CANTINA

Base Magic-Spend
Security Review

Cantina Managed review by:
Riley Holterhus, Lead Security Researcher
Rappie, Associate Security Researcher

March 6, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

From Feb 14th to Feb 19th the Cantina team conducted a review of magic-spend on commit hash 01a439d0. The team identified a total of **3** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 0
- Medium Risk: 0
- Low Risk: 0
- Gas Optimizations: 0
- Informational: 2

3 Findings

3.1 Critical Risk

3.1.1 Users can double withdraw their excess ETH

Severity: Critical Risk

Context: MagicSpend.sol#L74-L91

Description: When the `MagicSpend` contract is used as an ERC-4337 compatible paymaster, the user's `withdrawRequest` is used to cover the UserOp's gas fees, and any excess withdrawn ETH is recorded in the `_gasMaxCostExcess` mapping. When the user's transaction later executes, they have the option to either claim this excess immediately (using the `withdrawGasExcess()` function) or wait for the ERC-4337 `postOp()` function to refund it automatically. In either scenario, the `_gasMaxCostExcess` mapping should ultimately be reset to zero, since the user has received their excess ETH withdrawal.

However, this is currently not the case, as the `postOp()` function does not reset the `_gasMaxCostExcess` mapping. This means that a user can receive their excess once in the `postOp()` function, and again by calling `withdrawGasExcess()` in a subsequent transaction. By using multiple withdrawals with large excesses, an attacker can exploit this bug to drain all ETH in the `MagicSpend`.

Recommendation: Add a `delete` statement in the `postOp()` function:

```
  function postOp(IPaymaster.PostOpMode mode, bytes calldata context, uint256 actualGasCost)
      external
      onlyEntryPoint
  {
      if (mode == IPaymaster.PostOpMode.postOpReverted) {
          return;
      }

      (uint256 withheld, address account) = abi.decode(context, (uint256, address));

      // credit user difference between actual and withheld
      // and unwithdrawn excess
      uint256 excess = _gasMaxCostExcess[account] + (withheld - actualGasCost);

+     delete _gasMaxCostExcess[account];
      if (excess > 0) {
          _withdraw(address(0), account, excess);
      }
  }
```

Base: Fixed in PR 3.

Cantina Managed: Verified.

3.2 Informational

3.2.1 Accounting for `_gasMaxCostExcess` can be more precise

Severity: Informational

Context: MagicSpend.sol#L68, MagicSpend.sol#L78-L80

Description: In the `MagicSpend` contract, the `_gasMaxCostExcess` mapping is directly assigned to, instead of being adjusted through an increment. For example:

```
// In `validatePaymasterUserOp()`:
uint256 excess = withdrawRequest.amount - maxCost;
_gasMaxCostExcess[userOp.sender] = excess;
```

With this implementation, it's important that users can not overwrite a non-zero `_gasMaxCostExcess` value that remains from a previous transaction. Currently, there is one theoretical way in which a UserOp can leave behind a non-zero `_gasMaxCostExcess`: a revert in the `postOp()` function.

In the 4337 entrypoint flow, the paymaster's `postOp()` function will be called after the UserOp's execution completes. If this function reverts, the entrypoint catches this error and calls `postOp()` a second time, passing the `mode` as `PostOpMode.postOpReverted`. In the `MagicSpend` contract, this second `postOp()` call will have an early return:

```
if (mode == IPaymaster.PostOpMode.postOpReverted) {
    return;
}
```

So, if the `MagicSpend` reverts in the first call to `postOp()` (which might happen due to an insufficient ETH balance), the second `postOp()` call will return early, and the user's `_gasMaxCostExcess` mapping will remain non-zero after the transaction ends. As explained above, this can lead to issues where a later `validatePaymasterUserOp()` call can overwrite a user's balance of ETH they are owed.

However, there is one small nuance that prevents this issue from actually occurring. In version 0.6 of the 4337 entrypoint, there is a bug which causes short error messages to lead to reverts in the handling of the first `postOp()` failure. In the case of `MagicSpend`, a `postOp()` revert message will likely be the short `ETHTransferFailed()` error from the `safeTransferETH()` function in the `SafeTransferLib` library. Therefore, due to a bug in the v0.6 entrypoint, it appears impossible for a second `postOp()` call to even be reached in the `MagicSpend` contract. However, for extra safety, consider addressing the root-cause issue regardless of this obscure behavior.

Recommendation: In the `validatePaymasterUserOp()` function, consider incrementing the `_gasMaxCostExcess` mapping instead of always assigning new values:

```
  uint256 excess = withdrawRequest.amount - maxCost;
- _gasMaxCostExcess[userOp.sender] = excess;
+ _gasMaxCostExcess[userOp.sender] += excess;
```

Also, even though it may not be possible to reach, consider adding more precise accounting for the amount of `_gasMaxCostExcess` that remains in a `PostOpMode.postOpReverted` scenario:

```
  function postOp(IPaymaster.PostOpMode mode, bytes calldata context, uint256 actualGasCost)
      external
      onlyEntryPoint
  {
-     if (mode == IPaymaster.PostOpMode.postOpReverted) {
-         return;
-     }

      (uint256 withheld, address account) = abi.decode(context, (uint256, address));

+     if (mode == IPaymaster.PostOpMode.postOpReverted) {
+         _gasMaxCostExcess[account] += withheld - actualGasCost;
+         return;
+     }

      // credit user difference between actual and withheld
      // and unwithdrawn excess
      uint256 excess = _gasMaxCostExcess[account] + (withheld - actualGasCost);

      if (excess > 0) {
          _withdraw(address(0), account, excess);
      }
  }
```

Base: Fixed in PR 3.

Cantina Managed: Verified.

3.2.2 Pending comments

Severity: Informational

Context: MagicSpend.sol#L22, MagicSpend.sol#L73

Description: There are two pending comments in the `MagicSpend` contract code:

1. A comment that mentions `"TODO: Consider TSTORE"` above the `_gasMaxCostExcess` mapping. Since this mapping is always cleared by the end of every transaction, transient storage would indeed be useful in this scenario.
2. A comment that mentions `"TODO consider updating for entrypoint v0.7"` above the `postOp()` function. In v0.7 of the ERC-4337 spec, the `postOp()` functionality has indeed been changed, for example by taking an additional `actualUserOpFeePerGas` parameter.

Recommendation: Consider deleting these TODO comments in the final version of the code. Additionally, based on the intended timeline around deployment, consider implementing these proposed changes.

Base: Fixed in PR 3 by deleting the comments.

Cantina Managed: Verified.

4 Appendix

4.1 Additional Testing

Description: A fuzzing framework and test suite were developed during the audit. The code can be found here, and the results of running the tests on the final code are included below.

Result:

Property	Result
No unwanted reverts in `validatePaymasterUserOp`	PASSED
No unwanted reverts in `postOp`	PASSED
No unwanted reverts in `withdrawGasExcess`	PASSED
No unwanted reverts in `withdraw`	PASSED
Paymaster validating a user op does not change balances	PASSED
Calling `postOp` with mode `IPaymaster.PostOpMode.postOpReverted` must never revert	PASSED
Calling `postOp` with mode `IPaymaster.PostOpMode.opSucceeded` transfers correct amount to user	PASSED
No excess gas balance available after calling `postOp` with mode `IPaymaster.PostOpMode.opSucceeded`	PASSED
Withheld amount after failed `postOp` is correctly reflected by `magic.gasExcessBalance()`	PASSED
`withdrawGasExcess` transfers correct amount to user	PASSED
No excess gas balance available after calling `withdrawGasExcess`	PASSED
`withdraw` transfers correct amount to user	PASSED
Simulated user operation transfers amount in withdraw request minus actual gas costs to user	PASSED
Paymaster balance equals all deposits minus all withdraws	PASSED

Recommendation: Incorporate these tests into the existing test suite.