BAIL security

BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

LFJ

October 2024

Disclaimer:

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	LFJ - Universal Router
Website	Lfg.gg
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/traderjoe-xyz/joe-router/tree/5a60dad2bd422535f5ce64ce37ed3b0c5706c6d7/src
Resolution 1	https://github.com/traderjoe-xyz/joe-router/tree/4045683f0663fdce13c657b45804f4bf841f8acd/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1		
Medium	3	1		2
Low	12	6	1	5
Informational	9	5		4
Governance	1	1		
Total	26	14	1	11

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Global

Issue_01	Usage of “memory-safe” is not allowed if returndatacopy occupies more than 0x80
Severity	**Informational**
Description	Whenever memory is allocated, the general way of doing this is by allocating at offset of where the free memory pointer points to, which is usually at 0x80. However, for smaller return data, it is allowed to store them starting from offset 0. What is not allowed is to start from offset 0 and store up to (including 0x80) or even more. This is considered unsafe. The codebase often uses the following snippet: *returndatacopy(0, 0, returndatasize())* This is considered as unsafe if return data has 4 or more arguments as that would essentially occupy 0x80 and more.
Recommendations	Consider removing “memory-safe” for all operations where the return data has >= 4 arguments.
Comments / Resolution	Acknowledged.

Libraries

Flags

The Flags library is a simple helper library which is designed to extract specific values from the flags part of each specific swap data within the route calldata. It is used solely within the RouterLogic and RouterAdapter contracts.

Appendix: Flag Crafting

The flag is represented starting from the LSB within the swap data in the route. It consists of 2 bytes.

First Bit: Includes information about the swap direction

Second Bit: Includes information whether a callback is needed

Bits 2-7: Reserved for future use

Bits 8-15: Used for the DEX ID. The following DEXes are compatible:

- UNISWAP_V2_ID
- TRADERJOE_LEGACY_LB_ID
- TRADERJOE_LB_ID
- UNISWAP_V3_ID

Appendix: Core Invariants

INV 1: The flag part must consist out of 16 bits

INV 2: ZERO_FOR_ONE must occupy bit 0

INV 3: CALLBACK must occupy bit 1

INV 4: The bits 2-7 must be empty

INV 5: DEX ID must occupy bit 8-15

Privileged Functions

- none

Issue_02	ID_OFFSET is unused
Severity	**Informational**
Description	The ID_OFFSET state variable is currently not used within the contract. This will confuse third-party reviewers and increases the contract size for no reason.
Recommendations	Consider removing ID_OFFSET
Comments / Resolution	Resolved, ID_OFFSET is now used.

PackedRoute

The PackedRoute library contains functionality to decode and interpret packed data from the route for swaps, such as:

a) If the first token is a transfer-tax token
b) Token address for corresponding ID
c) Next swap data
d) Previous swap data
e) Decoding of swap data
   I. pair address
   II. percent
   III. flag
   IV. tokenIn ID
   V. tokenOut ID

This library provides functions to efficiently extract and decode this information using inline assembly and low-level operations for gas optimization and is solely used as a helper library within the RouterLogic contract to facilitate swaps.

Appendix: Route Crafting

The route calldata is being crafted as follows:

- number of tokens [1 byte]
- transfer-tax route [1 byte]
- token addresses [20 bytes each]
- swap info 1 [26 bytes each]
  - pair address [20 bytes]
  - BPS (percentage) [2 bytes]
  - flag (DEX ID, callback, ZERO_TO_ONE) [2 bytes]
  - tokenIn ID [1 byte]
  - tokenOut ID [1 byte]

Appendix: Core Invariants

The provided route data parameter must strictly confirm with the following invariants. Violation of these invariants can result in unexpected side-effects.

INV 1: The number of tokens must align with the amount of token addresses

INV 2: Token addresses must be unique

INV 3: Pair addresses must be unique

INV 4: The number of tokens must at least be two

INV 5: The IDs for tokenIn and tokenOut in the corresponding swap data must match with the corresponding index of the tokens

INV 6: tokenIn must align with the first index token

INV 7: tokenOut must align with the last index token

INV 8: The pair must always align with the corresponding DEX ID

INV 9: For partial swaps, the final swap must always use 10 000 as percentage.

INV 10: The callback flag must only be enabled for UniV3

INV 11: Only the first and the last token can be transfer tax tokens

INV 12: If the first token is a transfer-tax token, the route must be marked as such

INV 13: tokenOutId must not be used other than at the end of the route

INV 14: percentage values must be within 1 and 10_000

If for example the last swap is only using 90% of the token balance, the remaining 10% will be stuck in the RouterLogic.

Privileged Functions

- none

No issues found.

PairInteraction

The PairInteraction library is used within the RouterAdapter contract and is responsible for interacting with various pair contracts, such as Uniswap V2, Uniswap V3, and Trader Joe's Liquidity Book (LB) pairs.

Specifically, it provides low-level functions to retrieve reserves, calculate required input amounts for swaps, and execute token swaps across these different DEX protocols.

The library uses inline assembly for gas optimization and direct interaction with the pair contracts, enabling efficient retrieval of data and execution of swaps.

The following DEXes are currently supported:

a) UniswapV2
b) LB Legacy
c) LB V2
d) UniswapV3

Appendix: Core Invariants:

INV 1: The return data size of various functions must always match with the expected size

INV 2: Failure of external calls must result in revert

INV 3: The return data during a revert must not be > 128 bytes

INV 4: The free memory pointer must always either be updated or reset after it has been overridden

Privileged Functions

- none

No issues found.

RouterLib

The RouterLib library is a helper library used within the Router contract to facilitate token transfers, allowance management and swapping operations.
It provides functions such as validateAndTransfer, transfer, and swap to handle the token interactions and allowance checks.
The validateAndTransfer function ensures that a user has sufficient allowance to transfer a specified amount of tokens and performs the transfer if valid.
The transfer function interacts with the Router's fallback mechanism to transfer tokens from one address to another.
The swap function coordinates the swapping process by setting up temporary allowances for the logic contract to spend tokens on behalf of the caller (or Router in case of tokenIn = native), executing the swap through the RouterLogic, and then resetting the allowances afterward.

On top of that, the RouterLogic contract uses the transfer function to transfer in the first token of the swap path.

Appendix: Allowance Slot

Whenever the swapExactOut or swapExactIn function in the Router is invoked, the RouterLib library fetches a dedicated AllowanceSlot which is derived as follows:

keccak256(token, sender, from)

whereas:

- token = input token
- sender = RouterLogic
- from = msg.sender

In the scenario where the native token is the input token, from will be address(this).

The allowance value is stored at keccak256(key, slot)

At the beginning of the swap, this AllowanceSlot will be set to amountIn (swapExactIn) or maxAmountIn (swapExactOut), which grants the RouterLogic contract allowance to trigger either a transferFrom call on behalf of the Router contract or a simple transfer call (for native input). The allowance is then consumed whenever the validateAndTransfer function (via the Router's fallback function) is invoked and fully reset after the swap has ended.

Appendix: Core Invariants

INV 1: The key for the allowance slot must always be derived as keccak(token, sender, from)

INV 2: The allowance must be fully reset after a successful swap

INV 3: A transfer via validateAndTransfer must revert if the allowance is not sufficient

INV 4: A transfer via validateAndTransfer must correctly decrease the allowance

INV 5: A transfer must trigger the fallback function in the Router

INV 6: Allowance slots must be correctly and uniquely crafted via getAllowanceSlot

Privileged Functions

- none

Issue_03	Transfer does not work if first 4 bytes in calldata match with a function selector from the Router
Severity	**Low**
Description	The transfer function is executing a low-level call to the Router contract with the target to trigger the fallback function. Since the calldata is crafted as follows: *token, from, to, amount* It is possible that the first 4 bytes from the 20 bytes token address can accidentally match with a function selector from the Router which would effectively result in a revert. Such a token could never be used as tokenIn.
Recommendations	Consider adding 4 bytes at the beginning of the calldata which will not match with any function selector.
Comments / Resolution	Resolved, the first 4 bytes are now zero.

Issue_04	getAllowanceSlot overrides free memory pointer
Severity	**Informational**

Description	During the getAllowanceSlot function, memory is modified as follows:
	
	mstore(0, shl(96, token))
	mstore(20, shl(96, sender))
	mstore(40, shl(96, from))
	
	This practice is generally considered as dangerous as the free memory pointer should never be overridden without being updated or reset afterwards.
	
	In that specific scenario, this is however not an issue because the free memory pointer cannot grow too big, thus the upper 28 bits are always zeros, which means that zeros are overridden by zeros (due to the SHL operation)

Recommendations	We do not recommend a change. This should however be carefully considered in the future.
Comments / Resolution	Acknowledged.

TokenLib

The TokenLib library is a library designed to facilitate common token operations such as querying balances, transferring tokens, handling native transfers and wrapping/unwrapping ETH/WETH. It uses low-level assembly code to optimize gas usage and ensure compatibility with various token implementations, including those that may not fully adhere to the ERC20 standard.

This library is used within the RouterLib, Router, RouterLogic and RouterAdapter contracts.

Privileged Functions

- none

No issues found.

Core

Router

The Router contract allows users to facilitate token swaps using predefined routes. It allows for exchanging tokens by specifying either the exact input amount or the exact output amount desired, handling both ERC20 tokens and the native scenario by wrapping or unwrapping native tokens as needed.
The contract delegates the actual swapping logic to an external logic contract, which can be updated by the owner, while managing token allowances internally to ensure the logic contract can spend tokens on behalf of users safely but only during the execution of the swap.

It includes functions to simulate swaps without executing them, useful for estimating outcomes and integration testing, and incorporates security measures such as parameter validation, deadline checks, and ownership control to ensure only intended recipients receive tokens. The Router interacts with TokenLib and RouterLib libraries for low-level token interactions and shared swap logic.

Appendix: Refund Mechanism

The Router contract refunds unspent native tokens back to the caller as follows:

a) swapExactIn and tokenIn != address(0):

-> full msg.value is simply refunded

b) swapExactIn and tokenIn == address(0):

-> msg.value - amountIn is refunded, which simply covers any unspent msg.value

c) swapExactOut and tokenIn != address(0):

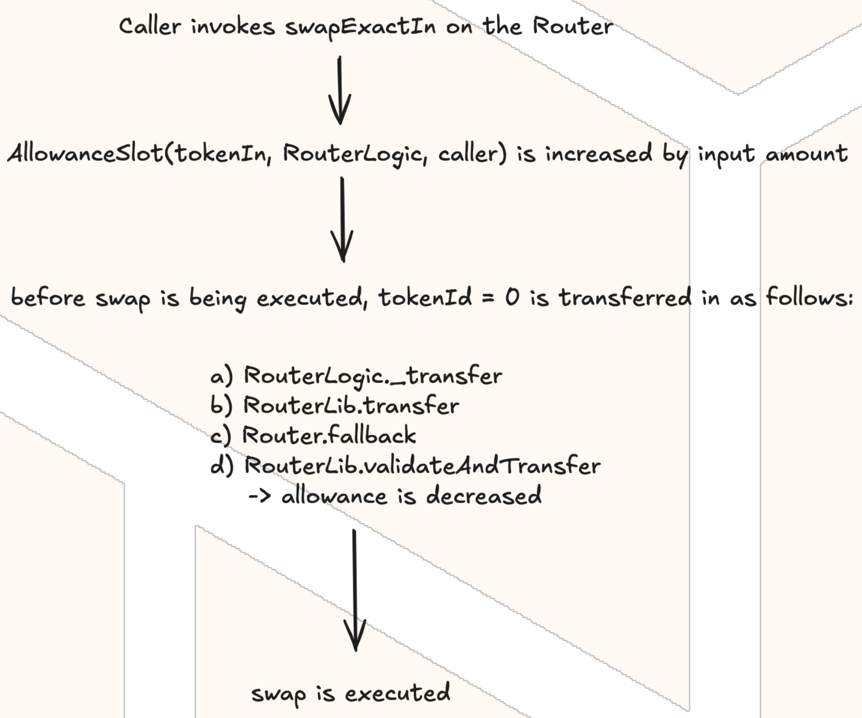
-> full msg.value is simply refunded

d) swapExactOut and tokenIn == address(0):

-> msg.value + (amountInMax - takenIn) - amountInMax is refunded, which simply covers any unspent msg.value

Appendix: Fallback logic

The Router contract exposes a fallback function which allows any address to trigger the transferFrom or transfer function on behalf of the Router contract, as long as the caller has sufficient approval via the AllowanceSlot. During the normal business logic, allowance is only granted to the RouterLogic contract during the execution of a swap, the fund transfer can be illustrated as follows:


Caller invokes swapExactIn on the Router
AllowanceSlot(tokenIn, RouterLogic, caller) is increased by input amount
before swap is being executed, tokenId = 0 is transferred in as follows:
a) RouterLogic._transfer
b) RouterLib.transfer
c) Router.fallback
d) RouterLib.validateAndTransfer
-> allowance is decreased
swap is executed

In the scenario where token0 is the native token, the key for the AllowanceSlot will be determined as keccak(tokenIn, RouterLogic, Router). As one can see, the Router will now be the from address because native ETH has been provided as msg.value which is now sitting as WETH within the Router.

Appendix: Simulation

The simulate and simulateSingle functions allow for precise swap simulations. While the simulateSingle function allows for simulation a single swap execution with its corresponding route, the simulate function allows for simulating multiple different swap paths with the goal to determine the best path. Depending if the execution is swapExactIn or swapExactOut, the output or input amount is being returned.

Appendix: Core Invariants

INV 1: Unconsumed msg.value during swapExactOut must be refunded as follows:

> msg.value + (amountInMax - takenIn) - amountInMax

INV 2: Unconsumed msg.value during swapExactOut must be refunded as follows:

> msg.value - amountIn

INV 3: Users must never receive less than amountOutMin during swapExactIn

INV 4: Users must never pay more than amountInMax during swapExactOut

INV 5: block.timestamp must never exceed the deadline

INV 6: tokenIn must never be equal to tokenOut

INV 7: During swapExactIn, swap must not result in less than amountOutMin

Privileged Functions

- updateRouterLogic

Issue_05	Governance: Exotic edge-case including _logic change allows governance to maliciously consume approvals
Severity	**Governance**
Description	Currently, governance can change the logic contract. Such a change can usually not result in any issues because even if a change happens shortly before a swap and the swap is being executed with the new logic contract, the slippage check would still be valid, ensuring that users will always receive their minimum desired output amount. Furthermore, the allowance will always be limited during the swap. However, below we will illustrate a very sophisticated scenario of how to steal tokens from a caller while not providing any real tokens. a) The victim places a limit order for 100e18 USDT b) The victim wants to swap 100e18 USDC to 100e18 USDT via the Router c) Governance updates the logic contract to a malicious contract before the transaction is submitted d) When the transaction is submitted a hook within the RouterLogic contract fulfills the limit order which increases the victim’s balance by 100e18 USDT e) The malicious RouterLogic contract consumes the whole approval while returning 100e18 as totalOut f) The first slippage check passes due to the return value and the second slippage check passes because the victim received 100e18 USDT from the limit order g) The victim essentially lost 100e18 USDT
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Resolved, the caller must provide the corresponding logic contract as parameter.

Issue_06	Integration issues for on chain contracts using hard coded paths
Severity	**Low**
Description	Whenever the swapExactIn or swapExactOut function is invoked, one important parameter is the route, which determines various important settings for the swap. This parameter is crafted in a very specific way. The RouterLogic contract is specifically designed to encode the route parameter (by using the PackedRoute library). In such a scenario where the RouterLogic is changed and the decoding of the route is changed as well, it can result in integration issues where other contracts with hardcoded route parameter rules now suddenly become incompatible.
Recommendations	Consider allowing the _logic contract to be passed as argument. To further limit user flexibility we recommend implementing a whitelist mechanism which validates this argument.
Comments / Resolution	Resolved, the caller must provide the corresponding logic contract as parameter.

Issue_07	WETH within Router is drainable
Severity	**Low**

Description	In certain edge-cases it is possible that the Router contract accumulates WETH, an explanation can be found in:
	
	“Dust accumulation due to TJ and UniV3 swaps”
	
	It is possible to drain this WETH in the Router by simply executing a swapExactIn call with native ETH as tokenOut, using a pair/contract which returns an erroneous output amount that is not actually received:
	
	uint256 amountOut = _swap(pair, tokenIn, actualAmountIn, recipient, flags);
	
	balances[tokenOutId] += amountOut;
	
	uint256 amountOut = balances[nbTokens - 1];
	
	return (amountIn, amountOut);
	
	(totalIn, totalOut) = RouterLib.swap(_allowances, tokenIn, tokenOut, amountIn, amountOut, from, recipient, route, exactIn, _logic);
	
	The Router will then unwrap totalOut while not having received any tokens, which simply uses the stuck balance within the Router:
	
	TokenLib.unwrap(WNATIVE, totalOut);
	
	TokenLib.transferNative(to, totalOut);

Recommendations	Consider validating all provided calldata accurately.

Comments / Resolution	Resolved, within the _swap function, the _verifySwap call now returns the WETH amount which is received by the swap (previously the return value from the pair was used). This will now accurately transfer out all received WETH.

Issue_08	Lack of reentrancy guard may result in malicious states
Severity	**Low**
Description	Currently, the swapExactIn and swapExactOut functions do not have a reentrancy guard which means in the scenario of malicious routing, one can reenter into these functions before the original execution has finished. While we could not craft any exploit via that scenario. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users. One can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider implementing a reentrancy guard for swapExactIn and swapExactOut.
Comments / Resolution	Resolved.

Issue_09	amountInMax/amountOut can be zero
Severity	**Low**
Description	Currently, during swapExactIn and swapExactOut there is no explicit validation for both aforementioned scenarios. This can lead to increased user flexibility and undesired side-effects.
Recommendations	Consider validating these scenarios.
Comments / Resolution	Resolved.

Issue_10	Simulations are not possible for addresses with no fallback/receive function
Severity	**Low**
Description	The simulate/simulateSingle functions are triggered before swaps to simulate the corresponding outcomes. This will not work if msg.sender != to address and the msg.sender has no receive/fallback function for a native out swap.
Recommendations	Consider incorporating the to address as parameter, similar as the swapExactIn and swapExactOut functions.
Comments / Resolution	Resolved.

Issue_11	Lack of parameter validation during simulate and simulateSingle
Severity	**Informational**
Description	The simulate and simulateSingle functions are used to simulate a swap outcome given a specific route. The logic of these functions flows exactly the standard swap logic with the only difference that the execution reverts at the end of the function flow. However, these functions do not implement the _verifyParameters call which allows for simulating transactions that could not be executed.
Recommendations	Consider invoking _verifyParameters within the simulateSingle function.
Comments / Resolution	Resolved

Issue_12	Input token can be output token in native scenario
Severity	**Informational**
Description	An important invariant is to ensure that tokenIn != tokenOut. In the scenario where tokenIn = address(0) and tokenOut = WETH it would mean that the swap starts with WETH and ends with WETH which violates this invariant.
Recommendations	Consider validating this scenario.
Comments / Resolution	Resolved

RouterAdapter

The RouterAdapter contract is an abstract contract designed to facilitate interactions with various DEX pairs, including Uniswap V2, Uniswap V3, Trader Joe Legacy LB, and Trader Joe LB pairs.
It provides a unified interface for obtaining the required input amounts for swaps (_getAmountIn) and executing token swaps (_swap) across these different protocols. By utilizing a system of flags and IDs from the Flags library, the contract determines the appropriate DEX type and dispatches the swap or calculation to the corresponding specialized function.
It leverages the PairInteraction library to handle protocol-specific logic. The contract also includes a callback mechanism for Uniswap V3 swaps, managing authorization through a _callback address to ensure security during swap callbacks. The adapter pattern allows higher-level contracts to perform swaps without needing to implement protocol-specific details.

Appendix: Swap variations

Users can execute swaps either via swapExactIn or via swapExactOut. Below we will illustrate how each path works.

swapExactIn: User provides a specific input amount which is used to execute the first swap and the subsequent swap is using the output amount of the first swap for the next swap. This continues until all swaps have been executed.

swapExactOut: User provides a specific output amount and the contract then calculates the corresponding input amount, starting from the pair at the end of the route until the final input amount is derived. Once all amounts have been successfully derived

Appendix: Temporary callback address

The uniswapV3Callback function allows for transferring tokens out from the RouterLogic contract and can be invoked by the UniswapV3 pair which is used for the corresponding swap. The UniswapV3Pair is temporarily cached into the _callback variable during the swap which then allows the UniswapV3Pair to exclusively invoke uniswapV3SwapCallback.

Appendix: Core Invariants

INV 1: The _callback address should always be reset after the swap execution

INV 2: Only a valid UniswapV3Pool address must be allowed to invoke UniswapV3SwapCallback

Privileged Functions

- none

Issue_13	0.3% fee may not apply for all UniV3 DEXes
Severity	**Low**
Description	The contract assumes a fee of 0.3% for UniswapV2 swaps. This is not true for all UniswapV2 forks and since it is desired to incorporate multiple different UniV2 DEXes, there will be compatibility issues.
Recommendations	Consider refactoring the logic to use the corresponding swap fee for each different UNIV2 dex.
Comments / Resolution	Acknowledged

Issue_14	_callback address can be set non-temporarily
Severity	**Informational**
Description	The _callback address is set upon _swapUV3 and reset upon uniswapV3SwapCallback. Using a malicious pair it is possible to set the _callback without subsequently invoking uniswapV3SwapCallback which results in _callback address remaining set even after function execution. Besides the already mentioned issues such as stealing from the RouterLogic, we could currently not determine a side-effect. However, this is an important invariant which should not be violated.
Recommendations	Consider validating the calldata accordingly.
Comments / Resolution	Resolved, callbackData is reset at the end of _swapUV3

RouterLogic

The RouterLogic contract implements the core logic for token swapping using predefined routes.
It interacts with the Router contract to execute swaps according to routes defined in the PackedRoute format. The contract supports both, exact input swaps (swapExactIn), where a specific amount of input tokens is swapped for the maximum possible output tokens, and exact output swaps (swapExactOut), where a specific amount of output tokens is obtained with the minimum required input tokens.
It partially verifies the validity of routes, checks token amounts for correctness, and manages token transfers between users, pairs, and the contract itself.

The RouterLogic contract also includes functions to calculate required input amounts for exact output swaps and handles edge cases like transfer taxes Additionally, it integrates with external libraries like TokenLib and RouterLib for low-level token operations and route handling.

Appendix: Excess Balance Unused

In the swapExactIn and swapExactOut functions, an important invariant is enforced which guarantees that all input amounts provided by the caller are fully consumed during the swap process. This ensures that no input tokens remain unused.

Methodology for swapExactIn:

Balances Tracking: The function maintains a balances array representing the token balances at each step of the swap. Each index corresponds to a specific token involved in the swap route.

Total Variable: A total variable tracks the net change in balances after each swap. It starts with the initial input amount and is adjusted by adding the net effect (amountOut - amountIn) of each individual swap.

Swap Execution: For each swap in the route, the _swapExactInSingle function is called, which performs the following:

a) Calculates amountIn and updates the input token balance by subtracting amountIn.
b) Executes the swap and obtains amountOut.
c) Updates the output token balance by adding amountOut.
d) Returns the net change (amountOut - amountIn), which is added to the total.
e) **Invariant Verification:** After all swaps are executed, the function checks that the total aligns exactly with the final output balance (amountOut). If there is any discrepancy, it reverts with an ExcessBalanceUnused error.

Goal: After the swap has been finalized, all input amounts have been added and again deducted from the total variable, only reflecting the final output balance.

This can be illustrated with an example as follows:

Swap Route:

1. **USDT** → **WETH** (UniswapV2)
2. **WETH** → **WAVAX** (LFJ V2)

Swap Details:

a) Initial Input: The caller provides 2500e6 USDT
total = 2500e6

b) First Swap: Swapping USDT for WETH on UniswapV2
rate: The swap results in 1e18 WETH
returns: 999999997500000000
total = 1e18

c) Second Swap: Swapping WETH for WAVAX on LFJ V2.
rate: The swap results in 100e18 WAVAX.
returns: 99e18
total = 100e18

d) Invariant enforcement: total equals received output amount of WAVAX

Methodology for swapExactOut:
This method is similar to swapExactIn. Each index corresponds to a specific token involved in the swap route. During _getAmountsIn, total is initially set to amountOut and then similarly adjusted as within swapExactIn by just using the reverse path, ensuring that total reflects the final input amount while other amounts are fully nullified.

Appendix: Partial Swaps

The swapExactIn and swapExactOut functions allow routing through different AMMs using the same input token. As an example, if a user wants to execute a simple WETH -> USDC swap, this can either be done as:

a)

- WETH -> USDC (UniV2)

b)

- WETH (50%) -> USDC (Univ2)
- WETH (50%) -> USDC (UniV3)

Appendix: swapExactIn flow

```
> Router.swapExactIn
        > RouterLib.swap
                > RouterLogic.swapExactIn
                        > RouterLogic._swapExactInSingle
                                > handles transfers and swaps
```

Appendix: swapExactOut flow

```
> Router.swapExactOut
        > RouterLib.swap
                > RouterLogic.swapExactOut
                        > RouterLogic._getAmountsIn
                                > calculates needed input amounts
                        > RouterLogic._swapExactOutSingle
                                -> handles transfers and swaps
```

Appendix: _getAmountsIn explanation

Whenever the swapExactOut path is invoked, the caller provides an amountOut and maxAmountIn parameter. The amountOut parameter represents the target output amount of tokenOut which the user will receive after the swap has been finalized. The contract uses a distinct mechanism to calculate the initial input amount which is based on the provided amountOut and the reversed path.

Illustrated example:

A user wants to swap WETH to USDC via USDT, which gives the following path:

[WETH; USDC; USDT]

The mechanism calculates how much USDC is needed to receive the desired USDT amount and then calculates how much WETH is needed to receive the needed USDC amount. All values will then be returned and the first swap requests the necessary WETH amount from the caller.

The following methodologies will be used to retrieve corresponding input amounts:

a) UniswapV2:

Calculates amountIn based on amountOut and existing reserves. Rounds up to ensure sufficient input amount will be provided:

*return (reserveIn * amountOut * 1000 - 1) / ((reserveOut - amountOut) * 997) + 1*

b) Legacy LB:

Interacts with LBRouter.getSwapIn to receive the appropriate input amount for the desired output amount

c) LB:

Interacts with LBPair.getSwapIn to receive the appropriate input amount for the desired output amount

d) UniV3:

Executes a virtual exactOutput swap which reverts upon the callback and provides the input amount as return value:

revert RouterAdapter__UniswapV3SwapCallbackOnly(amount0Delta, amount1Delta)

Appendix: Core Invariants:

INV 1: During swapExactIn, all input amounts must be consumed without any leftover, including all swap outputs which are not the output token.

INV 2: A route must consist at least of two tokens.

INV 3: During swapExactOut, percentage calculation must not result in truncation.

INV 4: During swapExactOut, all elements within amountsIn[] must properly reflect the received output amount from the previous swap.

INV 5: RouterLogic must never consume more input tokens than the provided amountIn / maxAmountIn

INV 6: During _getAmountsIn, the amountIn value must always be rounded up

INV 7: Only the Router can invoke swapExactIn/swapExactOut

INV 8: Amounts must never be larger than uint128.max

Privileged Functions

- sweep

Issue_15	Edge-case for UniV3/Algebra swaps can result in stuck funds and subsequent draining by malicious actor
Severity	**High**
Description	Whenever users swap tokens via the swapExactIn function, an important invariant is to ensure that all input tokens are fully consumed in the subsequent swap(s). This is ensured via the invariant check after all swaps: *if (total != amountOut) revert RouterLogic__ExcessBalanceUnused();* The input amount is always fully deducted during the return value, which ensures that the invariant is guaranteed: *uint256 amountIn = balances[tokenInId] * percent / BPS;* *unchecked {* *return amountOut - amountIn;* *}* For all swaps, the input amount is fully used during the swap, this is true for: a) UniV2 b) LB Legacy c) LB V2 d) UniV3 This lies in the nature of the swapping mechanism. There is one specific edge-case for UniV3 swaps, which will result in not all input amounts being consumed. This specific scenario will occur if the existing liquidity range cannot absorb the provided input amount:

	if (exactIn) {
	uint256 amountRemainingLessFee = FullMath.mulDiv(uint256(amountRemaining), 1e6 - feePips, 1e6);
	amountIn = zeroForOne
	? SqrtPriceMath.getAmount0Delta(sqrtRatioTargetX96, sqrtRatioCurrentX96, liquidity, true)
	: SqrtPriceMath.getAmount1Delta(sqrtRatioCurrentX96, sqrtRatioTargetX96, liquidity, true);
	if (amountRemainingLessFee >= amountIn) sqrtRatioNextX96 = sqrtRatioTargetX96;
	
	(see https://github.com/Uniswap/v3-core/blob/main/contracts/libraries/SwapMath.sol)
	
	In that scenario, the provided input amount is not fully consumed:
	
	state.amountSpecifiedRemaining -= (step.amountIn + step.feeAmount).toInt256();
	
	(amount0, amount1) = zeroForOne == exactInput
	? (amountSpecified - state.amountSpecifiedRemaining, state.amountCalculated)
	: (state.amountCalculated, amountSpecified - state.amountSpecifiedRemaining);
	
	IUniswapV3SwapCallback(msg.sender).uniswapV3SwapCallback(amount0, amount1, data);
	
	Which means that the invariant check erroneously assumes that the full amount was consumed, while it in fact was not, because the check is based on the amount which was initially intended to be consumed by the swap:
	*uint256 amountIn = balances[tokenInId] * percent / BPS;*

	The swap can still succeed if the price was very beneficial / the user has a reasonable slippage. All unconsumed funds remain stuck in the contract until governance withdraws it. This issue has been marked as high instead of medium severity because there are several ways for users to steal funds from the RouterLogic contract.
Recommendations	Consider ensuring that the input amount matches the expected input amount.
Comments / Resolution	Resolved, within RouterAdapter._swapUV3 it is now ensured that the callback transfers the exact amount in which was previously provided as input amount.

Issue_16	Usage of return value from swaps can result in void minAmountOut check
Severity	**Medium**
Description	Whenever a swap is executed, which can be via: -> UniswapV2 -> LegacyLB -> LB -> UniswapV3 The output amount is not casted via a balance before/after check but simply via the return value from the corresponding functions: `uint256 amountOut = _swap(pair, tokenIn, actualAmountIn, recipient, flags);` This is then furthermore used to increase the balances mapping:

	balances[tokenOutId] += amountOut; Which is then used to determine the slippage check: *uint256 amountOut = balances[nbTokens - 1];* *if (amountOut < amountOutMin) revert RouterLogic__InsufficientAmountOut(amountOut, amountOutMin);* In the scenario where the router receives less tokens than the return value of the swap expresses, which can be the case for transfer-tax tokens or debase tokens: (https://github.com/bailsec/BailSec/blob/main/Bailsec%20-%20Ponzio%20The%20Cat%20Final%20Report.pdf) This will result in the slippage check being inaccurate and the user receiving less tokens than anticipated. Fortunately, the RouterLogic is interacting with the Router and the Router itself is executing a _verifySwap call which incorporates a before-after check on the recipient address. However, there is an interesting edge-case which allows to tricking the Router into thinking that the recipient address has received at least amountOutMin: a) Charles creates a LO to swap 100000 USDC to 100000 PCAT b) Charles invokes the swapExactIn function on the Router with the goal to swap USDC to PCAT and amountOutMin of 10000 PCAT c) The swap is being executed and it routes over a pair with a malicious token (while still satisfying the swap target). The attacker now re enters upon the transfer of the malicious token and fulfills the LO which was previously created by Charles in a), which means that

	the PCAT balance of Charles has already increased by 90000 (10% transfer-tax) d) The last swap of the route is being executed and the pair returns a value of 10000 PCAT. This value passes the check within the RouterLogic contract (due to the usage of the pure return value) and errantly passes the check in the Router as well due to the LO fulfillment. e) Charles effectively only received 9000 PCAT while amountOutMin was 10000 PCAT
Recommendations	Consider either ensuring that the RouterLogic will only be used with the corresponding router or ensure that the recipient has received the correct balance.
Comments / Resolution	Resolved, the RouterLogic will only be used with the corresponding Router. While it is theoretically still possible for the above-mentioned edge-case to happen, the likelihood is so low that this can be considered as resolved.

Issue_17	Usage of rebase token during routing can result in stuck funds within the Router
Severity	**Medium**
Description	Whenever a swap is routed via different pairs for the exact input scenario, the amount which is received as result from the previous swap is going to be used for one or more subsequent swaps (using swapExactIn as example): *uint256 amountIn = balances[tokenInId] * percent / BPS;* In such a scenario where a rebase token is being used, this could result in the RouterLogic contract receiving a slightly higher token amount than expected/returned by the previous swap call (depending on the rebase logic). This amount would not be consumed but remains stuck within the Router.
Recommendations	Consider if it makes sense to skim the contract after the swap execution for any remaining balances of ERC20 tokens which are used during the swapping route (similar to Uniswap's Universal Router). Alternatively, this issue can safely be acknowledged and users can be refunded manually in such a rare scenario.
Comments / Resolution	Acknowledged, such tokens would not be used within the routes themselves; they can only be the first or the last token.

Issue_18	INV 13 is limiting swap flexibility
Severity	**Medium**
Description	INV 13 enforces that the tokenOutId must not be used anywhere else than during the very last swap: *address recipient = tokenOutId == balances.length - 1 ? to : address(this);* If the path would be as follows: WETH -UniV2> USDC -TJV2> WAVAX -UniV3> USDC The first swap would result in USDC being transferred to the recipient instead of the TJV2 pair, thus resulting in a revert. Additionally, INV 2 enforces that token addresses must be unique, which prevents the scenario where an additional tokenId can be used for the first USDC swap. This will inherently limit swap flexibility for some paths.
Recommendations	Consider if it is an acceptable limitation, if not, consider rethinking the enforcement of INV 2 and potential side-effects if that invariant is not enforced. It has to be noted that INV2 is not specifically code-enforced so it can still be provided incorrectly within the calldata. INV 13 instead is enforced with the recipient determination as the swap would simply revert continuation in that scenario due to a lack of existing funds.
Comments / Resolution	Acknowledged, this is expected behavior for the current version of the router.

Issue_19	Tokens within the RouterLogic can be stolen due to arbitrary pair address
Severity	**Low**
Description	First of all, we need to mention that this issue would usually be of low severity. However, if we incorporate: “Edge-case for UniV3/Algebra swaps can result in stuck funds and violation of INV 1” into the thought-process, this issue becomes high severity because it can happen that a significant amount of tokens will be sitting within the RouterLogic contract, which is meant to be withdrawn from LFJ and refunded to the initial swapper. An attacker can simply execute a swap with an arbitrary pair address which corresponds to a malicious contract that invokes the uniswapV3SwapCallback function with the desired amount. Furthermore, it must be ensured that the swap does not revert by fulfilling all slippage checks but this can be facilitated easily. **Another option** how to steal stuck tokens within the RouterLogic contract is by providing custom pairs for the swapExactOut flow. PoC: WETH -> USDT -> USDC ; amountOut = 2500 1. Correct _getAmountsIn would be: [1, 2500] 2. Providing malicious pair results in:

	[1, 3000] 3. Executing swap using malicious pair results in first swap receiving 2500 as output amount and second swap transferring in 3000 towards the pair (during a legit callback), which takes 500 leftover tokens within the RouterLogic contract. On top of that, there are several other scenarios which can be used to steal tokens from the RouterLogic contract due to arbitrary calldata, such as using a malicious contract as pair that returns an incorrect output amount which is then used on the subsequent swap for the _transfer call: *uint256 amountOut = _swap(pair, tokenIn, actualAmountIn, recipient, flags);* *(address tokenIn, uint256 actualAmountIn) = _transfer(route, tokenInId, from, Flags.callback(flags) ? address(this) : pair, amountIn);* Another notable scenario is a malicious pair address which simply transfers the provided amount back to the user and returns the desired output amount without actually transferring it to the Router. The subsequent swap will then funds which are sitting in the Router.
Recommendations	Consider not allowing for arbitrary pairs. A simple check within the corresponding functions: _swapUV2 _swapLegacyLB _swapLB _swapUV3 cross-checking within each factory is sufficient.

	Furthermore, we recommend ensuring that the provided pair in the swap data matches with the token addresses for the corresponding tokenInId and tokenOutId.
Comments / Resolution	The issue severity has been decreased from HIGH to LOW because "Edge-case for UniV3/Algebra swaps can result in stuck funds and subsequent draining by malicious actor" has been fixed and thus the RouterLogic contract will only accumulate dust. Partially resolved, within RouterAdapter._swapUV3, it is now checked that the callback transfers the exact amount in which is intended as input amount. However, there are still several other scenarios to drain the dust from the RouterLogic contract.

Issue_20	Unconsidered scenario within swapExactOut allows for draining funds in the RouterLogic contract
Severity	**Low**
Description	First of all, we need to mention that this issue would usually be of low severity. However, if we incorporate: "Edge-case for UniV3/Algebra swaps can result in stuck funds and violation of INV 1" into the thought-process, this issue becomes high severity because it can happen that a significant amount of tokens will be sitting within the RouterLogic contract, which is meant to be withdrawn from LFJ and refunded to the initial swapper. Besides providing arbitrary calldata, another sophisticated option exists to drain tokens from the RouterLogic contract by abusing the fact that the _getAmountsIn function does not account for slippage (if

	multiple times the same pair is used) and the calldata allows for providing the same pair multiple times. The slippage effect from swap A will then result in a different price in the pair which results in a different output in the subsequent swap when amountIn was provided. Combined with the fact that all swaps are enforced to be executed with amountIn, this results in the Router transferring in amountIn towards the UniV3 pair even if the previous swap has not resulted in the same output amount. PoC: In the following path, the _getAmountsIn function returns the following amountsIn: [1, 2500, 1, 2500] The swap execution is however as follows: WETH -> USDT (UniV2) -> provide 1, receive 2500 -> changes the price in the UniV2 [WETH/USDT] pair USDT -> WETH (TJ V2) -> provide 2500, receive 1 WETH -> USDT (UniV2) -> executes a swap on the pair where the price has decreased, swap results in less output -> provide 1, receive 2400 USDT -> WETH (UniV3)

-> provide 2500, receive 1

During the pre-last swap, the amount of 2500 has not been actually received in the Router, instead only 2400 has been received. However, the leftover amount of 100 in the Router was used for this swap. As one can see, in this example the user has not yet gained an advantage but stuck funds within the Router have been drained. If now the user wants to gain an advantage, he can rebalance the UniV2 pool and reclaim the “slippage” which has been previously paid.

Another option is whenever a transfer-tax token is used during swapExactOut for the first swap with TJ swaps. As the TJ pair accepts any input amount and the output amount will be smaller as expected, while still attempting to transfer amountIn (which has been previously calculated) for the subsequent swap, which will then take a part of the funds within RouterLogic contract.

A third option to steal tokens from the RouterLogic contract lies within the fact that the recipient address can be the to address even in the middle of a multi-hop swap:

address recipient = tokenOutId == balances.length - 1 ? to : address(this);

If a subsequent swap is happening, this swap would now consume any stuck funds within the RouterLogic (because the output amount from the previous swap has not been transferred to the RouterLogic but rather to the recipient).

The issue severity has been decreased from HIGH to LOW because “Edge-case for UniV3/Algebra swaps can result in stuck funds and subsequent draining by malicious actor” has been fixed and thus the RouterLogic contract will only accumulate dust.

Recommendations	Consider validating the calldata to ensure that pools are non-unique.
Comments / Resolution	The issue severity has been decreased from HIGH to LOW because “Edge-case for UniV3/Algebra swaps can result in stuck funds and subsequent draining by malicious actor” has been fixed and thus the RouterLogic contract will only accumulate dust. Acknowledged, since the contract will only accumulate dust the impact is negligible

Issue_21	Dust accumulation due to TJ and UniV3 swaps
Severity	**Low**
Description	During swaps, it is possible to accumulate dust within the RouterLogic and the Router contracts via UniV3 and TJ V1 swaps, which can result in amountOut being slightly lower than what was received in the final swap. There are two distinct scenarios: a) During swaps, the output token is received in the RouterLogic contract and the return value of the swap is cached. Between those two values is a slight difference due to rounding. The subsequent swap is a UniV3 swap which only transfers the return value to the pair while ignoring the accumulated dust. b) During the final swap, the output token WETH while the swap was meant to result in native ETH. This will result in the output WETH amount being transferred to the Router and the amountOut being unwrapped. In the scenario where amountOut is slightly lower, the difference will remain stuck in the Router.
Recommendations	This issue is inherently present in the business logic. We do not recommend a change.
Comments / Resolution	Acknowledged.

Issue_22	Legitimate swaps may revert under rare edge-cases within _swapExactOutSingle
Severity	**Low**

Description	The _swapExactOutSingle function uses as amountIn the corresponding element in the amountsIn array which is returned by the _getAmountsIn function:
	
	(uint256 amountIn, uint256[] memory amountsIn) = _getAmountsIn(route, amountOut, nbTokens, nbSwaps);
	
	for (uint256 i; i < nbSwaps; i++) {
	(ptr, value) = PackedRoute.next(route, ptr);
	
	swapExactOutSingle(route, nbTokens, from, to_, value, amountsIn[i]);
	}
	
	In rare edge-cases, it can happen that the subsequent swap does not result in a sufficient output amount which meets the desired amountIn. This would result in an attempt to transfer amountIn to the next pair and reverts:
	
	(address tokenIn, uint256 actualAmountIn) = _transfer(route, tokenInId, from, Flags.callback(flags) ? address(this) : pair, amountIn);

Recommendations	A solution would be to refactor the swapExactOut flow, using only the return value from the previous swap. However, that will become difficult for exotic routes with different percentages.
Comments / Resolution	Acknowledged.

Issue_23	swapExactOut will not revert with last token as transfer-tax token
Severity	**Low**
Description	Whenever users execute swaps via the swapExactOut flow, the swap would not revert even if the last token is a transfer-tax token. This is due to the fact that it is anticipated that amountOut is received. However, due to the tax, the output amount will be lower than expected
Recommendations	Consider communicating this scenario openly.
Comments / Resolution	Acknowledged.

Issue_24	Transfer-tax path does not work if first swap is UniV3 swap
Severity	**Informational**
Description	During swapExactIn, it is allowed that the first token is a transfer-tax token. This will however not work if the first swap is being done using UNIV3, which is due to the fact that the callback results in less tokens being received by the pair.
Recommendations	We do not recommend a change.
Comments / Resolution	Acknowledged.

Issue_25	Redundant actualAmountIn practice during swapExactOut
Severity	**Informational**
Description	The swapExactOut callpath does not allow for transfer-tax tokens, which is inherently limited by the way how the input amount is calculated. The check is present as follows: *if (PackedRoute.isTransferTax(route)) revert RouterLogic__TransferTaxNotSupported();* However, the _swapExactOutSingle function erroneously defines the amountIn return value as actualAmountIn, which is technically incorrect (as there is no before/after check).
Recommendations	Consider renaming the return value to amountIn.
Comments / Resolution	Acknowledged.

Issue_26	Redundant payable keyword for swapExactOut
Severity	**Informational**
Description	The swapExactOut function incorporates the payable keyword, which is however never used during the normal business logic.
Recommendations	Consider removing the payable keyword.
Comments / Resolution	Resolved.