Wincode 0.1.2

Security Assessment

January 26th, 2026 — Prepared by OtterSec

Alpha Toure shxdow@osec.io

Jamie Hill-Daniel jamie@osec.io

Table of Contents

01 — Executive Summary

Overview

Anza engaged OtterSec to assess the `wincode 0.1.2` library. This assessment was conducted between January 3rd and January 22nd, 2026. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 3 findings throughout this audit engagement.

In particular, we suggested marking the SchemaRead trait as unsafe, ensuring that implementers are aware of and responsible for upholding safety invariants (OS-WCD-SUG-00). Additionally, we advised removing or redesigning the Pod type such that consumers are required to acknowledge the safety invariants (OS-WCD-SUG-01). Lastly, we recommended including sufficient safety documentation for the Pod wrapper type (OS-WCD-SUG-02).

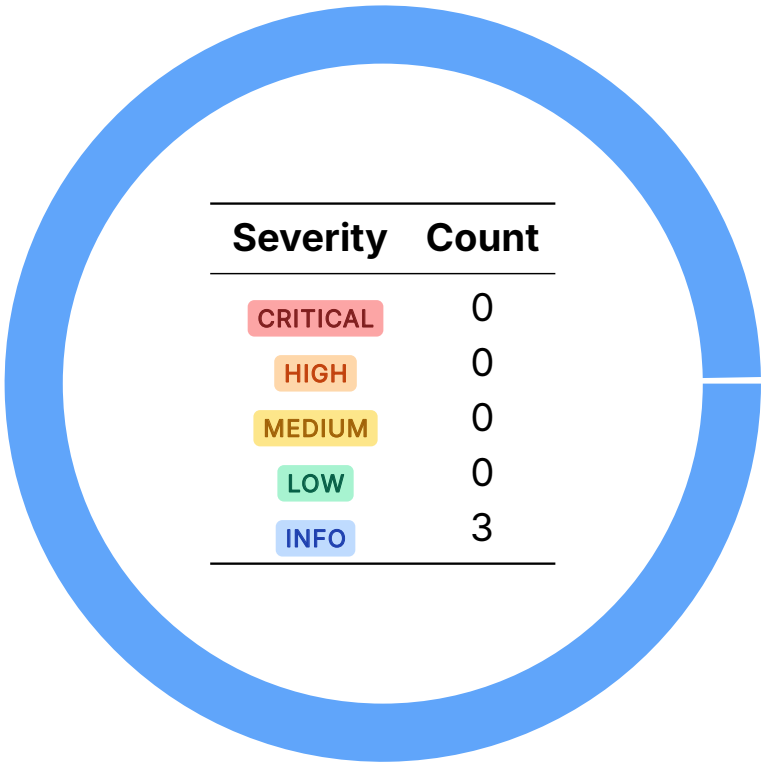
Scope

The source code was delivered to us in a Git repository at https://github.com/anza-xyz/wincode. This audit was performed against tag v0.1.2.

02 — 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	0
MEDIUM	0
LOW	0
INFO	3

03 — General Findings

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

ID	Description
OS-WCD-SUG-00	`SchemaRead` depends on correct implementation to maintain critical in-variants; thus, any bugs introduced by the implementer may compromise soundness.
OS-WCD-SUG-01	The `Pod` wrapper type is easy to misuse.
OS-WCD-SUG-02	The `Pod` wrapper type has insufficient safety documentation.

SchemaRead Safety OS-WCD-SUG-00

Description

The `SchemaRead` trait is defined as a safe, non-`unsafe` trait, yet its `read` method requires that the implementer correctly initialize `Self::Dst`. Failure to do so may compromise soundness in otherwise safe code.

>_ *demo.rs* RUST

```rust
use wincode::{io, ReadResult, SchemaRead, Deserialize};
use std::mem::MaybeUninit;

struct MyStruct {
    vec: Vec<u8>,
}

impl SchemaRead<'_> for MyStruct {
    type Dst = Self;

    // no-op read impl
    fn read(_: &mut io::Reader<'_>, _: &mut MaybeUninit<Self::Dst>) -> ReadResult<()> {
        Ok(())
    }
}

fn main() {
    let data = MyStruct::deserialize(&[]).unwrap();
    // accesses uninitialized data
    data.vec.get(0).unwrap();
}
```

Remediation

Mark the `SchemaRead` trait as `unsafe`, ensuring that implementers are aware of and responsible for upholding safety invariants.

Patch

This issue was resolved in PR#86.

Pod Safety OS-WCD-SUG-01

Description

The `Pod` wrapper type informs wincode that the contained type is *plain data*, and may be serialized/de-serialized with a single `memcpy`. However, there are no checks on this as shown below:

>_ *demo.rs* RUST

```rust
use wincode::{containers::Pod, Deserialize};

fn main() {
    let data = Pod::<&'static u8>::deserialize(&[
        0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88
    ]).unwrap();
    println!("{data}");  // SIGSEGV - dereferencing 0x8877665544332211
}
```

Remediation

Remove or redesign the `Pod` type such that consumers are required to acknowledge the safety invariants, for example, by renaming it `UnsafePod` or by utilizing `bytemuck::Pod` to enforce safety.

Pod Safety Documentation

OS-WCD-SUG-02

Description

Additionally, the currently documented safety invariants on `Pod` are insufficient to ensure Rust's rules are upheld. The current *Safety* section lists:

- The type must allow any bit pattern
- If used on a product type, all fields must be plain-data and have guaranteed layout (`repr(C)` or `repr(transparent)`
- The type must not contain references or pointers

However, this is not sufficient to ensure a type contains only plain data. In Rust, padding bytes are uninitialized. This implies that if the alignment of type members introduces padding, accessing the result of the serialization will access uninitialized bytes, as shown below:

>_ *demo.rs* RUST

```rust
use wincode::containers::{self, Pod};
use wincode::SchemaWrite;

#[derive(Clone, Copy, Default)]
#[repr(C)]
struct Address(u8, u32, u128); // Satisfies the listed requirements, but introduces padding
    ↪ bytes


#[derive(SchemaWrite)]
struct MyStruct {
    #[wincode(with = "containers::Vec<Pod<_>>")]
    vec: Vec<Address>,
}

fn main() {
    let data = wincode::serialize(&MyStruct {
        vec: vec![Address::default()]
    }).unwrap();
    println!("{data:?}");
}
```

Running this code under Miri will produce the following error:

>_ *output.txt* TEXT

```
Uninitialized memory occurred at alloc311[0x9..0xa], in this allocation:
alloc311 (Rust heap, size: 40, align: 1) {
```

```
    0x00 │ 01 00 00 00 00 00 00 00 __ __ __ 00 00 00 00 │ .........▒▒....
    0x10 │ __ __ __ __ __ __ __ __ 00 00 00 00 00 00 00 00 │ ▒▒▒▒▒▒▒▒........
    0x20 │ 00 00 00 00 00 00 00 00                         │ ........
}
```

Remediation

Include sufficient safety documentation for the `Pod` wrapper type.

Patch

This issue was resolved in PR#25.

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.