BLOCKSEC

Security Audit Report for OKX Smart Wallet

Date: September 24, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	OKX
Target	OKX Smart Wallet

Version History

Version	Date	Description
1.0	September 24, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-10-10 02:33:54+0000

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 OKX Smart Wallet of OKX.

The project implements an Account Abstraction (AA) wallet implementation on the Ethereum Virtual Machine (EVM) compatible chains, allowing users to deploy AA smart contract wallets. It also allows Externally Owned Accounts (EOAs) to extend transaction execution logic by setting the AA wallet code, simplifying the processing of complex transaction operations.

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

- src

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
okx-smart-wallet	`Version 1`	`f104cdc2b3a1999f1d5c97b26f14dfe599f148ec`
	`version 2`	`c935ece44678bf06a294ccf1edaed4507e41452e`

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/okx-smart-wallet

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 **two** potential security issues. Besides, we have **three** recommendations and **two** notes.

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

ID	Severity	Description	Category	Status
1	Medium	Unprotected initialization on EOA wallets	Security Issue	Fixed
2	Low	Incorrect ERC-7201 standard implementation	Security Issue	Fixed
3	-	Avoid emitting misleading events in the function `createAccount()`	Recommendation	Fixed
4	-	Add a validation of the input `validatorData` length	Recommendation	Fixed
5	-	Revise improper annotation	Recommendation	Fixed
6	-	Ensure safe integration with `EntryPoint` in the function `executeUserOp()`	Note	-
7	-	Version fragmentation in the proxy upgrade mechanism	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Unprotected initialization on EOA wallets

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SmartWallet`, the function `initialize()` lacks validation on the caller, allowing anyone to initialize a wallet. This creates a potential attack vector under EIP-7702 scenarios.

Specifically, when EOA users send a set code transaction, they designate the contract `SmartWallet` as their code with the wallet initialization calldata. The code delegation is processed before the actual transaction execution. This delegation will not be rolled back even if the transaction fails (e.g., due to out-of-gas errors).

This creates a window where the EOA wallet remains uninitialized. Attackers can invoke its function `initialize()` to set arbitrary owners, gaining control over the wallet.

```
73    /// @notice Initializes the wallet core with initial owners
74    /// @dev Storage is now integrated directly into SmartWallet
75    /// @param initialOwners Array of tuples containing keyHash and validator address pairs
76    function initialize(
```

```
77        InitialOwner[] calldata initialOwners
78    ) external initializer {
79        // Set up initial owners
80        // isAdmin = true, expiration = 0 (never expires), hook = address(0)
81        uint256 settings = packSettings(true, 0, address(0));
82        uint256 len = initialOwners.length;
83        for (uint256 i = 0; i < len; i++) {
84            bytes32 keyHash = initialOwners[i].keyHash;
85            address validator = initialOwners[i].validator;
86
87            _addOwner(keyHash, validator, settings);
88        }
89    }
```

Listing 2.1: src/SmartWallet.sol

Impact Attackers can gain unauthorized control over EOA wallets by setting malicious owners during failed initialization scenarios.

Suggestion Implement access control in the function `initialize()`.

2.1.2 Incorrect ERC-7201 standard implementation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contracts `OKXSmartWalletEntry` and `ERC7201`, the storage layout follows a custom implementation of the ERC-7201 standard. However, the design is problematic due to an incorrect storage root calculation that doesn't properly align with the ERC-7201 specification. Specifically, the storage location at 0x02a90b95e07536939d6b1617e9cf25c8d725ec1c5c4c03ccc00770cd202e6e00 doesn't correctly implement the standard's requirements.

```
15    bytes32 public constant CUSTOM_STORAGE_ROOT =
16        0x02a90b95e07536939d6b1617e9cf25c8d725ec1c5c4c03ccc00770cd202e6e00;
```

Listing 2.2: src/ERC7201.sol

```
6/// @notice Uses custom storage layout according to ERC7201
7/// @custom:storage-location erc7201:OKX.SmartWallet.1.0.0
8/// @dev keccak256(abi.encode(uint256(keccak256("OKX.SmartWallet.1.0.0")) - 1)) & ~bytes32(uint256
    (0xff))
9contract OKXSmartWalletEntry is SmartWallet layout at 0
    x02a90b95e07536939d6b1617e9cf25c8d725ec1c5c4c03ccc00770cd202e6e00 {}
```

Listing 2.3: src/OKXSmartWalletEntry.sol

Impact This implementation may lead to storage collisions and unintended data corruption when interacting with other contracts that comply with ERC-7201 specification.

Suggestion Compute the storage location correctly.

2.2 Recommendation

2.2.1 Avoid emitting misleading events in the function `createAccount()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `createAccount()`, an event `AccountCreated` is emitted after every account creation attempt. However, this event is also emitted when the variable `alreadyDeployed` is true, indicating the account already exists and no new initialization occurs. This implementation could potentially lead to confusion because it inaccurately signals a new account creation when none has occurred.

```
21  function createAccount(
22      InitialOwner[] calldata initialOwners,
23      uint256 salt
24  ) public payable returns (address account) {
25      (bool alreadyDeployed, address instance) = LibClone
26          .createDeterministicERC1967(
27              msg.value,
28              IMPLEMENTATION,
29              _getSalt(initialOwners, salt)
30          );
31
32      if (!alreadyDeployed) {
33          ISmartWallet(instance).initialize(initialOwners);
34      }
35
36      emit AccountCreated(instance, IMPLEMENTATION, initialOwners, salt);
37      account = instance;
38  }
```

Listing 2.4: src/SmartWalletFactory.sol

Suggestion It is recommended to modify the function to emit the event `AccountCreated` only when a new account is actually deployed.

2.2.2 Add a validation of the input `validatorData` length

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `SmartWallet`, the function `executeWithRelayer()` directly slices the input `validatorData` without verifying whether its length meets the minimum requirement of 38 bytes, as specified by its intended format. Although Solidity inherently prevents out-of-bounds access by reverting, the generated error message remains generic and uninformative. As a result, users and integrators receive unclear feedback when providing malformed data.

```
215  function validateUserOp(
216      PackedUserOperation calldata userOp,
217      bytes32 userOpHash,
```

```
218         uint256 missingAccountFunds
219     ) external onlyEntryPoint returns (uint256 validationData) {
220         _payPrefund(missingAccountFunds);
221
222         // Extract validation components from signature
223         // Signature format: pubKeyHash (32) + validUntil (6) + signatures
224         bytes32 pubKeyHash = bytes32(userOp.signature[:32]);
225         uint48 validUntil = uint48(bytes6(userOp.signature[32:38]));
```

Listing 2.5: src/SmartWallet.sol

```
122     function executeWithRelayer(
123         BatchedCall calldata batchedCall,
124         bytes calldata validatorData
125     ) external {
126         // Extract validation components from new format
127         // Format: pubkeyHash (32) + validUntil (6) + signatures
128         bytes32 pubKeyHash = bytes32(validatorData[:32]);
129         uint48 validUntil = uint48(bytes6(validatorData[32:38]));
```

Listing 2.6: src/SmartWallet.sol

Suggestion Add a length check to the beginning of the function `executeWithRelayer()`.

2.2.3 Revise improper annotation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The annotation in the function `executeUserOp()` states that "this contract is only compatible with Entrypoint versions v0.7.0 and v0.8.0". However, this annotation is improper, as the implementation contract only accepts invocation from the `Entrypoint v0.7.0`.

```
101     /// Note that this contract is only compatible with Entrypoint versions v0.7.0 and v0.8.0. It
            is not compatible with v0.6.0, as that version does not support the "executeUserOp"
            selector.
102     function executeUserOp(
103         PackedUserOperation calldata userOp,
104         bytes32
105     ) external onlyEntryPoint {
```

Listing 2.7: src/SmartWallet.sol

```
12  modifier onlyEntryPoint() {
13      if (msg.sender != entryPoint()) revert Errors.NotEntryPoint();
14      _;
15  }
16
17  /// @notice Returns the EntryPoint address
18  function entryPoint() public pure returns (address) {
19      return 0x0000000071727De22E5E9d8BAf0edAc6f37da032;
20  }
```

Listing 2.8: src/ERC4337Account.sol

Suggestion Revise the code annotation.

2.3 Note

2.3.1 Ensure safe integration with `EntryPoint` in the function `executeUserOp()`

Introduced by `Version 1`

Description In the contract `SmartWallet`, the function `executeUserOp()` is designed to execute ERC-4337 compatible transactions only through the external contract `EntryPoint`. However, the function does not implement the signature's expiration validation and replay protection. The project must confirm that these essential security checks are performed properly within the `EntryPoint`.

Feedback from the project The `EntryPoint` enforces the expiry verification and replay protection.

2.3.2 Version fragmentation in the proxy upgrade mechanism

Introduced by `Version 1`

Description In the project, each AA wallet is a proxy contract pointing to the same implementation contract. When the implementation contract is upgraded to a newer version, each wallet should execute an upgrade operation. However, this design leads to version inconsistency because it relies on user cooperation for upgrades. Specifically, if some users refuse or delay the upgrade, it results in version fragmentation where different wallets point to different implementation versions. The project should implement a version management strategy to ensure compatibility across upgrades.

Feedback from the project This is our self-custody design. Upgrades must be triggered by the users themselves, and we will ensure proper version control at the project level.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS