Bless & Bless Time Programs

Bless

Bless & Bless Time Programs - Bless

Prepared by: HALBORN

Last Updated 09/04/2025

Date of Engagement: August 25th, 2025 - August 29th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	0	0	1

TABLE OF CONTENTS

1. Introduction

`Bless` engaged Halborn to conduct a security assessment on their `Bless Token` and `Bless Time` programs beginning on August 25, 2025 and ending on August 29, 2025. The security assessment was scoped to the smart contracts provided in the GitHub repository `bless-time-contract` and `bless-contract`, commit hashes, and further details can be found in the Scope section of this report.

The `Bless team` is releasing a new version of their Solana programs, `bless-contract` and `bless-time`. These programs collectively allow for the management and distribution of tokens within the Bless ecosystem. Both programs are designed to facilitate token airdrops to whitelisted users. They use a Merkle tree to efficiently verify a user's eligibility for a claim, which includes the amount and a potential lock-up period.

2. Assessment Summary

Halborn was provided 5 days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Program.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified an improvement to reduce the likelihood and impact of risks, which was acknowledged by the `Bless team`:

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Differences analysis using GitLens to have a proper view of the differences between the mentioned commits
- Graphing out functionality and programs logic/connectivity/functions along with state changes

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability.

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORIES

(a) Repository: bless-time-contract

(b) Assessed Commit ID: 993e5d6

(c) Items in scope:

- bless-time/src/constants.rs
- bless-time/src/context/bless_time_token.rs
- bless-time/src/context/bless_time.rs
- bless-time/src/context/ed25519_verify.rs
- bless-time/src/context/merkle_tree_verify.rs
- bless-time/src/context/mod.rs
- bless-time/src/errors.rs
- bless-time/src/lib.rs
- bless-time/src/states/mod.rs

Out-of-Scope: Third party dependencies and economic attacks.

(a) Repository: bless-contract

(b) Assessed Commit ID: 66e48eb

(c) Items in scope:

- bless-token/src/constants.rs
- bless-token/src/context/mod.rs
- bless-token/src/errors.rs
- bless-token/src/lib.rs
- bless-token/src/states/bitvec.rs

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
AIRDROP STATE CAN BE INITIALIZED WITH AN INCORRECT TOKEN MINT	INFORMATIONAL	ACKNOWLEDGED - 09/03/2025

7. FINDINGS & TECH DETAILS

7.1 AIRDROP STATE CAN BE INITIALIZED WITH AN INCORRECT TOKEN MINT

// INFORMATIONAL

Description

The `bless-time` program is a Solana-based smart contract designed to facilitate token airdrops to a list of whitelisted users. It uses a Merkle tree to efficiently verify a user's eligibility for a claim, which includes the amount and a potential lock-up period.

The protocol's functionality is split into two primary states. The first, `BlessTimeTokenState`, is set up via the `initialize_time_token` instruction to establish an official protocol token. The second, `BlessTimeState`, is created by the `initialize_bless_time_state` instruction to configure an airdrop for a specific token, including creating a vault to hold the distributable tokens.

The `initialize_bless_time_state` instruction does not programmatically validate that the `bless_mint` it receives is the same one configured in the `BlessTimeTokenState`, as demonstrated in the code snippet below.

programs/bless-time/src/context/bless_time.rs

```
#[derive(Accounts)]
pub struct InitBlessTimeState<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(mut)]
    pub bless_mint: Account<'info, Mint>,

    #[account(
        init,
        payer = payer,
        seeds = [SEED_BLESS_TIME_STATE.as_bytes(), bless_mint.key().as_ref()],
        space = 8 + BlessTimeState::INIT_SPACE,
```

If the bless-time team accidentally initializes the airdrop state with an incorrect mint, all subsequent administrative actions for that airdrop (such as funding the vault and setting the Merkle root) would be permanently tied to this incorrect state. Any funds transferred to the vault would be locked in a context that does not correspond to the official token, leading to a failed airdrop campaign.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

It is recommended to enforce a link between the airdrop state and the official token state. This can be achieved by modifying the InitBlessTimeState context to require the BlessTimeTokenState account as part of the instruction.

By adding this account and deriving its address from the same bless_mint, Anchor's framework will automatically ensure that the BlessTimeTokenState has been previously initialized for that mint. This prevents the creation of airdrop states for unexpected tokens.

programs/bless-time/src/context/bless_time.rs

```rust
#[derive(Accounts)]
pub struct InitBlessTimeState<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(mut)]
    pub bless_mint: Account<'info, Mint>,

    #[account(
        init,
        payer = payer,
        seeds = [SEED_BLESS_TIME_STATE.as_bytes(), bless_mint.key().as_ref()],
        space = 8 + BlessTimeState::INIT_SPACE,
        bump,
    )]
    pub bls_time_state: Account<'info, BlessTimeState>,

    // By adding this account, we ensure that `initialize_time_token` must have been
    // called for this mint, effectively linking the two states.
    #[account(
        seeds = [SEED_BLESS_TIME_TOKEN_STATE.as_bytes(), bless_mint.key().as_ref()],
```

```
    pub token_program: Program<'info, Token>,
}
```

Remediation Comment

`ACKNOWLEDGED:` The `Bless team` acknowledged this finding, since the team stated that they preferred control over the token provided to the mentioned function instead of having hard control over it.

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the detection of well-known security issues and vulnerabilities. Among the tools used was `cargo-audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in **https://crates.io** are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the reviewers are including the output with the dependencies tree, and this is included in the `cargo audit` output to better know the dependencies affected by unmaintained and vulnerable crates.

Results

ID	PACKAGE	SHORT DESCRIPTION
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in `curve25519-dalek`'s `Scalar29::sub`/`Scalar52::sub`
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on èd25519-dalek`

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the