SMART CONTRACTS REVIEW

zokyo

September 5th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING LAB

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Scoring Calculation:
Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 1 Critical issue: 1 resolved = 0 points deducted
- 4 High issues: 2 resolved and 2 acknowledged = - 6 points deducted
- 7 Medium issues: 4 resolved and 3 acknowledged = - 3 points deducted
- 8 Low issues: 5 resolved and 3 acknowledged = 0 points deducted
- 2 Informational issues: 2 acknowledged = 0 points deducted

Thus, 100 - 6 - 3 = 91

TECHNICAL SUMMARY

This document outlines the overall security of the LAB smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the LAB smart contract/s codebase for quality, security, and correctness.

Contract Status

There was 1 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the LAB team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the LAB repository:
Repo: https://github.com/labpro-codebase/Lab_EVM

Last commit - 745c354467e05ddd1cc29e0e52b232b6b6646273

Contracts under the scope:

- contracts/LabDexRouterOptimized.sol
- contracts/extensions/FeeExtension.sol
- contracts/libs/CustomBytesLib.sol
- contracts/dexes/uniswapV3/v3-periphery/interfaces/ISwapRouter.sol
- contracts/dexes/uniswapV3/v3-periphery/interfaces/IQuoterV2.sol
- contracts/dexes/uniswapV3/v3-periphery/interfaces/IAerodromeQuoter.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of LAB smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

The contracts in this project namely **LAbDexRouterOptimized.sol** provide multi-protocol swap functionality with integrated fee collection mechanisms. The core component, **LabDexRouterOptimized**, is a comprehensive smart contract that supports token swaps across multiple DEX protocols including Uniswap V2/V3, Aerodrome, and PancakeSwap. It implements sophisticated routing capabilities that handle ETH-to-token and token-to-ETH swaps while supporting fee-on-transfer tokens with functions like **swapExactETHForTokensSupportingFeeOnTransferTokens** and **swapExactTokensForETHSupportingFeeOnTransferTokens**. The function **exactInputSingle()** allows interaction with uniswap v3 based protocol.

The **FeeExtension.sol** contract which is inherited by the **LabDexRouterOptimized.sol** contract helps calculate fees via functions like **calcFeeAndNewAmount()** while the vault set via **_setVault()** during the deployment of the contract in constructor collects the fee from swaps. The **withdraw()** function in the **LabDexRouterOptimized** allows the admin of the contract to withdraw any stuck tokens in the contract.

The **CustomBytesLib.sol** contract is based on Uniswap v3's BytesLib which is a critical library for manipulating bytes arrays, allowing the protocol to concatenate, slice and type cast bytes arrays both in memory and storage.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the LAB team and the LAB team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	The user might not receive swapped tokens using the method exactInputSingle()	Critical	Resolved
2	Missing verification of valid pool existence can lead to malicious calls to uniswapv3callback()	High	Acknowledged
3	It is not guaranteed that the first two tokens in the path include WETH, leading to potential permanent fund loss	High	Resolved
4	Missing validation for path addresses, poolAddress, and quoter	High	Acknowledged
5	Missing deadline check for the method exactInputSingle()	High	Resolved
6	Protocol will unexpectedly earn 0 fees	Medium	Acknowledged
7	Stuck WETH can be stolen using the method exactInputSingle() and v2 swap methods (tokens to eth)	Medium	Acknowledged
8	The restriction of using one single pool swap can be bypassed	Medium	Resolved
9	Incorrect _feeBps can lead to failed transactions or potential loss of swapped token output	Medium	Acknowledged
10	Unused modifier or incorrect comment	Medium	Resolved
11	Address `to` can be address(0) leading to ETH sent to address(0) in swap methods	Medium	Resolved
12	Swap methods do not check the order of tokens	Medium	Resolved
13	ETH Amount Mismatch	Low	Resolved

#	Title	Risk	Status
14	Reliance on balanceOf(address(this)) Instead of Operation Return Values	Low	Acknowledged
15	Reliance on On-Chain Reserve Calculations	Low	Acknowledged
16	Using not locked solidity versions can lead to unexpected behaviours	Low	Resolved
17	Missing Deadline / Transaction Expiration Check	Low	Resolved
18	CEI pattern not followed leading to a possible reentrancy attack	Low	Acknowledged
19	Method exactinputSingle() can have different params.amountIn and msg.value	Low	Resolved
20	Missing check for single pool swap and pair addresses length	Low	Resolved
21	The constructor is not setting feeNumerator	Informational	Acknowledged
22	Deprecated checks	Informational	Acknowledged

The user might not receive swapped tokens using the method exactInputSingle()

In Contract LabDexRouterOptimized.sol, the method exactInputSingle(...) allows users to swap WETH to an ERC20 token. It allows users to pass the parameters as follows: `ISwapRouter.ExactInputSingleParams` calldata params

Here, params has many fields, and one of those fields is recipient. Since the user passes it, it can be set to address(0), and since there is no check if this is address(0), it will go unnoticed until the exactInputInternal method is called. Now, in the method exactInputInternal(), there is this logic:

```
if (recipient == address(0)) recipient = address(this);
```

Let's say params.recipient is address(0), so now it will be set to address(this). If a user is trying to swap WETH to ERC20, the LabDexRouterOptimized contract will receive the swapped ERC20 tokens, and there is no logic to send these tokens back to the users.

Impact:
Users WETH will be sent to the pool, but received tokens will be in the LabDexRouterOptimized contract, leading to users losing funds directly after the swap. Further, these stuck funds can be stolen as well, as described in the next findings, so these funds become non-recoverable for the user and the protocol.

Recommendation:

Ensure that when a user is swapping from WETH to an ERC20 token, the recipient address is always set and not address(0).

Missing verification of valid pool existence can lead to malicious calls to uniswapv3callback()

Description:
The function uniswapv3callback() does not verify if the pool is a valid uniswapV3 pool or not. Ideally the router verifies using the factory, tokenIn, tokenOut and fee parameters to check whether the uniswap V3 pools is a valid pool or not. This is done by computing the address of the pool based on the fact that the Uniswap v3 pools are deployed using CREATE2, making their addresses predictable based on factory + token pair + fee deterministically. This ensure that the callback is actually coming from a legitimate Uniswap pool contract, not from a malicious contract pretending to be one. Without this verification, an attacker could call your callback function directly and potentially exploit your contract.

```
  function uniswapV3SwapCallback(int256 amount0Delta, int256
amount1Delta, bytes calldata _data) external onlyWhenLocked {
       require(amount0Delta > 0 || amount1Delta > 0); // swaps entirely
within 0-liquidity regions are not supported
       SwapCallbackData memory data = abi.decode(_data,
(SwapCallbackData));
       (address tokenIn, address tokenOut,) = decodeFirstPool(data.path);
...
}
```

In addition to that, in Contract LabDexRouterOptimized.sol, the methods swapExactTokensForETHSupportingFeeOnTransferTokens(), swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(), and exactInputSingle() all allow a user to swap ERC20 tokens to ETH. After the swap is done, the LabDexRouterOptimized contract sends ETH to the users as follows:

```
function sendValue(address payable recipient, uint256 amount) internal {
       require(address(this).balance >= amount, "Address: insufficient
balance");


       (bool success, ) = recipient.call{value: amount}("");
       require(success, "Address: unable to send value, recipient may
have reverted");
```

Here, a low-level call is made to the recipient, which can be a contract with logic to re-enter the calling contract.

Now, let's say the recipient is a contract with the receive() method that calls back the method uniswapV3SwapCallback(...) or the method pancakeV3SwapCallback(...) with forged params, as long as when these params are decoded, they provide the tokenIn, tokenOut, amount0Delta, amount1Delta to be the same as the stuck ERC20 tokens.

Since these methods have a modifier check `onlyWhenLocked`, the contract will check if it meets the requirement or not, and it does meet the requirement since the above-mentioned methods (swapExactTokensForETHSupportingFeeOnTransferTokens(), swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(), and exactInputSingle()) will set the lock to be true.

Also, these methods (uniswapV3SwapCallback(...) and pancakeV3SwapCallback(...)) do not check that the msg.sender is the uniswapV3 pool; any contract will be able to do this reentrancy attack and steal all stuck ERC20 tokens.

Moreover, the functions uniswapV3SwapCallback and pancakeV3SwapCallback do not validate that msg.sender is the expected V3 pool address. This allows any external account to call these callback functions and potentially trigger unauthorized token transfers via the internal pay logic.

Impact:
This could lead to a malicious contract posing as a pool to be accidentally or maliciously used as a pool address and directly calling the uniswapV3SwapCallback() function.

Another possible impact, as mentioned in another issue, the user will lose funds when they swap WETH to ERC20, settingthe recipient as address(0), but that was still recoverable by admin, but now these funds are non-recoverable since they can be stolen by the attacker.

An attacker could call the callback functions directly, bypassing the intended swap mechanism, and potentially steal tokens or manipulate the contract's balances. This represents a critical security risk for contracts interacting with V3 liquidity pools.

Recommendation:

It is recommended that the pools are verified they are valid uniswap v3 (or aerodrome) pools by deterministically computing that the pool is indeed created from the factory of the protocol.

```
      (address tokenIn, address tokenOut, uint24 fee) =
data.path.decodeFirstPool();
       CallbackValidation.verifyCallback(factory, tokenIn, tokenOut,
fee);
```

Refer- https://github.com/Uniswap/v3-periphery/blob/main/contracts/SwapRouter.sol#L65C43-L65C50

Add a check in the methods (uniswapV3SwapCallback(...) and pancakeV3SwapCallback(...)) that the callback msg.sender is the Uniswap V3 pool only. Also, use different lock variables for the v2 and v3 swap methods.

Add a validation step to ensure msg.sender is the legitimate pool address for the given token pair.

Client comments:

The client said that they did not validate pools because it increased gas usage and reduced the flexibility of the contract. They added that this behaviour was implemented to allow swaps across different Uniswap V3 forks (e.g., PancakeSwap V3) and other protocols that implement the same interface. They acknowledged that it was true that funds could be stolen (in the above scenario), but the contract is not intended to hold (any) funds.

Auditor's comments:

Any user can call this function with their own chosen parameters. If no verification checks are performed, this could allow arbitrary executions, including malicious ones. While the client may rely on their backend to provide valid parameters, risks remain, as the function can still be called externally. Moreover, although the contract is not supposed to hold tokens, it potentially could, and users might be directly affected even without the contract holding funds. The client seems to understand the associated risk, nonetheless. This issue was combined with 3 separated reported issues as requested by the client.

It is not guaranteed that the first two tokens in the path include WETH, leading to potential permanent fund loss

Description:
Some of the main functions within LabDexRouterOptimized.sol implement the onlyPathWithWeth modifier, which is intended to ensure that one of the first two tokens in the swap path is WETH. However, the current implementation uses:

```
onlyPathWithWeth(path[0], path[path.length - 1])
```

This only verifies that WETH is present at the first or last element of the path, rather than among the first two tokens as required. This flaw becomes critical because the router always wraps the ETH input into WETH and transfers it to pairAddresses[0], assuming the first hop consumes WETH.

If the first hop in the path does not actually involve WETH, the pair contract will not recognize the deposited WETH as valid input. As a result, the computed amountInput will be zero, the swap will output nothing, and the WETH sent to the pair will remain locked — effectively causing permanent loss of user funds.

Note that the described issue covers the case where WETH is required to be one of the first two tokens of the path. However, this modifier has been used for some different behaviors, like WETH being part of the last two tokens of the path. Please check every scenario to ensure that the modifier is actually executed the wanted behavior:

```
- `onlyPathWithWeth(path[0], path[1])`: Requires WETH in the first two tokens of the path.
- `onlyPathWithWeth(path[0], path[1])`: Requires WETH in the last two tokens of the path.
- `onlyPathWithWeth(path[0], path[1])`: Ensures the first two tokens in the path include WETH.
- `onlyPathWithWeth(path[0], path[1])`: Ensures the path includes WETH as required.
- `onlyPathWithWeth(params.tokenIn, params.tokenOut)`: Ensures one of the tokens in the swap is WETH.
```

Impact:
Users can lose their ETH/WETH irreversibly if they supply a swap path that passes the faulty onlyPathWithWeth check but does not actually have WETH in the first two positions.

The onlyPathWithWeth modifier only works as intended if combined with onlyOnePoolSwap, which restricts the path to exactly two tokens. Without this, multi-hop paths can bypass the intended invariant.

Recommendation:
Update the modifier to validate the first two tokens explicitly:

```
onlyPathWithWeth(path[0], path[1])
```

If only single-hop swaps are intended, ensure that onlyOnePoolSwap(path) is always applied alongside onlyPathWithWeth.

Review all functions using this modifier to guarantee consistency with the intended WETH-first-hop invariant.

Missing validation for path addresses, poolAddress, and quoter

In Contract LabDexRouterOptimized, the following methods have path[] and pairAddresses[] for enabling the swap between WETH and ERC20 tokens. These methods do not check if the pairAddress provided corresponds to the path tokens.

```
function swapExactETHForTokensSupportingFeeOnTransferTokens(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokens(..) {..}
function swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome(..)
{..}
function swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(..)
{..}
These methods do not check if the pairAddress provided corresponds to the
path tokens.
For eg: pairAddresses[0] should have token0 == path[0] and token1 ==
path[1].
```

Similarly, for the method exactInputSingle(...), the user can pass the parameter `poolAddress`, which can be any value. Ideally, poolAddress should be derived using the following logic:

```
function getPool(
        address tokenA,
        address tokenB,
        uint24 fee
    ) private view returns (IUniswapV3Pool) {
        return IUniswapV3Pool(PoolAddress.computeAddress(factory,
PoolAddress.getPoolKey(tokenA, tokenB, fee)));
    }
```

Similarly, for the method quoteExactInputSingle(..) and quoteExactInputSingleAerodrome(..), the user can pass the parameter `quoter`, which can be any value. The user can pass this value wrong. Also, a malicious actor could provide a contract address in place of quoter that returns manipulated values or consumes excessive gas as the function does not restrict which _quoter can be used.

Additionally, for getAmountOut methods, the function accepts a poolAddress as a user-supplied parameter and calls IAerodromePool(poolAddress).getAmountOut(...). A malicious user could supply a contract that implements getAmountOut in an arbitrary way, returning manipulated values. This can lead to incorrect fee calculations, token swaps, or other downstream logic relying on this value.

Impact:
Users can permanently lose their ETH/WETH or tokens by routing swaps through malicious pair addresses.

Attackers can manipulate getAmountOut to misrepresent expected output, bypass slippage protections, and drain liquidity.

All functions accepting user-supplied pairAddresses are affected.

This can also lead to DoS attack for users since transactions will keep reverting until the correct pair/pool address is set for the path tokens.

Recommendation:
Add logic to get pair using uniswapV2Factory.getPair(..) method using the path tokens instead of getting it as parameters, now that only 1 pool is allowed.

For Uniswap v3, use the above-mentioned method.

For the quoter method, set those values in the constructor and allow the admin to update them if needed.

Review all functions in the codebase where users supply external addresses and ensure similar vulnerabilities are mitigated.

Client comments:
The client said that the router was designed to support swaps across different Uniswap protocol forks. They added that directly binding to a factory increased gas costs and required the deployment of a new contract for each DEX.

Auditor's comments:
Any user can call this function with their own chosen parameters. If no verification checks are performed, this could allow arbitrary executions, including malicious ones. While the client may rely on their backend to provide valid parameters, risks remain, as the function can still be called externally. Moreover, although the contract is not supposed to hold tokens, it potentially could, and users might be directly affected even without the contract holding funds. The client understands the associated risk.

Missing deadline check for the method exactInputSingle()

In Contract LabDexRouterOptimized.sol, the method exactInputSingle(...) has input params which has a field `deadline` and the contract should be checking deadline is not expired, else revert. Now that it deadline is not checked, the **impact on the timing of execution** can be huge.

Impact:

- Uncontrolled execution timing: Miners/bots can delay transaction indefinitely, exploiting transaction without any timelimit.
- Front-running and sandwich attack: Attacker can monitor the mempool and select this transaction at the worst moment resulting in huge loss if minAmountOut is `0`.
- Txn failing due to slippage check: Even if minAmountOut is set correctly, due to attacker attack, the txn will fail due to slippage check. User will lose gas fee every time they swap.
- Uncertainty on user side: user will never be sure when txn will be confirmed.

Recommendation:

Add a check to ensure deadline is not expired as mentioned here.
https://github.com/Uniswap/contracts/blob/main/src/briefcase/protocols/v3-periphery/interfaces/ISwapRouter.sol#L10

Protocol will unexpectedly earn 0 fees

Description:
Several main functions within `LabDexRouterOptimized.sol` calls `takeFeeFromInput()` in order to subtract a fee amount from the input amount provided by the users. This function calculates the fee amount depending on a variable provided by parameter `feeNumerator`. The problem is that the main functions within `LabDexRouterOptimized.sol` allow users to decide the value for `feeNumerator`, allowing them to decide the amount of fee that they will pay.

This issue is present within every function receiving feeNumerator as function parameter and it is also present within the `feeBps` variable which is used to calculate fees across several functions as well.

Impact:
As the amount of paid fee is decided by the users, they can decide to pay 0 fees every time they execute a main function.

Recommendation:
Define `feeNumerator` as a contract variable which can be only modified by the admin/owner and do not allow users to define the amount of fee that whey are paying. Update everything function receiving `feeNumerator` as a function parameter.

Client comments:
The client acknowledged the risk that it was possible that if a user directly uses their router, they could bypass the fee. And added that due to their current architecture, the issue could not be resolved without backend changes.

Auditor's comments:
We would still recommend implementing the fix to avoid losing any fees as a protocol. However, the client is okay with the current implementation while they understand the associated risks. Client also added a minimum feeNumerator check, so now the fee can not be 0, but can still be manipulated by the user.

Stuck WETH can be stolen using the method exactInputSingle() and v2 swap methods (tokens to eth)

In Contract LabDexRouterOptimized.sol, the method exactInputSingle(...) allows a user to swap an ERC20 token to WETH. Since the contract intends to send ETH (not WETH) to the user, it receives the swapped WETH, unwraps it and sends it to the user as follows:

```
IWETH9(WETH).withdraw(IERC20(WETH).balanceOf(address(this))); //@audit-
issue user will get all weth in contract by swapping very small tokens or
create their own pool with small weth and swap here to get all weth stuck
here
     amountOut = takeFeeFromInput(address(this).balance, _feeNumerator);
     require(address(this).balance >= params.amountOutMinimum,
"LAbRouter: INSUFFICIENT_OUTPUT_AMOUNT");
     Address.sendValue(payable(params.recipient), address(this).balance);
```

Here, the LabDexRouterOptimized contract is expecting the address(this).balance will be equal to the swapped WETH only, but it will include the stuck WETH tokens.
A user can swap a very small amount of ERC20 tokens and get back all the stuck WETH very easily just by calling this method.

This issue is also valid for the following methods which also swap tokens to ETH.

```
function swapExactTokensForETHSupportingFeeOnTransferTokens(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(..)
{..}
```

Impact:
The LabDexRouterOptimized contract will lose stuck WETH, which might be stuck funds from a swap or accidentally sent to the LabDexRouterOptimized contract.

Recommendation:
Use the amountOut value returned by the method exactInputInternal() to withdraw from the WETH and send that amount only to the users.
For uniswap v2 methods, also try to unwrap only amountOut calculated in the method getAmountOutFee or getAmountOut.
Also, if there are stuck funds, use the method withdraw() to recover as soon as possible.

MEDIUM-3 | RESOLVED

The restriction of using a single pool swap can be bypassed

Description:
Some of the functions within `LabDexRouterOptimized.sol` contains the following NatSpec comment: `Restricts to a single pool swap`. This is controlled by the `onlyOnePoolSwap` modifier, which is also mentioned within the NatSpec comment and expected to be included. However, the modifier is missing.

Impact:
Users can input larger paths and executing multipool swaps, which is something against the expected behavior.

Recommendation:
Add the `onlyOnePoolSwap` modifier to every function containing the NatSpec comment which states that the swap should be restricted to a single pool swap.

MEDIUM-4 | ACKNOWLEDGED

Incorrect _feeBps can lead to failed transactions or potential loss of swapped token output

Description:
The _feeBps is a parameter of the swap functions such as swapExactETHForTokensSupportingFeeOnTransferTokens, swapExactTokensForETHSupportingFeeOnTransferTokens, swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome and swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome in the router contract where a user can specify the array of feeBps for each hop. But an incorrect or inaccurate feeBps value can lead to failed transactions if not set properly (when it is set lower than the value in the pool/pair contracts). If it is set higher it can lead to successful transactions but would lead to the user swapping worseoff by making him pay more than he should have actually paid.

```
address[] calldata pairAddresses,
 uint256[] calldata _feeBps,
 uint256 _feeNumerator
```

Impact:
This could lead to user not getting sufficient valid funds for his swaps and getting lower than he could have validly got. Or it could lead to his swaps being repeatedly failing discouraging him from interacting with the router.

Recommendation:
It is recommended that the feeBps is either fetched from a valid source (and not directly from the user) or hardcoded in the contract itself based on the type of Uniswap V2 pool with the corresponding fixed fee being interacted with.

Refer: https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2Library.sol#L46 & https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L159

Client comments:
The client said that they wanted to change the fee dynamically based on user activity inside the app. And that the fee for one user could differ from another. They added that the contract is not intended for external usage and was designed only for the app.

Auditor's comments:
Users can call these functions directly without using the client's backend. Even if the backend is able to select the fee, it will affect users who receive fewer funds. The client seems to understand the associated risks.

Unused modifier or incorrect comment

Description:
The to do comments mention the onlyOnePoolSwap as a modifier being used for the following functions:

swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome

(path)// / - `onlyOnePoolSwap(path)`: Restricts to a single pool swap. // is being commented above many functions but is being used in only couple of them (one directly and one indirectly). The modifier for these functions was not added and the comments must either be removed or the code being fixed to remain in consistent with the code comments.

```
address[] calldata pairAddresses,
 uint256[] calldata _feeBps,
 uint256 _feeNumerator
```

Impact:
This could lead to user not getting sufficient valid funds for his swaps and getting lower than he could have validly got. Or it could lead to his swaps being repeatedly failing discouraging him from interacting with the router.

Comments: The client team said that this comment was related to multihop paths initially being supported.

Recommendation:
It is recommended that the comments either be removed or the modifier be added to the implementation of the function. Additionally, ensure that the require check exists as follows exists for multihop swap paths.

```
require(pairAddresses.length == (path.length - 1))
```

Address `to` can be address(0) leading to ETH sent to address(0) in swap methods

In Contract LabDexRouterOptimized, the following methods have a parameter `to` which will be the address that can receive swapped ERC20 tokens or ETH.

function swapExactETHForTokensSupportingFeeOnTransferTokens(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokens(..) {..}
function swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(..) {..}

Since all these methods use the following method to send ETH, **it is possible to send ETH to address(0) even if `to` is address(0).**

```
function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient
balance");


        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Address: unable to send value, recipient may
have reverted");
    }
```

In the case of ERC20 tokens, it will revert back since ERC20 tokens are directly sent from the pool's pair address, and there is a check for the recipient being address(0) or not.

Impact:
This could lead to loss of funds for the user as the user's ERC20 tokens will be sent to the pool, but the received ETH will be sent to the address(0), lost forever.

Recommendation:

Add a check to ensure address `to` is not address(0).

Swap methods do not check the order of tokens

In Contract LabDexRouterOptimized, the following methods have path[] enabling the swap between WETH and ERC20 tokens.

```
function swapExactETHForTokensSupportingFeeOnTransferTokens(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokens(..) {..}
function swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome(..)
{..}
function swapExactTokensForETHSupportingFeeOnTransferTokensAerodrome(..)
{..}
```

Here, the methods does not check which token is first and which is second.
For eg. if users wants to swap from WETH to ERC20 token, path[0] should be WETH and path[1] should be ERC20 tokens.

The contract only ensures that one of the token is WETH and that is insufficient.

For eg. If a user wants to swap ETH to ERC20 tokens but provides path[0] as ERC20 and path[1] as WETH by mistake, the swap will fail since input and output tokens will reverse making the pool infer no tokens were send to swap and hence transaction will fail.

Impact:
There is no funds loss for users since the transaction will revert but it will keep failing until tokens in the correct order are send.

Recommendation:
Add check to ensure that

- Methods which swap ETH to tokens must have
  - Path[0] as WETH
  - Path[1] as ERC20 tokens
- Methods which swap ERC20 tokens to ETH must have
  - Path[0] as ERC20 tokens
  - Path[1] as WETH

ETH Amount Mismatch

Description:
In the exactInputSingle function, when ETH is used as the input token:

```
newParams.amountIn = takeFeeFromInput(msg.value, _feeNumerator);
```

The newParams.amountIn is updated after takeFeeFromInput so it does not have a greater impact

Impact:
Mismatch between amounts can be shown on UI / backend

Recommendation:
Ensure newParams.amountIn is strictly bounded by msg.value and does not consume existing contract ETH.

Reliance on balanceOf(address(this)) Instead of Operation Return Values

Description:
The contract frequently uses address(this).balance or ERC20.balanceOf(address(this))) after performing a swap or other operation, assuming that this reflects the output amount of the previous operation. However, this assumption can be unsafe, as the contract may already hold additional tokens of the same type before the operation. As a result, the balanceOf value may be larger than the actual swap output, leading to overestimations of available funds.

Impact:
The contract may attempt to use more tokens than the actual swap output, pulling in pre-existing balances that are unrelated to the transaction.
This can result in inaccurate accounting, misrouted funds, or inconsistencies in the contract's logic. In certain cases, if attackers are able to transfer tokens to the contract before or during execution, this could lead to manipulation of the assumed output.

Recommendation:
Avoid using balanceOf(address(this)) as a proxy for the operation's output. Instead, always rely on the explicit return values from the function call (e.g., the swap's amountOut return variable). This ensures that only the actual output of the operation is used in subsequent logic and prevents the inclusion of unrelated balances. Adding an internal accounting balance mechanism can be also a good option.

Implement the fix in every affected function.

Reliance on On-Chain Reserve Calculations

Description:
The _handleSwapIteration function calculates amountInput and amountOutput using on-chain reserves fetched from the pair contract:

```
(uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
(reserveInput, reserveOutput) = input == token0 ? (reserve0, reserve1) :
(reserve1, reserve0);
amountInput = IERC20(input).balanceOf(address(pair)) - reserveInput;
```

This approach relies on real-time on-chain state which can be manipulated by an attacker through a frontrunning transaction. An attacker could perform a quick swap before the victim's transaction executes, temporarily skewing the reserves and causing the victim to execute at a worse rate.

Impact:
Users may experience transactions executing at the worst price within their defined slippage tolerance (e.g., 10%).

Relying on on-chain reserves is not recommended.

Recommendation:
Consider using off-chain quoting so reserves can not get manipulated.

Using not locked solidity versions can lead to unexpected behaviours

Description:
Some contracts in the codebase (e.g., LabDexRouterOptimized.sol, FeeExtension.sol, and CustomBytesLib.sol) are using unlocked Solidity pragma statements such as pragma solidity ^0.8.15;. This allows the compiler to use any version greater than or equal to 0.8.15 and less than 0.9.0. While this provides flexibility, it can introduce risks if newer compiler versions contain undiscovered bugs, behavior changes, or incompatibilities.

Impact:
Future Solidity compiler versions could introduce breaking changes, security vulnerabilities, or unintended behavior.

Builds may become non-deterministic if developers use different compiler versions.

Auditability and reproducibility of the contracts are reduced.

Recommendation:
Lock the Solidity compiler to a specific, stable version (e.g., pragma solidity 0.8.15;) to ensure deterministic builds, avoid unexpected behavior, and improve long-term maintainability and auditability of the contracts.

Missing Deadline / Transaction Expiration Check

Description:
The exactInputSingle function currently does not include a deadline or transaction expiration mechanism. This means that a swap transaction could be executed at any time after submission if the calldata is reused or delayed in the mempool.

Impact:
Users are exposed to front-running or sandwich attacks, especially during volatile market conditions.
Transactions could execute at significantly worse prices than intended if delayed.
Reduces overall user safety and trust in time-sensitive swaps.

Recommendation:
Introduce a deadline parameter in ExactInputSingleParams or as a separate argument.
Include a transactionExpired(deadline) modifier to revert if block.timestamp > deadline.
Ensure that all swaps respect this deadline to protect users from delayed execution.

CEI pattern not followed leading to a possible reentrancy attack.

Description:
The functions within `LabDexRouterOptimized.sol` take some fee from the user input and send the fee amount to a vault contract. This execution is done by the `takeFeeFromInput()` function which is usually not called after updating the current contract state.

Impact:
The vault contract, which is out of scope, is gaining the control of the execution when receiving ether, opening the possibility of recalling any other function without having the main contract state updated. The severity has been marked to low as the vault contract is supposed to be a contract controlled by the protocol.

Recommendation:
Even if the vault contract is controlled by the protocol, it is recommended to always follow the CEI pattern to remove the technical possibility of executing reentrancy attacks. Update every function to first update the state and then transfer the fee to the vault.

Method exactinputSingle() can have different params.amountIn and msg.value

In Contract LabDexRouterOptimized.sol, the method exactInputSingle(...) allows users to swap WETH to an ERC20 token. It has the following check to ensure that the user has provided ETH for the swap.

```
require(msg.value > 0, "LAbRouter: msg.value is zero")
```

Here, it is checked msg.value > 0 or not but it is not checked if msg.value > params.amountIn.
So a user can pass amountIn > msg.value, but the next line sets amountIn to the correct value as follows:

```
newParams.amountIn = takeFeeFromInput(msg.value, _feeNumerator);
```

Impact:
If params.amountIn > msg.value, amountIn will be set to the correct value when the fee is deducted; however, this can cause data-related issues for both the UI and backend.

Recommendation:
Add a check to ensure that msg.value == params.amountIn

Missing check for single pool swap and pair addresses length

In Contract LabDexRouterOptimized, the following methods have comments mentioning that these methods should be used for a single pool swap, but there is no such check to ensure that.

```
function swapExactETHForTokensSupportingFeeOnTransferTokens(..) {..}
function swapExactTokensForETHSupportingFeeOnTransferTokens(..) {..}
function swapExactETHForTokensSupportingFeeOnTransferTokensAerodrome(..)
{..}
```

Because of this, the pair address length will be 1 as well.

Similarly, pairAddresses.length can be `0` and _feeBps length can be `0` leading to swap being reverted.

Impact:
Although the contract's logic can handle a multi-path swap but the requirement is just a single pool swap with just one pool address. And it is possibility of transaction being reverted due to these issues.

Recommendation:
Add a check to ensure that path.length == 2, pairAddresses.length == 1 and feeBps.length == 1.

INFORMATIONAL-1 | ACKNOWLEDGED

The constructor is not setting feeNumerator

Description:
The constructor within `LabDexRouterOptimized.sol` contains the following NatSpec comment:

```
`/// @notice Sets initial vault address, fee numerator, WETH address and distributes AccessControl roles`
```

However, feeNumerator is not defined.

Recommendation:
Update the NatSpec comment, removing the feeNumerator definition or include it.

INFORMATIONAL-2 | ACKNOWLEDGED

Deprecated checks

Description:
The `toAddress()` and `toUint24()` functions from `CustomBytesLib.sol` contain certain checks which were useful for solidity version prior to 0.8.x:

```
`require(_start + 20 >= _start, 'toAddress_overflow');`
`require(_start + 3 >= _start, 'toUint24_overflow');`
```

If the contract is compiled with a version greater than 0.8.x, these checks are automatically performed.
It is important to implement the recommendation from previous reported issues and lock the solidity version so a version greater than 0.8.x.

contracts/LabDexRouterOptimized.sol contracts/extensions/FeeExtension.sol contracts/libs/CustomBytesLib.sol contracts/dexes/uniswapV3/v3-periphery/interfaces/ISwapRouter.sol contracts/dexes/uniswapV3/v3-periphery/interfaces/IQuoterV2.sol contracts/dexes/uniswapV3/v3-periphery/interfaces/IAerodromeQuoter.sol	
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Lab team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Lab team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo