// Security Assessment

12.16.2025 - 12.18.2025

Psyche

Nous Research

HALBORN

Psyche - Nous Research

Prepared by: HALBORN

Last Updated 01/22/2026

Date of Engagement: December 16th, 2025 - December 18th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	0	3

TABLE OF CONTENTS

1. Introduction

`Nous Research` engaged Halborn to conduct a security assessment on their **Solana Distributor** program beginning on December 16th, 2025, and ending on December 18th, 2025. The security assessment was scoped to the Solana Programs provided in Psyche GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The **Solana Distributor** program is a Merkle-based airdrop system that allows an authority to configure and manage token distributions with built-in vesting schedules, enabling eligible users to gradually claim their allocated tokens over time using Merkle proofs.

2. Assessment Summary

`Halborn` was provided 3 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 Programs.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Nous Research team`. The main ones were the following:

- `Use init_if_needed to prevent external ATA creation attack.`
- `Validate that the merkle_root is non-zero during both airdrop creation and updates.`
- `Use checked arithmetic where necessary and return errors gracefully on failure.`

3. Test Approach And Methodology

`Halborn` performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation, automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`anchor test` and `cargo test`)

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. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

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
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: psyche

(b) Assessed Commit ID: ee20960

(c) Items in scope:

- architectures/decentralized/solana-distributor/programs/solana-distributor/Cargo.toml
- architectures/decentralized/solana-distributor/programs/solana-distributor/Xargo.toml
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/lib.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/claim_redeem.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/claim_create.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/mod.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_create.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_update.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_withdraw.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/vesting.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/merkle_hash.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/mod.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/airdrop.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/allocation.rs
- architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/claim.rs

Out-of-Scope: External dependencies and economic attack vectors.

REMEDIATION COMMIT ID:

- 29bf2ba

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
AIRDROP CREATION CAN BE DENIED BY FRONT RUNNING THE AIRDROP PDA'S ATA INITIALIZATION	INFORMATIONAL	ACKNOWLEDGED - 01/15/2026
MISSING VALIDATION ALLOWS AIRDROP MERKLE ROOT TO BE SET TO AN INVALID VALUE	INFORMATIONAL	SOLVED - 01/15/2026
UNCHECKED ARITHMETIC IS NOT USED	INFORMATIONAL	SOLVED - 01/15/2026

7. FINDINGS & TECH DETAILS

7.1 AIRDROP CREATION CAN BE DENIED BY FRONT RUNNING THE AIRDROP PDA'S ATA INITIALIZATION

// INFORMATIONAL

Description

The `airdrop_create` instruction initializes the `Airdrop` PDA and attempts to `init` its collateral vault as an associated token account (ATA) for `(airdrop, collateral_mint)`.

However, ATA addresses are deterministic and can be created by any external party on behalf of the `airdrop` PDA. As a result, a malicious attacker can front-run by creating the ATA beforehand, causing the `init` constraint for `airdrop_collateral` to fail and preventing the airdrop from being created, resulting in a denial-of-service (griefing) against airdrop initialization.

architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_create.rs

Copy Code

```
11 #[derive(Accounts)]
12 #[instruction(params: AirdropCreateParams)]
13 pub struct AirdropCreateAccounts<'info> {
14     #[account(mut)]
15     pub payer: Signer<'info>,
16
17     #[account()]
18     pub authority: Signer<'info>,
19
20     #[account(
21         init,
22         payer = payer,
23         space = Airdrop::space_with_discriminator(),
24         seeds = [Airdrop::SEEDS_PREFIX, &params.id.to_le_bytes()],
25         bump,
26     )]
27     pub airdrop: Box<Account<'info, Airdrop>>,
28
29     #[account(
30         init,
31         payer = payer,
32         associated_token::mint = collateral_mint,
33         associated_token::authority = airdrop,
34     )]
35     pub airdrop_collateral: Box<Account<'info, TokenAccount>>,
36
37     #[account()]
38     pub collateral_mint: Box<Account<'info, Mint>>,
39
40     #[account()]
41     pub associated_token_program: Program<'info, AssociatedToken>,
42
43     #[account()]
44     pub token_program: Program<'info, Token>,
45
46     #[account()]
47     pub system_program: Program<'info, System>,
48 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

To address this finding, it is recommended to use `init_if_needed` so that pre-creating the ATA cannot block airdrop creation.

Remediation Comment

ACKNOWLEDGED: The **Nous team** acknowledged the finding.

7.2 MISSING VALIDATION ALLOWS AIRDROP MERKLE ROOT TO BE SET TO AN INVALID VALUE

// INFORMATIONAL

Description

The `airdrop_create` and `airdrop_update` instructions are responsible for initializing and modifying the configuration of an Airdrop PDA, respectively.

However, neither instruction validates that the provided `merkle_root` is non-empty or non-zero, allowing an airdrop to be created or updated with an invalid Merkle root, which can prevent users from successfully claiming their allocations.

architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_create.rs

Copy Code

```
#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct AirdropCreateParams {
    pub id: u64,
    pub merkle_root: MerkleHash,
    pub metadata: AirdropMetadata,
}

pub fn airdrop_create_processor(
    context: Context<AirdropCreateAccounts>,
    params: AirdropCreateParams,
) -> Result<()> {
    let airdrop = &mut context.accounts.airdrop;

    airdrop.bump = context.bumps.airdrop;

    airdrop.id = params.id;
    airdrop.authority = context.accounts.authority.key();

    airdrop.collateral_mint = context.accounts.collateral_mint.key();
    airdrop.total_claimed_collateral_amount = 0;

    airdrop.claim_freeze = false;
    airdrop.merkle_root = params.merkle_root;
    airdrop.metadata = params.metadata;

    Ok(())
}
```

architectures/decentralized/solana-distributor/programs/solana-distributor/src/logic/airdrop_update.rs

Copy Code

```
#[derive(AnchorSerialize, AnchorDeserialize, Clone)]
pub struct AirdropUpdateParams {
    pub claim_freeze: Option<bool>,
    pub merkle_root: Option<MerkleHash>,
    pub metadata: Option<AirdropMetadata>,
}

pub fn airdrop_update_processor(
    context: Context<AirdropUpdateAccounts>,
    params: AirdropUpdateParams,
) -> Result<()> {
    let airdrop = &mut context.accounts.airdrop;

    if let Some(claim_freeze) = params.claim_freeze {
        msg!("claim_freeze: {}", claim_freeze);
```

```
            airdrop.claim_freeze = claim_freeze;
        }

        if let Some(merkle_root) = params.merkle_root {
            msg!("merkle_root: {:?}", merkle_root);
            airdrop.merkle_root = merkle_root;
        }

        if let Some(metadata) = params.metadata {
            msg!("metadata: {:?}", metadata);
            airdrop.metadata = metadata;
        }

        Ok(())
    }
```

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:C/I:N/D:N/Y:C (0.6)

Recommendation

To address this finding, it is recommended to validate that the `merkle_root` is non-zero during both airdrop creation and updates.

Remediation Comment

SOLVED: The **Nous team** resolved the finding by validating that the `merkle_root` is non-zero.

Remediation Hash

29bf2baa2367729a56f5ee2d3367bd9501df2591

7.3 UNCHECKED ARITHMETIC IS NOT USED

// INFORMATIONAL

Description

The `claim_create` and `claim_redeem` instructions perform arithmetic operations (e.g., computing `claimable_collateral_amount`, incrementing `claimed_collateral_amount`, and updating `total_claimed_collateral_amount`) using raw operators rather than checked arithmetics.

While this does not introduce a direct security risk under normal assumptions, some edge cases can cause panics, resulting in unexpected transaction failures rather than graceful, program defined errors.

architectures/decentralized/solana-distributor/programs/solana-distributor/src/state/vesting.rs

Copy Code

```
impl Vesting {
    pub fn compute_vested_collateral_amount(
        &self,
        now_unix_timestamp: i64,
    ) -> u64 {
        if now_unix_timestamp < self.start_unix_timestamp {
            return 0;
        }
        if self.duration_seconds == 0 {
            return self.end_collateral_amount;
        }

        let elapsed_seconds =
            u128::try_from(now_unix_timestamp - self.start_unix_timestamp)
                .unwrap();
        let duration_seconds = u128::from(self.duration_seconds);
        let end_collateral_amount = u128::from(self.end_collateral_amount);

        let vested_collateral_amount =
            end_collateral_amount * elapsed_seconds / duration_seconds;
        if vested_collateral_amount > end_collateral_amount {
            return self.end_collateral_amount;
        }

        u64::try_from(vested_collateral_amount).unwrap()
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To address this finding, it is recommended to use checked arithmetic (`checked_add` , `checked_sub` , etc.) and return explicit errors on failure.

Remediation Comment

SOLVED: The **Nous team** resolved the finding by implementing checked arithmetic during the arithmetic operation.

Remediation Hash

29bf2baa2367729a56f5ee2d3367bd9501df2591

8. AUTOMATED TESTING

STATIC ANALYSIS REPORT

Description

Halborn used automated security scanners to assist with 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 auditors 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.

Cargo Audit Results

ID	CRATE	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 `ed25519-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 project's integrity and addressing potential vulnerabilities introduced by code modifications.