Adevar Labs

Prepared for:
DoubleZero

November 13, 2025

Audit Report

Version: Final Audit Report

Table of Contents

1. Executive Summary

About DoubleZero

The DoubleZero network is a global, high-performance routing layer that enables validators to connect with minimal latency by moving data directly from point A to point B. This audit focused on swap program, which swaps 2Z tokens into SOL, to distribute the rewards for respective epoch.

Audit Summary

- Number of Findings: 2 total
- Critical: 0
- High: 0
- Medium: 1
- Low: 1
- Number of Enhancement Opportunities: 4

Risk Profile

The following table summarizes the distribution of identified vulnerabilities by risk level:

Risk Level	Count	Fixed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	1	0	1
Low	1	0	1

Overall Security Posture

The DoubleZero swap program shows a solid and security-conscious design. Core components of swap programs were thoroughly reviewed and found robust.

One medium-severity issue remains open: the fixed **sol_quantity** in the swap logic, which can cause sweep failures when balances do not align with the hardcoded amount.

A redundant operation in the discount rate formula was resolved, improving efficiency without affecting correctness. A potential slippage concern in **buy_sol** was acknowledged but does not pose an immediate vulnerability.

Fix Review Summary

The redundant logic in the discount rate calculation has been successfully removed. This update eliminates the unnecessary **.max(min_discount_rate_decimal)** operation and cleans up a Rust-level unit test that could never realistically occur due to existing validation checks. This resolves the previously

reported issue on redundant mathematical operations.

The potential slippage protection enhancement is acknowledged. Similarly, the medium-severity issue regarding a fixed swap amount in the **buy_sol** logic is still open and requires a fix to prevent possible sweep failures.

Launch Recommendations

To ensure a secure and reliable launch of the DoubleZero programs, we strongly recommend addressing all identified medium-severity issues before deployment. In particular, care should be taken to verify that fix of the medium issue does not introduce rounding edges or cause the fills registry to accumulate entries excessively.

Audit Scope

- **Repository:** double-zero-sol-2z-conversion (private)
- **Commit Hash:** d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b
- **Files/Modules in Scope:**
- on-chain/programs/converter-program/src/*.rs

2. Assumptions and Considerations

Scope Limitations

The following limitations and constraints should be considered when reviewing this security assessment report:

1. **Time-Limited Assessment**: The security assessment was conducted over a 5-day period. As such, not all potential security vulnerabilities may have been identified. Security assessment is a point-in-time exercise, and new vulnerabilities may emerge following the completion of this assessment.

1. **Defined Scope**: The assessment was limited to the systems, applications, and networks explicitly defined in the scope. Systems outside the defined scope, even if they interact with in-scope systems, were not assessed. A list of in-scope assets is provided in the Introduction section of this report.

3. Severity Definitions

Each issue identified in this report is assigned a severity level based on two dimensions: **Impact** and **Likelihood**. These dimensions help project our team's understanding of both the potential consequences of a vulnerability and how likely a vulnerability is to be discovered and exploited in the real world.

Impact

Impact reflects the potential consequences of the issue–particularly on **project funds**, **user funds**, and the **availability or integrity** of the protocol.

- **High Impact**: Successful exploitation could result in a complete loss of user or protocol funds, disruption of core protocol functionality, or permanent loss of control over critical components.
- **Medium Impact**: Exploitation could cause significant disruption or partial loss of funds, but not a total compromise. May impact some users or non-core functionality.
- **Low Impact**: The issue has minor or negligible consequences. It may affect edge cases, expose metadata, or degrade performance slightly without putting funds or core logic at serious risk.

Likelihood

Likelihood reflects how easy a vulnerability is to discover and exploit by an attacker, as well as how economically attractive the exploit is to an attacker.

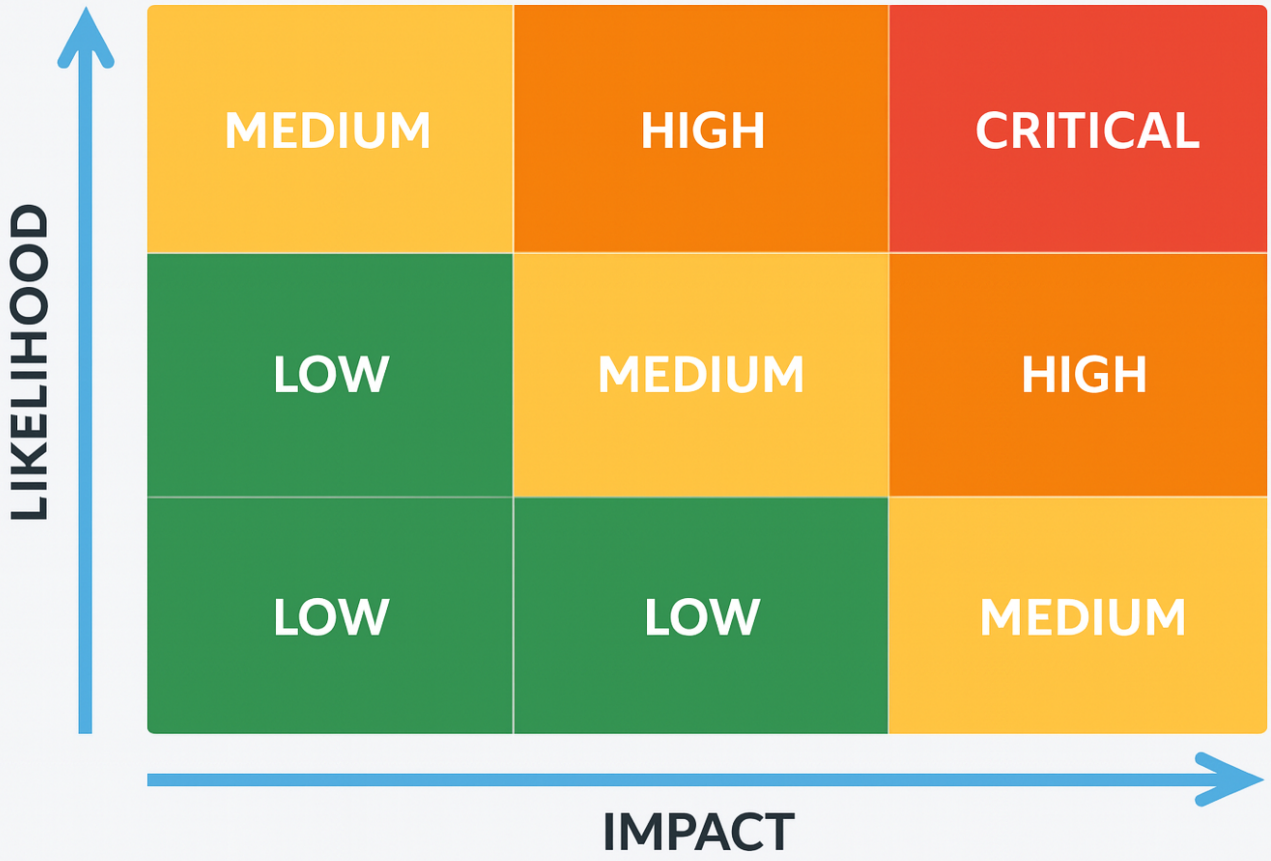
- **High Likelihood**: The vulnerability is trivially exploitable. This means it can be exploited by a wide range of actors without privileged access rights, with minimal capital requirements and low financial risks.
- **Medium Likelihood**: This type of vulnerability can be found and exploited with moderate effort. It might require a significant capital investment, but with manageable financial risk.
- **Low Likelihood**: Exploitation of these vulnerabilities is often technically unfeasible or requires highly specialized conditions. They may require extraordinary effort or a significant financial risk for an attacker, with a high chance of failure and minimal potential return.

Severity Classification Matrix

By combining **Impact** and **Likelihood**, we assign a severity level using the matrix below:

- **Critical**: High impact + high likelihood (e.g. a bug that could allow anyone to drain a substansial amount of protocol funds with minimal effort)
- **High**: High impact with medium likelihood, or medium impact with high likelihood
- **Medium**: Moderate impact and/or discoverability
- **Low**: Minimal impact or unlikely to be exploited

This structured approach helps teams prioritize fixes and mitigate the most dangerous threats first.



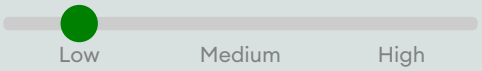
Severity Matrix

4. Findings

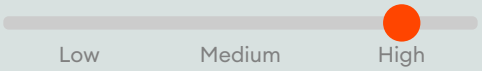
M01: Fixed Swap Amount Causes Sweep Process Failure

Status: Open

Impact:



Likelihood:



Severity: Medium

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/sol-2z-conversion-v1-d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b/on-chain/programs/converter-program/src/buy_sol.rs#L117

>_ buy_sol.rs RUST

```
115:         )?;
116:
117:         let sol_quantity = self.configuration_registry.sol_quantity;
118:
119:         // Get current ask price including discounts.
```

Description:

The **sol_quantity** used in the **buy_sol** process is hardcoded to a fixed amount. This design can cause the sweep process to get stuck when the remaining balance is smaller than the hardcoded swap amount.

Even if the total balance is sufficient to cover the total debt, the sweep cannot complete in such cases. While the admin can manually adjust **sol_quantity**, this is inefficient and error-prone, especially since balances are often not evenly divisible

Recommendation:

If **sol_quantity** is larger than **journal.total_sol_balance**, but part of it can still be swapped to cover **total_sol_debt**, we could allow the operation to proceed by only swapping the actual delta.

One way to implement this would be:
actual_sol_quantity = **min(journal.total_sol_balance, sol_quantity)**

Important: we need to account for possible rounding issues and the risk of small swaps filling up the fills registry, which could be abused as a DoS vector.

Proof of Concept:

Example scenario:

- **total_sol_balance = 95**
- **sol_quantity = 25**
- **total_sol_debt = 95**

Execution flow:

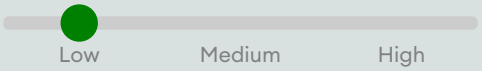
1. First swap → balance: 95 - 25 = 70, swapped: 25
2. Second swap → balance: 70 - 25 = 45, swapped: 50
3. Third swap → balance: 45 - 25 = 20, swapped: 75
4. Fourth swap → remaining balance: 20 SOL, but **sol_quantity** = 25 → cannot proceed

Although **total_sol_balance** equals **total_sol_debt**, the sweep fails due to the rigid swap amount.

L01: Missing Slippage Protection Enables Unexpected Token Expenditure

Status: Open

Impact:



Likelihood:

Severity: Low

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/sol-2z-conversion-v1-d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b/on-chain/programs/converter-program/src/buy_sol.rs#L87

>_ buy_sol.rs RUST

```rust
85: }
86:
87: impl<'info> BuySol<'info> {
88:     pub fn process(
89:         &mut self,
```

Description:

Currently, the **buy_sol** function lacks a mechanism for users to specify the maximum number of 2Z tokens they're willing to convert, creating a potential slippage vulnerability. While the function does check that the bid price (per SOL) meets or exceeds the ask price, it doesn't let users limit their total token expenditure.

The problem arises because:
- The **sol_quantity** parameter is controlled by the admin and can be changed at any time
- Users have no way to specify their maximum acceptable total token cost.
- If **sol_quantity** is increased after a user has submitted a transaction but before it's processed, the user may spend significantly more tokens than anticipated

This creates risk for users since they could unexpectedly spend more tokens than they intended, particularly if there's an admin configuration change during transaction confirmation.

Recommendation:

Option 1: Modify the **buy_sol** function to accept a new parameter **max_tokens_in** that specifies the maximum number of 2Z tokens a user is willing to spend and check this against **tokens_required**:

RUST

```rust
require!(
        tokens_required <= max_tokens_in,
        DoubleZeroError::SlippageExceeded  // New error code needed
    );
```

Option 2: Implement a **buy_sol_checked** instruction (similar to **transfer-checked** for SPL tokens), where instead of checking the inbound amount of 2Z tokens, the swap program checks the outbound SOL amount. This would allow users to pass in a specific SOL amount (e.g., 1 SOL) and the instruction would revert if the fixed **sol_quantity** does not equal the expected amount, providing protection against unexpected changes to the admin-configured swap quantity.

5. Enhancement Opportunities

E01: Redundant Mathematical Operation Reduces Performance

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/sol-2z-conversion-v1-d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b/on-chain/programs/converter-program/src/calculate_ask_price.rs#L95-L99

>_ calculate_ask_price.rs RUST

```rust
93:     let s_diff_decimal = Decimal::from_u64(s_diff)?;
94:
95:     let discount_rate_decimal = coefficient_decimal
96:         .checked_mul(s_diff_decimal)?
97:         .checked_add(min_discount_rate_decimal)?
98:         .min(max_discount_rate_decimal)
99:         .max(min_discount_rate_decimal);
100:
101:    // conversion_rate = oracle_swap_rate * (1 - discount_rate)
```

Description:

The discount rate formula contains a redundant **max()** operation with **D_min** that serves no purpose. The inner **min()** operation already ensures the result is between **D_min** and **D_max**, making the outer **max()** comparison unnecessary.

Current Formula:

```rust
discount_rate = max(min(γ * (S_now - S_last) + D_min, D_max), D_min)
```

Analysis:
The formula works as follows:
1. **y * (S_now - S_last) + D_min** - Calculate base discount plus slot-based scaling
2. **min(y * (S_now - S_last) + D_min, D_max)** - Cap the result at D_max
3. **max(result, D_min)** - Ensure result is at least D_min

However, step 3 is redundant because:
- When 0 slots pass: **y * 0 + D_min = D_min**, so **min(D_min, D_max) = D_min**
- When many slots pass: the **min()** caps at **D_max**, and since **D_max >= D_min**, **max(D_max, D_min) = D_max**
- The inner calculation always produces a value >= D_min, making the outer **max()** pointless

Potential Benefit:

Reduced computational overhead

Recommendation:

Simplify the formula to:

```rust
discount_rate = min(γ * (S_now - S_last) + D_min, D_max)
```

E02: Token-2022 Program Incompatibility Causes Transaction Failures

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/doublezero-solana-7019a4a403be9fad13cd63af5d1fe5e8925870b8/programs/revenue-distribution/src/processor.rs#L2940

>_ processor.rs RUST

```
2938:     //
2939:     // First, check that the program is the SPL Token program.
2940:     if sibling_ix.program_id != spl_token::ID {
2941:         msg!("Sibling instruction's program ID is not SPL Token");
2942:         return Err(ProgramError::InvalidInstructionData);
```

Description:

The **buy_sol** instruction in the swap program performs a CPI call to the revenue distribution program to withdraw SOL and transfer it to the user. However, the revenue distribution program's **withdraw_sol** implementation has a compatibility limitation with SPL Token-2022.

Currently, the distribution program performs validation that checks for a sibling instruction and requires its program ID to be the legacy SPL token program:

RUST

```
if sibling_ix.program_id != spl_token::ID {
    msg!("Sibling instruction's program ID is not SPL Token");
    return Err(ProgramError::InvalidInstructionData);
}
```

Note that the 2Z token mint was initialized with the legacy Token program and will never be migrated to Token-2022. However, if the swap program were to support Token-2022 tokens in the future (for other token types), transactions involving Token-2022 would fail at the withdrawal stage due to this strict program ID check in the revenue distribution program.

Potential Benefit:

Improved compatibility with Token-2022 integration for future swap program features

Recommendation:

If the swap program plans to support Token-2022 tokens in the future, the revenue distribution program's **withdraw_sol** instruction should be updated to accept both the legacy SPL token program (**spl_token::ID**) and the Token-2022 program (**spl_token_2022::ID**) as valid program IDs for the sibling instruction check.

E03: Increasing the max_len of new DenylistRegistry will help retain old blacklisted addresses

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/sol-2z-conversion-v1-d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b/on-chain/programs/converter-program/src/migration/sample_deny_list_registry_v2.rs#L8

>_ sample_deny_list_registry_v2.rs RUST

```
6: #[account]
7: #[derive(InitSpace, Debug)]
8: pub struct DenyListRegistryV2 {
9:     // we increase the size also
10:    #[max_len(100)]
```

Description:

The protocol implements a deny list system to block specific addresses from using the conversion functionality. When migrating from V1 to V2, the code changes the max length from 310 to 100 which is problematic:

In V1, the DenyListRegistry structure can hold up to 310 addresses:

RUST

```rust
#[account]
#[derive(InitSpace, Debug)]
pub struct DenyListRegistry {
    #[max_len(310)]
    pub denied_addresses: Vec<Pubkey>,
    pub last_updated: i64,
    pub update_count: u64,
}
```

However, in V2, the capacity is reduced to only 100 addresses:

RUST

```rust
#[account]
#[derive(InitSpace, Debug)]
pub struct DenyListRegistryV2 {
    #[max_len(100)]
    pub denied_addresses: Vec<Pubkey>,
    pub last_updated: i64,
    pub update_count: u64,
    pub new_field: u64,
}
```

During migration, the code attempts to clone all addresses from V1 to V2:
- If the V1 registry contains more than 100 addresses, It silently truncates the list upto just 100 addresses, up to 210 previously blacklisted addresses could regain access to the protocol, which is bad we don't want blacklisted addresses to regain access to the program

Potential Benefit:

Helps keep old blacklisted addresses.

Recommendation:

Increase the size of new denylist.

E04: Missing Validation for revenue_distribution_program ID in Buy-SOL Flow

Location: https://github.com/AdevarLabs/doublezero-rd-swap-merkle/blob/b68c1aaf876d74000a7ec67e6a6ebb99333df2fd/sol-2z-conversion-v1-d001e4ef49a2c1ddd867c26ba1c0a384f81fc65b/on-chain/programs/converter-program/src/buy_sol.rs#L81-L82

>_ buy_sol.rs RUST

```
79:     pub journal: UncheckedAccount<'info>,
80:     pub token_program: Interface<'info, TokenInterface>,
81:     /// CHECK: program address - TODO: implement address validations after client informs
         actual address
82:     pub revenue_distribution_program: UncheckedAccount<'info>,
83:     #[account(mut)]
84:     pub signer: Signer<'info>,
```

Description:

We reported that the buy_sol instruction lacked validation for the revenue_distribution_program ID.

This was addressed in PR #11, which added explicit program ID checks and updated the configuration registry accordingly.

'https://github.com/doublezerofoundation/sol-2z-conversion-v1/pull/11'

Potential Benefit:

Eliminates unnecessary TODO's in the buy_sol flow, enforce proper validation of **revenue_distribution_program**

Recommendation:

Add explicit program ID checks

About Us

About Adevar Labs

Adevar Labs is a boutique blockchain security firm specializing in web3 audits.

Built by a mix of experienced professionals in traditional enterprise and crypto natives who have contributed to some of the most critical projects in blockchain infrastructure.

Our team's background spans companies like Bitdefender, Asymmetric Research, Quantstamp, Chainproof, and Juicebox, and includes experience securing smart contracts, bridges, and L1 and L2 protocols across ecosystems like Solana, Ethereum, Polkadot, Cosmos, and MultiversX.

With over 100 audits completed and a portfolio that includes custom fuzzers, exploit modeling, and runtime testing frameworks, Adevar Labs brings both depth and precision to every engagement.

Our auditors have discovered critical vulnerabilities, built high-impact tooling, and placed in top positions in premier audit competitions including Code4rena and Sherlock.

Team members hold distinctions such as PhDs in software protection, and key roles at Fortune 500 companies, and leadership of flagship conferences like ETH Bucharest.

With team members having publications with over 1,100 academic citations and trusted by projects securing over $500M in on-chain value, Adevar Labs blends elite technical rigor with real-world security impact.

We also collaborate with some of the best independent security researchers in the web3 space.

Projects may optionally request specific contributors to be part of their audit.

Audit Methodology

Our audit methodology is specialized to provide thorough security assessments of Solana programs. We focus explicitly on rigorous manual analysis, detailed threat modeling, and careful validation of implemented fixes to ensure your programs operate securely and as intended.

1. Program Context and Architecture Analysis

Our auditors begin by deeply examining your Solana program's documentation, intended functionality, and account design. We meticulously map out program interactions, instruction processing flows, and state management logic. Special attention is given to understanding how your program interfaces with critical Solana system programs, such as the SPL Token Program, Stake Program, and System Program. Last but not least we check external integrations with other Solana projects and verify if the inputs and return values are handled properly.

2. Threat Modeling

We conduct targeted threat modeling tailored specifically for Solana's execution environment. We carefully define attacker capabilities and identify potential vulnerabilities that may arise from Solana-specific issues, including but not limited to:

- Unauthorized account data manipulation
- Improper ownership or signer verification

- Misuse of Program-Derived Addresses (PDAs)
- Incorrect use of Cross-Program Invocations (CPI)
- Failure to adequately handle account privileges or account states
- Risks stemming from rent-exemption and account initialization logic

3. In-depth Manual Security Review

Our experienced auditors perform an extensive manual security review of your Rust-based Solana programs. This involves a comprehensive line-by-line inspection of source code, focusing on common Solana vulnerabilities including, but not limited to:

- Missing or insufficient ownership checks
- Inadequate signer checks
- Incorrect handling of CPI calls and invocation privileges
- Arithmetic and integer overflow or underflow errors
- Unsafe deserialization and serialization of account data structures
- Improper token transfers and SPL-token logic issues
- Logic flaws in financial operations or state transitions
- Edge-case handling in instruction input validation
- Potential denial-of-service vectors related to transaction execution and account handling

During this stage, we clearly document any discovered vulnerabilities, including detailed descriptions, precise severity ratings, and recommendations for secure implementation. In situations where it is not clear how the vulnerability might be exploited we may also include a detailed proof-of-concept exploit including code snippets and instructions on how the exploit could be performed.

4. Detailed Fix Review and Validation

After the initial audit and your team's subsequent remediation efforts, we perform a comprehensive fix review to ensure the vulnerabilities identified have been effectively resolved.

We verify each fix individually, confirming that:

- Corrections effectively eliminate the security risks
- Changes do not inadvertently introduce new vulnerabilities or regressions
- The fixes align closely with Solana best practices and secure coding guidelines

Our detailed validation ensures that security improvements are robust, complete, and aligned with best practices specific to the Solana development ecosystem.

Confidentiality Notice

This report, including its content, data, and underlying methodologies, is subject to the confidentiality and feedback provisions in your agreement with Adevar Labs. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Adevar Labs.

Legal Disclaimer

The review and this report are provided by Adevar Labs on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Adevar Labs disclaims all warranties, expressed or implied, in connection with this report, its content, and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

You agree that access to and/or use of the report and other results of the review, including any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided.

You acknowledge that blockchain technology remains under development and is subject to unknown risks and flaws. Adevar Labs does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open-source or third-party software, code, libraries, materials, or information accessible through the report. As with the purchase or use of a product or service in any environment, you should use your best judgment and exercise caution where appropriate.