Security Audit - Wincode

conducted by Neodyme AG

Main Auditors:	Simon Klier
	Tobias Bucher
Supporting Auditors:	Justin Mietzner
	Florian Schweins
	Benjamin Walny
Administrative Lead:	Thomas Lambertz

February 12, 2026

Table of Contents

Appendices

1 | Executive Summary

Neodyme audited **Anza's** wincode library from December 2025 until January 2026.

The scope of this audit included both implementation security and deviations from the serialization format implemented by bincode.

The audit comprised both fuzzing of the library as well as extensive manual review by a team of experienced auditors familiar with Rust internals and the details of how wincode will be used in the context of Solana.

According to Neodyme's Rating Classification, **5 security-relevant** and **3 informational** were found. The number of findings identified throughout the audit, grouped by severity, can be seen in Figure 1.

Figure 1: Overview of Findings

The auditors reported all findings to the Anza developers, who addressed them promptly. The security fixes were verified for completeness by Neodyme.

2 | Introduction

From December 2025 until January 2026, Anza engaged Neodyme to do a detailed security analysis of wincode. A team of experienced security auditors from Neodyme conducted an audit of the library between the 1st of December 2025 and the 16th of January 2026.

The audit focused on the technical security of the Rust library and its behavioral differences from the bincode library, whose deserialization format it intends to reproduce. In the following sections, all findings are presented.

The auditors looked at the security of the library from different angles. First, they looked at classical memory safety and other security relevant bugs, using both fuzzing and manual review. Then, because wincode is a Rust library intended for use by the wider smart contract ecosystem, they also looked at the soundness of its exposed interface, i.e. whether the library can cause programs relying on it to trigger memory safety violations from safe Rust code.

Summary of Findings

All found issues were **quickly remediated**. In total, the audit revealed:

0 critical • **0** high-severity • **0** medium-severity • **5** low-severity • **3** informational

issues.

3 | Scope

The audit's scope comprised of the following components:

- Primarily, the **implementation** security of the library.
- Secondarily, the **serialization format** and whether it matches bincode's.
- Additionally, the soundness of the library, as defined by Rust's unsafe code guidelines.

Neodyme considers the source code, located at https://github.com/anza-xyz/wincode, in scope for this audit. Third-party dependencies are not in scope. Anza only relies on `thiserror`, a well-established library in the Rust ecosystem. For the code-generation part, it additionally relies on `darling`, `proc-macro2`, `quote`, and `syn`, all of which are well-established for code generation. During the audit, minor changes and fixes were made by Anza, which the auditors also reviewed in-depth.

Relevant source code revisions are:

- `9f0ffa346d95c31b94486b7bfea724b73330c42f` · Audited revision
- `8a1a002290aecddd0de74734e474debbc76ed92f` · Revision up to which fixes in response to Neodyme's findings were reviewed – note that not all changes since the revision above were reviewed

4 | Project Overview

This section briefly outlines wincode's functionality, design, and architecture.

Functionality

Wincode is serialization library for the Rust programming language with a bincode-compatible format. It provides out-of-the-box serialization for Rust's basic data types such as integers, floats, strings, etc. and has macros to automatically serialize custom composite data types as well.

The library also allows users to work around Rust's orphan rule by overriding the serialization format for third-party data types, similar to serde_with. There's also limited support for viewing byte slices directly as deserialized types, without any copying. This is called *zero-copy* deserialization.

Wincode in the context of Solana

The bincode serialization format is used pervasively both in the context of Agave, the reference implementation of the Solana validator, and in the context of smart contracts, where it is the "default" serialization format, with the Solana SDK providing explicit support for serialization program invocation arguments in this format. It's also used for account data serialization in Anchor, a popular framework for writing smart contracts for Solana. This makes the bincode format an important foundation of Solana.

Since maintenance recently stopped for bincode, wincode is the only maintained implementation of the wincode serialization format. In the light of that, wincode is going to get used more in the immediate future.

Performance is important for usage in Solana smart contracts because executions are limited to the amount of compute units (CUs) requested for a transaction. Wincode emphasizes performance and in-place initialization and the main APIs are structured accordingly. Rust code tends to be move-heavy since most results are returned by-value. E.g. the deserialization of a `Vec` in bincode contains `vec.push(decode(…))`. This places the deserialized vector element on the stack and then moves it into the vector. The equivalent in-place initialization is more performant because it requirement of that move, it looks more like `decode(&mut vec.spare_capacity_mut()[0])`, i.e. the deserialized data is directly written to the final memory location.

Serialization Format

Wincode largely inherits its serialization format from bincode, only differing intentionally in few places. The bincode format is a binary serialization format that is not self-describing. Integers and floats are simply emitted in little-endian byte order with no metadata, composite data types are laid out by concatenating the serialization of their member values. In the resulting byte stream, one

cannot tell apart objects without knowing the schema of the types that were serialized. This leads to a relatively succinct format.

Besides new features like customizable length encoding for containers and custom tag encoding for `enums`, **wincode differs in one point from bincode** in its default configuration: Dynamic container sizes are limited to 4 MiB by default, larger containers will fail both serialization and deserialization. This is to ensure in-place initialization and preventing large claimed container sizes in deserialization from causing out-of-memory or similar issues on the host system by triggering large memory allocations.

Architecture

Wincode consists of two Rust crates:

- `wincode-derive` offers a macro to automatically "derive" a customizable serialization format for composite data types (`struct`, `enum`).
- `wincode` provides traits to implement and customize serialization for data types, with a focus on in-place initialization by allowing callers to pass an out-parameter. It also contains facilities for fast serialization of types that can be directly copied from their in-memory representation, such as strings, integers on little-endian architectures, or custom data types solely consisting of such types. The serialization format can also be configured globally through a `Configuration` struct, for example allowing one to select endianness or variable-length integer encoding.

Tests

Wincode is extensively unit-tested using a property-testing library that automatically generates random test cases (and reduces them in case of failures). Wincode is tested for panics and behavior differences to bincode this way.

5 | Fuzzing

As one pillar of the audit, Neodyme conducted a security and correctness evaluation of the wincode deserialization crate using coverage-guided fuzzing with afl.rs.

The core of the project was a differential fuzzing setup in which the same untrusted byte streams were deserialized into identical Rust data structures using both wincode and bincode. Any behavioral divergence between the two implementations, such as crashes, panics, or mismatched outputs, was treated as a potential bug or security-relevant issue.

To maximize coverage across edge cases and structural complexity, the fuzzer first generated 100 random but well-typed Rust structures, which were emitted into an auto-generated Rust source file and compiled directly into the fuzz target.

See an example of a randomly generated struct type with a custom `PartialEq` trait below:

```rust
#[derive(Serialize, Deserialize, SchemaWrite, SchemaRead)]
struct RandomStruct94 {
    field0: [u32; 7],
    field1: Arc<[f32]>,
    field2: Box<[StaticStructSub]>,
    field3: VecDeque<i128>,
    field4: u8,
    field5: StaticStructSub,
    field6: f32,
    field7: BinaryHeap<i128>,
    field8: Box<[u64]>,
    field9: Rc<[i32]>,
    field10: DynamicStructSubSub,
    field11: bool,
    field12: ZeroCopyStructSub,
    field13: i32,
    field14: Rc<[u64]>,
}

impl PartialEq for RandomStruct94 {
    fn eq(&self, other: &Self) -> bool {
        self.field0 == other.field0 &&
        self.field1.iter().zip(other.field1.iter()).all(|(a, b)| a == b ||
(a.is_nan() && b.is_nan())) &&
        self.field2 == other.field2 &&
        self.field3 == other.field3 &&
        self.field4 == other.field4 &&
        self.field5 == other.field5 &&
        (self.field6 == other.field6 || (self.field6.is_nan() &&
other.field6.is_nan())) &&
```

```
29          self.field8 == other.field8 &&
30          self.field9 == other.field9 &&
31          self.field10 == other.field10 &&
32          self.field11 == other.field11 &&
33          self.field12 == other.field12 &&
34          self.field13 == other.field13 &&
35          self.field14 == other.field14
36      }
37  }
```

During each fuzzing iteration, AFL provided a random byte stream that the harness split into two logical components: an index selecting one of the 100 pre-generated structures, and the remaining bytes used as the serialized input to be deserialized into that structure.

Below is an excerpt of the fuzzing harness:

```
1  #[cfg(fuzzing)]
2  fn fuzz() {
3      afl::fuzz!(|data: &[u8]| {
4          if data.is_empty() {
5              return;
6          }
7          // Select a random generated struct based on input data
8          let idx = (data[0] as usize) % 100;  // Assuming 100 structs
9
10         let data = &data[1..];
11         try_random_generated!(data, idx);
```

This process was executed independently for both wincode and bincode, after which the resulting values were compared for semantic equivalence.

Below is an excerpt of the `try_random_generated` macro which attempts deserialization and then compares deserialization results between wincode and bincode:

```
1  94 => {
2      let mut copy = $data.to_vec();
3      let wincode_res: Result<RandomStruct94, _> = wincode::deserialize(&mut
   copy);
4      let bincode_res: Result<RandomStruct94, _> = bincode::deserialize($data);
5      match (wincode_res, bincode_res) {
6          (Ok(w), Ok(b)) => { if w != b { panic!("bincode and wincode results
   differ "); } },
7          (Ok(_), Err(_)) => { panic!("Wincode succeeded, Bincode failed"); },
8          (Err(_), Ok(_)) => { panic!("Bincode succeeded, Wincode failed"); },
9          (Err(_), Err(_)) => {}
10     }
```

```
}
95 => {
    let mut copy = $data.to_vec();
    let wincode_res: Result<RandomStruct95, _> = wincode::deserialize(&mut copy);
    let bincode_res: Result<RandomStruct95, _> = bincode::deserialize($data);
    match (wincode_res, bincode_res) {
        (Ok(w), Ok(b)) => { if w != b { panic!("bincode and wincode results differ "); } },
        (Ok(_), Err(_)) => { panic!("Wincode succeeded, Bincode failed"); },
        (Err(_), Ok(_)) => { panic!("Bincode succeeded, Wincode failed"); },
        (Err(_), Err(_)) => {}
    }
}
96 => {
    let mut copy = $data.to_vec();
    let wincode_res: Result<RandomStruct96, _> = wincode::deserialize(&mut copy);
    let bincode_res: Result<RandomStruct96, _> = bincode::deserialize($data);
    match (wincode_res, bincode_res) {
        (Ok(w), Ok(b)) => { if w != b { panic!("bincode and wincode results differ "); } },
        (Ok(_), Err(_)) => { panic!("Wincode succeeded, Bincode failed"); },
        (Err(_), Ok(_)) => { panic!("Bincode succeeded, Wincode failed"); },
        (Err(_), Err(_)) => {}
    }
}
```

Because some structures contained floating-point fields and other values with non-trivial equality semantics (such as NaNs), the structure generator also outputs custom comparison traits to ensure meaningful equivalence checks rather than relying on naïve structural equality.

Overall, this approach combined differential fuzzing, randomized type generation, and AFL's coverage guidance to systematically explore deserialization edge cases, validate behavioral parity between implementations, and the attempt to uncover subtle correctness and robustness issues.

Ultimately, this approach did not uncover any crashes or behavioral discrepancies. While this result suggests the absence of obvious implementation errors or mismatches, it does not formally guarantee soundness of the wincode deserializer.

6 | Findings

This section outlines all of our findings found via extensive manual review. They are classified into one of five severity levels, detailed in Appendix B.

All findings are listed in Table 1 and further described in the following sections.

Identifier	Name	Severity	Status
`ND-WIN01-LO-01`	Unsoundness via `Pod` struct instantiation	LOW	Fix proposed (#89, #165)
`ND-WIN01-LO-02`	Unsoundness via incorrect `SchemaRead::read`	LOW	Fixed (#86, #79)
`ND-WIN01-LO-03`	Unsoundness via incorrect `SchemaRead::TYPE_META`	LOW	Fixed (#86, #80)
`ND-WIN01-LO-04`	Unsoundness via incorrect `SchemaWrite::size_of`	LOW	Fixed (#86, #81)
`ND-WIN01-LO-05`	Unsoundness via incorrect `SeqLen::write_bytes`	LOW	Fixed (#82, #99)
`ND-WIN01-IN-01`	Invalid serialization for large inputs	INFORMATIONAL	Fixed (#90, #102)
`ND-WIN01-IN-02`	Broken implementation of `Writer` trait	INFORMATIONAL	Fixed (#87)
`ND-WIN01-IN-03`	Inconsistent zerocopy deserialization	INFORMATIONAL	Ack'ed (#91)

Table 1: Findings

[ND-WIN01-LO-01] Unsoundness via `Pod` struct instantiation

Severity	Impact	Affected Component	Status
LOW	Undefined behaviour	`struct Pod<T>`	Fix proposed (#89, #165)

Instantiating `Pod<T>` with an type that is not zero copy can result in undefined behaviour by safe code.

While it is documented that `Pod<T>` should only be used with zero copy types, this is actually not enforced anywhere. Therefore safe code can cause undefined behaviour by deserializing non zero copy types through `Pod<T>`.

This example creates a char which is not an Unicode scalar value which will cause a segmentation fault in the rust standard library when trying to print it.

```rust
let c = Pod::<char>::deserialize(&[0xff; 4]).unwrap();
println!("{:?}", c);
```
example.rs

Resolution

The unsafety has been acknowledged and at the time of writing different solutions are being discussed. See issues #89 and pull request #165.

[ND-WIN01-LO-02] Unsoundness via incorrect `SchemaRead::read`

Severity	Impact	Affected Component	Status
LOW	Uninitialized memory exposed to safe code	`SchemaRead`	Fixed (#86, #79)

The trait `SchemaRead` is used to describe how a type is deserialized. For a new struct, one can either derive it to delegate deserialization to the struct's fields, or write a custom implementation.

The custom deserialization is done in the `SchemaRead::read` method which gets a `&mut MaybeUninit<T>` parameter which it should initialize when returning `Ok(())` or leave in any state when returning an error.

wincode/src/schema/mod.rs

```rust
pub trait SchemaRead<'de> {
    // […]
    fn get(reader: &mut impl Reader<'de>) -> ReadResult<Self::Dst> {
        let mut value = MaybeUninit::uninit();
        Self::read(reader, &mut value)?;
        // SAFETY: `read` must properly initialize the `Self::Dst`.
        Ok(unsafe { value.assume_init() })
    }
}
```

As the `SAFETY` comment specifies, `SchemaRead::get` relies on `SchemaRead::read` initializing the passed-in `&mut MaybeUninit<T>`.

Defining the `SchemaRead::read` method and returning `Ok(())` without calling `dst.write(…)` first results in uninitialized memory being used by safe code.

Proof of concept reading 64 bytes of uninitialized stack space in safe code:

example.rs

```rust
#![allow(unused)]

use std::mem::MaybeUninit;
use wincode::io::Reader;
use wincode::ReadResult;
use wincode::SchemaRead;
use wincode::TypeMeta;

struct Uninit([u64; 16]);

impl<'de> SchemaRead<'de> for Uninit {
    type Dst = Uninit;
    fn read(reader: &mut impl Reader<'de>, dst: &mut MaybeUninit<Uninit>)
        -> ReadResult<()>
    {
        Ok(())
```

```
    }
}

fn main() {
    let uninit: Uninit = wincode::deserialize(&[]).unwrap();
    println!("{:?}", uninit.0);
}
```

```
[0, 0, 0, 140729608245616, 107220333894416, 107220333893904, 240,
140729608122368, 1732123328, 0, 140729599868928, 140729599864832,
107220333894416, 126737626158849, 140729599864832, 4096]
```

example output

Suggested Mitigation

One possible remediation would be to declare the `SchemaRead` trait as unsafe. This would still allow most users to derive the trait, that works without any `unsafe` in the user's code.

Resolution

Fixed in PR #86 by marking the trait unsafe, as suggested. Neodyme has verified the fix.

[ND-WIN01-LO-03] Unsoundness via incorrect `SchemaRead::TYPE_META`

Severity	Impact	Affected Component	Status
LOW	UB exposed to safe code	`trait SchemaRead`	Fixed (#86, #80)

Implementations of the trait `SchemaRead` can specify metadata about the implementing type, e.g. a fixed number of bytes that the deserialization will read or that the implementing type can simply be `memcpy`ed for deserialization. Both options exist for optimizations.

src/schema/containers.rs

```rust
impl<'de, T, Len> SchemaRead<'de> for Vec<T, Len>
// […]
{
    fn read(reader: &mut impl Reader<'de>, dst: &mut MaybeUninit<Self::Dst>)
        -> ReadResult<()>
    {
        let len = Len::read::<T::Dst>(reader)?;
        let mut vec: vec::Vec<T::Dst> = vec::Vec::with_capacity(len);

        match T::TYPE_META {
            TypeMeta::Static { zero_copy: true, .. } => {
                let spare_capacity = vec.spare_capacity_mut();
                // SAFETY: T::Dst is zero-copy eligible (no invalid bit patterns, no layout requirements, no endianness checks, etc.).
                unsafe { reader.copy_into_slice_t(spare_capacity)? };
                // SAFETY: `copy_into_slice_t` fills the entire spare capacity or errors.
                unsafe { vec.set_len(len) };
            }
            // […]
        }

        dst.write(vec);
        Ok(())
    }
}
```

As the `SAFETY` comment specifies, the implementation relies on the `SchemaRead::TYPE_META` being accurate for the implementing type.

This means that setting the associated constant `SchemaRead::TYPE_META` to `TypeMeta::Static { size: _, zero_copy: true }` can lead to undefined behavior in safe code.

Proof of concept reaching an unreachable state in safe code:

example.rs

```rust
#![allow(unused)]

use std::mem::MaybeUninit;
use wincode::io::Reader;
use wincode::ReadResult;
use wincode::SchemaRead;
use wincode::TypeMeta;

enum Never {}

impl<'de> SchemaRead<'de> for Never {
    type Dst = Never;
    const TYPE_META: TypeMeta = TypeMeta::Static {
        size: 0,
        zero_copy: true,
    };
    fn read(reader: &mut impl Reader<'de>, dst: &mut MaybeUninit<Never>)
        -> ReadResult<()>
    {
        panic!();
    }
}

#[derive(SchemaRead)]
#[repr(C)]
struct NeverWrapper {
    pub never: Never,
}

fn main() {
    let wrapper: &NeverWrapper = wincode::deserialize(&[]).unwrap();
    println!("entered unreachable code");
    match wrapper.never {}
}
```

example output

```
entered unreachable code
Illegal instruction
```

Suggested Mitigation

One possible remediation would be to declare the `SchemaRead` trait as unsafe. This would still allow most users to derive the trait, that works without any `unsafe` in the user's code.

Resolution

Fixed in PR #86 by marking the trait unsafe, as suggested. Neodyme has verified the fix.

[ND-WIN01-LO-04] Unsoundness via incorrect `SchemaWrite::size_of`

Severity	Impact	Affected Component	Status
LOW	Out of bounds memory access	`trait SchemaWrite`	Fixed (#86, #81)

Defining the `SchemaWrite::size_of` method to return less than the actually needed bytes can result in out of bounds memory accesses by safe code.

When serializing, wincode trusts the size returned by `SchemaWrite::size_of`. If this length is too small it will create a buffer with insufficient size for deserialisation.

src/serde.rs

```rust
#[cfg(feature = "alloc")]
fn serialize(src: &Self::Src) -> WriteResult<Vec<u8>> {
    let capacity = Self::size_of(src)?;
    let mut buffer = Vec::with_capacity(capacity);
    let mut writer = buffer.spare_capacity_mut();
    Self::serialize_into(&mut writer, src)?;
    let len = writer.len();
    unsafe {
        #[allow(clippy::arithmetic_side_effects)]
        buffer.set_len(capacity - len);
    }
    Ok(buffer)
}
```

Suggested Remediation

One possible remediation would be to declare the `SchemaWrite` trait as unsafe.

Resolution

Fixed in PR #86 by marking the trait unsafe, as suggested. Neodyme has verified the fix.

[ND-WIN01-LO-05] Unsoundness via incorrect `SeqLen::write_bytes`

Severity	Impact	Affected Component	Status
LOW	Out of bounds memory access	`trait SeqLen`	Fixed (#82, #99)

Defining the `SeqLen::write_bytes_needed` method to return less than the actually needed bytes can result in out of bounds memory accesses by safe code.

If `SeqLen::write_bytes_needed` returns a smaller size than is actually needed data will be serialized out of bounds of the buffer.

src/schema/mod.rs

```rust

fn write_elem_slice<T, Len>(writer: &mut impl Writer, src: &[T::Src]) -> WriteResult<()>
where
    Len: SeqLen,
    T: SchemaWrite,
    T::Src: Sized,
{
    if let TypeMeta::Static {
        size,
        zero_copy: true,
    } = T::TYPE_META
    {
        let needed = Len::write_bytes_needed(src.len())? + src.len() * size;
        // SAFETY: `needed` is the size of the encoded length plus the size of the slice (bytes).
        // `Len::write` and `writer.write(src)` will write `needed` bytes,
        // fully initializing the trusted window.
        let writer = &mut unsafe { writer.as_trusted_for(needed) }?;
        Len::write(writer, src.len())?;
        // SAFETY: `T::Src` is zero-copy eligible (no invalid bit patterns, no layout requirements, no endianness checks, etc.).
        unsafe { writer.write_slice_t(src)? };
        writer.finish()?;
        return Ok(());
    }
    write_elem_iter::<T, Len>(writer, src.iter())
}

```

This trait could probably be replaced by `SchemaRead<Dst=usize> + SchemaWrite<Src=usize>` as it is just a specialized trait for serializing `usize`.

Suggested Remediation

One possible remediation would be to declare the `SeqLen` trait as unsafe.

Resolution

Fixed in PR #99 by marking the trait unsafe, as suggested. Neodyme has verified the fix.

[ND-WIN01-IN-01] Invalid serialization for large inputs

Severity	Impact	Affected Component	Status
INFORMATIONAL	Deserialization failure	bincode compat	Fixed (#90, #102)

When deserializing, wincode tries to make sure that allocated buffers are not too large, in order to prevent simple denial of service (DoS) by inputs pretending to be large. However, when serializing, `BincodeLen::write` does not check whether the serialized data conforms to size limitations. This means that serialization can succeed despite outputting data that will fail to be deserialized.

Proof of concept showing failed roundtrip of large `Vec<u8>` (4 MiB + 1 B):

example.rs

```rust
fn main() {
    let large: Vec<u8> = vec![0; 4 * 1024 * 1024 + 1];
    let serialized = wincode::serialize(&large).unwrap();
    println!("{:?}", wincode::deserialize::<Vec<u8>>(&serialized));
}
```

example output

```
Err(PreallocationSizeLimit { needed: 4194305, limit: 4194304 })
```

This behavior also differs from the bincode crate, which does not implement such limits by default:

example2.rs

```rust
use bincode::config::legacy;
use bincode::config::standard;

fn main() {
    let large: Vec<u8> = vec![0; 4 * 1024 * 1024 + 1];
    let serialized = wincode::serialize(&large).unwrap();
    println!("{:?}", wincode::deserialize::<Vec<u8>>(&serialized));
    println!("{:?}",
        bincode::decode_from_slice::<Vec<u8>, _>(&serialized, standard())
            .map(|_| ())
    );
    println!("{:?}",
        bincode::decode_from_slice::<Vec<u8>, _>(&serialized, legacy())
            .map(|_| ())
    );
}
```

example output

```
Err(PreallocationSizeLimit { needed: 4194305, limit: 4194304 })
Ok(())
Ok(())
```

Suggested Remediation

It would be possible to check the written length for deserializability, in `BincodeLen::write`. Alternatively, one could accept larger inputs but not preallocate as much, for bincode compatibility.

Resolution

This was fixed with PR #102 by introducing a length check during serialization. The fix was verified by Neodyme.

[ND-WIN01-IN-02] Broken implementation of `Writer` trait

Severity	Impact	Affected Component	Status
INFORMATIONAL	None	`impl Writer for [u8]`	Fixed (#87)

`impl Writer for [u8]` is broken, it mangles the serialization of the passed data by restarting serialization at the beginning of the provided buffer for each item. This means, for example, that `&str` gets serialized by serializing a length as little-endian `u64`, and then overwriting that length field with the actual string.

The bug is hidden due to another bug that prevents actually using the `Writer` implementation on `[u8]` – all methods receiving a `Writer` take `&mut impl Writer` instead of `&mut (impl Writer + ?Sized)`, excluding the `Writer` implementation for `[u8]`, as `[u8]` is not `Sized`.

Search-replacing using `sed -i 's/mut impl Writer/mut (impl Writer+?Sized)/g' **.rs`, one can observe the bug using the following proof of concept:

example.rs

```rust
use std::io;
use std::io::Write as _;
use wincode::SchemaWrite as _;

fn main() {
    let mut buffer = [0xbb; 8];
    <&str>::write(&mut buffer[..], &"aaa").unwrap();
    io::stdout().write_all(&buffer).unwrap();
}
```

Hexdump of output:

```
00000000  61 61 61 00 00 00 00 00  bb bb bb bb bb bb bb bb  |aaa.............|
00000010
```

Suggested Remediation

The `impl Writer for [u8]` trait implementation should probably be removed: It can't be fixed, as `[u8]` cannot track the cursor position needed for the `Writer::write` method. Users can use the `impl Writer for &mut [u8]` instead, which *can* track the cursor position.

One could additionally replace `&mut impl Writer` by `&mut (impl Writer + ?Sized)` to relax the unnecessary restrictions, same for `&mut impl Reader`.

Resolution

The trait implementation was removed in PR #87. Neodyme verified the fix.

[ND-WIN01-IN-03] Inconsistent zerocopy deserialization

Severity	Impact	Affected Component	Status
INFORMATIONAL	None	`trait ZeroCopy`	Ack'ed (#91)

Integer types implement `ZeroCopy` if the target CPU is little-endian. Since these types have alignment requirements, it depends on the (accidental) alignment of the passed-in byte slice whether the deserialization will succeed.

Since the user usually has little control over the alignment of the input byte slice, this can create inconsistent errors.

The crate seems to conflate "plain old data", which can be `memcpy`ed and types with no alignment requirements, both under the `trait ZeroCopy`. This can also be seen in the documentation of the trait, which states "The type must not have any [...] layout requirements [...]."

example.rs

```rust
fn dump(buffer: &[u8]) {
    println!("{:?}", wincode::deserialize::<&i32>(buffer));
}

fn main() {
    let buffer = [0; 7];
    dump(&buffer[0..4]);
    dump(&buffer[1..5]);
    dump(&buffer[2..6]);
    dump(&buffer[3..7]);
}
```

example output

```
Ok(0)
Err(UnalignedPointerRead)
Err(UnalignedPointerRead)
Err(UnalignedPointerRead)
```

Suggested Remediation

One possible solution would be to create two different traits, one capturing the "plain old data" requirement, i.e. the type being `memcpy`able, and one trait for having no alignment requirements.

Resolution

Anza acknowledged the behavior in issue #91, but concluded that no code changes were necessary at this time.

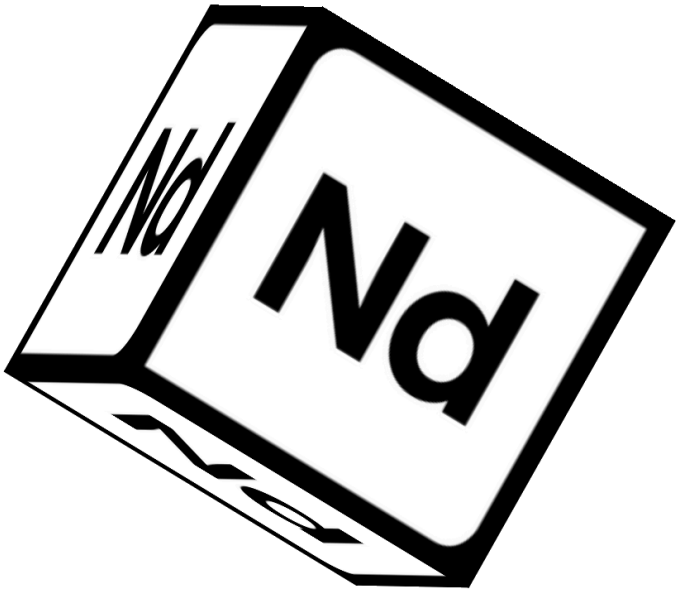
A | About Neodyme

Security is difficult.

To understand and break complex things, you need a certain type of people. People who thrive in complexity, who love to play around with code, and who don't stop exploring until they fully understand every aspect of it. That's us.

Our team never outsources audits. Having found over 80 High or Critical bugs in Solana's core code itself, we believe that Neodyme hosts the most qualified auditors for Solana programs. We've also found and disclosed critical vulnerabilities in many of Solana's top projects and have responsibly disclosed issues that could have resulted in the theft of over $10B in TVL on the Solana blockchain.

All of our team members have a background in competitive hacking. During such hacking competitions, called CTFs, we competed and collaborated while finding vulnerabilities, breaking encryption, reverse engineering complicated algorithms, and much more. Through the years, many of our team members have won national and international hacking competitions, and keep ranking highly among some of the hardest CTF events worldwide. In 2020, some of our members started experimenting with validators and became active members in the early Solana community. With the prospect of an interesting technical challenge and bug bounties, they quickly encouraged others from our CTF team to look for security issues in Solana. The result was so successful that after reporting several bugs, in 2021, the Solana Foundation contracted us for source code auditing. As a result, Neodyme was born.



B | Vulnerability Severity Rating

We use the following guideline to classify the severity of vulnerabilities. Note that we assess each vulnerability on an individual basis and may deviate from these guidelines in cases where it is well-founded. In such cases, we always provide an explanation.

Severity	Description
CRITICAL	Vulnerabilities that will likely cause loss of funds. An attacker can trigger them with little or no preparation, or they are expected to happen accidentally. Effects are difficult to undo after they are detected.
HIGH	Bugs that can be used to set up loss of funds in a more limited capacity, or to render the contract unusable.
MEDIUM	Bugs that do not cause direct loss of funds but that may lead to other exploitable mechanisms, or that could be exploited to render the contract partially unusable.
LOW	Bugs that do not have a significant immediate impact and could be fixed easily after detection.
INFORMATIONAL	Bugs or inconsistencies that have little to no security impact, but are still noteworthy.

Additionally, we often provide the client with a list of nit-picks, i.e. findings whose severity lies below Informational. In general, these findings are not part of the report.

Neodyme AG

Dirnismaning 55
Halle 13
85748 Garching
Germany

E-Mail: contact@neodyme.io

https://neodyme.io