OtterSec

# DoubleZero Sentinel

Security Assessment

October 31st, 2025 — Prepared by OtterSec

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

# Table of Contents

# 01 — Executive Summary

## Overview

DoubleZero engaged OtterSec to assess the `sentinel` program. This assessment was conducted between September 8th and October 6th, 2025. For more information on our auditing methodology, refer to Appendix B.

## Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability where the function responsible for checking the leader schedule may incorrectly approve validators who never led, since it checks schedule existence rather than actual validator participation (OS-DZS-ADV-00). Additionally, Sentinel ignores access requests made via CPIs, risking locked SOL or spoofed requests (OS-DZS-ADV-01). Furthermore, Sentinel processes only the first `RequestAccess` instruction in a transaction, ignoring subsequent ones (OS-DZS-ADV-02).

We also made a suggestion regarding proper naming convention to ensure adherence to coding best practices (OS-DZS-SUG-00).

# 02 — Scope

The source code was delivered to us in a Git repository at github.com/doublezerofoundation/doublezero-offchain. This audit was performed against commit 8f7d7a2.

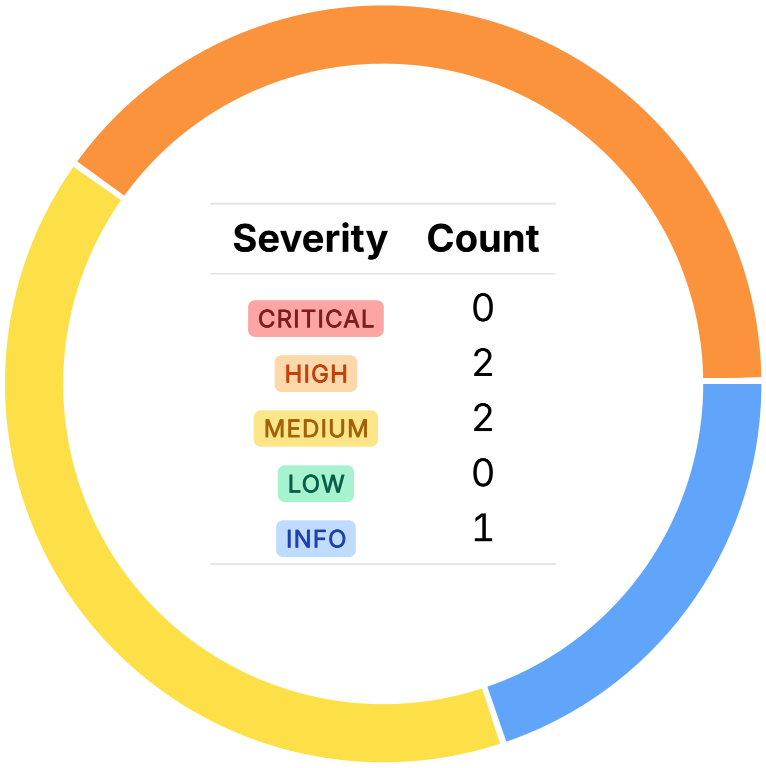
A brief description of the program is as follows:

| Name | Description |
|---|---|
| sentinel | The off-chain program that manages and enforces access control within the DoubleZero/Passport framework. |

# 03 — Findings

Overall, we reported 5 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 | 2 |
| MEDIUM | 2 |
| LOW | 0 |
| INFO | 1 |

# 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-DZS-ADV-00 | HIGH | RESOLVED ✓ | `check_leader_schedule` may incorrectly approve validators who never led, since it checks schedule existence rather than actual validator participation. |
| OS-DZS-ADV-01 | HIGH | RESOLVED ✓ | Sentinel ignores access requests made via CPIs, risking locked `SOL` or spoofed requests. |
| OS-DZS-ADV-02 | MEDIUM | RESOLVED ✓ | The Sentinel only processes the first `RequestAccess` instruction in a transaction, ignoring additional ones. |
| OS-DZS-ADV-03 | MEDIUM | RESOLVED ✓ | The listener and client lack validation for failed transactions involving spoofed accounts, allowing fake access requests to be processed. |

## Flawed Leader Schedule Validation HIGH OS-DZS-ADV-00

### Description

`solana::check_leader_schedule` has a logical flaw that may allow unqualified validators to pass the leader check. It currently returns true if the leader schedule does not exist or is empty, rather than explicitly verifying that the validator actually appears in the schedule. As a result, validators who never led in previous epochs may still pass the check. This undermines the Sentinel's access control and potentially grants access to unauthorized validators.

>_ *crates/sentinel/src/client/solana.rs* RUST

```rust
pub async fn check_leader_schedule(
    &self,
    validator_id: &Pubkey,
    previous_leader_epochs: u8,
) -> Result<bool> {
    let latest_slot = self.client.get_slot().await?;
    for slot in PreviousEpochSlots::new(latest_slot).take(previous_leader_epochs as usize) {
        let config = RpcLeaderScheduleConfig {
            identity: Some(validator_id.to_string()),
            ..Default::default()
        };
        if !self
            .client
            .get_leader_schedule_with_config(Some(slot), config)
            .await?
            .is_some_and(|schedule| schedule.is_empty())
        {
            return Ok(true);
        }
    }
    Ok(false)
}
```

### Remediation

Ensure to return true only when the result of `get_leader_schedule_with_config` is `Some` in `check_leader_schedule`.

### Patch

Resolved in PR#157.

## Overlooking of CPI Access Requests HIGH OS-DZS-ADV-01

### Description

The Sentinel only scans top-level instructions in a transaction, ignoring cross-program invocations (CPIs). If a `RequestAccess` is called via a CPI, Sentinel will not detect it, leaving the `SOL` locked in the access request account. Dropping the program ID check to detect such requests will open up spoofing risks, allowing attackers to fake requests and potentially gain unauthorized approvals. This issue specifically arises because Sentinel relies on the Passport program ID to identify instructions.

### Remediation

Restrict calls to `RequestAccess` to top-level calls only.

### Patch

Resolved in PR#62.

## Absence of Logic to Detect Multiple Access Requests MEDIUM OS-DZS-ADV-02

### Description

In the existing implementation, it is not possible for Sentinel to see multiple access requests within the same transaction. `solana::deserialize_access_request_ids` processes only the first Passport instruction in a transaction, ignoring any subsequent ones. If multiple `RequestAccess` instructions are included, only the first is deserialized, while the others remain unseen by Sentinel. This creates a mismatch between on-chain state (where all requests are valid) and Sentinel's monitoring state (where only one request is tracked). The issue stems from utilizing `.find` instead of iterating through all matching instructions.

>_ *crates/sentinel/src/client/solana.rs* RUST

```rust
fn deserialize_access_request_ids(txn: VersionedTransaction) -> Result<AccessIds> {
    let signature = txn.signatures.first().ok_or(Error::MissingTxnSignature)?;
    let compiled_ix = txn
        .message
        .instructions()
        .iter()
        .find(|ix| ix.program_id(txn.message.static_account_keys()) == &passport_id())
        .ok_or(Error::InstructionNotFound(*signature))?;
    [...]
}
```

### Remediation

Ensure to iterate over all the instructions in the transaction instead of finding and returning a single matching instruction.

### Patch

Resolved in PR#127.

## Failure to Filter Failed Transactions MEDIUM OS-DZS-ADV-03

### Description

The `listener` currently processes all log events, including those from failed transactions involving spoofed accounts. This allows attackers to emit fake `ACCESS_REQ_INIT_LOG` entries from failed transactions. Similarly, in the client, within `get_access_requests_from_signature`, the program does not verify that the fetched accounts are actually owned by the passport program, enabling spoofed accounts to be misinterpreted as valid access requests.

### Remediation

Filter out failed transactions in `listener::run` by checking the log event's RPC `value` and only handle logs from successful transactions. Also, check that `account.owner == passport_id` in `get_access_requests_from_signature`.

### Patch

Resolved in PR#160.

# 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-DZS-SUG-00 | Suggestion regarding ensuring adherence to coding best practices. |

## Code Maturity

OS-DZS-SUG-00

### Description

The variable `access_ids` is misleadingly named, as it is a plural while representing a single access request rather than multiple IDs. Change the name to `access_id`.

### Remediation

Implement the above-mentioned suggestion.

# 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.