain` contract.

The valid pre-hook of the honest pool is executed first, increasing the `BunniHook` balance by 5,000 USDT; then the malicious post-hook is executed, increasing the balance of Eve's pool by 5,000 USDT. Eve then withdraws the liquidity from her malicious pool and steals the 5,000 USDT. She can repeat this action until she fully drains the USDT balance of the honest pool, and then performs swaps in order to drain the other half of the pool.

Recommendations

Short term, add a stateful mechanism to the execution of pre- and post-hooks to ensure that they must be executed together.

Long term, redesign the rebalance flow to connect the validation and execution of the rebalance actions. Ensure that this feature is thoroughly tested for common adversarial situations.

8. Missing maximum bounds for rebalance parameters

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-BUNNI-8
Target: `src/BunniHook.sol`	

Description

The rebalance parameter validation lacks maximum bounds. This could allow the parameters to be so large that it affects the correct function of the pool.

When a pool and `BunniToken` are deployed by using the `deployBunniToken` function of the `BunniHub` contract, the parameters for rebalancing are validated using the hooks contract, as shown in figure 8.1:

```
function deployBunniToken(HubStorage storage s, Env calldata env,
IBunniHub.DeployBunniTokenParams calldata params)
    external
    returns (IBunniToken token, PoolKey memory key)
{
    //...

    // ensure hook params are valid
    if (address(params.hooks) == address(0)) revert BunniHub__HookCannotBeZero();
    if (!params.hooks.isValidParams(params.hookParams)) revert
BunniHub__InvalidHookParams();
```

Figure 8.1: Validation of the rebalance parameters when deploying a pool (`src/lib/BunniHubLogic.sol#L489–L490`)

If the hooks contract is the `BunniHook` contract, the validation ensures that either all or none of the values are set, as shown in figure 8.2:

```
function isValidParams(bytes calldata hookParams) external pure override returns
(bool) {
    DecodedHookParams memory p = BunniHookLogic.decodeHookParams(hookParams);
    unchecked {
        return (p.feeMin <= p.feeMax) && (p.feeMax < SWAP_FEE_BASE)
            && (p.feeQuadraticMultiplier == 0 || p.feeMin == p.feeMax ||
p.feeTwapSecondsAgo != 0)
            && (p.surgeFee < SWAP_FEE_BASE)
            && (uint256(p.surgeFeeHalfLife) * uint256(p.vaultSurgeThreshold0) *
uint256(p.vaultSurgeThreshold1) != 0)
```

```
            && (
                (
                    p.rebalanceThreshold == 0 && p.rebalanceMaxSlippage == 0 &&
p.rebalanceTwapSecondsAgo == 0
                        && p.rebalanceOrderTTL == 0
                )
                    || (
                        p.rebalanceThreshold != 0 && p.rebalanceMaxSlippage != 0 &&
p.rebalanceTwapSecondsAgo != 0
                            && p.rebalanceOrderTTL != 0
                    )
            ) && (p.oracleMinInterval != 0);
    }
}
```

Figure 8.2: The rebalance parameter validation in the `isValidParams` function (src/BunniHook.sol#L319–L337)

However, there are no maximum bounds placed on these values, other than the inherent bounds of their types. This means that a user can potentially deploy a pool with rebalance parameters that would prevent the pool from working correctly.

Recommendations

Short term, determine reasonable maximum bounds for the `rebalanceMaxSlippage`, `rebalanceTwapSecondsAgo`, and `rebalanceOrderTTL` parameters and enforce them. Clearly document these limits in the user- and developer-facing documentation.

Long term, carefully document all caller-specified values in the system and ensure that they are properly constrained and documented.

9. Excess liquidity can be inflated to create arbitrarily large rebalance orders

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-9
Target: `src/lib/BunniHookLogic.sol`, `src/lib/BunniHubLogic.sol`	

Description

The liquidity density functions used by Bunni v2 pools can be configured to adapt to changes in market conditions. As a result, the protocol also includes a mechanism to autonomously rebalance the holdings of a pool to maximize the liquidity available. However, the way liquidity is added and removed from pools can be leveraged to artificially inflate pool balances temporarily in order to create abnormally large and profitable rebalance orders.

The amount of liquidity in a pool can be calculated by multiplying a token's balance by the overall token density according to the LDF. Since we have two tokens per pool, each with independent balances and densities, we can infer two possible liquidity amounts (figure 9.1). Under normal conditions, these values should be approximately equal, but if the LDF shifts or morphs somehow, the densities of each token may change. Since the balances will remain constant, this means that the liquidity values implied by the balances may differ. If the difference in these liquidity estimates exceeds a pool-configured threshold (figure 9.2), a rebalance order will be issued through `flood.bid` that trades a portion of the excess token to increase the overall liquidity.

```
totalDensity0X96 = density0RightOfRoundedTickX96 + density0OfRoundedTickX96;
totalDensity1X96 = density1LeftOfRoundedTickX96 + density1OfRoundedTickX96;
uint256 totalLiquidityEstimate0 =
    (balance0 == 0 || totalDensity0X96 == 0) ? 0 : balance0.fullMulDiv(Q96,
totalDensity0X96);
uint256 totalLiquidityEstimate1 =
    (balance1 == 0 || totalDensity1X96 == 0) ? 0 : balance1.fullMulDiv(Q96,
totalDensity1X96);
if (totalLiquidityEstimate0 == 0) {
    totalLiquidity = totalLiquidityEstimate1;
} else if (totalLiquidityEstimate1 == 0) {
    totalLiquidity = totalLiquidityEstimate0;
} else {
    totalLiquidity = FixedPointMathLib.min(totalLiquidityEstimate0,
totalLiquidityEstimate1);
}
```

Figure 9.1: A snippet of the `queryLDF` function showing how `totalLiquidity` is calculated (`src/lib/QueryLDF.sol#L72–L84`)

```
// should rebalance if excessLiquidity / totalLiquidity >= 1 / rebalanceThreshold
bool shouldRebalance0 =
    excessLiquidity0 != 0 && excessLiquidity0 >= totalLiquidity /
input.hookParams.rebalanceThreshold;
bool shouldRebalance1 =
    excessLiquidity1 != 0 && excessLiquidity1 >= totalLiquidity /
input.hookParams.rebalanceThreshold;
if (!shouldRebalance0 && !shouldRebalance1) return (false, inputToken, outputToken,
inputAmount, outputAmount);
```

Figure 9.2: A snippet of the `_computeRebalanceParams` function that checks if a rebalance order should be created (`src/lib/BunniHookLogic.sol#L618–L623`)

To simplify the process of adding and removing liquidity after the pool has been initialized with some balance, these operations allow tokens to be added or removed from the pool only in proportion to the current balances (figure 9.3). Note that these balances include any excess liquidity; as a result, the overall proportion of excess liquidity will remain the same when liquidity is added or removed.

```
(returnData.balance0, returnData.balance1) =
    (inputData.state.rawBalance0 + reserveBalance0, inputData.state.rawBalance1 +
reserveBalance1);

// update TWAP oracle and optionally observe
bool requiresLDF = returnData.balance0 == 0 && returnData.balance1 == 0;

if (requiresLDF) {
    ...
} else {
    // already initialized liquidity shape
    // simply add tokens at the current ratio
    // need to update: reserveAmount0, reserveAmount1, amount0, amount1

    // compute amount0 and amount1 such that the ratio is the same as the current
ratio
    uint256 amount0Desired = inputData.params.amount0Desired;
    uint256 amount1Desired = inputData.params.amount1Desired;
    uint256 balance0 = returnData.balance0;
    uint256 balance1 = returnData.balance1;

    returnData.amount0 = balance1 == 0
        ? amount0Desired
        : FixedPointMathLib.min(amount0Desired, amount1Desired.mulDiv(balance0,
balance1));
    returnData.amount1 = balance0 == 0
        ? amount1Desired
        : FixedPointMathLib.min(amount1Desired, amount0Desired.mulDiv(balance1,
balance0));
```

Figure 9.3: A snippet of the `_depositLogic` function that shows how the token amounts to be deposited are calculated for initialized pools (`src/lib/BunniHubLogic.sol#L226–L312`)

Since rebalance orders are emitted automatically after a swap as a result of pool/market conditions, a malicious fulfiller can anticipate when this will occur and inject a large amount of liquidity before the rebalance order is calculated, remove the liquidity after the order is issued, then fill the order himself, profiting from the slippage tolerance (figure 9.4) and leaving the pool in an incredibly unbalanced state.

```
bool willRebalanceToken0 = shouldRebalance0 && (!shouldRebalance1 ||
excessLiquidity0 > excessLiquidity1);

// compute target amounts (i.e. the token amounts of the excess liquidity)
uint256 excessLiquidity = willRebalanceToken0 ? excessLiquidity0 : excessLiquidity1;
uint256 targetAmount0 = excessLiquidity.fullMulDiv(totalDensity0X96, Q96);
uint256 targetAmount1 = excessLiquidity.fullMulDiv(totalDensity1X96, Q96);

// determine input & output
(inputToken, outputToken) = willRebalanceToken0
    ? (input.key.currency0, input.key.currency1)
    : (input.key.currency1, input.key.currency0);
uint256 inputTokenExcessBalance =
    willRebalanceToken0 ? balance0 - currentActiveBalance0 : balance1 -
currentActiveBalance1;
uint256 inputTokenTarget = willRebalanceToken0 ? targetAmount0 : targetAmount1;
uint256 outputTokenTarget = willRebalanceToken0 ? targetAmount1 : targetAmount0;
if (inputTokenExcessBalance < inputTokenTarget) {
    // should never happen
    return (false, inputToken, outputToken, inputAmount, outputAmount);
}
inputAmount = inputTokenExcessBalance - inputTokenTarget;
outputAmount = outputTokenTarget.mulDivUp(1e5 -
input.hookParams.rebalanceMaxSlippage, 1e5);
```

Figure 9.4: A snippet of the `_computeRebalanceParams` function that shows the calculation of the amounts of tokens for the rebalance order (`src/lib/BunniHookLogic.sol#L652–L672`)

If there is an active `AmAmm` manager, withdrawals would be subject to a timeout window, but the version of the code under review has a known issue that allows this to be bypassed. Since the `AmAmm` manager is also included on the allowlist of fulfillers, this would allow arbitrary attackers to fulfill orders provided they win the `AmAmm` auction 24 hours in advance. Otherwise, any allowlisted fulfiller would be able to execute this attack.

Additionally, regular depositors could use flash loans to provide just-in-time liquidity to inflate the rebalance order to require a greater amount of tokens than would naturally be in the pool, requiring fulfillers to also take the extra step of providing just-in-time liquidity back into the pool to carry out the rebalance, resulting in a swap that is less profitable than it appeared based on the order parameters alone.

Exploit Scenario

A pool's LDF shifts and has 120 TokenA and 100 TokenB, but the active balance of TokenA is only 100. Alice, a malicious fulfiller, notices that the pool will rebalance and deposits 1,080

TokenA and 900 TokenB so the pool has 1,200 TokenA and 1,000 TokenB in total. She makes a small swap to trigger the creation of a rebalance order. This pool has a 5% rebalance slippage tolerance so the rebalance order will sell 100 TokenA for 95 TokenB. Alice withdraws all of her original liquidity and immediately fills the rebalance order. As a result, the pool has 20 TokenA and 195 TokenB. If Alice had not provided just-in-time liquidity, the pool would have only swapped 10 TokenA for 9.5 TokenB and ended at a total of 110 TokenA and 109.5 TokenB. Since the target ratio for the tokens is currently 1:1, we can assume they have the same price, and see that if each token was worth $1, Alice made an excess profit of $4.50, or just over 2% of the original total pool value.

Recommendations

Short term, add a check to enforce the withdrawal queue when `rebalanceOrderDeadline` is in the future.

Long term, consider changing the add/remove liquidity flows to account for excess liquidity or adding checks to the rebalance hooks that enforce that the pool is in an improved state after the rebalance.

10. Insufficient event generation	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-BUNNI-10
Target: `bunni-v2/src/*`, `biddog/src/AmAmm.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; malfunctioning contracts and attacks could go undetected.

The following operations should trigger events:

- `claimReferralRewards` (`bunni-v2/src/BunniTokens.sol#L155-L196`)
- `distributeReferralRewards` (`bunni-v2/src/BunniTokens.sol#L132-L152`)
- `claimProtocolFees` (`bunni-v2/src/BunniHook.sol#L244-L246`)
- `rebalanceOrderPreHook` (`bunni-v2/src/BunniHook.sol#L429-L462`)
- `rebalanceOrderPostHook` (`bunni-v2/src/BunniHook.sol#L465-L511`)
- `_rebalance` (`bunni-v2/src/lib/BunniHookLogic.sol#L532-L550`)
- `updateStateMachine` (`biddog/src/AmAmm.sol#L432-L434`)

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.

11. AmAmm manager can manipulate TWAP prices without risk

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BUNNI-11
Target: `biddog/src/AmAmm.sol`	

Description

The `AmAmm` manager of a pool can manipulate the TWAP price of the pool by swapping a large amount of assets and then setting the swap fee to a very high value in order to prevent others from arbitraging the pool.

The `AmAmm` manager of a pool can set the swap fees of this pool by calling the `setBidPayload` function of the `AmAmm` contract, as shown in figure 11.1:

```
function setBidPayload(PoolId id, bytes7 payload, bool topBid) external virtual
override {
    address msgSender = LibMulticaller.senderOrSigner();

    if (!_amAmmEnabled(id)) {
        revert AmAmm__NotEnabled();
    }

    // update state machine
    _updateAmAmmWrite(id);

    Bid storage relevantBid = topBid ? _topBids[id] : _nextBids[id];

    if (msgSender != relevantBid.manager) {
        revert AmAmm__Unauthorized();
    }

    if (!_payloadIsValid(id, payload)) {
        revert AmAmm__InvalidBid();
    }

    relevantBid.payload = payload;

    emit SetBidPayload(id, msgSender, payload, topBid);
}
```

Figure 11.1: Function to set the payload used for this pool (`biddog/src/AmAmm.sol#L406-L429`)

The payload they set contains the swap fee parameter for each direction of the swap and a Boolean flag that determines if the pool will charge surge fees, as shown in figure 11.2:

```
function decodeAmAmmPayload(bytes7 payload)
    pure
    returns (uint24 swapFee0For1, uint24 swapFee1For0, bool enableSurgeFee)
{
    swapFee0For1 = uint24(bytes3(payload));
    swapFee1For0 = uint24(bytes3(payload << 24));
    enableSurgeFee = uint8(payload[6]) != 0;
}
```

Figure 11.2: The parameters contained in the AmAmm payload (src/lib/AmAmmPayload.sol#L11–L18)

Once a swap is initiated and the `beforeSwap` function of the `BunniHook` contract is called, the function will determine the swap fees charged based on the bid payload of the current manager (if a manager is set), as shown in figure 11.3:

```
// update am-AMM state
uint24 amAmmSwapFee;
bool amAmmEnableSurgeFee;
if (hookParams.amAmmEnabled) {
    bytes7 payload;
    IAmAmm.Bid memory topBid = IAmAmm(address(this)).getTopBidWrite(id);
    (amAmmManager, payload) = (topBid.manager, topBid.payload);
    uint24 swapFee0For1;
    uint24 swapFee1For0;
    (swapFee0For1, swapFee1For0, amAmmEnableSurgeFee) = decodeAmAmmPayload(payload);
    amAmmSwapFee = params.zeroForOne ? swapFee0For1 : swapFee1For0;
}
```

Figure 11.3: A code snippet of the beforeSwap function (src/lib/BunniHookLogic.sol#L295–L305)

However, a malicious `AmAmm` manager could use this mechanism to protect themselves from arbitrage while they are manipulating the TWAP price. It is important to note that the `BunniHook` defines a maximum swap fee that can be used; however, this value could be misconfigured or naively set to a high value. Additionally, the use of a truncated Oracle makes this manipulation take longer since the TWAP price will gradually update over several blocks.

Exploit Scenario

Alice deploys a pool with a `BunniHook` that has 100% as the maximum swap fee. Her pool's TWAP price is used in an external lending protocol to determine asset prices. Eve creates a large bid and becomes the `AmAmm` manager of this pool. She then proceeds to borrow a large amount of assets to manipulate the TWAP price over several blocks and sets the swap fees to 100% to protect her manipulation against arbitrage. Eve drains the lending pool that overvalued her assets due to the inflated TWAP price.

Recommendations

Short term, consider determining and setting a reasonable max swap fee upper bound in the `BunniHook` contract.

Long term, create developer- and user-facing documentation that clearly outlines the risks of different pool and hook configurations, as well as the powers of the `AmAmm` manager.

12. Lack of zero-value checks

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-12
Target: `src/BunniHub`, `src/BunniHook`, `src/lib/BunniHookLogic`	

Description
Certain functions fail to validate incoming arguments, so callers of these functions could mistakenly set important state variables to a zero value, misconfiguring the system.

For example, the constructor in the `BunniHub` contract sets the `poolManager`, `weth`, and `bunniTokenImplementation` variables, which store the addresses of external contracts Bunni v2 relies on:

```
constructor(
    IPoolManager poolManager_,
    WETH weth_,
    IPermit2 permit2_,
    IBunniToken bunniTokenImplementation_,
    address initialOwner
) Permit2Enabled(permit2_) {
    poolManager = poolManager_;
    weth = weth_;
    bunniTokenImplementation = bunniTokenImplementation_;
    _initializeOwner(initialOwner);
}
```

Figure 12.1: The constructor of the `BunniHub` contract (`src/BunniHub.sol#L75–L86`)

If `weth` is set to a zero value, this deployment of the Bunni v2 protocol would be unable to handle native ether. Since all of the variables mentioned above are tagged as immutable, the contract will have to be redeployed to update to the correct address. This misconfiguration may not be noticed immediately, and forcing LPs to migrate would create a poor user experience.

The following functions are missing zero-value checks:

- `BunniHook.constructor`
- The `inputAmount` returned by the call to `BunniSwapMath.computeSwap` in `BunnyHookLogic.beforeSwap` for an `exactOut` swap

Recommendations

Short term, add zero-value checks to all function arguments to ensure that callers cannot set incorrect values, misconfiguring the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

13. Lack of systematic approach to rounding and arithmetic errors

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-13
Target: `bunni-v2/src/*`	

Description

While reviewing the codebase, we noted several areas with seemingly excessive input validation that may hint at underlying issues stemming from improper rounding directions or other arithmetic errors. While we did not identify root causes or ways to exploit these instances, they warrant further investigation and testing. If the root cause is determined to be truly benign, it should be documented.

Operations that suggest a lack of a systematic approach to rounding and arithmetic errors include the following:

- The `computeSwap` function of the `BunniSwapMath` library computes the `outputAmount` multiple times, which indicates that there is a rounding or arithmetic error that is improperly handled:

```
// compute first pass result
(updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount) = _computeSwap(input,
amountSpecified);

// ensure that the output amount is lte the output token balance
if (outputAmount > outputTokenBalance) {
    // exactly output the output token's balance
    // need to recompute swap
    amountSpecified = outputTokenBalance.toInt256();
    (updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount) =
_computeSwap(input, amountSpecified);

    if (outputAmount > outputTokenBalance) {
        // somehow the output amount is still greater than the balance due to
rounding errors
        // just set outputAmount to the balance
        outputAmount = outputTokenBalance;
    }
}
```

Figure 13.1: A snippet of the `computeSwap` function that may result in two trial swaps before a cap is applied to `outputAmount` (`src/lib/BunniSwapMath.sol#L62–L77`)

- The `_computeSwap` function of the `BunniSwapMath` library allows a user to get up to 2 wei of tokens for free:

```
if (exactIn) {
    uint256 inputAmountSpecified = uint256(-amountSpecified);
    if (inputAmount > inputAmountSpecified && inputAmount < inputAmountSpecified +
3) {
        // if it's an exact input swap and inputAmount is greater than the specified
input amount by 1 or 2 wei,
        // round down to the specified input amount to avoid reverts. this assumes
that it's not feasible to
        // extract significant value from the pool if each swap can at most extract
2 wei.
        inputAmount = inputAmountSpecified;
    }
}
```

Figure 13.2: A snippet of the `_computeSwap` function that shows the input amount being rounded down in certain situations, favoring the user instead of the pool (src/lib/BunniSwapMath.sol#L336–L344)

- The `_computeRebalanceParams` function of the `BunniHookLogic` library has a case where both tokens have excess liquidity. This indicates a rounding or arithmetic error in the way excess liquidity is computed:

```
// decide which token will be rebalanced (i.e., sold into the other token)
bool willRebalanceToken0 = shouldRebalance0 && (!shouldRebalance1 ||
excessLiquidity0 > excessLiquidity1);
```

Figure 13.3: A snippet of the `_computeRebalanceParams` function (src/lib/BunniHookLogic.sol#L651–L652)

- The token densities in the `queryLDF` function are rounded down. However, the density is later used as a denominator, which can result in the `totalLiquidityEstimates` rounding up. The `getAmountsForLiquidity` function always rounds down throughout the codebase, which may be incorrect in some cases, as shown in figure 13.4:

```
(uint256 density0OfRoundedTickX96, uint256 density1OfRoundedTickX96) =
LiquidityAmounts.getAmountsForLiquidity(
    sqrtPriceX96, roundedTickSqrtRatio, nextRoundedTickSqrtRatio,
uint128(liquidityDensityOfRoundedTickX96), false
);
totalDensity0X96 = density0RightOfRoundedTickX96 + density0OfRoundedTickX96;
totalDensity1X96 = density1LeftOfRoundedTickX96 + density1OfRoundedTickX96;
uint256 totalLiquidityEstimate0 =
    (balance0 == 0 || totalDensity0X96 == 0) ? 0 : balance0.fullMulDiv(Q96,
totalDensity0X96);
```

```
uint256 totalLiquidityEstimate1 =
    (balance1 == 0 || totalDensity1X96 == 0) ? 0 : balance1.fullMulDiv(Q96,
totalDensity1X96);
```

Figure 13.4: A snippet of the queryLDF function (src/lib/QueryLDF.sol#L69–L77)

- If the `cumulativeAmounts0`/`cumulativeAmounts1` functions round down, the resulting excess liquidity is rounded up since the cumulative amounts are used as the denominator:

```
uint256 excessLiquidity0 = balance0 > currentActiveBalance0
    ? (balance0 - currentActiveBalance0).divWad(
        bunniState.liquidityDensityFunction.cumulativeAmount0(
            input.key,
            minUsableTick,
            WAD,
            input.arithmeticMeanTick,
            input.updatedTick,
            bunniState.ldfParams,
            input.newLdfState
        )
    )
    : 0;
uint256 excessLiquidity1 = balance1 > currentActiveBalance1
    ? (balance1 - currentActiveBalance1).divWad(
        bunniState.liquidityDensityFunction.cumulativeAmount1(
            input.key,
            maxUsableTick,
            WAD,
            input.arithmeticMeanTick,
            input.updatedTick,
            bunniState.ldfParams,
            input.newLdfState
        )
    )
    : 0;
```

Figure 13.5: A snippet of the _computeRebalanceParams function (src/lib/BunniHookLogic.sol#L591–L616)

Recommendations

Short term, review the system arithmetic and devise a systematic approach to rounding, ensuring rounding always favors the protocol. Implement smart contract fuzzing to determine the relative error bounds of each operation.

Long term, explore whether exposing the rounding direction as an explicit parameter in higher-level functions may help to prevent these types of issues.

14. Native assets deposited to pools with no native currencies are lost

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BUNNI-14
Target: `src/lib/BunniHubLogic.sol`	

Description

Native assets deposited to pools with no native currencies are lost.

The `deposit` function of the `BunniHub` contract is payable since it needs to handle native assets if one of the tokens of the pool is a native token, as shown in figure 14.1:

```
function deposit(DepositParams calldata params)
    external
    payable
    virtual
    override
    nonReentrant
    checkDeadline(params.deadline)
    returns (uint256 shares, uint256 amount0, uint256 amount1)
{
    return BunniHubLogic.deposit(
        s,
        BunniHubLogic.Env({
            weth: weth,
            permit2: permit2,
            poolManager: poolManager,
            bunniTokenImplementation: bunniTokenImplementation
        }),
        params
    );
}
```

Figure 14.1: The payable `deposit` function in `BunniHub` (`src/BunniHub.sol#L93–L112`)

If a user provides an excess of native assets, the function will refund the difference back to the user, as shown in figure 14.2:

```
if (params.poolKey.currency0.isNative()) {
    if (address(this).balance != 0) {
        params.refundRecipient.safeTransferETH(
            FixedPointMathLib.min(address(this).balance, msg.value - amount0)
        );
    }
```

```
} else if (params.poolKey.currency1.isNative()) {
    if (address(this).balance != 0) {
        params.refundRecipient.safeTransferETH(
            FixedPointMathLib.min(address(this).balance, msg.value - amount1)
        );
    }
}
```

Figure 14.2: A code snippet of the `deposit` function in `BunniHubLogic` (`src/lib/BunniHubLogic.sol#L172–L184`)

However, if the pool's currencies are both non-native and a user mistakenly provides a nonzero `msg.value`, this value will not be refunded and will remain in the `BunniHub`. Although this value can be withdrawn by deploying a new pool and abusing some of the calls, there is no direct and easy way to do this.

Recommendations

Add a check to the `deposit` function of `BunniHub` that `msg.value` is zero if both pool tokens are non-native; otherwise, have the function revert.

15. Users can gain free tokens through the BunniSwap swap functionality

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BUNNI-15
Target: `src/lib/BunniSwapMath.col`	

Description

During token swaps, users can receive a nonzero amount of output tokens even when they provide zero input tokens, allowing them to acquire free tokens without contributing any input tokens for the swap.

The `computeSwap` function in the `BunniSwapMath` library executes swap operations based on a user-specified token `amountSpecified` as one of the inputs to the function. As shown in figure 15.1, depending on whether the `amountSpecified` is negative or positive, the swap is configured to be an `ExactIn` or `ExactOut` swap.

```
// initialize input and output amounts based on initial info
bool exactIn = amountSpecified < 0;
inputAmount = exactIn ? uint256(-amountSpecified) : 0;
outputAmount = exactIn ? 0 : uint256(amountSpecified);
```

Figure 15.1: Snippet of the function `_computeSwap` showing the computation of `ExactIn` and `ExactOut` (`src/lib/BunniSwapMath.sol#L101–L104`)

As shown in figure 15.2, after a swap occurs, `computeSwap` returns the updated pool state and the amount of input tokens swapped by the user for the output tokens. The expectation is that, given a valid pool state, if the `amountSpecified` is nonzero, the swap should proceed, and if the output token amount is nonzero, the input token amount should also not be zero. In other words, users should not be able to acquire tokens for free. However, even if the user provides a nonzero `amountSpecified` during the swap, the function can still result in zero input tokens provided by the user while returning a nonzero amount of output tokens, enabling users to gain tokens for free during the swap.

```
function computeSwap(BunniComputeSwapInput calldata input,
uint256 balance0, uint256 balance1)
    external
    view
    returns (uint160 updatedSqrtPriceX96, int24 updatedTick,
    uint256 inputAmount, uint256 outputAmount){
    uint256 outputTokenBalance = input.swapParams.zeroForOne ? balance1 : balance0;
    int256 amountSpecified = input.swapParams.amountSpecified;
```

```
    ...
}
```

Figure 15.2: Snippet of the `computeSwap` function in the library `BunniSwapMath` (`src/lib/BunniSwapMath.sol#L49–78`)

We found this issue with Medusa during the invariant testing process.

Recommendations

Short term, triage this issue's underlying root cause and deploy a fix such that swap functionality behaves as expected.

Long term, consider integrating Medusa to generate invariants at both the function and system levels, enabling the testing of operations that involve complex mathematics and liquidity pricing curves.

16. Users can gain tokens during round-trip swaps

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-16
Target: `src/lib/BunniSwapMath.col`	

Description

The swap functionality of `BunniSwapMath` library is not accurately implemented, allowing users to gain tokens during round-trip swaps (i.e., swapping `token0` for `token1` and then swapping the same amount of `token1` for `token0`).

The `computeSwap` function in the `BunniSwapMath` library allows users to swap tokens on a valid pool state. Given an `amountSpecified` and the swap direction, the function calculates the input tokens exchanged for the output tokens. In a `zeroForOne` swap, the user exchanges `token0` for `token1`, and the pool state is updated to reflect the new token balances and price ratio, as shown in figure 16.1.

```
function computeSwap(BunniComputeSwapInput calldata input,
uint256 balance0, uint256 balance1)
    external
    view
    returns (uint160 updatedSqrtPriceX96, int24 updatedTick,
    uint256 inputAmount, uint256 outputAmount)
{
    uint256 outputTokenBalance = input.swapParams.zeroForOne ? balance1 : balance0;
    int256 amountSpecified = input.swapParams.amountSpecified;
    ...
    // compute first pass result
    (updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount) =
    _computeSwap(input, amountSpecified);
    ...
}
```

Figure 16.1: Snippet of the `computeSwap` function in the `BunniSwapMath` library showing the updated pool state (`src/lib/BunniSwapMath.sol#L49-78`)

If the user then performs a `OneForZero` swap—exchanging the previously gained `token1` for `token0`—the expectation is that the user should receive the same amount of `token0` they initially provided. However, due to the incorrect implementation of `computeSwap`, the input tokens received after the `OneForZero` swap exceed the expected amount, allowing users to gain tokens during round-trip swaps.

This is, in part, due to the `updatedTick` never being initialized when both branches in the highlighted lines of figure 16.2 are skipped.

```
if (
    (zeroForOne && sqrtPriceLimitX96 <= leastChangeSqrtPriceX96)
        || (!zeroForOne && sqrtPriceLimitX96 >= leastChangeSqrtPriceX96)
) {
    // ...
    if (naiveSwapResultSqrtPriceX96 == sqrtPriceNextX96) {
        // Equivalent to `updatedTick = zeroForOne ? tickNext - 1 : tickNext;`
        unchecked {
            // cannot cast a bool to an int24 in Solidity
            int24 _zeroForOne;
            assembly {
                _zeroForOne := zeroForOne
            }
            updatedTick = tickNext - _zeroForOne;
        }
    } else if (naiveSwapResultSqrtPriceX96 != startSqrtPriceX96) {
        // recompute unless we're on a lower tick boundary (i.e. already
transitioned ticks), and haven't moved
        updatedTick = TickMath.getTickAtSqrtPrice(naiveSwapResultSqrtPriceX96);
    }

    // ...

    return (updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount);
```

Figure 16.1: Snippet of the `_computeSwap` function in the `BunniSwapMath` library (src/lib/BunniSwapMath.sol#L228–L302)

We found this issue with Medusa during the invariant testing process.

Recommendations

Short term, initialize the `updatedTick` to the correct value when the branches in figure 16.2 are skipped. Run the invariant tests again with the updated code and triage any outstanding invariant failures.

Long term, consider integrating Medusa to generate invariants at both the function and system levels, enabling the testing of operations that involve complex mathematics and liquidity pricing curves.

17. Different amount of input/output tokens can be returned in ExactIn and ExactOut configurations during the swap

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-17
Target: `src/lib/BunniSwapMath.col`	

Description

The swap functionality of `BunniSwapMath` can yield different input and output token amounts on the same pool state, depending on the swap configuration (`ExactIn` or `ExactOut`). This enables users to provide fewer tokens by choosing one configuration over the other.

The `computeSwap` function in the `BunniSwapMath` library enables users to swap tokens on a valid pool state. Based on the `amountSpecified` and swap direction, it determines the input tokens exchanged for output tokens. If the `amountSpecified` is negative, the swap is an `ExactIn` swap, meaning the function calculates the amount of output tokens for an `amountSpecified` input token amount. Conversely, if the `amountSpecified` is positive, the swap is an `ExactOut` swap, where the function determines the required input tokens for an `amountSpecified` output token amount, as shown in the figure 17.1.

```
// initialize input and output amounts based on initial info
bool exactIn = amountSpecified < 0;
inputAmount = exactIn ? uint256(-amountSpecified) : 0;
outputAmount = exactIn ? 0 : uint256(amountSpecified);
```

Figure 17.1: Snippet of the function `_computeSwap` showing the computation of `ExactIn` and `ExactOut` (`src/lib/BunniSwapMath.sol#L101–L104`)

The expectation is that, for the same pool state, both swap configurations (`ExactIn` and `ExactOut`) should result in the same amounts of input and output tokens. However, the `computeSwap` function deviates from this expected behavior. For example, in an `ExactIn` configuration, `computeSwap` may exchange x amount of input tokens for y output tokens, but in an `ExactOut` configuration, it may exchange x' input tokens for the same y output tokens, where x' is smaller than x. This allows users to provide fewer input tokens in the `ExactOut` configuration for the same amount of y tokens compared to an `ExactIn` swap.

We found this issue with Medusa during the invariant testing process.

Recommendations

Short term, triage the underlying root cause of the above finding and deploy a fix such that `ExactIn` and `ExactOut` swaps behave as expected.

Long term, consider integrating Medusa to generate invariants at both the function and system levels, enabling the testing of operations that involve complex mathematics and liquidity pricing curves.

18. BunniSwap swap functionality can cause panics during the swap

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BUNNI-18
Target: `src/lib/BunniSwapMath.col`	

Description

The swap functionality of `BunniSwapMath` can result in arithmetic underflow during the swap on a valid pool state, disrupting swap operations.

The `computeSwap` function in the `BunniSwapMath` library enables users to swap tokens on a valid pool state. Based on the `amountSpecified` and swap direction, it calculates the input tokens exchanged for output tokens and updates the pool's state and pricing ratio at the end of the swap.

```
(inputAmount, outputAmount) = zeroForOne
    ? (
        updatedActiveBalance0 - currentActiveBalance0,
        currentActiveBalance1 < updatedActiveBalance1 ? 0 : currentActiveBalance1 -
updatedActiveBalance1
    )
    : (
        updatedActiveBalance1 - currentActiveBalance1,
        currentActiveBalance0 < updatedActiveBalance0 ? 0 : currentActiveBalance0 -
updatedActiveBalance0
    );
```

Figure 18.1: Snippet of the `_computeSwap` function (`src/lib/BunniSwapMath.sol#L326–334`)

Given a valid pool state and input parameters for the `computeSwap` function, the swap should succeed. However, `BunniSwapMath` deviates from this expected behavior, leading to an arithmetic underflow during the swap. For a `OneForZero` swap, during the calculation of input and output amount, the `updatedActiveBalance` for `token1` becomes less than the `currentActiveBalance` of `token1`; this condition is not accounted for, as highlighted in figure 18.1. This results in an arithmetic underflow that causes the swap operation to revert.

We discovered this issue with Medusa during the invariant testing process.

Recommendations

Short term, triage the underlying root cause of the above finding and deploy a fix such that the swap operation does not cause an underflow.

Long term, consider integrating Medusa to generate invariants at both the function and system levels, enabling the testing of operations that involve complex mathematics and liquidity pricing curves.

19. cumulativeAmount0 can be greater than the cumulative amount computed through inverse functionality for certain LDFs

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BUNNI-19
Target: Various targets	

Description

The cumulative amount of `token0`, when computed through different methods, can vary for certain types of LDFs, leading to incorrect calculations of liquidity densities for `token0`.

Given a `cumulativeAmount0`, the function `inverseCumulativeAmount0`, in the contracts `UniformDistribution` and `CarpetedGeometricDistribution`, computes the rounded tick whose `cumulativeAmount0` is expected to be less than or equal to the given `cumulativeAmount0`.

```
if (exactIn) {
        (inputAmount, outputAmount) = zeroForOne
            ? ...
            : (
                naiveSwapAmountIn + cumulativeAmount - currentActiveBalance1,
                currentActiveBalance0 + naiveSwapAmountOut
                    - input.liquidityDensityFunction.cumulativeAmount0(...)
            );
    } else {
        (inputAmount, outputAmount) = zeroForOne
            ? (
                input.liquidityDensityFunction.cumulativeAmount0(...)
            : ...
    }
    return (updatedSqrtPriceX96, updatedTick, inputAmount, outputAmount);
}
```

Figure 19.1: Snippet of the `_computeSwap` function in the `BunniSwapMath` library (src/lib/BunniSwapMath.sol#L246–L303)

However, both distributions deviate from the expected behavior, as in some cases, the computed `cumulativeAmount0` exceeds the given `cumulativeAmount0`. These `cumulativeAmount0` values are used throughout the swap process to calculate the input and output amounts, as illustrated in figure 19.1, leading to inaccurate swap computations.

We discovered this issue with Medusa during the invariant testing process.

Recommendations

Short term, triage the underlying root cause of the above finding and deploy a fix such that the LDF functionalities behave as expected.

Long term, consider integrating Medusa to generate invariants at both the function and system levels, enabling the testing of operations that involve complex mathematics and liquidity pricing curves.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

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

C. Code Quality Issues

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

- **Constants assigned to an expression including a constant in the `OrderHashMemory` library will be recomputed during runtime.** This will lead to higher gas consumption.
- **The return value of the `_updateAmAmmWrite` function in the `AmAmm` contract is never used.**
- **Unnecessary code duplication in the `_depositUnlockCallback` function of the `BunniHub` contract.** The body of the `if` statements could be moved to an internal helper function to reduce code duplication.
- **The `_stateTransitionWrite` and `_stateTransition` functions of the `AmAmm` contract differ in only one line.** To remove the need for two almost identical implementations, the refund state modification could be conditionally handled in the function that calls `_stateTransitionWrite` based on a return value from the `_stateTransitionWrite` function.
- **The `console2` utility contract from the Forge standard library should not be used in production code.** This contract is imported into the following contracts: `BunniHubLogic`, `BunniSwapMath`, `QueryLDF`, `LibBuyTheDipGeometricDistribution`, `LibCarpetedDoubleGeometricDistribution`, `LibCarpetedGeometricDistribution`, `LibDoubleGeometricDistribution`, `LibGeometricDistribution`, and `LibUniformDistribution`.
- **The `/// @solidity memory-safe-assembly` annotation is used throughout the codebase to mark assembly blocks as adhering to Solidity's memory model.** Code that does not need to support versions of Solidity prior to 0.8.13 should prefer the `assembly ("memory-safe") { ... }` syntax instead.
- **The `if/else` statements in the `observeSingle` and `getSurroundingObservations` functions of the `Oracle` contract could be simplified by removing the `else if/else` statements since execution is already interrupted by a `return` statement.** Examples on how to simplify the functions are provided in figures C.1 and C.2.
- **The unchecked block in the `observeDouble` function of the Oracle contract is useless because unchecked is applied only to arithmetic expressions inside of the body, not to functions.**

- **An old Solidity compiler version (>=0.6.0) and `abicoder v2` is used in the `Errors.sol`, `Constants.sol`, and `SharedStructs.sol` files and the `IBunniHub`, `IBunniZone`, `IBunniQuoter`, `ILiquidityDensityFunction`, and `IBunniToken` interfaces.** This should be updated to a more modern compiler version, such as v0.8.0 or above.

- **The query functions in `LibUniformDistribution` and `LibGeometricDistributrion` contain a lot of duplicate code for calculating the token densities to the left and right of the current rounded tick.** This can be replaced by calling their respective `cumulativeAmount0` and `cumulativeAmount1` functions, similar to how the other LDF libraries handle the calculation.

```
if (target == beforeOrAt.blockTimestamp) {
    // we're at the left boundary
    return beforeOrAt.tickCumulative;
}
if (target == atOrAfter.blockTimestamp) {
    // we're at the right boundary
    return atOrAfter.tickCumulative;
}
// we're in the middle
uint32 observationTimeDelta = atOrAfter.blockTimestamp - beforeOrAt.blockTimestamp;
uint32 targetDelta = target - beforeOrAt.blockTimestamp;
return beforeOrAt.tickCumulative
    + ((atOrAfter.tickCumulative - beforeOrAt.tickCumulative) /
int56(uint56(observationTimeDelta)))
        * int56(uint56(targetDelta));
```

Figure C.1: Example of how to simplify the `observeSingle` function

```
if (beforeOrAt.blockTimestamp == target) {
    // if newest observation equals target, we're in the same block, so we can
ignore atOrAfter
    return (beforeOrAt, atOrAfter);
}
// otherwise, we need to transform
return (beforeOrAt, transform(beforeOrAt, target, tick));
```

Figure C.2: Example of how to simplify the `getSurroundingObservations` function

D. Invariant Testing and Harness Design

When reviewing protocols with a large potential state space, Trail of Bits creates various stateful and stateless fuzz testing harnesses to verify system properties that would be challenging or even impossible to verify using manual review.

We performed automated testing using a set of stateless invariants tested using fuzzing to test the complex math operations such as the swap functionality and liquidity density functions.

Stateless Invariant Testing

Stateless invariant testing involves verifying that invariants of a system are true across various operations, particularly when the system does not maintain internal state between operations. Since the system is stateless, each request's input should be sanitized such that it produces a valid output, regardless of any past history.

We used Medusa, maintained by Trail of Bits. Medusa is used in conjunction with a test harness: a special contract that sits in front of the system under test and is called by the fuzzers to produce transaction sequences that explore the state space, as measured by code coverage. In the following section, we provide some details of the harness we wrote and some unit tests instances that resulted in failures.

Stateless Invariants for Swap Functionality

- **Users should not be able to gain any tokens through round trip swaps:** Figures D.1 and D.2 show our test configuration and unit test reproducer for this invariant. This resulted in issue TOB-BUNNI-16.

```
function compare_swap_with_reverse_swap_with_zeroForOne_vs_oneForZero(int24
tickSpacing, uint64 balance0, uint64 balance1, int64 amountSpecified, uint160
sqrtPriceLimit, int24 tickLower, int24 tickUpper, int24 currentTick) public{
    // Initialize LDF to Uniform distribution
    ldf = ILiquidityDensityFunction(address(new UniformDistribution()));

    // Initialize parameters before swapinng
    BunniSwapMath.BunniComputeSwapInput memory input1 = _compute_swap(tickSpacing,
balance0, balance1, amountSpecified, sqrtPriceLimit, tickLower, tickUpper,
currentTick);

    // Check for exactIn or exactOut
    // amountSpecifiec negative means exactIn, so zeroForOne needs to be true
    input1.swapParams.zeroForOne = true;
    try this.swap(input1, balance0, balance1) returns (uint160 updatedSqrtPriceX96,
int24 updatedTick, uint256 inputAmount0, uint256 outputAmount0) {
```

```
        require(outputAmount0 != 0 && inputAmount0 !=0);
        if (inputAmount0 != uint64(inputAmount0) || outputAmount0 !=
uint64(outputAmount0))
            return;

        BunniSwapMath.BunniComputeSwapInput memory input2 =
_compute_swap(tickSpacing, uint64(balance0+inputAmount0),
uint64(balance1-outputAmount0), -amountSpecified, sqrtPriceLimit, tickLower,
tickUpper, updatedTick);
        input2.swapParams.amountSpecified = amountSpecified < 0 ?
-int256(outputAmount0) : int256(inputAmount0);
        input2.swapParams.zeroForOne = false;

        (uint160 updatedSqrtPriceX960, int24 updatedTick0,uint256 inputAmount1,
uint256 outputAmount1) = BunniSwapMath.computeSwap(input2,
uint64(balance0+inputAmount0), uint64(balance1-outputAmount0));

        if (amountSpecified < 0){
            assertWithMsg(inputAmount0 >= outputAmount1, "Round trips swaps are
profitable");
        }
        else{
            assertWithMsg(outputAmount0 <= inputAmount1, "Round trips swaps are
profitable");
        }
    }
    catch Panic(uint /*errorCode*/) {
        // This is executed in case of a panic,
        // i.e. a serious error like division by zero
        // or overflow. The error code can be used
        // to determine the kind of error.
        return;
    }
    catch(bytes memory reason) {
       emit LogBytes(reason);
       return;
    }
}

function swap(BunniSwapMath.BunniComputeSwapInput calldata input, uint256 balance0,
uint256 balance1) public returns (uint160 updatedSqrtPriceX96, int24 updatedTick,
uint256 inputAmount0, uint256 outputAmount0) {
    require(msg.sender == address(this));
    (updatedSqrtPriceX96, updatedTick,inputAmount0, outputAmount0) =
BunniSwapMath.computeSwap(input, balance0, balance1);
}
```

```
function _compute_swap(int24 tickSpacing, uint64 balance0, uint64 balance1, int64
amountSpecified, uint160 sqrtPriceLimitX96, int24 tickLower, int24 tickUpper, int24
currentTick) internal returns (BunniSwapMath.BunniComputeSwapInput memory input)
{
    tickSpacing = int24(clampBetween(tickSpacing, MIN_TICK_SPACING,
MAX_TICK_SPACING));
    (int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

    tickLower = roundTickSingle(int24(clampBetween(tickLower, minUsableTick,
maxUsableTick - tickSpacing)), tickSpacing);
    tickUpper = roundTickSingle(int24(clampBetween(tickUpper, tickLower +
tickSpacing, maxUsableTick)), tickSpacing);
    currentTick = roundTickSingle(int24(clampBetween(currentTick, minUsableTick,
maxUsableTick)), tickSpacing);
    console2.log("ticklower", tickLower);
    console2.log("tickUpper", tickUpper);
    console2.log("currentTick", currentTick);
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.STATIC, tickLower,
tickUpper));
   // set up pool key
    PoolKey memory key;
    key.tickSpacing = tickSpacing;

    if (!ldf.isValidParams(key, 0, ldfParams))
        revert();

    // set up BunniComputeSwapInput
    input.key = key;
    input.sqrtPriceX96 = TickMath.getSqrtPriceAtTick(currentTick);
    input.currentTick = currentTick;
    input.liquidityDensityFunction = ldf;
    input.arithmeticMeanTick = int24(0);
    input.ldfParams = ldfParams;
    input.ldfState = LDF_STATE;

    // initialize swap params
    input.swapParams.amountSpecified = clampBetween(amountSpecified,
type(int64).min, type(int64).max);
    input.swapParams.sqrtPriceLimitX96 = uint160(clampBetween(sqrtPriceLimitX96,
MIN_SQRT_PRICE, MAX_SQRT_PRICE));

    // query the LDF to get total liquidity and token densities
    (
        uint256 totalLiquidity,
```

```
        uint256 totalDensity0X96,
        uint256 totalDensity1X96,
        uint256 liquidityDensityOfRoundedTickX96,
        bytes32 newLdfState,
        bool shouldSurge
    ) = queryLDF({
        key: key,
        sqrtPriceX96: input.sqrtPriceX96,
        tick: input.currentTick,
        arithmeticMeanTick: input.arithmeticMeanTick,
        ldf: ldf,
        ldfParams: ldfParams,
        ldfState: LDF_STATE,
        balance0: balance0,
        balance1: balance1
    });

    input.totalLiquidity = totalLiquidity;
    input.totalDensity0X96 = totalDensity0X96;
    input.totalDensity1X96 = totalDensity1X96;
    input.liquidityDensityOfRoundedTickX96 = liquidityDensityOfRoundedTickX96;

    return input;
}
```

Figure D.1: Test harness to reproduce TOB-BUNNI-16

```
// Reproduced from:
medusa/test_results/1729179771923198000-6e49efa8-7631-413f-8f35-8e2b6876bcb5.json
function test_auto_compare_swap_with_reverse_swap_with_zeroForOne_vs_oneForZero_8()
public {

    vm.warp(block.timestamp + 360607);
    vm.roll(block.number + 23050);
    vm.prank(0x0000000000000000000000000000000000030000);

target.compare_swap_with_reverse_swap_with_zeroForOne_vs_oneForZero(int24(452771),
uint64(557917482759850018), uint64(280850432322856), int64(2582874864561197724),
uint160(0), int24(-3536473), int24(-957432), int24(2031986));
}
```

Figure D.2: Unit test reproducer for TOB-BUNNI-16

- **The `computeSwap` function in `BunniSwapMath` should output the same amount of input and output tokens in both the `ExactIn` and `ExactOut` configurations given the same pool state:** Figures D.3 and D.4 show our test configuration and unit test reproducer for this invariant. This resulted in issue TOB-BUNNI-17.

```
function compare_exact_in_swap_with_exact_out_swap(int24 tickSpacing, uint64
balance0, uint64 balance1, int64 amountSpecified, uint160 sqrtPriceLimit, int24
tickLower, int24 tickUpper, int24 currentTick, bool zeroForOne) public {
    // Initialize LDF to Uniform distribution
    ldf = ILiquidityDensityFunction(address(new UniformDistribution()));

    // Initialize parameters before swapinng
    BunniSwapMath.BunniComputeSwapInput memory input1 = _compute_swap(tickSpacing,
balance0, balance1, amountSpecified, sqrtPriceLimit, tickLower, tickUpper,
currentTick);

    // Check for exactIn or exactOut
    // amountSpecifiec negative means exactIn
    input1.swapParams.zeroForOne = zeroForOne;
    try this.swap(input1, balance0, balance1) returns (uint160 updatedSqrtPriceX96,
int24 updatedTick, uint256 inputAmount0, uint256 outputAmount0) {
        require(inputAmount0 != 0 && outputAmount0 != 0);

        if (inputAmount0 != uint64(inputAmount0) || outputAmount0 !=
uint64(outputAmount0))
            return;

        BunniSwapMath.BunniComputeSwapInput memory input2 =
_compute_swap(tickSpacing, balance0, balance1, -amountSpecified, sqrtPriceLimit,
tickLower, tickUpper, currentTick);
        input2.swapParams.amountSpecified = amountSpecified < 0 ?
int256(outputAmount0) : -int256(inputAmount0);
        input2.swapParams.zeroForOne = zeroForOne;

        (,,uint256 inputAmount1, uint256 outputAmount1) =
BunniSwapMath.computeSwap(input2, balance0, balance1);


        if (amountSpecified < 0){
            assertWithMsg(inputAmount0 >= inputAmount1, "Exact In and Exact Out
input amounts does not match");
        }
        else{
            assertWithMsg(outputAmount0 >= outputAmount1, "Free tokens for Exact In
and Exact Out combination");
        }
    }
    catch Panic(uint /*errorCode*/) {
        return;
    }
    catch(bytes memory reason) {
```

```
        emit LogBytes(reason);
        return;
    }
}


function swap(BunniSwapMath.BunniComputeSwapInput calldata input, uint256 balance0,
uint256 balance1) public returns (uint160 updatedSqrtPriceX96, int24 updatedTick,
uint256 inputAmount0, uint256 outputAmount0) {
    require(msg.sender == address(this));
    (updatedSqrtPriceX96, updatedTick,inputAmount0, outputAmount0) =
BunniSwapMath.computeSwap(input, balance0, balance1);
}

function _compute_swap(int24 tickSpacing, uint64 balance0, uint64 balance1, int64
amountSpecified, uint160 sqrtPriceLimitX96, int24 tickLower, int24 tickUpper, int24
currentTick) internal returns (BunniSwapMath.BunniComputeSwapInput memory input)
{
    tickSpacing = int24(clampBetween(tickSpacing, MIN_TICK_SPACING,
MAX_TICK_SPACING));
    (int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

    tickLower = roundTickSingle(int24(clampBetween(tickLower, minUsableTick,
maxUsableTick - tickSpacing)), tickSpacing);
    tickUpper = roundTickSingle(int24(clampBetween(tickUpper, tickLower +
tickSpacing, maxUsableTick)), tickSpacing);
    currentTick = roundTickSingle(int24(clampBetween(currentTick, minUsableTick,
maxUsableTick)), tickSpacing);
    console2.log("ticklower", tickLower);
    console2.log("tickUpper", tickUpper);
    console2.log("currentTick", currentTick);
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.STATIC, tickLower,
tickUpper));
   // set up pool key
    PoolKey memory key;
    key.tickSpacing = tickSpacing;

    if (!ldf.isValidParams(key, 0, ldfParams))
        revert();

    // set up BunniComputeSwapInput
    input.key = key;
    input.sqrtPriceX96 = TickMath.getSqrtPriceAtTick(currentTick);
    input.currentTick = currentTick;
    input.liquidityDensityFunction = ldf;
```

```
    input.arithmeticMeanTick = int24(0);
    input.ldfParams = ldfParams;
    input.ldfState = LDF_STATE;

    // initialize swap params
    input.swapParams.amountSpecified = clampBetween(amountSpecified,
type(int64).min, type(int64).max);
    input.swapParams.sqrtPriceLimitX96 = uint160(clampBetween(sqrtPriceLimitX96,
MIN_SQRT_PRICE, MAX_SQRT_PRICE));

    // query the LDF to get total liquidity and token densities
    (
        uint256 totalLiquidity,
        uint256 totalDensity0X96,
        uint256 totalDensity1X96,
        uint256 liquidityDensityOfRoundedTickX96,
        bytes32 newLdfState,
        bool shouldSurge
    ) = queryLDF({
        key: key,
        sqrtPriceX96: input.sqrtPriceX96,
        tick: input.currentTick,
        arithmeticMeanTick: input.arithmeticMeanTick,
        ldf: ldf,
        ldfParams: ldfParams,
        ldfState: LDF_STATE,
        balance0: balance0,
        balance1: balance1
    });

    input.totalLiquidity = totalLiquidity;
    input.totalDensity0X96 = totalDensity0X96;
    input.totalDensity1X96 = totalDensity1X96;
    input.liquidityDensityOfRoundedTickX96 = liquidityDensityOfRoundedTickX96;

    return input;
}
```

Figure D.3: Test configuration to reproduce TOB-BUNNI-17

```
// Reproduced from:
medusa/test_results/1729179771923832000-8f79f3b5-8695-4e99-a008-76ee8eb851f6.json
function test_auto_compare_exact_in_swap_with_exact_out_swap_14() public {

    vm.warp(block.timestamp + 147361);
    vm.roll(block.number + 7424);
    vm.prank(0x0000000000000000000000000000000000010000);
    target.compare_exact_in_swap_with_exact_out_swap(int24(-119478),
uint64(8097015911325976438), uint64(2021425479330258804), int64(35848633053),
uint160(0), int24(0), int24(1683232), int24(1935518), true);
}
```

Figure D.4: Unit test reproducer for TOB-BUNNI-17

- **Users should not be able to get free output tokens for zero input tokens when `amountSpecified` is nonzero for a given valid pool state:** Figures D.5 and D.6 show our test configuration and unit test reproducer for this invariant. This resulted in issue TOB-BUNNI-15.

```
function test_free_or_loss_of_tokens_during_swap(int24 tickSpacing, uint64 balance0,
uint64 balance1, int64 amountSpecified, uint160 sqrtPriceLimit, int24 tickLower,
int24 tickUpper, int24 currentTick, bool zeroForOne) public{
    // Initialize LDF to Uniform distribution
    ldf = ILiquidityDensityFunction(address(new UniformDistribution()));

    // Initialize parameters before swapinng
    BunniSwapMath.BunniComputeSwapInput memory input1 = _compute_swap(tickSpacing,
balance0, balance1, amountSpecified, sqrtPriceLimit, tickLower, tickUpper,
currentTick);

    // Check for exactIn or exactOut
    // amountSpecifiec negative means exactIn, so zeroForOne needs to be true
    input1.swapParams.zeroForOne = zeroForOne;
    try this.swap(input1, balance0, balance1) returns (uint160 updatedSqrtPriceX96,
int24 updatedTick, uint256 inputAmount0, uint256 outputAmount0) {
        if (outputAmount0 > 0 && inputAmount0 == 0){
            assertWithMsg(false, "Free tokens");
            return;
        }
    }

    catch Panic(uint /*errorCode*/) {
        // This is executed in case of a panic,
        // i.e. a serious error like division by zero
        // or overflow. The error code can be used
        // to determine the kind of error.
```

```
        return;
    }
    catch(bytes memory reason) {
       emit LogBytes(reason);
       return;
    }
}


function swap(BunniSwapMath.BunniComputeSwapInput calldata input, uint256 balance0,
uint256 balance1) public returns (uint160 updatedSqrtPriceX96, int24 updatedTick,
uint256 inputAmount0, uint256 outputAmount0) {
    require(msg.sender == address(this));
    (updatedSqrtPriceX96, updatedTick,inputAmount0, outputAmount0) =
BunniSwapMath.computeSwap(input, balance0, balance1);
}

function _compute_swap(int24 tickSpacing, uint64 balance0, uint64 balance1, int64
amountSpecified, uint160 sqrtPriceLimitX96, int24 tickLower, int24 tickUpper, int24
currentTick) internal returns (BunniSwapMath.BunniComputeSwapInput memory input)
{
    tickSpacing = int24(clampBetween(tickSpacing, MIN_TICK_SPACING,
MAX_TICK_SPACING));
    (int24 minUsableTick, int24 maxUsableTick) =
        (TickMath.minUsableTick(tickSpacing), TickMath.maxUsableTick(tickSpacing));

    tickLower = roundTickSingle(int24(clampBetween(tickLower, minUsableTick,
maxUsableTick - tickSpacing)), tickSpacing);
    tickUpper = roundTickSingle(int24(clampBetween(tickUpper, tickLower +
tickSpacing, maxUsableTick)), tickSpacing);
    currentTick = roundTickSingle(int24(clampBetween(currentTick, minUsableTick,
maxUsableTick)), tickSpacing);
    console2.log("ticklower", tickLower);
    console2.log("tickUpper", tickUpper);
    console2.log("currentTick", currentTick);
    bytes32 ldfParams = bytes32(abi.encodePacked(ShiftMode.STATIC, tickLower,
tickUpper));
   // set up pool key
    PoolKey memory key;
    key.tickSpacing = tickSpacing;

    if (!ldf.isValidParams(key, 0, ldfParams))
        revert();

    // set up BunniComputeSwapInput
    input.key = key;
```

```
    input.sqrtPriceX96 = TickMath.getSqrtPriceAtTick(currentTick);
    input.currentTick = currentTick;
    input.liquidityDensityFunction = ldf;
    input.arithmeticMeanTick = int24(0);
    input.ldfParams = ldfParams;
    input.ldfState = LDF_STATE;

    // initialize swap params
    input.swapParams.amountSpecified = clampBetween(amountSpecified,
type(int64).min, type(int64).max);
    input.swapParams.sqrtPriceLimitX96 = uint160(clampBetween(sqrtPriceLimitX96,
MIN_SQRT_PRICE, MAX_SQRT_PRICE));

    // query the LDF to get total liquidity and token densities
    (
        uint256 totalLiquidity,
        uint256 totalDensity0X96,
        uint256 totalDensity1X96,
        uint256 liquidityDensityOfRoundedTickX96,
        bytes32 newLdfState,
        bool shouldSurge
    ) = queryLDF({
        key: key,
        sqrtPriceX96: input.sqrtPriceX96,
        tick: input.currentTick,
        arithmeticMeanTick: input.arithmeticMeanTick,
        ldf: ldf,
        ldfParams: ldfParams,
        ldfState: LDF_STATE,
        balance0: balance0,
        balance1: balance1
    });

    input.totalLiquidity = totalLiquidity;
    input.totalDensity0X96 = totalDensity0X96;
    input.totalDensity1X96 = totalDensity1X96;
    input.liquidityDensityOfRoundedTickX96 = liquidityDensityOfRoundedTickX96;

    return input;
}
```

Figure D.5: Test configuration to reproduce TOB-BUNNI-15

```
// Reproduced from:
medusa/test_results/1729179375940890000-0ecd55b5-1e4c-423e-9a1c-43a0034ed02d.json
function test_auto_test_free_or_loss_of_tokens_during_swap_16() public {
    vm.warp(block.timestamp + 559527);
    vm.roll(block.number + 50486);
    vm.prank(0x0000000000000000000000000000000000010000);
    target.test_free_or_loss_of_tokens_during_swap(int24(-4956763),
uint64(3220491473978168974), uint64(857672123281148825), int64(3695396752905551871),
uint160(532839684007055479803847087098878 52), int24(393243), int24(-260336),
int24(5029467), true);
}
```

Figure D.6: Unit test reproducer for TOB-BUNNI-15

E. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

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

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

F. Fix Review Results

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

From January 2 to January 6, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Bacon Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

We marked the fix status of TOB-BUNNI-4 as resolved since it sufficiently mitigates the risks described in this finding. However, the fix potentially introduces a denial-of-service attack vector, which is described in the detailed fix review result for this finding. The findings TOB-BUNNI-9, TOB-BUNNI-18, and TOB-BUNNI-19 were considered partially resolved and would benefit from further investigation. During the fix review, we were unable to determine the fix status of the three issues marked as undetermined due to the scope and complexity of the changes.

We recommend that the Bacon Labs team conduct a follow-up security review of the system with a focus on the system arithmetic, and continue improving the unit and fuzz testing suite to verify the system behaves as expected. This follow-up review should address the concerns raised in the detailed fix review results for TOB-BUNNI-4 and the partially resolved and undetermined issues.

In summary, of the 19 issues described in this report, Bacon Labs has resolved 10 issues, has partially resolved four issues, and has not resolved two issues. We could not determine the resolution status of the remaining three issues during this fix review. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	BunniToken permit cannot be revoked	Informational	Resolved
2	Token approvals to ERC-4626 vaults are never revoked	Informational	Resolved
3	Overly strict bid withdrawal validation reduces am-AMM efficiency by enabling griefing	Low	Unresolved
4	Users can bid arbitrarily low rent during the bidding process	Undetermined	Resolved

5	Dirty bits of narrow types are not cleaned	Informational	Resolved
6	Rebalance mechanism access control can be bypassed	High	Resolved
7	Pools can be drained via the rebalance mechanism by selectively executing the rebalanceOrderPreHook and the rebalanceOrderPostHook	High	Resolved
8	Missing maximum bounds for rebalance parameters	Informational	Resolved
9	Excess liquidity can be inflated to create arbitrarily large rebalance orders	High	Partially Resolved
10	Insufficient event generation	Informational	Partially Resolved
11	AmAmm manager can manipulate TWAP prices without risk	Medium	Resolved
12	Lack of zero-value checks	Informational	Resolved
13	Lack of systematic approach to rounding and arithmetic errors	Undetermined	Undetermined
14	Native assets deposited to pools with no native currencies are lost	Informational	Resolved
15	Users can gain free tokens through the BunniSwap swap functionality	High	Undetermined
16	Users can gain tokens during roundtrip swaps	High	Undetermined
17	Different amount of input/output tokens can be returned in ExactIn and ExactOut configurations during the swap	Low	Unresolved
18	BunniSwap swap functionality can cause panics during the swap	Undetermined	Partially Resolved

19	cumulativeAmount0 can be greater than the cumulative amount computed through inverse functionality for certain LDFs	Undetermined	Partially Resolved

Detailed Fix Review Results

TOB-BUNNI-1: BunniToken permit cannot be revoked
Resolved in PR #67. An `incrementNonce` function was added to the `BunniToken` contract that allows users to invalidate their previously created permit signatures by increasing the nonce by one.

TOB-BUNNI-2: Token approvals to ERC-4626 vaults are never revoked
Resolved in PR #68. A check was added to the `BunniHubLogic` and `BunniHub` contracts that resets the approvals to the ERC-4626 vault after a call to the `deposit` function, if the allowance to the vault is not equal to zero.

TOB-BUNNI-3: Overly strict bid withdrawal validation reduces am-AMM efficiency by enabling griefing
Unresolved. The client provided the following context for this finding's fix status:

> *Acknowledged, won't fix. The bid withdrawal validation rule as specified by the am-AMM paper D_top / R_top + D_next / R_next >= K would allow the current manager to prevent anyone from replacing him by making a higher bid and immediately withdrawing its deposit, which is a worse outcome than the griefing attack described by the issue. This griefing risk will be noted in our documentation.*

TOB-BUNNI-4: Users can bid arbitrarily low rent during the bidding process
Resolved in PR #66, PR #3, PR #75, and commit `fbbe374`. A `MIN_RENT` function was added to the `AmAmm` and `BunniHook` contracts that dynamically defines the minimum rent that a user can bid as a percentage of the total supply of Bunni tokens. While this sufficiently resolves the issue described in this finding, there is a chance this could be used to perform a denial-of-service attack on other user bids by front-running their bid with a deposit in order to increase the total supply of Bunni tokens, in turn making the affected user's rent too low. Additionally, since the `minRentMultiplier` parameter is immutable, the system should be tested with various values of the parameter to determine its impact on the economic and incentive structure of the system.

TOB-BUNNI-5: Dirty bits of narrow types are not cleaned
Resolved in PR #69. The `SwapMath` and `SqrtPriceMath` library contracts have been updated to use a more recent version of these libraries, taken from Uniswap's v4-core repository at commit `c817314`. The higher order bits are correctly cleaned before use.

TOB-BUNNI-6: Rebalance mechanism access control can be bypassed

Resolved in PR #70. The Bacon Labs team uses a modified version of the `FloodPlain` contract that disallows direct calls to the `permit2` contract in the pre- and post-hooks, as well as appends the generated order hash to the end of the calldata for verification. When a pre- or post-hook is executed, the order hash is checked against the one saved in state for that ID. This prevents access control bypass and selective execution of pre- and post-hooks. We did not review the bytecode used for deployment of `FloodPlain` in the `FloodDeployer` contract.

TOB-BUNNI-7: Pools can be drained via the rebalance mechanism by selectively executing the rebalanceOrderPreHook and the rebalanceOrderPostHook
Resolved in PR #70. The changes described in the detailed fix review results of TOB-BUNNI-6 prevent selective execution of the `rebalanceOrderPreHook` and `rebalanceOrderPostHook`.

TOB-BUNNI-8: Missing maximum bounds for rebalance parameters
Resolved in PR #71. Additional constants were added that act as upper bounds for the `rebalanceMaxSlippage`, `rebalanceTwapSecondsAgo`, `rebalanceOrderTTL`, `surgeFeeHalfLife`, and `surgeFeeAutostartTreshold` parameters. Additional checks were added to the `isValidParams` function of the `BunniHook` contract to enforce these upper bounds.

TOB-BUNNI-9: Excess liquidity can be inflated to create arbitrarily large rebalance orders
Partially resolved in PR #74. The PR adds a `canWithdraw` function that is called when a user attempts to withdraw liquidity through the `BunniHub` contract. The `canWithdraw` function checks if a rebalance order is currently active, and if it is, it prevents withdrawals. While this does mitigate the exploit scenario described in the finding, it does not prevent the inflation of the rebalance order amounts. A malicious user could still deposit liquidity prior to the creation of the rebalance order to inflate the amounts; however, they would need to execute the rebalance order prior to being able to withdraw, which may not be profitable. Additionally, it might be possible to intentionally perform a denial-of-service attack on user withdrawals by disbalancing the pool, but we were unable to determine if this is possible during the fix review.

This issue should be investigated further by listing out all of the possible scenarios where the rebalance order amounts could be inflated and abused, and by creating unit and fuzz tests to determine the profitability and likelihood of such scenarios taking place.

TOB-BUNNI-10: Insufficient event generation
Partially resolved in PR #83 and PR #79. Additional event emission was added to the `_claimFees` and `claimReferralRewards` functions. Other events mentioned in the finding were deemed unnecessary by the Bacon Labs team and omitted to reduce transaction gas costs.

TOB-BUNNI-11: AmAmm manager can manipulate TWAP prices without risk
Resolved in PR #66. An additional parameter was added to the `DecodedHookParams` struct that defines the maximum `AmAmm` swap fee. A constant was added to limit the maximum swap fee to 10%. This reduces the risk of an `AmAmm` manager manipulating the TWAP price, since they cannot prevent other users from using the pool via the swap fee parameter. However, if the pool-defined maximum swap fee is less than the maximum `AmAmm` swap fee, an `AmAmm` manager can set the swap fee to be larger than the pool-defined maximum due to a code change in the `_payloadIsValid` function. The Bacon Labs team stated that this is an intentional design choice.

TOB-BUNNI-12: Lack of zero-value checks
Resolved in PR #72. Zero-value checks were added to the `BunniHook` and `BunniHub` contract constructors, validating the input parameters. A check was added to the `beforeSwap` function of the `BunniHookLogic` library and the `quoteSwap` function of the `BunniQuoter` contract that ensures the `inputAmount` cannot be zero if the swap is `ExactOut`.

TOB-BUNNI-13: Lack of systematic approach to rounding and arithmetic errors
Undetermined in PR #80. This PR adds explicit rounding directions in the LDF and `BunniSwapMath` contract arithmetic and updates the behavior of the LDF library contracts. While the explicit rounding directions are a good step toward implementing a systematic approach to rounding and arithmetic errors, we were unable to determine if the rounding directions are correct and if arithmetic errors are reduced.

We recommend that the Bacon Labs team continue to develop the fuzzing harness and system- and function-level invariants to verify that the system behaves as expected. Additionally, the `lnQ96RoundingUp` function added to the `ExpMath` library contract should be fuzzed against a verifiably correct reference implementation.

TOB-BUNNI-14: Native assets deposited to pools with no native currencies are lost
Resolved in PR #73. The `deposit` function of the `BunniHubLogic` library will revert if the `msg.value` is larger than zero, but neither of the tokens in the pool is native. This prevents the value from becoming stuck in the contract.

TOB-BUNNI-15: Users can gain free tokens through the BunniSwap swap functionality
Undetermined in PR #81, PR #76, and PR #82. The PRs address the incorrect `updatedTick` assignment, add new internal accounting for the unused/idle balance of pools, add cumulative amount limit checks to the LDFs, and partially modify the way rebalance order state is reset. While some minor code issues are present in the PRs (e.g., the `shouldSurge` variable uses a comparison operator instead of assignment), they are addressed in subsequent commits. The idle balance of the pool is deducted from the total liquidity of the pool in order to prevent attacks in which the pool's inactive balance becomes instantly active, enabling malicious users to profit from more favorable pool conditions.

While the idle balance seems to be correctly accounted for and deducted from the pool liquidity, due to the scope and complexity of the changes and their side effects, we were unable to determine the fix status of this issue.

TOB-BUNNI-16: Users can gain tokens during round-trip swaps
Undetermined in PR #81, PR #76, and PR #82. We were unable to determine if the PRs fully address this issue during the fix review duration. For details, see the above detailed fix review result for finding TOB-BUNNI-15.

TOB-BUNNI-17: Different amount of input/output tokens can be returned in ExactIn and ExactOut configurations during the swap
Unresolved. The client provided the following context for this finding's fix status:

> *Acknowledged, won't fix. The issue seems to stem from Uniswap v4's SwapMath library which has slightly different rates for exact input and exact output swaps, so we decided to not fix it since it's not an issue in our own swap logic.*

TOB-BUNNI-18: BunniSwap swap functionality can cause panics during the swap
Partially resolved in PR #81. The `computeSwap` function of the `BunniSwapMath` library contract was updated so that the arithmetic does not revert on an underflow, but rather returns zero. While this does partially address this issue, we did not manage to determine if this has any important side effects on other parts of the system, or if it can lead to incorrect calculation during certain edge cases. We recommend thoroughly testing this change using unit tests and fuzz tests to determine the exact conditions when this case might trigger and to verify that no new issues were introduced as a result.

TOB-BUNNI-19: cumulativeAmount0 can be greater than the cumulative amount computed through inverse functionality for certain LDFs
Partially resolved in PR #81. The cumulative amounts calculated in the `inverseCumulativeAmount0` and `inverseCumulativeAmount1` functions of the `LibUniformDistribution` library contract are checked against a maximum value for the respective cumulative amount. If the computed value is greater than the maximum value, the function will return zero and `false`, indicating a failure in the operation. While this does prevent incorrect calculations from being considered correct in the system, the solution appears to be a workaround rather than a systematic fix of the underlying issue. We were unable to determine the side effects of this change during the duration of the fix review.

G. Fix Review Status Categories

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

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

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.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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