CANTINA

Base Paymaster & Smart Account

Security Review

Cantina Managed review by:

Riley Holterhus, Lead Security Researcher

Blockdev, Security Researcher

January 7, 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 user base on-chain. The `paymaster` protocol contains a verifying paymaster contract that can be used for gas subsidies for ERC-4337 transactions. It contains a clone of the eth-infinitism VerifyingPaymaster with an additional `receive()` function for simple deposits.

From Dec 11th to Dec 13th the Cantina team conducted a review of paymaster and smart-account on commit hashes 67b44ad1 and 2779bed4 respectively. The team identified a total of **15** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 5
- Gas Optimizations: 3
- Informational: 7

3 Findings

3.1 Low Risk

3.1.1 Add `__gap` to all upgradeable contracts

Severity: Low Risk

Context: ERC4337Account.sol, MultiOwnable.sol

Description: This is the current storage layout of `ERC4337Account`:

Name	Type	Slot	Offset	Bytes
`ownerIndex`	`uint8`	0	0	1
`ownerAtIndex`	`mapping(uint8 => bytes)`	1	0	32
`_isOwner`	`mapping(bytes => bool)`	2	0	32
`_initialized`	`bool`	3	0	1

Suppose during an upgrade, `uint256 newMOVar` is added to the end of `MultiOwnable.sol`, here's the new layout:

Name	Type	Slot	Offset	Bytes
`ownerIndex`	`uint8`	0	0	1
`ownerAtIndex`	`mapping(uint8 => bytes)`	1	0	32
`_isOwner`	`mapping(bytes => bool)`	2	0	32
`newMOVar`	`uint256`	3	0	32
`_initialized`	`bool`	4	0	1

`_initialized` can get even get slot packed with `newMOVar` if it is `uint8` for example.

The new variable takes the storage slot that previously belonged to `_initialized`. It's a good practice to preserve the storage slots for the variables. OpenZeppelin until v4 recommended declaring a `__gap` array in each contract at the end of all variable declarations. Whenever a new variable has to be introduced, declare it just above `__gap` and reduce its size by 1.

Recommendation: There are few best practices to follow regarding variable declarations (listed here). Regarding this issue, introduce `uint256[50] __gap` variable at the end of `ERC4337Account` and `MultiOwnable` and remember to reduce the array size by the number of new storage slots being taken in each contract when upgrading.

Since this is a small contract, another good practice would be to manually view the storage layout to ensure it is preserved for older variables.

Base: Fixed in PR 13. The chosen fix was to instead use namespaced storage per the ERC-7201 spec. This is indeed a valid way to address this issue, since new storage structs can easily be added in non-intersecting storage slots derived from the hash of each namespace.

Cantina Managed: Verified.

3.1.2 `ERC4337Account` implementation can be initialized

Severity: Low Risk

Context: ERC4337Account.sol#L83-L87

Description: When a user creates a new `ERC4337Account`, they are actually deploying a minimal ERC1967 proxy to a shared `ERC4337Account` implementation. As part of this flow, the `ERC4337Account` has an `initialize()` function that's expected to run in the proxy's storage context.

However, there is nothing stopping someone from calling `initialize()` on the implementation itself. This type of issue has previously caused problems by allowing a self-destruct through `UUPSUpgradeable`, but most libraries (including Solady) are now smart enough to prevent this. Regardless, it is considered best practice to prevent initializing implementation contracts.

Recommendation: Prevent initialization of the implementation by adding the following constructor to `ERC4337Account`:

```
constructor() {
    _initialized = true;
}
```

Base: Fixed in PR 9.

Cantina Managed: Verified.

3.1.3 Owner `bytes` can be decoded in unexpected ways

Severity: Low Risk

Context: ERC4337Account.sol#L211-L238, MultiOwnable.sol

Description: To allow different verification schemes, the `MultiOwnable` contract stores its owners as arbitrary `bytes` values. With the current `ERC4337Account` implementation, these bytes should either be encoded `address` values (32 bytes with 12 leading zero bytes), or encoded `(uint256,uint256)` values (64 bytes). Although these are the only two useful options, other types of `bytes` can be added, which is a potential footgun.

In addition to this, the current decoding logic can interpret the `bytes` values in unexpected ways. For one example, consider the `address` decoding:

```
bytes memory ownerBytes = ownerAtIndex[ownerIndex];
address owner;
assembly {
    owner := mload(add(ownerBytes, 32))
}
```

This implementation reads the 32 bytes following the `ownerBytes` length section, and then casts it to an `address`. This is unsafe if `ownerAtIndex[ownerIndex]` is empty, because the 32 bytes after the `ownerBytes` length will be unrelated memory. Fortunately, this can't be exploited, since the memory appears to always be zero bytes. This implementation would also consider the last 20 bytes of a secp256r1 x-coordinate as a valid `address` owner. This similarily can't be exploited, since it wouldn't be possible to know the private key for this "fake" owner. However, it would be safest if this behavior was explicitly prevented.

Recommendation: Firstly, consider restricting the types of `bytes` values that can be added as owners, for example, by only allowing properly encoded `address` or `(uint256,uint256)` values to be added.

Secondly, whenever the owner `bytes` are decoded and used, ensure that their length makes sense for the verification type. To be even safer, consider refactoring the code to encode a "type" with each owner, so that the intended usage of the `bytes` is always known, even if future verification schemes expect the same length of owner `bytes`.

Base: Fixed in PR 7.

Cantina Managed: Verified.

3.1.4 `entryPoint` cannot be changed

Severity: Low Risk

Context: MetaPaymaster.sol#L13

Description: `MetaPaymaster` keeps `entryPoint` as a storage variable even though there's no way to update it after initializing the proxy. So if `entryPoint` changes for any reason (most likely because a bug is discovered there), the only way is to upgrade the contract.

This can be mitigated by having a setter function for `entryPoint`. However, if this is not a concern, you may also consider setting `entryPoint` immutable and setting it in the implementation contract's constructor. This saves some gas.

Recommendation: With all the above context, you may want to do one of the following:

- Make `entryPoint` immutable. If it needs to be changed, the contract has to be upgraded (which is the case currently as well).
- Have a setter function guarded by `onlyOwner` to set `entryPoint`.

Base: Fixed in PR 23.

Cantina Managed: Verified.

3.1.5 `authenticatorData.length` has to be at least `33`

Severity: Low Risk

Context: WebAuthn.sol#L129

Description: `verifySignature()` checks `authenticatorData` is well formed by verifying its length and the flags stored at index `32`.

```
if (authenticatorData.length < 32 || !checkAuthFlags(authenticatorData[32], requireUserVerification)) {
```

However, a malformed `authenticatorData` of length `32` passes the length check and then the second check reverts because it access an index out of range. Since `authenticatorData` has to be at least `33` bytes long due to `rpIdHash` (32 bytes) and `flags` (1 byte), the length check should check for `33` instead of `32`.

However, the risk is limited as the code just reverts instead of returning `false`.

Recommendation: Update the check to:

```
if (authenticatorData.length < 33 || !checkAuthFlags(authenticatorData[32], requireUserVerification)) {
```

Base: Fixed in PR 8. Since a well-formed `authenticatorData` has to be at least 37 bytes long, we updated the length check to `37`.

Cantina Managed: Verified.

3.2 Gas Optimization

3.2.1 `signature` can be kept in `calldata`

Severity: Gas Optimization

Context: ERC4337Account.sol#L219-L227

Description: `_validateSignature()` copies `signature` from `signaturePacked`:

```
bytes memory signature = signaturePacked[1:];
```

`signature` can be kept in calldata and `SignatureCheckLib.isValidSignatureNowCalldata()` can be used to avoid copying it to memory again instead of `isValidSignatureNow()`:

```
return SignatureCheckerLib.isValidSignatureNow(owner, message, signature);
```

Code for passkey signature verification remains the same.

Recommendation: Update the code as follows:

```
- bytes memory signature = signaturePacked[1:];
+ bytes calldata signature = signaturePacked[1:];

  if (signature.length == 65) {
      bytes memory ownerBytes = ownerAtIndex[ownerIndex];
      address owner;
      assembly {
          owner := mload(add(ownerBytes, 32))
      }
-     return SignatureCheckerLib.isValidSignatureNow(owner, message, signature);
+     return SignatureCheckerLib.isValidSignatureNowCalldata(owner, message, signature);
  }
```

Base: Fixed in PR 6.

Cantina Managed: Verified.

3.2.2 Length can be checked to identify empty bytes

Severity: Gas Optimization

Context: MultiOwnable.sol#L51

Description: `existingOwner` is of type `bytes memory`. To check if it's empty, its hash is compared against the hash of empty bytes (`""`):

```
if (keccak256(existingOwner) != EMPTY) revert IndexNotEmpty(index, existingOwner);
```

The same can be achieved just by checking its length.

Recommendation: Update the code as:

```
if (existingOwner.length != 0) revert IndexNotEmpty(index, existingOwner);
```

Base: Fixed in PR 5.

Cantina Managed: Verified.

3.2.3 Unnecessary `if`/`else` block

Severity: Gas Optimization

Context: MetaPaymaster.sol#L78-L82

Description: The following two branches add the same quantity to `total`:

```
if (amount > balanceOf[account]) {
    total += amount - balanceOf[account];
} else {
    total -= balanceOf[account] - amount;
}
```

This is to avoid underflow when calculating `amount - balanceOf[account]`. There's a way to update `total` without using `if`/`else`.

Recommendation: Refactor the code to:

```
total = total + amount - balanceOf[account];
```

Since `total >= balanceOf[account]`, this can never underflow. You can also add a comment here to warn the reader why `total += amount - balanceOf[account]` shouldn't be used.

Base: Fixed in PR 24.

Cantina Managed: Verified.

3.3 Informational

3.3.1 ERC1271 hash structure considerations

Severity: Informational

Context: ERC1271.sol#L12

Description: To mitigate signature replay issues when multiple accounts share an owner, the `ERC1271` contract validates a `bytes32 hash` in the following way (where `_hashTypedData()` is coming from EIP712):

```
bool success = _validateSignature(SignatureCheckerLib.toEthSignedMessageHash(_hashTypedData(hash)), signature);
```

While this successfully prevents replay issues, there are two informational considerations with this flow:

1. According to the EIP712 spec, the encoding of the typed structure should be `"\x19\x01" ‖ domainSeparator ‖ hashStruct(message)`. In the code, the `_hashTypedData()` function expects the `hashStruct(message)` portion as input, and then computes the hash of the full encoding. However, the current implementation passes the raw `message` as input to `_hashTypedData()`. To be more compliant with EIP712, a simple type struct such as `"BaseAccountMessage(bytes32 message)"` could be used to give a struct hash to `_hashTypedData()`. Note that even with this change, the `bytes32 message` would still appear to the user as an arbitrary hash, which is an unfortunate consequence of needing to deal with the 1271 replay issue.

2. The `toEthSignedMessageHash()` function implements the 0x45 "version" of EIP191, while EIP712 is a spec for the 0x01 "version" (see here). So, since EIP712 is already being used, the `toEthSignedMessageHash()` is not necessary, as the `_hashTypedData()` alone will distinguish the hash from representing an actual RLP encoded transaction. Note that the `bytes32 hash` *itself* may come from the 0x45 or 0x01 "versions" of EIP191, in which case the caller of `isValidSignature()` would do its own hashing.

Recommendation: Although the current code is fine from an on-chain perspective, consider simplifying the process for off-chain signing as follows:

1. Add a `"BaseAccountMessage(bytes32 message)"` typestruct, and pass the struct hash as input to `_hashTypedData()`:

```
+ bytes32 constant STRUCT_HASH = keccak256("BaseAccountMessage(bytes32 message)");

  function isValidSignature(bytes32 hash, bytes calldata signature) public view virtual returns (bytes4
↪ result) {
+     hash = keccak256(abi.encode(STRUCT_HASH , hash));
      // ...
  }
```

2. Remove the `toEthSignedMessageHash()` in the code, and only use EIP712.

Base: Fixed in PR 12.

Cantina Managed: Verified.

3.3.2 Consider commenting a warning for `executeBatch()` for future changes

Severity: Informational

Context: ERC4337Account.sol#L170

Description: To store return data from all calls in `executeBatch()`, `m` is advanced by `returndatasize()`:

```
m := add(p, returndatasize()) // Advance `m`.
```

In this case if `returndatasize()` isn't divisible by 32 bytes, multiple return values can exist in the same memory slot which isn't the correct layout for the return calldata. See how it describes return value encoding here and specifically bytes encoding here (notice the "*pad_right*" part).

However, in this case, Solidity automatically formats this data correctly by copying it explicitly to a new memory location. If however, `return` opcode would have been used, this would result in incorrect ABI encoding resulting in a calldata without proper structure.

Recommendation: A warning can be added as a comment explaining why `return` opcode shouldn't be used.

Base: Thanks for the note, will consider adding but not urgent.

Cantina Managed: Acknowledged.

3.3.3 Consider reverting `removeOwnerAtIndex()` if the index is empty

Severity: Informational

Context: MultiOwnable.sol#L57-L64

Description: If the `removeOwnerAtIndex()` is called on an index that doesn't contain an owner, the function will succeed in removing the non-existent owner. Since `_addOwnerAtIndex()` reverts if an owner already exists, it could make more sense if `removeOwnerAtIndex()` reverts if the owner doesn't exist.

Recommendation: Consider adding an existence check in `removeOwnerAtIndex()`, so that it's more obvious that this scenario results in a no-op.

Base: Fixed in PR 11.

Cantina Managed: Verified.

3.3.4 Remove commented-out function

Severity: Informational

Context: ERC4337Account.sol#L201-L208

Description: A legacy version of `isValidSignature()` still remains commented-out in `ERC4337Account`. This function is now implemented in the inherited `ERC1271` contract, so the commented-out version can be removed.

Recommendation: Remove the commented-out `isValidSignature()` function from `ERC4337Account`.

Base: Fixed in PR 10.

Cantina Managed: Verified.

3.3.5 Number the `WebAuthn` verification steps to match its spec

Severity: Informational

Context: WebAuthn.sol#L133-L147

Description: As done in `checkAuthFlags()`, comments describing verification steps can be numbered to correspond to the same numbered bullets in webauthn spec. It makes it easier to reference for cross-checking.

Recommendation: Number the inline comments describing verification steps to match webauthn verification steps.

Base: May get to this eventually, but not in this audit response.

Cantina Managed: Acknowledged.

3.3.6 `assembly` block without `("memory-safe")`

Severity: Informational

Context: ERC4337Account.sol#L107

Description: Function `validateUserOp()` has an `assembly` block without `("memory-safe")`, while all other blocks are marked as `memory-safe`. Using the same pattern everywhere is more consistent.

```
assembly {
    validationData := iszero(success)
}
```

Recommendation: Consider adding `("memory-safe")` here too.

Base: Fixed in PR 7.

Cantina Managed: Verified.

3.3.7 `addOwner()` can only add 255 owners

Severity: Informational

Context: MultiOwnable.sol#L38, MultiOwnable.sol#L44

Description: `addOwner()`'s Natspec says:

```
/// @dev convenience function that can be used to add the first
/// 256 owners.
```

However, `addOwner()` can only add 255 owners. When `ownerIndex` is `i`, this means the first `i` owners have been added. `ownerIndex`'s max is 255. Hence only 255 owners can be added. After that it'll revert due to overflow. This doesn't pose any risk. Even then, `addOwnerAtIndex()` can be called to add the 256th owner.

Recommendation: Update the highlighted comments to say 255 instead of 256.

Base: Fixed in PR 4.

Cantina Managed: Verified.