DoubleZero SOL 2Z Swap

Security Assessment

October 24th, 2025 — Prepared by OtterSec

Xiang Yin	soreatu@osec.io
Ajay Shankar Kunapareddy	d1r3wolf@osec.io
Alpha Toure	shxdow@osec.io
Robert Chen	r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

DoubleZero engaged OtterSec to assess the `sol-2z-swap` program. This assessment was conducted between September 15th and September 19th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we identified a vulnerability concerning the lack of strict verification of the token mint and revenue program when buying SOL, allowing users to execute fake trades and transfer arbitrary tokens (OS-DSS-ADV-00).

We also made suggestions for modifying the codebase to improve consistency, functionality, and mitigate potential security issues (OS-DSS-SUG-00), and advised removing redundant code instances (OS-DSS-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/doublezerofoundation/sol-2z-conversion-v1. This audit was performed against commit d001e4e.

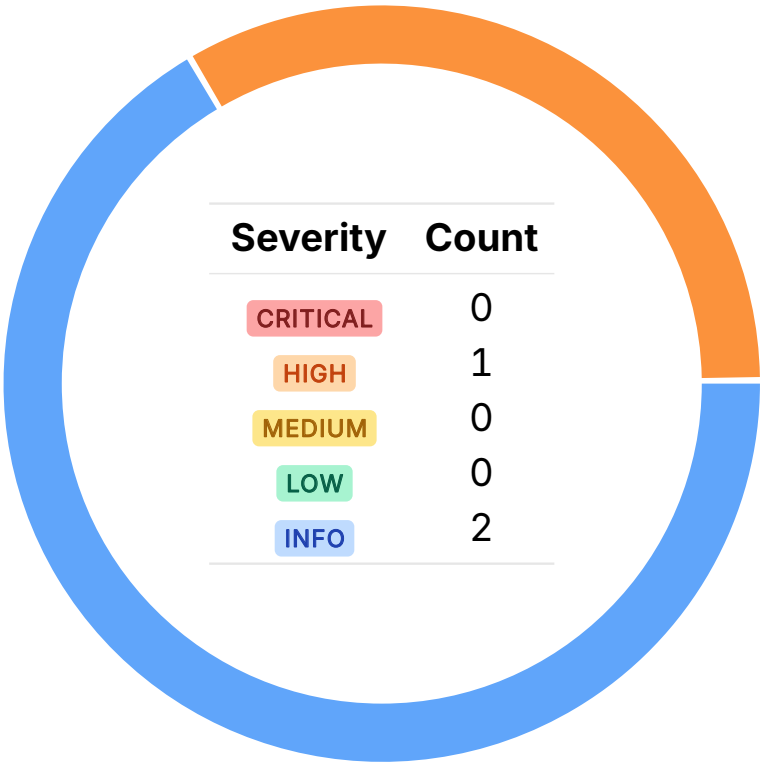
A brief description of the program is as follows:

Name	Description
sol-2z-swap	It allows users to swap 2Z tokens for SOL at dynamically calculated prices, verified via oracle attestation. It enforces trade restrictions, records transactions in a fills registry, and securely handles token transfers through on-chain and off-chain mechanisms.

03 — Findings

Overall, we reported 3 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	0
LOW	0
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-DSS-ADV-00	HIGH	RESOLVED	The `BuySol` instruction lacks strict verification of the token mint and revenue program, allowing users to execute fake trades and transfer arbitrary tokens.

Missing Account Validations HIGH OS-DSS-ADV-00

Description

In `BuySol`, the `double_zero_mint` and `revenue_distribution_program` accounts are not strictly validated. A user may supply a fake token mint instead of the official `2Z` token, and an arbitrary program instead of the intended revenue distribution program. This allows malicious actors to transfer worthless tokens to the treasury while triggering `SOL` withdrawals from any program they control. Consequently, users may receive `SOL` without spending legitimate `2Z` tokens, while also polluting the `fills_registry` with fake trades.

>_ on-chain/programs/converter-program/src/buy_sol.rs RUST

```rust
#[derive(Accounts)]
pub struct BuySol<'info> {
    [...]
    #[account(mut)]
    pub double_zero_mint: InterfaceAccount<'info, Mint>,
    [...]
    /// CHECK: program address - TODO: implement address validations after client informs actual
        ↪ address
    pub revenue_distribution_program: UncheckedAccount<'info>,
    #[account(mut)]
    pub signer: Signer<'info>,
}
```

Remediation

Verify the `revenue_distribution_program` account against the expected revenue distribution program to ensure the correct CPI is made, and also check that the `double_zero_mint` account equals the official 2Z mint (`DOUBLEZERO_MINT_KEY`).

Patch

Resolved in PR#11

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-DSS-SUG-00	Recommendation for modifying the codebase to improve consistency, functionality, and mitigate potential security issues.
OS-DSS-SUG-01	The codebase contains multiple cases of redundancy that should be removed for better maintainability and clarity.

Code Refactoring OS-DSS-SUG-00

Description

1. If `price_maximum_age` is set to a negative value, the timestamp check in `attestation:_utils::verify_attestation` will always fail because the absolute difference between the current time and the oracle timestamp is non-negative. This will result in all Oracle price verifications to be rejected. Ensure `price_maximum_age` is non-negative during its configuration.
2. Both `tokens_required` in `BuySol::process`, and `conversion_rate_u64` in `calculate_conversion_rate` are currently truncated when converting from decimal calculations to integers. This may result in underestimating the amount of tokens required or the conversion rate, resulting in the program undercharging users or underpaying `SOL`. By applying round-up (`ceil`) logic, these values would always fully account for fractional amounts, preventing small losses from accumulating.
3. In `BidTooLowEvent`, rename `bid_amount` to `bid_price` to clearly indicate it represents the offered price per `SOL`, not a token quantity. This improves clarity and consistency with the `ask_price` variable.

>_ on-chain/programs/converter-program/src/common/events/trade.rs RUST

```rust
#[event]
pub struct BidTooLowEvent {
    pub sol_amount: u64,
    pub bid_amount: u64,
    pub ask_price: u64,
    pub timestamp: i64,
    pub buyer: Pubkey,
    pub epoch: u64
}
```

Remediation

Update the codebase with the above refactors.

Code Redundancy OS-DSS-SUG-01

Description

1. The `deny_list_registry` account in `CalculateAskPrice` is unnecessary, as the module only computes conversion rates and does not enforce access control or modify state.

Remediation

Remove redundant code and functionality.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.