Mugen Security Review

Pashov Audit Group

Conducted by: pashov

January 19th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Mugen** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Mugen

The protocol is a DEX/swap adapter that allows complex transactions, for example multiple tokens in to multiple/single tokens out and vice versa on several DEXes (Uniswap, Sushiswap, 3xcalibur). The protocol integrates the native adapters provided by the DEXes to ensure flawless transfers of tokens. It also allows cross-chain swaps as it is integrated with LayerZero's Stargate protocol.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - 61564a3e1eac743cb9b89976cbefbcb0fd15f38f**

Scope

The following smart contracts were in scope of the audit:

- `ArbitrumSwaps`
- `StargateArbitrum`

7. Executive Summary

Over the course of the security review, pashov engaged with Mugen to review Mugen. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Mugen
Date	January 19th, 2023

Findings Count

Severity	**Amount**
High	2
Medium	1
Low	3
QA	10
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Anyone can use or steal ArbitrumSwaps native asset balance	High	Resolved
[H-02]	Malicious user can easily make the protocol revert on every USDT swap on Uniswap	High	Resolved
[M-01]	Use quoteLayerZeroFee instead of sending all native asset balance as gas fee for swap call	Medium	Resolved
[L-01]	Check array arguments have the same length	Low	Resolved
[L-02]	A require check can easily be bypassed	Low	Resolved
[L-03]	The gasLeft() after gas-limited external call might not be enough to complete the transaction	Low	Resolved
[QA-01]	Prefer battle-tested code over reimplementing common patterns	QA	Resolved
[QA-02]	Use an enum for the "step" types in ArbitrumSwaps	QA	Resolved
[QA-03]	Move code to bring cohesion up	QA	Resolved
[QA-04]	Use x != 0 to get positive-only uint values	QA	Resolved
[QA-05]	Remove not needed custom error	QA	Resolved
[QA-06]	Solidity safe pragma best practices are not used	QA	Resolved
[QA-07]	External method missing a NatSpec	QA	Resolved
[QA-08]	Mismatch between contract and file names	QA	Resolved
[QA-09]	Missing override keyword	QA	Resolved

[QA-10]	Typos in comments	QA	Resolved

8. Findings

8.1. High Findings

[H-01] Anyone can use or steal `ArbitrumSwaps` native asset balance

Severity

Likelihood: High, because this can easily be noticed and exploited

Impact: Medium, because value can be stolen, but it should be limited to gas refunds

Description

An attacker can steal the `ArbitrumSwaps` native asset balance by doing a call to the `arbitrumSwaps` method with steps `WETH_DEPOSIT` and `WETH_WITHDRAW` - this will send over the whole contract balance to a caller-supplied address. This shouldn't be a problem, because the contract is a "swap router" and is not expected to hold any native asset balance at any time. Well this assumption does not hold, because in the `stargateSwap` method the `_refundAddress` argument of the `swap` method call to the `stargateRouter` is `address(this)`. This means that all of the native asset that is refunded will be held by the `ArbitrumSwaps` contract and an attacker can back-run this refund and steal the balance.

Recommendations

The refund address should be `msg.sender` and not `address(this)`. This way the protocol won't be expected to receive native assets, so they can be stolen only if someone mistakenly sends them to the `ArbitrumSwaps` contract which is an expected risk.

[H-02] Malicious user can easily make the protocol revert on every `USDT` swap on Uniswap

Severity

Likelihood: High, attack can easily be done and it exploits a well-known attack vector of `USDT`

Impact: Medium, because the protocol will not work with only one ERC20 token, but it is a widely used one

Description

A malicious user can get the `ArbitrumSwaps` contract to revert on each `USDT` swap on Uniswap, because of a well-known attack vector of the token implementation. The problem is in the following code from `UniswapAdapter.sol` and is present in both `swapExactInputSingle` and `swapExactInputMultihop`

```
TransferHelper.safeApprove(
            swapParams.token1, address(swapRouter), IERC20
              (swapParams.token1).balanceOf(address(this))
        );
```

Here is how the attack can be done:

1. Malicious user transfers manually 1 wei of USDT to `ArbitrumSwaps`
2. Now he calls `ArbitrumSwaps::arbitrumSwaps` with `step == UNI_SINGLE` to swap 1 USDT
3. Now `swapExactInputSingle` approves 1 + 1e-18 USDT, because of the `balanceOf` call
4. Transaction will complete successfully, but next time anyone wants to swap USDT on Uniswap the transaction will revert, because of USDT approval race condition - to do an `approve` call the allowance should be either 0 or `type(uint256).max`, which is not the case because allowance is 1 wei

Recommendation

Instead of using the `IERC20(multiParams.token1).balanceOf(address(this))` as the approved allowance, use the `amountIn` parameter.

8.2. Medium Findings

[M-01] Use `quoteLayerZeroFee` instead of sending all native asset balance as gas fee for `swap` call

Severity

Likelihood: High, because the wrong value will be sent always

Impact: Low, because the `swap` function has a gas refund mechanism

Description

Currently in `StargateArbitrum::stargateSwap` when doing a call to the `swap` method of `stargateRouter`, all of the contract's native asset balance is sent to it so it can be used to pay the gas fee. The Stargate docs show that there is a proper way to calculate the fee and it is by utilizing the `quoteLayerZeroFee` method of `stargateRouter`.

Recommendations

Follow the documentation to calculate the fee correctly instead of always sending the whole contract's balance as a fee, even though there is a refund mechanism.

8.3. Low Findings

[L-01] Check array arguments have the same length

In both `ArbitrumSwaps::arbitrumSwaps` and in `StargateArbitrum::stargateSwap` you have multiple array-type arguments. Validate in both places that the arguments have the same length so you do not get unexpected errors if they don't.

[L-02] A `require` check can easily be bypassed

In `StargateArbitrum::stargateSwap` we have the following code

```
if (msg.value <= 0) revert MustBeGt0();
```

This check can easily be bypassed by just sending 1 wei. Remove the check completely, since `msg.value` is not used in the method anyway.

[L-03] The `gasLeft()` after gas-limited external call might not be enough to complete the transaction

In `StargateArbitrum::sgReceive` we have the following piece of code

```
try IArbitrumSwaps(payable(address(this))).arbitrumSwaps{gas: 200000}
  (steps, data) {}
        catch (bytes memory) {
            IERC20(_token).safeTransfer(to, amountLD);
            failed = true;
        }
```

Now if the `arbitrumSwaps` call took up all of the gas it is possible that there is not enough gas left for the `safeTransfer` call, as well for the code below it. Consider a different approach, that will check `gasleft()` and make sure that there will be enough, something like in this method

8.4. QA Findings

[QA-01] Prefer battle-tested code over reimplementing common patterns

Replace the `locked` modifier in `ArbitrumSwaps` with the `nonReentrant` from OpenZeppelin, since it is well tested and optimized.

[QA-02] Use an enum for the "step" types in `ArbitrumSwaps`

Currently the step types are handled by constants that have an integer value and are not sequential (numbers 7 to 11 are missing). This is a great use case for an enum, where you will have sequential numbering and proper naming. Remove the constants and use an enum.

[QA-03] Move code to bring cohesion up

The event `FeePaid` and the `calculateFee` method should both be in `ArbitrumSwaps` instead of in `StargateArbitrum` since they have nothing to do with the Stargate logic, but are used in some swap/transfer scenarios.

[QA-04] Use `x != 0` to get positive-only uint values

The "positive uint" checks in the code are not done in the best possible way, one example is `_amount <= 0` - if a number is expected to be of a `uint` type, then you can check that it is positive by doing `x != 0` since `uint` can never be a negative number. Replace all `x <= 0` occurrences with `x != 0` when `x` is a `uint`.

[QA-05] Remove not needed custom error

The `MoreThanZero` custom error in `ArbitrumSwaps` is badly named and also duplicates the inherited from `StargateArbitrum` custom error `MustBeGt0` - prefer using the latter and remove the former.

[QA-06] Solidity safe pragma best practices are not used

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. Also `IStargateReceiver` and `IStargateRouter` interfaces are using an old compiler version - upgrade it to a newer version, use the same pragma statement throughout the whole codebase.

[QA-07] External method missing a NatSpec

The `arbitrumSwaps` method in `ArbitrumSwaps` is missing a NatSpec doc, add one to improve the code technical documentation.

[QA-08] Mismatch between contract and file names

The `ArbitrumSwaps` contract inherits from `SushiLegacyAdapter` which is imported from `SushiAdapter.sol`. Use the same name for the smart contract and the file.

[QA-09] Missing `override` keyword

The method `arbitrumSwaps` in `ArbitrumSwaps` is inheriting the method from `IArbitrumSwaps` but is missing the `override` keyword which should be there.

[QA-10] Typos in comments

In `StargateArbitrum` you wrote `arrat` -> `array`