BAIL security

Hypertrade
V3 Core

FINAL REPORT

December ‘2025

Disclaimer:

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.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Hypertrade – V3 Core - Audit Report
Website	ht.xyz
Language	Solidity
Methods	Manual Analysis
Github repository	https://gitlab.com/qwerty_labs/ht.xyz/uniswap-v2-fork/-/tree/617e5fabd4bdc131f1a2ff8c5adadf76a08abc3e https://gitlab.com/qwerty_labs/ht.xyz/uniswap-v3-fork/-/tree/ce737af231d3368398d457e456dba25c3b6ec584
Resolution 1	https://gitlab.com/qwerty_labs/ht.xyz/uniswap-v2-fork/-/commit/9d531c9a3cd12bcaeeeb0051660c972c339d019b

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High	4	2		2		
Medium	8	3		5		
Low	4			4		
Informational	10			10		
Governance	4			4		
Total	30	5		25		

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Uniswap V2 Fork

ACL

The ACL contract is a simple registry-like contract which allows the contract owner to define multiple configurations which are consulted by the UniswapV2Pool contract. These are the following:

- **General Whitelist Restriction**: This feature allows the contract owner to fully pause swaps for non-whitelisted addresses

- **Individual Whitelist**: In the scenario of the general whitelist, only whitelisted individuals can execute swaps

- **Block Delay**: This value exposes a swap delay which is bound per msg.sender

- **Block Delay Disabled**: This feature allows to globally disable swap delays

Privileged Functions

- setWhitelisted
- setWhitelistEnabled
- setBlockDelay
- setBlockDelayDisabled

Issue_01	Governance Privilege: Blocking of swaps
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, governance can simply set a large blockDelay or enable a global whitelist without having any individual whitelisted.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

UniswapV2Pair

The UniswapV2Pair is a standard UniswapV2 pool implementation which implements a whitelist and blockDelay mechanism that is dependent on the ACL implementation.

All changes can be found in the following link:

https://www.diffchecker.com/HtlluKqZ/

Appendix: Throttle Mechanism

The throttle mechanism is applied during the execution of the swap function to enforce a minimum block delay between consecutive swap calls originating from the same caller address. Its purpose is to rate-limit interactions at the pair level and constrain high-frequency execution patterns.

During a swap call, after basic output and liquidity validations but before any token transfers or external callbacks occur, the pair queries the factory-linked ACL contract to determine whether throttling is active. If whitelisting is enabled, the caller must first satisfy the whitelist requirement. Subsequently, the throttle logic is evaluated.

The throttle derives a global blockDelay value from the ACL. If the delay is zero, or if the caller address is explicitly exempted via blockDelayDisabled, the throttle is bypassed entirely. Otherwise, the contract loads the caller's last recorded execution block from lastCallBlock and enforces that the current block number strictly exceeds the previous block plus the configured delay.

Upon successful validation, the caller's lastCallBlock entry is updated to the current block number, and swap execution continues normally. If the block delay condition is not satisfied, the transaction reverts.

This mechanism operates strictly on a per-address basis and only affects the swap control-flow. Liquidity provision, liquidity removal, reserve synchronization, and balance skimming remain unthrottled. As a result, the throttle selectively constrains swap frequency without impacting other lifecycle operations of the pair.

The strictness of this mechanism is entirely governed by the external ACL configuration, making swap execution behavior sensitive to governance-controlled parameters defined outside of the pair contract itself.

Privileged Functions

- none

Core Invariants:

INV 1: If the global whitelisting is enabled, the swapper must be whitelisted

INV 2: If blockDelay > 0 and caller is not exempt from the delay, a swap is only allowed after the blockDelay has passed, based on the last swap execution

Issue_02	Throttling is applied per msg.sender
Severity	High
Description	The UniswapV2Pair contract integrates an access-control and rate-limiting layer into swap via an external ACL obtained from IUniswapV2Factory(factory).acl(). If ACL.whitelistedEnabled() is active, swap requires ACL.isWhitelisted(msg.sender). Independently, the pair applies _throttle(msg.sender, _acl) which enforces a minimum block delay between consecutive calls for the same caller address by checking block.number > lastCallBlock[msg.sender] + blockDelay and then updating lastCallBlock[msg.sender] = block.number. This throttle executes before any token transfers, callbacks, or reserve updates, and reverts the entire swap if the delay condition is not satisfied. The throttle accounting is keyed on msg.sender only. In typical usage patterns, swap is not called by the end user directly, but by a shared routing contract such as a canonical router or aggregator. If the system's whitelisting policy forces swaps through such a shared router address, then all user swaps share a single throttle bucket because msg.sender is always the router. This creates a shared-rate-limit surface where any single swap routed through the shared caller updates lastCallBlock[router] and blocks subsequent swaps through that same caller for the configured blockDelay. The mechanism therefore does not throttle per end user in the common routed execution path and instead throttles an entire class of users collectively. Under configurations where a shared router is the primary or only whitelisted swap caller, an attacker can grief by repeatedly triggering swaps via that shared router to continuously refresh

	lastCallBlock[router], causing unrelated users' swaps through the same router to revert with "BD" for as long as the attacker sustains activity at the required cadence. The impact is denial of service and disruption of trading availability through the canonical path, with secondary effects including degraded execution reliability for integrators, increased failure rates for multi-call routes that depend on completing the swap step, and selective suppression of market activity through the most common entrypoint.
Recommendations	Consider implementing a rate limitation on the router level. It has to be noted that this will still remain an issue for aggregators and integrators that also have a shared entrypoint.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: "The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security."

Issue_03	Sybil-attack allows for bypassing msg.sender limitation in case of disabled whitelist.
Severity	**Medium**
Description	When global whitelisting is disabled, any address can call swap, and the throttle becomes an address scoped limiter keyed purely by msg.sender. There is no binding to an end user identity, transaction origin, or any shared actor concept. As a result, a single actor can bypass the throttle by distributing calls across multiple caller addresses. Each new EOA begins with an unset lastCallBlock entry and therefore passes the throttle condition irrespective of prior throttled calls performed by the same actor via other addresses.
Recommendations	As with most sybil attacks, there is no solution to prevent these.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

Issue_04	Off-by-one issue during blockDelay enforcement
Severity	**Informational**
Description	The throttle check uses a strict inequality: require(block.number > last + delay, "BD"). This enforces that the caller can only execute again in a block strictly greater than last + delay. If blockDelay is expected to represent the number of blocks that must elapse before a new call is permitted, then the strict comparison enforces an additional block of delay beyond the configured value. For example, if a caller executes at block 100 with blockDelay = 1, a natural reading of "1 block delay" permits the next call at block 101 and disallows swaps at block 100, but the current logic only permits it starting at block 102. The behavior is therefore off by one relative to the common interpretation of block delay parameters.
Recommendations	Consider switching to >=.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: "The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security."

UniswapV2Factory

The UniswapV2Factory is almost a 1:1 fork from Uniswap's trusted implementation with the only change that the ACL contract is stored and settable by the feeToSetter address.

All changes can be found in the following link:

https://www.diffchecker.com/Fb7yjguf/

Privileged Functions

- setFeeTo
- setFeeToSetter
- setAcl

Issue_05	Governance Privilege: Invalid setting of ACL
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, it is allowed for governance to set an incompatible ACL address which then can revert on each swap.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: "The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security."

UniswapV2Staker

The UniswapV2Staker contract is a highly customized staking contract which derives its root mechanics from the Synthetix Staking Rewards implementation:

github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol

It allows for the permissionless creation of incentives which distributes a specified amount of reward tokens over a fixed duration. Users can stake LP tokens to any created incentive and will then receive a share of the reward distribution proportionally to their provided stake and the overall staked amount.

The contract furthermore implements a novel refund mechanism which allows the incentive creator to retrieve any unconsumed rewards.

Appendix: Refund Mechanism

The refund mechanism is driven by the unusedRewards accounting, which tracks rewards that were scheduled to accrue over time but could not be distributed because the incentive had zero total stake during the relevant time interval. This tracking is updated opportunistically during reward accounting updates and again during incentive finalization, ensuring that reward tokens that were never attributable to stakers are recoverable by the incentive refundee.

Unused rewards are primarily accumulated inside the updateReward modifier, which executes before stake balance changes in stake, unstake, and claimReward. Each time updateReward runs, the incentive first updates rewardPerTokenStored to the current value returned by rewardPerToken, then computes lastTimeApplicable = lastTimeRewardApplicable(incentiveId) and compares it against lastUpdateTime. If totalStaked is zero and time has advanced, the contract computes timeDelta = lastTimeApplicable - lastUpdateTime and increments unusedRewards by timeDelta * rewardRate. The incentive's lastUpdateTime is then advanced to lastTimeApplicable, meaning subsequent calls do not double count the same interval. In the zero stake regime, rewardPerToken remains constant and the undistributable portion is redirected into unusedRewards rather than being silently lost.

Because updateReward is only executed when users interact with the incentive, the contract also performs a similar unused rewards synchronization step in endIncentive to capture any remaining undistributed time window that occurred after the last user interaction. During endIncentive, the contract computes lastTimeApplicable and, if totalStaked is still zero and

lastTimeApplicable > lastUpdateTime, it adds the missing interval to unusedRewards and updates lastUpdateTime accordingly. This ensures that the unused rewards accounting is fully synchronized up to the end of the incentive even if no one interacted during the final period.

After unused rewards have been synchronized, endIncentive enforces that the incentive has ended, has not been ended before, and that the caller is the recorded refundee. The incentive is then marked as ended. The function computes the rewards that were actually promised by the incentive as duration * rewardRate, where duration is endTime - startTime. Since rewardRate is computed as integer division at creation time, this promised amount may be strictly less than totalReward, leaving a residual dust amount equal to totalReward - promisedRewards.

The total refund amount is computed as unusedRewards + dust, which represents the complete set of reward tokens that are not attributable to stakers. These remaining funds are then transferred back to the original incentive creator.

Privileged Functions

- none

Core Invariants:

INV 1: Cannot create an incentive which is too far in the future

INV 2: Incentive duration cannot exceed the maxIncentiveDuration

INV 3: Cannot create the same incentiveID twice

INV 4: startTime cannot be in past

INV 5: Stake/unstake/claimRewards can only be called for valid incentiveId

INV 6: Stake can only be called after incentive has started and before it has ended

INV 7: lastTimeRewardApplicable must never become larger than incentive.endTime

INV 8: endIncentive can only be called once for an incentiveId

INV 9: endIncentive can only be called when the endTime is reached

INV 10: Incentives can only be distributed for valid pairs

INV 11: endIncentive can only be called by the initial incentive creator

INV 12: An incentive can only be called for a valid LP token

Issue_06	Unclaimed but allocated rewards are zero-d out during emergencyWithdraw
Severity	**High**
Description	The emergencyWithdraw function is a safety feature in the scenario where there are accounting bugs in the reward update mechanism which would result in a revert for normal unstake functions. Therefore, it is generally expected that rewards are not updated and users forfeit any rewards between [user.lastUpdateTime; block.timestamp]. A problem occurs because not only these rewards are forfeited but also any unclaimed rewards which have already been accounted to a user, are forfeited: *deposit.amount = 0;* ***deposit.rewards = 0;*** *deposit.userRewardPerTokenPaid = 0;* Aligning with the logic of emergency withdrawals, there is no such reason to forfeit rewards which are already allocated to a user. These will remain permanently locked in the contract.
Recommendations	Consider removing the highlighted line above, as this will allow users to still claim their already allocated rewards, even if they called emergencyWithdraw.
Comments / Resolution	Resolved, unclaimed rewards are not zero'd out anymore and the caller can still claim these.

Issue_07	Forfeited rewards between [user.lastUpdateTime; block.timestamp] remain permanently locked after emergencyWithdraw
Severity	**Medium**
Description	The emergencyWithdraw function is a safety feature in the scenario where there are accounting bugs in the reward update mechanism which would result in a revert for normal unstake functions. Therefore, it is generally expected that rewards are not updated and users forfeit any rewards between [user.lastUpdateTime; block.timestamp]. A problem occurs because these remain now permanently locked within the contract, instead of allocated to the refundee or to other stakers. The original MasterChef implementation simply avoids the update of rewardPerTokenStored and lastUpdateTime and simply decreases the total stake. This will have the effect that rewards which have been allocated to the user for [user.lastUpdateTime; block.timestamp] will be re-routed to existing stakers because the future increase of rewardPerTokenStored will experience a division by the reduced totalStakedAmount.
Recommendations	Consider whether it makes sense and aligns with the design to re-route any forfeited rewards to existing stakers.
Comments / Resolution	Resolved, the global reward update is now bypassed on emergencyWithdraw which means the overall stake is reduced before the next global reward update and the new decreased overall stake amount is used as a divisor for the rewardPerToken update which results in reward distribution among active stakers. **It is important to note the following remaining side effect:** If rewardPerToken is advanced via a global update by other interactions than emergencyWithdraw, it will be done via the totalStaked amount, which still includes the “to be emergency-withdrawn” amount. If a user now calls emergencyWithdraw, it will

	not update a user's deposit.rewards. This has the effect that in this specific scenario, still a part of the rewards are being stuck. Our recommendation is that users should always call claimReward before emergencyWithdraw is ever triggered. We do consider this as acceptable because a fix of that specific side-effect would be intrusive as it would result in additional arithmetic operations within the emergencyWithdraw function and this function was specifically implemented to avoid arithmetic operations and guarantee a safe-exit.

Issue_08	Lack of support for transfer-tax tokens will create immediate insolvency state
Severity	**Medium**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting. For instance, incentive creation does not apply the before/after pattern (while the stake function does). This means the contract will receive less tokens than it will account for, creating an immediate insolvency scenario.
Recommendations	Consider applying the before/after pattern within createIncentive. IMPORTANT: The createIncentive function must expose the reentrancy guard if such a pattern is implemented
Comments / Resolution	Acknowledged.

Issue_09	PRECISION of 1e18 is insufficient for low decimal reward tokens
Severity	Medium
Description	The UniswapV2Staker implements reward distribution using a cumulative “reward per staked LP” accounting model. For each incentive, rewardPerToken computes newly accrued rewards since lastUpdateTime as timeDelta * rewardRate and converts that to a per-LP increment using PRECISION = 1e18 divided by totalStaked. The updateReward modifier is executed before stake balance changes in stake, unstake, and claimReward. It updates rewardPerTokenStored by calling rewardPerToken, then advances lastUpdateTime to lastTimeRewardApplicable. Separately, the contract tracks unusedRewards only when totalStaked == 0 and time advances, to ensure undistributable rewards can later be refunded. The per-LP increment inside rewardPerToken is subject to integer truncation. When the reward token has low decimals or when rewards are small relative to totalStaked, the computed increment rewardAccrued * PRECISION / totalStaked can floor to zero for a given update interval. Despite the increment being zero, updateReward still advances lastUpdateTime to the current applicable time. This consumes the elapsed interval without increasing rewardPerTokenStored and without crediting unusedRewards (since totalStaked is nonzero). As a result, rewards accrued during that interval are not attributable to any user and are not recoverable through the refund path, creating a deterministic loss of accounting precision. This manifests concretely in scenarios such as low-decimal reward tokens, large LP totals, and frequent updates, where sub-unit per-LP increments are repeatedly rounded away. Rewards can become permanently unclaimable by both stakers and the incentive refundee. Users receive less than the intended

	proportional distribution, and the contract can retain reward token balances that are not accounted in either deposit.rewards or unusedRewards. The effect is amplified when updateReward is triggered frequently, because frequent updates advances lastUpdateTime while keeping accruedRewards minimal and can repeatedly truncate away fractional entitlements. This creates a griefing-style surface where frequent incentive interactions can increase the cumulative rounding loss, reducing effective reward delivery without violating any explicit checks. Simple numeric example: floor(0.9e6*1e18/1000000e18) = 0
Recommendations	Consider increasing the precision to 1e24.
Comments / Resolution	Resolved, precision has been increased. Note: In case of any upgrades, this contract should be newly

Issue_10	Lack of safeTransfer usage
Severity	**Medium**
Description	The contract uses the standard ERC20 pattern for transfers/approvals. This will malfunction for tokens that do not follow the IERC20 interface, for example those that return false or do not return a boolean on the transfer/approve call.
Recommendations	Consider implementing the safeERC20 library
Comments / Resolution	Resolved, safeERC20 is now used for most transfer operations (not for LP token transfers - this is fine as they are standard ERC20).

Issue_11	Lack of minimum duration for new incentive
Severity	**Low**
Description	The createIncentive function enforces various different validations but lacks a minimum incentive enforcement which allows to create an incentive for just 1 block. This could be a precondition for an exploit in the scenario where a bug remains existing. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider implementing a minimum duration (e.g. 1 day) for an incentive to increase robustness of the contract.
Comments / Resolution	Acknowledged.

Issue_12	Incentive can never be ended if refundee is blacklisted for token
Severity	**Informational**
Description	The endIncentive function attempts to transfer dust/unused funds to the refundee. This will revert if the refundee is blacklisted and will prevent the incentive ending.
Recommendations	Consider implementing a recipient address which will be the transfer beneficiary.
Comments / Resolution	Acknowledged.

Issue_13	Lack of reentrancy protection during createIncentive opens theoretical loss of funds risk in case of misconfigured factory
Severity	**Informational**
Description	The UniswapV2Staker contract allows the creation of new staking incentives through createIncentive, which initializes incentive state and then transfers the reward tokens from the incentive creator into the contract. The function is not protected by a reentrancy guard, while other state-mutating user-facing functions such as stake, unstake, and claimReward are protected via nonReentrant. The stake function relies on a balance-delta pattern when transferring LP tokens into the contract, measuring the actual staked amount by comparing the contract's LP balance before and after transferFrom. Incentive creation and staking are therefore part of overlapping token and balance-sensitive control-flows within the same contract. Under the hypothetical assumption that hook-enabled tokens (for example ERC777-style tokens with transfer callbacks) were permitted as staking assets, the lack of reentrancy protection on createIncentive introduces an inconsistent reentrancy surface. While createIncentive itself follows a checks-effects-interactions structure, it performs an external token transfer at the end of execution without reentrancy protection. In a sophisticated execution path, reentrancy from a hook token could trigger a nested createIncentive call during a stake operation, allowing additional token transfers into the contract to occur between the balance snapshot and balance reconciliation used in the staking logic. This would cause the balance-delta calculation in stake to observe more tokens than were actually transferred for staking, inflating the recorded stake amount. The root cause is the combination of an unguarded external call in createIncentive and reliance on balance-delta accounting in stake, rather than an immediate violation of CEI within a single function.

	An attacker can therefore stake while creating a short term incentive (1 block) at the end of a block, with the specified hook token re-gaining: - stake + rewards - rewards The issue does not materialize in the current implementation because pair validation prevents arbitrary staking tokens and standard Uniswap V2 LP tokens do not implement transfer hooks. As such, this remains a theoretical edge-case highlighting an architectural inconsistency in reentrancy assumptions rather than an immediately exploitable vulnerability. However, if the factory is set to a different implementation, this would become security relevant and could invalidate the correctness assumptions of the staking balance accounting and result in a full loss of funds for such a specified reward token.
Recommendations	Consider implementing a reentrancy guard within the createIncentive function.
Comments / Resolution	Acknowledged.

Issue_14	Incentive struct complexity requires pragma ABI coderv2 to be read
Severity	**Informational**
Description	The Incentive struct in UniswapV2Staker contains 13 fields, which exceeds the number of variables that can be efficiently accessed via the default auto-generated getter from a public mapping. It is likely that the stack-too-deep issue willl be encountered when writing Foundry tests, or via on-chain reads.
Recommendations	Write an explicit external getter function that returns the Incentive struct in memory. Requires pragma abicoder v2; Alternatively, acknowledge this issue.
Comments / Resolution	Acknowledged.

Issue_15	unusedRewards update does not include dust amount between incentive.totalReward and promisedRewards
Severity	**Informational**
Description	When an incentive ends, the dust calculated from the difference between incentive.totalReward and promisedRewards is refunded together with the unclaimed rewards, but this dust amount isn't added into incentive.unclaimedRewards
Recommendations	Consider incrementing unclaimedRewards by the dust amount.
Comments / Resolution	Acknowledged.

Issue_16	Inconsistent error names
Severity	**Informational**
Description	Some errors come with the contract name and function prefix, eg. UniswapV2Staker::createIncentive, while others don't.
Recommendations	It is ideal for the errors name style to be consistent, either keep the prefix or remove it.
Comments / Resolution	Acknowledged.

Global

Issue_17	Deployment with solidity pragma >= 0.8.0 will break the math
Severity	**High**
Description	The solidity pragma logic has been modified to support compilation with versions above 0.7.6. These versions implement an internal overflow/underflow protection. This is a problem because Uniswap works with inherent underflow/overflow assumptions for certain parts of the code. Reverting on these occasions will break the full contract. This applies to all contracts within the UniswapV3 repository.
Recommendations	Consider not allowing deployments with versions > 0.7.6
Comments / Resolution	Acknowledged. “All Uniswap V3 contracts are compiled and deployed strictly with Solidity 0.7.6, as required by Uniswap’s arithmetic assumptions. Deployments with Solidity ≥0.8.0 are intentionally disallowed.”

ACL

The ACL contract is a simple registry-like contract which allows the contract owner to define multiple configurations which are consulted by the UniswapV2Pool contract. These are the following:

- **General Whitelist Restriction**: This feature allows the contract owner to fully pause swaps for non-whitelisted addresses

- **Individual Whitelist**: In the scenario of the general whitelist, only whitelisted individuals can execute swaps

- **Block Delay**: This value exposes a swap delay which is bound per msg.sender

- **Block Delay Disabled**: This feature allows to globally disable swap delays

Notably, contrary to the V2 ACL implementation, whitelist is enabled by default for V3

Privileged Functions

- setWhitelisted
- setWhitelistEnabled
- setBlockDelay
- setBlockDelayDisabled

Issue_18	Governance Privilege: Blocking of swaps
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, governance can simply set a large blockDelay or enable a global whitelist without having any individual whitelisted.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

Issue_19	Whitelisting is enabled by default
Severity	**Low**
Description	Currently, this contract has whitelisted enabled by default which is inconsistent with the V2 pool. After consultation with the development team, it is intended to deactivate the whitelisted mechanism for the new deployment.
Recommendations	Consider setting this variable to false.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

UniswapV3PoolDeployer

The UniswapV3PoolDeployer contract is forked from Uniswap's UniV3PoolDeployer contract and modified to be compatible as standalone contract, callable by the UniswapV3Factory only.

Privileged Functions

- none

Core Invariants:

INV 1: Only the UniswapV3Factory can call the deploy function

No issues found.

UniswapV3Factory

The UniswapV3Factory contract is forked from Uniswap's UniswapV3Factory contract and is modified in an effort to support the ACL mechanism and outsourcing the UniswapV3PoolDeployer.

Privileged Functions

- setACL
- setDeployer
- enableFeeAmount

Core Invariants:

INV 1: The UniswapV3Deployer address must never be changed

Issue_20	Governance Privilege: Change of UniswapV3PoolDeployer address can result in stolen approvals and other harmful actions
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. - DoS of new deployments - Stealing of approvals - Stealing of liquidity addition The most sophisticated attack is the stealing of approvals which requires the following set of executions: a) Change UniswapV3PoolDeployer to a malicious implementation b) Deploy such a contract via the factory c) Users approve funds to NonfungiblePositionManager for regular operations d) Malicious deployed contract calls LiquidityManagement.uniswapV3Callback and consumes any approved funds. This works because the callbackValidation logic considered the UniswapV3Factory.getPool function as source of truth, which contains the malicious pool address.
Recommendations	Consider making UniswapV3PoolDeployer immutable.
Comments / Resolution	Acknowledged.

Issue_21	Concerns due to forfeiting deterministic address logic
Severity	**Low**
Description	Within this version, the deterministic approach of fetching a pairs address is not supported: *function computeAddress(address factory, PoolKey memory key) internal pure returns (address pool) {* *require(key.token0 < key.token1);* *pool = address(* *uint256(* *keccak256(* *abi.encodePacked(* *hex'ff',* *factory,* *keccak256(abi.encode(key.token0, key.token1, key.fee)),* *POOL_INIT_CODE_HASH* *)* *)* *)* *);* *}* This can not only expose problems for protocol internal contracts which are out of scope but also for integrators that assume this protocol follows the same deterministic address procedure as UniswapV3.
Recommendations	Consider conducting careful due-diligence if anything breaks due to that change. Additionally, integrators must be informed about this behavior.
Comments / Resolution	Acknowledged.

UniswapV3Pool

The UniswapV3Pool contract is forked from Uniswap's UniswapV3Pool contract and implements a similar whitelisting and _throttle mechanism as the UniswapV2Pool. We refer to the corresponding appendix.

On top of that, the protocolFee is already set during initialization.

Privileged Functions

- None

Core Invariants:

INV 1: If whitelisting is activated, only whitelisted addresses can swap and flash

INV 2: Every address must go through _throttle for swaps and flash unless the blockDelay is zero or the caller is exempted from the blockDelay

Issue_22	Throttling is applied per msg.sender
Severity	High
Description	The UniswapV3Pool contract integrates an access-control and rate-limiting layer into swap via an external ACL obtained from IUniswapV2Factory(factory).acl(). If ACL.whitelistedEnabled() is active, swap requires ACL.isWhitelisted(msg.sender). Independently, the pool applies _throttle(msg.sender, _acl) which enforces a minimum block delay between consecutive calls for the same caller address by checking block.number > lastCallBlock[msg.sender] + blockDelay and then updating lastCallBlock[msg.sender] = block.number. This throttle executes before any token transfers, callbacks, or reserve updates, and reverts the entire swap if the delay condition is not satisfied. The throttle accounting is keyed on msg.sender only. In typical usage patterns, swap is not called by the end user directly, but by a shared routing contract such as a canonical router or aggregator. If the system's whitelisting policy forces swaps through such a shared router address, then all user swaps share a single throttle bucket because msg.sender is always the router. This creates a shared-rate-limit surface where any single swap routed through the shared caller updates lastCallBlock[router] and blocks subsequent swaps through that same caller for the configured blockDelay. The mechanism therefore does not throttle per end user in the common routed execution path and instead throttles an entire class of users collectively. Under configurations where a shared router is the primary or only whitelisted swap caller, an attacker can grief by repeatedly triggering swaps via that shared router to continuously refresh

	lastCallBlock[router], causing unrelated users' swaps through the same router to revert with "BD" for as long as the attacker sustains activity at the required cadence. The impact is denial of service and disruption of trading availability through the canonical path, with secondary effects including degraded execution reliability for integrators, increased failure rates for multi-call routes that depend on completing the swap step, and selective suppression of market activity through the most common entrypoint. Please note that this issue does **not correspond** to the flash function, since this function is usually invoked by individuals and not via a shared router.
Recommendations	Consider implementing a rate limitation on the router level. It has to be noted that this will still remain an issue for aggregators and integrators that also have a shared entrypoint.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: "The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security."

Issue_23	Sybil-attack allows for bypassing msg.sender limitation in case of disabled whitelist.
Severity	**Medium**
Description	When global whitelisting is disabled, any address can call swap/flash, and the throttle becomes an address scoped limiter keyed purely by msg.sender. There is no binding to an end user identity, transaction origin, or any shared actor concept. As a result, a single actor can bypass the throttle by distributing calls across multiple caller addresses. Each new EOA begins with an unset lastCallBlock entry and therefore passes the throttle condition irrespective of prior throttled calls performed by the same actor via other addresses.
Recommendations	As with most sybil attacks, there is no solution to prevent these.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

Issue_24	Off-by-one issue during blockDelay enforcement
Severity	**Informational**
Description	The throttle check uses a strict inequality: require(block.number > last + delay, "BD"). This enforces that the caller can only execute again in a block strictly greater than last + delay. If blockDelay is expected to represent the number of blocks that must elapse before a new call is permitted, then the strict comparison enforces an additional block of delay beyond the configured value. For example, if a caller executes at block 100 with blockDelay = 1, a natural reading of “1 block delay” permits the next call at block 101 and disallows swaps at block 100, but the current logic only permits it starting at block 102. The behavior is therefore off by one relative to the common interpretation of block delay parameters.
Recommendations	Consider switching to >=.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

Issue_25	Duplicate whitelistedEnabled check
Severity	**Informational**
Description	The acl.whitelistedEnabled() is called twice, one instance in the swap() and flash() functions, the other in _checkWhitelisted. This is redundant as the control-flow ensures that this whitelisting is true whenever _checkWhitelisted is invoked.
Recommendations	Consider removing the duplicate check within _checkWhitelisted.
Comments / Resolution	Acknowledged, the team indicated that the ACL feature will be disabled in production: “The ACL mechanism (whitelisting, blockDelay, throttling) is disabled by default and not intended to be enabled in production deployments. Therefore, issues related to ACL enforcement and throttling do not affect operational security.”

NonfungiblePositionManager

The UniswapV3Pool contract is forked from Uniswap's UniswapV3Pool contract and no changes have been made besides a more loose solidity pragma implementation

No issues found - refer to global issue.

UniswapV3Staker

The UniswapV3Staker contract is a reward distribution contract for UniswapV3 NFT positions which allows for the permissionless creation of incentives which reward stakers over an incentive period.

Appendix: Stake / Deposit

The contract incorporates a distinct difference between deposited and staked liquidity. Deposited liquidity essentially reflects a tokenId which has been transferred to this contract and triggered the onERC721Received function. It is then assigned to the “from” address, which was the initial owner of the tokenId.

Subsequently, once successfully deposited, the owner can then invoke the stakeToken function which allows a user to stake his tokenId to one or multiple different incentiveIDs

Appendix: Seconds Per Liquidity

The seconds per liquidity is a metric which was invented by UniswapV3 and essentially illustrates how many seconds a specific liquidity fragment was active combined to the overall liquidity. This mechanism is required because liquidity constantly changes via adding or removing liquidity and therefore it is required to recompute the average share of a liquidity fragment based on multiple different epochs.

Consider the overall liquidity within a specific range is 100e18 and Alice owns 10e18 within this liquidity range. If now for example, 100 seconds are passed, while the range was active, the seconds per liquidity will be calculated as:

> secondsPerLiquidityX128 = 100 << 128 / 100e18

> secondsPerLiquidityX128 = 340282366920938463463

Using this value, we can now determine Alice’s share on seconds per liquidity as:

> secondsPerLiquidityX128Alice = 340282366920938463463 * 10e18

> secondsPerLiquidityX128Alice = 3402823669209384634630000000000000000000

Note that in Uniswap, the denominator for the actual global cumulative value is the **active liquidity:**

> cumulativeSecondsPerLiquidity = secondsDelta / activeLiquidity

and this contract uses the oracle to fetch the value within the specified range. If therefore, the range was active, it will increase the secondsPerLiquidityInsideX128 value.

Appendix: Reward Amount Computation

TLDR: Distribute rewards based on the weighted average of a user's liquidity proportionally to the overall liquidity for the same range for the duration where the liquidity was active, within a determined time period.

A reward mechanism has been implemented which is based on the secondsPerLiquidityX128 concept, which is described above. Each incentive is tied to a specific range of [tickLower; tickUpper] and a pool/token and time window and each specific range has a corresponding secondsPerLiquidityInsideX128 at all times.

This value is calculated via the UniswapV3Pool.snapshotCumulativesInside function and is based on lower.secondsPerLiquidityOutsideX128 and upper.secondsPerLiquidityOutsideX128 and on the current position of [tickLower; tickUpper] within the pool:

Scenario A: Current tick below range (T < tickLower)

Returns "inside" values by subtracting upper boundary "outside" from lower boundary "outside":

- secondsPerLiquidityInsideX128 = L.secondsPerLiquidityOutsideX128 - U.secondsPerLiquidityOutsideX128

Scenario B: Current tick inside range (tickLower <= T < tickUpper)

Computes current cumulative values via oracle and subtracts both boundary "outside" values:

- secondsPerLiquidityInsideX128 = secondsPerLiquidityCumulativeX128 - L.secondsPerLiquidityOutsideX128 - U.secondsPerLiquidityOutsideX128

Scenario C: Current tick above range (T >= tickUpper)

Returns “inside” values by subtracting lower boundary “outside” from upper boundary “outside”:

- secondsPerLiquidityInsideX128 = U.secondsPerLiquidityOutsideX128 - L.secondsPerLiquidityOutsideX128

Whenever a new stake is happening, it will fetch secondsPerLiquidityInsideX128 and marks it as the start of the stake duration. If now unstake is happening at a later point, it will fetch the most updated secondsPerLiquidityInsideX128 value for the range and determines the delta between after and before and multiplies it with the staked liquidity from the user:

> secondsInsideX128 = (secondsPerLiquidityInsideX128Now - secondsPerLiquidityInsideX128Initial) * liquidity

This will return the average weighted share of a user’s liquidity, as a “**position’s share seconds**” which is used as a source of truth for the reward calculation in context to the overall incentive seconds. An intermediate step will now calculate the value of the “**total share seconds**” over which the remaining rewards are allocated:

> totalSecondsUnclaimedX128 = (max(endTime, block.timestamp) - startTime << 128) - totalSecondsClaimedX128

The last step will now simply calculate the proportional amount of a users “**position share seconds**” on the “**total share seconds**” and the remaining unclaimed rewards:

> reward = totalRewardUnclaimed * secondsInsideX128 / totalSecondsUnclaimedX128

Afterwards, totalRewardUnclaimed is decreased by the claimed reward amount and totalSecondsClaimedX128 is increased by the secondsInsideX128 amount

Privileged Functions

- none

Core Invariants:

INV 1: Only the owner of a tokenId can call transferDeposit

INV 2: Only the owner of a tokenId can call withdrawToken

INV 3: withdrawToken is only callable if tokenId is staked in no incentive

INV 4: An incentive can only be created for block.timestamp or in the future

INV 5: An incentive cannot start too far in the future (maxIncentiveStartLeadTime)

INV 6: Depositing rewards for an existing incentiveId must increment the reward amount

INV 7: Staking is only allowed >= startTime and < endTime

INV 8: A token can only be staked for an existing incentive

INV 9: Unstake token can be called by anyone after incentive time has ended and only by the owner before incentive time has ended

INV 10: An incentive should never pay out more than what was initially provided

INV 11: Incentive duration cannot exceed the maxIncentiveDuration

INV 12: Only the NonfungiblePositionManager is allowed to call onEC721Received

INV 13: A tokenId cannot be staked to the same incentiveId twice (at the same time)

INV 14: A tokenId can be staked to multiple different incentiveIds at the same time

INV 15: Incentive can only end once endTime has been reached

INV 16: Incentive can only end once numberOfStakes = 0

Issue_26	Lack of support for transfer-tax tokens will create immediate insolvency state
Severity	**Medium**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting. For instance, incentive creation does not apply the before/after pattern. This means the contract will receive less tokens than it will account for, creating an immediate insolvency scenario.
Recommendations	Consider applying the before/after pattern within createIncentive. IMPORTANT: The createIncentive function must expose the reentrancy guard if such a pattern is implemented
Comments / Resolution	Acknowledged.

Issue_27	Current reward design exposes various limitations
Severity	**Medium**
Description	UniswapV3Staker distributes incentive rewards to staked Uniswap V3 LP NFTs by measuring a position's time-weighted share of in-range liquidity using Uniswap V3's secondsPerLiquidityInsideX128 primitive. Upon staking, the contract snapshots secondsPerLiquidityInsideX128 for the NFT's [tickLower, tickUpper] range via IUniswapV3Pool.snapshotCumulativesInside and stores it as secondsPerLiquidityInsideInitialX128. Upon unstaking, it takes a new snapshot, computes secondsInsideX128 = (ΔsecondsPerLiquidityInsideX128) * liquidity, and pays rewards proportionally against a remaining "seconds capacity" and remaining unclaimed reward. After endTime, unstakeToken is permissionless, and remaining rewards can be refunded to refundee only once all stakes are removed. The reward mechanism does **not** clamp accrual to the incentive window [startTime, endTime]. On unstake, the contract always uses the **current** secondsPerLiquidityInsideX128 returned by snapshotCumulativesInside, even when block.timestamp > endTime. Because secondsPerLiquidityInsideX128 continues to evolve after endTime (depending on whether the pool tick remains in range and what the in-range active liquidity is), the credited secondsInsideX128 can include post-end accrual. The reason why this was developed in such a manner lies within the underlying concept of the pool for secondsPerLiquidityCumulative, as there is no way to retrieve this value for a range at a determined timestamp. It is only possible to retrieve it at block.timestamp. The current reward logic exposes multiple minor issues which stem from a design limitation. For example, the post-end accrual can result in some rewards which were considered to be refunded to be allocated to late unstakers, which is derived from the fact that the calculation is still ongoing post-end while the active liquidity can change.

The root-cause of this issue is the fact that reward calculation continues post-end while the range's total liquidity can change.

Example A: Rewards are increased

Initial state at t1 = 0
Alice's liquidity: 80
Bob's liquidity: 10
Charles's liquidity: 10
totalLiquidityT0 = 80 + 10 + 10 = 100
startTime = t0 = 0
endTime = t1 = 100
totalRewardUnclaimed = 100 tokens
totalSecondsUnclaimed = max(currentTime, endTime) - startTime = 100
totalLiquidityT0 = 100
secondsPerLiquidity = 0

- Alice and Bob stakes their positions throughout the incentive duration, Charles does not stake his position
- Expected reward distribution:
  - Alice: 80 tokens
  - Bob: 10 tokens
  - Refundee: 10 tokens

State at t1 = 100
secondsPerLiquidity += timeDelta / liquidity
= (100 - 0) / 100
= 1

- Alice unstakes her position, receives 80 tokens and withdraws liquidity
  - secondsInside
    = secondsPerLiquidityT1 * positionLiquidity
    = 1 * 80
    = 80
  - reward
    = totalRewardUnclaimed * secondsInside / totalSecondsUnclaimed

= 100 * 80 / 100
= 80
- totalSecondsClaimed += secondsInside
= 80
- totalRewardUnclaimed -= reward
= 100 - 80
= 20

- Charles withdraws liquidity; Bob becomes the only staker
  - totalLiquidityT1 = 10

At this point, Bob is supposed to receive 10 tokens. However, Bob remains staked for longer, until t2 = 300 and now unstakes.

State at t2 = 300
totalLiquidityT2 = 10
secondsPerLiquidity += timeDelta / liquidity
= 1 + (300 - 100) / 10
= 21
secondsInside = 21 * 10 = 210
totalSecondsUnclaimed = (300 - 0) - 80 = 220
reward = 20 * 210 / 220 = 19.09

As one can see, Bob receives more rewards because he remains in the staked liquidity. Post-end logic reroutes rewards to Bob.

##

Example B: Rewards are decreased
If at t1, instead of withdrawing, Charles increases his position liquidity to 200 while Bob remains staked, the logic at t2 would be:

secondsPerLiquidity += 200 / (200 + 10)
= 1 + 200 / 210
= 1.952

State at t2 = 300
Bob unstakes
- secondsInside = 1.952 * 10 = 19.52

- totalSecondsUnclaimed = (300 - 0) - 80 = 220
- reward = 20 * 19.52 / 220 = 1.77

##

Example C: Later unstaker will receive rewards from earlier unstaker

Another iteration now highlights how Alice will receive a reduced reward amount due to post-end diluting while Bob receives an increased reward amount

t0 = 0
t1 = 100
t2 = 200
t3 = 500
rewards = 100
secondsPerLiquidityT0 = 0
secondsPerLiquidityT1 = 1
secondsPerLiquidityT2 = 1.1
secondsPerLiquidityT3 = 31.1

Alice stakes 90 tokens over the full duration
Bob stakes 10 tokens over the full duration

At t1 = 100, Alice would receive 90 tokens and Bob would receive 10 tokens. But none of them unstakes at this juncture.

Charles adds 900 tokens in the corresponding range.

Alice unstakes at t2 = 200:

- secondsInside = 1.1 * 90
- secondsInside = 99
- totalSecondsUnclaimed = 200
- reward = 100 * 99 / 200
- reward = 49.5
- totalSecondsClaimed = 99

	- totalUnclaimedRewards = 51.5 Alice's rewards are now decreased because they have been diluted with Charles' liquidity addition. Alice fully withdraws her liquidity and Charles as well. Now starting from t2 = 200, Bob's liquidity is the sole liquidity within the range. Bob unstakes at t3 = 500 - secondsInside = 311 - totalSecondsUnclaimed = 401 - rewards = 51.5 * 311 / 401 - rewards = 39.94 Due to the fact that Alice withdrew after a period where she was diluted and the fact that Bob withdrew after a period where he was the sole liquidity provider, Bob essentially consumed rewards from Alice.
Recommendations	This issue is considered as known and constrained by the design limitation of the UniswapV3Staker contract. The only fix would be to develop a new contract which mimics Algebra's farming system: https://github.com/cryptoalgebra/Algebra/tree/v1.0-integral/src/farming/contracts We recommend acknowledging this issue.
Comments / Resolution	Acknowledged, this is a design limitation.

Issue_28	Lack of minimum duration for new incentive
Severity	**Low**
Description	The createIncentive function enforces various different validations but lacks a minimum incentive enforcement which allows to create an incentive for just 1 block. This could be a precondition for an exploit in the scenario where a bug remains existing. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider implementing a minimum duration (e.g. 1 day) for an incentive to increase robustness of the contract.
Comments / Resolution	Acknowledged.

Issue_29	Incentive can never be ended if refundee is blacklisted for token
Severity	**Informational**
Description	The endIncentive function attempts to transfer dust/unused funds to the refundee. This will revert if the refundee is blacklisted and will prevent the incentive ending.
Recommendations	Consider implementing a recipient address which will be the transfer beneficiary.
Comments / Resolution	Acknowledged.

Issue_30	Lack of validation for incentive creation for invalid pool
Severity	**Informational**
Description	Currently, the createIncentive function does not validate the pool for an incentive. While that does not seem to be problematic because a validation exists within _stakeToken, it still increases the attack surface. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider validating that such a pool indeed exists by requiring tokenX/tokenY to be passed as parameters which then uses the Factory.getPool function to check whether such a pool indeed exists.
Comments / Resolution	Acknowledged.

RewardMath

The RewardMath library is a simple helper which calculates the received reward amount. All technical information can be found above.

No issues found.