AMBIRE

Ambire Wallet - Invitational Findings & Analysis Report

2023-08-04

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Ambire Wallet smart contract system written in Solidity. The audit took place between May 23 - May 26 2023.

Following the C4 audit, 3 wardens (adriro, carlitox477 and rbserver) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports:

1. adriro
2. bin2chen
3. carlitox477
4. d3e4
5. rbserver

This audit was judged by Picodes.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 2 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Ambire Wallet - Invitational repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 329 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (5)

[M-01] Fallback handlers can trick users into calling functions of the AmbireAccount contract

Submitted by adriro

Selector clashing can be used to trick users into calling base functions of the wallet.

Fallback handlers provide extensibility to the Ambire wallet. The main idea here is that functions not present in the wallet implementation are delegated to the fallback handler by using the `fallback()` function.

Function dispatch in Solidity is done using function selectors. Selectors are represented by the first 4 bytes of the keccak hash of the function signature (name + argument types). It is possible (and not computationally difficult) to find different functions that have the same selector.

This means that a malicious actor can craft a fallback handler with a function signature carefully selected to match one of the functions present in the base AmbireAccount contract, and with an innocent looking implementation. While the fallback implementation may seem harmless, this function, when called, will actually trigger the function in the base AmbireAccount contract. This can be used, for example, to hide a call to `setAddrPrivilege()` which could be used to grant control of the wallet to the malicious actor.

This is similar to the exploit reported on proxies in this article, which caused the proposal of the transparent proxy pattern.

As further reference, another similar issue can be found in the DebtDAO audit that could lead to unnoticed calls due to selector clashing (disclaimer: the linked report is authored by me).

Recommendation

It is difficult to provide a recommendation based on the current design of contracts. Any whitelisting or validation around the selector won't work as the main entry point of the wallet is the AmbireAccount contract itself. The solution would need to be based on something similar to what was proposed for transparent proxies, which involves segmenting the calls to avoid clashing, but this could cripple the functionality and simplicity of the wallet.

Ivshti (Ambire) commented:

> I'm not sure if this is applicable: the use case of this is the Ambire team pushing out fallback handlers and allowing users to opt into them. While this does leave an opportunity for us to be that malicious actor, I'm not sure there's a better trade off here.

Picodes (judge) commented:

> The scenario is convincing; provided the attacker manages to have its malicious implementation of `fallbackHandler` used by Ambire wallet users, which seems unlikely, but doable. Furthermore, as there are no admin roles here, the possibility of this attack by the Ambire team is worth stating.

> Overall, I think Medium severity is appropriate. I agree with the previous comments that there is no clear mitigation though, aside from warning users about this.

[M-02] Attacker can force the failure of transactions that use `tryCatch`

Submitted by adriro

An attacker, or malicious relayer, can force the failure of transactions that rely on `tryCatch()` by carefully choosing the gas limit.

The `tryCatch()` function present in the AmbireAccount contract can be used to execute a call in the context of a wallet, which is eventually allowed to fail; i.e. the operation doesn't revert if the call fails.

```
function tryCatch(address to, uint256 value, byte
        require(msg.sender == address(this), 'ONL
        (bool success, bytes memory returnData) =
        if (!success) emit LogErr(to, value, data
}
```

EIP-150 introduces the "rule of 1/64th" in which 1/64th of the available gas is reserved in the calling context and the rest of it is forward to the external call. This means that, potentially, the called function can run out of gas, while the calling context may have some gas to eventually continue and finish execution successfully.

A malicious relayer, or a malicious actor that front-runs the transaction, can carefully choose the gas limit to make the call to `tryCatch()` fail due out of gas, while still saving some gas in the main context to continue execution. Even if the underlying call in `tryCatch()` would succeed, an attacker can force its failure while the main call to the wallet is successfully executed.

Proof of Concept

The following test reproduces the attack. The user creates a transaction to execute a call using `tryCatch()` to a function of the `TestTryCatch` contract, which simulates some operations that consume gas. The attacker then executes the bundle by carefully choosing the gas limit (450,000 units of gas in this case) so that the call to `TestTryCatch` fails due to out of gas, but the main call to `execute()` in the wallet (here simplified by using `executeBySender()` to avoid signatures) gets correctly executed.

Note: the snippet shows only the relevant code for the test. Full test file can be found here.

```
contract TestTryCatch {
    uint256[20] public foo;

    function test() external {
        // simulate expensive operation
        for (uint256 index = 0; index < 20; index++) {
            foo[index] = index + 1;
        }
    }
}

function test_AmbireAccount_ForceFailTryCatch() public {
    address user = makeAddr("user");

    address[] memory addrs = new address[](1);
    addrs[0] = user;
    AmbireAccount account = new AmbireAccount(addrs);

    TestTryCatch testTryCatch = new TestTryCatch();

    AmbireAccount.Transaction[] memory txns = new AmbireA
    txns[0].to = address(account);
    txns[0].value = 0;
    txns[0].data = abi.encodeWithSelector(
        AmbireAccount.tryCatch.selector,
        address(testTryCatch),
        uint256(0),
        abi.encodeWithSelector(TestTryCatch.test.selector
    );
```

```
        // This should actually be a call to "execute", we si
        // to avoid the complexity of providing a signature.
        vm.expectEmit(true, false, false, false);
        emit LogErr(address(testTryCatch), 0, "", "");
        vm.prank(user);
        account.executeBySender{gas: 450_000}(txns);

        // assert call to TestTryCatch failed
        assertEq(testTryCatch.foo(0), 0);
    }
```

Recommendation

The context that calls in `tryCatch()` , can check the remaining gas after the call to determine if the remaining amount is greater than 1/64 of the available gas, before the external call.

```
     function tryCatch(address to, uint256 value, bytes call
         require(msg.sender == address(this), 'ONLY_IDENTITY
+        uint256 gasBefore = gasleft();
         (bool success, bytes memory returnData) = to.call{
+        require(gasleft() > gasBefore/64);
         if (!success) emit LogErr(to, value, data, returnDa
     }
```

Ivshti (Ambire) commented:

> @Picodes - we tend to disagree with the severity here. Gas attacks are possible in almost all cases of using Ambire accounts through a relayer. It's an inherent design compromise of ERC-4337 as well, and the only way to counter it is with appropriate offchain checks/reputation systems and griefing protections.

> Also, the solution seems too finicky. What if the `tryCatch` is called

> within execute (which it very likely will), which requires even more gas left to complete? Then 1) the solution won't be reliable 2) the attacker can make the attack anyway through execute

Picodes (judge) commented:

> The main issue here is, the nonce is incremented, despite the fact the transaction wasn't executed as intended, which would force the user to resign the payload and would be a griefing attack against the user. I do break an important invariant, which if the nonce is incremented, the transaction signed by the user was included as he intended.

> Also, I think this can be used within `tryCatchLimit` to pass a lower `gasLimit`: quoting EIP150: "If a call asks for more gas than the maximum allowed amount (i.e. the total amount of gas remaining in the parent after subtracting the gas cost of the call and memory expansion), do not return an OOG error; instead, if a call asks for more gas than all but one 64th of the maximum allowed amount, call with all but one 64th of the maximum allowed amount of gas (this is equivalent to a version of EIP-901 plus EIP-1142)."

Ivshti (Ambire) commented:

> @Picodes - I'm not sure I understand. The whole point of signing something that calls into `tryCatch` is that you don't care about the case where the nonce is incremented, but the transaction is failing. What am I missing?

Picodes (judge) commented:

> The whole point of signing something that calls into `tryCatch` is that you don't care about the case where the nonce is incremented but the transaction is failing

> You don’t care if the transactions fail because the sub-call is invalid, but you do if it’s because the relayer manipulated the gas, right?

Ivshti (Ambire) commented:

> @Picodes - Ok, I see the point here - probably repeating stuff that others said before, but trying to simplify. The relayer can rug users by taking their fee, regardless of the fact that the inner transactions fail, due to the relayer using a lower `gasLimit`. This would be possible if some of the sub-transactions use `tryCatch`, but the fee payment does not.

> However, I’m not sure how the mitigation would work. Can the relayer still calculate a “right” gas limit for which the `tryCatch` will fail, but the rest will succeed?

Picodes (judge) commented:

> My understanding is that using `gasleft() > gasBefore/64`, we know for sure than the inner call didn’t fail due to an out of gas, as it was called with `63*gasBefore/64`. So the relayer has to give enough gas for every subcall to execute fully, whether it is successful or not.

Ivshti (Ambire) commented:

> I see, this sounds reasonable. I need a bit more time to think about it and if it is, we’ll apply this mitigation.

Ambire mitigated:

> Check gasleft to prevent this attack.

Status: Not fully mitigated. Full details in reports from adriro and carlitox477, and also included in the Mitigation Review section below.

[M-03] Recovery transaction can be replayed after a cancellation

Submitted by adriro*, also found by* bin2chen

The recovery transaction can be replayed after a cancellation of the recovery procedure, reinstating the recovery mechanism.

The Ambire wallet provides a recovery mechanism in which a privilege can recover access to the wallet if they lose their keys. The process contains three parts; all of them considered in the `execute()` function:

1. A transaction including a signature with `SIGMODE_RECOVER` mode enqueues the transaction to be executed after the defined timelock. This action should include a signature by one of the defined recovery keys to be valid.
2. This can be followed by two paths; the cancellation of the process or the execution of the recovery:

    - If the timelock passes, then anyone can complete the execution of the originally submitted bundle.
    - A signed cancellation can be submitted to abort the recovery process, which clears the state of `scheduledRecoveries`.

Since nonces are only incremented when the bundle is executed, the call that triggers the recovery procedure can be replayed as long as the nonce stays the same.

This means that the recovery process can be re-initiated after a cancellation is issued by replaying the original call that initiated the procedure.

Note: this also works for cancellations. If the submitted recovery bundle is the same, then a cancellation can be replayed if the recovery process is initiated again while under the same nonce value.

Proof of Concept

1. Recovery process is initiated using a transaction with `SIGMODE_RECOVER` signature mode.
2. Procedure is canceled by executing a signed call with `SIGMODE_CANCEL` signature mode.
3. Recovery can be re-initiated by replaying the transaction from step 1.

Recommendation

Increment the nonce during a cancellation. This will stop the nonce, preventing any previous signature from being replayed.

```
...
  if (isCancellation) {
    delete scheduledRecoveries[hash];
+   nonce = currentNonce + 1;
    emit LogRecoveryCancelled(hash, recoveryInfoHash, rec
  } else {
    scheduledRecoveries[hash] = block.timestamp + recover
    emit LogRecoveryScheduled(hash, recoveryInfoHash, rec
  }
  return;
  ...
```

Ivshti (Ambire) commented:

> Excellent finding.