SPEARBIT

Router Rebates Security Review

Auditors

Noah Marconi, Lead Security Researcher

R0bert, Lead Security Researcher

Ladboy233, Security Researcher

Hake, Associate Security Researcher

Report prepared by: Lucas Goiriz

June 26, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Uniswap is an open source decentralized exchange that facilitates automated transactions between ERC20 token tokens on various EVM-based chains through the use of liquidity pools and automatic market makers (AMM).

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Router Rebates according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 1 days in total, Uniswap Labs engaged with Spearbit to review the router-rebates protocol. In this period of time a total of **28** issues were found.

Summary

Project Name	Uniswap Labs
Repository	router-rebates
Commit	4ce7a4a1
Type of Project	DeFi, AMM
Audit Timeline	May 15th to May 16th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	4	4	0
High Risk	2	2	0
Medium Risk	5	5	0
Low Risk	7	5	2
Gas Optimizations	0	0	0
Informational	10	6	4
Total	**28**	**22**	**6**

5 Findings

5.1 Critical Risk

5.1.1 ClaimWithSignature signatures can be replayed

Severity: Critical Risk

Context: RouterRebates.sol#L60-L62, RouterRebates.sol#L70-L71

Description: In the `RouterRebates` contract, the `claimWithSignature` function uses a mapping of `lastBlockClaimed` to track which blocks have already been claimed for a given chain ID and beneficiary. This is intended to prevent replaying claims across the same block ranges.

However, the implementation contains a flaw:

```
// consume the block number to prevent replaying claims
lastBlockClaimed[chainId][beneficiary] = blockRange.endBlockNumber - 1;
```

By setting `lastBlockClaimed` to `endBlockNumber - 1` rather than `endBlockNumber`, the function creates a vulnerability where subsequent calls can include the `endBlockNumber` in their block range. The validation check:

```
if (blockRange.startBlockNumber < lastBlockClaimed[chainId][beneficiary]) revert InvalidBlockNumber();
```

only prevents claims where the `startBlockNumber` is less than the last claimed block. This means a claim with `startBlockNumber` equal to the previous claim's `endBlockNumber` will pass validation, allowing the last block to be claimed multiple times.

Recommendation: Update the `claimWithSignature` function to store the exact `endBlockNumber` and prevent any block from being claimed more than once:

```
- lastBlockClaimed[chainId][beneficiary] = blockRange.endBlockNumber - 1;
+ lastBlockClaimed[chainId][beneficiary] = blockRange.endBlockNumber;

- if (blockRange.startBlockNumber < lastBlockClaimed[chainId][beneficiary]) revert InvalidBlockNumber();
+ if (blockRange.startBlockNumber <= lastBlockClaimed[chainId][beneficiary]) revert
↪  InvalidBlockNumber();
```

Uniswap Labs: Fixed in PR 42.

Spearbit: Fix verified.

5.1.2 Missing address verification for swap logs

Severity: Critical Risk

Context: rebate.ts#L42-L45

Description: In the `calculateRebate` function, when processing transaction receipts, the code parses all logs matching the `Swap` event signature without verifying the address that emitted these events:

```
const swapEvents = parseEventLogs({
  abi: abi,
  logs: txnReceipt.logs,
}).filter((log) => log.eventName === "Swap");
```

The function then processes these events and calculates rebates based on the event data. For each event, it checks only if the pool `ID` exists in the database and has associated hooks:

```
const { id } = swapEvent.args;
const result = await dbClient
  .select({ hooks: schema.pool.hooks })
  .from(schema.pool)
  .where(eq(schema.pool.poolId, id));

if (result.length > 0 && result[0]?.hooks !== zeroAddress) {
  return rebatePerSwap + rebatePerHook;
}
```

However, the code never validates that the event originated from the legitimate UniswapV4 `PoolManager` contract. This allows an attacker to:

- Deploy a malicious contract with the same event signature as UniswapV4's `Swap` event.
- Emit events with pool IDs that match entries in the database.
- Receive rebates for these spoofed events, despite not actually using the UniswapV4 protocol.

Recommendation: Add verification of the event source address to ensure events are only processed from legitimate UniswapV4 contracts:

```
  const swapEvents = parseEventLogs({
    abi: abi,
    logs: txnReceipt.logs,
  }).filter((log) => log.eventName === "Swap" &&
+           log.address === UNISWAP_V4_POOL_MANAGER_ADDRESS);
```

With the addition of storing `PoolManager` contract address for reference.

Uniswap Labs: Fixed in PR 41.

Spearbit: Fix verified.

5.1.3 Sign endpoint can be abused by providing the same tx multiple times to inflate the rebated amount

Severity: Critical Risk

Context: main.ts#L15-L22

Description: The aggregator `/sign` endpoint allows providing a comma-separated list of `txnHashes`. If the same transaction hash appears multiple times, the aggregator treats each entry as a distinct transaction. For example:

```
curl -G '.../sign' \
  --data-urlencode 'chainId=11155111' \
  --data-urlencode 'txnHashes=0xABC,0xABC,0xABC'
```

causes the aggregator to run `calculateRebate` thrice, summing the same gas usage and `blockNumber` data. Consequently, the amount returned is effectively tripled. This is demonstrated by the example below using 2 $\times$ vs. 4 $\times$ duplicates, where the resulting signature's amount scales accordingly:

```
# X4 -> Amount = 189226053346140
% curl -G 'https://router-rebates-testnet.up.railway.app/sign' --data-urlencode 'chainId=11155111'
↪   --data-urlencode 'txnHashes=0x6a7fb847ae79fbd3689e8c103c8b8c35a27568ab7cf51595d325faa9e559fafe,0x6a ⌋
↪   7fb847ae79fbd3689e8c103c8b8c35a27568ab7cf51595d325faa9e559fafe,0x6a7fb847ae79fbd3689e8c103c8b8c35a2 ⌋
↪   7568ab7cf51595d325faa9e559fafe,0x6a7fb847ae79fbd3689e8c103c8b8c35a27568ab7cf51595d325faa9e559fafe'
{"claimer":"0x7024cc7e60D6560f0B5877DA2bb921FCbF1f4375","signature":"0x720bb125f1e8e8a5811ff1b96abad097 ⌋
↪   ed7cc0c2dd052877277b63acb6ce7931223931becc49c95f2fab52c2be44ff1de280e0ceb55a362d7f95ce41d7a456781c" ⌋
↪   ,"amount":"189226053346140","startBlockNumber":"7839634","endBlockNumber":"7839634"}%

# X2 -> Amount = 94613026673070
% curl -G 'https://router-rebates-testnet.up.railway.app/sign' --data-urlencode 'chainId=11155111'
↪   --data-urlencode 'txnHashes=0x6a7fb847ae79fbd3689e8c103c8b8c35a27568ab7cf51595d325faa9e559fafe,0x6a ⌋
↪   7fb847ae79fbd3689e8c103c8b8c35a27568ab7cf51595d325faa9e559fafe'
{"claimer":"0x7024cc7e60D6560f0B5877DA2bb921FCbF1f4375","signature":"0xdbd94e3bb5f8d2efbf07697734862cc7 ⌋
↪   5814b91714098ba9099506bc4c48f77c3c598613de19b5b6f0c32cd7ad9e933c34a3ff718131a3bb9f8befc5aba6fc461b" ⌋
↪   ,"amount":"94613026673070","startBlockNumber":"7839634","endBlockNumber":"7839634"}%
```

Recommendation: De-duplicate transaction hashes before computing. For example:

```
const uniqueTxns = Array.from(new Set(txnHashList));
// proceed with uniqueTxns only
```

Uniswap Labs: Fixed in PR 43.

Spearbit: Fix verified.

5.1.4 Malicious beneficiary can claim rebates for transactions from other routers

Severity: Critical Risk

Context: main.ts#L23

Description: The router api's internal `batch` function takes in an array of transaction hashes for processing. The hashes are supplied by the caller and require validation.

- Scenario 1: Without validation, a malicious caller may order the array of transactions to include their transaction first, followed by several transactions from other routers.

  The `batch` function processes the array by fetching transaction receipts for each hash in the array, processing, and returning a promise:

  ```
  Promise<{
    beneficiary: Address;
    gasToRebate: bigint;
    txnHash: `0x${string}`;
    blockNumber: bigint;
  }>
  ```

  Then only the first element in the processed array is referenced to determine the beneficiary:

  ```
  const result = await Promise.all(
    txnHashes.map((txnHash) => calculateRebate(publicClient, txnHash))
  );

  // ... snip...

  const beneficiary: `0x${string}` = result[0].beneficiary;
  ```

- Scenario 2: Even if a single transaction is supplied, there is opportunity for malicious behavior within the transaction. The `calculateRebate` function, responsible for fetching the transaction receipts and parsing the logs, references the amounts from any log with `log.eventName === "Swap"`. This means that any bundled calls are credited to the first to emit the event. In other words, a 4337 bundle with a `Swap` event credited to one benefitciary, followed by sever other `Swap`s that should be credited to other beneficiaries, are all credited to the first. There is a TODO in the code alluding to this scenario: `// TODO: require all events are from the same sender`.

Recommendation:.

- Scenario 1: Given the two scenarios, there are two places in the current codebase to amend. The first in `util/main.ts`:

```
-   const amount = result.reduce(
-     (total: bigint, data) => total + data.gasToRebate,
-     0n
-   );
  const beneficiary: `0x${string}` = result[0].beneficiary;
  const claimer = await getRebateClaimer(publicClient, beneficiary);
-   const startBlockNumber = result.reduce(
-     (min: bigint, data) => (data.blockNumber < min ? data.blockNumber : min),
-     result[0].blockNumber
-   );
-   const endBlockNumber = result.reduce(
-     (max: bigint, data) => (data.blockNumber > max ? data.blockNumber : max),
-     result[0].blockNumber
-   );

+   // reduce start block, end block, and amount
+   const { amount, startBlockNumber, endBlockNumber } = result.reduce(
+     (acc, data) => {
+       // verify beneficiary is the same
+       if (data.beneficiary !== beneficiary) {
+         throw new Error("Beneficiary is not the same");
+       }
+
+       // Start and end block numbers
+       acc.startBlockNumber =
+         data.blockNumber < acc.startBlockNumber
+           ? data.blockNumber
+           : acc.startBlockNumber;
+
+       acc.endBlockNumber =
+         data.blockNumber > acc.endBlockNumber
+           ? data.blockNumber
+           : acc.endBlockNumber;
+
+       // amount
+       acc.amount += data.gasToRebate;
+
+       return acc;
+     },
+     {
+       amount: 0n,
+       startBlockNumber: result[0].blockNumber,
+       endBlockNumber: result[0].blockNumber,
+     }
+   );
```

The above edit collapses the iterators into one for efficiency as well.

Scenario 2: The second adds a check in `util/rebates.ts`:

```
  const rebates = await Promise.all(
    swapEvents.map(async (swapEvent) => {
      const { id } = swapEvent.args;
      const result = await dbClient
        .select({ hooks: schema.pool.hooks })
        .from(schema.pool)
        .where(eq(schema.pool.poolId, id));

      if (result.length > 0 && result[0]?.hooks !== zeroAddress) {
-        return rebatePerSwap + rebatePerHook;
+        return { amount: rebatePerSwap + rebatePerHook, sender: swapEvent.args.sender};
      } else {
-        return 0n;
+        return { amount: 0n, sender: swapEvent.args.sender}
      }
    })
  );
- let gasUsedToRebate = rebates.reduce((total, rebate) => total + rebate, 0n);
+ let { gasUsedToRebate } = rebates.reduce((acc, rebate) => {
+   if (acc.sender !== rebate.sender) {
+     throw new Error("Sender is not the same");
+   }
+   acc.gasUsedToRebate += rebate.amount;
+   return acc;
+ }, { gasUsedToRebate: 0n, sender: rebates[0].sender });
```

Uniswap Labs: Fixed in PR 46.

Spearbit: Fix verified.

5.2 High Risk

5.2.1 Lack of rate limit on sign endpoint leads to alchemy key exhaustion and DoS

Severity: High Risk

Context: chain.ts#L20, chain.ts#L30, chain.ts#L40, chain.ts#L50

Description: Currently, the aggregator /sign endpoint can be queried freely without authentication or throttling. Each incoming request triggers RPC calls to Alchemy (or similar providers) to fetch transaction receipts, blocks, logs, etc. If an attacker sends enough requests in quick succession, they can consume the aggregator's entire Alchemy rate limit or subscription credits. Once those are gone, legitimate queries fail and the service effectively experiences a denial of service.

Recommendation: Consider rate limiting the endpoint to some safe threshold. For example, use IP-based or token-based limiting to only allow a certain number of requests per second. On the other hand, consider limiting the amount of transaction hashes per request. For instance, a maximum of 20 unique transactions at once to control the aggregator's workload. Finally, consider maintaining multiple sign endpoints with separate API keys if the aggregator needs higher concurrency or resilience.

Uniswap Labs: Fixed in PR 51.

Spearbit: Fix verified.

5.2.2 Alchemy API key leakage and DoS through 429 error

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When the aggregator is flooded with calls to the `/sign` endpoint, it exhausts Alchemy's rate limit or compute units. Alchemy then responds with a 429 status including a body that references a URL like: `https://eth-sepolia.g.alchemy.com/v2/API_KEY`.

This discloses the aggregator's Alchemy API key in plain text. Attackers can glean this key from the aggregator's logs or error messages and then use it further draining the subscription or forging data requests. Additionally, repeated 429 errors cause partial unavailability for legitimate requests.

Recommendation: Ensure that API keys are never displayed in error logs. For instance, parse out the key if a 429 or other error is returned.

Uniswap Labs: Fixed in PR 56.

Spearbit: Fix verified.

5.3 Medium Risk

5.3.1 Unbounded ponder database growth from "Initialize" events

Severity: Medium Risk

Context: index.ts#L4-L14

Description: In the aggregator's `index.ts`:

```
ponder.on("PoolManager:Initialize", async ({ event, context }) => {
  await context.db.insert(schema.pool).values({
    poolId: event.args.id,
    ...
    chainId: context.network.chainId,
  });
});
```

Every time a new Uniswap v4 pool is created on any supported chain, it's inserted into `schema.pool`. Because the aggregator is configured for multiple networks, if they see heavy pool creation activity, the database can balloon indefinitely. With no built-in constraints or cleanup, the aggregator risks running out of disk space crashing or seeing very slow queries.

Recommendation:

- Add a pruning/archival strategy if you only need pools from recent history.
- Limit indexing to pools with a minimal TVL or certain popular tokens.
- Only index pools that have a hook set.
- Monitor the size and performance of the underlying DB so it doesn't silently fail or crash due to unbounded growth.

Uniswap Labs: Fixed in PR 44.

Spearbit: Fix verified.

5.3.2 Signer api is susceptible to reorgs

Severity: Medium Risk

Context: main.ts#L25-L32

Description: At the smart contract level, duplicate rebates are handled by referencing start and end block dates. The design aims to rebate for a complete range and refuse rebates within an already claimed range. The signer API does not control for how old, or far from current block height, transaction hashes are. Meaning, the signer will sign for any hash the moment a transaction is included in a block. If and when a reorg occurs, the block a transaction was included in may change, however, the previously supplied signature will still be valid and accepted when submitted to the smart contract.

Recommendation: Similar to CEX deposits, wait until all transactions are some number of blocks deep before allowing the signing service to sign.

Uniswap Labs: Fixed in PR 45.

Spearbit: Fix verified.

5.3.3 Consider handle the tx with no swap event gracefully

Severity: Medium Risk

Context: main.ts#L5-L24

Description:

```
const result = await Promise.all(
  txnHashes.map((txnHash) => calculateRebate(publicClient, txnHash))
);
```

The function can return an empty result if no swap events is detected:

```
export async function calculateRebate(
  client: PublicClient,
  txnHash: `0x${string}`
): Promise<{
  beneficiary: Address;
  gasToRebate: bigint;
  txnHash: `0x${string}`;
  blockNumber: bigint;
}> {
  const txnReceipt = await client.getTransactionReceipt({ hash: txnHash });
  const { rebatePerSwap, rebatePerHook, rebateFixed } =
    await getRebatePerEvent();

  // Use baseFee and do not use priorityFee, otherwise miners will set a high priority fee (paid back to
  ↪   themselves)
  // and be able to wash trade
  const gasPrice = (
    await client.getBlock({ blockNumber: txnReceipt.blockNumber })
  ).baseFeePerGas!;

  const swapEvents = parseEventLogs({
    abi: abi,
    logs: txnReceipt.logs,
  }).filter((log) => log.eventName === "Swap");

  if (swapEvents.length === 0) {
    return {
      beneficiary: zeroAddress,
      gasToRebate: 0n,
      txnHash: "0x0",
      blockNumber: 0n,
    };
  }
```

If `beneficiary` returns the `address(0)`,

```
const claimer = await getRebateClaimer(publicClient, beneficiary);
```

querying `claimer` when it is the `address(0)` will raise an error:

```
export async function getRebateClaimer(
  publicClient: PublicClient,
  beneficiary: `0x${string}`
): Promise<`0x${string}`> {
  return await publicClient.readContract({
    address: beneficiary,
    abi: [parseAbiItem("function rebateClaimer() view returns (address)")],
    functionName: "rebateClaimer",
  });
}
```

Recommendation: Consider returning early if no swap event is detected.

Uniswap Labs: Fixed in PR 48.

Spearbit: Fix verified.

5.3.4 EIP4337 bundled transactions are not supported

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `FindSender()` function iterates over a list of transaction receipts and searches their logs for the `SwapEvId` event from the pool manager contract. Upon locating a matching swap log, it checks whether the corresponding pool ID (i.e. `l.Topics[1]`) is present in the poolids map. If so, it immediately returns the sender address from `l.Topics[2]`, exiting the function on the first discovered match:

```
func (c *OneChain) FindSender(receipts []*types.Receipt, poolids map[common.Hash]binding.PoolKey)
↪   common.Address {
    pmAddr := Hex2addr(c.PoolMgr)
    swapEvId := Hex2hash(SwapEvId)
    for _, r := range receipts {
        for _, l := range r.Logs {
            if l.Address == pmAddr && l.Topics[0] == swapEvId {
                if _, ok := poolids[l.Topics[1]]; ok {
                    return common.BytesToAddress(l.Topics[2][12:])  // Returns immediately after finding
                    ↪   first match
                }
            }
        }
    }
    return ZeroAddr
}
```

This early return ignores all subsequent logs and, in turn, any additional routers that may appear later within the same transaction. As a result, multiple swaps in the same transaction (potentially from different routers) remain unrecognized. For instance, consider a single transaction in block 1337 that executes:

- `swap1` for `Router1`.
- `swap2` for `Router2`.

As soon as the function encounters the `swap1` log for `Router1`, it returns Router1 as `beneficiary` without checking the subsequent logs. If `Router2` has previously set `lastBlockClaimed[chainid][Router2]` to 1337 in `RouterRebates`, then the second swap from `Router2` is never identified. Because the function never recognizes that second router's presence, any claim attempting to reference `Router2`'s swap cannot use this transaction data.

Recommendation: Consider adding an extra parameter to the endpoint that identifies the router address upon we want to execute the claim.

Uniswap Labs: Fixed in PR 46.

Spearbit: Fix verified.

5.3.5 Incorrect minimum block number in batch() due to zero-rebate transactions

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: When the `batch` function processes transactions that do not produce a rebate (`gasToRebate = 0n`), it still includes them in the reducer used to determine the minimum and maximum block numbers. Since `calculateRebate` sets `blockNumber = 0n` for zero-rebate transactions, the reducer will yield `startBlockNumber = 0`, even if legitimate rebate-bearing transactions occur at a much higher block. This is counterintuitive, as the "real" earliest relevant block should reflect only those transactions that actually triggered a swap event. Illustration:

```
const startBlockNumber = result.reduce(
  (min: bigint, data) => (data.blockNumber < min ? data.blockNumber : min),
  BigInt(Number.MAX_SAFE_INTEGER)
);
```

Because some entries in `result` may have `blockNumber = 0n`, `startBlockNumber` becomes 0 if any transaction returned no valid swap event.

Recommendation: First, filter out transactions with `gasToRebate = 0n` prior to calculating `startBlockNumber` and `endBlockNumber`. Only transactions that truly generate a rebate should impact the block range. For instance:

```
const relevantResults = result.filter((r) => r.gasToRebate > 0n);
if (relevantResults.length === 0) {
  // early return with zero results
} else {
  // reduce relevantResults to find min & max block numbers
}
```

This approach ensures `startBlockNumber` and `endBlockNumber` reflect only the valid, rebate-producing transactions.

Uniswap Labs: Fixed in PR 64.

Spearbit: Fix verified.

5.4 Low Risk

5.4.1 Incorrect zero-address check in setSigner function

Severity: Low Risk

Context: RouterRebates.sol#L92-L95

Description: In the `RouterRebates` contract, the function to set a new signer is:

```
function setSigner(address _signer) external onlyOwner {
    require(signer != address(0));
    signer = _signer;
}
```

It checks `signer != address(0)`, but that's the old signer variable. The correct check should be `require(_signer != address(0))`, otherwise the owner can accidentally set the new signer to `address(0)` if the old one was non-zero, disabling signature verification.

Recommendation: Consider updating the require check in the `setSigner` function to:

```
require(_signer != address(0), "zero signer not allowed");
```

Uniswap Labs: Fixed in PR 53.

Spearbit: Fix verified.

5.4.2 Lack of validation and revert handling when querying `rebaseClaimer`

Severity: Low Risk

Context: main.ts#L24

Description: According to the `Readme.md`

> Because rebates are claimed on *Unichain* for trades happening on other networks, *beneficiaries* need to specify the authorized claimer.
> The swap router contract (the contract calling `poolManager.swap()`) should implement this function selector:

```
function rebateClaimer() external view returns (address);
```

> See `IRebateClaimer` for the full interface.
> The wallet which claims the rebate is specified by this address, *`rebateClaimer`*. The backend system performs verification against this address, to prevent griefing scenarios where malicious attackers are claiming rebates they are not entitled to.

However, the function `getRebateClaimer` just query the claimer directly. If the `beneficiary` contract get hacker and malicious actor may change the `rebateClaimer()` to claim the gas refund.

```
export async function getRebateClaimer(
  publicClient: PublicClient,
  beneficiary: `0x${string}`
): Promise<`0x${string}`> {
  return await publicClient.readContract({
    address: beneficiary,
    abi: [parseAbiItem("function rebateClaimer() view returns (address)")],
    functionName: "rebateClaimer",
  });
}
```

The code also does not handle the case when the function call `rebaseClaimer` intentionally revert, which can disrupt the signing operation.

Recommendation: Treat `rebateClaimer` as an untrusted external call and handle the revert case and follow the readme to performs verification against this address, to prevent griefing scenarios where malicious attackers are claiming rebates they are not entitled to.

Uniswap Labs: Fixed in PR 49.

Spearbit: Fix verified.

5.4.3 Consider query `getRebatePerEvent` before `calculateRebate` to avoid making 3 RPC read per tx hash

Severity: Low Risk

Context: main.ts#L5-L17

Description: If the `txnHashes` array contains 1000 tx hashes, `calculateRebate` will be executed 1000 times as for each `calculateRebate` call, there will be one `await getRebatePerEvent()` call:

```
export async function calculateRebate(
  client: PublicClient,
  txnHash: `0x${string}`
): Promise<{
  beneficiary: Address;
  gasToRebate: bigint;
  txnHash: `0x${string}`;
  blockNumber: bigint;
}> {
  const txnReceipt = await client.getTransactionReceipt({ hash: txnHash });
  const { rebatePerSwap, rebatePerHook, rebateFixed } =
    await getRebatePerEvent();
```

The function `getRebatePerEvent` just queries the rebate configuration by executing 3 `readContract` calls.

```
async function getRebatePerEvent(): Promise<{
  rebatePerSwap: bigint;
  rebatePerHook: bigint;
  rebateFixed: bigint;
}> {
  const client = getClient(Number(process.env.REBATE_CHAIN_ID));
  const rebatePerSwap = await client.readContract({
    address: process.env.REBATE_ADDRESS as Address,
    abi: [parseAbiItem("function rebatePerSwap() view returns (uint256)")],
    functionName: "rebatePerSwap",
  });
  const rebatePerHook = await client.readContract({
    address: process.env.REBATE_ADDRESS as Address,
    abi: [parseAbiItem("function rebatePerHook() view returns (uint256)")],
    functionName: "rebatePerHook",
  });
  const rebateFixed = await client.readContract({
    address: process.env.REBATE_ADDRESS as Address,
    abi: [parseAbiItem("function rebateFixed() view returns (uint256)")],
    functionName: "rebateFixed",
  });
  return { rebatePerSwap, rebatePerHook, rebateFixed };
}
```

Therefore:

- If the `txnHashes` array contains 1000 tx hashes, 3000 read contract RPC calls will be performed.
- If the `txnHashes` array contains 10000 tx hashes, 30000 read contract RPC calls will be performed.

Given that the code does not validate the max length of the `txHash` array, malicious users can supply a very long list of transaction hashes to quickly exhaust the API call rate limit and hinder application performance.

Recommendation: Consider running `getRebatePerEvent`:

```
const { rebatePerSwap, rebatePerHook, rebateFixed } =
  await getRebatePerEvent();
```

before:

```
const result = await Promise.all(
  txnHashes.map((txnHash) => calculateRebate(publicClient, txnHash))
);
```

The code can be updated to:

```
const { rebatePerSwap, rebatePerHook, rebateFixed } =
  await getRebatePerEvent();

const result = await Promise.all(
  txnHashes.map((txnHash) => calculateRebate(publicClient, txnHash,  rebatePerSwap, rebatePerHook,
  ↪  rebateFixed ))
```

This way, the code will not execute 3 RPC calls per transaction hash. Validating the `txnHash` array length is also recommended.

```
if (txnHashes.length > MAX_TX_ARRAY_LENGTH) {
  throw new Error(`Too many transaction hashes: ${txnHashes.length} provided, maximum allowed is
  ↪  ${MAX_TX_ARRAY_LENGTH}.`);
}
```

Uniswap Labs: Fixed in PR 50.

Spearbit: Fix verified.

5.4.4 Missing block range validation in `handleOutput` function

Severity: Low Risk

Context: RouterRebates.sol#L147-L159

Description: In the `handleOutput()` function, there is no validation to ensure that the beginning block number (`beginBlk`) is less than the ending block number (`endBlk`). This validation is correctly implemented in the `claimWithSignature()` function but is missing from the ZK proof claim path. The function reads the block range values from the app circuit output:

```
uint64 beginBlk = uint64(bytes8(_appOutput[40:48]));
uint64 endBlk = uint64(bytes8(_appOutput[48:56]));

require(beginBlk > lastBlockClaimed[uint256(chainid)][router], "begin blocknum too small");
lastBlockClaimed[uint256(chainid)][router] = endBlk;
```

While the function verifies that `beginBlk` is greater than the `lastBlockClaimed` value, it does not verify the internal consistency of the block range itself.

Recommendation: Add a check to ensure that `beginBlk` is less than `endBlk` in the `handleOutput()` function:

```
  function handleOutput(uint64 chainid, bytes calldata _appOutput) internal returns (uint256) {
      require(_appOutput.length == 72, "incorrect app output length");
      address router = address(bytes20(_appOutput[0:20]));
      address claimer = address(bytes20(_appOutput[20:40]));
      require(msg.sender == claimer, "msg.sender is not authorized claimer");
      uint64 beginBlk = uint64(bytes8(_appOutput[40:48]));
      uint64 endBlk = uint64(bytes8(_appOutput[48:56]));

+     require(beginBlk <= endBlk, "invalid block range");
      require(beginBlk > lastBlockClaimed[uint256(chainid)][router], "begin blocknum too small");
      lastBlockClaimed[uint256(chainid)][router] = endBlk;
      return uint128(bytes16(_appOutput[56:72]));
  }
```

Uniswap Labs: Fixed in PR 59.

Spearbit: Fix verified.

5.4.5 ZK app gas parameters can differ from on-chain RouterRebates config

Severity: Low Risk

Context: RouterRebates.sol#L32-L35

Description: Currently, the ZK application uses hard-coded value for `GasPerSwap` (and other rebate-related parameters) in `server/config.toml`:

```
db = "localhost:26257"
grpcport = 9001
httpport = 9002
# send reqs to prove server which generates app proof then submit to brv gateway
prover = "http://localhost:9003"

[[multichain]]
chainID = 11155111
name = "Sepolia"
gateway = "https://sepolia.infura.io/v3/api_key"
BlkInterval = 12
BlkDelay = 2

GasPerSwap = 80000 <---------------------------------------------
PoolMgr = "0x8C4BcBE6b9eF47855f97E675296FA3F6fafa5F1A"
VkHash = ""
```

Meanwhile, the on-chain `RouterRebates` contract separately tracks values like `rebatePerSwap` internally:

```
// (n * rebatePerSwap) + rebateFixed
uint256 public rebatePerSwap = 80_000; // gas units to rebate per swap event
uint256 public rebatePerHook = 0;
uint256 public rebateFixed = 80_000; // fixed gas units to rebate (to be appended to the total rebate)
```

These two values can easily diverge. If they do, the ZK circuit will compute a rebate that doesn't match what the on-chain contract expects, and therefore, users using the `claimWithSignature` will get a different rebated amount than if they used the `claimWithZkProof` function to execute their rebate claim.

Recommendation: Introduce a single source of truth for gas rebate parameters. Either load the on-chain `rebatePerSwap` parameter in the ZK application dynamically or ensure that updating the on-chain contract automatically updates the ZK app config. This synchronization prevents mismatches and ensures the off-chain proof generation matches on-chain logic.

Uniswap Labs: Acknowledged. We will coordinate on any updates with the brevis team if they were to happen.

Spearbit: Acknowledged.

5.4.6 GasPerTx missing in ZK app configuration

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the ZK App's `server/config.toml` file, `GasPerTx` (the equivalent of a fixed per-transaction rebate) is never set. If the ZK application expects this parameter but cannot find it, it likely defaults to zero in the circuit's calculations which differs from the on-chain `RouterRebates` default config:

```
uint256 public rebateFixed = 80_000;
```

Consequently, the total rebate might omit any fixed per-tx component, yielding a mismatch between what the on-chain contract expects and what the ZK app computes. Once again, this will also cause that the amounts rebated between `claimWithSignature` and `claimWithZkProof` differ for the same set of transactions.

Recommendation: Add a `GasPerTx` entry to the config file so the ZK app accurately reflects the intended `rebateFixed` portion of gas rebates. Ensure this value matches the `rebateFixed` value set at the `RouterRebates` contract. This ensures consistent rebate logic between the on-chain contract and the off-chain circuit.

Uniswap Labs: Acknowledged. The brevis team is opting to not make the change at this time, given that changing the value is permissioned and unlikely to happen frequently. Its something we will coordinate with their team.

Spearbit: Acknowledged.

5.4.7 Missing non-zero vkHash check in `claimWithZkProof`

Severity: Low Risk

Context: RouterRebates.sol#L116

Description: The contract currently allows calling `claimWithZkProof` even if `vkHash` is zero. Without verifying that `vkHash != 0`, the proof validation step may not check against a valid verification key hash.

Recommendation: Add a requirement in `claimWithZkProof` to ensure that `vkHash` is non-zero before proceeding with proof submission. For example:

```
require(vkHash != bytes32(0), "vkHash not set");
```

This enforces that an actual valid verification key hash is in place before any ZK-based claim is accepted.

Uniswap Labs: Fixed in PR 52.

Spearbit: Fix verified.

5.5 Informational

5.5.1 Chains with non-ether native currencies should not be supported

Severity: Informational

Context: ponder.config.ts#L34-L38

Description: The aggregator's "`baseFee` × `gasUsed`" approach assumes the cost is paid in Ether. If the chain's native token differs, the aggregator might incorrectly interpret "baseFeePerGas" as ETH-based. Users can pay minimal fees in a cheaper token yet claim a comparatively larger rebate in Ether, resulting in a profit.

Recommendation: Support only chains that use Ether as their native currency. Do not support chains like Polygon where the underlying native token (POL/Matic) is not Ether.

Uniswap Labs: Fixed in PR 55.

Spearbit: Fix verified.

5.5.2 RouterRebates contract does not implement any function to reclaim surplus ether

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `RouterRebates` contract accepts Ether to fund rebates. However, it has no function enabling the owner or anyone else to withdraw any unclaimed Ether. Consequently, if the contract still holds leftover Ether after all claims are complete, after the protocol was deprecated, or even if it's over-funded, those funds are permanently locked. This may be an intentional design choice, but can also be an oversight.

Recommendation: Consider implementing an `onlyOwner` function to transfer out leftover Ether.

Uniswap Labs: Fixed in PR 60.

Spearbit: Fix verified.

5.5.3 Missing nonReentrant modifier in claim functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `RouterRebates` contract, both `claimWithSignature` and `claimWithZkProof` update internal mappings like `lastBlockClaimed[chainId][beneficiary]` and then do a low-level call to transfer the rebate:

```
(bool sent,) = recipient.call{value: amount}("");
require(sent, "failed to send eth");
```

Although the code currently updates `lastBlockClaimed` first, and there is no obvious subsequent state change, there is still a best-practice concern: external calls can invoke reentrancy. An attacker's fallback function could re-enter the contract if another function has a vulnerability or modifies state incorrectly. Another possible vector is within the `claimWithSignature` function, in the signature verification:

```
signature.verify(_hashTypedDataV4(digest), signer);
```

if `signer.code.length > 0`, it calls:

```
IERC1271(claimedSigner).isValidSignature(hash, signature);
```

This is an external call to the signer contract which would temporary get the control of the flow and be able to reenter. Adding a `nonReentrant` guard is a standard measure to avoid unforeseen reentrancy complexities.

Recommendation: Consider adding a `nonReentrant` modifier on both `claimWithSignature` and `claimWith-ZkProof`. A good suggestion is to simply make use of the ReentrancyGuardTransient library from OpenZeppelin, which reduces gas costs while retaining the same reentrancy protection.

Uniswap Labs: Fixed in PR 58.

Spearbit: Fix verified.

5.5.4 Consider router specific nonces and signature deadlines

Severity: Informational

Context: ClaimableHash.sol#L6

Description: With no immediate reason to allow beneficiaries the ability to accumulate signed payloads, consider explicitly restricting the potential to collect more than one valid signed payload at a time by including nonce then validating against the nonce in the contract. Similarly, including a deadline would force those claiming to use the signed payload in a timely manner or to request a new signature if delayed for some reason.

Uniswap Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.5 Missing event emission in `setSigner` and `setZkConfig` functions

Severity: Informational

Context: RouterRebates.sol#L92-L100

Description: The `signer` is in charge of `claimWithSignature` validation. The `brvProof` and `vkHash` are in charge of `claimWithZkProof` validation. However, changing these state does not emit events.

Recommendation: Consider emit events in `setSigner` and `setZkConfig` functions.

Uniswap Labs: Fixed in PR 54.

Spearbit: Fix verified.

5.5.6 Have a dedicated function for topping up the rebate balance

Severity: Informational

Context: RouterRebates.sol#L162-L163

Description: The `receive()` and `fallback` functions are somewhat redundant and can be replace by a dedicated function to receive top ups.

Recommendation: Consider implementing a `topUp()` payable function to avoid unintentional top ups by either user or owner mistakes.

Uniswap Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.7 Lack of a double-step transfer ownership pattern

Severity: Informational

Context: RouterRebates.sol#L19

Description: The `RouterRebates` contract is using the standard `Solmate's Owned` library. The standard Solmate's Owned contract allows transferring the ownership of the contract in a single step:

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    owner = newOwner;

    emit OwnershipTransferred(msg.sender, newOwner);
}
```

If the nominated EOA account is not a valid account, it is entirely possible that the owner may accidentally transfer ownership to an uncontrolled account, losing the access to all functions with the `onlyOwner` modifier.

Recommendation: It is recommended to implement a two-step transfer process in the `RouterRebates` contract where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of the ownership to fully succeed. This ensures the nominated EOA account is a valid and active account. A good code example could be OpenZeppelin's Ownable2Step contract:

```
/**
 * @dev Starts the ownership transfer of the contract to a new account. Replaces the pending transfer if
 ↪  there is one.
 * Can only be called by the current owner.
 *
 * Setting `newOwner` to the zero address is allowed; this can be used to cancel an initiated ownership
 ↪  transfer.
 */
function transferOwnership(address newOwner) public virtual override onlyOwner {
    _pendingOwner = newOwner;
    emit OwnershipTransferStarted(owner(), newOwner);
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner`) and deletes any pending owner.
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual override {
    delete _pendingOwner;
    super._transferOwnership(newOwner);
}

/**
 * @dev The new owner accepts the ownership transfer.
 */
function acceptOwnership() public virtual {
    address sender = _msgSender();
    if (pendingOwner() != sender) {
        revert OwnableUnauthorizedAccount(sender);
    }
    _transferOwnership(sender);
}
```

Uniswap Labs: Acknowledged.

Spearbit: Acknowledged.

5.5.8 Unlicensed smart contract

Severity: Informational

Context: RouterRebates.sol#L1

Description: The `RouterRebates` contract is currently marked as unlicensed, as indicated by the SPDX license identifier at the top of the file:

```
// SPDX-License-Identifier: UNLICENSED
```

Using an unlicensed contract can lead to legal uncertainties and potential conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project and could lead to legal issues in the future.

Recommendation: It is recommended to choose and apply an appropriate open-source license to the smart contract. Some popular options for smart contract projects are:

1. MIT License: A permissive license that allows for reuse with minimal restrictions.
2. GNU General Public License (GPL): A copyleft license that ensures derivative works are also open-source.
3. Apache License 2.0: A permissive license that provides an express grant of patent rights from contributors to users.

Uniswap Labs: Fixed in PR 57.

Spearbit: Fix verified.

5.5.9 Unused imports

Severity: Informational

Context: RouterRebates.sol#L4, RouterRebates.sol#L9

Description: The `RouterRebates` contract imports `IERC20` and `IRebateClaimer` interfaces but does not use them anywhere in the code.

Recommendation: Consider removing the imports of the `IERC20` and `IRebateClaimer` interfaces in the `RouterRebates` contract.

Uniswap Labs: Fixed in PR 53.

Spearbit: Fix verified.

5.5.10 Database can have multiple pools with same ID

Severity: Informational

Context: rebate.ts#L66-L74

Description: The database can potentially contain duplicate pool ids if two pools with the same token addresses and hooks are deployed on different chains. This could lead to the wrong pool being fetched from the database. Fetching the pool from the wrong chain should not have any impact on the current functionality as the pools would be identical, however in the event this code needs to be extended in a manner that something needs to be stored in the database, it would end up being stored in the wrong item.

Recommendation: Consider adding a filter to get the pool from the correct chain id.

Uniswap Labs: Acknowledged.

Spearbit: Acknowledged.