BLOCKSEC

Security Audit

Report for smart-wallet-recovery && groth16-solana

Date: October 17, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	OKX
Target	smart-wallet-recovery && groth16-solana

Version History

Version	Date	Description
1.0	October 17, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-10-21 02:26:48+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	Rust
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1][2] of smart-wallet-recovery && groth16-solana of OKX.

This project is a smart account recovery system deployed on Solana. It combines ZK-Email zero-knowledge proofs, ECDSA signature verification, and a DKIM registry to provide scalable, low-interaction account recovery. Smart wallet providers can pre-register their DKIM public keys via the DKIM registry, eliminating the need to trust third-party DNS providers. When a user needs to recover an account, they submit proofs to the recovery-manager contract, which consist of an off-chain generated ZK-Email proof demonstrating ownership of a specific email and an off-chain 2FA-verified ECDSA signature. After verification, the smart account's owner key can be updated, completing the account recovery process.

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

- solana-recovery-manager/programs/*
- groth16-solana/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
solana-recovery-manager	`Version 1`	`6427495cdb393af55c68a5b4dbf73ed2ea1e9122`
	`Version 2`	`0f7d4fa6f0592f675b5896fb8b2d2480bde76a38`
groth16-solana	`Version 1`	`2ff1cfa1432067dd5ec304a1555c6def201c84ac`

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

[2] https://github.com/Lightprotocol/groth16-solana/tree/v0.2.0

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 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 [3] and Common Weakness Enumeration [4]. 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.

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

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

- **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.

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **one** recommendation and **six** notes.

- Low Risk: 2
- Recommendation: 1
- Note: 6

ID	Severity	Description	Category	Status
1	Low	Lack of binding check for `smart_account_program` in function `recover()`	Security Issue	Fixed
2	Low	Potential DoS during submitting proofs	Security Issue	Fixed
3	-	Applying global index in the function `pack_bytes_2fields()`	Recommendation	Fixed
4	-	Execution ordering of concurrent proposals with identical identifiers	Note	-
5	-	Ensure off-chain negation of `proof_a` before submission in `Groth16` verification	Note	-
6	-	Ensure timestamp consistency for multi-verifier proofs	Note	-
7	-	Potential centralization risks	Note	-
8	-	Ensure consistency between user-provided recovery methods and `smart_account` configured recovery methods	Note	-
9	-	Verifier program initialization consistency are not enforced	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Lack of binding check for `smart_account_program` in function `recover()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Within the `recover()` function, once a provided `proof_pda` is validated and its seeds match the input parameters, the function invokes the target `smart_account_program` using the authority of the `recovery_signer` to add a new passkey. However, the code does not enforce a check to ensure that the invoked `smart_account_program` is the one that the `recovery_signer` was originally derived for. As a result, if multiple `smart_account_program` instances exist, the `PDA` for the same `recovery_signer` may be reused across different programs, which breaks the intended program-specific binding of recovery authority.

```
129    #[account(
130        mut,
131        seeds = [RECOVERY_SIGNER.as_bytes(), args.get_recovery_signer_seeds().as_ref()],
132        bump,
133    )]
134    pub recovery_signer: UncheckedAccount<'info>,
```

Listing 2.1: programs/recovery-manager/src/instructions/recover.rs

Impact An attacker could exploit this by replaying a valid `PDA` of `recovery_signer` to authorize recovery operations on multiple `smart_account_program` instances.

Suggestion Ensure that the `PDA` derivation for `recovery_signer` includes a binding to the target `smart_account_program`.

2.1.2 Potential DoS during submitting proofs

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `submit_proof()` function is responsible for creating a `proof_pda` that stores proof submission data, derived from inputs such as `recovery signer`, `passkey`, `timestamp`, and `verifier`. Since the account creation uses the `init` modifier, the `PDA` can be deterministically derived in advance. If an attacker predicts these parameters, they can front-run and initialize the `proof_pda` before the legitimate user transaction. When the user later attempts to invoke function `submit_proof()`, the transaction will fail because the account already exists, effectively resulting in denial of service.

```
52pub struct SubmitProof<'info> {
53    #[account(mut)]
54    pub tx_payer: Signer<'info>,
55
56    #[account(
57        init,
58        seeds = [
59            PROOF.as_bytes(), args.recovery_signer.as_ref(),
60            PASSKEY.as_bytes(), args.passkey.get_passkey_hash().as_ref(),
61            TIMESTAMP.as_bytes(), args.timestamp.to_le_bytes().as_ref(),
62            VERIFIER.as_bytes(), verifier.key().as_ref(),
63            PUBKEY_HASH.as_bytes(), args.pubkey_hash.as_ref(),
64        ],
65        bump,
66        space = 8 + 32,
67        payer = tx_payer,
68    )]
69    pub proof_pda: Account<'info, Proof>,
70
71    /// CHECK: This is the verifier program
72    #[account(executable)]
73    pub verifier: UncheckedAccount<'info>,
```

```
74
75    pub system_program: Program<'info, System>,
76}
```

Listing 2.2: programs/recovery-manager/src/instructions/submit_proof.rs

Impact Legitimate users can be permanently blocked from submitting proofs, leading to disruption of the recovery process.

Suggestion Use `init_if_needed` instead of `init` to mitigate denial of service from pre-created proof `PDA`s, combined with strict validation of account discriminator and state to ensure safe reuse.

2.2 Recommendation

2.2.1 Applying global index in the function `pack_bytes_2fields()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `gen_pub_signals()`, `pack_bytes_2fields()` function is invoked to pack the email domain and subject data into public input fields for the zero-knowledge proof.

However, when handling byte packing, the function uses an incorrect boundary check condition in the inner loop: `if (j >= padded_size)`. Here, `j` is the relative index within a field (0–30), while `padded_size` represents the total number of bytes globally (e.g., 775). The comparison is therefore meaningless, as the boundary check will never be triggered.

```
9pub const SUBJECT_BYTES: usize = 775;
```

Listing 2.3: programs/zk-email-verifier/src/utils/zk_utils.rs

```
85pub fn pack_bytes_2fields(bytes: &[u8], padded_size: usize) -> Vec<U256> {
86    // Calculate number of fields and initialize fields array
87    let num_fields = (padded_size + 30) / 31;
88    let mut fields = vec![U256::ZERO; num_fields];
89
90    for i in 0..num_fields {
91        let base_idx = i * 31;
92        for j in 0..31 {
93            let idx: usize = base_idx + j;
94            if j >= padded_size {
95                break;
96            }
97            // Fetch byte value or use 0 for padding
98            let byte_val = if idx < bytes.len() {
99                U256::from(bytes[idx])
100           } else {
101               U256::ZERO
102           };
103           // Add byte value to the field
104           fields[i] += byte_val << (8 * j);
```

```
105        }
106    }
107    fields
108}
```

Listing 2.4: programs/zk-email-verifier/src/utils/zk_utils.rs

Suggestion Revise the logic accordingly.

2.3 Note

2.3.1 Execution ordering of concurrent proposals with identical identifiers

Introduced by Version 1

Description It is possible that two proposals can have the same `domain_hash` and `key_hash` but are in different `proposal_type`. Once the timelock has passed, the execution order of these proposals may affect the outcome, leading to divergent system states depending on which proposal is executed first. In the current design, this ordering behavior is intentional, and the system is expected to handle proposals with identical identifiers but different types according to the sequence in which they are executed.

2.3.2 Ensure off-chain negation of `proof_a` before submission in `Groth16` verification

Introduced by Version 1

Description In the `submit_proof()` function, the verifier program is invoked via function `invoke_validate_proof()`, which initializes a `Groth16Verifier` instance from the `groth16-solana` library. During this initialization, the input `proof_a` is passed directly without elliptic curve point negation. According to the `Groth16` protocol specification, proof verification requires `proof_a` to be negated for correct pairing checks. In the current design, this step is delegated to the off-chain proof generator. It is important to ensure that `proof_a` is correctly negated before submission, otherwise valid proofs may be incorrectly rejected on-chain.

2.3.3 Ensure timestamp consistency for multi-verifier proofs

Introduced by Version 1

Description In the current design, if a smart account requires multiple verifiers (e.g., Email and 2FA), the `submit_proof()` function must be called separately for each verifier, generating distinct `proof_pda` accounts. Since the timestamp is part of the `PDA` seeds and must match the timestamp passed into the function `recover()`, both proofs are expected to share the same timestamp. This requirement is enforced by the off-chain backend system.

2.3.4 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., `admin`) can conduct sensitive operations, which introduces potential centralization risks. For example, the admin can grant or revoke the proposal role. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.5 Ensure consistency between user-provided recovery methods and `smart_account` configured recovery methods

Introduced by `Version 1`

Description The smart wallet application supports two recovery schemes and they are `ZKEmail 1/1 Recovery` and `ZKEmail+EOA 2/2 Recovery`. Under the `ZKEmail+EOA 2/2 Recovery` scheme, users are expected to provide valid `proof_pda` accounts from both verifiers, and the program is intended to validate both proofs before permitting recovery.

However, in the current implementation of the `recover()` function, this requirement can be circumvented. A user may supply only a single valid `proof_pda` account from any verifier, after which the recovery operation is still invoked via CPI. This effectively reduces the `2/2 recovery scheme` to a weaker `1/1 scheme`.

To prevent this, the `smart_account_program` should enforce strict consistency between the recovery method chosen by the user and the recovery method configured for that account. In particular, it must ensure that the correct number and type of proofs are always present and validated before recovery is executed.

2.3.6 Verifier program initialization consistency are not enforced

Introduced by `Version 1`

Description The `submit_proof()` function accepts any executable account as the verifier when creating `proof_pda`. The smart account's initialization logic (outside the scope of this audit) is responsible for embedding the expected verifier ID into the `PDA` seeds of the `recovery_signer`. Consequently, a proof submitted with a verifier that does not match the verifier encoded in the `recovery_signer` bindings will fail to be accepted in function `recover()`, as the verifier consistency is enforced by the smart account configuration.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS