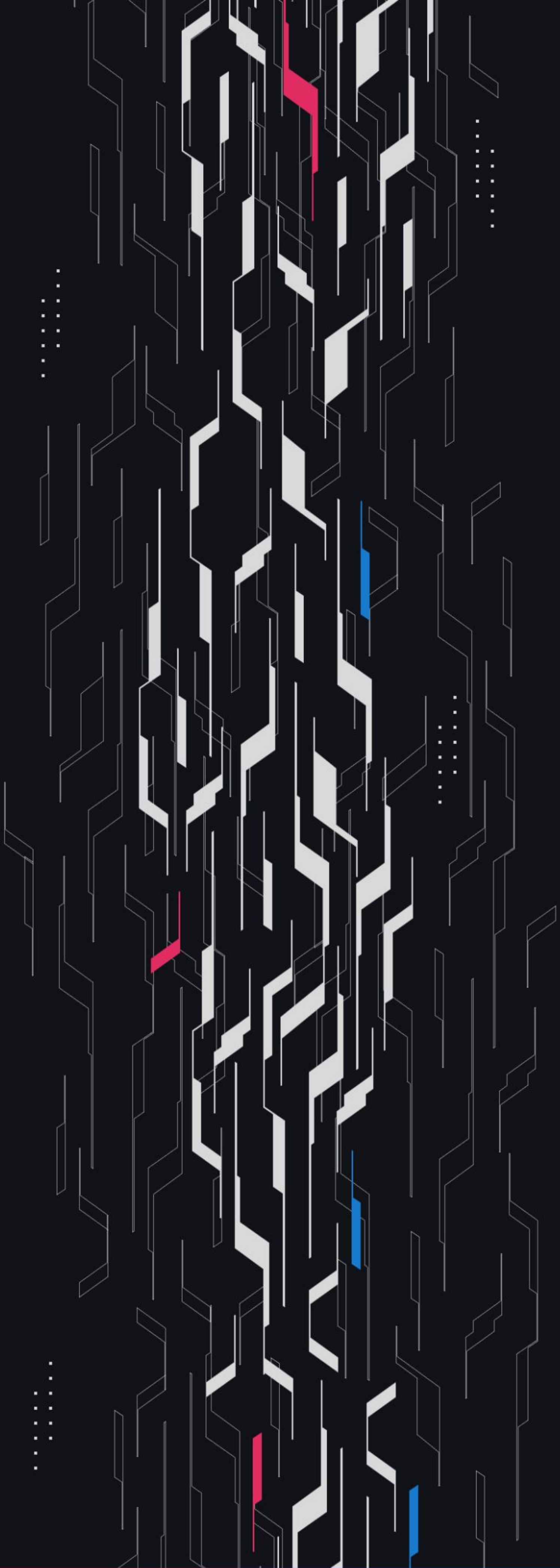
GUARDIAN

# M0

## Uniswap V4 Hooks

## Security Assessment

June 26th, 2025



# Summary

**Audit Firm** Guardian

**Prepared By** Curiousapple, Roberto Reigada, Nicholas Chew, 0xCiphkey, Vladimir Zotov

**Client Firm** M0

**Final Report Date** June 26, 2025

## Audit Summary

M0 engaged Guardian to review the security of their M0's Uniswap V4 Hooks. From the 2nd of June to the 4th of June, a team of 5 auditors reviewed the source code in scope. All findings have been recorded in the following report.

## Confidence Ranking

Given the number of non-critical issues detected and code changes following the main review, Guardian assigns a Confidence Ranking of 4 to the protocol. Guardian advises the protocol to consider periodic review with future changes. For detailed understanding of the Guardian Confidence Ranking, please see the rubric on the following page.

🔗 Blockchain network: **Ethereum, Arbitrum, Optimism, Unichain**

✅ Verify the authenticity of this report on Guardian's GitHub: https://github.com/guardianaudits

📊 Code coverage & PoC test suite: https://github.com/GuardianOrg/uniswap-v4-hooksm0-hooks-team1, https://github.com/GuardianOrg/uniswap-v4-hooksm0-hooks-team2, https://github.com/GuardianOrg/uniswap-v4-hooksm0-hooks-fuzz

# Guardian Confidence Ranking

| Confidence Ranking | Definition and Recommendation | Risk Profile |
|---|---|---|
| **5: Very High Confidence** | Codebase is mature, clean, and secure. No High or Critical vulnerabilities were found. Follows modern best practices with high test coverage and thoughtful design.<br>**Recommendation:** Code is highly secure at time of audit. Low risk of latent critical issues. | 0 High/Critical findings and few Low/Medium severity findings. |
| **4: High Confidence** | Code is clean, well-structured, and adheres to best practices. Only Low or Medium-severity issues were discovered. Design patterns are sound, and test coverage is reasonable. Small changes, such as modifying rounding logic, may introduce new vulnerabilities and should be carefully reviewed.<br>**Recommendation:** Suitable for deployment after remediations; consider periodic review with changes. | 0 High/Critical findings. Varied Low/Medium severity findings. |
| **3: Moderate Confidence** | Medium-severity and occasional High-severity issues found. Code is functional, but there are concerning areas (e.g., weak modularity, risky patterns). No critical design flaws, though some patterns could lead to issues in edge cases.<br>**Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes. | 1 High finding and ≥ 3 Medium. Varied Low severity findings. |
| **2: Low Confidence** | Code shows frequent emergence of Critical/High vulnerabilities (~2/week). Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues.<br>**Recommendation:** Post-audit development and a second audit cycle are strongly advised. | 2-4 High/Critical findings per engagement week. |
| **1: Very Low Confidence** | Code has systemic issues. Multiple High/Critical findings (≥5/week), poor security posture, and design flaws that introduce compounding risks. Safety cannot be assured.<br>**Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring. | ≥5 High/Critical findings and overall systemic flaws. |

# Table of Contents

**Project Information**

**Smart Contract Risk Assessment**

**Addendum**

# Project Overview

## Project Summary

| | |
|---|---|
| Project Name | M0 |
| Language | Solidity |
| Codebase | https://github.com/m0-foundation/uniswap-v4-hooks |
| Commit(s) | Initial commit(s): 130696e77740c3b1a63803df15547db704600fc0<br>Final commit: 4a223d96073de95c0e4f0f7fa9fbf9ce3ccfb369 |

## Audit Summary

| | |
|---|---|
| Delivery Date | June 26, 2025 |
| Audit Methodology | Static Analysis, Manual Review, Test Suite, Contract Fuzzing |

## Vulnerability Summary

| Vulnerability Level | Total | Pending | Declined | Acknowledged | Partially Resolved | Resolved |
|---|---|---|---|---|---|---|
| ● Critical | 0 | 0 | 0 | 0 | 0 | 0 |
| ● High | 0 | 0 | 0 | 0 | 0 | 0 |
| ● Medium | 3 | 0 | 0 | 2 | 0 | 1 |
| ● Low | 13 | 0 | 0 | 11 | 0 | 2 |
| ● Info | 0 | 0 | 0 | 0 | 0 | 0 |

# Audit Scope & Methodology

Scope and details:

```
contract,source,total,comment
./src/AllowlistHook.sol,216,408,119
./src/TickRangeHook.sol,22,57,27
./src/abstract/BaseTickRangeHook.sol,67,140,45
source count: {total: 605, source: 305, comment: 191, single: 35, block: 156, mixed: 0, empty: 109, todo: 0,
blockEmpty: 0, commentToSourceRatio: 0.6262295081967213}
```

# Audit Scope & Methodology

## Vulnerability Classifications

| **Severity** | Impact: *High* | Impact: *Medium* | Impact: *Low* |
|---|---|---|---|
| Likelihood: *High* | ● Critical | ● High | ● Medium |
| Likelihood: *Medium* | ● High | ● Medium | ● Low |
| Likelihood: *Low* | ● Medium | ● Low | ● Low |

## Impact

**High** Significant loss of assets in the protocol, significant harm to a group of users, or a core functionality of the protocol is disrupted.

**Medium** A small amount of funds can be lost or ancillary functionality of the protocol is affected. The user or protocol may experience reduced or delayed receipt of intended funds.

**Low** Can lead to any unexpected behavior with some of the protocol's functionalities that is notable but does not meet the criteria for a higher severity.

## Likelihood

**High** The attack is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount gained or the disruption to the protocol.

**Medium** An attack vector that is only possible in uncommon cases or requires a large amount of capital to exercise relative to the amount gained or the disruption to the protocol.

**Low** Unlikely to ever occur in production.

# Audit Scope & Methodology

## Methodology

Guardian is the ultimate standard for Smart Contract security. An engagement with Guardian entails the following:

- Two competing teams of Guardian security researchers performing an independent review.
- A dedicated fuzzing engineer to construct a comprehensive stateful fuzzing suite for the project.
- An engagement lead security researcher coordinating the 2 teams, performing their own analysis, relaying findings to the client, and orchestrating the testing/verification efforts.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross-referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts. Comprehensive written tests as a part of a code coverage testing suite.
- Contract fuzzing for increased attack resilience.

# Invariants Assessed

During Guardian's review of M0, fuzz-testing was performed on the protocol's main functionalities. Given the dynamic interactions and the potential for unforeseen edge cases in the protocol, fuzz-testing was imperative to verify the integrity of several system invariants.

Throughout the engagement the following invariants were assessed for a total of 10,000,000+ runs with a prepared fuzzing suite.

| ID | Description | Tested | Passed | Remediation | Run Count |
|---|---|---|---|---|---|
| G-01 | Pool tick must be 0 or 1 after swap | ✅ | ✅ | ❌ | 10M+ |
| AL-01 | Liquidity providers allowlist state mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-02 | Swappers allowlist state mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-03 | Liquidity provider allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-04 | Swapper allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-05 | Batch liquidity provider allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-06 | Batch swapper allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-07 | Batch swapRouter allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-08 | Batch position manager allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |
| AL-09 | Batch position manager allowlist status mismatch | ✅ | ✅ | ✅ | 10M+ |

# Findings & Resolutions

| ID | Title | Category | Severity | Status |
|---|---|---|---|---|
| M-01 | Tick Range Update Could Lead To Pool DoS | Warning | ● Medium | Acknowledged |
| M-02 | Missing Domain Separator | Validation | ● Medium | Acknowledged |
| M-03 | Token Sorting Not Accounted | Logical Error | ● Medium | Resolved |
| L-01 | Unused PositionManagerStatus.REDUCE_ONLY | Warning | ● Low | Resolved |
| L-02 | Potential For Front-running & Griefing | Warning | ● Low | Acknowledged |
| L-03 | ZeroForOne Swap Logic Breaks Tick–Price Assumption | Warning | ● Low | Acknowledged |
| L-04 | Predicate Message omits sqrtPriceLimitX96 | Validation | ● Low | Acknowledged |
| L-05 | Allow/Deny-list Depends On msgSender() | Warning | ● Low | Acknowledged |
| L-06 | Tick Range Risk Due To Paired Asset Volatility | Warning | ● Low | Acknowledged |
| L-07 | Pectra Upgrade Enables EOAs | Validation | ● Low | Acknowledged |
| L-08 | Potential Front-run DoS | Warning | ● Low | Acknowledged |
| L-09 | TickRangeHook Does Not Support Multiple Pools | Warning | ● Low | Acknowledged |
| L-10 | WRAPPED_M Missing On Unichain | Warning | ● Low | Acknowledged |

# Findings & Resolutions

| ID | Title | Category | Severity | Status |
|---|---|---|---|---|
| L-11 | Donate() Not Guarded By Hook Logic | Warning | ● Low | Resolved |
| L-12 | Pools Can Be Initialized With An Out-Of-Range Tick | Warning | ● Low | Acknowledged |
| L-13 | Pool Price Is Initialized At The Lower Bound | Warning | ● Low | Resolved |

# M-01 | Tick Range Update Could Lead To Pool DoS

| Category | Severity | Location | Status |
|---|---|---|---|
| Validation | ● Medium | TickRangeHook.sol | Acknowledged |

## Description

In the TickRangeHook, liquidity providers are restricted to adding liquidity within an allowed tick range. For example, let's imagine that the pool operates with a current tick of 5 and the allowed tick range is set to [0, 10].

This means LPs have provided liquidity only between ticks 0 and 10 and the pool functions normally within this range. Then, the setTickRange(20, 30) function is called by a manager. This updates the allowed tick range to [20, 30], while the current tick remains at 5.

After this update, LPs can only add new liquidity within the new range of [20, 30]. However, in case there is a big amount of liquidity in the ticks between 5 and 20, the slot0.tick will not reach the new lower bound (20) unless the swap atomically consumes all the liquidity placed between the ticks 5 and 20.

Therefore, such setTickRange update, could cause a Denial of Service in the Uniswap V4 pool unless the liquidity providers remove the liquidity from those ticks (5 to 20) after the update. There is no guarantee that this will occur.

## Recommendation

Consider updating the _afterSwap implementation to allow a swap as long as the new getTickAtSqrtPrice(slot0.sqrtPriceX96) resultant tick after the swap is closer to the pertinent new range bound.

This should only be allowed for swaps that starts on a tick that is already out of the valid range.

## Resolution

M0 Team: In our case, we plan to only have liquidity between tick range [0,1] , so the described scenario should not occur. We are currently debating if we should keep the tick range check since there can be some undesirable side effects like the one described in this issue.

# M-02 | Missing Domain Separator

| Category | Severity | Location | Status |
|---|---|---|---|
| Validation | ● Medium | AllowlistHook.sol | Acknowledged |

## Description

In the AllowlistHook._beforeSwap function, the encodeSigAndArgs method encodes parameters for the Predicate's authorization process. However, this encoding does not include a domain separator or any chain-specific field, such as the chain ID and, therefore, signatures generated for a swap on one chain could potentially be reused on another chain where the Uniswap V4 PoolManager and Predicate's ServiceManager contracts are deployed.

If the same addresses and parameters are valid across chains, this could lead to unauthorized swaps. The current implementation of encodeSigAndArgs is as follows:

```
bytes memory encodeSigAndArgs_ =
abi.encodeWithSignature("_beforeSwap(address,address,address,uint24,int24,address,boo
l,int256)", caller_, key_.currency0, key_.currency1, key_.fee, key_.tickSpacing,
address(key_.hooks), params_.zeroForOne, params_.amountSpecified);
```

This encoding includes the caller, pool key details (currencies, fee, tick spacing, hooks address) and swap parameters (direction and amount), but lacks any identifier tying the signature to a specific chain. In cross-chain environments, this omission is a significant risk, as signatures could be replayed on unintended chains where the same contract addresses and parameters exist.

## Recommendation

To mitigate the risk of cross-chain signature reuse, modify the encodeSigAndArgs function to include a domain separator. At a minimum, add the chain ID to the encoded parameters to ensure signatures are chain-specific. An updated version could look like this:

```
bytes memory encodeSigAndArgs_ =
abi.encodeWithSignature("_beforeSwap(uint256,address,address,address,uint24,int24,add
ress,bool,int256)", block.chainid, caller_, key_.currency0, key_.currency1, key_.fee,
key_.tickSpacing, address(key_.hooks), params_.zeroForOne, params_.amountSpecified);
```

For a more robust solution, consider adopting a full EIP-712-compliant domain separator.

## Resolution

M0 Team: Acknowledged.

# M-03 | Token Sorting Not Accounted

| Category | Severity | Location | Status |
|---|---|---|---|
| Logical Error | ● Medium | Config.sol: 25-28 | Resolved |

## Description

From the deployment config, it appears that the M0 team intends to set the tick lower and upper bounds as 0 to 1. This corresponds to a price range where token0 is valued between 1e18 and 1.0001e18 token1. However, since Uniswap determines token0 and token1 based on lexicographic address sorting, it is not guaranteed that wrapped M will always be assigned as token0.

If wrapped M becomes token1, then the same tick range (0 → 1) would apply to the inverse price (token1/token0), flipping the interpretation. In that case, the tick range 0 → 1 would represent a price range of 0.999999e18 to 1e18 wrapped M per 1 USDC, which is likely the opposite of the intended bound. As per M0's config, they aim to deploy wrappedM:USDC pools across 4 chains:

```
address public constant USDC_ETHEREUM = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;
address public constant USDC_ARBITRUM = 0xaf88d065e77c8cC2239327C5EDb3A432268e5831;
address public constant USDC_OPTIMISM = 0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85;
address public constant USDC_UNICHAIN = 0x078D782b760474a361dDA0AF3839290b0EF57AD6;
```

Given this, token sorting across the pairs is as follows:

| Chain | token0 | token1 |
|---|---|---|
| Ethereum | 0x437c… (wrapped M) | 0xA0b8… (USDC) |
| Arbitrum | 0x437c… (wrapped M) | 0xaf88… (USDC) |
| Optimism | 0x0b2C… (USDC) | 0x437c… (wrapped M) |
| Unichain | 0x078D… (USDC) | 0x437c… (wrapped M) |

For Ethereum and Arbitrum, wrapped M remains token0, so the tick range 0 → 1 works as expected. However, for Optimism and Unichain — where wrapped M becomes token1 — this tick range binds the price of wrapped M from ~0.999999 to 1 USDC, effectively inverting the bound and constraining the price in the wrong direction.

## Recommendation

Consider making the deployment script dynamic — accounting for token order and flipping tick ranges accordingly — or be aware of this possibility and choose tick ranges manually based on token sorting

## Resolution

M0 Team: Resolved.

# L-01 | Unused PositionManagerStatus.REDUCE_ONLY

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | AllowlistHook.sol | Resolved |

## Description

In the AllowlistHook contract, the PositionManagerStatus enum defines three possible states for position managers: ALLOWED, REDUCE_ONLY, and an implicit FORBIDDEN (when no status is explicitly set). However, the contract's logic for controlling liquidity addition only checks if the sender's status is ALLOWED. The relevant code in the _beforeAddLiquidity function is:

```
if (_positionManagers[sender_] = PositionManagerStatus.ALLOWED) {revert
PositionManagerNotTrusted(sender_);}
```

This check means that any position manager not explicitly marked as ALLOWED, including those with REDUCE_ONLY status, will be rejected when attempting to add liquidity, effectively treating REDUCE_ONLY the same as FORBIDDEN. The REDUCE_ONLY status, which likely intends to allow position managers to remove liquidity but not add it, is not leveraged in any meaningful way within the hook.

Furthermore, since the hook does not implement a beforeRemoveLiquidity function, liquidity removal is not restricted by this status either. As a result, the distinction between REDUCE_ONLY and other non-ALLOWED states is redundant, adding unnecessary complexity to the contract.

## Recommendation

To simplify the AllowlistHook contract and eliminate the underutilized REDUCE_ONLY status, replace the PositionManagerStatus enum with a straightforward boolean mapping, such as mapping(address > bool) public isAllowedLPer. This mapping would store whether a position manager is permitted to add liquidity. The updated check in _beforeAddLiquidity would then be:

```
if (isAllowedLPer[sender_]) {revert PositionManagerNotTrusted(sender_);}
```

This change preserves the current functionality, only allowing trusted position managers to add liquidity, while reducing the contract's complexity and lowering gas costs by eliminating the need for an enum.

## Resolution

M0 Team: Resolved

# L-02 | Potential For Front-running & Griefing

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | TickRangeHook.sol | Acknowledged |

## Description

The BaseTickRangeHook contract enforces a specific tick range ([tickLowerBound, tickUpperBound]) for swaps via the _afterSwap function.

This function reverts any swap that causes the current tick to fall outside the designated range, ensuring trading occurs within a predefined price window. However, because of this requirement any swap can be front-run or griefed by another swap.

For example:

- A legitimate swap is submitted, which, if executed, would move the current tick to a value still within the allowed range.

- An attacker observes this pending swap in the mempool and submits a small, preemptive swap that shifts the current tick closer to the boundary of the allowed range (e.g., near tickLowerBound or tickUpperBound).

- When the legitimate swap executes, it builds on the pool state altered by the attacker's swap, pushing the current tick just outside the permitted range. This triggers a revert, preventing the legitimate swap from completing.

This allows an attacker to block and grief legitimate swaps by manipulating the pool’s tick position.

## Recommendation

Merely an informative issue as this is a direct consequence of the intended design.

## Resolution

M0 Team: This is by design and it is indeed possible that transactions may revert if a previous one shifts the current tick closer to the boundary.

# L-03 | ZeroForOne Swap Logic Breaks Tick–Price Assumption

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | BaseTickRangeHook.sol: 93-94 | Acknowledged |

## Description

As per Uniswap's swap logic, when `result.sqrtPriceX96 = step.sqrtPriceNextX96` at the end of a swap step, and the direction is `zeroForOne`, the protocol sets the current tick to `tickNext - 1`. This behavior is a known quirk in Uniswap's design and is documented in M-02 of this Certora audit report.

M0's `afterSwap` includes a validation step on the current tick. However, in the `zeroForOne` scenario described above, `slot0.tick` is set to `tickNext - 1`, which may not match `getTickAtSqrtPrice(sqrtPriceX96)` when the swap ends exactly at a tick boundary.

Example scenario:

- Initial setup: `sqrtPriceX96 = SSSSS` and hook tick range is [0, 1]
- A swap from `token1 → token0` for amount X moves the price slightly above tick 0
- A reverse swap `(token0 → token1)` for the same amount X brings the pool back to `sqrtPriceX96 = SSSSS`
- However, the current tick is now -1, not 0, due to Uniswap's internal handling in `zeroForOne` swaps

As a result, M0's `afterSwap` logic—which asserts the current tick to be ≥ 0—reverts, even though the price is correct and in-range. This creates an unintuitive edge case: preventing full symmetric swaps. This leads to:

- Unexpected reverts in contracts assuming full liquidity can be consumed
- Confusion for integrators: e.g., an M0 integrator sees liquidity X in the pool and attempts a full swap, but it fails due to this subtle tick mismatch

## Recommendation

Document this edge case explicitly for integrators relying on symmetric swap behavior or full-range liquidity visibility. For the stated example, swap goes through for X-1. Alternatively, you could consider deriving the current tick directly from `sqrtPriceX96` using `getTickAtSqrtPrice` in `afterSwap` hook. This will allow full use of liquidity; however, this would set `tick = -1` for 0 to 1 tick range case

## Resolution

M0 Team: Acknowledged

# L-04 | Predicate message omits sqrtPriceLimitX96

| Category | Severity | Location | Status |
|---|---|---|---|
| Validation | ● Low | AllowlistHook.sol | Acknowledged |

## Description

AllowlistHook._beforeSwap builds the payload that off-chain operators must sign as follows:

```
bytes memory encodeSigAndArgs_ =
abi.encodeWithSignature("_beforeSwap(address,address,address,uint24,int24,address,bool,int25
6)", caller_, key_.currency0, key_.currency1, key_.fee, key_.tickSpacing,
address(key_.hooks), params_.zeroForOne, params_.amountSpecified);
```

The sqrtPriceLimitX96 field (present in IPoolManager.SwapParams) is not part of the signed data and therefore:
• In exact-in swaps (amountSpecified < 0), the signer approves spending a fixed input amount but has no guarantee on the minimum output.
• In exact-out swaps (amountSpecified > 0), the signer approves delivering a fixed output amount but does not cap the maximum input. A price spike before execution can force the caller to pay far more than the signers deemed acceptable.
• Because the task hash lacks any price limit, the user can use a still-valid signature later (until expireByBlockNumber) when the price has shifted, while remaining within the signed amountSpecified.

While the end-user's router usually enforces its own amountOutMin/amountInMax to ensure the swap meets the user's minimum expectations, these protections are separate from the Predicate layer's role. The Predicate layer, which uses operator signatures to authorize swaps, does not include sqrtPriceLimitX96 in the signed data.

This omission means the signatures do not enforce any price constraints, allowing swaps to execute at uncontrolled prices despite the authorization. As a result, the Predicate layer fails to uphold the economic conditions (such as acceptable price or slippage) that the signers intended, weakening its effectiveness as a policy guard.

## Recommendation

Bind the signature to an explicit price or slippage limit by including sqrtPriceLimitX96 in the encoded arguments:

```
bytes memory encodeSigAndArgs_ =
abi.encodeWithSignature("_beforeSwap(address,address,address,uint24,int24,address,bool,int25
6)", caller_, key_.currency0, key_.currency1, key_.fee, key_.tickSpacing,
address(key_.hooks), params_.zeroForOne, params_.amountSpecified, params_.sqrtPriceLimitX96
// NEW FIELD);
```

## Resolution

M0 Team: Acknowledged.

# L-05 | Allow/Deny-list Depends On msgSender()

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | AllowlistHook.sol | Acknowledged |

## Description

To identify the ultimate user, the hook queries the router that invoked it:

```
address caller_ = IBaseActionsRouterLike(sender_).msgSender();
```

This is the expected approach as documented in the Uniswap V4 docs. If a malicious router is mistakenly placed in the trusted-router mapping (`_swapRouters` or `_positionManagers`), it can return any address it chooses.

That forged caller_ will:

- Pass the liquidity-provider or swapper allow-list checks even when the real user is not authorized.

- Be embedded in the Predicate message, misleading off-chain policy signers.

Thus the entire allow/deny mechanism is only as strong as the set of routers that are granted “trusted” status.

## Recommendation

Ensure that only audited, non-modifiable and well-known router implementations are ever whitelisted.

## Resolution

M0 Team: This is by design and we will ensure that only audited and open source routers are added to the allowlist.

# L-06 | Tick Range Risk Due To Paired Asset Volatility

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | Global | Acknowledged |

## Description

Even if the price of `wrappedM` remains constant and within the bounds the M0 team expects, there is no guarantee that the price of the paired asset (USDC) won’t deviate.

If USDC depegs or appreciates, it can push the pool outside of the configured tick range—even if the price of `wrappedM` remains within the intended bounds.

## Recommendation

Be aware of this scenario and be prepared to rebalance the pool if necessary.

## Resolution

M0 Team: Acknowledged. We will update the tick range if necessary.

# L-07 | Pectra Upgrade Enables EOAs

| Category | Severity | Location | Status |
|---|---|---|---|
| Validation | ● Low | AllowlistHook.sol | Acknowledged |

## Description

The AllowlistHook UniswapV4 hook is designed to enforce strict access control by restricting token swaps and liquidity provisioning to trusted addresses listed in _swappersAllowlist and _liquidityProvidersAllowlist, respectively.

This mechanism ensures that only authorized entities can interact with the pool to swap and add liquidity. However, the Ethereum Pectra upgrade, particularly through EIP-3074, introduces a significant concern in regards to both allowlists.

EIP-3074 enables Externally Owned Accounts (EOAs) to execute smart contract code within a single transaction, allowing them to act as proxies for other addresses. This capability compromises the hook's ability to restrict actions to trusted parties.

Specifically, a malicious EOA that is already present on either the _swappersAllowlist or the _liquidityProvidersAllowlist could delegate its execution to a smart contract that calls the Uniswap V4 PoolManager's respective pool to perform a swap or liquidity addition on behalf of an untrusted address, in exchange for a fee.

When the AllowlistHook checks the caller, it recognizes the trusted EOA and approves the transaction, allowing the untrusted address to bypass the allowlist restrictions.

## Recommendation

Consider monitoring the EOAs whitelisted in the hook and remove them from the allowlists if they show this behaviour.

## Resolution

M0 Team: Our team will monitor pools and ensure that addresses added to the allowlist will not perform swaps or liquidity additions for third parties.

# L-08 | Potential Front-run DoS

| Category | Severity | Location | Status |
|---|---|---|---|
| Validation | ● Low | AllowlistHook.sol | Acknowledged |

## Description

ServiceManager.validateSignatures() rejects a task if its identifier has already been consumed:

```
require(spentTaskIds[_task.taskId], "Predicate.validateSignatures: task ID already spent");
```

spentTaskIds is declared once and shared by every client that relies on the same ServiceManager instance:

```
mapping(string > bool) public spentTaskIds;   // global scope
```

Because the key is only the free-form taskId string, any operator that is authorized to call validateSignatures() can **f**ront-run another client's legitimate transaction, submit the same taskId first and set the flag to true. The second, honest call then reverts with *"task ID already spent"*, blocking the legit user's action.

The AllowlistHook uses:

```
authorizeTransaction → ServiceManager.validateSignatures()
```

for every swap or liquidity change when isPredicateCheckEnabled = true.

A malicious or compromised operator therefore has a trivial denial-of-service vector against all pools that use the hook.

## Recommendation

Isolate replay-protection per client rather than globally in the ServiceManager contract:

```
// Predicate-contracts: ServiceManager.sol // before mapping(string > bool) public spentTaskIds; //
aftermapping(address > mapping(string > bool)) public spentTaskIds;  // key-scoped// usage
require(spentTaskIds[_task.msgSender][_task.taskId], "Predicate.validateSignatures: task ID already
spent"); spentTaskIds[_task.msgSender][_task.taskId] = true;
```

By including the originating contract address (or alternatively the policyID) in the key, one client can no longer "burn" another client's taskId, removing the griefing vector without altering external behaviour for honest users.

## Resolution

M0 Team: Acknowledged.

# L-09 | TickRangeHook Does Not Support Multiple Pools

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | TickRangeHook.sol | Acknowledged |

## Description

The BaseTickRangeHook contract in Uniswap V4 currently defines a single pair of state variables, tickLowerBound and tickUpperBound, to enforce tick range restrictions for operations like liquidity provision or swaps. This design implies that the hook is intended to serve a single Uniswap V4 pool, as these tick bounds are global and not tied to any specific pool.

Therefore, the current expectation is that a new instance of BaseTickRangeHook should be deployed for each pool requiring a unique configuration. This approach works but is inefficient, as it increases deployment costs and scatters logic across multiple contract instances.

A more elegant solution exists: by leveraging the poolId, a unique identifier for each Uniswap V4 pool, the hook could manage pool-specific configurations within a single contract. This would allow multiple pools to use the same hook while maintaining independent tick bounds and other settings, such as allowlists.

## Recommendation

Consider updating the BaseTickRangeHook contract to make its state variables pool-specific by incorporating the poolId into the storage design. This would enable a single hook instance to support multiple pools, each with its own tailored tick bounds and configurations. Specifically: Replace the global tickLowerBound and tickUpperBound with mappings keyed by poolId:

```
mapping(bytes32 > int24) public tickLowerBounds; mapping(bytes32 > int24) public
tickUpperBounds;
```

Extend this pattern to other state variables, such as allowlists:

```
mapping(bytes32 > mapping(address > bool)) public allowLists;
```

Adjust the hook's internal logic to fetch pool-specific settings based on the poolId provided in hook calls (e.g., _beforeAddLiquidity or _afterSwap):

```
function _beforeAddLiquidity(bytes32 poolId, ...) internal view {int24 lower =
tickLowerBounds[poolId]; int24 upper = tickUpperBounds[poolId];  // Apply pool-specific tick
range checks}
```

This redesign allows a single BaseTickRangeHook contract to manage multiple pools, reducing deployment overhead and centralizing logic while ensuring each pool operates with its own independent settings.

## Resolution

M0 Team: Acknowledged.

# L-10 | WRAPPED_M Missing On Unichain

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | Config.sol: 43 | Acknowledged |

## Description

The WRAPPED_M address defined in Config.sol for Unichain does not point to a deployed contract. As a result, any attempt to create pools involving this token will fail.

## Recommendation

Deploy a valid WRAPPED_M ERC-20 contract at the specified address on Unichain, or update Config.sol to reference a valid address before proceeding with any pool creation.

## Resolution

M0 Team: We will deploy Wrapped M to Unichain at the same address than other networks before deploying hooks and pools on this network.

# L-11 | Donate() Not Guarded By Hook Logic

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | AllowlistHook.sol | Acknowledged |

## Description

The `AllowlistHook` enforces strict access control on liquidity provision, requiring addresses to be explicitly approved before they can add liquidity to the pool.

However, the pool's `donate` function bypasses these restrictions, allowing anyone—including unapproved or blacklisted addresses—to send tokens directly into the pool.

While this action does not increase the pool's liquidity, it does add to the fee balance available to existing liquidity providers.

## Recommendation

Consider whether this behavior is acceptable within the intended threat model. If not, use the `beforeDonate` hook to block all donation attempts.

## Resolution

M0 Team: Resolved.

# L-12 | Pools Can Be Initialized With An Out-Of-Range Tick

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | TickRangeHook.sol | Acknowledged |

## Description

When a pool is initialized in Uniswap V4 with a sqrtPriceX96 that translates to an initial tick outside the allowed range, defined by tickLowerBound and tickUpperBound, it creates significant problems for liquidity providers.

Liquidity providers can only provide liquidity within the predefined range of [tickLowerBound, tickUpperBound]. If the pool starts with a tick beyond these bounds (e.g., above tickUpperBound), their liquidity positions are immediately out of range.

As a result, LPs are forced to deposit only one token (e.g., token1) instead of a balanced mix of both tokens, since the price is already outside their specified range. This one-sided exposure increases their risk, leaving them vulnerable to price movements in one direction without the offsetting balance of holding both assets.

Let's imagine the scenario where the pool is initialized at tick 20, tickLowerBound is set to 0 and tickUpperBound to 10. Multiple liquidity providers provide liquidity right away, some in the 0,1 range, others in the 6,9 range etc.

These liquidity providers will obviously provide one sided liquidity as the current slot0.tick is outside of the range where they are allowed to provide by the TickRangeHook and where they actually provided liquidity. Consequently, they are very exposed to impermanent lost, in this concrete case, especially the ones that deposited in ranges closer to the tickLowerBound.

## Recommendation

Consider using the _afterInitialize hook in the BaseTickRangeHook contract to verify that the initial tick falls within the allowed range [tickLowerBound, tickUpperBound) after pool initialization.

If the tick is outside this range, the process should revert with a descriptive error, such as InitialTickOutOfRange. This constraint ensures the pool always starts in a state where LPs can provide balanced liquidity, earn fees and avoid the risks of one-sided exposure.

## Resolution

M0 Team: The initial tick at deployment is not enforced to allow market makers to create single sided liquidity positions once the pool is created and then allow them to swap and bring the liquidity in range.

# L-13 | Pool Price Is Initialized At The Lower Bound

| Category | Severity | Location | Status |
|---|---|---|---|
| Warning | ● Low | Deploy.s.sol: 118 | Acknowledged |

## Description

Currently, the Deploy.s.sol script deploys the Uniswap V4 pool as:

`IPoolManager(config_.poolManager).initialize(pool_, TickMath.getSqrtPriceAtTick(0));`

This sets the initial pool sqrtPriceX96 at the lower bound forcing initial liquidity providers to provide one sided liquidity, in this case, only WrappedM tokens.

## Recommendation

Unless this is intended, consider initializing the Uniswap V4 pool as:

`IPoolManager(config_.poolManager).initialize(pool_, 79230143144055126352967237632);`

which corresponds to tick (0.5):

```
python

import math

def get_sqrt_price_x96(tick: float) -> int:

Q96 = 2 ** 96

price = 1.0001 ** tick

sqrt_price = math.sqrt(price)

sqrt_price_x96 = int(sqrt_price * Q96)

return sqrt_price_x96

tick = 0.5
result = get_sqrt_price_x96(tick)

print(f"sqrtPriceX96 for tick {tick}: {result}") # 79230143144055126352967237632
```

## Resolution

M0 Team: Resolved

# Disclaimer

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Guardian to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. Guardian’s position is that each company and individual are responsible for their own due diligence and continuous security. Guardian’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by Guardian is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

Notice that smart contracts deployed on the blockchain are not resistant from internal/external exploit. Notice that active smart contract owner privileges constitute an elevated impact to any smart contract’s safety and security. Therefore, Guardian does not guarantee the explicit security of the audited smart contract, regardless of the verdict.

# About Guardian

Founded in 2022 by DeFi experts, Guardian is a leading audit firm in the DeFi smart contract space. With every audit report, Guardian upholds best-in-class security while achieving our mission to relentlessly secure DeFi.

To learn more, visit https://guardianaudits.com

To view our audit portfolio, visit https://github.com/guardianaudits

To book an audit, message https://t.me/guardianaudits