STATE MIND

Valantis Swap Router

02-09-2024 – 09-09-2024

Table of contents

1. Project brief 2

2. Finding severity breakdown 3

3. Summary of findings 4

4. Conclusion 4

5. Findings report 5

Medium

The false routing assumption to steal accumulated tokens 5

Informational

GaslessSwapParams.feeRecipient can be zero 6

The refund logic inconsistency 6

Slippage check can be bypassed for some recipients 7

Possibility of transferring user assets to flagged addresses 7

Sanity limit for plain ether transfers 8

1. Project brief

Title	Description
Client	Valantis Labs
Project name	Valantis Swap Router
Timeline	02-09-2024 - 09-09-2024
Initial commit	bb4159a4a159599306ab5933911bf9ef45a009c6
Final commit	cac3f6dd625bf9ead3e1cf142d834f4374e16e98

Short Overview

Valantis Swap Router is a periphery smart contract facilitating interaction with Valantis pools. It supports swaps between any two tokens routed through Valantis Universal or Sovereign pools.

Additional features:

- Intent based swaps using Permit2 with configurable fee charge
- Batched swaps

Project Scope

The audit covered the following files:

ValantisSwapRouter.sol	ValantisSwapRouterStructs.sol	SignatureVerification.sol
GaslessSwap.sol	DirectSwap.sol	EIP712.sol
GaslessSwapIntentHash.sol	NonceBitmap.sol	

2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	1 (1 fixed, 0 acknowledged)
Informational	5 (2 fixed, 3 acknowledged)
Total	6 (3 fixed, 3 acknowledged)

4. Conclusion

During the audit of the codebase, 6 issues were found in total:

- 1 medium severity issue (1 fixed)
- 5 informational severity issues (2 fixed, 3 acknowledged)

The final reviewed commit is cac3f6dd625bf9ead3e1cf142d834f4374e16e98

5. Findings report

MEDIUM-01	The false routing assumption to steal accumulated tokens	Fixed at: cac3f6d

Description

Lines: ValantisSwapRouter.sol#L431-L444
The **_executeSwaps** iterates over the **address[] pools** array and executes swaps consecutively.
If **amountInSpecified[i] = 0**, the **amountIn** is equal to the value of the last swap **amountOutSwap**.

```
params.amountInSpecified[i] > 0 ? params.amountInSpecified[i] : amountOutSwap,
```

Here, the router assumes that the **tokenOut** of the last swap is the **tokenIn** of the next one.
This is not the case, as there are no restrictions on the **pool** input array, to ensure the routing correctness.
E.g. user can provide the pools **[(a->b), (c->d)]**.
The inconsistency above can be used to claim excess tokens without the **protocolManager** role:

1. The router has an excess amount **x** of **tokenC**.
2. The user provides the route, where **amountOutSwap = x** for the first swap **(tokenIn, tokenOut)**.
3. The second pool of the route will be **(tokenC, tokenOut)** to swap out the excess to the user's wallet.

Additionally, user can deploy their pool with arbitrary tokens to have full control in constructing the **amountOutSwap** value.
E.g:

1. The router has an excess amount x of the tokenC
2. A user deploys his own pool with (tokenX, tokenY) and has full control of those tokens supply.
3. A user configures the first swap to match the values **amountOutSwap = x** for (tokenX, tokenY) pool.
4. The second token of the second pool of the route can be anything valuable (tokenC, tokenD).

A user configures the first swap token output amount to match the current excess balance on the router and then swaps the excess to his wallet.
Impact: an arbitrary user can sweep excess tokens.

Recommendation

We recommend introducing the limitations to the provided swap paths. One way to ensure the path correctness is by checking the equality of **tokenOut** of the last pool to the **tokenIn** of the next one.

Client's comments

> We are fully removing support for rescuing tokens, thus any token accumulated can be sweeped by anyone using sweep function and Valantis will not be responsible about any token accidently sent to this contract.

INFORMATIONAL-01	GaslessSwapParams.feeRecipient can be zero	Fixed at: ef261e2

Description

Line: GaslessSwap.sol#L14

The gasless swap fee recipient lacks a zero address check.

Impact: lost fees for incorrectly configured swap.

Recommendation

We recommend adding a sanity check to the **checkGaslessSwapParams** function in the **GaslessSwap.sol** library.

```
if (gaslessSwapParams.feeRecipient == address(0)) {
  revert GaslessSwap__checkGaslessSwapParams_invalidFeeRecipient();
}
```

INFORMATIONAL-02	The refund logic inconsistency	Acknowledged

Description

Lines: ValantisSwapRouter.sol#L306-L309

At the end of the **swap** function, a user can receive all unspent ETH.

```
if (msg.value > 0) {
   // Refund unspent ETH to recipient
   _sendNativeToken(_directSwapParams.recipient, address(this).balance);
}
```

This is not the case for all the other tokens; therefore, leftover **tokenIn** will accumulate.

Recommendation

We recommend modifying the refund mechanism to include other input tokens.

Client's comments

Acknowledged it as bundling swap with sweep solves this issue

INFORMATIONAL-03	Slippage check can be bypassed for some recipients	Acknowledged

Description

Line: ValantisSwapRouter.sol#L302.
The **amountOutTotal** variable check was added to support contract recipients with deposit/withdraw functionality. However, the additional check can be bypassed too.
amountOutTotal is incremented when **tokenOutSwap == params.tokenOut**. Then a malicious solver can do the following:

1. **tokenIn -> tokenOut** with Router as the recipient, this satisfies **amountOutTotal < gaslessSwapParams.intent.amountOutMin** condition.
2. **tokenOut -> tokenIn** swap back with Router as the recipient.
3. **tokenIn -> tokenOut** swap through the malicious ALM which deposits to the intent recipient contract. This satisfies the **amountOut < gaslessSwapParams.intent.amountOutMin** in the recipient.
4. After the swap the solver calls the withdraw method in the recipient contract.

In the steps 2 and 3 it can be any token instead of **tokenIn**.
Impact: the user loses **tokenIn** amount.

Recommendation

We recommend to do all swaps with Router as recipient and check the slippage. Then transfer tokens to the intent recipient.

Client's comments

Won't fix as the premise of this finding requires user signing malicious signature.

INFORMATIONAL-04	Possibility of transferring user assets to flagged addresses	Fixed at: 8e0419e

Description

Line: ValantisSwapRouter.sol#L411
At the end of the gasless swap, user assets are transferred to pay the fees. The user selects and signs the token in which the fee will be paid and the fee value. The fee receiver is defined by the broadcaster (**authorizedSender**). In the case of a malicious broadcaster, the user's assets may be transferred to a flagged or sanctioned address, which will poison the user's address.

Recommendation

We recommend including the user's intent field with **feeReceiver** to have the ability to check the address.

Client's comments

We have remove feeRecipient and are now using authorized sender itself as feeRecipient

INFORMATIONAL-05	Sanity limit for plain ether transfers	Acknowledged

Description

Line: ValantisSwapRouter.sol#L542

In the function **_sendNativeToken()** ether from the contract is transferred to **recipient** using low-level **call** without any gas restrictions. According to EVM implementation if there is an ether transfer during **call**, then 2300 gas is automatically passed.

Recommendation

We recommend setting or zeroing the gas limit when making a **call** to **recipient**.

Client's comments

Won't fix as it will lead to failure of transaction with contracts with execution in their receive or fallback function

STATE
MIND