OPTIMUM
BLOCKCHAIN SECURITY

Reflex Security Assessment

September, 2025

Contents

Disclaimer

Executive Summary

Overview

The security assessment was made by **one** researcher over a period of **2 full days**.

Project Name	Reflex
URL	
Code	https://github.com/reflex-mev/reflex
Commit Hash	`785662cd5fc3a671accc82129af7f89822f850c8`
Mitigations Commit Hash	`3ad2ec49a0f85a385fdc67ff2bc4c3e572f1ec71`
Language	Solidity

Contracts Assessed

Contract Name	Path
ReflexRouter.sol	core/src/ReflexRouter.sol
ReflexAfterSwap.sol	core/src/integrations/ReflexAfterSwap.sol

Classification of Issues

Severity	Description
Critical	Issues that may directly result in loss of funds, and thus require an urgent fix.
High	Issues that may not be directly exploitable, or with a limited impact, are still required to be fixed.
Medium	Issues that are not necessarily security vulnerabilities, that are required to be fixed unless there is a clear reason not to.
Low	Subjective issues with a negligible impact.
Info	Subjective issues or observations with negligible or no impact.

System Architecture Overview

The **Reflex System** is designed to extract and distribute post-swap arbitrage profits in a safe, modular, and non-intrusive manner. It is composed of three main parts:

- **ReflexAfterSwap**
- **ReflexRouter**
- **ReflexQuoter**

ReflexAfterSwap

- An **abstract integration contract** that DEXs or protocols can extend.
- Provides the `reflexAfterSwap()` entry point, which can be invoked **after a user swap has already completed**.
- Its role is purely to **signal the ReflexRouter** to attempt a backrun (arbitrage opportunity).
- **Critical Property**:
  - The system **never takes custody of user funds**.

  - It does not move, approve, or manage tokens belonging to users or the integrating protocol.
  - If the ReflexRouter fails during execution, the failure is silently caught (via `try/catch`), ensuring the original swap completes unaffected.

ReflexRouter

- The **execution engine** of the system.
- Handles arbitrage logic across **Uniswap V2, Uniswap V3, and compatible forks**, using flash-loan-style mechanisms.
- Integrates with **ConfigurableRevenueDistributor** to distribute profits to configured recipients.
- Implements strict **failsafe patterns**:
  - Graceful reentrancy protection.
  - Multi-route backrun execution with per-route isolation (a single failed backrun does not prevent others).
- The router only operates on its **own internal funds and temporary loaned liquidity**.

ReflexQuoter

- A **closed-source, read-only utility contract**.
- Provides quoting and route decoding functionality for the ReflexRouter.
- It has **no state and holds no funds**.
- Functions as a **pure logic library** to support profit estimation and swap path construction.

Security & Trust Model

- **Non-custodial**: Reflex does not hold or control user assets. Protocols integrating `reflexAfterSwap()` simply **notify** the system of swap deltas; the Reflex system operates independently on its own liquidity.

- **Failsafe by design**: Arbitrage attempts are sandboxed—if they fail, the user's transaction is unaffected.

- **Configurable distribution**: Profits are split and distributed according to configuration IDs managed by the Reflex admin.

Findings

Security Considerations for Integrators

Integrator contracts extending `ReflexAfterSwap` should **never transfer tokens/native assets** nor provide ERC20 approvals to any of the Reflex contracts. The expected integration pattern is simply invoking `ReflexAfterSwap.reflexAfterSwap()` once user operations have completed.

The primary theoretical risk is a **denial-of-service** if `reflexAfterSwap()` reverts. However, during the review it was verified that the function does not revert as long as it is called with correctly typed arguments. Integrators can further mitigate this risk by wrapping the call in a `try/catch` block.

The only residual risk lies in scenarios where:

- The Reflex contracts fail to capture arbitrage opportunities due to flawed internal algorithms, or

- The Reflex team acts maliciously, e.g., by redirecting seized profits to a team-controlled address under the claim that no profits were available for distribution.

1. Missing SafeERC20 Usage for Token Transfers

Severity: Low
Location: ReflexRouter.sol

Description: `ReflexRouter` uses `IERC20(token).transfer(...)` directly for ERC20 token transfers, e.g., in `_triggerBackrun()` and `withdrawToken()`. This does not handle tokens that do not return a boolean on `transfer()` or `transferFrom()`, which may cause unexpected failures with non-standard ERC20 implementations.

Recommendation:
- Use OpenZeppelin's SafeERC20 library and replace `.transfer()` with `.safeTransfer()`. - Similarly, update `.transferFrom()` calls to `.safeTransferFrom()` where applicable. This change improves compatibility with all ERC20 tokens and reduces the risk of failed transfers.

Resolution:
Fixed by implementing the reviewer recommendation.

2. Missing Events

Severity: Info
Location: ReflexAfterSwap.sol#L11, ReflexRouter.sol#L41, ConfigurableRevenueDistributor.sol#L82

Description: Both `ReflexAfterSwap` and `ReflexRouter`, as well as `ConfigurableRevenueDistributor`, lack event emissions in state-changing functions, such as:

- `setReflexRouter()`
- `setReflexConfigId()`
- `setReflexQuoter()`
- `updateDefaultConfig()`

Without events, it becomes difficult for off-chain services, monitoring tools, or users to track important changes like router updates, configuration changes, or default revenue configurations. This reduces transparency and makes debugging or auditing more challenging.

Recommendation:
Introduce events for critical state updates. For example:

```
event ReflexRouterUpdated(address oldRouter, address newRouter);
event ReflexConfigIdUpdated(bytes32 oldConfigId, bytes32 newConfigId);
event ReflexQuoterUpdated(address indexed oldQuoter, address indexed
 ↪  newQuoter);
event DefaultConfigUpdated(bytes32 oldConfigId, bytes32 newConfigId);
```

Resolution:
Partially fixed, `updateDefaultConfig()` is still missing an event emission.

3. ReflexAdmin Stored in Storage Instead of Fetched Dynamically

Severity: Info
Location: ReflexAfterSwap.sol#L57

Description:
`ReflexAfterSwap` stores the `reflexAdmin` address in contract storage at construction and updates it only when `setReflexRouter()` is called. This creates an implicit assumption that the `ReflexRouter`'s admin will never change. While `ReflexRouter` currently uses an immutable owner, this may not always be the case in future upgrades. If the router changes its ownership model, the stored `reflexAdmin` will become outdated, leading to authorization inconsistencies.

Additionally, storing `reflexAdmin` in state increases gas usage unnecessarily, as it could be derived on demand.

Recommendation:
Instead of storing `reflexAdmin`, fetch it dynamically when needed:

```
function getReflexAdmin() public view returns (address) {
    return IReflexRouter(reflexRouter).getReflexAdmin();
}
```

This ensures correctness even if ReflexRouter's ownership model changes, while also reducing gas costs by removing redundant storage writes.

Resolution:
Fixed by removing the `reflexAdmin` storage variable.

4. Inconsistent Usage of Underscore in Function Naming

Severity: Info
Location: ReflexRouter.sol

Description:
There is inconsistent usage of leading underscores (_) in function names:

- `_triggerBackrunSafe()` is declared as **`external`**, yet its name uses a leading underscore, which typically indicates an **internal/private** function.

- Other functions such as `triggerSwapRoute()`, `handleLoanCallback()`, `swapFlow()`, `swapUniswapV3Pool()`, `decodeUniswapV3LikeCallbackParams()`, `decodeUniswapV2LikeCallbackParams()`, and `bytesToAddress()` are marked as **`internal`** but lack a leading underscore.

This inconsistency makes it unclear which functions are part of the public API and which are intended as internal helpers. It also reduces readability and may confuse auditors and integrators.

Recommendation:
Adopt a consistent naming convention:
- Remove the underscore for `external` or `public` functions (e.g., rename `_triggerBackrunSafe()` → `triggerBackrunSafe()`).
- Add a leading underscore for `internal` or `private` functions (e.g., rename `triggerSwapRoute()` → `_triggerSwapRoute()`, `bytesToAddress()` → `_bytesToAddress()`).

Following a consistent style will improve readability, better communicate function intent, and align with Solidity best practices.

Resolution:
Fixed by implementing the reviewer recommendation.

5. Misleading Modifier Name: `isAdmin`

Severity: Info
Location:
ReflexRouter.sol#L64

Description:
The `ReflexRouter` contract uses a modifier named `isAdmin` to restrict access to the contract owner:

```
modifier isAdmin() {
    require(msg.sender == owner);
    _;
}
```

The name isAdmin is misleading because it implies a generic admin role, while in reality it enforces owner-only access. This can confuse auditors, integrators, and developers about the intended access control semantics.

Recommendation:
Rename the isAdmin modifier to isOwner to accurately reflect its purpose. Update all functions using isAdmin to use the new isOwner modifier. Optionally, update documentation and comments to match the new naming.

Resolution:
Fixed by implementing the reviewer recommendation.

6. Code Duplication in `_setShares()` and `_setDefaultShares()`

Severity: Info
Location: ConfigurableRevenueDistributor.sol

Description:
The contract implements `_setShares()` and `_setDefaultShares()` functions that share very similar logic for configuring revenue shares. This results in duplicated code, making the contract harder to maintain and increasing the risk of inconsistent behavior if one function is updated without updating the other.

Recommendation:
- Generalize the logic into a single internal helper function that both `_setShares()` and `_setDefaultShares()` can call.
- Keep the two external/internal functions as wrappers if needed for clarity, but delegate the shared logic to the helper.

This approach reduces code duplication, improves maintainability, and lowers the likelihood of errors in future updates.

Resolution:
Fixed by removing `_setDefaultShares()`.

7. Discouraged Use of Reentrancy Guards in Internal Functions

Severity: Info
Location: ReflexRouter.sol,ReflexAfterSwap.sol

Description:
The contracts currently apply the `gracefulNonReentrant` modifier to internal functions, such as `_triggerBackrunSafe()` and `reflexAfterSwap()`.

Using reentrancy guards on internal functions is discouraged because:
- Internal functions are not directly callable from external actors, so reentrancy risk is generally mitigated by the public/external entry point.
- Applying the guard internally can unnecessarily restrict function reuse and complicate call flows.

Recommendation:
- Move `gracefulNonReentrant` to external/public functions that serve as entry points (e.g., `triggerBackrun()`, `backrunedExecute()`).
- Internal functions should assume that proper reentrancy protection is enforced at the external boundary.

This improves flexibility, avoids redundant guards, and follows standard Solidity best practices.

Resolution:
Fixed by implementing the reviewer recommendation.

8. Rename `dustShareBps` and `dustRecipient` for Clarity

Severity: Info
Location: ConfigurableRevenueDistributor.sol

Description:
The contract currently uses `dustShareBps` and a function parameter named `dustRecipient` in revenue splitting functions.

- The term `dust` implies leftover or insignificant tokens, but in practice this address simply receives a configurable share of funds.
- This naming can confuse auditors, integrators, and users about the purpose of the parameter.

Recommendation:
- Rename `dustShareBps` to something more descriptive, e.g., `recipientShareBps` or `miscShareBps`.
- Rename the `dustRecipient` parameter to reflect its actual role, e.g., `shareRecipient` or `fundRecipient`.
- Update all related functions, events, and documentation to reflect the new naming.

This improves clarity and reduces potential misinterpretation of the revenue distribution logic.

Resolution:
Fixed by implementing the reviewer recommendation.

9. `decodeIsZeroForOne()` Type Mismatch

Severity: Info
Location: ReflexRouter.sol#L481

Description:
The function `decodeIsZeroForOne()` is documented to decode a single byte of metadata, but its parameter type is currently `uint256`.

- This does not match the intended 1-byte usage described in the comments.

- Using `uint256` may confuse auditors or integrators and leads to unnecessary type widening in the assembly implementation.

Recommendation:
- Change the parameter type from `uint256` to `uint8` to match the function's description and intended usage.
- Ensure the assembly logic continues to correctly extract the `zeroForOne` flag.

This improves readability, correctness, and consistency between documentation and implementation.

Resolution:
Fixed by implementing the reviewer recommendation.

10. Avoid Using `tx.origin` for Access Control

Severity: Info
Location: ReflexRouter.sol#L57, ConfigurableRevenueDistributor.sol#L33

Description:
`ReflexRouter` currently sets the owner using `tx.origin` in the constructor, and `ConfigurableRevenueDistributor` may rely on similar patterns.

- Using `tx.origin` for access control is discouraged because it can be exploited in phishing-style attacks via intermediary contracts.

- `msg.sender` is safer and aligns with standard Solidity practices for authorization.

Recommendation:
- Replace `tx.origin` with `msg.sender` for setting ownership and access control, unless there is a specific reason that requires `tx.origin`.
- Audit related contracts to ensure access control remains secure after the change.

This improves security and prevents potential attacks that exploit `tx.origin`.

Resolution:
Fixed by implementing the reviewer recommendation.

11. Include `quoteProfit` in `BackrunExecuted` Event

Severity: Info
Location: ReflexRouter.sol#L222

Description:
The `BackrunExecuted` event currently emits `triggerPoolId`, `swapAmountIn`, `token0In`, `profit`, `profitToken`, and `recipient`, but it does **not include the value of `quoteProfit`** obtained from `ReflexQuoter.getQuote`.

- Including `quoteProfit` would provide visibility into the expected profit from the quote versus the actual profit realized after executing the backrun.
- This is useful for monitoring, analytics, and debugging arbitrage performance.

Recommendation:
- Update the `BackrunExecuted` event to include `quoteProfit` as an additional parameter.
- Modify the event emission in `_triggerBackrun()` to pass the obtained `quoteProfit`.
- Update any listeners or off-chain monitoring systems to handle the new parameter.

Adding `quoteProfit` improves observability and allows better assessment of arbitrage efficiency.

Resolution:
Fixed by implementing the reviewer recommendation.

Security Best Practices Reference

This report references security practices and guidelines from the **Optimum Blockchain Security Guide**. The repository provides a comprehensive and continuously updated collection of best practices for securing smart contracts. Readers are encouraged to review the guide for additional context, rationale, and the latest updates on secure development standards.