2023-09-28 Ambire Smart Contracts Audit Report

Date: 2023-09-28
Author: Facundo Spagnuolo
Commit: 0e35b32df045113298b119986b619e0a5d2328bb
Amendment: TBD

Introduction

The following report provides a comprehensive overview of various recommendations that emerged from a security review conducted on the `AmbireAccount`, `AmbirePaymaster`, and `DKIMRecoverySigValidator` contracts within the AmbireTech/ambire-common repository, specifically focusing on the commit identified as 0e35b32df045113298b119986b619e0a5d2328bb.

Assumptions

A1. Dependencies

All the dependency contracts used by `AmbireAccount`, `AmbirePaymaster`, and `DKIMRecoverySigValidator` were considered previously audited and are not part of the scope under analysis for this security review process.

A2. External signature validation method must handle nonces

The `AmbireAccount` contract makes use of a mechanism of signature validation where the decision is trusted to an external contract. This mechanism does not take into account the `AmbireAccount`'s nonce when validating signatures. Therefore, the external contract in charge of telling whether a signature is valid must handle nonces in a custom way.

A3. Paymaster relies on the account's nonce management

The `AmbirePaymaster` contract relies on the account's nonce for replay protection. In case the account is allowed to process the same operation multiple times, the paymaster signature would remain valid multiple times.

High

H1. Missing account funds may not be sent to Entry Point on external validation mode

The `AmbireAccount` contract implements the account contract standard promoted by ERC 4337. One of the critical functions that must be implemented is `validateUserOp`, in which the account contract must validate whether a signed user operation is valid or not to be executed by a third party named `EntryPoint`. This function receives three parameters: the user operation, the hash of the user operation, and the number of funds that must be transferred to the `EntryPoint` in order to cover the transaction costs.

Based on how this function is implemented, there are different ways of signature validation. One of them is an external signature validation method that trusts that decision to another contract. In case the external signature validation method is requested no funds are being transferred to the `EntryPoint`. This means that in case the account does not have enough deposit to cover the costs, the transaction won't be allowed to go through.

Note external signature validation might be used in recovery mode. Although this is not the only way to execute transactions based on an external validation flow, consider making sure to transfer the requested number of funds to the `EntryPoint` to be consistent with the ERC 4337 standard.

H2. External signature validation does not fully rely on the user operation data in some cases

The `AmbireAccount` contract implements the account contract standard promoted by ERC 4337. One of the critical functions that must be implemented is `validateUserOp`, in which the account contract must validate whether a signed user operation is valid or not to be executed by a third party named `EntryPoint`. This function receives three parameters: the user operation, the hash of the user operation, and the number of funds that must be transferred to the `EntryPoint` in order to cover the transaction costs.

The implementation provided in `AmbireAccount` distinguishes different signature validation mechanisms. One of them is an external signature validation method that trusts that decision to another contract. The problem is that only a portion of the calldata and the signature itself are passed to the external validator to process the signature (L238). Note that there are many other parameters that are part of the user operation data defined in the standard that are not included, for example: `callGasLimit`, `verificationGasLimit`, `preVerificationGas`, `maxFeePerGas`, `maxPriorityFeePerGas`, and others. This means a malicious `EntryPoint` could change those parameters of the operation which will still be validated by the account.

Consider including the user operation hash as part of the signature verification process, which already includes all the relevant information that cannot be manipulated.

Medium

M1. Signature time validation might be silently shortened

The `AmbireAccount` contract implements the account contract standard promoted by ERC 4337. One of the critical functions that must be implemented is `validateUserOp`, in which the account contract must validate whether a signed user operation is valid or not to be executed by a third party named `EntryPoint`. This function receives three parameters: the user operation, the hash of the user operation, and the number of funds that must be transferred to the `EntryPoint` in order to cover the transaction costs. This function must return a 256-bit unsigned integer with the following content:

- 20 bytes where 0 means valid signature and 1 means invalid signature or an address of an external authorizer can be given.
- 6 bytes to indicate a timestamp until when the user operation is considered valid or 0 to indicate “infinite” (`validUntil`)
- 6 bytes to indicate a timestamp after which the user operation is considered valid (`validAfter`)

Based on how this function is implemented, there are different ways of signature validation. One of them is an external signature validation method that trusts that decision to another contract. In case the external signature validation method is requested the `validAfter` returned value is computed using a timestamp represented by a 256-bit unsigned integer returned as part of the external signature validation process (see L243). Note the latter is simply padded to force 6 bytes representation which will end up simply dropping its 26 most significant bytes.

Although it sounds ridiculous to work with huge timestamps, to make sure there is no undesired behavior and enforce consistency, consider validating the timestamp returned by the external signature validation process can actually be represented using 6 bytes.

M2. The relayer address cannot be changed on the paymaster

The `AmbirePaymaster` contract implements an "un-staked" version of the paymaster contract standard promoted by ERC 4337. Paymasters are in charge of promoting account transactions by making use of a previously deposited balance in the `EntryPoint` contract.

Since the implementation provided in the `AmbirePaymaster` contract does not rely on any storage at all, it can be considered an "un-staked entity" making it possible to avoid including all the staking-related logic required by the `EntryPoint`.

However, this implementation relies on an immutable address variable called `relayer` to achieve two critical functionalities (see L13). On one hand, it is the signer of the paymaster data that will be used during the validation process of the ERC 4337 standard. On the other hand, it is the address that can trigger withdrawals from the `EntryPoint` contracts.

Consider making sure a secure scheme is picked for this address at deployment time, for example, a 1/3 multisig. It might be also useful to have two different accounts to represent the paymaster signer and the address allowed to withdraw funds from the `EntryPoint` contracts.

M3. Validate user op should treat `execute` and `executeMultiple` equally

The `AmbireAccount` contract implements the account contract standard promoted by ERC 4337. One of the critical functions that must be implemented is `validateUserOp`, in which the account contract must validate whether a signed user operation is valid or not to be executed by a third party named `EntryPoint`. This function receives three parameters: the user operation, the hash of the user operation, and the number of funds that must be transferred to the `EntryPoint` in order to cover the transaction costs.

The way this function is implemented there is an edge-case handled differently when the encoded operation is a call to the `execute` method (see here). In particular, no signature is required since the `execute` method already requires a signature to be validated.

This is also the case of `executeMultiple`, which is a function that basically calls the `execute` function many times (see L141). Consider handling the case of `executeMultiple` in `validateUserOp` to be consistent with the expected behavior.

M4. The authorized addresses cannot be changed on the `DKIMRecoverySigValidator`

The `DKIMRecoverySigValidator` contract offers a recovery mechanism to allows users to set new privileges of an `AmbireAccount` contract to an external address in case they need to.

This implementation relies on two immutable addresses called `authorizedToSubmit` and `authorizedToRevoke` to achieve two critical functionalities (see L119-L120). The first one is the one allowed to submit DKIM keys to the whitelist, and the other one is the one allowed to remove keys from the whitelist. This whitelist is optional for users, even though they are not forced to trust these keys, they might rely on them in case the trusted keys they set up for recovery are no longer valid. Note this is a possible scenario since email providers routinely rotate their DKIM keys.

The problem here is that losing access to these authorized account might also affect users willing to trigger their recovery process.

Consider making sure a secure scheme is picked for this address at deployment time, for example, a 1/3 multisig.

Low

L1. Unknown hardcoded value of a supported interface for ERC 165 implementation

The `AmbireAccount` contract implements the standard interface detection promoted by ERC 165. The standard states that the interface identifier should be computed as the XOR of all function selectors in the interface.

Based on how this method is implemented, there is an unknown hardcoded value `0x0a417632` that doesn’t seem to match the identifier of the `AmbireAccount`’s contract interface (see L202). This approach could be error-prone since interfaces might change during the development process and developers might forget to update these constants. Consider computing the interface ID using function selectors instead of using hardcoded values.

L2. Using hardcoded values for ERC 165 implementation

The `AmbireAccount` contract implements the standard interface detection promoted by ERC 165. The standard states that the interface identifier should be computed as the XOR of all function selectors in the interface.

Based on how this method is implemented, there are different interface IDs whose values are hardcoded instead of being computed as suggested in the standard. This approach could be error-prone since interfaces might change during the development process and developers might forget to update these constants. Even though many of them may refer to well-known values by the community, it is recommended to use function selectors for all the supported interfaces instead of hardcoded values.

L3. Repeated code

Many parts of the logic are repeated through the codebase. It's fully understandable from a gas optimization point of view, however, it would improve considerably the code's readability. Additionally, it will make much easier the development experience at the moment of fixing bugs or implementing new features. Otherwise, making sure to adjust the code in a repeated number of places is considered error-prone. Consider analyzing this aspect of the codebase and making sure there is a good trade-off between gas cost consumption and code readability.

L4. `tryCatch` and `tryCatchLimit` can be combined into a single method

The `AmbireAccount` contract defines two different methods `tryCatch` and `tryCatchLimit` to allow the account contract to execute calls that will be handled in case they revert by simply logging the result instead. These two functions are exactly the same, with the only difference being that `tryCatchLimit` receives an extra parameter named `gasLimit`. This third parameter allows the caller to specify a fixed number of gas that will be used to execute the desired call, while `tryCatch` will simply use all the remaining gas.

Following the suggestion mentioned in L4, these two methods can be combined into one introducing a numeric flag instead to denote whether the call should be executed using a fixed number of gas or forwarding all the available gas.

L5. Implementing logic already standardized by other libraries

The `AmbireAccount` contract supports different standards like ERC1155 Receiver, ERC721 Receiver, ERC 165, and ERC 1271. Many of them are already provided by Open Zeppelin and can be reused to make sure the required logic is not implemented from scratch.

Following the suggestion mentioned in L4, consider importing these when possible from Open Zeppelin instead.

L6. Copy-pasted Open Zeppelin libraries

The `DKIMRecoverySigValidator` contract makes use of both `Strings` and `SignedMath` libraries developed by OpenZeppelin. However, instead of relying on dependency imports, these were copy-pasted into de repository.

This approach is not recommended since it's error prone and it makes it more difficult to seize bug fixes or optimizations that can be provided by Open Zeppelin in the future. Consider importing these using a package manager's installed dependencies instead.

L7. Missing events

The `AmbireAccount` contract does not emit any event when executing transactions. This will make things harder in the future in case there is a need to track this information off-chain. Consider adding events to track this information.

L8. `DKIMRecoverySigValidator` may not trigger a DKIM recovery process

The `DKIMRecoverySigValidator` contract offers a recovery mechanism to allows users to set new privileges of an `AmbireAccount` contract to an external address in case they need to. There are three different ways of recovery:

- The first one is about using a DKIM signature to allow a third address to be granted with the new privilege to access the account.
- The second one is about using a DKIM signature along with a secondary off-chain signature to re-validate what's being signed via DKIM
- The last one is simply using an off-chain signature ignoring completely the DKIM signature validation process

The problem here is the `DKIMRecoverySigValidator` contract is not just about DKIM recovery, it may trigger a recovery process where DKIM signatures are not used at all. This makes things extremely hard when trying to follow the logic through the code. Note this is one of the causes that makes you have a method with more than 100 lines of code.

Consider splitting this contract into two contracts, one for DKIM verification and another one for off-chain signatures verification, and a third contract making use of those two in case both processes must be evaluated.

L9. The `DKIMRecoverySigValidator` lacks of indexed event parameters

The `DKIMRecoverySigValidator` contract implements some events but non of them have indexed parameters (see L102-L112). Consider indexing the ones that makes sense to make things easier for off-chain services in case they need to search and filter transaction logs.

Notes

N1. Missing or non-standard inline documentation

Consider thoroughly documenting all functions and their parameters following the Ethereum Natural Specification Format (NatSpec).

N2. Use interfaces instead of implementation contracts to reference selectors

The `AmbirePaymaster` contract relies on the `AmbireAccount` only to refer to a method selector (see L47). Consider importing the `AmbireAccount` interface instead.

N3. No custom errors

Custom errors in Solidity can improve the transparency and effectiveness of smart contract code, providing more insightful feedback for developers and smart contract consumers. It can also benefit gas consumption in some cases. Consider using custom errors instead of string literals.

N4. No magic values

The following magic values were found in the codebase:

- `0` in L229 of the `AmbireAccount` contract

- `0` in L259 of the `AmbireAccount` contract
- `address(69)` in L316 of the `DKIMRecoverySigValidator`
- `0x9b` in L352 of the `DKIMRecoverySigValidator`
- `0x9b` in L360 of the `DKIMRecoverySigValidator`
- `646e7373656362726964676506616d6269726503636f6d0000100001000001` in L382

Consider using constants for these cases instead when possible.

N5. Typos

The following typos were found in the codebase:

- “th” in L251 of the `AmbireAccount` contract
- “valdiated” in L274 of the `AmbireAccount` contract
- “existant” in L184 of the `DKIMRecoverySigValidator` contract

N6. Unsolved TODO comment on `AmbireAccount` contract

There is one TODO comment in the `AmbireAccount` contract (see L276). Consider removing it and tracking it in a backlog issue in case it deserves further analysis.

N7. Misleading inline comment on `AmbireAccount` fallback

The `AmbireAccount` contract defines a custom fallback function where the received call can be delegated to a configurable external contract. This behavior is implemented using inline assembly which starts with a `calldatacopy` instruction in order to copy the calldata to memory before performing the delegatecall.

This line is followed by a comment that states “`We can use memory addr 0, since it's not occupied`" (see L66) which is not necessarily true. The reason why memory address zero can be used instead of using the standard free memory pointer is because that logic is taking full control of the execution making sure it returns itself and does not rely on any further Solidity code.

N8. Unnecessary imports on `AmbirePaymaster` contract

The `AmbirePaymaster` contract imports the `ExternalSigValidator` contract on L5. This import is not being used, consider removing it.

N9. Declare `uint` as `uint256`

Throughout the codebase, there are some places where `uint` is being used as opposed to `uint256`. In favor of explicitness, consider replacing all instances of `uint` with `uint256`.

N**10. Inconsistent ordering**

The Solidity Style Guide specifies a recommended order for the layout of elements within a contract file in order to facilitate finding declarations grouped together in predictable locations. Consider following this recommendation to make sure the variables and functions declared in your contracts follow a consistent pattern.