# Verifying paymaster v2

## Security Review

Cantina Managed review by:

**Riley Holterhus**, Lead Security Researcher

**Akshay Srivastav**, Security Researcher

December 3, 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

Coinbase's Verifying Paymaster is an ERC-4337 compliant paymaster, designed for use within the Coinbase Developer Platform. It uses signature-based validation and supports ERC20 token fee payments.

From Nov 21st to Nov 24th the Cantina team conducted a review of verifying-paymaster-v2 on commit hash f2bf8ab0. The team identified a total of **8** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 0
- Gas Optimizations: 0
- Informational: 7

# 3 Findings

## 3.1 Medium Risk

### 3.1.1 `verifyingSigner` has no authority over paymaster-related gas limits

**Severity:** Medium Risk

**Context:** VerifyingPaymaster.sol#L268-L281

**Description:** In ERC4337 v0.7, the gas limits used for calling the paymaster functions `validatePaymasterUserOp()` and `postOp()` are now included in the `userOp.paymasterAndData` bytes, as detailed in PR 363.

In the new version of the `VerifyingPaymaster`, these gas limit values are not inspected or included in the hash signed by the `verifyingSigner`. This means that the userOp sender has the ability to modify these gas limits after the `verifyingSigner` signature is generated, without affecting the validity of the signature.

This may be a concern in scenarios where `allowAnyBundler == true`. For example, if the userOp sender decreases the `postOp()` gas limit to be much lower than the `verifyingSigner` initially expected, they could potentially cause the entire userOp to fail and leave the `VerifyingPaymaster` liable for the gas fees. At the very least, it seems that the `verifyingSigner` would have stronger guarantees if they knew their signature was agreeing to specific paymaster gas limits.

Note that in the previous version of the `VerifyingPaymaster`, the `verifyingSigner` always had full authority over the gas limits used in the paymaster calls, since the gas limit was simply the `verificationGasLimit` and the contributed hash found in VerifyingPaymaster.sol#L239 that was signed.

**Recommendation:** To prevent tampering with the intended gas limits for `validatePaymasterUserOp()` and `postOp()`, consider including these values in the hash generated by `getHash()`. One possible way to implement this is the following:

```
- function getHash(PackedUserOperation calldata userOp, PaymasterData memory paymasterData) public view
↪   returns (bytes32) {
+ function getHash(PackedUserOperation calldata userOp, PaymasterData memory paymasterData, uint256
↪   paymasterValidationGasLimit, uint256 postOpGasLimit) public view returns (bytes32) {
      // can't use userOp.hash(), since it contains also the paymasterAndData itself.
      return keccak256(
          abi.encode(
              userOp.getSender(),
              userOp.nonce,
              calldataKeccak(userOp.initCode),
              calldataKeccak(userOp.callData),
              userOp.accountGasLimits,
              userOp.preVerificationGas,
              userOp.gasFees,
              block.chainid,
              address(this),
-             paymasterData
+             paymasterData,
+             paymasterValidationGasLimit,
+             postOpGasLimit
          )
      );
  }


  function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 maxCost)
      internal
      override
      returns (bytes memory context, uint256 validationData)
  {
      (PaymasterData memory paymasterData, bytes memory signature) =
↪   parsePaymasterData(userOp.paymasterAndData[UserOperationLib.PAYMASTER_DATA_OFFSET:]);

      // ...
+     (, uint256 paymasterValidationGasLimit, uint256 postOpGasLimit) =
↪   UserOperationLib.unpackPaymasterStaticFields(userOp.paymasterAndData);
+     bytes32 hash = MessageHashUtils.toEthSignedMessageHash(getHash(userOp, paymasterData,
↪   paymasterValidationGasLimit, postOpGasLimit));
-     bytes32 hash = MessageHashUtils.toEthSignedMessageHash(getHash(userOp, paymasterData));
      // ...
  }
```

**Coinbase:** Fixed in commit 0a0453f9.

**Cantina Managed:** Verified.

## 3.2 Informational

### 3.2.1 Entrypoint version typo

**Severity:** Informational

**Context:** VerifyingPaymaster.sol#L18

**Description:** The comment above the `VerifyingPaymaster` states it is compatible with entrypoint v0.6. However, the updated `VerifyingPaymaster` is now compatible with entrypoint v0.7, so the comment can be updated.

**Recommendation:** Update the comment above the `VerifyingPaymaster` as follows:

```
- /// @notice ERC4337 Paymaster implementation compatible with Entrypoint v0.6.
+ /// @notice ERC4337 Paymaster implementation compatible with Entrypoint v0.7.
```

**Coinbase:** Fixed in commit 0a0453f9.

**Cantina Managed:** Verified.

### 3.2.2 `_trySafeTransfer()` succeeds if `token` has no code

**Severity:** Informational

**Context:** VerifyingPaymaster.sol#L396-L404

**Description:** The `_trySafeTransfer()` function is used in `_postOp()` for attempting an ERC20 `transfer()` call. This function uses a low-level call to the `token` address, and returns `true` if the call succeeded and the `returnData` is either empty or decodes to `true`:

```
(bool success, bytes memory returnData) = token.call(
    abi.encodeWithSelector(ERC20.transfer.selector, to, amount)
);
return success && (returnData.length == 0 || abi.decode(returnData, (bool)));
```

Note that if the `token` address has no code, the low-level call will still succeed, and the `returnData` will be empty, which results in the `_trySafeTransfer()` function returning `true`. This may be unexpected.

However in practice, this behavior doesn't seem to cause issues, because the `VerifyingPaymaster` is not expected to interact with atypical tokens or addresses without code.

**Recommendation:** This finding has been given for informational purposes, and no changes are necessary to the code.

**Coinbase:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.2.3 `paymasterData.postOpGas` and `userOp.unpackPostOpGasLimit()` considerations

**Severity:** Informational

**Context:** VerifyingPaymaster.sol#L46-L47

**Description:** The `VerifyingPaymaster` defines a `PaymasterData` struct that includes a `postOpGas` value. This value is important, because the `actualGasCost` given to the `postOp()` function does not include the gas consumed by the `postOp()` call itself (since this would be circular logic), so the `postOpGas` value is manually added in the calculations to correct this.

On the other hand, note that in ERC4337 v0.7, there is now a dedicated gas limit for the `postOp()` call that is provided in `userOp.paymasterAndData`. This can be accessed through `userOp.unpackPostOpGasLimit()`.

In most circumstances, these two values will be very similar, and it is likely always the case that `paymasterData.postOpGas <= userOp.unpackPostOpGasLimit()`. This is because `paymasterData.postOpGas` estimates the gas used in `postOp()` that should be credited to the paymaster, whereas `userOp.unpackPostOpGasLimit()` defines the maximum allowable gas for the `postOp()` call. It may be beneficial to enforce this relationship explicitly in the code.

**Recommendation:** Consider adding a `require` statement to ensure that `paymasterData.postOpGas <= userOp.unpackPostOpGasLimit()` in the `_validatePaymasterUserOp()` function. For example:

```
  function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 maxCost)
↪  /* .. */ {
      (PaymasterData memory paymasterData, bytes memory signature) =
↪  parsePaymasterData(userOp.paymasterAndData[UserOperationLib.PAYMASTER_DATA_OFFSET:]);
+     require(paymasterData.postOpGas <= userOp.unpackPostOpGasLimit());
      // ...
  }
```

It is also recommended to evaluate whether simplifying the design by removing the `postOpGas` field from the `PaymasterData` struct is beneficial. Instead, the implementation could rely solely on `userOp.unpackPostOpGasLimit()` to determine both the gas limit and reimbursement amount.

**Coinbase:** Addressed in commit 6bee4b91.

**Cantina Managed:** Verified. A check has been added to `_validatePaymasterUserOp()` that ensures that `paymasterData.postOpGas <= userOp.unpackPostOpGasLimit()`.

### 3.2.4 maxTokenCost calculation can be optimized

**Severity:** Informational

**Context:** VerifyingPaymaster.sol#L303-L304

**Description:** The maxTokenCost and actualTokenCost calculations in _validatePaymasterUserOp() and _postOp() both include an addition relating to postOpGas, which can be seen below:

```
function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 maxCost) /*
↪   ... */ {
    // ...
    if (/* ... */) {
        if (/* ... */) {
            uint256 maxFeePerGas = userOp.unpackMaxFeePerGas();
            uint256 maxTokenCost =
                _calculateTokenCost(maxCost + paymasterData.postOpGas * maxFeePerGas,
↪   paymasterData.exchangeRate);
                // ...
            }
        }
    }
    // ...
}

function _postOp(PostOpMode,
    bytes calldata context,
    uint256 actualGasCost,
    uint256 actualUserOpFeePerGas) /* ... */ {
    // ...
    if (/* ... */) {
        uint256 actualTokenCost = _calculateTokenCost(actualGasCost + c.postOpGas * actualUserOpFeePerGas,
↪   c.exchangeRate);
        // ...
    } else {
        // ...
    }
}
```

In the case of _postOp(), this is an important addition, since the actualGasCost does not include the cost of the postOp() call itself, and therefore the postOpGas cost needs to be manually added.

However, in _validatePaymasterUserOp() the addition of postOpGas does not appear necessary. In ERC4337 v0.7, the entrypoint's _getRequiredPrefund() function already adds an upper-bound on the gas cost for the postOp() call. Since the maxCost passed to _validatePaymasterUserOp() originates from _getRequiredPrefund(), the postOpGas addition in _validatePaymasterUserOp() is redundant, and can be removed to lower the upfront costs paid by users.

Note that this optimization assumes that paymasterData.postOpGas <= userOp.unpackPostOpGasLimit(), which is a relationship described in a separate issue. If this condition is not met, the upper-bound cost added in _getRequiredPrefund() might be smaller than the actual gas cost added in _postOp(), which would make this optimization invalid.

**Recommendation:** To lower the maxTokenCost required in prepayments, consider removing the unnecessary addition of postOpGas in _validatePaymasterUserOp():

```
  function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash, uint256 maxCost)
↪   /* ... */ {
      // ...
      if (/* ... */) {
          if (/* ... */) {
              uint256 maxFeePerGas = userOp.unpackMaxFeePerGas();
              uint256 maxTokenCost =
-                 _calculateTokenCost(maxCost + paymasterData.postOpGas * maxFeePerGas,
↪   paymasterData.exchangeRate);
+                 _calculateTokenCost(maxCost, paymasterData.exchangeRate);
              // ...
          }
      }
      // ...
  }
```

**Coinbase:** Removed the addition in commit 0a0453f9 and removed the now-unused `maxFeePerGas` variable in commit 6bee4b91.

**Cantina Managed:** Verified.

### 3.2.5 Unused gas penalty considerations

**Severity:** Informational

**Context:** VerifyingPaymaster.sol#L328-L331

**Description:** In ERC4337 v0.7, a 10% penalty has been introduced for unused gas relative to the `callGasLimit` and `paymasterPostOpGasLimit`. This is described in PR 356 and can be seen in the following code snippet (EntryPoint.sol#L718-L728) from the entrypoint:

```
// Calculating a penalty for unused execution gas
{
    uint256 executionGasLimit = mUserOp.callGasLimit + mUserOp.paymasterPostOpGasLimit;
    uint256 executionGasUsed = actualGas - opInfo.preOpGas;
    // this check is required for the gas used within EntryPoint and not covered by explicit gas limits
    if (executionGasLimit > executionGasUsed) {
        uint256 unusedGas = executionGasLimit - executionGasUsed;
        uint256 unusedGasPenalty = (unusedGas * PENALTY_PERCENT) / 100;
        actualGas += unusedGasPenalty;
    }
}
```

Note that this code executes after the call to the paymaster's `postOp()` function. As a result, unless the paymaster explicitly accounts for this 10% penalty within its own calculations, the penalty is incurred by the paymaster. This is because the `actualGasCost` passed to `postOp()` would not include the penalty, yet the `actualGas` used to determine refunds to the paymaster is incremented by the penalty above.

**Recommendation:** Consider whether this behavior is relevant to the `VerifyingPaymaster`. Since the necessary variables from the entrypoint's penalty calculation are not available within the `postOp()` function, it may be difficult to incorporate this penalty into the paymaster's `postOp()` calculations.

Therefore, this penalty should likely be monitored off-chain, and the paymaster should avoid sponsoring transactions that leave a large amount of unused gas.

**Coinbase:** Acknowledged, we will be monitoring to ensure gas values are as close to what we estimate as we deem reasonable.

**Cantina Managed:** Acknowledged.

### 3.2.6 Failing test case for `VerifyingPaymaster.getHash()` function

**Severity:** Informational

**Context:** VerifyingPaymaster.t.sol#L68-L81

**Description:** The `test_getHash_isCorrect` test case in `test/VerifyingPaymaster.t.sol` is failing as the hash returned by `VerifyingPaymaster.getHash` function is not equal to the expected hash value.

**Recommendation:** Consider updating the hardcoded hash value in the test case:

```
  bytes32 hash = paymaster.getHash(
      userOp,
      paymasterData
  );
  // Replace with the expected hash value
  assertEq(
      hash,
-     0xc372ec6e68f9f54f8a4aa771437e3852d647b56011383d36389782a755caa2da
+     0x4b912bd027816bd34aa017f207ebd689fb309cfc4e5c626bcc7160c47bd3f6f9
  );
```

**Coinbase:** Fixed in commit f42791eb.

**Cantina Managed:** Verified. The discrepancy was traced to a difference in downstream `creationCode` metadata, which caused the test case to use different addresses in the hash calculation. The fix resolves this by hardcoding the address to always be the same.

### 3.2.7 Project compilation fails due to missing `via_ir` flag in foundry configuration

**Severity:** Informational

**Context:** foundry.toml#L1-L7

**Description:** The `VerifyingPaymaster.t.sol` test file uses a lot of local variables in functions which hits the stack limit of solidity. Due to this the compilation of solidity project fails with `Stack too deep` error.

```
Error: Compiler run failed:
Error: Compiler error (/solidity/libsolidity/codegen/LValue.cpp:51):Stack too deep. Try compiling with
↪  `--via-ir` (cli) or the equivalent `viaIR: true` (standard JSON) while enabling the optimizer. Otherwise,
↪  try removing local variables.
CompilerError: Stack too deep. Try compiling with `--via-ir` (cli) or the equivalent `viaIR: true` (standard
↪  JSON) while enabling the optimizer. Otherwise, try removing local variables.
   --> test/VerifyingPaymaster.t.sol:125:13:
    |
125 |             v
    |
```

**Recommendation:** Consider adding the `via_ir` flag in `foundry.toml` file.

```
  [profile.default]
  src = "src"
  out = "out"
  libs = ["lib"]

  optimizer = true
+ via_ir = true
```

**Coinbase:** Fixed in commit 0a0453f9.

**Cantina Managed:** Verified.