Ambire Security Assessment

Report Version 0.1

January 26, 2024

Conducted by the **Hunter Security** team:
George Hunter, Lead Security Researcher

Table of Contents

1 About Hunter Security **3**

2 Disclaimer **3**

3 Risk classification **3**
3.1 Impact . . . 3
3.2 Likelihood . . . 3
3.3 Actions required by severity level . . . 3

4 Executive summary **4**

5 Consultants **5**

6 System overview **5**
6.1 Codebase maturity . . . 5
6.2 Privileged actors . . . 6
6.3 Threat model . . . 6
6.4 Observations . . . 7
6.5 Useful resources . . . 7

7 Findings **8**
7.1 Low . . . 8
7.1.1 Signer address is not included in transactions message payload . . . 8
7.1.2 EIP-165 specification not enforced . . . 9
7.1.3 Unsafe deployment possible through the Ambire factory . . . 9
7.1.4 Transactions batching can be griefed by front-running . . . 10
7.2 Informational . . . 10
7.2.1 Typographical mistakes and code-style suggestions . . . 10

8 Final remarks **12**

1 About Hunter Security

Hunter Security is a duo team of independent smart contract security researchers. Having conducted over 50 security reviews and reported tens of live smart contract security vulnerabilities, our team always strives to deliver top-quality security services to DeFi protocols. For security review inquiries, you can reach out to us on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

The Hunter Security team was engaged by Ambire to review the Ambire Wallet smart contracts during the period from January 8, 2024, to January 24, 2024.

Overview

Project Name	Ambire Wallet
Repository	https://github.com/AmbireTech/ambire-common
Commit hash	518452b901149e5c745f2bafab38588b2605ff18
Resolution	-
Methods	Manual review

Timeline

-	January 8, 2024	Audit kick-off
v0.1	January 26, 2024	Preliminary report
v1.0	-	Mitigation review

Scope

contracts/AmbireAccount.sol
contracts/AmbireAccountFactory.sol
contracts/AmbirePaymaster.sol
contracts/DKIMRecoverySigValidator.sol
contracts/libs/SignatureValidator.sol

Issues Found

High risk	0
Medium risk	0
Low risk	4
Informational	1

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, SmarDex, Ambire, and other leading protocols. George's extensive experience and meticulous attention to detail contribute to Hunter Security's reviews, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

6 System overview

Ambire Wallet V2 is an immutable smart contract wallet protocol. Users can have one or more privileged addresses attached which can execute operations on behalf of their Ambire wallet. It supports transactions batching as well as numerous signature verification schemes such as EIP-712, EIP-1271, MultiSig, Schnorr, etc. The `AmbireAccount` contract also allows users to attach a so-called fallback handler, which gives the opportunity to extend the functionality of the wallet.

The Ambire Wallet contracts are deployed as immutable minimal proxies (EIP-1167) through the `AmbireAccountFactory`. The initialization bytecode is crafted by a script located in the GitHub repository as there is no `initialize` function on the implementation contract. EIP-4337 account abstraction is also supported via the `validateUserOp` function of the `AmbireAccount` contract and the `AmbirePaymaster` contract.

The wallet implements signature recovery using external signature validator contracts, one of which is the `DKIMRecoverySigValidator`. It allows users to recover access to their wallet in case they've lost their private key by executing a recovery using a DKIM signature (from an email provider) and/or a `secondaryKey` signature. In case only one of the two methods is used, a timelock is applied. The `DKIMRecoverySigValidator` contract address and the recovery account information parameters have to be set upfront via the `privileges` mapping in the user's `AmbireAccount` wallet contract.

6.1 Codebase maturity

Code complexity - *Good*
The smart contracts are kept small in size and do not inherit any other contracts. The number of supported signature verification schemes and the amount of arguments used for DKIM signature recovery slightly increase the complexity.

Security - *Excellent*
The codebase has already undergone multiple security audits by independent security researchers as well as a competition.

Decentralization - *Excellent*
The protocol is built with focus on complete immutability and users have full control over their accounts.

Testing suite - *Good*
Comprehensive testing suite consisting of unit and integration tests.

Documentation - *Good*
The documentation explains the general concepts of the protocol and provides information about specific design choices. The NatSpec and inline code comments thoroughly describe each function.
Best practices - *Excellent*
The contracts are tight and well written, there is no redundant complexity or any inheritance making the code more easily auditable.

6.2 Privileged actors

- **Privileged users** - Allowed to execute operation on behalf of the corresponding Ambire wallet.
- **Fallback handler** - Set by the privileged users; allows to execute custom logic via delegatecall.
- **DNSSEC oracle** - Trusted for validation when adding a new DKIM key.
- **secondaryKey** - The account set by a privileged user that has the ability to restore lost access via signature recovery using an additional DKIM signature or a timelock.
- **relayer** - Executes ERC4337 user operation bundles; also allowed to withdraw stuck tokens in the paymaster contract.
- **allowedToDrain** - Actor allowed to withdraw stuck tokens in the factory contract.
- **authorizedToSubmit** - Allowed to add new DKIM keys.
- **authorizedToRevoke** - Allowed to remove compromised DKIM keys.

6.3 Threat model

What in the Ambire V2 protocol has value in the market?

The funds stored in users smart wallets/accounts (both deployed and not yet deployed) and any access-control privileges Ambire Wallet contracts have in other protocols.

What are the worst-case scenarios for an Ambire Wallet account?

- Signature malleability/replay attack.
- Impersonating a signer.
- Spoofing signature passes on-chain.
- A direct call to the AmbireAccount contract implementation sets a privilege of a fallback handler that allows to destroy the implementation.
- Stealing funds from a wallet.
- Blocking signature recovery.
- External party manipulates the outcome of a signature recovery.
- DKIM headers not properly validated.

What are the main entry points and non-permissioned functions?

- `AmbireAccount.fallback()` - checks if there is a set fallback handler and `delegatecall`s it.
- `AmbireAccount.execute|executeMultiple|executeBySender()` - allow transactions signed by privileged users to be executed.
- `AmbireAccountFactory.deploy|deployAndExecute|deployAndExecuteMultiple()` - deploy Ambire wallet contracts and optionally execute calls in the same transaction.
- `AmbireAccount.validateUserOp()` & `AmbirePaymaster.validatePaymasterUserOp()` - validate a bundle of user operations to be executed.
- `DKIMRecoverySigValidator.validateSig()` - validates a signature recovery request using either DKIM or `secondaryKey` signatures or both.

- `DKIMRecoverySigValidator.addDKIMKeyWithDNSSec|removeDKIMKey()` - controls the set of DKIM keys.

6.4 Observations

Several interesting design decisions were observed during the review of the Ambire Wallet V2 smart contracts, some of which are listed below:

- Signature replay protection should be handled by external validators if used as the AmbireAccount nonce is not checked.
- So-called anti-blocking mechanism is enforced in every `execute` function so that privileged users cannot remove their own role and eventually leave the wallet with no signers.
- The AmbireAccount supports tryCatch methods that allow execution of bundled transactions where 1 or more can fail without reverting the whole batch.
- A timelock is enforced in case a recovery is done using either only DKIM signature or only a secondaryKey signature.
- A "bridge" option is implemented for email providers that do not support DNSSEC.
- The `deploy` methods in `AmbireAccountFactory` will not fail if the proxy has already been deployed, but continue execution of the passed transaction calls.
- Signature spoofing is supported for easier off-chain gas estimations.
- None of the contracts inherit other except for simple interfaces.
- The `AmbireAccountFactory` and `AmbirePaymaster` implement a "call" function that allows a privileged address to withdraw any stuck tokens.

6.5 Useful resources

The following resources were used to research topics around the Ambire Wallet V2 protocol such as Account Abstraction, Schnorr and DKIM signatures verification, as well as previous audits and documentation:

- *Documentation for C4 contest*
- *Previous audit reports*
- *EIP-4337*
- *Schnorr signatures*
- *DKIM signatures*

7 Findings

7.1 Low

7.1.1 Signer address is not included in transactions message payload

Severity: *Low*

Context: Transaction.sol#L7-L11, AmbireAccount.sol#L164-L165

Description: To allow users execute operations on behalf of an AmbireAccount, the `execute` method is implemented. It accepts an array of `Transaction` struct elements and a signature that are verified using the signature verification schemes supported by the `SignatureValidator` library - EIP-712, EIP-1271, MultiSig, Schnorr, etc., or an external signature validator contract (i.e. `DKIMRecoverySigValidator`).

After the signature is verified, the signer address is checked whether it has privileges in the contract before and after executing the transaction.

The potential problem here is the way standard and SmartWallet/EIP-1271 signatures are verified. Consider the following scenario:
1. A user has an EOA address `0x123` added as a privileged account in his AmbireAccount wallet.
2. They also have a smart wallet (i.e. another AmbireAccount) added as a privileged account.
3. The user wants to execute a transaction that removes address `0x123` as a privileged role.
4. They ssign a message with the necessary data using their `0x123` address.
5. The `execute` arguments are:
- a `Transaction` that calls into the `AmbireAccount` and executes `setAddrPrivilege(0x123, 0)`
- a signature of the signer `0x123`

The above flow should lead to a revert of the transaction as the privileged account is trying to remove itself which is forbidden by the anti-bricking mechanism enforced in `execute`:

```
// The actual anti-bricking mechanism - do not allow a signerKey to drop their own
   privileges
require(privileges[signerKey] != bytes32(0), 'PRIVILEGE_NOT_DOWNGRADED');
```

However, since another smart wallet that has the same signer is also a privileged account in this AmbireAccount, duo to the way signature verification is done for EIP-1271 wallets (i.e. using the `isValidSignature` method), anyone can simply use the same signature (`v`, `r`, `s` values), but modify its structure by changing the `mode` from Standard to SmartWallet and passing the other wallet address as signer. That way, having one signer signed one message, just modifying the signature structure, we are able to manipulate the execution context of the `execute` function and change the value of the `signerKey` variable.

Note that this doesn't cause any impact on the current version of the `execute` function as it doesn't use the recovered `signerKey` variable for other purpose than simply checking its privilege. However, if we look at the `DKIMRecoverySigValidator.validateSig`, we see that a similar scenario occurs where we pass a `secondaryKey` and a `secondSig`, which if manipulated like that (i.e. executed using a SmartWallet that the `secondaryKey` is part of instead of the `secondaryKey` itself) could theoretically cause the recovery to be replayed as the `identifier` hash would be different but the signature still valid for another signer.

Recommendation: Consider adding the `expectedSigner` address as a property to the `Transaction` struct. Given that this is not an issue impacting the current version of the protocol, a fix is not necessary, but this behavior should be considered in future upgrades.

Resolution: Pending.

7.1.2 EIP-165 specification not enforced

Severity: *Low*

Context: AmbireAccount.sol#L271

Description: The following 2 statements are included in EIP-165 using the `MUST` keyword for implementing a `supportsInterface` function:

- *"This function must return a bool and use at most 30,000 gas."*
- *"@dev You must not set element 0xffffffff to true"*

Since AmbireAccount.supportsInterface forwards the call over to the fallback handler in case the passed selector is none of the previously checked one, it's not guaranteed that these requirements are met.

Recommendation: Consider enforcing a gas limit of 30,000 subtracting the already spent gas. Additionally, implement an explicit check that returns false if `0xffffffff` is passed.

Resolution: Pending.

7.1.3 Unsafe deployment possible through the Ambire factory

Severity: *Low*

Context: AmbireAccountFactory.sol

Description: `AmbireAccount`s are currently deployed as EIP-1167 minimal proxy contracts through the `AmbireAccountFactory` by using a script located in `src/libs/proxyDeploy/deploy.ts`.

The AmbireAccount contract has no constructor or `initialize` method hence the `deploy.ts` prepends the initialization bytecode which consists of SSTORE operations that set the initial privileged accounts. For that reason, it's guaranteed that the deterministic address calculated using the bytecode will always be per the set of initial signers.

However, the `AmbireAccountFactory` makes no validation on the contracts that are deployed except whether they support the AmbireAccount interface. Therefore, any smart wallet contarcts can be deployed using the `deploy` function or `deployAndExecute`/`deployAndExecuteMultiple` if the interfaces is supported.

Therefore, if a user deploys a normal wallet proxy contract through `deploy` (not generated using `src/libs/proxyDeploy/deploy.ts`) that initializes the privileges through an `initialize` function, the user's transaction can be front-runned by anyone and their wallet can be basically stolen since its address computation did not take into account the set of privileged addresses. This is dangerous as the goal of using create2 is to be able to store funds in your predicted wallet address even before deployment. Therefore, users funds may be stolen if such wallet is to be deployed through the factory.

Recommendation: Consider documenting the above behavior so that users do not deploy standard wallet contracts via create2 by assuming it's safe.

Resolution: Pending.

7.1.4 Transactions batching can be griefed by front-running

Severity: *Low*

Context: AmbireAccount.sol#L182-L184

Description: The `AmbireAccount` and `AmbireAccountFactory` contracts support batching `execute` calls/transactions by simply passing an array of the arguments that would be used by the `execute` function to the `executeMultiple` function.

However, anyone can simply front-run the call to `deployAndExecuteMultiple` or `executeMultiple` and invoke the same function or `execute` with just 1 of the transactions in the batch. That way the original user's transaction will revert due to the nonce increment.

This doesn't cause any damage to the user other than simply griefing.

Recommendation: Consider documenting the above behavior.

Resolution: Pending.

7.2 Informational

7.2.1 Typographical mistakes and code-style suggestions

Severity: *Informational*

Context: contracts/*

Description: The contracts contains one or more typographical issue(s). In an effort to keep the report size reasonable, we enumerate these below:

1. Wrong comment on line #9 in `AmbireAccount.sol` copied from `DKIMRecoverySigValidator.sol`:

```
* @notice  A validator that performs DKIM signature recovery
```

2. A typo is made in the NatSpec comment of `AmbireAccount.onERC1155BatchReceived`:

```
    * @return  bytes4  onERC1155Received function selector // @audit should be
      onERC1155BatchReceived
```

3. uint256 can be used instead of uint on line 80 of `AmbireAccount.sol`:

```
        address fallbackHandler = address(uint160(uint(privileges[
            FALLBACK_HANDLER_SLOT])));
```

4. Several gas optimizations can be applied to the `fallback` and `executeCall` functions of `AmbireAccount` like not caching the `returndatasize()` and using `iszero(result)` instead of `eq(result, 0)`:

```
        address fallbackHandler = address(uint160(uint(privileges[
            FALLBACK_HANDLER_SLOT])));
```

Consider reusing OpenZeppelin's Proxy.sol fallback function implementation.

5. The `recoverAddr` function of the `SignatureValidator` library is never used.
6. Consider implementing a check for "malleable S" when using `ecrecover(hash, v, r, s)`.
7. A typo is made in the NatSpec comment of `AmbireAccountFactory.deploy`:

```
    * @notice  Allows anyone to deploy any contracft with a specific code/salt //
      @audit should be 'contracts'
```

8. `<= uint8(type(SignatureMode).max)` can be used instead of `< SignatureMode.LastUnused` to get rid of the warning:

```
        // WARNING: must always be last
        LastUnused
```

Recommendation: Consider fixing the above typographical issues and suggestions.

Resolution: Pending.

8 Final remarks

Hunter Security attests to the quality and professional approach to security that the Ambire team ensures during the development of their smart contracts for the Ambire Wallet V2 protocol. The codebase has already undergone multiple security reviews conducted by some of the most reputable and talented security researchers in the space. The team implements all best practices in their smart contracts and testing suite. Our team is confident in the project and the team behind it.