BAIL security

Smardex
Router

FINAL REPORT

March ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

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

Project	Smardex Router
Website	smardex.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SmarDex-Ecosystem/universal-router/tree/782967e51eb92c1cfe1d83e9412acaa08528e76a/src
Resolution 1	https://github.com/SmarDex-Ecosystem/universal-router/tree/2b9465ba87c23654c2eacce2088f5794ea08a006

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	1	1			
Medium	2			2	
Low	1	1			
Informational	11	5		6	
Governance	1			1	
Total	16	7		9	

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

Base

Dispatcher

The Dispatcher contract is an abstract contract that serves as a central hub for executing different operations based on predefined commands and is heavily inspired by Uniswap's dispatcher contract with additional callpaths:

https://github.com/Uniswap/universal-router/blob/main/contracts/base/Dispatcher.sol

It is inherited by the UniversalRouter and invoked within the execute function

To accomplish all different callpaths, the following logic contracts are used via inheritance:

a) Payments: Facilitates all different token transfers and ETH wrapping/unwrapping.

b) Sweep: Facilitates sweeping of ERC20 tokens and native ETH

c) V2SwapRouter: Facilitates swapping on UniswapV2

d) V3SwapRouter: Facilitates swapping on UniswapV3

e) UsdnProtocolRouter: Facilitates USDN entry operations

f) LidoRouter: Facilitates stETH operations (wrapping & unwrapping)

g) SmardexSwapRouter: Facilitates swapping on Smardex

Additionally, there are callpaths which allow for:

- Permit2 transfers
- Sweeping balances
- Standard transfers
- Permit calls + transferFrom

Important: The SWEEP operation should be executed always at the end of a call to ensure that no leftover funds remain in the router contract. Any leftover funds can/will be immediately swept out by bots.

Appendix: Callpath Determination (Gas Optimization)

The dispatch function allows for the determined execution of operations based on provided commands. To ensure this will be handled in a gas-efficient approach, certain commands are only checked below their corresponding boundaries.

a) Boundary 1: Below 0x08

-> V3_SWAP_EXACT_IN
-> V3_SWAP_EXACT_OUT
-> PERMIT2_TRANSFER_FROM
-> PERMIT2_PERMIT_BATCH
-> SWEEP
-> TRANSFER
-> PAY_PORTION

b) Boundary 2: Below 0x10 and above/equal 0x08

-> V2_SWAP_EXACT_IN
-> V2_SWAP_EXACT_OUT
-> PERMIT2_PERMIT
-> WRAP_ETH
-> UNWRAP_ETH
-> PERMIT2_TRANSFER_FROM_BATCH
-> PERMIT

c) Boundary 3: Below 0x1a and above/equal 0x10

-> INITIATE_DEPOSIT
-> INITIATE_WITHDRAWAL
-> INITIATE_OPEN
-> VALIDATE_DEPOSIT
-> VALIDATE_OPEN
-> VALIDATE_CLOSE
-> LIQUIDATE

-> VALIDATE_PENDING
-> REBALANCER_INITIATE_DEPOSIT

d) Boundary 4: Below 0x20 and above/equal 0x1a

-> WRAP_USDN
-> UNWRAP_USDN
-> WRAP_STETH
-> UNWRAP_STETH

e) Boundary 5: Above/equal 0x20

-> SMARDEX_SWAP_EXACT_IN
-> SMARDEX_SWAP_EXACT_OUT

We will highlight the gas optimization using a simple example:

If a user wants to use the V2_SWAP_EXACT_IN command, it first checks if the command is in boundary 5, then it checks if it is in boundary 4, etc. Until the desired boundary is found. This approach ensures that there is no need to if-check all specific commands until the desired command is found. It is simply a more efficient approach to reach the target command.

Appendix: Permit2Payments

The Dispatcher inherits several contracts from Uniswap, whereas many of these contracts inherit the Permit2Payments library:

https://github.com/Uniswap/universal-router/blob/main/contracts/modules/Permit2Payments.sol

This library leverages the AllowanceTransfer contract to transfer tokens from users to recipients:

https://github.com/Uniswap/permit2/blob/main/src/AllowanceTransfer.sol#L43

This works as follows:

a) User signs an off-chain message

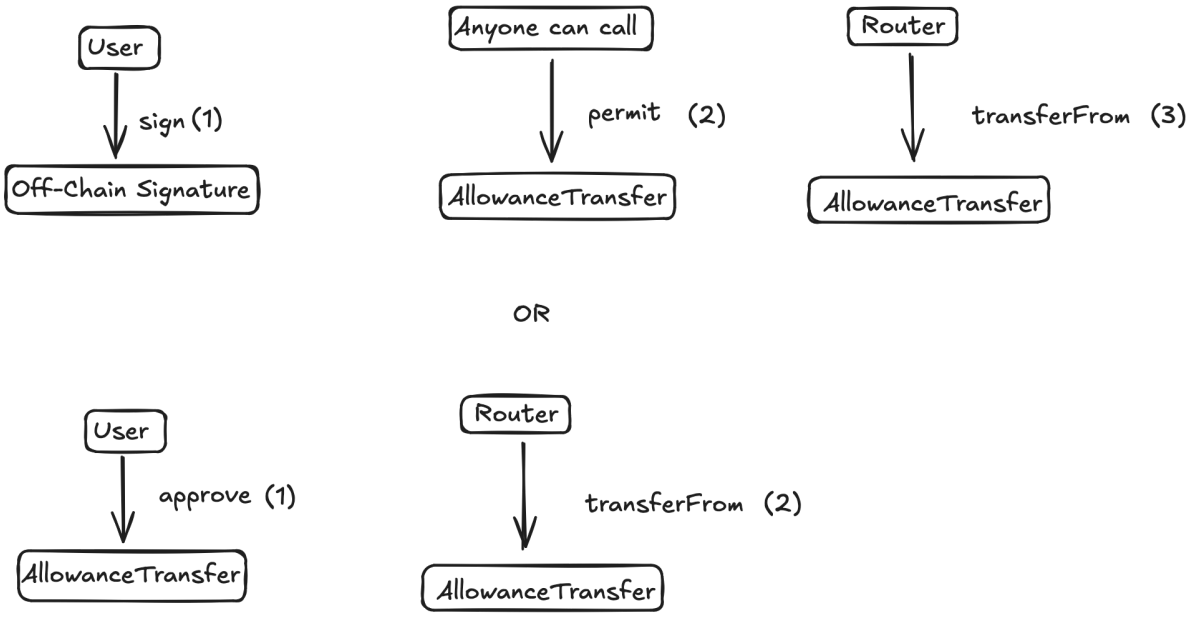
b) User invokes the permit function which allows the router to transfer funds on the user's behalf

c) Router invokes transferFrom which transfers funds from an allowed user to a recipient

OR

a) User invokes approve to allow the router to spend funds from a user's wallet

b) Router invokes transferFrom which transfers funds from an allowed user to a recipient



Privileged Functions

- none

Issue_01	PERMIT_TRANSFER_FROM operation can be exploited to maliciously consume approvals
Severity	High
Description	The PERMIT_TRANSFER_FROM operation allows the following flow: a) Invoke the permit function on an ERC20Permit token (DAI etc) for a corresponding message which has been signed. b) Consume the granted approval in the same transaction via executing a transferFrom That whole flow is built on the assumption that spender = address(this), as otherwise, the transferFrom would not be allowed to be executed (because the spender must have the approval). In the same turn, this will inherently result in the transferFrom transferring token to address(this), because spender = address(this). There are two ways to exploit this flow: a) Exploit a signed message by invoking PERMIT_TRANSFER_FROM with the owner being the signer to steal his approval and transfer tokens from the victim/owner to the router, these can then be immediately swept. (one would need to do this immediately after an address has signed the corresponding message, the most trivial attack flow is to just inspect a to-be-executed transaction, take the parameters and frontrun it with higher gas) b) Sophisticated: Steal all granted approvals to the contract. Due to the fact that the permit call is not needed to succeed, a user can simply do a PERMIT_TRANSFER_FROM operation with any owner address and the spender being the user. The permit call will revert but that's not an issue because the success is not required. Tokens

	are being directly transferred from the owner to the exploiter. Apparently, the Smardex team was already aware of this issue before we reported it to them.
Recommendations	This issue could be trivial fixed by implementing additional validations. However, at Bailsec, we are of the opinion that the fundamental of a secure smart contract is to limit user possibilities to the most reasonable extent. This will ensure that the attack-vectors are already inherently limited. This applies also to the PERMIT_TRANSFER_FROM operation as this operation is not necessarily needed for a functioning router. Therefore, we highly encourage simply removing this function.
Comments / Resolution	Resolved.

Issue_02	Griefing possibility via Permit2 usage
Severity	**Informational**
Description	Due to the known permit DoS issue, an attacker can DoS the execute call flow if any permit related operation is involved. There are exactly four instances of permit calls: a) PERMIT2_PERMIT_BATCH b) PERMIT2_PERMIT c) PERMIT d) PERMIT_TRANSFER_FROM From these 4 scenarios, only the latter two scenarios are low-level calls with an optional success_ return value. The first two calls are standard function calls which can revert the whole execution.
Recommendations	Since the same issue is existing in Uniswap, we do not recommend a change but rather recommend sticking to battle-tested code. However, this should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_03	WRAP_USDN command is non-sequential
Severity	**Informational**
Description	The WRAP_USDN callpath is handled between the 4th and 3rd boundary (smaller than 32 and large/equal 26), which means it would ideally start with 26 (0x1a). However, it starts with 27 (0x1b), leaving 0x1a unallocated. Compared to all other commands, this is non-sequential and exposes an inconsistency
Recommendations	We do not recommend a change. However, this should be kept in mind.

Comments / Resolution	Acknowledged.

Issue_04	dispatch function contains multiple unreachable conditions
Severity	**Informational**
Description	The dispatch function enters conditions based on the provided commandType parameter. There are several unreachable conditions, such as the following example: *} else if (command == Commands.PERMIT_TRANSFER_FROM) {* */** *equivalent: abi.decode(* *inputs, (* *address,* *address,* *address,* *uint256,* *uint256,* *uint8,* *bytes32,* *bytes32* *)* *)* **/* *address token;* *address owner;* *address spender;* *uint 256 amount;* *uint256 deadline;* *uint8 v;* *bytes32 r;* *bytes32 s;* *assembly {*

	token := calldataload(inputs.offset) *owner := calldataload(add(inputs.offset, 0x20))* *spender := calldataload(add(inputs.offset, 0x40))* *amount := calldataload(add(inputs.offset, 0x60))* *deadline := calldataload(add(inputs.offset, 0x80))* *v := calldataload(add(inputs.offset, 0xa0))* *r := calldataload(add(inputs.offset, 0xc0))* *s := calldataload(add(inputs.offset, 0xe0))* *}* *(success_, output_) = token.call(* *abi.encodeWithSelector(* *IERC20Permit.permit.selector, owner, spender, amount, deadline, v, r, s* *)* *);* *// slither-disable-next-line unchecked-transfer,arbitrary-send-erc20* *IERC20(token).transferFrom(owner, spender, amount);* *} else {* *revert InvalidCommandType(command);* *}* The highlighted else-branch is never reachable because the outer else condition is only triggered whenever command is smaller than 16 and larger/equal 8. However, all these conditions are already explicitly checked for which means there is no scenario where this else condition is ever triggered.
Recommendations	We do not recommend a change. However, this should be kept in mind.
Comments / Resolution	Acknowledged.

RouterImmutables

The RouterImmutables contract is a simple helper contract that keeps track of the RouterParameters struct which is used within the constructor of the UniversalRouter contract:

- permit2: AllowanceTransfer contract: https://github.com/Uniswap/permit2/blob/main/src/AllowanceTransfer.sol address
- weth9: WETH wrapper contract address
- v2Factory: UniswapV2 factory contract address
- v3Factory: UniswapV3 factory contract address
- pairInitCodeHash: Hash of creation bytecode from UniswapV2Pair contract
- poolInitCodeHash: Hash of creation bytecode from UniswapV3Pool contract
- usdnProtocol: USDN protocol address
- wstEth: wstETH address
- WUSDN: WUSDN address
- smardexFactory: Smardex factory contract address

Privileged Functions

- none

No issues found.

Libraries

BytesLib

The BytesLib library is a simple utility library that provides manipulation of byte arrays via slicing and address conversion. It is heavily inspired from the following library:

https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/BytesLib.sol

Slicing is exclusively used within the Path library to get the first pool from a path or to remove the first pool (as bytes type) from a path (for multi-hop swaps), similar to address conversion which is also exclusively within the Path library to fetch token addresses from the path array.

Privileged Functions

- none

No issues found

Commands

The Commands library is a simple helper library that exposes different commands for different operations. Commands are allowed to be between 0 (0x00) and 63 (0x3f). However, currently the largest command is only 33 (0x21).

It is used within the Dispatcher contract to determine which operation should be executed based on the provided input.

Appendix: COMMAND_TYPE_MASK & FLAG_ALLOW_REVERT usage

Both aforementioned bytes1 data types represent bitmasks which are used to isolate a command type from a provided bytes1 data:

COMMAND_TYPE_MASK = 0x3f (00111111)

FLAG_ALLOW_REVERT = 0x80 (10000000)

If for example one would like to do a simple TRANSFER while allowing for reverts, the command 0x85 (10000101) can be crafted and checked as follows:

```
bytes1 command = commandType & COMMAND_TYPE_MASK;

bool allowRevert = (commandType & FLAG_ALLOW_REVERT) != 0;
```

This simply extracts 00000101 (0x05) and 10000000 (0x80) from 10000101, resulting in a transfer which does not require the call to succeed (accepts revert).

Privileged Functions

- none

No issues found

Path

The Path library is a helper contract which is used within SmardexSwapRouterLib. It offers a set of utility functions to encode, decode, and manipulate byte arrays that represent swap paths and is inspired by the following contract:

https://github.com/Uniswap/v3-periphery/blob/main/contracts/libraries/Path.sol

This library is exclusively used within the SmardexSwapRouterLib library

Privileged Functions

- none

No issues found.

SmardexSwapRouterLib

The SmardexSwapRouterLib library exposes the logic for facilitating token swaps via the Smardex core protocol. It is solely used by the SmardexSwapRouter contract following the dispatch flow.

Users can either swap tokens by providing an exact input amount or by providing an exact output amount. The core logic is almost similar to the battle-tested SmardexRouter, with the additional implementation of permit2 usage:

https://github.com/SmarDex-Dev/smart-contracts/blob/main/contracts/periphery/SmardexRouter.sol

Appendix: Swap Scenarios (simple)

As already explained, this library facilitates swapping either by determining the exact input amount or the exact output amount. One can consider the following examples based on the WETH/USDC pair and the provided paths:

a) smardexSwapExactInput; path = [WETH/USDC]

b) smardexSwapExactInput; path = [USDC/WETH]

c) smardexSwapExactOutput; path = [WETH/USDC]

d) smardexSwapExactOutput; path = [USDC/WETH]

All mentioned scenarios have been formally verified.

Appendix: Multi-Hop swaps

In addition to simple swaps, this library also allows for multi-hop swaps which include more than one liquidity pair. This can be useful if there is no pair existing for a desired swap path. Consider the scenario where a user wants to swap WETH to USDT but only the following pairs are existing:

a) WETH/USDC

b) USDC/USDT

To facilitate this swap, the following path can be provided: [WETH; USDC; USDT]. This can also either be done by using the smardexSwapExactInput or smardexSwapExactOutput functions.

Using the first function, a user can determine the desired input amount of WETH, using the second function, a user can determine the desired output amount of USDT.

The flow for smardexSwapExactInput is as follows:

a) Determine the desired input amount of WETH

b) Invoke the swap on WETH/USDC and receive the USDC to the router contract while providing WETH to the pair

c) Invoke swap on USDC/USDT by providing the just received USDC to the pair and receive USDT to the determined recipient

The flow for smardexSwapExactOutput is as follows:

a) Determine the desired amount of USDT

b) Invoke swap on the USDC/USDT pair

c) Transfer USDT to the recipient

d) Invoke the callback function on the router contract to get the USDC amount

e) Due to the fact that there is currently no USDC in the router contract, and the payer wants to pay with WETH, a special flow is crafted:

1) Invoke _swapExactOut using the WETH/USDC pair
2) Transfer USDC to the USDC/USDT pair to fulfill the need of d)
3) Transfer the needed WETH amount for the corresponding USDC amount from the payee to the WETH/USDC pair

Using this flow, it is ensured that during d) the USDC/USDT pair has received USDC for the USDT which has been transferred out.

f) Swap has been fully executed

Both scenarios have been formally verified.

Privileged Functions

- none

Issue_05	Callback logic does not work with transfer-tax tokens
Severity	**Medium**
Description	The smardexSwapCallback function is invoked whenever a swap is executed and the desired token is being transferred to the caller (pair): *_payOrPermit2Transfer(permit2, tokenIn, decodedData.payer, msg.sender, amountToPay);* This will never work if tokenIn is a token with a transfer-tax as the balanceOf check within the pair always reverts due to an insufficient balance in the pair (because the tax is deducted): *require(* *_balanceInBefore + feeToAmount0 + (amount0_).toUint256() <= _params.balanceIn,* *"SmarDex: INSUFFICIENT_TOKEN0_INPUT_AMOUNT"* *);*
Recommendations	Consider if it is desired to trade FOT tokens on Smardex, if yes, this logic needs to be refactored.
Comments / Resolution	Acknowledged.

Issue_06	Missing slippage check for smardexSwapExactOutput
Severity	**Informational**
Description	The smardexSwapExactOutput function has, contrary to the smardexSwapExactInput function, no slippage check for the input amount. This issue is only present if the SmardexSwapRouterLib is used in another context, as in the current architecture it is solely invoked by the SmardexSwapRouter contract which checks for slippage.
Recommendations	Consider only using the SmardexSwapRouterLib together with the correct SmardexSwapRouter.
Comments / Resolution	Acknowledged.

Modules

Modules are these contracts which are used within the Dispatcher contract to execute operations. While the Dispatcher inherits several modules, only these below mentioned are explicitly in scope.

Lido

LidoImmutables

The LidoImmutables contract exposes immutable variables which are used by the LidoRouter, namely WSTETH and STETH, which are set in the constructor.

It is inherited by the LidoRouter contract.

Privileged Functions

- none

No issues found.

LidoRouter

The LidoRouter contract facilitates the wrapping of stETH into wstETH and the unwrapping of wstETH to stETH. It is inherited by the Dispatcher contract.

Appendix: Lido wstETH

WSTETH (Wrapped Staked Ether) is a tokenized representation of staked Ether (stETH) provided by Lido Finance.

The WSTETH contract is deployed at 0x7f39c581f595b53c5cb19bd0b3f8da6c935e2ca0 and allows users to wrap their stETH into wstETH and unwrap their wstETH to stETH via wrap() and unwrap(). While holding stETH results in a steady balance increase due to rebases, holding wstETH will result in an underlying stETH balance increase while keeping the wstETH balance consistent.

Privileged Functions

- none

Issue_07	stETH corner case can result in 1-2 wei of dust after stETH transfers
Severity	Informational
Description	stETH transfers can often result in 1-2 wei leftover balance after a transfer, which is owed due to the division operation. More information can be found here: https://docs.lido.fi/guides/lido-tokens-integration-guide/#1-2-wei-corner-case
Recommendations	We do not recommend a change to the code but rather acknowledging the issue that there may be some dust leftover in the router even after sweeping.
Comments / Resolution	Acknowledged.

Smardex

SmardexImmutables

The SmardexImmutables contract exposes immutable variables which are used by the SmardexSwapRouter, namely SMARDEX_FACTORY, WETH and SMARDEX_PERMIT2, which are set in the constructor.

It is inherited by the SmardexSwapRouter contract.

Privileged Functions

- none

Issue_08	Governance Privilege: Setting of malicious SMARDEX_FACTORY address allows for stealing PERMIT2 approvals
Severity	**Governance**
Description	During deployment, the SMARDEX_FACTORY address is set (immutable). If this address is a malicious address, one can invoke the callback function to steal all PERMIT2 approvals from all users to the router.
Recommendations	We do not recommend a change, one should simply ensure that SMARDEX_FACTORY is set to the legit smardex factory.
Comments / Resolution	Acknowledged. The same counts for the USDN_PROTOCOL address.

SmardexSwapRouter

The SmardexSwapRouter contract facilitates swapping via the Smardex protocol by leveraging the SmardexSwapRouterLib library which exposes all necessary mechanisms. It is inherited by the Dispatcher contract.

Privileged Functions

- none

No issues found.

UniswapV2

V2SwapRouter

The V2SwapRouter contract facilitates swapping via UniswapV2. It allows for providing an exact amount of tokens or requesting an exact output amount of tokens. (v2SwapExactInput / v2SwapExactOutput) and is inspired by Uniswap's implementation:

https://github.com/Uniswap/universal-router/blob/main/contracts/modules/uniswap/v2/V2SwapRouter.sol

Appendix: Exact Input

The v2SwapExactInput function allows for providing an exact amount of inputToken and receiving the corresponding amount of outputToken.

The flow for single swaps is as follows:

a) Transfer inputToken to the corresponding pool which is derived from the path

b) Invoke _v2Swap which calculates the corresponding amountOutput by incorporating the token sorting mechanism and the reserves.

c) Invoke swap on the corresponding pair with the desired recipient

The flow for multi-hop swaps is as follows:

d) Transfer inputToken to the corresponding pool which is derived from the path

e) Invoke _v2Swap which calculates the corresponding amountOutput by incorporating the token sorting mechanism and the reserves.

f) Fetch the next pair from the multi-hop path

g) Invoke swap on the pair with recipient = next pair

h) Execute the same calculation using the next pair's reserve and invoke swap with recipient being the desired target address

Appendix: Exact Output

The v2SwapExactOutput function allows for providing an exact amount of outputToken which is desired and transfers the corresponding needed amount of inputToken from the user in.

The flow for single swaps is as follows:

a) Invoke UniswapV2Library.getAmountInMultiHop to determine the pair and needed amountIn for the desired outputAmount

b) Transfer amountIn to the corresponding pair

c) Invoke _v2Swap which calculates amountOut from the provided input amount

d) Invoke swap on the pair with the calculated amountOutput

The flow for multi swaps is as follows:

e) Invoke UniswapV2Library.getAmountInMultiHop to determine the initial pair and needed amountIn for the desired outputAmount. The needed amountIn is denominated in the very first token of the path, using reverse calculations

f) Transfer amountIn to the corresponding pair

g) Invoke _v2Swap which calculates amountOut from the provided input amount and executes swaps until the final pair is reached which then transfers tokens to the recipient

Privileged Functions

- none

Issue_09	Slippage check during v2SwapExactOutput is void and users will lose funds in case of tokens with a transfer-tax
Severity	Medium
Description	The core flow of the v2SwapExactOutput function is as follows: a) Calculate needed amountIn for desired output amount using: https://github.com/Uniswap/universal-router/blob/main/contracts/modules/uniswap/v2/UniswapV2Library.sol#L125 b) Transfer needed amountIn to the pair c) Calculate the received outputAmount based on the received balance in the pair d) Invoke swap with the calculated output amount Since c) calculates amountOutput based on the received amount of tokens in the pair, amountOutput will be smaller than the actual provided amountOut parameter (because the pair has received less tokens due to the transfer-tax). Many other routers would just revert in that scenario, because they do not explicitly support transfer-tax tokens. However, since this router is meant to be compatible with transfer-tax tokens, it does not revert. This edge-case is not incorporated in the slippage check because the slippage check is only happening in the first place on amountIn. This means while the provided amountInMaximum is not exceeded, users will never get their desired amountOut. Furthermore, it is clear that this function is supposed to be working flawlessly for transfer-tax tokens as it uses the received balance to calculate the output amount. Illustrated:

	a) Charles wants to swap 100 FOT to 100 USDC (current price = 1) b) Charles invokes swapExactOut with amountOut = 100e18 and amountInMax = 100e18 c) amountIn = 100e18 and will be transferred to the pair d) The pair has only received 90e18 due to the tax e) The calculation results in 90e18 USDC as output amount The exact output amount is effectively undercut.
Recommendations	Since the exact same issue is present in Uniswap as well, we suggest simply limiting these kinds of transactions via frontend/API.
Comments / Resolution	Acknowledged.

USDN

UsdnProtocolImmutables

The UsdnProtocolImmutables contract exposes essential immutable variables to interact with the USDN protocol, namely:

USDN_PROTOCOL: The address of the USDN protocol

PROTOCOL_ASSET: The wstETH address

SDEX: The address of the SDEX token

USDN: The address of the USDN token

WUSDN: The address of the WUSDN token

These variables are set upon contract deployment and the contract is inherited by the UsdnProtocolRouter

Privileged Functions

- none

No issues found

UsdnProtocolRouter

The UsdnProtocolRouter contract exposes all necessary logic to interact with the USDN protocol. It allows for the following interactions:

a) _usdnInitiateDeposit: Initiate a wstETH deposit to the USDN vault protocol

b) _usdnValidateDeposit: Validate an initiated and executable deposit

c) _usdnInitiateWIthdrawal: Initiate a wstETH withdrawal from the USDN vault protocol

d) _usdnValidateWithdrawal: Validate an initiated and executable withdrawal

e) _usdnInitiateOpenPosition: Initiate a leveraged long position opening on the USDN protocol

f) _usdnValidateOpenPosition: Validate an initiated and executable long position opening

g) _usdnValidateClosePosition : Validate an initiated and executable position closure

h) _wrapUsdnShares: Wrap USDN into WUSDN

i) _unwrapUsdn: Unwrap WUSDN to USDN

j) _usdnLiquidate: Execute liquidations in the USDN protocol

k) _rebalancerInitiateDeposit: Initiate a rebalancer deposit

All other operations such as closing positions and transferring position ownership must be direct calls from the position owner to the USDN protocol.

Privileged Functions

- none

Issue_10	Usage of liquidate selector is outdated
Severity	**Low**
Description	The _usdnLiquidate function allows users to liquidate unhealthy long positions on the USDN protocol. The provided iterations parameter determines how many ticks should be liquidated. However, due to a bug which we have reported during the USDN core audit, the liquidation flow will be refactored, not allowing for an iterations parameter.
Recommendations	Consider adjusting this implementation accordingly.
Comments / Resolution	Resolved.

Issue_11	Selector mismatch for initiateDepositAssets
Severity	**Informational**
Description	The _rebalancerInitiateDeposit function allows users to initiate a deposit into the rebalancer. This call uses an incorrect uint type: (success_, data_) = rebalancerAddress.call(abi.encodeWithSelector(IRebalancer.initiateDepositAssets.selector, uint80(amount), to)); The correct uint type is uint88: *function initiateDepositAssets(uint88 amount, address to) external*
Recommendations	Consider adjusting the casting to uint88.
Comments / Resolution	Resolved.

Issue_12	Lack of automatic SDEX refund upon deposit initiations
Severity	**Informational**
Description	The amount of SDEX to be burned can change between the transaction submission and the actual execution, therefore, it makes sense to provide some more SDEX to the contract than is actually burned. (This is handled by the different commands, for SDEX via PERMIT transfers). In such a scenario, there will often be some SDEX remaining after the execution. This SDEX must be swept afterwards in the same transaction.
Recommendations	This issue is just a reminder for the team to carefully craft the frontend execution logic. The team confirmed that they are aware of it.
Comments / Resolution	Resolved. This will be handled properly in the frontend. This is furthermore a design choice to allow for potential SDEX swaps with the leftover amount, post-deposit.

Issue_13	Sudden rebase may result in less USDN shares being transferred to the contract than anticipated
Severity	**Informational**
Description	The _usdnInitiateWithdrawal function expects the USDN to be already sitting in the router before this function is invoked. This is usually done via the PERMIT2.transferFrom. This flow exposes a small issue in the scenario where a rebase happens after the execute function has been invoked (or after a rebase has been triggered by a USDN protocol interaction from a previous command in the same transaction). In such a case the router contract will receive less USDN shares than anticipated because the nominal transfer amount is the same as before the rebase and transfers to the router never happen via the transferShares function.

Recommendations	We do not recommend a change because this does not expose a real issue, in the worst case a user can simply execute another withdrawal (to withdraw the remaining shares in the wallet).
Comments / Resolution	Acknowledged.

Issue_14	Inconsistency in approval handling for USDN operations
Severity	**Informational**
Description	The approve function on the USDN token is called on the following occasions: a) _usdnInitiateWithdrawal: *USDN.approve(address(USDN_PROTOCOL), USDN.convertToTokensRoundUp(amount));* b) _wrapUsdnShares: *USDN.forceApprove(address(WUSDN), type(uint256).max);* As one can see, the approval handling is inconsistent as one time forceApprove is used and the other time approve is used.
Recommendations	There is no harm being done here, however, we recommend sticking to one method and staying consistent
Comments / Resolution	Resolved, this has been refactored.

Issue_15	Optional permit2TokenBitfield logic will never work
Severity	Informational
Description	Certain functionalities like initiateDeposit allow for a permit2TokenBitfield parameter that determines which tokens are meant to be transferred via permit2: *@param permit2TokenBitfield The bitfield indicating which tokens should be used with permit2* This will never work because the msg.sender is always the router and there are no approvals granted from the router towards PERMIT2.
Recommendations	We do not recommend a change, instead it should be ensured that the frontend just uses the correct setting.
Comments / Resolution	Resolved, this has been refactored.

Sweep

The Sweep contract is a simple helper contract that facilitates sweeping of ETH and ERC20 tokens via the sweep function to a determined recipient.

Issue_16	Lack of address(0) check for ETH sweeping
Severity	Informational
Description	In the scenario where the recipient is address(0), all ETH which is swept would be permanently lost.
Recommendations	Consider implementing the aforementioned check.
Comments / Resolution	Resolved.

Router

UniversalRouter

The UniversalRouter contract is the entry contract for users to facilitate all different operations as it inherits all contracts and exposes the execute function which allows users to provide commands and corresponding inputs. More information about commands can be found in the Commands section.

This contract is heavily inspired by Uniswap's UniversalRouter:

https://github.com/Uniswap/universal-router/blob/main/contracts/UniversalRouter.sol

Privileged Functions

- none

No issues found.

Smardex - Router - Round 2

1. Project Details

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

Project	Smardex - Router - Round 2
Website	smardex.io
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SmarDex-Ecosystem/universal-router/tree/2b9465ba87c23654c2eacce2088f5794ea08a006
Resolution 1	https://github.com/SmarDex-Ecosystem/universal-router/tree/c259940f1b58824b91239c561575099625f5b6c6

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High					
Medium	2	1		1	
Low					
Informational	8	2		6	
Governance	1			1	
Total	11	3		8	

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

Base

UniversalRouter

The UniversalRouter is similar to the already audited contract. For description we refer to the previous report above. The only change which has been made is the addition of the PERMIT2 address towards the USDNProtocolImmutables construction. The reason for that is the introduction of the callback functionality within the UsdnProtocolRouter.transferCallback function which uses the USDN_PROTOCOL_PERMIT2 address for potential PERMIT2 transfers.

Previous acknowledged issues will not be raised again.

No issues found.

Dispatcher

The Dispatcher contract is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) Introduction of library usage for:
   i) v2SwapExactInput
   ii) v2SwapExactOutput
   iii) USDN interactions
b) Removal of PERMIT_TRANSFER_FROM to fix previous bug
c) Introduction of TRANSFER_FROM
d) Refactoring of most USDN interactions
   i) INITIATE_DEPOSIT
   ii) INITIATE_WITHDRAWAL
   iii) INITIATE_OPEN
   iv) VALIDATE_DEPOSIT
   v) VALIDATE_WITHDRAWAL
   vi) VALIDATE_OPEN
   vii) VALIDATE_CLOSE
   viii) LIQUIDATE

ix) VALIDATE_PENDING
x) REBALANCER_INITIATE_DEPOSIT
xi) WRAP_USDN
xii) UNWRAP_USDN

e) Refactoring of stETH interactions
i) WRAP_STETH
ii) UNWRAP_STETH

f) Addition of new commands
i) TRANSFER_POSITION_OWNERSHIP
ii) REBALANCER_INITIATE_CLOSE
iii) USDN_TRANSFER_SHARES_FROM
iv) SMARDEX_ADD_LIQUIDITY
v) SMARDEX_REMOVE_LIQUIDITY

Previous acknowledged issues will not be raised again.

Issue_01	_mapSafe is not used consistent
Severity	**Medium**
Description	The _mapSafe function is a modified version of the map function which ensures that the recipient cannot be address(this). This function can be found within the LockAndMap contract. This logic is useful for various setups, including but not limited to the validator and the recipient of positions. Currently, it is used for the following USDN interactions: - INITIATE_DEPOSIT - INITIATE_WITHDRAWAL - INITIATE_OPEN - REBALANCER_INITIATE_CLOSE However, it is not used for: - INITIATE_CLOSE - REBALANCER_INITIATE_DEPOSIT
Recommendations	Consider staying consistent with the usage of _safeMap
Comments / Resolution	Resolved.

Issue_02	Inconsistency in high and low-level calls
Severity	**Informational**
Description	Throughout the contract, some calls are made via low-level calls and some calls are made via the normal interface. The idea behind low-level calls is to ignore reverts and thus prevent griefing attacks, for example during permit. However, this behaviour is not really consistent as some commands are made with a low-level call without a real griefing risk: - TRANSFER_FROM - INITIATE_CLOSE - REBALANCER_INITIATE_DEPOSIT - ...
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_03	Redundant success check for Smardex and Lido interactions
Severity	**Informational**
Description	The Lido interactions as well as Smardex interactions expose a high-level call with a success_ return value. While the success_ return value check of the Lido interaction can be at least considered as arguable, the success_ return value check for the Smardex interaction is pointless as it either returns true or directly reverts.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_04	Lack of output check during various USDN interactions
Severity	**Informational**
Description	During USDN interactions, it may be possible that for example no new deposit is initiated because there are currently to-liquidated positions. In that scenario, an early return is executed which is reflected in the return value. However, for most interactions the return value is not checked, resulting in a continuation of the command execution.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_05	Griefing possibility of chained commands
Severity	**Informational**
Description	The Dispatcher functionality targets to batch multiple different commands to be executed subsequently. For example a user might execute a swap followed by liquidity addition and a USDN deposit initiation. Due to the nature of the business logic it is possible to grief the swap or the liquidity addition by triggering the slippage checks such that the whole execution reverts.
Recommendations	There is no fix for this issue as it is inherently present due to the business logic.
Comments / Resolution	Acknowledged.

RouterImmutables

The RouterImmutables is similar to the already audited contract. For description we refer to the previous report above. No changes have been made.

Previous acknowledged issues will not be raised again.

No issues found.

Libraries

lido/LidoRouterLib

The LidoRouterLib is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) The contract has been compiled into a library
b) Immutable parameters are now provided as arguments by the Dispatcher
c) Custom amount for wrapping/unwrapping is allowed (instead of the full balance)
d) Before/After check during unwrapWSTETH
e) Transferring stETH via shares

Previous acknowledged issues will not be raised again.

No issues found.

uniswap/UniswapV2RouterLib

The UniswapV2RouterLib is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) The contract has been compiled to a library
b) Immutable variables are provided as parameters by the Dispatcher

Previous acknowledged issues will not be raised again.

No issues found.

usdn/PaymentLib

The PaymentLib contract is a simple helper library which is used by the UsdnProtocolRouterLib library. It is responsible for setting the transient storage slot with the corresponding PaymentType:

a) transfer
b) transferFrom
c) permit2

This slot is used during the initiateDeposit, initiateOpen and initiateWithdraw functions and simply defines the transfer logic for the callback functionality. At the end of the corresponding function calls, the transient storage slot is reset.

Previous acknowledged issues will not be raised again.

Privileged Functions

- none

No issues found

usdn/UsdnProtocolRouterLib

The UsdnProtocolRouterLib contract is the library contract which is used by the Dispatcher to facilitate the majority of USDN interactions.

The following interactions are handled:

a) usdnInitiateDeposit
b) usdnValidateDeposit
c) usdnInitiateWithdrawal
d) usdnValidateWithdrawal
e) usdnInitiateOpenPosition
f) usdnValidateOpenPosition
g) usdnValidateClosePosition
h) usdnValidateActionablePendingActions
i) wrapUSDNShares
j) unwrapUSDN
k) usdnLiquidate
l) rebalancerInitiateDeposit
m) rebalancerInitiateClosePosition

Moreover, the transferCallback and usdnTransferCallback functions are exposed which are called by the UsdnProtocolRouter contract upon the usdnInitiateDeposit, usdnInitiateOpenPosition and usdnInitiateWithdrawal functions. It simply handles the callback nature of the USDN protocol which requires wstETH, SDEX and USDN to be transferred in via callback to facilitate corresponding actions.

Privileged Functions

- none

Issue_06	Griefing possibility via INITIATE_CLOSE and INITIATE_REBALANCER_CLOSE
Severity	**Medium**
Description	The INITIATE_CLOSE and INITIATE_REBALANCER_CLOSE commands are callable by everyone as long as the owner has signed the transaction. The problem which has its root-cause in the underlying USDN protocol is that the validator address is not included in the signature check. This exposes a griefing vulnerability where the caller can provide his own validator address which rejects the ETH receipt upon the first phase, then shortly before the second phase starts, the ETH receipt can be enabled and the transaction can be validated, to make sure the griefer does not actually donate his securityDeposit. Following this practice, one can prevent closures for as long as the duration of the first phase is (until everyone can validate an action).
Recommendations	Consider acknowledging this issue as it has its root-cause in the underlying USDN protocol.
Comments / Resolution	Acknowledged.

Issue_07	Potential mismatch in PaymentsType due to SDEX and wstETH transfer
Severity	**Informational**
Description	During the initiateDeposit function, the USDN protocol transfers not only wstETH in but also SDEX in. This could result in mismatches if for example the caller chooses to use the PERMIT2 pattern but only wstETH is approved via PERMIT2 while SDEX may be desired as transferFrom.
Recommendations	Consider ensuring that both tokens are approved consistently.
Comments / Resolution	Acknowledged.

Issue_08	Redundant usage of forceApprove during usdnInitiateOpenPosition
Severity	**Informational**
Description	The usdnInitiateOpenPosition function approves the asset to the USDN protocol. This practice is redundant as the callback nature is used for transfers and no transferFrom is executed.
Recommendations	Consider removing the forceApprove call.
Comments / Resolution	Resolved.

Issue_09	Unused return value
Severity	**Informational**
Description	The UsdnProtocolRouterLib facilitates various interactions with the USDN protocol and upon these actions checks and returns the return value. This can be for example illustrated via the usdnValidateClosePosition function: *return outcome_ == IUsdnProtocolTypes.LongActionOutcome.Processed;* Many return values, including the above one are actually never used by the dispatcher: *UsdnProtocolRouterLib.usdnValidateClosePosition(* *USDN_PROTOCOL, map(validator), closePriceData, previousActionsData, ethAmount* *);*
Recommendations	Consider simply acknowledging this issue, as it does not expose any harm.
Comments / Resolution	Acknowledged.

Commands

The Commands library is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) Addition of new commands
b) Slight refactoring of sequences

Previous acknowledged issues will not be raised again.

No issues found.

TransientStorageLib

The TransientStorageLib contract is a simple helper library which is exclusively used by the PaymentLib contract and executes a tstore and tload which writes and fetches the transient storage slot for the TRANSIENT_PAYMENT_SLOT

No issues found

Modules

lido/LidoImmutables

The LidoImmutables contract is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) IERC20Metadata replaced with IStETH

Previous acknowledged issues will not be raised again.

No issues found

usdn/LockAndMap

The LockAndMap contract is a simple helper contract which inherits the LockAndMsgSender contract. It exposes the _safeMap function which is similar to the map function and uses constants for recipient determination but additionally ensures that recipient cannot be address(this).
This is particularly useful in context with the USDN protocol as this ensures that recipient in certain cases for example during deposit initiation cannot be address(this). The same counts for the validator.

No issues found.

usdn/UsdnProtocolImmutables

The UsdnProtocolImmutables contract is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) USDN_PROTOCOL_PERMIT2 has been implemented

Previous acknowledged issues will not be raised again.

Issue_10	Governance Issue: USDN_PROTOCOL variable
Severity	**Governance**
Description	Currently, the contract can be deployed with an immutable USDN_PROTOCOL variable which allows (in address is malicious) to abuse the callback functionality: if (msg.sender != usdnProtocol) { revert IUsdnProtocolRouterErrors.UsdnProtocolRouterInvalidSender()}
Recommendations	Consider ensuring that the USDN_PROTOCOL variable is correct upon deployment.
Comments / Resolution	Acknowledged.

Issue_11	Unused variable
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. - SDEX Furthermore, the USDN_PROTOCOL_PERMIT2 variable seems redundant as the protocol already defines a PERMIT2 variable which handles approvals.
Recommendations	Consider clarifying the usage of the aforementioned variable(s).
Comments / Resolution	Resolved, both variable have been removed and the already declared PERMIT2 variable is now used to substitute USDN_PROTOCL_PERMIT2

usdn/UsdnProtocolRouter

The USDNProtocolRouter contract is directly inherited by the Dispatcher and handles the callback mechanism. In the previous iteration of the router, funds were simply transferred towards the USDN protocol via a transferFrom. In this iteration, a callback mechanism is introduced which relies on the fact that the IPaymentCallback interface is exposed which automatically signals the USDN protocol to transfer in wstETH , SDEX and USDN via the callback mechanism.

There are two distinct callback functions:

a) transferCallback: Handles the transfer of wstETH and SDEX during initiateDeposit and wstETH during initiateOpenPosition
b) usdnTransferCallback: Handles the transfer of USDN during initiateWithdraw

The actual transfer logic is outsourced towards the UsdnProtocolRouterLib library which includes the caller validation, ensuring that only the USDN protocol can invoke the callback and funds can only be transferred from the lockedBy address which always must be the caller.

Privileged Functions

- none

No issues found

Sweep

The Sweep contract is similar to the already audited contract. For description we refer to the previous report above. The following changes have been made:

a) address(0) sanity check

Previous acknowledged issues will not be raised again.

No issues found.

utils/Payment

The Payment contract is a simple helper library which facilitates token transfers either via a direct transfer from the router or via a PERMIT2 transferFrom from the payer to a recipient. It is used within the UniswapV2RouterLib contract.

Privileged Functions

- none

No issues found.