Ambire Security Review

Pashov Audit Group

Conducted by: pashov

May 27th, 2023

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Ambire** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ambire

Ambire is a smart wallet protocol. Users have wallets (accounts) which are controlled by them or other addresses that have "privileges" to do so. A user can do an off-chain signature of a bundle of transactions and anyone can execute it on-chain. Different signature schemes are allowed, for example EIP712, Schnorr, Multisig and others. The protocol works in a counterfactual manner, meaning a user wallet gets deployed only on its first transaction. The actual deployment is an EIP1167 minimal proxy for the wallet smart contract.

Observations

The factory contract is used for deploying (via `CREATE2`) a custom EIP1167 proxy that prepends SSTORE commands to initialize the privileged accounts of a wallet. This bytecode generation is done here and is controlled by the Ambire team, who manage the back end code of the application. Users will interact with their smart wallets through this proxy. The `AmbireAccount` code has a `delegatecall` opcode in it, which in combination with the `CREATE2` opcode opens up a "metamorphic contract" use case.

A special `allowedToDrain` address can execute arbitrary calls in `AmbireAccountFactory` but this is safe as the contract does not have state and should not hold balances.

A "non-bricking" technique is implemented in most methods, not allowing a user to downgrade his privileges resulting in a "brick" state of the wallet.

There is a relayer in the protocol architecture which can deploy wallets or execute transaction bundles. Relayers are an attack vector for gas griefing attacks and should do transaction simulations off-chain as well as it can use an on-chain gas limited multi call contract.

There are no external dependencies in the protocol (No OpenZeppelin/Solmate imports).

Threat Model

When it comes to `AmbireAccount`:

- Unprivileged user can only call the `fallback` function
- Privileged user can only directly call `execute`, `executeMultiple` and `executeBySender`
- The contract's external calls can call `setAddrPrivilege`, `tryCatch`, `tryCatchLimit` and `executeBySelf`

Privileged Roles & Actors

- Relayer - executes bundles for users, usually including a fee transaction to himself, can also hold a recovery key
- AllowedToDrain - a role given to an address during `AmbireAccountFactory` deployment. It can execute arbitrary calls in `AmbireAccountFactory`
- Privileged user - can execute arbitrary calls in the wallet, complete control
- Fallback handler - can extend the `AmbireAccount` functionality, receives `delegatecall`s

Security Interview

Q: What in the protocol has value in the market?

A: The funds held in a user's smart wallet.

Q: In what case can the protocol/users lose money?

A: If they can't use their funds or they get stolen from their wallet.

Q: What are some ways that an attacker achieves his goals?

A: Manage to trick the wallet that they are authorized to execute arbitrary calls by providing some special signature or launch a replay attack.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

***review commit hash* - [3f17320132e3bc67fa9a505fafded7bb16fc3e96]**

***fixes review commit hash* - [5c54f8005e90ad481df8e34e85718f3d2bfa2ace]**

Scope

The following smart contracts were in scope of the audit:

- `libs/Bytes`
- `libs/SignatureValidator`
- `AmbireAccount`
- `AmbireAccountFactory`

7. Executive Summary

Over the course of the security review, pashov engaged with Ambire to review Ambire. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	Ambire
Date	May 27th, 2023

Findings Count

Severity	**Amount**
Critical	1
Medium	1
Low	4
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Anyone can cancel another user's scheduled recovery	Critical	Resolved
[M-01]	Invalid signature execution is possible if address(0) has non-zero privileges	Medium	Resolved
[L-01]	The ecrecover precompile is vulnerable to signature malleability	Low	Resolved
[L-02]	Schnorr signatures are insufficiently validated	Low	Resolved
[L-03]	Not all call paths are anti-bricking protected	Low	Resolved
[L-04]	Signature expiry is not implemented in the protocol	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Anyone can cancel another user's scheduled recovery

Severity

Impact: High, as important protocol feature can be permanently blocked for a victim user

Likelihood: High, as it has no preconditions and can be exploited by anyone

Description

A wallet recovery mechanism in Ambire allows a pre-set account (usually the relayer) to set a new main signer after the initial one was stolen/lost. It works by calling `execute` with a `SIGMODE_RECOVER` signature. This will store the request in the contract like this, scheduling it for the future:

```
scheduledRecoveries[hash] = block.timestamp + recoveryInfo.timelock;
emit LogRecoveryScheduled
  (hash, recoveryInfoHash, recoveryKey, currentNonce, block.timestamp, txns);
```

The contract also allows an option to cancel a scheduled recovery, by using a `SIGMODE_CANCEL` signature. The problem is that the `hash` that is stored as a scheduled recovery does not contain the `isCancellation` flag (which just checks if the signature mode is `SIGMODE_CANCEL`). Since the flag isn't part of this hash, anyone can call `execute` with the same `signature` parameter as when `SIGMODE_RECOVER` was used, but just changing the last byte so it uses `SIGMODE_CANCEL`. This means that anyone can cancel another user's scheduled recovery without any preconditions, which leads to a griefing attack vector on wallet recoveries. An attacker can go to an extend to write a script that will cancel a scheduled recovery a few minutes before it is about to pass.

Recommendations

Add the `isCancellation` flag to the `hash` so that the initial `SIGMODE_RECOVER` signature can't be changed into a `SIGMODE_CANCEL` one and reused.

Discussion

pashov: Resolved.

8.2. Medium Findings

[M-01] Invalid signature execution is possible if `address(0)` has non-zero privileges

Severity

Impact: High, as anyone will be able to steal all funds from a given wallet

Likelihood: Low, as it requires `address(0)` to have non-zero privileges

Description

The problem is similar to this issue from a previous audit. The code in `SignatureValidator::recoverAddrImpl` will return `address(0)` if it receives a valid `Schnorr` signature but not one that is for the given `hash` (transactions hash). Now the result will be checked like this:

```
signerKey = SignatureValidator.recoverAddrImpl(hash, signature, true);
require(privileges[signerKey] != bytes32(0), 'INSUFFICIENT_PRIVILEGE');
```

Meaning `signerKey` will be `address(0)` and now if the value for `privileges[address(0)]` is non-zero, then anyone will be able to execute any transaction for this wallet, for example stealing all funds.

Also, it is possible to get `recoverAddrImpl` to return `address(0)` if you provide a signature with `SignatureMode == Multisig` and then the signatures inside it to be of type `SignatureMode == Spoof`. Since `allowSpoofing` argument will be `false`, then we will get to the `return address(0);` code in the end of the method.

A third way to get `recoverAddrImpl` to return `address(0)` is if you use `SignatureMode == Multisig` and just provide an empty array of signatures - then it will return the default value of `address signer` which is `address(0)`

Recommendations

Make sure to never have a path where `recoverAddrImpl` returns `address(0)`, instead just revert the transaction. Also remove the comment `// should be impossible to get here` as it is false.

Discussion

pashov: Resolved.

8.3. Low Findings

[L-01] The `ecrecover` precompile is vulnerable to signature malleability

By flipping `s` and `v` it is possible to create a different signature that will amount to the same hash & signer. This is fixed in OpenZeppelin's ECDSA library like this. While this is not a problem since there are nonces in the system signatures, it is still highly recommended that problem is addressed.

Discussion

pashov: Acknowledged.

[L-02] Schnorr signatures are insufficiently validated

As per Chainlink's Schnorr signatures verification implementation it is recommended to check that the Schnorr signature is strongly less than the `Q` constant as well as the public key's X coordinate to be less than `Q/2`. This protects from signature malleability attacks which are not currently a problem in the system because there are nonces in the signatures.

Discussion

pashov: Acknowledged.

[L-03] Not all call paths are anti-bricking protected

Both the `execute` and `executeBySender` call paths have anti-bricking protection that looks like:

```
require(privileges[signerKey] != bytes32(0), 'PRIVILEGE_NOT_DOWNGRADED');
```

and

```
require(privileges[msg.sender] != bytes32(0), 'PRIVILEGE_NOT_DOWNGRADED');
```

The problem is that the `fallback` call path is missing this protection, which means not 100% of cases are covered. A possible partial solution is to check if the caller's privileges were downgraded in the `delegatecall` and if they were to revert. Still, not all cases can be handled properly here, so document this well when explaining the expected fallback handler behavior.

Discussion

pashov: Acknowledged.

[L-04] Signature expiry is not implemented in the protocol

Let's say a user signs a bundle of transactions and expects the relayer to execute it, including a tip for him. Now if the gas price spikes up very quickly it can become not economically viable for the relayer to execute the bundle, so the transaction might stay stale for a while. When the gas price falls back the market could have changed, which can result in for example swaps receiving less value than they should have. To combat this it is useful to add a signature expiry/deadline property, after which a bundle would be invalid to execute.

Discussion

pashov: Acknowledged.