BLOCKSEC

Security Audit
Report for Smart Wallet Recovery

Date: September 30, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	OKX
Target	Smart Wallet Recovery

Version History

Version	Date	Description
1.0	September 30, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-10-10 02:34:54+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of Smart Wallet Recovery of OKX.

The project implements a cross-chain recovery system for unified smart accounts, focusing on the Ethereum Virtual Machine (EVM) stack implementation. The system integrates ZKEmail and ECDSA technologies to deliver an extensible, low-interaction, and multi-chain reusable recovery mechanism for EVM-compatible chains.

Note this audit only focuses on the smart contracts in the following directories/files:

- packages/evm-recovery-signers

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
smart-wallet-recovery	`Version 1`	`6427495cdb393af55c68a5b4dbf73ed2ea1e9122`
	`Version 2`	`26e309c6f74398460d120f7a1bf46c3b6bf47f5d`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does

[1] https://github.com/okx/smart-wallet-recovery

not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **two** recommendations and **three** notes.

- Low Risk: 1
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Potential front-running on recovery operations	Security Issue	Confirmed
2	-	Revise improper annotation	Recommendation	Fixed
3	-	Verify the input length in the function `createRecoverySigner()`	Recommendation	Fixed
4	-	Security assumption on external dependencies	Note	-
5	-	Risks regarding shared `RecoverySigner` instances	Note	-
6	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential front-running on recovery operations

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the contract `RecoverySigner`, the function `recover()` implements batch account recovery functionality. However, the implementation is vulnerable to front-running attacks due to the lack of account binding in the ZK proof or ECDSA signature. Specifically, while the proof constraints `newOwner`, `validator`, and `timestamp` (line 131), the `accounts` array is not included in the proof's public signals. Consequently, attackers can monitor pending transactions in the mempool and front-run them by modifying the `accounts` array to recover only a subset of the intended accounts.

When the legitimate transaction to recover all accounts is executed, the transaction reverts due to the nonce consumed (line 116). The relayer would have to send another transaction to recover the remaining accounts.

```
115         // Consume nonces using batch methods for all cases
116         NonceManager(nonceManager).batchConsume(
117             _recoveryData.accounts,
118             _recoveryData.timestamp
119         );
120
```

```
121         // Encode addOwner call with admin settings (reused for all accounts)
122         bytes memory addOwnerData = abi.encodeWithSelector(
123             ISmartWallet.addOwner.selector,
124             _recoveryData.newOwner,
125             _recoveryData.validator,
126             ADMIN_SETTINGS
127         );
128
129         // Pre-allocate the calls array to avoid repeated memory allocation
130         ISmartWallet.Call[] memory calls = new ISmartWallet.Call[](1);
131         calls[0].value = 0;
132         calls[0].data = addOwnerData;
133
134         // Execute recovery for all accounts
135         for (uint256 i; i < accountsLength; ) {
136             calls[0].to = _recoveryData.accounts[i];
137             ISmartWallet(_recoveryData.accounts[i]).execute(calls);
138
139             unchecked {
140                 ++i;
141             }
```

Listing 2.1: packages/evm-recovery-signers/src/RecoverySigner.sol

```
111     */
112     function computePubSignals(
113         RecoveryTypes.RecoveryData calldata _data,
114         EmailSignature memory _sig
115     ) public view returns (uint256[37] memory publicSignals) {
116         // Pack domain string into fields (first 9 slots)
117         uint256[] memory domainFields = _packBytes2Fields(
118             bytes(_sig.domain),
119             DOMAIN_BYTES
120         );
121         for (uint256 i = 0; i < DOMAIN_FIELDS; i++) {
122             publicSignals[i] = domainFields[i];
123         }
124
125         publicSignals[DOMAIN_FIELDS] = uint256(_sig.dkimKeyHash);
126         publicSignals[DOMAIN_FIELDS + 1] = uint256(_sig.nullifier);
127         publicSignals[DOMAIN_FIELDS + 2] = uint256(_sig.fromEmailHash);
128
129         uint256[] memory subjectFields = _packBytes2Fields(
130             bytes(
131                 computeSubject(_data.newOwner, _data.validator, _data.timestamp)
132             ),
133             SUBJECT_BYTES
134         );
135         for (uint256 i = 0; i < SUBJECT_FIELDS; i++) {
136             publicSignals[DOMAIN_FIELDS + 3 + i] = subjectFields[i];
137         }
138     }
```

Listing 2.2: packages/evm-recovery-signers/src/zkemail/ZKEmailVerifier.sol

```
34    function verify(
35        RecoveryTypes.RecoveryData calldata _recoveryData,
36        RecoveryTypes.RecoverySignature calldata _recoverySignature
37    ) external pure returns (bool) {
38        bytes32 messageHash = keccak256(
39            abi.encode(
40                _recoveryData.validator,
41                _recoveryData.newOwner,
42                _recoveryData.timestamp
43            )
44        );
```

Listing 2.3: packages/evm-recovery-signers/src/ECDSAVerifier.sol

Impact Malicious actors can block the full recovery process, increasing costs for relayers to recover all accounts.

Suggestion Revise the logic accordingly.

Feedback from the project We consider it an acceptable risk. While we considered requiring ECDSA signature + zkEmail verification of account addresses, this approach would limit recovery to around 10 addresses per email due to subject length constraints, forcing users with multiple accounts to perform multiple recoveries and increasing costs. Since front-runners would pay gas fees to execute recovery transactions and failed recovery costs are lower than complete recovery costs, the current approach provides better user experience and cost efficiency despite the security trade-off.

Clarification from BlockSec The impact can be minimized if the relayer can submit another valid transaction to complete the remaining recovery within the maximum time period. Otherwise, users will need to initiate another recovery request.

2.2 Recommendation

2.2.1 Revise improper annotation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The annotation for the function `_setKeyHash()` in the contract `DKIMRegistry` contains a misleading statement indicating that the function does not validate non-zero values for the parameter `keyHash`. However, the implementation clearly includes a validation that ensures the parameter is non-zero. This erroneous annotation creates developer confusion and poses a maintenance risk if future code changes are made based on the inaccurate specification. The NatSpec annotation should be corrected to properly reflect the actual implementation behavior.

```
57    /**
58     * @notice Set a DKIM key hash as valid for a domain (internal version)
59     * @dev Internal function without access control for derived contracts
60     * @dev Emits a {KeyHashRegistered} event
61     * @dev Note: This function does not validate that keyHash is non-zero
```

```
62     * @param domainHash The hash of the domain name
63     * @param keyHash The hash of the DKIM public key to register
64     */
65    function _setKeyHash(bytes32 domainHash, bytes32 keyHash) internal {
66        if (domainHash == 0) revert InvalidDomainHash();
67        if (keyHash == 0) revert InvalidKeyHash();
68        _keyHashes[domainHash][keyHash] = true;
69        emit KeyHashRegistered(domainHash, keyHash);
70    }
```

Listing 2.4: packages/evm-recovery-signers/src/zkemail/DKIMRegistry.sol

Suggestion Correct the NatSpec annotation.

2.2.2 Verify the input length in the function `createRecoverySigner()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `RecoverySignerFactory`, the function `createRecoverySigner()` creates a `RecoverySigner` instance with initialization parameters. However, the function lacks validation on the length consistency between the `_pubKeyHashes` and `_verifiers` parameters.

If their lengths mismatch, the deployed contract `RecoverySigner` becomes unusable. Specifically, in its function `recover()`, the `computedHash` is computed using the `keyHashes` and `verifiers`, which are of the same length. The result must match the initialization parameter to allow successful recovery.

```
45    function createRecoverySigner(
46        bytes32[] calldata _pubKeyHashes,
47        address[] calldata _verifiers
48    ) public returns (address signer) {
49        bytes32 _salt = _getSalt(abi.encode(_pubKeyHashes, _verifiers));
50
51        // Check if already deployed
52        address predictedAddress = LibClone.predictDeterministicAddress(
53            implementation,
54            _salt,
55            address(this)
56        );
57
58        // Return existing clone if already deployed
59        if (predictedAddress.code.length > 0) {
60            return predictedAddress;
61        }
62
63        // Deploy new deterministic clone
64        address instance = LibClone.cloneDeterministic(implementation, _salt);
65
66        // Initialize the clone with the salt (config hash)
67        IRecoverySigner(instance).initialize(_pubKeyHashes, _verifiers);
68        emit RecoverySignerCreated(instance, _pubKeyHashes, _verifiers);
69
```

```
70        return instance;
71    }
```

Listing 2.5: packages/evm-recovery-signers/src/RecoverySignerFactory.sol

Suggestion Validate the length of input arrays.

2.3 Note

2.3.1 Security assumption on external dependencies

Introduced by Version 1

Description This audit assumes the security of all external dependencies, including the underlying cryptographic primitives, the correctness of the `Circom` circuit implementations, and the proper execution of the trusted setup ceremony, which generated the `Groth16Verifier` contract. The scope of this assessment is contingent upon the integrity of these external components and procedures.

2.3.2 Risks regarding shared `RecoverySigner` instances

Introduced by Version 1

Description The contract `RecoverySigner` enables batch recovery of all smart wallets belonging to a single user. If different users share a `RecoverySigner` instance, malicious actors could potentially add themselves as owners to another user's wallets.

The project typically designs two recovery modes: 1/1 recovery, which requires only email verification from the user, and 2/2 recovery, which requires verification from both the user's email and an OKX team EOA. When following these modes, each RecoverySigner is configured with a unique `keyVerifiersHash` specific to that user, which prevents unauthorized cross-user access.

However, the contract also supports custom recovery configurations where these design-level protections may not be implemented. In such cases, the security risks are not eliminated at the code level. Therefore, users must ensure that `RecoverySigner` instances are never shared between different users to maintain wallet security.

2.3.3 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., the contract owner) can conduct sensitive operations, which introduces potential centralization risks. For example, if the owner of the contract `DKIMRegistry` invokes the function `renouncePauser()`, it will permanently disable pausing functionality and prevent the contract from being paused or unpaused. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS