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, bytes callc
            require(msg.sender == address(this), 'ONLY_IDENT
            (bool success, bytes memory returnData) = to.cal
            if (!success) emit LogErr(to, value, data, retur
    }
```

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 AmbireAccount.
        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 simplify
        // to avoid the complexity of providing a signature. Core is
        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 calldata da
          require(msg.sender == address(this), 'ONLY_IDENTITY_CAN_CA
+         uint256 gasBefore = gasleft();
          (bool success, bytes memory returnData) = to.call{ value:
+         require(gasleft() > gasBefore/64);
```

```
        if (!success) emit LogErr(to, value, data, returnData);
    }
```

[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-90[1] plus EIP-114[2])."

[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, recoveryKe
  } else {
    scheduledRecoveries[hash] = block.timestamp + recoveryInfo.t
    emit LogRecoveryScheduled(hash, recoveryInfoHash, recoveryKe
  }
  return;
  ...
```

Ivshti (Ambire) commented:

> Excellent finding.

Ivshti (Ambire) commented:

> Solved.

Ambire mitigated:

> Increment the nonce to prevent replaying recovery transactions.

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

[M-04] Project may fail to be deployed to chains not compatible with Shanghai hardfork

Submitted by [adriro]

Current settings may produce incompatible bytecode with some of the chains supported by the protocol.

The Ambire wallet supports and targets different chains, such as Ethereum, Polygon, Avalanche, BNB, Optimism, Arbitrum, etc. This information is available on their website.

All of the contracts in scope have the version pragma fixed to be compiled using Solidity 0.8.20. This new version of the compiler uses the new `PUSH0` opcode introduced in the Shanghai hard fork, which is now the default EVM version in the compiler and the one being currently used to compile the project.

Here is an excerpt of the bytecode produced for the `AmbireAccount` contract, in which we can see the presence of the `PUSH0` opcode (full bytecode can be found in the file `artifacts/contracts/AmbireAccount.sol/AmbireAccount.json`):

byecode

This means that the produced bytecode for the different contracts won't be compatible with the chains that don't yet support the Shanghai hard fork.

This could also become a problem if different versions of Solidity are used to compile contracts for different chains. The differences in bytecode between versions can impact the deterministic nature of contract addresses, potentially breaking counterfactuality.

Recommendation

Change the Solidity compiler version to 0.8.19 or define an evm version, which is compatible across all of the intended chains to be supported by the protocol (see https://book.getfoundry.sh/reference/config/solidity-compiler?highlight=evm_vers#evm_version).

Ivshti (Ambire) commented:

> Valid finding. Do you know of any big mainstream chains that do not support PUSH0?

Picodes (judge) commented:

> @Ivshti - I haven't checked for myself, but it seems Arbitrum doesn't support PUSH0 yet. For example https://github.com/ethereum/solidity/issues/14254

Picodes (judge) commented:

> Regarding the severity of the finding, I don't think the generic finding of "this contract uses 0.8.20, so won't work on some L2s" is of Medium severity as there is 0 chance that this leads to a loss of funds in production (the team will obviously see that it doesn't work and just change the compiler version).

> However, in this context, I do agree with the warden that "the differences in bytecode between versions can impact the deterministic nature of contract addresses, potentially breaking counterfactuality". Therefore, Medium severity seems appropriate.

Ivshti (Ambire) commented:

> Solved.

Ambire mitigated:

> Downgrade Solidity to allow deploying on pre-Shanghai networks.

Status: Mitigation confirmed. Full details in reports from adriro, carlitox477, and rbserver.

[M-05] AmbireAccount implementation can be destroyed by privileges

Submitted by adriro

The `AmbireAccount` implementation can be destroyed, resulting in the bricking of all associated wallets.

The `AmbireAccount` contract has a constructor that sets up privileges. These are essentially addresses that have control over the wallet.

```
	constructor(address[] memory addrs) {
		uint256 len = addrs.length;
		for (uint256 i = 0; i < len; i++) {
			// NOTE: privileges[] can be set to any
			privileges[addrs[i]] = bytes32(uint(1));
			emit LogPrivilegeChanged(addrs[i], bytes
		}
	}
```

Normally, this constructor is not really used, as wallets are deployed using proxies. The proxy constructor is the actual piece of code that sets up the privileges storage to grant initial permission to the owner of the wallet.

However, these proxies need to rely on a reference implementation of the `AmbireAccount` contract. A single contract is deployed and its address is then injected into the proxy code.

The main issue is, privileges defined in the reference implementation have control over that instance, and could eventually force a destruction of the contract using a fallback handler with a `selfdestruct` instruction (see PoC for a detailed explanation). This destruction of the implementation would render all wallets non-functional, as the proxies won't have any underlying logic code. Consequently, wallets would become inaccessible, resulting in a potential loss of funds.

It is not clear of the purpose of this constructor in the `AmbireAccount` contract. It may be present to facilitate testing. This issue can be triggered by a malicious deployer (or any of the defined privileges) or by simply setting up a wrong privilege accidentally. Nevertheless, its presence imposes a big and unneeded security risk, as the destruction of the reference implementation can render **all** wallets useless and inaccessible.

Proof of Concept

The following test reproduces the described issue. A deployer account deploys the implementation of the `AmbireAccount` contract that is later used by the user account to create a proxy (`AccountProxy` contract) over the implementation. The deployer then forces the destruction of the reference implementation, using a fallback handler (`Destroyer` contract). The user's wallet is now inaccessible, as there is no code behind the proxy.

The majority of the test is implemented in the `setUp()` function, in order to properly test the destruction of the contract (in Foundry, contracts are deleted when the test is finalized).

Note: the snippet shows only the relevant code for the test. Full test file can be found [here](#).

```
contract Destroyer {
    function destruct() external {
        selfdestruct(payable(address(0)));
    }
}

contract AccountProxy is ERC1967Proxy {
    // Simulate privileges storage
    mapping(address => bytes32) public privileges;

    constructor(address[] memory addrs, address _logic) ERC1967I
            uint256 len = addrs.length;
            for (uint256 i = 0; i < len; i++) {
                    // NOTE: privileges[] can be set to any
                    privileges[addrs[i]] = bytes32(uint(1));
            }
        }
}

contract AuditDestructTest is Test {
    AmbireAccount implementation;
    AmbireAccount wallet;

    function setUp() public {
        // Master account implementation can be destroyed by any
        address deployer = makeAddr("deployer");
        address user = makeAddr("user");
```

```
        // Lets say deployer creates reference implementation
        address[] memory addrsImpl = new address[](1);
        addrsImpl[0] = deployer;
        implementation = new AmbireAccount(addrsImpl);

        // User deploys wallet
        address[] memory addrsWallet = new address[](1);
        addrsWallet[0] = user;
        wallet = AmbireAccount(payable(
            new AccountProxy(addrsWallet, address(implementatior
        ));

        // Test the wallet is working ok
        assertTrue(wallet.supportsInterface(0x4e2312e0));

        // Now privilege sets fallback
        Destroyer destroyer = new Destroyer();
        AmbireAccount.Transaction[] memory txns = new AmbireAccc
        txns[0].to = address(implementation);
        txns[0].value = 0;
        txns[0].data = abi.encodeWithSelector(
            AmbireAccount.setAddrPrivilege.selector,
            address(0x6969),
            bytes32(uint256(uint160(address(destroyer))))
        );
        vm.prank(deployer);
        implementation.executeBySender(txns);

        // and destroys master implementation
        Destroyer(address(implementation)).destruct();
    }

    function test_AmbireAccount_DestroyImplementation() public {
        // Assert implementation has been destroyed
        assertEq(address(implementation).code.length, 0);

        // Now every wallet (proxy) that points to this master i
        wallet.supportsInterface(0x4e2312e0);
    }
}
```

Recommendation

Remove the constructor from the `AmbireAccount` contract.

[Picodes (judge) commented](#):

> There is a constructor but no initializer; so I don't get how a wallet could be deployed behind a minimal proxy: how do you set the first privilege addresses?

> It seems that either the constructor needs to be changed to an initializer, or the intent is to deploy the whole bytecode for every wallet.

[Ivshti (Ambire) commented](#):

> @Picodes - we use a completely different mechanism in which we generate bytecode, which directly `SSTORES` the relevant `privileges` slots.

> We absolutely disagree with using an initializer. It is leaving too much room for error, as it can be seen from the two Parity exploits.

> That said, this finding is valid, and removing the constructor is one solution. Another is ensuring we deploy the implementation with no privileges.

[Ivshti (Ambire) commented](#):

> @Picodes - we are in the process of fixing this by removing the constructor.

> I would say this finding is excellent, but I am considering whether the severity should be degraded, as once the implementation is deployed with empty `privileges`, this issue doesn't exist. You can argue that this creates sort of a "trusted setup", where someone needs to watch what we're deploying, but this is a fundamental effect anyway, as someone needs to watch whether we're deploying the right code. The way we'll mitigate this in the future when we're deploying is by pre-signing deployment transactions with different gas prices and different networks and placing them on github for people to review, or even broadcast themselves.

[Ivshti (Ambire) commented](#):

> @Picodes - we decided to remove the constructor, because it just makes things more obvious (production privileges are not set via the constructor).

> With that said, I just remembered that this vulnerability is mitigated by the fact the implementation will be deployed via `CREATE2` and can be re-deployed

Picodes (judge) commented:

> So:

- This is in the end a trust issue and the report shows how the team could have used the constructor to grief users.
- The fact that the implementation is deployed via `CREATE2` doesn't change the severity, as the team could still be malicious. If anything, it makes it even worse because it creates a scenario where the team could blackmail users to get paid for the redeployment.

> Overall, considering that there shouldn't be any trust assumption in this repository, I think Medium severity is appropriate, under "the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements".

Note: for full discussion, see here

Ambire mitigated:

> To mitigate this and avoid confusion, we removed the constructor as it's not used anyway.

Status: Mitigation confirmed. Full details in reports from adriro, carlitox477, and rbserver.

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **d3e4** received the top score from the judge.

The following wardens also submitted reports: carlitox477, adriro, rbserver and bin2chen.

[L-01] `AmbireAccountFactory.deploySafe()` does not guarantee that no call hasn't already been made on the deployed contract

When deploying and executing (`AmbireAccountFactory.sol#L24`), it is possible that another one of the privileged signers might have made a call on the already deployed contract, changing its state. While this can only be one of the designated signers with privilege as set by the deployer, it may be against the wishes of the deployer that someone else makes a first function call. Consider allowing only a single address with privilege in the constructor, such that the deployer would be the only one who could make a first call.

[L-02] Fallback handler should not be allowed to be `this`

In `AmbireAccount.sol` , if `address(uint160(uint(privileges[FALLBACK_HANDLER_SLOT]))) == address(this)` , anyone could call the functions protected by `require(msg.sender == address(this), 'ONLY_IDENTITY_CAN_CALL');` , with obvious and disastrous consequences. This seems like an unnecessary attack surface to expose. Consider checking this is not the case, either when setting the privileges (in `setAddrPrivilege()`) or in the fallback itself.

[L-03] Schnorr signature validation may be incompatible with the intended signers

The Schnorr signature scheme implemented for validation in `SignatureValidator.recoverAddrImpl()` (`SignatureValidator.sol#L63-L82`) is engineered to leverage Ethereums `ecrecover` for its calculations. It also uses a specific hash function (keccak256 of a certain encoding of data). As far as I can tell, this is not a standard Schnorr scheme. It is important to note, that both the signer and validator must agree on the same group and hash function. The implemented Schnorr validation will not work with any other Schnorr scheme. Consider whether `AmbireAccount` is expected to interface with other Schnorr scheme implementations, and if so, make sure that the same Schnorr scheme is used.

[L-04] Schnorr signature length is not checked

When `SignatureValidator.recoverAddrImpl()` (`SignatureValidator.sol#L63`) is called with a Schnorr signature, it is not checked that it has the correct length, unlike for `SignatureMode.EIP712` and `SignatureMode.EthSign`. Consider checking that `sig`, before trimming, has a length of `129` (`(bytes32, bytes32, bytes32, uint8)` plus the `modeRaw` byte).

[L-05] `LogPrivilegeChanged` does not adequately describe the change

`LogPrivilegeChanged(addr, priv)` is emitted when the privilege of `addr` is changed to `priv` (`AmbireAccount.sol#L115`) (also in the constructor, but there it is first set, rather than changed). Since the previous value is not emitted, it is difficult to know whether and how it was meaningfully changed; especially considering that privileges are `bytes32`, but generally carry their meaning only in being non-zero, but may also encode for recovery and the fallback handler. Consider including the previous privilege in the event, and perhaps emit a different event when the fallback handler is changed and when the recovery info hash is set. This would then probably involve creating a separate function for setting this.

[L-06] Consider indexing unindexed events

Instances:

`AmbireAccountFactory.sol#L7`

[L-07] Error message with opposite meaning

The anti-bricking checks return `PRIVILEGE_NOT_DOWNGRADED` if the sender/signer key removes their own privilege. If this is attempted, this error message suggests that the privilege should have been downgraded but wasn't, which is the opposite of what is intended. The error message should therefore rather be `PRIVILEGE_DOWNGRADED` or `PRIVILEGE_MUST_NOT_BE_DOWNGRADED` or similar.

Instances:

`AmbireAccount.sol#L193` `AmbireAccount.sol#L207`

[L-08] Non-scheduled recoveries can be cancelled

When a recovery is cancelled, `LogRecoveryCancelled` is emitted (`AmbireAccount.sol#L173`). This happens even if the recovery wasn't previously scheduled, giving a false impression that it was. Consider reverting attempts to cancel a recovery that hasn't already been scheduled.

[L-09] `AmbireAccount.constructor()` does not allow for custom privileges

`AmbireAccount.constructor()` only sets `privileges[addrs[i]] = bytes32(uint(1));` (`AmbireAccount.sol#L62`). A second call is necessary to set the remaining `privileges`, at `FALLBACK_HANDLER_SLOT` and the value `recoveryInfoHash`. Consider adding a parameter with the values to set at `addrs`.

[L-10] It makes more sense to check `signatures.length > 0` than `signer != address(0)` for multisigs

In `SignatureValidator.recoverAddrImpl()` for `SignatureMode.Multisig`, it is checked that last `signer != address(0)` (`SignatureValidator.sol#L92`) after validating each signature in the array `signatures`. This can be `address(0)` only if `signatures.length == 0`, in which case the for-loop is skipped, leaving `signer` unassigned. It would make more sense to instead check `require(signatures.length != 0, 'SV_ZERO_SIG');`, just after L85.

[L-11] Redundant require/revert

In `SignatureValidator.recoverAddrImpl()` it is first checked that `require(modeRaw < uint8(SignatureMode.LastUnused), 'SV_SIGMODE');` (`SignatureValidator.sol#L49`). This ensures that `SignatureMode mode = SignatureMode(modeRaw);` will be one of the available signature modes. Each of these modes is then considered and the function returns in each case. But at the end of the function, there is a `revert('SV_TYPE');` (`SignatureValidator.sol#L120`). This line can therefore not be reached. Consider removing either of these checks, as they have the same effect.

[L-12] Group order denoted `Q` may be confused with the public key

In `SignatureValidator.sol`, the Schnorr signature scheme group order is denoted `Q`. While in the context of Schnorr signatures, a lowercase `'q'` is sometimes used to denote the group order. This particular Schnorr signature scheme uses the subgroup generated by the secp256k1 base point, the order of which is usually denoted `n`. I.e. the value which is here denoted `Q` is usually known as `n`. Furthermore, secp256k1 is primarily thought of in the context of ECDSA, where `Q` usually denotes the public key. Consider renaming `Q` to `n`.

[L-13] Use `uint256` instead of `uint`

Consider using the explicit `uint256` consistently instead of its alias `uint`.

Instances:

`AmbireAccount.sol#L15`

`AmbireAccount.sol#L62`

`AmbireAccount.sol#L63`

`AmbireAccount.sol#L94`

[L-14] Typos

Instances:

contracft -> contract (`AmbireAccountFactory.sol#L15`)

`// bytes4(keccak256("isValidSignature(bytes32,bytes)") -> // bytes4(keccak256("isValidSignature(bytes32,bytes)"))`

(`AmbireAccount.sol#L243`)

[L-15] `require(sp != 0);` fails to protect Schnorr signature validation

An incorrect check drastically reduces the security of Schnorr validation. Whether it is completely broken depends on whether fixed points can be found for `keccak256(f(x))`.

Proof of Concept

Schnorr signatures are validated like this:

```
} else if (mode == SignatureMode.Schnorr) {
// px := public key x-coord
// e := schnorr signature challenge
// s := schnorr signature
// parity := public key y-coord parity (27 or 28)
// last uint8 is for the Ambire sig mode - it's ignored
sig.trimToSize(sig.length - 1);
(bytes32 px, bytes32 e, bytes32 s, uint8 parity) = abi.decode(si
// ecrecover = (m, v, r, s);
bytes32 sp = bytes32(Q - mulmod(uint256(s), uint256(px), Q));
bytes32 ep = bytes32(Q - mulmod(uint256(e), uint256(px), Q));

require(sp != 0);
// the ecrecover precompile implementation checks that the `r` a
// inputs are non-zero (in this case, `px` and `ep`), thus we dc
// check if they're zero.
address R = ecrecover(sp, parity, px, ep);
require(R != address(0), 'SV_ZERO_SIG');
require(e == keccak256(abi.encodePacked(R, uint8(parity), px, ha
return address(uint160(uint256(px)));
```

`ecrecover(sp, parity, px, ep)` `ecrecover(m, v, r, s)` returns a hash of `(1/r) * (s*P - m*G)`, where `P` is a point derived from `r` and `v`, and the operations are to be interpreted as elliptic curve point operations. In our case, where `ecrecover(sp, parity, px, ep)`, we get `(1/px) * (ep*P - sp*G)`. We see that if `sp % Q == 0` then `G*sp == O`, where `O` is the identity; because `Q` is the order of `G`, so `(1/px) * (ep*P - sp*G) = ep/px * P`. This means we could make ANY public key `px` pass validation in `ecrecover`.

Achieving `sp % Q == 0` should have been prevented by `require(sp != 0);`, which fails to consider that `sp == Q` is equally impermissible. In fact, the check cannot trigger because `sp > 0` since `mulmod(uint256(s), uint256(px), Q) < Q`. So if we simply let `s == 0`, which we are free to do, then `sp` evaluates to `bytes32(Q - mulmod(0, uint256(px), Q)) == bytes32(Q - 0) == bytes32(Q)`, which leads to the issue described above.

If this was normal ECDSA validation it would already have been broken. But we now also require that `e == keccak256(abi.encodePacked(R, uint8(parity), px, hash))`. Recall that `R = ep/px * P`. We can therefore consider `R` a function of `e`. `R` is then hashed with `px` and `hash`, which are predetermined values that we want to attack. This line can therefore be thought of as a keccak256-based hash function `H` of `e`. Therefore, we have the problem of finding an `e` such that `e == H(e)`.

Finding this enables the attacker to validate and execute any transaction hash.

keccak256 is considered safe against pre-image attacks, i.e. given `y` find `x` such that `y == keccak256(x)`. But finding a fixed point, i.e. `e` such that `e == H(e)` may be considerably easier. I have been unable to confirm whether it is known to be possible for keccak256, [but it does seem possible for `sha256`]. Considering that [SHA-1 has already been broken for a chosen-prefix attack], it does not seem unrealistic that a fixed point attack will be achievable in this case.

Recommended Mitigation Steps

Fortunately, this is trivial to mitigate:

```
- require(sp != 0);
+ require(sp != Q);
```

since `0 < sp <= Q`. Or, for peace of mind, `require(sp % Q != 0)`.

Assessed type

Invalid Validation

[L-16] Transactions bundles signed for a future nonce cannot be cancelled

Transaction bundles signed for a future nonce cannot be cancelled, except by possibly and unfeasibly, many calls to `execute()`.

Proof of Concept

`AmbireAccount.execute()` validates a signature against a hash based on an incrementing nonce (`AmbireAccount.sol#L138`). Only a transaction bundle hash with the current nonce can be executed. A signature for the current nonce may thus be invalidated by signing a dummy transaction bundle, which only causes the nonce to increment, rendering the undesirable signature forever inexecutable. But if a transaction bundle is signed for a nonce in the future, either by mistake or in anticipation of a `executeMultiple()` call, the only way to cancel the signature would be to repeatedly call `execute()` until the nonce has passed. This might cost significant gas (the signed nonce might be arbitrarily high), and the transaction bundle might then be executed anyway by a frontrunner.

Recommended Mitigation Steps

Implement a mapping which stores cancelled transaction bundle hashes, and check this before executing.

Assessed type

Context

[L-17] The anti-bricking mechanism only applies to the normal mode, but not the recovery mode

`recoveryInfoHash` is critical for recovery but is not protected in the same way a signer's own privilege is protected. It is therefore possible for a recovery key to remove its own ability to recover the account, bricking the account.

Proof of Concept

The anti-bricking mechanism is supposed to prevent a signer from signing away their own privilege. This is checked by `[require(privileges[signerKey] != bytes32(0), 'PRIVILEGE_NOT_DOWNGRADED');` (`AmbireAccount.sol#L193`) at the end of `AmbireAccount.execute()` . But this presupposes that it was `signerKey` that signed the transaction just executed. This is not the case in the case of a recovery. Then `signerKey` is `signerKeyToRecover` such that `[privileges[signerKeyToRecover] == recoveryInfoHash]` (`AmbireAccount.sol#L153`). `signerKeyToRecover` is supposedly the key we are trying to recover, meaning it is lost, so it doesn't matter what happens to its privileges. We are probably about to give privilege to a new address.

It seems plausible that this new address might be incorrectly entered, and in this case it is critical that the recovery key (which signed the transaction about to be executed) cannot revoke its own ability to recover, leaving the account with no authorised signer. Recovery keys do not operate based on `privileges`, but on there being an appropriate `recoveryInfoHash` set, as noted above. Therefore this value must not be allowed to change, which would be the analogous anti-bricking mechanism for recoveries.

Recommended Mitigation Steps

At a minimum simply check that `privileges[signerKeyToRecover] == recoveryInfoHash` still.

But note that we might actually want to be able to set `privileges[signerKeyToRecover] = 0` for fear that the lost key might become compromised. Currently, this is not possible. It would be acceptable to move `recoveryInfoHash` somewhere else, i.e. `privileges[someOtherSignerKeyToRecover] = recoveryInfoHash`. Note that `someOtherSignerKeyToRecover` may actually be an arbitrary address; it is only used to retrieve `recoveryInfoHash`. So it is not critical that `someOtherSignerKeyToRecover` is a valid address that we can sign with; we can still use it to recover with.

An alternative could then be, to allow `privileges[signerKeyToRecover] == 0` for recoveries being finalised and instead require that `privileges[someOtherSignerKeyToRecover] == recoveryInfoHash` for `someOtherSignerKeyToRecover`. This can be seen as having two separate but analogous anti-bricking mechanisms, one for normal execution and one for recovery.

Assessed type

Invalid Validation

Gas Optimizations

For this audit, 2 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by **d3e4** received the top score from the judge.

The following warden also submitted a report: [adriro](#).

[G-01] `splitSignature()` may be redundant

`SignatureValidator.splitSignature()` seems to only be used once in `AmbireAccount.sol#L145`, where only its first return value is used, which is `signature` with its last byte removed. We can thus simply inline `Bytes.trimToSize()` instead:

```
+ import './Bytes.sol';
...
- (bytes memory sig, ) = SignatureValidator.splitSignature(signa
+ bytes memory sig = Bytes.trimToSize(signature, signature.lengt
```

[G-02] Cache variable

`sig.length - 1` may be cached in memory in:

```
unchecked {
    modeRaw = uint8(sig[sig.length - 1]);
}
sig.trimToSize(sig.length - 1);
```

[G-03] Unnecessary require in `AmbireAccountFactory.deploySafe()`

`AmbireAccountFactory.deploySafe(code, salt)` returns the address where the bytecode “code” is deployed itself, deploying it first if it isn’t already deployed. Therefore, it is meaningless to check that the fresh deployment is successful, which only amounts to checking that the opcode `CREATE2` works as expected. Specifically: `require(addr == expectedAddr, 'FAILED_MATCH');` in

`AmbireAccountFactory.deploySafe()` can be removed, as it must be assumed that the calculation of `expectedAddr` is correct. `require(addr != address(0), 'FAILED_DEPLOYING');` will never trigger, as `CREATE2` only returns `address(0)` when trying to deploy to the same address twice, which is not the case since it is already checked that `extcodesize(addr) == 0`.

[G-04] Unnecessary require in `SignatureValidator.recoverAddrImpl()`

`address signer = address(wallet);` so `require(signer != address(0), 'SV_ZERO_SIG');` is redundant in `SignatureValidator.recoverAddrImpl()`, as it would already have reverted in `wallet.isValidSignature(hash, sig)`, because of a function call to the zero address.

[G-05] Unnecessary declaration of `currentNonce

In `AmbireAccount.execute()` `currentNonce` is declared: `uint256 currentNonce = nonce;` (`AmbireAccount.sol#L136`), but never changed, and then used to update `nonce`: `nonce = currentNonce + 1;` (`AmbireAccount.sol#L189`). `nonce` alone can be used throughout and then incremented by `nonce++`.

[G-06] It is more gas efficient to revert with a custom error than a `require` with a string

See [Gas Optimizations Report](#) for details.

[G-07] Constructors may be declared `payable` to save gas

Instances:
AmbireAccount.sol#L58 AmbireAccountFactory.sol#L11

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (adriro, carlitox477 and rbserver) reviewed the mitigations for all identified issues. Additional details can be found within the C4 Ambire Wallet Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

We fixed 4 of the vulnerabilities after they were found, and we chose not to mitigate one.

Mitigation Review Scope

Mitigation of	Purpose
M-02	Check gasleft to prevent this attack
M-03	Increment the nonce to prevent replaying recovery transactions
M-04	Downgrade Solidity to allow deploying on pre-Shanghai networks
M-05	To mitigate this and avoid confusion, we removed the constructor as it's not used anyway

Mitigation Review Summary

Original Issue	Status	Full Details
M-02	Not fully mitigated	Reports from adriro and carlitox477, and also shared below
M-03	Not fully mitigated	Reports from adriro, carlitox477, and rbserver, and also shared below
M-04	Mitigation confirmed	Reports from adriro, carlitox477, and rbserver
M-05	Mitigation confirmed	Reports from adriro, carlitox477, and rbserver

See below for details regarding the two issues that were not fully mitigated.

Mitigation of M-02: Not fully mitigated

Submitted by adriro, also found by [carlitox477](#).

Original Issue

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

Comments

While the issue mentioned in M-02 has been technically mitigated, the same attack can be performed in another function present in the wallet.

The report describes an attack in which a malicious relayer can force the failure of calls to `tryCatch`. The issue in this specific function has been mitigated, however the same attack can be performed in the function `tryCatchLimit`.

Mitigation of M-03: Not fully mitigated

Submitted by adriro, also found by [carlitox477](#) and [rbserver](#).

Original Issue

M-03: [Recovery transaction can be replayed after a cancellation](#)

The mitigation of M-03 contains an error in the implementation of the fix. The original issue is still present.

Impact

The report in M-03 describes an issue related to the replay of the recovery transaction. After a cancellation is executed, the same transaction that initiated the recovery procedure can be replayed since the nonce is not incremented after canceling the recovery.

The intended fix is present. The updated implementation of `execute()` is as follows:

```
    function execute(Transaction[] calldata calls, bytes cal
            uint256 currentNonce = nonce;
            // NOTE: abi.encode is safer than abi.encodePack
```

```
		bytes32 hash = keccak256(abi.encode(address(this

		address signerKey;
		// Recovery signature: allows to perform timeloc
		uint8 sigMode = uint8(signature[signature.length

		if (sigMode == SIGMODE_RECOVER || sigMode == SIG
			(bytes memory sig, ) = SignatureValidato
			(RecoveryInfo memory recoveryInfo, bytes
				sig,
				(RecoveryInfo, bytes, address)
			);
			signerKey = signerKeyToRecover;
			bool isCancellation = sigMode == SIGMODE
			bytes32 recoveryInfoHash = keccak256(abi
			require(privileges[signerKeyToRecover] =

			uint256 scheduled = scheduledRecoveries[
			if (scheduled != 0 && !isCancellation) {
				require(block.timestamp > schedu
				nonce++;
				delete scheduledRecoveries[hash]
				emit LogRecoveryFinalized(hash,
			} else {
				bytes32 hashToSign = isCancellat
				address recoveryKey = SignatureV
				bool isIn;
				for (uint256 i = 0; i < recovery
					if (recoveryInfo.keys[i]
						isIn = true;
						break;
					}
				}
				require(isIn, 'RECOVERY_NOT_AUTH
				if (isCancellation) {
					delete scheduledRecoveri
					emit LogRecoveryCancelle
				} else {
					scheduledRecoveries[hash
					emit LogRecoverySchedule
				}
				return;
			}
		} else {
			signerKey = SignatureValidator.recoverAc
			require(privileges[signerKey] != bytes32
```

```
180:                }
181:
182:                // we increment the nonce to prevent reentrancy
183:                // also, we do it here as we want to reuse the p
184:                // and respectively hash upon recovery / canceli
185:                // doing this after sig verification is fine bec
186:                nonce = currentNonce + 1;
187:                executeBatch(calls);
188:
189:                // The actual anti-bricking mechanism - do not a
190:                require(privileges[signerKey] != bytes32(0), 'PR
191:        }
```

The patched line is 154. Note that this line belongs to a path that is executed when the recovery process is finally executed after the timelock, and has nothing to do with the cancellation of the process. Note also that the change is in fact a no operation, since the increment is overwritten by line 186:

1. Line 132 sets `currentNonce` to the actual value of `nonce`, say `N`.
2. Line 154 executes `nonce++` which sets the storage value of `nonce` to `N+1`.
3. Recovery bundle is executed and line 186 sets the storage value of `nonce` to `currentNonce + 1`, which means that `nonce` is still `N+1`.

The intended fix contains an error and fails to mitigate the issue. It is not clear if this was mistakenly placed in the wrong code path or if there was a confusion related to the different code paths that this function may take. In any case, the nonce is not incremented after a cancellation, which means that the replay attack is still possible.

Recommendation

The `nonce` should be incremented in the path that executes the cancellation, as recommended in the report for M-03.

```
...
  if (isCancellation) {
    delete scheduledRecoveries[hash];
+   nonce = currentNonce + 1;
    emit LogRecoveryCancelled(hash, recoveryInfoHash, recoveryKe
  } else {
    scheduledRecoveries[hash] = block.timestamp + recoveryInfo.t
```

```
        emit LogRecoveryScheduled(hash, recoveryInfoHash, recoveryKe
    }
    return;
    ...
```

Ivshti (Ambire) commented:

> Fixed.

adriro (warden) commented:

> Fixed.

> LGTM

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top