MEV-X Plugin

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

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

Project	MEV-X – Plugin - Audit Report
Website	mev-x.com
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/OIS-Solutions/MEV-X-Homelander/blob/3f36988a43beb670daa2c194b6378f82d9683e5b/contracts/ArbitragePlugin.sol
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High						
Medium						
Low						
Informational	7			7		
Governance	2			2		
Total	9			9		

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

The hook does not revert under normal operating conditions, provided that the governance contracts are configured correctly (see **Issue_01**). If no valid internal backrun exists or any internal validation fails, the hook returns early without affecting the user's transaction outcome. Furthermore, the Algebra administrator will always be able to remove the plugin from the pool.

The plugin does not pose any risk of pool funds being stolen.

MevXPlugin

Note: Only the MevXPlugin is within the audit scope, all other external contracts require to be audited in a separate audit

The MEVX plugin is a plugin which was developed in an effort to catch profit which is usually collected by MEV bots.

Whenever a (large) swap is executed, it will inherently shift the pool's price which can be in such a magnitude that MEV bots backrun this swap for a profit.

Consider the simple example of an WETH/USDC pool where WETH is currently valued at 4000 USDC. A user sells a large amount of WETH into the pool which results in a price decrease of WETH to 3500, while the external market price still reflects a WETH price of 4000. This presents an arbitrage opportunity where MEV bots usually backrun the swap and buy out WETH for a price of 3500 and then sell it into other markets for a profit.

This plugin makes use of the afterSwap hook which represents an atomic opportunity to execute a backrun right in the same transaction - even before MEV bots have the chance to execute their logic.

The plugin essentially only exposes logic within the afterInitialize and afterSwap functions. All other hooks simply return the expected selector but are not even invoked due to the setting of defaultPluginConfig = uint8(Plugins.AFTER_INIT_FLAG | Plugins.AFTER_SWAP_FLAG).

Appendix: Initialization

Whenever a pool is initialized, the afterInitialize hook on the MEVXPlugin contract is triggered which then executes an external call to the mevxRouter contract with the initializePool() selector.

We are currently unable to assess the executed logic as the mevxRouter contract is not present in this scope. However, we assume it is a simple notification to prepare the router that from now on, this pool might be part of a backrun strategy.

Appendix: After Swap Logic

Whenever a swap is executed, the afterSwap hook on the MEVXPlugin contract is triggered. The hook derives a poolId from the calling pool address and then performs an external call to the mevxRouter contract using the constructArbitrageRoute() selector, passing the pool identifier, the swap direction (zeroToOne), and the swap delta amounts (amount0, amount1).

If the router call succeeds and returns non-empty data, the returned payload is decoded into an “arbitrage possible” flag as well as routing parameters. In the case where arbitrage is deemed possible, the plugin then executes a follow up call to the mevxExecutor contract via executeRoute(), forwarding the encoded route data, the list of pools involved, the amountIn, and the profitDistributor as the profit receiver.

If the execution completes successfully, the plugin performs an additional external call to the profitDistributor contract via distributeProfit(), passing the current configId, the identified profitToken, and the original swap recipient as the (partial?) beneficiary. Both the executor call and the profit distribution call are wrapped in try/catch blocks and failures are silently ignored, meaning the hook attempts to perform backrun execution and profit distribution opportunistically without enforcing success.

Privileged Functions

- setPluginConfigToPool
- setConfigId
- setProfitDistributor
- setMevxExecutor
- setMevxRouter

Core Invariants:

INV 1: The plugin can be connected to any pool

INV 2: defaultPluginConfig must always be AFTER_INIT_FLAG and AFTER_SWAP_FLAG

INV 3: return data of constructArbitrageRoute must be: isArbPossible, profitToken, pools, amountIn, encodedRoute

INV 4: poolId must always represent pool address

INV 5: The used fee for a swap must always be globalState.lastFee

INV 6: During deployment, the deployer of the pool must expose logic within beforeCreatePoolHook which returns the address of the plugin

INV 7: The pluginFee must always be zero

INV 8: beforeInitialize must not execute any logic

INV 9: Only the plugin owner or Algebra administrator can set the pluginConfig to match the MevXPlugin and start the plugin logic

Issue_01	Revert due to non-matching decode data
Severity	**Governance**
Description	The AlgebraPool supports plugin hooks that execute during lifecycle events. When AFTER_SWAP_FLAG is enabled in globalState.pluginConfig, the pool calls plugin.afterSwap(...) after completing swap accounting. The pool validates hook compliance by requiring the plugin to return the correct selector through .shouldReturn(...). The MevxPlugin implements afterSwap by querying mevxRouter via a low level external call to construct a potential arbitrage route, and if an arbitrage is possible it attempts to execute it via mevxExecutor, then optionally distributes profit via profitDistributor. MevxPlugin.afterSwap performs a low level call to mevxRouter.constructArbitrageRoute(...) and, when the call returns success == true with non empty returnData, it decodes the returned bytes using abi.decode(returnData, (bool, address, address[], uint256, bytes)). Because the pool enforces the plugin hook's successful completion and correct selector return, the plugin revert propagates and causes the entire swap to revert. The root cause is that the plugin treats a successful low level call with non-empty return data as implicitly ABI compatible, while the pool treats afterSwap as a mandatory success path when the flag is enabled. Likewise, any expectation of return data can result in a revert.
Recommendations	Consider ensuring that governance is trusted.
Comments / Resolution	Acknowledged.

Issue_02	Gas-griefing attack on user via various different pathways
Severity	**Governance**
Description	The AlgebraPool integrates with an external plugin system controlled by globalState.pluginConfig. When AFTER_SWAP_FLAG is enabled, the pool executes the plugin hook afterSwap(...) at the end of swap execution and requires the hook to return the correct selector via .shouldReturn(...). The provided MevxPlugin implements afterSwap by performing an external call to mevxRouter to construct a potential arbitrage route, and, if arbitrage is deemed possible, it optionally calls mevxExecutor to execute the route and then calls profitDistributor to distribute profit to the swap recipient. These external interactions forward gas and are executed in the same transaction as the user swap. MevxPlugin.afterSwap introduces an unbounded post-swap execution segment whose gas usage depends on external components (mevxRouter, mevxExecutor, profitDistributor) and their return data and runtime behavior. The pool treats afterSwap as a mandatory success path whenever AFTER_SWAP_FLAG is enabled, meaning that any revert or out-of-gas condition in the plugin propagates and reverts the entire swap. Because the router call is performed via a low-level call that forwards gas subject to EIP-150, the plugin may be left with only a small residual gas budget to finalize decoding and return the selector. Any condition that causes the plugin to exceed this residual budget, including large return payloads, expensive router execution, expensive executor execution, or expensive profit distribution execution, can cause the plugin to run out of gas and revert. The root cause is the pool's strict enforcement of hook completion

	combined with the plugin's design of performing potentially heavy external work inside afterSwap without bounding gas usage, constraining return sizes, or isolating failures from swap finality. This can also purposely be used to grief users into paying more gas than expected and minting gas tokens on behalf of the user for a profit, for example the CHI token can be minted. Furthermore, another iteration of this issue is for example the fact that the poolsArray can be extremely large which then fully reverts the swap due to OOG. This is even possible when the user provides a large gas amount as the leftover 1/64 gas may be always insufficient depending on the memory expansion.
Recommendations	There is no simple solution for this issue because it inherently depends on the implementation of the three external contracts. These should be fully audited and governance should be trusted to ensure correctness. An advanced solution would be to use a customized gas limit for the external calls to ensure that there will be sufficient leftover gas. But even with that, the memory expansion attack could not be fully prevented. The simplest solution to address this issue is therefore trusted governance.
Comments / Resolution	Acknowledged.

Issue_03	Lack of revert propagation within afterInitialize can leave a pool permanently unregistered in the mevxRouter
Severity	**Informational**
Description	The AlgebraPool supports optional plugin hooks controlled by globalState.pluginConfig. When the AFTER_INIT_FLAG is enabled, AlgebraPool.initialize() calls plugin.afterInitialize(...) and enforces correct hook behavior via a selector check. The provided MevxPlugin implements afterInitialize by deriving a poolId from the pool address and issuing an external call to mevxRouter.initializePool(poolId, ALGEBRA_POOL_TYPE, data), where data encodes the initial price. This router interaction appears intended to notify or register the pool for subsequent MEV routing logic. MevxPlugin.afterInitialize wraps the router call in a try/catch block that swallows all failures. As a result, if mevxRouter.initializePool reverts for any reason, the plugin still returns the expected selector, and the pool initialization proceeds successfully with no onchain indication that router registration failed. The root cause is that the initialization hook does not propagate router failures and the pool lifecycle does not provide a replay mechanism for afterInitialize. Since poolId is derived from the pool caller address, direct external retries of afterInitialize from non pool callers cannot register the correct pool identifier, unless a governance functionality would be implemented. If router registration is a prerequisite for MEV routing, backrun execution, or profit distribution, the pool can become operationally misconfigured immediately after initialization. The pool remains live and tradable, but the MEV system may never recognize it, causing persistent loss of intended functionality and creating an inconsistent deployment state that is difficult to detect and remediate. The severity depends on whether mevxRouter offers a separate post deployment registration or recovery pathway. If no

	such pathway exists and initializePool can revert under realistic conditions, a pool may remain unregistered indefinitely, breaking the MEV integration for that pool without any revert signal at deployment time.
Recommendations	Depending on the implementation of the MevXRouter, this issue can be considered as fixed if such a governance function exists which allows for registering a pool within the router.
Comments / Resolution	Acknowledged. It is expected that there are multiple control-flows which allow for the registration of a pool.

Issue_04	ProfitDistributor may be prone to mis-accounting in case of unsuccessful call
Severity	**Informational**
Description	Currently, after the successful arbitrage execution, an external call to the profitDistributor is executed: *try* *profitDistributor.distributeProfit(* *configId,* *profitToken,* *recipient* *)* *{} catch {}* If that external call reverts while at the same time the ProfitDistributor does not account for undistributed profit, this profit may be locked or distributed to the next recipient.
Recommendations	Consider ensuring that the ProfitDistributor properly handles distribution of profit.
Comments / Resolution	Acknowledged.

Issue_05	Unprotected hooks will enable unexpected external calls
Severity	**Informational**
Description	The MevxPlugin hook functions are externally callable and do not enforce that the caller is an Algebra pool or a registered pool. In particular, afterSwap derives poolId from msg.sender and then performs an external call to mevxRouter.constructArbitrageRoute(poolId, zeroToOne, amount0, amount1). When called outside the pool context, msg.sender is an arbitrary external address, producing a poolId that does not correspond to a real pool, yet the plugin still performs the router call and may proceed to route execution and profit distribution based on router output. Similarly, afterInitialize can be called by any address and will attempt to initialize router state for a poolId derived from the arbitrary caller. The root cause is the absence of caller context validation in hook functions, combined with the use of msg.sender as an implicit authenticity signal for poolId derivation. If mevxRouter, mevxExecutor, or profitDistributor assume that hook invocations only occur as part of genuine pool lifecycle events, unprotected hook entry points allow third parties to trigger unexpected external calls and potentially influence the MEV routing subsystem. This can lead to state pollution in router registries, unbounded external call spam, and operational griefing through repeated expensive route construction attempts. In the worst case, if downstream components do not validate that poolId corresponds to an authorized registered pool and do not authenticate the call context, this can enable attacker-driven execution flows that were intended to be reachable only after real swaps or pool initialization. If downstream components strictly validate pool registration and caller authenticity, the impact may be limited to nuisance calls and monitoring confusion rather than direct loss of funds.
Recommendations	Consider ensuring robustness of external calls and corresponding

	control-flows.
Comments / Resolution	Acknowledged.

Issue_06	Pool can be initialized with wrong price if deployment flow is not robust
Severity	**Informational**
Description	AlgebraPool instances require an explicit initialization step via initialize(uint160 initialPrice) to set the initial square root price and tick. The pool stores the initialization state in globalState.price and rejects subsequent initialization attempts once a nonzero price is set. During initialization, the pool also triggers optional plugin hooks depending on configuration flags and sets default parameters such as fee, tick spacing, and community fee based on the factory's default configuration. AlgebraPool.initialize is externally callable and lacks access control, allowing any account to perform the initial call. The only gating condition is that globalState.price must be zero. The function accepts an arbitrary initialPrice and persists it directly into globalState.price and globalState.tick without validating it against any expected market ratio, oracle, or deployment supplied reference price. As a result, if a pool can exist in an uninitialized state, the first caller can initialize it with a wrong price and permanently lock in that starting state. The root cause is the combination of unrestricted initialization authority and the absence of a robust deployment invariant guaranteeing atomic initialization by a trusted deployer. In itself, this is not an issue for AlgebraPools but it can become an issue since the plugin forwards the price parameter towards the MevXRouter: *try* *mevxRouter.initializePool(poolId, ALGEBRA_POOL_TYPE, data)* *{} catch {}*

	Depending on the logic within mexRouter, this can become an issue (or not).
Recommendations	Consider elaborating whether this could expose any unexpected edge-cases.
Comments / Resolution	Acknowledged.

Issue_07	Pool with previous accumulated plugin fee may result in locked fee post-plugin switch
Severity	**Informational**
Description	setPlugin does not clear or reconcile pluginFeePending0 and pluginFeePending1. Pending plugin fees remain stored in the pool across plugin switches. However, the only code path that can transfer pending plugin fees is inside _changeReserves, and the fee-handling logic is only executed when at least one fee input to _changeReserves is nonzero. If the pool is switched to a plugin configuration that never produces any plugin fees and the community fee is also zero, subsequent swaps and position actions can reach _changeReserves with all fee inputs equal to zero. In that case, _changeReserves does not process pending fees, leaving pluginFeePending0 and pluginFeePending1 permanently untransferred. Additionally, when a transfer eventually does occur, the recipient is the current plugin address loaded at transfer time, meaning previously accrued pending fees are not bound to the plugin that originally generated them. The above description solely references the root in Algebra and obviously this seems to not be related to the MevXPlugin at all, because it does not expose such a plugin fee. However, a potentially occurring edge-case could be that a pool was previously attached to a plugin with a fee and is now attached to the MevXPlugin, while at the same time, having pending fees. These fees would then be distributed to the MevXPlugin and handlePluginFee would be successfully called. The fees however would remain lost.
Recommendations	Consider ensuring that there is never such a case where a pool was previously attached to a plugin with a fee. Alternatively, it could also be valuable to have a governance recovery function for tokens

	which could be transferred to this contract due to that circumstance. Reference: https://github.com/cryptoalgebra/Algebra/blob/integral-v1.2.2/src/plugin/contracts/base/BasePlugin.sol#L51C1-L54C4
Comments / Resolution	Acknowledged.

Issue_08	Deployment flow should be fully considered
Severity	**Informational**
Description	Currently, the deployment flow of a new pool related to the MevXPlugin is not part of the audit. Usually this would include a customPluginFactory contract and the full control-flow of the deployment, which would be something like: - PluginFactory.createCustomPool - AlgebraCustomPoolEntryPoint.createCustomPool - AlgebraFactory.createCustomPool - AlgebraCustomPoolEntryPoint.beforeCreatePoolHook - PluginFactory.beforeCreatePoolHook - PluginFactory.deployPool - AlgebraCustomPoolEntryPoint.afterCreatePoolHook - PluginFactory.afterCreatePoolHook - before/after init The current iteration works in itself by simply assuming that the pool will either be initialized with the MevXPlugin address or governance changes the plugin. There is nothing which speaks against this flow but it might not be ideal.
Recommendations	Consider thinking about whether it makes sense to develop a deployment framework.
Comments / Resolution	Acknowledged.

Issue_09	Pool will always use globalState.lastFee
Severity	**Informational**
Description	In the current integration, the fee override mechanism is not practically usable. The MevxPlugin implementation of beforeSwap always returns (0, 0) and the default configuration does not enable BEFORE_SWAP_FLAG. More importantly, the pool enforces that any nonzero overrideFee or pluginFee returned by beforeSwap requires DYNAMIC_FEE to be enabled, but MevxPlugin does not implement the dynamic fee interface the pool expects under DYNAMIC_FEE mode, causing incompatibility at the integration level. Additionally, even if a dynamic-fee-capable plugin were used, the pool treats overrideFee == 0 as a sentinel meaning "no override", so returning zero cannot set the swap fee to zero and instead leaves globalState.lastFee in effect. As implemented, the pool will always apply globalState.lastFee for swaps under MevxPlugin, and the system cannot implement "zero fee backruns" via beforeSwap in this configuration. This can invalidate economic assumptions for MEV backrun design, where the executor expects fee-free or near-fee-free execution. Even with future changes, the pool's sentinel semantics mean that a literal zero-fee override cannot be expressed through overrideFee without modifying the pool's fee logic.
Recommendations	We do not think this will need a change. There is a potential scenario of overriding the fee during the beforeSwap hook but this will come with its own complexity and would essentially require a full audit of this control-flow, including logic within SwapCalculation._calculateSwap. It would furthermore require DYNAMIC_FEE setup which then further interferes with other parts of the logic such as the fee() function reverting because the plugin does not expose a getCurrentFee() function and governance never being able to change the globalState.lastFee. It is not recommended to apply a change therefore. LPs should also always receive an appropriate fee in any transaction and should not be excluded from any fee during backrun transactions.

	NOTE: Please be informed that the implementation of such a feature will require additional audit efforts.
Comments / Resolution	Acknowledged.