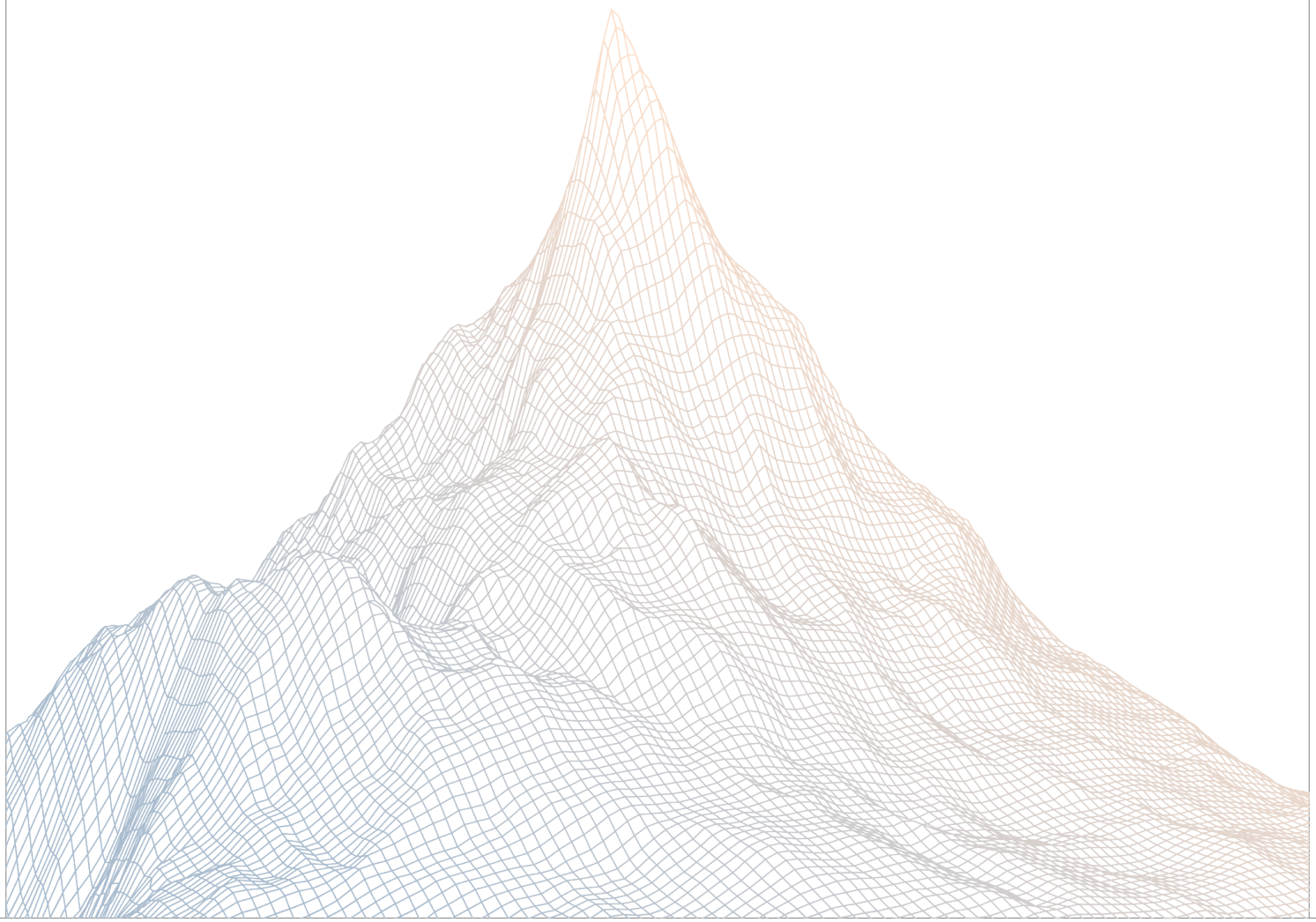
Zenith

Syrax

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 20th to November 21st, 2025

AUDITED BY: montecristo
oakcobalt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 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

2

Executive Summary

2.1 About Syrax

Syrax is an independent, self-funded trading platform built by a team of experienced builders. The platform is designed to support long-term trading, aiming to solve the challenge of trading on all chains.

2.2 Scope

The engagement involved a review of the following targets:

Target	evm-router
Repository	https://github.com/Syrax-AI/evm-router
Commit Hash	`c43f9e723ca69a1cdb73449ddd2acf46b9faace8`
Files	`src/SwapRouter.sol`

2.3 Audit Timeline

November 20, 2025	Audit start
November 20, 2025	Audit end
November 24, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	1
Informational	1
Total Issues	**6**

3

Findings Summary

ID	Description	Status
M-1	Compromised signer can drain user funds breaking security guarantee	Resolved
M-2	Potential fee theft via Reentrancy	Resolved
M-3	User fund may be stuck during an exact-output swap	Resolved
M-4	Sell fee is taken from input instead of output	Resolved
L-1	EIP-712 Type Hash Mismatch with Struct Definition	Resolved
I-1	Signature replay attack breaks one-time use guarantee	Acknowledged

4

Findings

4.1 Medium Risk

A total of 4 medium risk findings were identified.

[M-1] Compromised signer can drain user funds breaking security guarantee

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapRouter.sol#L247

Description:

According to Doc, the current security guarantee is that `if the admin is compromised, there still will be no user funds at risk`. However, this assumption doesn't hold. A compromised signer can drain user funds directly.

The SwapRouter contract allows `targetContract` to be set to any address except itself. We suppose: (1) Set `tokenIn = address(0)` and `amountIn = 0` to bypass payment requirements (2) Set `targetContract = tokenOut = address(tokenX)` where `tokenX` is a token the user has approved to SwapRouter (3) Set `params.data` to call `transferFrom(user, address(this), stealAmount)` directly on the token contract

Once signed by a compromised signer, `buy()` function:

- Passes the `msg.value == params.amountIn == 0` check.
- Calls `targetContract.call(params.data)` which executes `tokenX.transferFrom(user, address(this), amount)`
- The token contract checks `allowance[user][SwapRouter]` which passes because the user approved SwapRouter
- Tokens are transferred from user to SwapRouter
- The balance check passes because SwapRouter received tokens.
- SwapRouter automatically sends all `tokenOut` to `msg.sender`

Recommendations:

Consider implementing multiple layers of protection. For example: (1) prevent token contracts as targets (e.g. ‘if (params.targetContract == params.tokenIn || params.targetContract == params.tokenOut) {//revert‘) (2) Block dangerous function selectors

Syrax: Resolved with `@950a13ba8a ...`.

Zenith: Verified. Whitelisted selector checks are implemented that prevent user loss of funds.

[M-2] Potential fee theft via Reentrancy

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- SwapRouter.sol#L286

Description:

When a reentrant call to `buy()` or `sell()` uses the same token as both `tokenIn` (in the reentrant call) and `tokenOut` (in the original call), the fee paid in the reentrant call is incorrectly included in the original call's balance delta calculation.

The SwapRouter contract uses a balance measurement pattern to determine how many tokens were received from a swap:

```
uint256 balanceBefore = IERC20(params.tokenOut).balanceOf(address(this));

(success, ) = params.targetContract.call{value: amountToSwap}(params.data);
...
uint256 balanceAfter = IERC20(params.tokenOut).balanceOf(address(this));
uint256 amountReceived = balanceAfter - balanceBefore;

IERC20(params.tokenOut).safeTransfer(msg.sender, amountReceived);
```

Vulnerabilities: If the external call to `targetContract` triggers a reentrant call to SwapRouter (e.g., through a UniswapV4 hook or other custom hooks), and the reentrant call uses the same token as both `tokenIn` (reentrant) and `tokenOut` (original), the fee left in the router from the reentrant call gets included in the original call's balance delta.

Attack Flow:

1. Alice initiates a swap: `tokenX → tokenY` with valid signature
2. During the external call, a hook calls back to Alice's contract B
3. Contract B (with valid signature) reenters SwapRouter: `tokenY → tokenZ`
4. Contract B's swap leaves a fee in `tokenY` (e.g., 5 tokens)
5. Alice's swap completes, sending `tokenY` to router (e.g., 980 tokens)

6. Router measures: `balanceAfter = 5 (B's fee) + 980 (Alice's swap) = 985`
7. Router sends 985 `tokenY` to Alice, but Alice should only receive 980
8. Result: Alice gains 5 `tokenY` that should have been B's fee, essentially avoiding her own fee

Impacts: High-Medium: Fee theft. Repeated exploit is possible to escape fees.

Likelihoods: Low-Medium: Depending on custom hook calldata.

Recommendations:

Consider adding reentrancy guard for both `buy` and `sell` methods.

Syrax: Resolved with `@950a13ba8a ...`.

Zenith: Verified.

[M-3] User fund may be stuck during an exact-output swap

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- SwapRouter.sol

Description:

Some DEX'es have exact-output-swap feature, where `amountOut` is known but `amountIn` is not determined.

For example, Uniswap V3 Router has `exactOutput` function:

```
function exactOutput(
  struct ISwapRouter.ExactOutputParams params
) external returns (uint256 amountIn)
```

where:

```
struct ExactOutputParams {
    bytes path;
    address recipient;
    uint256 deadline;
    uint256 amountOut;
    uint256 amountInMaximum;
}
```

If the user wants to use `exactOutput` swap, they'll have to approve and transfer `amountInMaximum` tokens to Syrax router.

For example, let's assume that a user wants to exactly buy `100e6` USDC with some WETH. `BuyParams` will look like the following:

- `targetContract`: Uniswap V3 Router address
- `tokenIn`: WETH address
- `tokenOut`: USDC address
- `amountIn`: `0.04e18`

- `minAmountOut: 100e6`
- `data`: ‘abi.encode(IUniswapV3Router.exactOutput.selector, params)
  - `params.amountOut: 100e6`
  - `params.amountInMaximum: 0.04e18`

Syrax router will approve `0.04e18` WETH to Uniswap Router. However, let's assume Uniswap Router spends only `0.03e18` WETH and outputs `100e6` USDC.

In this case, `0.01e18` WETH will be stuck at Syrax router.

This is a violation to one of main invariants:

> Balance Accounting: Contract balance always equals accumulated fees only - no user tokens stuck

Recommendations:

The correct fix would be to refund any unused `tokenIn` to `msg.sender`. The amount to refund is:

```
balanceBeforeSwap - feeAmount - balanceAfterSwap
```

Syrax: Resolved with `@950a13ba8a ...`. Just to note though with our current logic absolutely everything is based on fixed amounts in and variable amounts out.

Zenith: Verified. One note: Users using buy() for exact output swaps may pay fees on unused input tokens that are refunded, potentially resulting in higher router fees compared to sell() for exact output swaps. This could be documented for transparency.

[M-4] Sell fee is taken from input instead of output

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

SwapRouter::sell

Description:

According to README, sell fee should be collected from `tokenOut`:

> **SELL**: Sell tokenIn (ERC20) → receive tokenOut (ERC20 or ETH). Fee taken from output.

This is also confirmed in Main Invariants section:

> Fee Math: Fee calculations never overflow and follow:
>
> Buy: fee from input → amountToSwap = amountIn - fee Sell: fee from output → amountToUser = amountReceived - fee

However, current implementation collect fees from `tokenIn`:

File: src/SwapRouter.sol

```
301:         // Calculate fee upfront
302:         uint256 feeAmount = (params.amountIn * params.feeBps) / 10000;
303:         uint256 totalToTake = params.amountIn + feeAmount;
```

This creates a discrepancy between the documented behavior and actual implementation.

Recommendations:

When selling a token, take fees from `tokenOut`, instead of `tokenIn`.

Syrax: Resolved with `@950a13ba8a ...`.

Zenith: Verified

4.2 Low Risk

A total of 1 low risk findings were identified.

[L-1] EIP-712 Type Hash Mismatch with Struct Definition

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- SwapRouter.sol#L29
- SwapRouter.sol#L34

Description:

The `BUY_TYPEHASH` and `SELL_TYPEHASH` constants do not match their struct definitions. The type hash includes `address user` which is missing from the struct, while the struct includes `bytes signature` which is not in the type hash. This violates EIP-712 compliance where type hashes must exactly match struct definitions.

```
bytes32 public constant BUY_TYPEHASH =
    keccak256("BuyParams(address user,uint256 deadline,address
    targetContract,address tokenIn,address tokenOut,uint256 amountIn,uint256
    minAmountOut,bytes data,uint16 feeBps)");
...
struct BuyParams {
    uint256 deadline;  //@audit Missing address user
...
    bytes signature;   //@audit Not in type hash
}
```

The contract currently works around this by passing `user` as `msg.sender` separately and excluding `signature` from the hash computation. However, this violates EIP-712 compliance and creates ambiguity about what data is being signed.

Recommendations:

Refactor to pass `signature` as a separate parameter and add `user` field to struct, ensuring type hash exactly matches struct definition. For example:

```
struct BuyParams {
    address user;
    uint256 deadline;
    address targetContract;
    address tokenIn;
    address tokenOut;
    uint256 amountIn;
    uint256 minAmountOut;
    bytes data;
    uint16 feeBps;
}
...
function buy(
    BuyParams calldata params,
    bytes calldata signature
) external payable whenNotPaused {
    if (params.user ≠ msg.sender) revert InvalidUser();
    _verifyBuySignature(params, signature);
...
}
```

Syrax: Resolved with @950a13ba8a ... .

Zenith: Verified.

4.3 Informational

A total of 1 informational findings were identified.

[I-1] Signature replay attack breaks one-time use guarantee

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- SwapRouter.sol#L374-L385

Description:

The contract lacks on-chain protection against signature replay, allowing users to execute the same signature multiple times until the deadline expires. This breaks the security guarantee of one-time signature use and also allows user abuse of the reward programs.

The `buy()` and `sell()` functions verify signatures and check deadlines but do not track whether a signature has already been used.

```
function buy(BuyParams calldata params) external payable whenNotPaused {
    _verifyBuySignature(params, msg.sender);  // @audit Validates signature
    with no nonce protection
    if (block.timestamp > params.deadline) revert DeadlineExpired();  //
    @audit only checks deadline
...
}

    function _verifyBuySignature(BuyParams calldata params, address user)
    internal view {
        bytes32 structHash = keccak256(
            abi.encode(
                BUY_TYPEHASH,
                user,
                params.deadline,
                params.targetContract,
                params.tokenIn,
                params.tokenOut,
```

```
                params.amountIn,
                params.minAmountOut,
                keccak256(params.data),
                params.feeBps
            )
        );
```

Attack paths:

1. Backend signs a signature for User A with `feeBps=50` (0.5% discounted fee) for a reward program swap, intended for one-time use
2. User A calls `buy(params)` with the signature - transaction succeeds, fee of 0.5% is charged
3. User A immediately calls `buy(params)` again with the same parameters and signature before `deadline` expires
4. Signature verification passes (same valid signature), deadline check passes (not expired), tx succeeds again
5. User A receives the discounted 0.5% fee again instead of the normal 1% fee
6. User A can repeat this multiple times until `deadline` expires.

Impacts: (1) This breaks the security guarantee of one-time signature use. (2) It potentially allows economic exploits through signature reuse.

Recommendations:

Consider adding on-chain per-user nonce tracking and including nonce in buy/sell message struct. For example:

```
mapping(address ⇒ uint256) public userNonces;

bytes32 public constant BUY_TYPEHASH =
    keccak256(
        "BuyParams(address user,uint256 nonce,...)"
    );

function buy(BuyParams calldata params) external payable whenNotPaused {
    uint256 currentNonce = userNonces[msg.sender];
    _verifyBuySignature(params, msg.sender, currentNonce);
    userNonces[msg.sender]++;  // Increment after verification
    ...
}
```

Syrax: Acknowledged. Due to having to manage on-chain nonces for rapid trading, which can have multiple in a block, having a second layer of nonces will make it even more difficult to manage for negligible gain.

Zenith: Acknowledged. The reward program abuse risk (users replaying discounted fee signatures) remains, but this is an accepted trade-off given the mitigations (short deadlines, backend tracking, MAX_FEE_BPS cap)