certora

Security Assessment & Formal Verification Report

coinbase WALLET

February 2024

Prepared for
Coinbase

certora coinbase WALLET

Table of content

Project Summary

Project Scope

Repo Name	Repository	Commits	Compiler version	Platform
smart-wallet	https://github.com/coinbase/smart-wallet (public)	7aa092a	Solidity 0.8.23	EVM

Project Overview

This document describes the specification and verification of the **Coinbase smart-account** using the Certora Prover and manual code review findings. The work was undertaken from **8 Feb 2023** to **29 February 2024**.

The following contract list is included in our scope:

```
src/ERC1271.sol
src/ERC4337Account.sol
src/ERC4337Factory.sol
src/MultiOwnable.sol
src/WebAuthn.sol
```

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	0	0
High	0	0	0
Medium	1	1	1
Low	1	1	0
Informational	7	7	6
Total	9	9	7

Detailed Findings

Medium Severity Concerns

M-1. createAccount works with an empty owners array and will create a frontrunnable/stealable account

Impact: High
Probability: Low
Description:

It's possible to call `ERC4337Factory.createAccount()` with an empty `bytes[] calldata owners`. As this value is **allowed**, we can imagine a scenario of someone **wanting** to create an account and fill the array of owners later.

The following test passes:

```
Unset
diff --git a/test/ERC4337Factory.t.sol b/test/ERC4337Factory.t.sol
index 7032838..fbe51f9 100644
--- a/test/ERC4337Factory.t.sol
+++ b/test/ERC4337Factory.t.sol
@@ -18,6 +18,8 @@ contract ERC4337FactoryTest is Test {
   }

   function test_createAccountSetsOwnersCorrectly() public {
+      bytes[] memory b;
+      address c = factory.createAccount{value: 1e18}(b, 0);
       address a = factory.createAccount{value: 1e18}(owners, 0);
       assert(ERC4337Account(payable(a)).isOwnerAddress(address(1)));
       assert(ERC4337Account(payable(a)).isOwnerAddress(address(2)));
```

This will deploy a deterministic contract that won't have any owners and will **NOT be initialized** due to the loop in `_initializeOwners` being skipped and therefore `nextOwnerIndex` staying at `0`:

```
Unset
File: MultiOwnable.sol
  function _initializeOwners(bytes[] memory owners) internal virtual {
    for (uint256 i = 0; i < owners.length; i++) {
      if (owners[i].length != 32 && owners[i].length != 64) { //@note require(length of 32 or 64)
        revert InvalidOwnerBytesLength(owners[i]);
      }
      if (owners[i].length == 32 && uint256(bytes32(owners[i])) > type(uint160).max) {//@note require(could be 32 so the check here is in the case of 32, if it happens that the owner's bytes are more than 160 bytes)
        revert InvalidEthereumAddressOwner(owners[i]);
      }
      _addOwnerAtIndexNoCheck(owners[i], _getMultiOwnableStorage().nextOwnerIndex++);
    }
  }
```

Such a contract holds funds and can be initialized in a second transaction.

Although this can happen only once per nonce, the fact that the deployed contract doesn't have any owners and needs to be initialized in a separate transaction means that such a deployed contract will be vulnerable to frontrunning attacks (anyone can call initialize()).

Some potential frontrunning exploits would be:

- Monitoring the mempool to copy the list of owners from the rightful user's attempted call to initialize, and frontrun it while a**ppending an extra malicious owner**
- If funds were sent to the deployed contract during deployment (createAccount() is payable and msg.value is forwarded to the deployed contract), the frontrunner could simply **steal ownership** of the account containing the **funds**

Recommendation

Consider checking that `owners.length > 0` in `ERC4337Factory.createAccount()`

Coinbase's response: addressed in the updated code.

Low Severity Concerns

L-1. EVM Version Shanghai may not work on other chains due to PUSH0

Impact: Low
Probability: Depends on the chain
Description:

This is a general recommendation to bring awareness to a prevalent problem that currently exists in the ecosystem.

The compiler for Solidity 0.8.20 switches the default target EVM version to Shanghai, which includes the new `PUSH0` op code. This op code may not yet be implemented on all L2s, so deployment on these chains will fail. It's not necessary to specifically use PUSH0 in YUL for it to be included.

As an example, this op code is not supported on Base, which is built upon Optimism Bedrock (see here). See also this relevant issue on the official Solidity github for reference.

Due to this, deployment of any in scope contract to the Base chain will always fail with error "invalid opcode: PUSH0".

Affected code:

```
Unset
# File: src/ERC4337Account.sol
ERC4337Account.sol:2: pragma solidity 0.8.23;

# File: src/ERC4337Factory.sol
ERC4337Factory.sol:2: pragma solidity ^0.8.4;

# File: src/MultiOwnable.sol
MultiOwnable.sol:2: pragma solidity ^0.8.4;

# File: src/WebAuthn.sol
WebAuthn.sol:2: pragma solidity ^0.8.0;
```

Coinbase's response:
We agree with this observation and assume that PUSH0 is now well adopted across L2s and are comfortable using EVM versions that use it (>= Shanghai).

Informational Concerns

I-1. Constants should be in CONSTANT_CASE

For `constant` variable names, each word should use all capital letters, with underscores separating each word (CONSTANT_CASE).
Affected code:

```
Unset
# File: src/MultiOwnable.sol
MultiOwnable.sol:25:     bytes32 private constant MultiOwnableStorageLocation =
```

Coinbase's response:
We agreed with the recommendation and implemented the changes in this PR, specifically in this commit.

I-2. Default Visibility for constants

Some constants are using the default visibility. For readability, consider explicitly declaring them as `internal`.
Affected code:

```
Unset
# File: src/WebAuthn.sol
WebAuthn.sol:29:     bytes1 constant AUTH_DATA_FLAGS_UP = 0x01;
WebAuthn.sol:31:     bytes1 constant AUTH_DATA_FLAGS_UV = 0x04;
WebAuthn.sol:33:     uint256 constant P256_N_DIV_2 =
57896044605178124381348723474703786764998477612067880171211129530534256022184;
WebAuthn.sol:34:     address constant VERIFIER = address(0x100);
```

Coinbase's response:
We agreed with the recommendation and implemented the changes in this PR, setting all constants to private visibility.

I-3. Function ordering does not follow the Solidity style guide

According to the Solidity style guide, functions should be laid out in the following order :`constructor()`, `receive()`, `fallback()`, `external`, `public`, `internal`, `private`, but the cases below do not follow this pattern
Affected code:

```
Unset
# File: src/ERC1271.sol
ERC1271.sol:1:
              Current order:
              public isValidSignature
              public replaySafeHash
              external eip712Domain
              public domainSeparator
              internal _eip712Hash
              internal _hashStruct
              internal _domainNameAndVersion
              internal _validateSignature

              Suggested order:
              external eip712Domain
              public isValidSignature
              public replaySafeHash
              public domainSeparator
              internal _eip712Hash
              internal _hashStruct
              internal _domainNameAndVersion
              internal _validateSignature


# File: src/ERC4337Factory.sol
ERC4337Factory.sol:1:
                     Current order:
                     public createAccount
                     external getAddress
                     public initCodeHash
                     internal _getSalt

                     Suggested order:
                     external getAddress
```

```
                        public createAccount
                        public initCodeHash
                        internal _getSalt
```

Coinbase's response:
We agreed with the recommendation and implemented the changes in this PR, specifically in this commit.

I-4. Consider using named mappings

Consider moving to solidity version 0.8.18 or later, and using named mappings to make it easier to understand the purpose of each mapping.
Affected code:

```
Unset
# File: src/MultiOwnable.sol
MultiOwnable.sol:15:    mapping(uint8 => bytes) ownerAtIndex;
MultiOwnable.sol:16:    mapping(bytes => bool) isOwner;
```

Coinbase's response:
We agreed with the recommendation and implemented the changes in this PR, specifically in this commit.

I-5. address shouldn't be hard-coded

It is often better to declare `address`es as `immutable`, and assign them via constructor arguments. This allows the code to remain the same across deployments on different networks, and avoids recompilation when addresses need to change.
Affected code:

```
Unset
# File: src/ERC4337Account.sol
ERC4337Account.sol:162:        return 0x5FF137D4b0FDCD49DcA30c7CF57E578a026d2789;
```

Coinbase's response:
Addressed here

I-6. Internal and private variables and functions names should begin with an underscore

According to the Solidity Style Guide, Non-`external` variable and function names should begin with an underscore
Affected code:

```
Unset
# File: src/WebAuthn.sol
WebAuthn.sol:86:     function verify()
```

Coinbase's response:
While we agree this does not follow the style guide, in practice Solidity libraries often use internal functions without prefixes. We want an internal function so that this library is included in the contract's bytecode, which optimizes gas, and we do not like the ergonomics of calling WebAuthn._verify().

I-7. Variables need not be initialized to zero

The default value for variables is zero, so initializing them to zero is superfluous.
Affected code:

```
Unset
# File: src/ERC4337Account.sol
ERC4337Account.sol:151:        for (uint256 i = 0; i < calls.length;) {...}

# File: src/MultiOwnable.sol
MultiOwnable.sol:95:        for (uint256 i = 0; i < owners.length; i++) {....}
```

Coinbase's response:
We agreed with the recommendation and implemented the changes in this PR, specifically in this commit.

certora coinbase WALLET

Formal Verification

Assumptions and Simplifications Made During Verification

General Assumptions

A. Any loop was unrolled to two iterations at most.
B. `optimisticFallback` flag. When it's enabled, the low-level call will either execute the fallback function in the specified contract, revert, or execute a transfer (documentation).

Code refactoring and explicit summarizations of internal parts of the code

- Functions `SignatureCheckerLib.isValidSignatureNow()` and `SignatureCheckerLib.isValidSignatureNowCalldata()` were summurized with CVL implementation to ease the verification process. They approximate to return a deterministic value based on the timestamp and all parameters of the function.
- Low-level calls cannot be generally modeled. But assumed to only affect the storage of other contracts, which are not the main contract.
    - Modifier `payPrefund()` was refactored to Solidity, keeping the original intention of the modifier: calling Entrypoint.
    - Function `upgradeToAndCall()` was refactored to verify the implementation of the contract, not the proxy. Therefore, upgrade functionality was omitted.

Formal Verification Properties

Notations

✅ Indicates the rule is formally verified.
❌ Indicates the rule is violated.

Since the protocol consists of different contracts, we will present the relative properties for each of the main contracts in separate sections.

The following files were formally verified, and the properties are listed below per library/contract:

A. ERC4337Account.sol

ERC4337Account.sol

Assumptions

- We verified the contract functions against an arbitrary storage state.

Properties

1. ✅ After `initialize()` has been called, `nextOwnerIndex > 0`.
   (*afterInitialize*)
   a. Owner's array passed as a function argument isn't empty, otherwise the function does nothing.
2. ✅ `initialize()` can't be called twice.
   (*cantInitTwice*)
   a. Owner's array passed as a function argument to the first `initialize()` isn't empty, otherwise the function does nothing and the second `initialize()` call won't revert.
3. ✅ `nextOwnerIndex` should increase monotonically.
   (newUserIndexMonotonicGrowth)
4. ✅ After initialization, if `isOwner[i]` changes, `msg.sender` must be an owner
   (*onlyOwnerCanChangeIsOwnerBytes*)
   a. `initialize()` function isn't check since it's a state "after initialization" and it was proved that `initialize()` can't be called twice.
5. ✅ After initialization, if `ownerAtIndex[i]` changes, `msg.sender` must be an owner
   (*onlyOwnerCanChangeOwnerAtIndex*)
   a. `initialize()` function isn't check since it's a state "after initialization" and it was proved that `initialize()` can't be called twice.

6. ✅ Only owner or self can call:
    a. `addOwnerPublicKey`;
    b. `addOwnerAddress`;
    c. `removeOwnerAtIndex`;
    d. `upgradeToAndCall`.

   (*OnlyOwnerOrSelf*)
7. ✅ Only EntryPoint, owner, or self can call:
    a. `execute`;
    b. `executeBatch`.

   (*OnlyOwnerSelfOrEntryPoint*)
8. ✅ Only EntryPoint can call:
    a. `executeWithoutChainIdValidation`;
    b. `validateUserOp`.

   (*OnlyEntryPoint*)
9. ✅ When we add an owner and index isn't `max_uint8` then only the latest index was changed and the length is increased by 1.
   (*addNewOwnerCheck*)
10. ✅ Can't have the same owner at two different indices in `ownerAtIndex`.
    (*notTheSameOwnerAgain*)
11. ✅ For any index `i` if `ownerAtIndex[i].length != 0` then `isOwner[ownerAtIndex[i]] == true`.
    (*notTheSameOwnerAgain*)
12. ✅ Owner with `length == 0` can't be an owner.
    (*emptyInNotAnOwner*)
13. ✅ Unless it's the `initialize()` function, index should grow by 1 only.
    (*newUserIndexGrowsBy1*)
14. ✅ There is no owner at index greater or equal to `nextOwnerIndex` unless `nextOwnerIndex == max_uint8`.
    (*noMoreThanNextOwnerIndex*)
15. ✅ ETH balance of an account should decrease by at least `missingAccountFunds` (if the `msg.value` is 0).
    (*ethBalanceDecreaseByMissingAccountFunds*)

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.