Radically Open Security

Code Audit Report

Diesel

V 1.0
Amsterdam, October 6th, 2025
Confidential

Document Properties

Client	Diesel
Title	Code Audit Report
Target	• Diesel ORM for Rust
Version	1.0
Pentesters	Morgan Hill, Stefan Grönke
Authors	Morgan Hill, Stefan Grönke, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	October 2nd, 2025	Morgan Hill, Stefan Grönke	Initial draft
0.2	October 3rd, 2025	Marcus Bointon	Review
1.0	October 6th, 2025	Marcus Bointon	1.0

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Table of Contents

1 Executive Summary

1.1 Introduction

Between September 1, 2025 and October 3, 2025, Radically Open Security B.V. carried out a penetration test for Diesel.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

1.2 Scope of Work

The scope of the penetration test was limited to the following target:

- Diesel ORM for Rust

The scoped services are broken down as follows:

- Code audit: 4 days
- Reporting: 1 days
- **Total effort: 5 days**

1.3 Project Objectives

ROS will perform an audit of the Diesel ORM in order to assess its security. To do so ROS will access Diesel source code and guide its developers in attempting to find vulnerabilities, exploiting any such found to try and gain further access and elevated privileges.

1.4 Timeline

The security audit took place between September 1, 2025 and October 3, 2025.

1.5 Results In A Nutshell

During this crystal-box penetration test we found 1 Moderate and 2 Low-severity issues.

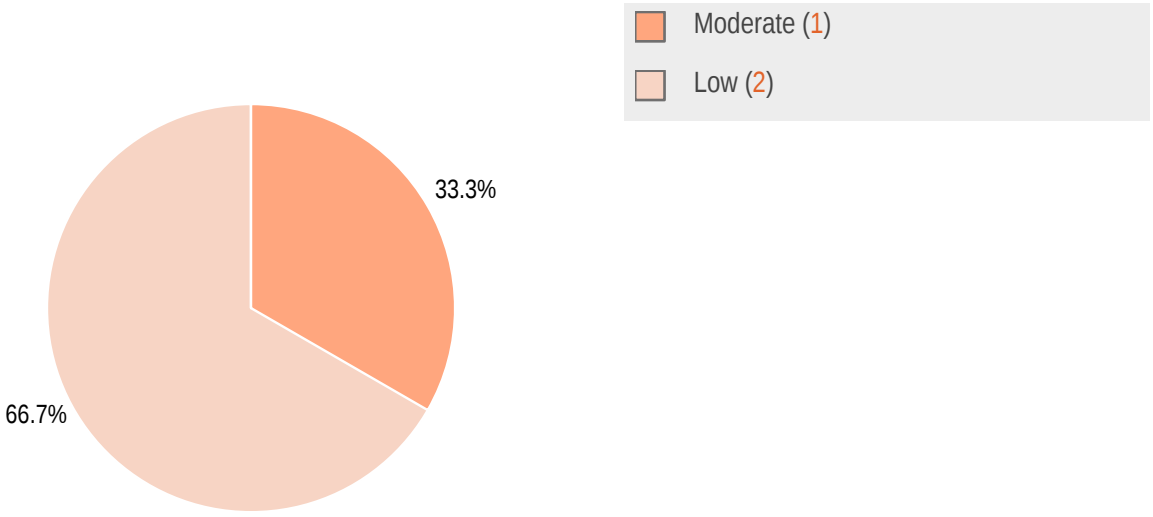
In this Diesel code audit we discovered a handful of minor issues in the database driver code near the FFI boundary. The most significant of which exposed a small region of memory with the precondition that earlier checks had been bypassed

CLN-002 (page 9). The other issues include a panic with similar preconditions CLN-003 (page 10), or in the case of CLN-004 (page 12), a hypothetical question about Rust's memory model.

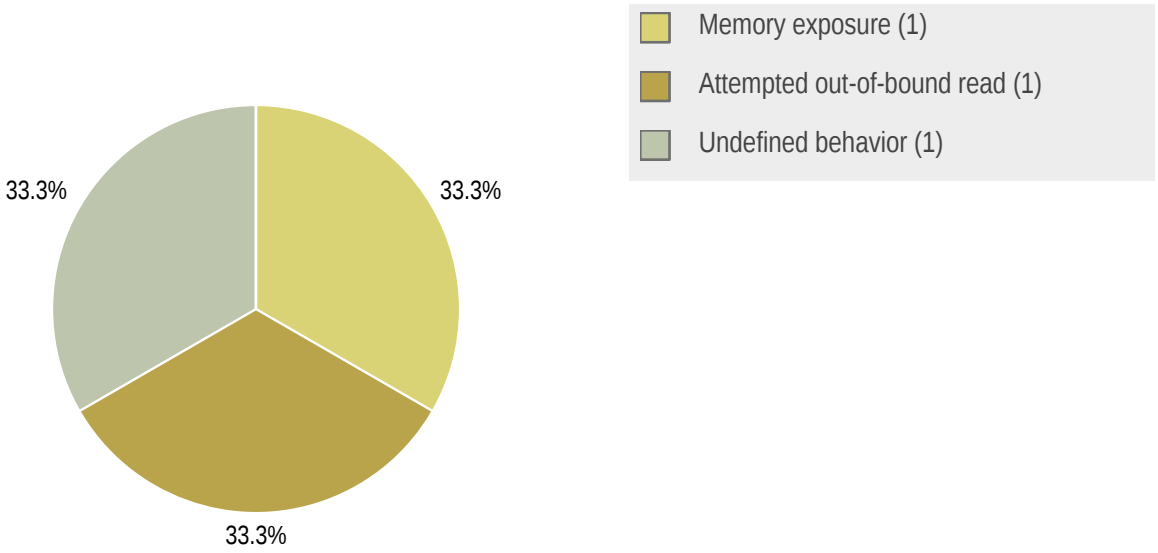
1.6 Summary of Findings

Info	Description
CLN-002 **Moderate** **Type:** Memory exposure **Status:** resolved	A missing bounds checks in MysqlValue::time_value results in an over read, potentially returning up to 47 bytes of uninitialized memory.
CLN-003 **Low** **Type:** Attempted out-of-bound read **Status:** resolved	A MysqlType can be constructed for which the numeric_value method panics.
CLN-004 **Low** **Type:** Undefined behavior **Status:** resolved	A caveat of Rust's memory model and mem::Forget results in the theoretical unsoundness of reading from invalid pointers.

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

Info	Recommendation
CLN-002 **Moderate** **Type:** Memory exposure **Status:** resolved	• Validate the length of the buffer before reading the `MysqlTime`.
CLN-003 **Low** **Type:** Attempted out-of-bound read **Status:** resolved	• Add a length check to the `numeric_value` method.
CLN-004 **Low** **Type:** Undefined behavior **Status:** resolved	• Wrap the vector in `mem::ManuallyDrop` to ensure that drop will not be called when the vector leaves scope instead of calling `mem::Forget`.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. **Reconnaissance**
   We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.
2. **Enumeration**
   We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.
3. **Scanning**
   Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.
4. **Obtaining Access**
   We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately though provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2021) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: http://www.pentest-standard.org/index.php/Reporting

These categories are:

- **Extreme**
  Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**

  High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**

  Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**

  Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**

  Low risk of security controls being compromised with measurable negative impacts as a result.

3 Findings

We have identified the following issues:

3.1 CLN-002 — MySQL time over read

Vulnerability ID: CLN-002 **Status:** Resolved

Vulnerability type: Memory exposure

Threat level: Moderate

Description:

A missing bounds checks in `MysqlValue::time_value` results in an over read, potentially returning up to 47 bytes of uninitialized memory.

Technical description:

The `time_value` method does not perform length checking on the buffer before performing an unaligned read on the pointer to the buffer to fill a `MysqlTime` type. This results in a 47-byte over read. The majority of the time the bytes read from a valid `MysqlTime` that is returned to the caller, however sometimes an error is returned if one of the type's invariants is violated.

Setting the first byte of the buffer to `1` appears to improve the reliability.

```
use std::{slice,mem};
let val = MysqlValue::new_internal(&[1u8], MysqlType::DateTime);
let val = val.time_value().expect("sometimes the memory read isn't a supported MysqlTime");
let mem = unsafe {
    slice::from_raw_parts(
        &raw const val as *const u8,
        mem::size_of::<MysqlTime>()
        )
    };
println!("{:?}\n{:?}",val,mem);
```

```
MysqlTime { year: 1836020481, month: 1835627621, day: 1948283749, hour: 1830839656, minute:
 1919905125, second: 1701978233, second_part: 8369779874455381089, neg: false, time_type:
 MysqlTimestampType(1869639797), time_zone_displacement: 1684370546 }
[1, 115, 111, 109, 101, 116, 105, 109, 101, 115, 32, 116, 104, 101, 32, 109, 101, 109, 111, 114,
 121, 32, 114, 101, 97, 100, 32, 105, 115, 110, 39, 116, 32, 97, 32, 115, 117, 112, 112, 111, 114,
 116, 101, 100, 32, 77, 121, 115]
```

The over read occurs here: `diesel/src/mysql/value.rs#L47`

```
pub(crate) fn time_value(&self) -> deserialize::Result<MysqlTime> {
```

```
    match self.tpe {
        MysqlType::Time | MysqlType::Date | MysqlType::DateTime | MysqlType::Timestamp => {
            let ptr = self.raw.as_ptr() as *const MysqlTime;
            let result = unsafe { ptr.read_unaligned() };
            if result.neg {
                Err("Negative dates/times are not yet supported".into())
            } else {
                Ok(result)
            }
        }
        _ => Err(self.invalid_type_code("timestamp")),
    }
}
```

Much like CLN-003 (page 10) the binding process already checks length of this buffer. Consequently, reaching this bug requires a bug in the binding process or manual construction of the `MysqlType`.

Impact:

Up to 47 bytes of memory are exposed when a `MysqlValue` is constructed with a buffer of insufficient size to store a `MysqlTime`. Some invariant violations result in errors.

Recommendation:

- Validate the length of the buffer before reading the `MysqlTime`.

Update 2025-09-12 07:25:

Fixed by `diesel/pull/4756` .

3.2 CLN-003 — MySQL numeric values are not length checked

Vulnerability ID: CLN-003 **Status:** Resolved

Vulnerability type: Attempted out-of-bound read

Threat level: Low

Description:

A `MysqlType` can be constructed for which the `numeric_value` method panics.

Technical description:

The `MysqlValue::numeric_value` method does not validate the size of the raw buffer contained in the `MysqlValue` before attempting to read bytes from the buffer to form a numerical type.

The code in question can be found in `diesel/src/mysql/value.rs#L66` .

This snippet demonstrates the attempted over read:

```
let val = MysqlValue::new_internal(&[0u8], MysqlType::Long);
let _ = val.numeric_value();
```

Rust guards for such attempts and panics:

```
range end index 4 out of range for slice of length 1
```

The `new_internal` method is locally available within the crate, and is accessed in `BindData::value`. The construction of a `BindData` type via `from_output` ensures that the buffer is allocated to fit the length of the expected type. This ensures that `length` will always be within the buffer, and thus the over read will not be attempted in standard operation.

```
fn from_tpe_and_flags((tpe, flags): (ffi::enum_field_types, Flags)) -> Self {
    // newer mysqlclient versions do not accept a zero sized buffer
    let len = known_buffer_size_for_ffi_type(tpe).unwrap_or(1);
    let mut bytes = vec![0; len];
    let length = bytes.len() as libc::c_ulong;
    let capacity = bytes.capacity();
    let ptr = NonNull::new(bytes.as_mut_ptr());
    mem::forget(bytes);

        Self {
        tpe,
        bytes: ptr,
        length,
        capacity,
        flags,
        is_null: super::raw::ffi_false(),
        is_truncated: Some(super::raw::ffi_false()),
    }
}
```

A developer can choose to expose the `MysqlValue::new` method via the `i-implement-a-third-party-backend-and-opt-into-breaking-changes` feature. Developers doing so are then responsible for providing a buffer that matches the type in order to avoid the panic.

Impact:

A panic may be triggered in the event that a `MysqlValue` is constructed with a buffer that is not at least the length of the `MysqlType`. However, this case will not occur in normal operation.

Recommendation:

- Add a length check to the `numeric_value` method.

Update 2025-09-12 07:25:

Fixed by `diesel/pull/4756` .

3.3 CLN-004 — Semantically unsound pointer in MySQL bindings

Vulnerability ID: CLN-004	**Status:** Resolved
Vulnerability type: Undefined behavior	
Threat level: Low	

Description:

A caveat of Rust's memory model and `mem::Forget` results in the theoretical unsoundness of reading from invalid pointers.

Technical description:

The MySQL driver extracts the raw components (pointer, capacity, and length) of vectors then calls `mem::Forget` on the vector. This is done to "leak" the vector and transfer ownership of the vector to the point in the code where the vector is reconstructed from its component parts.

This approach is theoretically unsound as the pointer is not guaranteed to be valid after `mem::Forget` has been called. Therefore, using the pointer after this point is undefined behavior. The data will linger as it has not been freed but the pointer is not necessarily valid.

In reality, the pointer is always valid given current implementations of Rust, but it is not guaranteed that this will remain true.

One example of this can be found here: https://github.com/diesel-rs/diesel/blob/3f5e603565ac9f60d93c8844bdbb5691a48a6a0b/diesel/src/mysql/connection/bind.rs#L232

In principle, all instances of `mem::Forget` are affected.

Impact:

Theoretical unsoundness that may become practical unsoundness in future Rust versions.

Recommendation:

Wrap the vector in `mem::ManuallyDrop` to ensure that drop will not be called when the vector leaves scope instead of calling `mem::Forget`. This will inhibit the de-allocation of the vector without semantically invalidating the pointer.

Example:

```
let mut v = vec![0, 1];
let mut v = ManuallyDrop::new(v);
let ptr = v.as_mut_ptr();
let cap = v.capacity();
let len = v.len();
let v2 = unsafe { Vec::from_raw_parts(ptr, len, cap) };
```

Update `2025-10-02 11:10`:

Fixed by pull request `diesel/pull/4756` .

It is unclear whether this is undefined behavior in Rust, as discussed in a Zulip thread.

4 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

4.1 NF-001 — GitHub release workflow

We looked at the release workflow in the GitHub repository. This workflow is interesting because it produces binaries used by Diesel end users. We evaluated the workflow against three district threats: malicious dependencies, secret exposure, and malicious activation.

The workflow uses `cargo-dist` from a release binary installed via a release script in the project's GitHub repo (https://github.com/axodotdev/cargo-dist). The installation script verifies checksums of the artifacts it downloads, however the workflow does not verify the installation script itself. Diesel therefore trusts that the installation script for a release has not been tampered with. Diesel already trusts the `cargo-dist` tool and it is likely that an attacker able modify the installation script hosted in the release would also be able to modify the tool itself, so the trust isn't significantly broadened. Consequently, we do not list this as a finding.

In a similar vein `cargo-cyclonedx` is used to generate an SBOM. It is installed in the same fashion as `cargo-dist` we therefore apply the same reasoning to determine that this is not a finding.

The actions in the workflow all come from the GitHub actions organization. We consider this a trusted source.

The only secret used is a `GITHUB_TOKEN` which is set as an environment variable. The workflow logs are available to all GitHub users, however we did not find any instances of the token being leaked into the logs. Assuming that all tools run in the workflow are trusted then this secret will not be leaked.

The workflow triggers on pull requests, including PRs from untrusted forks if they contain tags formatted like semantic versions. A feature of GitHub actions is that the `GITHUB_TOKEN` used when the PR is a fork only has read-only access. Therefore, the malicious code from a fork may run in the workflow, but it will not have sufficient access to create a release or perform any other actions on the Diesel repository.

4.2 NF-005 — Potentially superfluous unsafe mysql_bind

Converting the Rust bind type to a `C` MySQL client library bind type uses `unsafe` to initialize the type. The current code appears sound but requires additional effort to comprehend.

The correctness relies on a zero byte being a valid value in all positions of all struct members.

`diesel/src/mysql/connection/bind.rs#L453` :

```
unsafe fn mysql_bind(&mut self) -> ffi::MYSQL_BIND {
    use std::ptr::addr_of_mut;

    let mut bind: MaybeUninit<ffi::MYSQL_BIND> = mem::MaybeUninit::zeroed();
    let ptr = bind.as_mut_ptr();
```

```
    unsafe {
        addr_of_mut!((*ptr).buffer_type).write(self.tpe);
        addr_of_mut!((*ptr).buffer).write(
            self.bytes
                .map(|p| p.as_ptr())
                .unwrap_or(std::ptr::null_mut()) as *mut libc::c_void,
        );
        addr_of_mut!((*ptr).buffer_length).write(self.capacity as libc::c_ulong);
        addr_of_mut!((*ptr).length).write(&mut self.length);
        addr_of_mut!((*ptr).is_null).write(&mut self.is_null);
        addr_of_mut!((*ptr).is_unsigned)
            .write(self.flags.contains(Flags::UNSIGNED_FLAG) as ffi::my_bool);

        if let Some(ref mut is_truncated) = self.is_truncated {
            addr_of_mut!((*ptr).error).write(is_truncated);
        }

        // That's what the mysqlclient examples are doing
        bind.assume_init()
    }
}
```

We would propose a more explicit safe initialization of the type:

```
unsafe fn mysql_bind(&mut self) -> ffi::MYSQL_BIND {
    ffi::MYSQL_BIND {
        length: &mut self.length,
        is_null: &mut self.is_null,
        buffer: self
            .bytes
            .map(|p| p.as_ptr())
            .unwrap_or(std::ptr::null_mut()) as *mut libc::c_void,
        error: self.is_truncated.map_or(null_mut(), |ref mut t| t),
        row_ptr: 0 as *mut u8,
        store_param_func: None,
        fetch_result: None,
        skip_result: None,
        buffer_length: self.capacity as libc::c_ulong,
        offset: 0,
        length_value: 0,
        param_number: 0,
        pack_length: 0,
        buffer_type: self.tpe,
        error_value: false,
        is_unsigned: self.flags.contains(Flags::UNSIGNED_FLAG) as ffi::my_bool,
        long_data_used: false,
        is_null_value: false,
        extension: 0 as *mut libc::c_void,
    }
}
```

However, with further context from the developer the code is structured they way it is for intentional polymorphism. The `MYSQL_BIND` type has different layouts when used against MySQL versus when it is used against MariaDB. The `unsafe` initialization is necessary so that the common fields can be set without referencing implementation specific fields that would break compilation against either backend.

We suggest as an alternative that the `Default` trait could be implemented in `mysqlclient-sys`. In principle this would provide the required polymorphism without the `unsafe` block. Making this change would need to be done upstream and result in a major backwards compatibility breaking change for Diesel. We therefore accept that there are good reasons the for the code to remain as it is.

The function itself must remain unsafe. As stated in the comment, it is the caller's responsibility to ensure the lifetime of `MYSQL_BIND` does not exceed the lifetime of the `BindData` object. The lifetime can't be assured automatically across the FFI boundary.

5 Future Work

- **Continually re-assess unsafe usage**
  As the language and the underlying libraries develop, it may become possible to replace unsafe blocks with safe Rust code. We advise staying abreast of new developments to progressively ratchet down unsafe usage.
- **Retest of findings**
  When mitigations for the vulnerabilities described in this report have been deployed, perform a repeat test to ensure that they are effective and have not introduced other security problems.
- **Regular security assessments**
  Security is a process that must be continuously evaluated and improved; this penetration test is just a single snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security.

6 Conclusion

We discovered 1 Moderate and 2 Low-severity issues during this penetration test.

This code audit of Diesel focussed on the FFI boundary and stateful aspects of the ORM. Some effort was also directed at the query builder, however it proved difficult to develop effective tooling within the timeframe of the engagement.

Diesel is a stalwart of the Rust ecosystem that aims to provide the ergonomics of an ORM without the performance penalty. To achieve this, much of the work of building queries is performed at compile time. As in many Rust projects it is not unusual to find large ASTs built in the type system. The type system rigorously ensures the validity of queries. Diesel also (with one exception) only performs prepared statements in combination with most queries being built at compile time from a strongly typed AST, and consequently, SQL injections are extremely unlikely.

All of the findings were related to the database drivers. In these drivers Rust must interact with C libraries which naturally requires unsafe blocks where the developers must provide their own memory safety guarantees. However, only one finding was (CLN-002 (page 9)) was explicitly due to unsafe memory handling.

The findings we encountered only affect developers using Diesel APIs against the documented path. These secondary vulnerabilities were immediately addressed, allowing us to conclude the project with all findings resolved.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process that must be continuously evaluated and improved; this penetration test is just a one-time snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Appendix 1 Testing Team

Morgan Hill	Morgan is a seasoned security consultant with a background in IoT and DevOps. He currently specialises in Rust and AVoIP.
Stefan Grönke	Stefan applies his curiosity and love for development to the breaking of and into systems constructively.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.