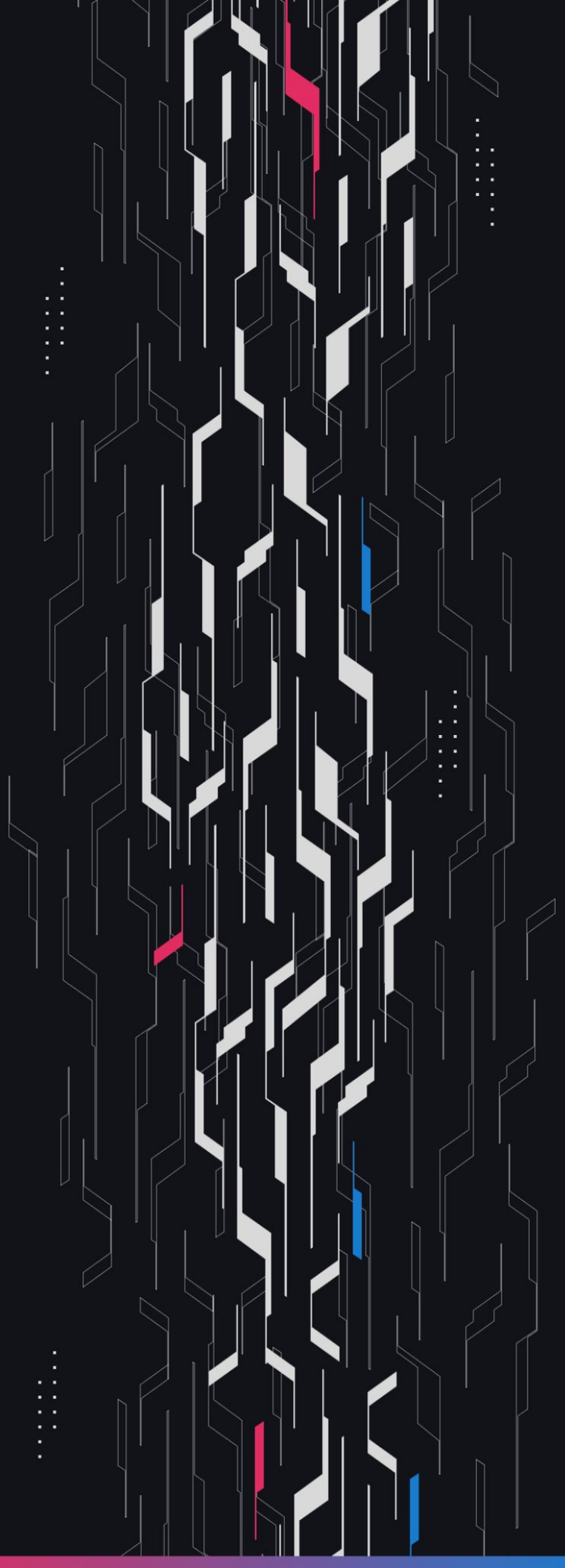
GA GUARDIAN

Universal Hook

Contracts Review

Security Assessment

December 21st, 2025



Summary

Audit Firm Guardian

Prepared By Ali Shehab, Zdravko Hristov, Michael Lett, Jordan, CrypticDefense

Client Firm Universal Hook

Final Report Date December 21, 2025

Audit Summary

Universal Hook engaged Guardian to review the security of their Universal Hook Contracts. From the 27th of November to the 5th of December, a team of 5 auditors reviewed the source code in scope. All findings have been recorded in the following report.

Confidence Ranking

Given the absence of High and Critical issues during the main review, Guardian assigns a Confidence Ranking of 5 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.

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

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. **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 1 Significant issue was uncovered per week. Design patterns are sound, and test coverage is strong. **Recommendation:** Suitable for deployment after remediations; consider periodic review with changes.	0-1 High/Critical findings per engagement week and little to no Medium severity issues. Varied Low 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. **Recommendation:** Address issues thoroughly and consider a targeted follow-up audit depending on code changes.	1-2 High/Critical findings per engagement week.
2: Low Confidence	Code shows frequent emergence of Critical/High vulnerabilities. Audit revealed recurring anti-patterns, weak test coverage, or unclear logic. These characteristics suggest a high likelihood of latent issues. **Recommendation:** Post-audit development and a second audit cycle are strongly advised.	2-4 High/Critical findings per engagement week. Or additional High/Critical findings uncovered in remediation review which have not been resolved and confirmed by Guardian.
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. **Recommendation:** Halt deployment and seek a comprehensive re-audit after substantial refactoring.	≥5 High/Critical findings and overall systemic flaws.

Table of Contents

Project Overview

Project Summary

Project Name	Universal Hook
Language	Solidity
Codebase	https://github.com/Alongside-Finance/universal-evm-contracts
Commit(s)	Main Review commit: 1d00e159324e4127dd84793b3b202a2749efbecf Remediation Review commit: 7b1d7f32febe2e34a46473a86eff793aaf6a3cd8 Remediation V2 Review commit: 58d991b403fa5276eaa36633d821909494aae182

Audit Summary

Delivery Date	December 21, 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	5	0	0	0	0	5
Low	14	0	0	4	0	10
Info	39	0	0	14	2	23

Audit Scope & Methodology

```
contract,source,total,comment
universal-evm-contracts-audit-11-27-25/src/v4-hook/UniversalJITOracle.sol,112,156,14
universal-evm-contracts-audit-11-27-25/src/v4-hook/UniversalJitHook.sol,316,435,45
universal-evm-contracts-audit-11-27-25/src/assets/MerchantController.sol,180,278,51
universal-evm-contracts-audit-11-27-25/src/assets/MintLimiter.sol,106,167,22
universal-evm-contracts-audit-11-27-25/src/assets/WrapFactoryV2.sol,19,27,1
universal-evm-contracts-audit-11-27-25/src/assets/WrappedAssetV2.sol,58,108,30
universal-evm-contracts-audit-11-27-25/src/v4-hook/lib/VirtualPrice.sol,65,95,8
universal-evm-contracts-audit-11-27-25/src/assets/lib/Nonces.sol,20,36,8

source count: {
  total: 1302,
  source: 876,
  comment: 179,
  empty: 247,
  single: 122,
  block: 43,
  mixed: 5,
  todo: 5,
  blockEmpty: 0,
  commentToSourceRatio: 0.204337899543379
}
```

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 Universal Hook, 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
INV-GLB-01	Users must not profit by doing atomic swaps	✅	✅	✅	10M+
INV-MNT-LMTR-01	Whitelisted assets must have non-zero assetPrices	✅	✅	✅	10M+
INV-MNT-LMTR-02	The user's balance cannot increase more than the mint limit after a mint operation	✅	✅	✅	10M+
INV-HK-01	After sweeping, the whole ERC6909 balance of the hook must be transferred to the merchant	✅	✅	✅	10M+
INV-HK-02	After a swap, upperVirtualSqrtPriceX96 must be in the range [upperTick; upperTick + upperRange]	✅	❌	✅	10M+
INV-HK-03	After a swap, lowerVirtualSqrtPriceX96 must be in the range [lowerTick - lowerRange; lowerTick]	✅	✅	✅	10M+
INV-HK-04	After oneForZero , the burned ERC6909 balance of the hook must be the minimum between the ERC6909 balance and the delta0 which the user received	✅	✅	✅	10M+

Invariants Assessed

ID	Description	Tested	Passed	Remediation	Run Count
INV-HK-05	After zeroForOne , the burned ERC6909 balance of the hook must be the minimum between the ERC6909 balance and the delta1 which the user received	✅	✅	✅	10M+
INV-HK-06	The amounts of zeroToOne swap must match the return value of SwapMath.computeSwapStep	✅	✅	✅	10M+
INV-HK-07	The amounts of oneForZero swap must match the return value of SwapMath.computeSwapStep	✅	✅	✅	10M+
INV-ORCL-01	After a price update, the report timestamp must strictly increase	✅	✅	✅	10M+
INV-ORCL-02	After a price update, the report ranges must be greater than 0	✅	✅	✅	10M+
INV-ORCL-03	After a price update, the report end ticks must be in the range [MIN_TICK; MAX_TICK]	✅	✅	✅	10M+
INV-ORCL-04	After a price update, the report tokens liquidity must be greater than 0	✅	✅	✅	10M+

Findings & Resolutions

ID	Title	Category	Severity	Status
M-01	Users Can Swap Non-whitelisted uAsset For USDC	Validation	Medium	Resolved
M-02	Blacklist Bypass Via Uniswap ERC6909	Unexpected Behavior	Medium	Resolved
M-03	Swap Can Use Stale Prices	Validation	Medium	Resolved
M-04	Pool Drain Depletes Mint Limits & Merchant Funds	Validation	Medium	Resolved
M-05	Paused Actors Can Still Burn uAssets	Validation	Medium	Resolved
L-01	Wrong EIP712 Struct Hashes	Signatures	Low	Acknowledged
L-02	Missing Delta Tick Validation	Validation	Low	Resolved
L-03	sweepToMerchant Does Not Verify PoolKey	Validation	Low	Resolved
L-04	sweepToMerchant Enable Mint Limit Exhaustion	DoS	Low	Resolved
L-05	Missing sqrtPriceLimitX96 Check	Validation	Low	Resolved
L-06	Missing Tick Crossover Validation	Validation	Low	Acknowledged
L-07	Mint Cap Griefing Blocks USDC->uAsset Swaps	DoS	Low	Resolved
L-08	Oracle Updates Limited By MAX_DELTA_TICK	Logical Error	Low	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
L-09	Burn Event Is Not Emitted For Burns	Events	Low	Resolved
L-10	Merchant Can Bypass Burn Request Verification	Unexpected Behavior	Low	Acknowledged
I-01	Unused Constructor Param	Best Practices	Info	Resolved
I-02	_getGlobalBalance Is Internal And Not Used	Best Practices	Info	Resolved
I-03	Unused State Variable liquidityLeft	Informational	Info	Resolved
I-04	Unresolved TODOs Within VirtualPrice.sol	Best Practices	Info	Acknowledged
I-05	Swap Events Emit Zero Amounts	Events	Info	Resolved
I-06	Unused Imports/Errors	Best Practices	Info	Resolved
I-07	Missing MAX_DELTA_TICK Validation	Validation	Info	Partially Resolved
I-08	No Validation Of Pool Fee Tier	Validation	Info	Acknowledged
I-09	No Desired Price Is Signed In Merchant Requests	Best Practices	Info	Acknowledged
I-10	Unnecessary Approval Given To The PoolManager	Best Practices	Info	Resolved
I-11	Notes About Nonce Behavior	Warning	Info	Acknowledged

Findings & Resolutions

ID	Title	Category	Severity	Status
I-12	Unnecessary Storage Read Wastes Gas	Gas Optimization	Info	Resolved
I-13	Redundant onlyPoolManager Modifier Usage	Gas Optimization	Info	Resolved
I-14	Unused And Missing Event Emissions	Events	Info	Acknowledged
I-15	Pool Creation May Be Front-ran	Informational	Info	Acknowledged
I-16	Unnecessary Type Casting	Best Practices	Info	Resolved
I-17	Inconsistent Naming Conventions	Best Practices	Info	Resolved
I-18	Inconsistent Merchant Controller Upgradability	Upgradeability	Info	Acknowledged
I-19	Merchant May Be Forced To Trade	Warning	Info	Acknowledged
I-20	Misleading NatSpec	Documentation	Info	Resolved
I-21	WrappedAssetV2 Salt May Not Be Unique	Best Practices	Info	Resolved
I-22	Misleading NatSpec	Documentation	Info	Resolved
I-23	Admin Can't Burn For Blacklisted Address	Unexpected Behavior	Info	Acknowledged
I-24	Burn Operations Ignore Asset Pause Mechanism	Validation	Info	Resolved

Findings & Resolutions

ID	Title	Category	Severity	Status
I-25	Unrecoverable ERC6909 Balances Risk	Warning	Info	Resolved
I-26	Mint Limit Bypass Via Tiny Swaps	Rounding	Info	Resolved
I-27	MintLimitReached Doesn't Include Current Value	Best Practices	Info	Acknowledged
I-28	Unused Role Definitions	Informational	Info	Resolved
I-29	Warning About Temporary Unbacked uAssets	Warning	Info	Acknowledged

M-01 | Users Can Swap Non-whitelisted uAsset For USDC

Category	Severity	Location	Status
Validation	Medium	UniversalJitHook.sol: 134	Resolved

Description

MintLimiter allows the Pauser Role to pause uAssets via pauseAsset(), which removes an asset from the set of whitelisted uAssets.

The protocol documentation states the following regarding the flow of selling uAssets: "The order input token is a valid whitelisted uAsset and the output token is USDC (token received by the user)".

Once the token is paused, it can no longer be minted, thus USDC -> uTokens swaps will begin to fail. In addition, pools with the paused uToken cannot be initialized, as the beforeInitialize hook enforces that the uToken must be whitelisted.

However, there is nothing preventing swaps from a paused uToken -> USDC within the hook contract. In a situation where a uAsset is removed from the whitelist, users holding this asset can still swap them into USDC, thus taking merchant USDC and breaking the intended flow of selling uAssets.

Recommendation

Check if the input uAsset is whitelisted within the _beforeSwap hook. Also consider checking if the output uAsset is whitelisted, in case the hook owns any ERC6909 of non-whitelisted uAsset within PoolManager, USDC → uAsset swaps will still be successful (which will allow users to continue purchasing non-whitelisted uAsset).

Resolution

Universal Hook Team: Resolved.

M-02 | Blacklist Bypass Via Uniswap ERC6909

Category	Severity	Location	Status
Unexpected Behavior	Medium	WrappedAssetV2.sol: 83-101	Resolved

Description

WrappedAssetV2 contains a blacklist mechanism which blocks certain users from transferring and receiving uAsset.

However, there is a way blacklisted users can circumvent this. Consider a scenario where Bob is blacklisted from transferring/receiving uBTC.

1. Bob swaps USDC -> uBTC. Now the Hook contract will transfer uBTC to Uniswap's PoolManager contract.

2. Instead of calling PoolManager.take() (which will fail since Bob is blacklisted), Bob calls PoolManager.mint(), which wraps the uBTC to ERC6909 owned by Bob.

3. Bob can now freely transfer the uBTC through Uniswap's ERC6909 transfer functions.

4. If Bob swaps uBTC back to USDC at a later time, he simply has to burn the ERC6909, where the UniversalJitHook then receives the assets via PoolManager.mint().

In the above scenario, Bob has successfully circumvented the blacklist by receiving uBTC, freely transferring, and selling back for USDC freely at a later time.

Recommendation

Uniswap encodes msg.sender when calling the beforeSwap hook. Consider reverting if the sender or tx.origin is the address of a blacklisted address.

Note that this does not prevent blacklisted addresses from receiving and then transferring ERC6909 version of uAssets, but does prevent them from buying/selling these assets directly via the UniversalJitHook.

Resolution

Universal Hook Team: Resolved.

M-03 | Swap Can Use Stale Prices

Category	Severity	Location	Status
Validation	Medium	UniversalJitHook.sol: 144-147	Resolved

Description PoC

When a swap happens, the before swap hook sets the virtual price based on the oracle report timestamp, to catch latest updates:

```
// If the oracle report is newer or happened in the same block as the last update, update the virtual price
if (swapParams.zeroForOne && report.timestamp > lowerVirtualPrice[poolId].timestamp) {
lowerVirtualPrice[poolId].update(TickMath.getSqrtPriceAtTick(report.jitLowerTick));
} else if (!swapParams.zeroForOne && report.timestamp > upperVirtualPrice[poolId].timestamp) {
upperVirtualPrice[poolId].update(TickMath.getSqrtPriceAtTick(report.jitUpperTick));
}
```

The `update` function incorrectly sets the price timestamp to the current block timestamp, rather than the oracle report timestamp:

```
function update(VirtualPrice storage virtualPrice, uint160 virtualSqrtPriceX96) internal {
virtualPrice.timestamp = uint48(block.timestamp);
virtualPrice.virtualSqrtPriceX96 = virtualSqrtPriceX96;
}
```

This allows for a scenario where a swap happens, then an update price occurs, then another swap occurs all in the same block. In this case, the second swap would not see the updated virtual price because the timestamps would be equal, and the condition would fail.

The first swap in the previous example should either be the first swap ever or the first after a price update, else it wouldn't need to synchronize the virtual price.

This would result in the second swap using stale virtual price data.

Recommendation

Consider changing the `update` to set the price's timestamp as the report's.

Resolution

Universal Hook Team: Resolved.

M-04 | Pool Drain Depletes Mint Limits & Merchant Funds

Category	Severity	Location	Status
Validation	Medium	UniversalJitHook.sol: 384-385	Resolved

Description PoC

When swapping uAssets → USDC, the hook first tries to fulfill the swap by burning any ERC6909 claims it owns. If the claims are insufficient, the hook transfers the remaining USDC from the merchant.

The amount burned is computed as the minimum of:
• the requested USDC amount
• the hook's ERC6909 claims
• the PoolManager's USDC balance

This can be abused by wrapping a swap inside a custom take → swap → settle sequence. By interacting directly with the PoolManager (instead of the router), an attacker can temporarily drain all USDC held by the PoolManager, perform their swap, and then return and settle the USDC.

Because the PoolManager's USDC balance becomes 0 during the attacker's swap, the hook is forced to fulfill the entire swap using the merchant's USDC balance, even if the hook actually owns enough claims to cover the swap.

Similarly, for USDC -> uAsset swaps, the _settleWithClaims function checks if the hook address has any wrapped ERC6909 uAssets to settle before minting uAssets to the user. With a similar take → swap → settle sequence, an attacker can temporarily drain all uAssets held by the PoolManager, forcing the hook to mint new uAssets to the user instead of settling with existing wrapped uAssets, depleting the hook's mint limit.

This DoSes all future swaps until the next epoch, where the mint limit resets. uAsset -> USDC example:

If there are 3 pools, each with 10k USDC liquidity, and the merchant holds exactly 30k USDC, an attacker can force the hook to pull more than 10k USDC from the merchant for a single pool. This may prevent honest users from swapping USDC → uAsset in that pool until the merchant sweeps the accumulated claims back from the hook.

Recommendation

Consider removing the PoolManager balance check when deciding how much to burn from claims.

Settling with claims (burn = true) does not transfer any assets from the PoolManager, and the burned amount is already fully accounted for in the PoolManager's internal deltas. Because of this, the hook does not need to rely on the PoolManager's temporary balance during the swap.

Resolution

Universal Hook Team: Resolved.

M-05 | Paused Actors Can Still Burn uAssets

Category	Severity	Location	Status
Validation	Medium	MerchantController.sol: 228	Resolved

Description

The docs mention that paused actors should not be able to burn `uAssets`:

Account pausing to prevent a witness or merchant from being able to mint or burn `uAssets` in the event of a security incident.

However, when pausing an actor only the mint limit is set to 0, which is only validated when minting `uAssets`. When burning `uAssets` the mint limit is not validated, so even if the actor (merchant/hook) is paused they can still burn `uAssets`.

For example, if a security incident happens, and the project decided to pause the hook to block any future mints or burns, the hook could still be able to burn assets in exchange for USDC.

This would also allow merchant to burn assets as well, but would require witness signatures.

Recommendation

Consider validating if the actor is paused when performing a burn operation.

Resolution

Universal Hook Team: Resolved.

L-01 | Wrong EIP712 Struct Hashes

Category	Severity	Location	Status
Signatures	Low	MerchantController.sol: 59	Acknowledged

Description

The contract's EIP-712 struct hashes are not constructed according to the canonical EIP-712 spec. Specifically, the type hashes mix unrelated type strings and omit dependent type definitions:

```
bytes32 internal constant OPERATION_TYPE_HASH = keccak256(abi.encodePacked(OPERATION_TYPE,
REQUEST_TYPE));
bytes32 internal constant REQUEST_TYPE_HASH = keccak256(abi.encodePacked(REQUEST_TYPE));
```

Under EIP-712 rules:
• Each struct must have its own independent type hash:

```
keccak256("Operation(uint8 opType,address asset,uint256 amount)")
```

• A primary struct (e.g., Request) must include its dependent struct types as part of its canonical type string.

In this implementation:
• OPERATION_TYPE_HASH incorrectly includes the Request type definition.
• REQUEST_TYPE_HASH omits the Operation subtype entirely.

As a result, the final EIP-712 digest differs from what standard libraries will produce.

Recommendation

Consider constructing type hashes according to the EIP-712 canonical rules, for example:

```
bytes32 constant OPERATION_TYPE_HASH = keccak256("Operation(uint8 opType,address
asset,uint256 amount)");
bytes32 constant REQUEST_TYPE_HASH =
keccak256(
"Request(Operation[] operations,address account,uint256 expiration,uint256 nonce)"
"Operation(uint8 opType,address asset,uint256 amount)"
);
```

Resolution

Universal Hook Team: Acknowledged.

L-02 | Missing Delta Tick Validation

Category	Severity	Location	Status
Validation	Low	[src/v4-hook/UniversalJITOracle.sol:_setPrice], UniversalJITOracle.sol: 125-131	Resolved

Description

In `_setPrice` function of `UniversalJITOracle.sol`, when updating the oracle price after the initial setup, the contract validates that the change in `jitLowerTick` does not exceed `MAX_DELTA_TICK` compared to the previous value.

This check prevents excessive price movements for `zeroForOne` swaps. However, the corresponding validation for `jitUpperTick` is absent, creating an asymmetric security control when setting the price of `oneForZero` swaps.

Recommendation

Add a matching delta tick validation for `jitUpperTick` to ensure symmetric protection for both swap directions.

Resolution

Universal Hook Team:Resolved.

L-03 | sweepToMerchant Does Not Verify PoolKey

Category	Severity	Location	Status
Validation	Low	UniversalJitHook.sol: 408	Resolved

Description

`UniversalJitHook.sweepToMerchant()` accepts an arbitrary `PoolKey`, but does not verify that the pool corresponds to a valid (`USDC`, `uAsset`) pair.

Because of this, a caller can supply a malicious or non-whitelisted `PoolKey`, for example a pair of arbitrary tokens or two `uAssets`. The sweep logic will incorrectly interpret which token should be burned and which should be sent to the merchant:

```
pairIs0 = _isPairCurrency(key.currency0);
pairCurrency = pairIs0 ? key.currency0 : key.currency1;
wrappedCurrency = pairIs0 ? key.currency1 : key.currency0;
```

This means that tokens intended to be burned can be transferred to the `Merchant` address instead.

This breaks the invariant that `uAssets` accumulated in the `PoolManager` are meant to be burned during sweep, and sending them to the merchant instead causes temporary excess circulating `uAsset` supply until an admin manually burns the surplus.

Recommendation

Validate the key is a whitelisted (`USDC`, `uAsset`) pair.

Resolution

Universal Hook Team: Resolved.

L-04 | sweepToMerchant Enable Mint Limit Exhaustion

Category	Severity	Location	Status
DoS	Low	UniversalJitHook.sol: 408-436	Resolved

Description

The protocol implements a mint limit on `MerchantController.mintFromHook` that resets every 8-hour epoch to prevent unbounded token minting in case of vulnerability exploitation. The `UniversalJitHook` is designed to avoid reaching this limit by reusing existing `ERC6909` claims in the `PoolManager` rather than minting new `uAsset` tokens when claims are available.

However, the `sweepToMerchant` function is publicly callable by anyone and performs the following operations:
1. Burns all wrapped `uAsset` claims held by the Hook in the `PoolManager`.
2. Transfers other assets to the merchant.

By burning the Hook's `ERC6909` claims, an attacker forces the Hook to call `mintFromHook` on subsequent swaps that buy `uAsset`. This allows an attacker to artificially inflate the Hook's mint counter without performing trades.

The attack flow:
1. Attacker observes `uAsset` claims accumulating in the Hook (from swaps).
2. Attacker calls `sweepToMerchant` to burn these claims.
3. When the next user buys `uAsset`, the Hook must call `mintFromHook` instead of settling claims.
4. By repeatedly calling `sweepToMerchant` and triggering `uAsset` mints, an attacker can exhaust the epoch's mint limit.
5. Once the limit is reached, legitimate `uAsset` purchases are blocked, creating a temporary denial of service.

Recommendation

• Restrict `sweepToMerchant` Access

Add access control to ensure only authorized addresses (e.g. merchant) can call `sweepToMerchant`.

Resolution

Universal Hook Team: Resolved.

L-05 | Missing sqrtPriceLimitX96 Check

Category	Severity	Location	Status
Validation	Low	UniversalJitHook.sol: 289-342	Resolved

Description

The UniversalJITHook.sol overrides the swap delta calculation and settles balances based on oracle prices, but completely ignores the sqrtPriceLimitX96 parameter provided by users in SwapParams.

This parameter is the standard mechanism for slippage protection in Uniswap V4, it allows users to specify a maximum or minimum acceptable execution price.

By ignoring this limit, the hook removes the user's ability to enforce price bounds on their swap.

The consequences are:

1. Users can receive less amountOut.

2. Users can pay more amountIn than expected

Recommendation

Add validation in the swap functions to enforce the user's price limit. After computing the swap amounts, verify that the execution price respects the user's limit.

Resolution

Universal Hook Team: Resolved.

L-06 | Missing Tick Crossover Validation

Category	Severity	Location	Status
Validation	Low	UniversalJITOracle.sol: 144-150	Acknowledged

Description

The `_checkTicksAndRange` function in `UniversalJITOracle.sol` validates that the new price ticks are internally consistent (`update.jitLowerTick <= update.jitUpperTick`), but fails to validate that the new ticks don't cross the previous ticks. This creates a guaranteed arbitrage opportunity that can be sandwiched for risk-free profit.

Consider the following scenario:
• Current oracle state: `jitLowerTick = 100`, `jitUpperTick = 150`
• New oracle update: `jitLowerTick = 50`, `jitUpperTick = 80`

The new ticks pass the internal consistency check (50 <= 80), but a crossover situation exists where the entire new price range falls below the old price range. An attacker observing this price update can:

1. For a downward crossover (new ticks fall below old ticks): Execute a `zeroForOne` swap at the old price before the update, then execute a `oneForZero` swap at the new lower price after the update for guaranteed profit.
2. For an upward crossover (new ticks rise above old ticks): Execute a `oneForZero` swap at the old price before the update, then execute a `zeroForOne` swap at the new higher price after the update for guaranteed profit.

The issue occurs because the `_beforeSwap` hook in `UniversalJITHook` uses the new oracle price immediately for swap pricing, regardless of whether it represents a logical continuation of the previous price. A crossover represents a discontinuous price jump that should be rejected.

Recommendation

Add crossover validation to `_checkTicksAndRange` that prevents the new ticks from crossing or inverting relative to the previous ticks.

This ensures price updates maintain logical continuity and eliminates the guaranteed arbitrage window that crossover situations create.

Resolution

Universal Hook Team: Acknowledged.

L-07 | Mint Cap Griefing Blocks USDC->uAsset Swaps

Category	Severity	Location	Status
DoS	Low	UniversalJitHook.sol: 369	Resolved

Description

When users swap USDC → uAsset, the JIT hook first tries to settle the required uAssets using existing PoolManager claims. If the available claims are insufficient, the hook mints additional uAssets through the MerchantController.

Minting is rate-limited per minter per epoch (8 hours).

If the limit is reached, further USDC → uAsset swaps cannot mint and will revert.

Separately, the hook exposes sweepToMerchant, which allows anyone to burn the accumulated PoolManager claims and then burn the corresponding uAssets.

Because claims are burned but the epoch mint cap is not reset, an attacker can artificially deplete the cap for the entire epoch.
1. Attacker takes a flashloan of USDC.
2. Swaps USDC → uAsset, forcing the hook to mint uAssets.
3. Swaps uAsset → USDC back.
4. Repays the flashloan (only pays slippage/spread).
5. Calls sweepToMerchant, burning all uAsset claims created during the swaps.
6. Result: the mint cap for the epoch is fully consumed, but the system holds zero claims.

Now, any user attempting USDC → uAsset swaps will fail because:
• the hook cannot settle from claims (they were burned), and
• minting is blocked due to the exhausted epoch limit.

This creates a griefing DoS that lasts until the epoch resets, preventing normal user swaps.

Note: Even if sweepToMerchant were permission-restricted, the same DoS occurs as long as a sweep is executed in the same epoch after the attacker performs the cycle.

Recommendation

Consider keeping a configurable buffer of uAsset claims in the PoolManager that is not burned when calling sweepToMerchant, this could either be a percentage of the mint limit or an absolute value, as long as it is enough to fulfill "normal" swaps.

Resolution

Universal Hook Team: Resolved.

L-08 | Oracle Updates Limited By MAX_DELTA_TICK

Category	Severity	Location	Status
Logical Error	Low	UniversalJITOracle.sol: 120-131	Resolved

Description

The Universal protocol contains 80+ assets including uBTC, uETH, uSOL, uXRP, uDOGE, and more. If any of these tokens experiences a price drop, the MAX_DELTA_TICK check prevents the oracle from keeping pace with market reality. Consider the following scenario:

Assume 1 uAsset = 20,000 USDC and the UniversalJitHook currently reflects this price.

The uAsset now experiences a price drop, where 1 uAsset now equals 10,000 USDC. The UniversalJitOracle will not have enough time to update to the market price, because of the following check.

The new price can only update as far as the immutable MAX_DELTA_TICK is configured to, which in tests is configured to 120, reflecting a tight delta tick bound on price updates for all (uAsset, USDC) pools.

In addition, oracle updates must happen at least 5 seconds from the previous update, which means one update every 5 blocks.

This would mean, moving from a tick of -177,300 (token0 = uAsset, token1 = USDC, where 1 uAsset = 20,000 USDC) to a tick of -184,200 (1 uAsset = 10,000 USDC) would require about 58 calls to setPrice(), as the MAX_TICK_DELTA only allows a price shift of 120 ticks per update. In addition, these calls must be separate by at least 5 blocks due to the MIN_FREQUENCY check.

This allows an attacker to buy the asset tokens at 10,000 USDC each, convert to uAsset (or buy uAsset directly from, for example separate dex's) and continue to sell by swapping through the UniversalJitHook for 1 uAsset = 20,000 USDC, effectively draining Merchant liquidity and stealing funds.

Due to the MAX_DELTA_TICK and MIN_FREQUENCY limits, the oracle will not have enough time to reflect the true market price and will continue to slowly adjust to the current market price, allowing the attacker to continue stealing funds.

Recommendation

Avoid using an immutable MAX_DELTA_TICK. Instead, make this parameter configurable so the oracle can adjust prices in a single update for highly volatile assets.

Resolution

Universal Hook Team: Resolved.

L-09 | Burn Event Is Not Emitted For Burns

Category	Severity	Location	Status
Events	Low	MerchantController.sol: 168-170	Resolved

Description

For normal burn requests, the `TokensBurnt()` event is emitted, which would allow the offchain service to track the burns and sell the underlying assets on Coinbase.

However, there is no event emitted in the `burnFromHook()` function.

Recommendation

Consider whether the event needs to be emitted in `burnFromHook()` as well for offchain usage.

Resolution

Universal Hook Team: Resolved.

L-10 | Merchant Can Bypass Burn Request Verification

Category	Severity	Location	Status
Unexpected Behavior	Low	UniversalJitHook.sol: 420-426	Acknowledged

Description

`WrappedAssetV2::burn` enforces access control to ensure `uAsset` can only be burned from the `MerchantController`.

This is to ensure that Merchants can only execute burn requests that are signed and verified by trusted witnesses, in case a Merchant address is compromised, or to prevent malicious redemption transactions. This is specifically for the redemption flow, where users can redeem `uAsset` for underlying.

However, a malicious Merchant can bypass the burn request process by transferring `uAsset` directly to `PoolManager` and wrapping in `ERC6909`, then transferring the `ERC6909` `uAsset` to the hook address. The hook address will proceed to burn the `uAsset` during the next `sweepToMerchant`.

This allows a malicious merchant to destroy `uAssets` held by legitimate users (since users sent them to the Merchant to unlock underlying during redemption flow), potentially causing user losses (until admin directly re-mints them to users) and undermining trust in the protocol.

In addition, the access control can be bypassed by regular users who can achieve direct burns via the same method.

Recommendation

Consider tracking the burnable amount via internal accounting rather than relying on the entire `balanceOf` `ERC6909` `uAssets` within `PoolManager` during `sweepToMerchant()`.

Resolution

Universal Hook Team: Acknowledged.

I-01 | Unused Constructor Param

Category	Severity	Location	Status
Best Practices	Info	/src/v4-hook/UniversalJITOracle.sol.sol	Resolved

Description

The variable `tickSpacing` in `UniversalJITOracle.sol` is passed to the constructor but never used.

Recommendation

Remove the variable from constructor as it is never used.

Resolution

Universal Hook Team: Resolved.

I-02 | _getGlobalBalance Is Internal And Not Used

Category	Severity	Location	Status
Best Practices	Info	UniversalJITOracle.sol	Resolved

Description

The function `_getGlobalBalance` is internal view function, but is never called in any function.

Recommendation

Remove the `_getGlobalBalance` function.

Resolution

Universal Hook Team: Resolved.

I-03 | Unused State Variable liquidityLeft

Category	Severity	Location	Status
Informational	Info	UniversalJitHook.sol: 50	Resolved

Description

The `UniversalJitHook.sol` contract contains an unused state variable mapping `liquidityLeft`.

This mapping is never written to or read from anywhere in the contract codebase.

Recommendation

Remove the mapping `liquidityLeft` from the contract or implement the necessary functionality.

Resolution

Universal Hook Team: Resolved.

I-04 | Unresolved TODOs Within VirtualPrice.sol

Category	Severity	Location	Status
Best Practices	● Info	VirtualPrice.sol: 49	Acknowledged

Description

The `VirtualPrice` contract has multiple unresolved TODOs:

1. A TODO noting that `LiquidityAmounts` is imported from Uniswap V4's test utils, suggesting the usage still needs to be validated for production use.

2. Several require invariant checks marked as "TODO: remove after testing", which look like leftover test assertions.

While these do not currently present a security issue, they indicate unfinished review/cleanup work and could cause unnecessary reverts or confusion in future maintenance.

Recommendation

Explicitly validate/document the use of `LiquidityAmounts` as production-safe, and keep invariants as assertions or remove them if no longer needed.

Resolution

Universal Hook Team: Acknowledged.

I-05 | Swap Events Emit Zero Amounts

Category	Severity	Location	Status
Events	Info	PoolManager.sol: 239-248	Resolved

Description

The protocol's custom Uniswap v4 hook overwrites the `SwapReturnDelta`, which forces all swap executions to operate with `amountSpecified = 0` inside `PoolManager.swap()`.

Because the hook adjusts the swap amount, `Hook.sol`'s `beforeSwap()` updates `amountToSwap`, but the logic in `Pool.sol` treats any `params.amountSpecified == 0` as a no-op swap.

As a consequence, `Pool.swap()` immediately returns `BalanceDeltaLibrary.ZERO_DELTA`, and the emitted `Swap` event always has `amount0 = 0` and `amount1 = 0`.

This results in all emitted swap events being incorrect and unusable.

Recommendation

Consider emitting an event inside the Hook.

Resolution

Universal Hook Team: Resolved.

I-06 | Unused Imports/Errors

Category	Severity	Location	Status
Best Practices	Info	UniversalJitHook.sol/UniversalJITOracle.sol/VirtualPrice.sol/MerchantController.sol	Resolved

Description

Errors:
1. AccountPaused in src/assets/MerchantController.sol
2. ThresholdTooHigh in src/assets/MerchantController.sol
3. ZeroThreshold in src/assets/MerchantController.sol
4. InvalidTickSpacing in src/v4-hook/UniversalJITOracle.sol
5. NotHook in src/v4-hook/UniversalJITOracle.sol
6. PriceOutOfBounds in src/v4-hook/UniversalJITOracle.sol
7. SpreadTooLarge in src/v4-hook/UniversalJITOracle.sol
8. SpreadTooSmall in src/v4-hook/UniversalJITOracle.sol

Unused Imports:
1. StateLibrary in src/v4-hook/UniversalJitHook.sol
2. TickMath in src/v4-hook/lib/VirtualPrice.sol
3. FullMath in src/v4-hook/lib/VirtualPrice.sol
4. SafeCast in src/v4-hook/lib/VirtualPrice.sol
5. LiquidityAmounts in src/v4-hook/lib/VirtualPrice.sol

Recommendation

Remove unused errors/imports

Resolution

Universal Hook Team: Resolved.

I-07 | Missing MAX_DELTA_TICK Validation

Category	Severity	Location	Status
Validation	Info	UniversalJITOracle.sol: 68	Partially Resolved

Description

The `UniversalJitOracle` utilizes an immutable variable `MAX_DELTA_TICK` to bound how much the newly reported tick is allowed to change between oracle updates.

The contract also has defined the following, which is unused: `error InvalidMaxDeltaTick();`, suggesting the intention to properly validate the max delta tick value before it is permanently set, as a misconfigured or negative `MAX_DELTA_TICK` can permanently brick price updates.

Recommendation

Properly validate the `MAX_DELTA_TICK` in the constructor, for example adding lower/upper bounds.

Resolution

Universal Hook Team: Partially Resolved.

I-08 | No Validation Of Pool Fee Tier

Category	Severity	Location	Status
Validation	Info	UniversalJITOracle.sol: 144	Acknowledged

Description

Inside `_checkTicksAndRange`, the oracle validates tick ordering and JIT ranges, but it does not validate anything about the pool's fee tier, even though all swaps executed through the hook are intended to behave as zero-fee swaps with no LPs.

This creates a configuration mismatch scenario where a pool may be initialized with a non-zero fee tier, but all swaps executed through the hook behave as fee-free swaps.

Recommendation

Consider adding a check to `_checkTicksAndRange` to ensure that the pool's fee tier is zero, this requires passing the pool key instead of the pool ID.

Resolution

Universal Hook Team: Acknowledged.

I-09 | No Desired Price Is Signed In Merchant Requests

Category	Severity	Location	Status
Best Practices	Info	MerchantController.sol	Acknowledged

Description

When a mint merchant request is processed, the minted amount is multiplied by the price of the asset and the resulting USD value is consumed from the limit of the merchant and all of the witnesses.

Because there is no data signed about the price of the token, the merchant and the witnesses may end up executing an undesirable request if `setMintLimitPrice` or `setMintLimitPrices` is called in between.

Recommendation

Consider whether a price information should be included in the signed data.

Resolution

Universal Hook Team: Acknowledged.

I-10 | Unnecessary Approval Given To The PoolManager

Category	Severity	Location	Status
Best Practices	Info	UniversalJitHook.sol: 370	Resolved

Description

In the `_settleUAsset()` function, the `uAsset` is first minted to the hook and then an allowance is given to the poolManager before executing `settle()`.

```
function _settleUAsset(Currency asset, uint256 amount) internal {
uint256 settled = _settleWithClaims(asset, amount);
uint256 remain = amount - settled;
if (remain > 0) {
address assetAddress = Currency.unwrap(asset);
merchantController.mintFromHook(assetAddress, address(this), remain);
IERC20(assetAddress).approve(address(poolManager), remain);
asset.settle(poolManager, address(this), remain, false);
}
}
```

This allowance is not needed because `settle()` transfers the tokens to the pool manager, which means the manager itself never has to pull them.

`IERC20(Currency.unwrap(currency)).safeTransfer(address(poolManager), amount);`

Recommendation

Remove the approval.

Resolution

Universal Hook Team: Resolved.

I-11 | Notes About Nonce Behavior

Category	Severity	Location	Status
Warning	Info	MerchantController.sol: 247-252	Acknowledged

Description

MerchantController._validateRequest() calls _useUnorderedNonce() for the merchant of the request and each of the witnesses. It ensures a nonce cannot be used twice by the same address. However, there are a few things that should be noted about this nonce behavior.

When a request is validated, the nonce is invalidated for the current merchant.

```
_useUnorderedNonce(merchant, nonce);
```

This means the merchant won't be able to execute a new request with the same nonce. On the other hand, other merchants are able to use the same nonce, since the nonce is account specific. For the new merchant's request, all of the witnesses of the previous request, plus the previous merchant (if witness as well), will produce invalid signatures.

This is because they have already attested for that nonce in the first request. In addition, if the merchant is also a witness, they won't be able to attest to their own request.

In result, the merchant will have to submit additional signatures of other witnesses if any are available, or create a new request with different nonce.

This also increases the risk of merchants griefing each others. For example, merchant A submits a request with nonceX. Merchant B does the same for a different request and frontruns merchant A. If any of the witnesses are the same, merchant A's transaction will revert.

Recommendation

If this behavior is acceptable, consider documenting it as code comment. Otherwise, rework the nonce system to not use global nonces for different merchants and witnesses.

Resolution

Universal Hook Team: Acknowledged.

I-12 | Unnecessary Storage Read Wastes Gas

Category	Severity	Location	Status
Gas Optimization	Info	UniversalJITOracle.sol: 100	Resolved

Description

In the `safeGetPrice` function of `UniversalJITOracle.sol`, the `round` variable is loaded from storage but never used.

The function reads the entire `OracleRound` struct from storage into memory but then retrieves the price data again via the separate `getPrice(poolId)` call.

This unnecessary storage read wastes gas on every call.

Recommendation

Remove the unused variable assignment to reduce gas consumption.

This change eliminates the redundant `SLOAD` operation and produces cleaner, maintainable code without any change to functionality.

Resolution

Universal Hook Team: Resolved.

I-13 | Redundant onlyPoolManager Modifier Usage

Category	Severity	Location	Status
Gas Optimization	Info	UniversalJitHook.sol: 113	Resolved

Description

The internal _beforeInitialize() function has the onlyPoolManager applied to it. This is not needed because this function is called only in BaseHook.beforeInitialize() which already applies the modifier.

Recommendation

Remove the onlyPoolManager modifier from _beforeInitialize().

Resolution

Universal Hook Team: Resolved.

I-14 | Unused And Missing Event Emissions

Category	Severity	Location	Status
Events	Info	MerchantController.sol: 81-84	Acknowledged

Description

The protocol contains several state-modifying functions that fail to emit appropriate events, reducing transparency and hindering off-chain monitoring.

Specifically, `MerchantController.sol` defines four unused events that should be emitted during critical state changes. Additionally, `MintLimiter.sol` and `UniversalJITOracle.sol` lack event definitions and emissions for important state modifications.

`MerchantController.sol` defines but never emits:
• `AssetWhitelistSet` - should emit when asset whitelist status changes
• `OracleSet` - should emit when oracle configuration is updated
• `PauseSet` - should emit when pause status is toggled
• `EpochReset` - should emit when account epoch is reset

`MintLimiter.sol` defines `MintLimitSet` but never emits it, and lacks events for:
• `_setActorMintLimit()` - when individual actor mint limits are configured
• `pauseAsset()` - when an asset is paused
• `pauseActor()` - when an actor is paused

`UniversalJITOracle.sol` is missing event emissions for:
• `_setPrice()` - when price state is updated

Recommendation

Emit all existing events in their corresponding state-modifying functions in `MerchantController.sol`. Create and emit new events for the missing functions in `MintLimiter.sol` and `UniversalJITOracle.sol`. This will improve protocol transparency and enable proper off-chain event indexing and monitoring.

Resolution

Universal Hook Team: Acknowledged.

I-15 | Pool Creation May Be Front-ran

Category	Severity	Location	Status
Informational	Info	UniversalJitHook.sol: 113-129	Acknowledged

Description

`UniversalJitHook._beforeInitialize()` validates the tokens of the pool are the correct ones and an oracle update for that pool is available.

In case when the oracle update was submitted, but no pool was created, anyone can call `PoolManager.initialize()` with whatever price they want to.

This will have no effect on the functionality of the hook because the price is never used, but may result in unexpected state for the pool.

Recommendation

Keep that in mind when initializing pools.

Resolution

Universal Hook Team: Acknowledged.

I-16 | Unnecessary Type Casting

Category	Severity	Location	Status
Best Practices	Info	UniversalJitHook.sol: 217	Resolved

Description

In the `_computeReportZeroForOne()` and `_computeReportOneForZero()` functions, the `report.jitLowerRange` and `report.jitUpperRange` variables are casted to `uint24`. This is not needed because the variables are already `uint24`s.

```
struct OracleReport {
int24 jitLowerTick;
int24 jitUpperTick;
uint24 jitLowerRange;
uint24 jitUpperRange;
uint256 uAssetLiquidity;
uint256 usdcLiquidity;
uint256 timestamp;
}
```

Recommendation

Remove the casting.

Resolution

Universal Hook Team: Resolved.

I-17 | Inconsistent Naming Conventions

Category	Severity	Location	Status
Best Practices	Info	UniversalJITOracle.sol: 14, UniversalJitHook.sol: 35	Resolved

Description

The codebase contains naming inconsistencies between related contracts. The `UniversalJITOracle` contract (in `UniversalJITOracle.sol`) and the `UniversalJitHook` contract (in `UniversalJitHook.sol`) use different capitalization conventions.

Specifically, `UniversalJITOracle` uses all-caps "JIT" while `UniversalJitHook` uses only the first letter capitalized as "Jit".

Recommendation

Choose one capitalization style and apply it uniformly to both contract names and their corresponding file names.

Resolution

Universal Hook Team: Resolved.

I-18 | Inconsistent Merchant Controller Upgradability

Category	Severity	Location	Status
Upgradeability	Info	UniversalJitHook.sol: 46	Acknowledged

Description

The `merchantController` in `WrappedAssetV2` is immutable, meaning all deployed `uAssets` permanently reference the same controller.

However, in the hook, the merchant controller is not immutable and can be updated via `setMerchantController`.

If the merchant controller is ever changed, the hook will start using the new contract, but existing `uAssets` will still reference the old one.

Newly deployed `uAssets` (using a new implementation with the updated controller) would use the new controller, while previously deployed ones cannot be updated, creating inconsistent behavior across assets.

Recommendation

Consider either making the merchant controller immutable in the hook as well, or adding a `setMerchantController` function to `WrappedAssetV2` to allow updating the merchant controller for already deployed `uAssets` instead of having to deploy a whole new implementation.

Resolution

Universal Hook Team: Acknowledged.

I-19 | Merchant May Be Forced To Trade

Category	Severity	Location	Status
Warning	Info	Global	Acknowledged

Description

One of the known issues of the project is that there exists a griefing vector for the merchant. Users can request to be minted `uAssets` by signing `UniswapX` orders, but later revoke their approval before the actual swap happens.

Because minting `uAssets` and swapping them for the user's tokens happens atomically, both actions will fail, but the merchant will already have purchased the underlying token of the `uAsset` on Coinbase.

This means any malicious user can force the merchant to execute spot buys on Coinbase, which doesn't pose a risk for the solvency of the `uAssets` because they become overcollateralized. However, the merchant will be charged fees for each buy and sell. Furthermore, their USD balance will decrease and unexpected rebalances may be triggered.

After such forced buy happens, if the merchant doesn't immediately sell, but decides to keep the tokens for backing future `uAssets`, any negative price movement will affect the net worth of the merchant.

On the other hand, if the merchant sells, the malicious user can potentially execute the griefing attack again. If there are no strict validations on the orders this may be even repeated until the merchant balance goes close to 0.

Blacklisted users can also make mint/burn revert.

Recommendation

Carefully consider the risks before deploying the offchain system and enforce the needed validation for each request. Migrating to an escrow design would be best.

Resolution

Universal Hook Team: Acknowledged.

I-20 | Misleading NatSpec

Category	Severity	Location	Status
Documentation	Info	WrappedAssetV2.sol: 85	Resolved

Description

The parameter documentation for `WrappedAssetV2.setUserBlacklist()` incorrectly states "The chain to blacklist/whitelist", while in reality it is a user that is being blacklisted or whitelisted.

Recommendation

Fix the comment.

Resolution

Universal Hook Team: Resolved.

I-21 | WrappedAssetV2 Salt May Not Be Unique

Category	Severity	Location	Status
Best Practices	Info	WrapFactoryV2.sol: 26	Resolved

Description

The salt in `WrapFactoryV2.deployBeaconProxy()` uses `abi.encodePacked(name, symbol)`. Different inputs can produce the same packed bytes (e.g., ("ab","c") and ("a","bc")) leading to the same salt. This can result in inability to deploy certain tokens if their address was already used.

Recommendation

Use `abi.encode()` instead.

Resolution

Universal Hook Team: Resolved.

I-22 | Misleading NatSpec

Category	Severity	Location	Status
Documentation	Info	WrappedAssetV2.sol: 73	Resolved

Description

The comment above `getUserBlacklist` states that this function should only be called by the admin. However, it is a public function that can be called by anyone.

Recommendation

Fix the comment.

Resolution

Universal Hook Team: Resolved.

I-23 | Admin Can't Burn For Blacklisted Address

Category	Severity	Location	Status
Unexpected Behavior	Info	WrappedAssetV2.sol: 99	Acknowledged

Description

When an account is blacklisted, the admin can’t burn its `uAsset` balance because the update method checks that the user is not blacklisted, so the tokens remain stuck.

Burning them would require unblacklisting the user first, but that would allow the user to frontrun the burn and swap the tokens into something else.

Recommendation

Consider fixing the update method so that when the `to` address in `_update` is `address(0)`, it does not revert.

Resolution

Universal Hook Team: Acknowledged.

I-24 | Burn Operations Ignore Asset Pause Mechanism

Category	Severity	Location	Status
Validation	Info	MerchantController.sol: 228-231	Resolved

Description

The MintLimiter.sol contract implements a pause mechanism via the pauseAsset() function, which sets an asset's price to 0.

This effectively prevents minting operations, as the _mint() function calls calculateMintValue() which checks the assets price and will revert if the asset is paused.

However, the _burn() function in MerchantController.sol does not include this whitelist check, allowing burn operations to proceed even when an asset has been paused by an admin.

This inconsistency could enable unintended token burning during periods when the asset should be frozen.

Recommendation

Consider adding the _isAssetWhitelisted() check to the _burn() function if that was the intended pause functionality.

To ensure that burn operations respect the pause mechanism in the same manner as mint operations.

Resolution

Universal Hook Team: Resolved.

I-25 | Unrecoverable ERC6909 Balances Risk

Category	Severity	Location	Status
Warning	Info	UniversalJitHook.sol: 408-436	Resolved

Description

The `UniversalJITHook.sol` contract contains the immutable `MERCHANT` address. The `sweepToMerchant()` function sends the `ERC6909` `currency0`/`currency1` of a specific Pool to `MERCHANT`. Once deployed, this address cannot be modified.

In a (unlikely) scenario where the private keys associated with the `MERCHANT` address are lost or compromised, there is no mechanism to redirect future fund sweeps to an alternative address.

Recommendation

Consider implementing a changeable merchant address mechanism.

Resolution

Universal Hook Team: Resolved.

I-26 | Mint Limit Bypass Via Tiny Swaps

Category	Severity	Location	Status
Rounding	Info	MerchantController.sol: 158-166	Resolved

Description

When minting new tokens from USDC -> uAsset swaps, the mint value is calculated and checked against the mint limit:

MintLimiter::calculateMintValue

```
SafeCastLib.toUint104(FixedPointMathLib.mulWad(amount, price));
```

amount is in 18 decimals (uAsset), and price must be in 8 decimals for the result to be in 8 decimals (mulWad), as the documentation states that mint limits have 8 decimal precision:

```
// Note: both mint limit and minted should be denominated in 8 decimals precision
struct Actor {
uint104 mintLimit;
Epoch epoch;
}
```

If amount * price < 1e18, then SafeCastLib.toUint104(FixedPointMathLib.mulWad(amount, price)); rounds down to 0 and completely skips the mint limit check, which is possible if amount is a very low value.

Even if a user batches a large number of these dust mints, the total amount that can bypass the mint limit is bounded to a negligible USD value and does not meaningfully bypass the limit.

Recommendation

Either round up when calculating SafeCastLib.toUint104(FixedPointMathLib.mulWad(amount, price)); or revert if value == 0 within MerchantController::mintFromHook

Resolution

Universal Hook Team: Resolved.

I-27 | MintLimitReached Doesn't Include Current Value

Category	Severity	Location	Status
Best Practices	Info	MintLimiter.sol: 35	Acknowledged

Description

MintLimiter.MintLimitReached() is used to revert when a mint would make the total minted usd value of an actor goes beyond their limit. The error has the following signature:

```
MintLimitReached(uint256 limitValue, uint256 mintValue)
```

The first parameter is the limit that must not be exceeded and the second one is the value that the user tried to mint. The error doesn't include the value that has already been minted until now.

Recommendation

Consider adding a third parameter that shows how much usd value has been already minted.

Resolution

Universal Hook Team: Acknowledged.

I-28 | Unused Role Definitions

Category	Severity	Location	Status
Informational	Info	MerchantController.sol: 49-50	Resolved

Description

The codebase defines several role constants that are not actively utilized in the contracts. Specifically, `MerchantController.sol` declares both `EXECUTOR_ROLE` and `RESPONDER_ROLE`, but neither role is enforced in any function or modifiers.

According to the `README`, these roles were intended to perform specific administrative functions—`EXECUTOR_ROLE` for resetting epochs and `RESPONDER_ROLE` for pausing merchants or witnesses during incidents.

However, the actual pause functionality is instead implemented through a separate `PAUSER_ROLE` defined in `MintLimiter.sol`.

Recommendation

It's recommended to assess the access control strategy across the protocol to align with the documented roles in the `README`.

Either implement the `EXECUTOR_ROLE` and `RESPONDER_ROLE` in their intended functions within `MerchantController.sol`, or remove these unused constants. This will improve code clarity and reduce maintenance burden.

Resolution

Universal Hook Team: Resolved.

I-29 | Warning About Temporary Unbacked uAssets

Category	Severity	Location	Status
Warning	Info	UniversalJITOracle.sol: 32	Acknowledged

Description

The `UniversalJitHook` uses virtual liquidity for both `token0` and `token1` to facilitate `uAsset` → `USDC` swaps.

The USDC virtual liquidity for each pool can exceed the total USDC actually traded into that pool through USDC → `uAsset` swaps. Sponsors confirmed this is intentional, as virtual liquidity shapes the price curve, reduces price movement, and caps trade sizes independently of the actual USDC contributed to that pool.

This can introduce temporary periods where the merchant's available USDC is lower than the amount required to fully collateralize outstanding uAssets. Consider the following example:
`uAssets: uBTC (1 uBTC = 90,000 USDC)` and uETH (1 uETH = 3,000 USDC). Assume pools (uBTC, USDC) and (uETH, USDC).

Bob owns 1 uBTC from direct issuance (deposit 1 WBTC). Carole owns 12 uETH by depositing 12 ETH into custodian. MERCHANT holds 10,000 USDC.

Assume the Merchant's 10,000 USDC was received through USDC → uBTC swaps, and no USDC → uETH swaps occurred.

Alice swaps 90k USDC for 30 uETH. Merchant now holds 100k USDC.

Oracle updates (uBTC, USDC) pool USDC virtual liquidity to 45k (despite only 10k entering that pool).

Bob swaps 0.5 uBTC for 45k USDC, leaving the merchant with 55k.

Alice now holds 30 uETH (worth 90k USDC) and wants to redeem. She can sell only ~18 uETH via the hook, so she burns the remaining 12 uETH and unlocks 12 ETH from custody (the underlying Carole deposited).

Carole is left with 12 uETH that cannot be sold through the hook (insufficient USDC) and cannot be redeemed through custody (since Alice withdrew the underlying). Her 12 uETH become unbacked and unredeemable until the Merchant sells Bob's 0.5 WBTC for USDC, and uses it to purchase 12 ETH to deposit into custodian.

During the timeframe between the burn and the moment when the Merchant sells Bob's underlying and uses the proceeds to re-back the 12 uETH, these tokens remain temporarily unbacked.

Recommendation

Consider documenting and defining procedures (e.g., sweeping frequency or minimum liquidity buffers) to minimize temporary under-backing.

Resolution

Universal Hook Team: Acknowledged.

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
L-01	Swaps May Revert Because Of Clearing	DoS	Low	Resolved
L-02	Burns Do Not Decrease Witness Mint Value	DoS	Low	Resolved
L-03	Swaps May Revert Because Of Mint Value Rounding	Rounding	Low	Resolved
L-04	Exact In Swaps Are Leaked To The Uniswap Pool	Unexpected Behavior	Low	Acknowledged
I-01	Salt Uniqueness Issue Was Fixed In A Wrong File	Best Practices	Info	Resolved
I-02	setUserBlacklist() NatSpec Not Fixed	Documentation	Info	Resolved
I-03	mintLimit Should Be Changed To Int104	Best Practices	Info	Resolved
I-04	Ambiguous Revert Messages In _beforeSwap	Events	Info	Resolved
I-05	Inconsistent Naming Convention	Best Practices	Info	Acknowledged
I-06	maxDeltaTick Validation Not Resolved	Informational	Info	Partially Resolved
I-07	Non-standard Handling Of sqrtPriceLimit == 0	Best Practices	Info	Acknowledged
I-08	Redundant Invariant Checks In Limit Branches	Best Practices	Info	Resolved
I-09	Unused Variable In _burn Function	Gas Optimization	Info	Resolved

Remediation Findings & Resolutions

ID	Title	Category	Severity	Status
I-10	Sweeping Issues Due To Validation	Warning	Info	Acknowledged

L-01 | Swaps May Revert Because Of Clearing

Category	Severity	Location	Status
DoS	Low	UniversalJitHook.sol: 425-427	Resolved

Description

After the hook settles a swap with its `ERC6909` balance, it calls `_clearCurrencyClaims()` to sweep any remaining `ERC6909` balance of that token back to the merchant. However, `UniswapV4` uses flash accounting. This means actual token transfers happen at the end of the transaction.

So if a few swaps happen in the same transaction, it's possible for the hook to have `ERC6909` balance, while at the same time there are not enough funds in the pool manager yet. Because of this, the transaction will revert, since the hook is trying to transfer these tokens from the manager.

Recommendation

When executing `_clearCurrencyClaims()`, cap the amount being cleared to the balance of the pool manager for that token.

Resolution

Universal Hook Team: The issue was resolved in commit 95cce7d.

L-02 | Burns Do Not Decrease Witness Mint Value

Category	Severity	Location	Status
DoS	Low	MerchantController.sol: 231-241	Resolved

Description

The newly added `_decreaseMintValue` function lowers the minted amount of an actor upon `uAsset` burns, preventing the mint limit from exhausting early for the current epoch.

When mint requests are executed, both merchant and witness minted amounts are increased. However, for burn requests, only the merchant minted amount is decreased.

This means that for the same witnesses, the mint limit will be reached earlier than expected, blocking further mint requests from executing for the current epoch.

Recommendation

Within `MerchantController::_burn` consider removing `_decreaseMintValue(account, value);` .

Resolution

Universal Hook Team: The issue was resolved in commit 7b1d7f3.

L-03 | Swaps May Revert Because Of Mint Value Rounding

Category	Severity	Location	Status
Rounding	Low	MintLimiter.sol: 111	Resolved

Description

MintLimiter.calculateMintValue() was changed to revert if the calculated value rounds down to 0.

```
uint104 value = SafeCastLib.toUint104(FixedPointMathLib.mulWad(amount, price));
if(value == 0) revert InvalidMintValue(amount, price);
```

This function is called in both mintFromHook() and burnFromHook(). This means both of these actions may revert - either on its own or due to a malicious user.

For example, when settling an uAsset if the ERC6909 is less than the needed amount, a user can donate the exact amount needed to cause the revert.

The same can happen on sweeps and now swaps as well because of the new _clearCurrencyClaims() function.

Recommendation

Consider removing the revert from calculateMintValue() and instead:

• round value up if the operation is mint

• round value down if the operation is burn

This will avoid reverts and still round against the merchant/hook.

Resolution

Universal Hook Team: The issue was resolved in commit 58d991b.

L-04 | Exact In Swaps Are Leaked To The Uniswap Pool

Category	Severity	Location	Status
Unexpected Behavior	Low	VirtualPrice.sol	Acknowledged

Description

One of the latest changes was done to the VirtualPriceLibrary. The functions for exactIn swaps don't revert anymore if the amount being swapped is more than the maximum allowed amount, but instead cap it.

```
amount0InAfter = amount0In > maxAmount0 ? maxAmount0 : amount0In;
```

This means there may be a difference between the user provided amountSpecified and the actual swapped amount. For example, consider the case where amountInSpecified = -1000, but maxAmount0 = 800.

In that case maxAmount0 will be returned as delta from the hook and the pool manager will calculate the new amountInSpecified to be -200. Until now, all swaps amounts were resulting in 0, the following if was hit in Pool.swap() and no actual swap actions were performed in the pool itself.

```
if (params.amountSpecified == 0) return (BalanceDeltaLibrary.ZERO_DELTA, 0, swapFee, result);
```

With the new changes, it's possible to not hit this if statement and continue executing the swap. If the swap returns a non-zero delta, the user can manipulate the amount paid. However, it's expected that the swap will always return 0 delta, because there is no liquidity added to the pool.

There is another side effect - the price moves at the pool level. This may lead to unexpected behaviors. One of them is that if all available tokens of a liquidity band are bought, the price of the pool can be pushed to its boundaries - either minimum or maximum. Following swaps that use price limits may revert.

For example, we have prices A < B < C < D. User starts buying at C, buys all the tokens and pushes the price at the pool level to its maximum. A new oracle report resupplies liquidity at the same prices (for simplicity). Now, if another buy specifies a limit, the transaction will revert due to the following check in the Pool.

Having swaps execute at the pool level may have another unexplored risks as well.

Recommendation

For this version of the project, consider reverting if the amount swapped is more than the maximum possible

Resolution

Universal Hook Team: Acknowledged.

I-01 | Salt Uniqueness Issue Was Fixed In A Wrong File

Category	Severity	Location	Status
Best Practices	Info	WrapFactoryV2.sol: 26	Resolved

Description

The issue `[I-21] WrappedAssetV2 Salt May Not Be Unique` is located in `WrapFactoryV2`, but the fix was done to the `WrapFactory` file which is out of scope for the review. The issue is still present in `WrapFactoryV2`.

Recommendation

Fix the issue in `WrapFactoryV2`.

Resolution

Universal Hook Team: Resolved.

I-02 | setUserBlacklist() NatSpec Not Fixed

Category	Severity	Location	Status
Documentation	Info	WrappedAssetV2.sol: 85	Resolved

Description

Issue [I-20] Misleading NatSpec is present in WrappedAssetV2.sol, but was fixed in WrappedAsset.sol instead. This leaves the issue still present in the WrappedAssetV2.sol file.

Recommendation

Fix the NatSpec in the WrappedAssetV2.sol file.

Resolution

Universal Hook Team: Resolved.

I-03 | mintLimit Should Be Changed To Int104

Category	Severity	Location	Status
Best Practices	Info	Global	Resolved

Description

`Epoch.minted` was changed to `int104`, but `Operator.mintLimit` and `Actor.mintLimit` were left `uint104`. Because of this, the `mintLimit` is casted to `int104` before it's used.

However, the maximum value that can be represented with `uint104` is larger than `type(int104).max`. Therefore, if the `mintLimit` is set to such a large value, the `MerchantController` will start reverting.

Recommendation

Change `mintLimit` to `int104`. By doing this, you will also get rid of the type casting.

Resolution

Universal Hook Team: Resolved.

I-04 | Ambiguous Revert Messages In _beforeSwap

Category	Severity	Location	Status
Events	Info	UniversalJitHook.sol: 153-154	Resolved

Description

The `_beforeSwap()` function in `UniversalJITHook` contains two separate validation checks that both revert with the identical error message `BlacklistedUser`. The first check validates that the asset is whitelisted, while the second check validates that the caller is not blacklisted.

When either check fails, the caller receives the same revert message despite the failures being caused by two distinct conditions.

A user with a non-whitelisted asset will receive the same error message as a blacklisted user, creating confusion about the actual reason for the transaction failure.

Recommendation

Define and use separate, descriptive revert messages for each validation condition.

This approach provides clear, actionable error information to users and improves the debugging experience for both direct users and external integrators of the protocol.

Resolution

Universal Hook Team: Resolved.

I-05 | Inconsistent Naming Convention

Category	Severity	Location	Status
Best Practices	Info	UniversalJITHook.sol	Acknowledged

Description

There is an inconsistency between the naming of the `UniversalJitHook.sol` file and `UniversalJITOracle.sol` - the former is named with not uppercased `JIT`, even though the contract defined inside, `UniversalJITHook`, capitalizes it.

Recommendation

Rename the file to `UniversalJITHook.sol`

Resolution

Universal Hook Team: Acknowledged.

I-06 | maxDeltaTick Validation Not Resolved

Category	Severity	Location	Status
Informational	Info	UniversalJITOracle.sol: 61-69	Partially Resolved

Description

Issue [I-07] reported that the `UniversalJitOracle` contract has defined the following error, which is unused: `error InvalidMaxDeltaTick();`. This suggests the intention to properly validate the max delta tick value, as the following configured `maxDeltaTick` values can be problematic:

Negative value: The check `delta > maxDeltaTick || delta < -maxDeltaTick` will behave unexpectedly. If `maxDeltaTick = -10`, then `delta > -10` is true for any positive delta, thus blocking price updates.

Zero: Would require `delta == 0` for both ticks, completely preventing price updates.

Excessively large: Defeats the purpose of bounding price movements.

Recommendation

Consider validating maxDeltaTick in the following function and constructor:

```
function setMaxDeltaTick(int24 _maxDeltaTick) external onlyRole(DEFAULT_ADMIN_ROLE) {
if (_maxDeltaTick <= 0 || _maxDeltaTick > MAX_ALLOWED_DELTA) revert InvalidMaxDeltaTick();
maxDeltaTick = _maxDeltaTick;
}
```

Resolution

Universal Hook Team: Partially Resolved.

I-07 | Non-standard Handling Of sqrtPriceLimit == 0

Category	Severity	Location	Status
Best Practices	Info	VirtualPrice.sol	Acknowledged

Description

`VirtualPrice` library treats `sqrtPriceLimit == 0` as a special case that disables price-limit checks. This differs from Uniswap v4, where a limit of 0 is always invalid and causes a revert because it is below the minimum allowed sqrt price,

https://github.com/Uniswap/v4-core/blob/main/src/libraries/Pool.sol#L322-L338.

Recommendation

Consider either reverting when `sqrtPriceLimit == 0` to match Uniswap behavior, or clearly documenting that 0 is treated as a “disable limit” flag.

Resolution

Universal Hook Team: Acknowledged.

I-08 | Redundant Invariant Checks In Limit Branches

Category	Severity	Location	Status
Best Practices	Info	VirtualPrice.sol	Resolved

Description

In the following functions:
• `quoteExactInputZeroForOneWithLimit`
• `quoteExactInputOneForZeroWithLimit`
• `quoteExactOutputOneForZeroWithLimit`

the limit branch contains an invariant check on `sqrtPriceAfter` (e.g. `>= sqrtPriceLimit` or `<= sqrtPriceLimit`).

However, each of these functions calls its corresponding inner function with `sqrtPriceLimit` as the price bound:
• `quoteExactInputZeroForOne(...)`
• `quoteExactInputOneForZero(...)`
• `quoteExactOutputOneForZero(...)`

Those inner functions already enforce the same invariant internally (`sqrtPriceAfter >= sqrtPriceLower` or `<= sqrtPriceUpper`).

Because the limit is passed as the bound, the inner check guarantees the condition before returning, making the outer invariant redundant.

Recommendation

Consider removing the outer invariant checks from:
• `quoteExactInputZeroForOneWithLimit`
• `quoteExactInputOneForZeroWithLimit`
• `quoteExactOutputOneForZeroWithLimit`

Resolution

Universal Hook Team: Resolved.

I-09 | Unused Variable In _burn Function

Category	Severity	Location	Status
Gas Optimization	Info	MerchantController.sol: 235	Resolved

Description

In `_burn`, `calculateMintValue` is called and assigned to `value`, but the variable is never used. This results in unnecessary gas consumption.

Recommendation

Remove the unused calculation to save gas.

Resolution

Universal Hook Team: Resolved.

I-10 | Sweeping Issues Due To Validation

Category	Severity	Location	Status
Warning	Info	Global	Acknowledged

Description

The same pool key validation from _beforeInitialize() was added to sweepToMerchant()

```
if (_isPairCurrency(key.currency0)) {
if (!merchantController.isAssetWhitelisted(address(Currency.unwrap(key.currency1)))) revert
AssetNotAllowed();
} else if (_isPairCurrency(key.currency1)) {
if (!merchantController.isAssetWhitelisted(address(Currency.unwrap(key.currency0)))) revert
AssetNotAllowed();
} else {
revert PairCurrencyNotAllowed();
}
```

If an asset is paused, it will be impossible to claim any hook ERC6909 balance until that asset is unpaused again. Also, if there are no other pools with a whitelisted uAssets, the USDC ERC6909 balance will be stuck as well.

Even if that check wasn't present there, unlockCallback() calls burnFromHook() if the hook holds any ERC6909 balance of the uAsset. In burnFromHook(), the flow reverts if the asset is paused or the hook is disabled from minting/burning.

```
if (!_isAssetActiveForActor(asset, HOOK_ADDRESS)) revert AssetNotActive();
```

This means any sweeps with positive uAsset balance will revert. Even if the hook doesn't have any, a user can forcibly send 1 wei to trigger the call to burnFromHook(). The same way as above, if there are no other pools that can be swept, the USDC balance will be stuck.

Recommendation

Keep in mind this behavior.

Resolution

Universal Hook Team: Acknowledged.

Remediation Findings & Resolutions V2

ID	Title	Category	Severity	Status
L-01	_beforeInitialize() Should Have Access Control	DoS	Low	Resolved
I-01	Unused Parameter In _clearCurrencyClaims()	Gas Optimization	Info	Resolved
I-02	Unused Error In MintLimiter	Best Practices	Info	Resolved

L-01 | _beforeInitialize() Should Have Access Control

Category	Severity	Location	Status
DoS	Low	UniversalJitHook.sol: 127	Resolved

Description

For exactIn swaps, the `amountToSwap` returned from the hook may be non-zero in case the caller specified an `exactIn > maxExactIn` allowed for the current price band.

In that case, the `sqrtPriceLimit` specified by the user (or `V4Router`) is checked against the current `sqrtPrice` (Pool.sol).

The problem is that anyone can initialize whitelisted USDC/`uAsset` pools specifying an arbitrary `sqrtPriceX96`. An attacker can front-run pool initialization and set it to an extreme value, such as `TickMath.MIN_SQRT_PRICE + 1` or `TickMath.MAX_SQRT_PRICE - 1`.

This will cause DoS to any 0 to 1 or 1 to 0 swaps (depending on initial price set) where there is a change in the `exactIn` delta specified by a user. This will continue until users swap the opposite direction, as it will move the price away from the extreme.

Note that a similar issue was reported and acknowledged in the review phase: `[I-15] Pool Creation May Be Front-ran`. At that time there was no issue since pool prices were not used, however that has since changed.

Recommendation

The first parameter of `_beforeInitialize()` is the address who called `PoolAddress.initialize()`. Use it to validate only trusted addresses can create pools.

Resolution

Universal Hook Team: The issue was resolved in commit 32aca0b.

I-01 | Unused Parameter In _clearCurrencyClaims()

Category	Severity	Location	Status
Gas Optimization	Info	UniversalJitHook.sol: 433	Resolved

Description

When settling with claims, the leftover claims is passed into `_clearCurrencyClaims`, but never used because the remaining claims amount is fetched again:

```
uint256 claims = poolManager.balanceOf(address(this), currency.toId());
if (claims == 0) return;
```

In fact, the check `if (claims > useAmt)` already implies a non-zero claims amount, making the above code redundant.

Recommendation

Consider using the passed in `amount` parameter instead of re-fetching claims, or remove the `amount` parameter to save gas:

```
function _clearCurrencyClaims(Currency currency, uint256 amount) internal {
uint256 poolBalance = currency.balanceOf(address(poolManager));
uint256 available = amount < poolBalance ? amount : poolBalance;
if (available == 0) return;
...
}
```

Resolution

Universal Hook Team: The issue was resolved in commit cabbdef.

I-02 | Unused Error In MintLimiter

Category	Severity	Location	Status
Best Practices	Info	MintLimiter.sol: 36	Resolved

Description

The L-03 issue was fixed and the revert was removed. However, the `InvalidMintValue` error is still present but unused.

Recommendation

Consider removing the unused `InvalidMintValue` error.

Resolution

Universal Hook Team: The issue was resolved in commit d0f845a.

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