The Hazardous Interface

Why String-Based Query Protocols Are Architectural Defects

Patrick Borowy
Opoka Research — January 2026

ABSTRACT

This study analyzes **1,374** classified CVE records to evaluate the structural causes of database vulnerabilities. We propose a distinction between **Implementation Defects** (bugs in vendor code) and **Interface Hazards** (risks transferred to users by design). We find that Interface Hazards outnumber Implementation Defects by **3.08:1**, with **79.1%** of SQL injection flaws occurring outside the database engine. Even ORMs—abstractions built specifically to mitigate these risks—exhibit an **88.5%** Interface Hazard rate. These findings confirm that the vulnerability is not a user error, but a mathematical inevitability of the protocol. You cannot document your way out of a toxic interface.

---

1. Introduction

For thirty years, the security community has framed SQL injection as a failure of operator diligence. The standard remediation model—sanitization and "secure coding"—presumes the protocol is sound and the developer is the defect. This framing is wrong.

A database that accepts string concatenation for queries (`"SELECT..." + input`) is not a tool with sharp edges; it is a toxic material. In civil engineering, when lead pipes were found to poison water, the solution was not "careful drinking"—it was replacing the infrastructure. String-based query protocols are the lead pipes of the software stack.

The industry has failed to make this transition. By retaining a protocol that permits catastrophic failure by default, database vendors have externalized the risk to every downstream consumer.

1.1 Formal Characterization of the Defect

The injection hazard is a failure of compositional semantics: there exists an untrusted input that causes the parser to produce a structurally different AST than the one intended by the developer. The following Coq proof [17] verifies this claim.

```
(* Full source: github.com/opoka-cloud/hazardous-interface *)

Inductive AST : Type :=
  | Data      : string → AST    (* Terminal data node *)
  | Structure : string → AST    (* Structural node    *)
  | Empty     : AST.

(* T: trusted query    U: untrusted input *)
Definition T : string := "select name where id=".
Definition U : string := "1; drop table".

Definition parse (s : string) : AST :=
  if string_dec s (T ++ U) then Structure "SPLIT_DETECTED"
  else Data s.

Definition inject_data (t : AST) (val : string) : AST :=
  match t with
  | Data s => Data (s ++ val)
  | _      => Data val
  end.

Lemma parse_TU : parse (T ++ U) = Structure "SPLIT_DETECTED".
Proof. vm_compute. reflexivity. Qed.

Lemma inject_TU : inject_data (parse T) U = Data (T ++ U).
Proof. vm_compute. reflexivity. Qed.

Theorem injection_condition_satisfiable :
  ∃ Input, parse (T ++ Input) ≠ inject_data (parse T) Input.
Proof.
  exists U.
  rewrite parse_TU, inject_TU.
  discriminate.
Qed.
```

The proof proceeds by witness: the concrete input `U = "1; drop table"` causes the parser to produce a `Structure` node where the intended semantics requires a `Data` node. These are distinct constructors—`discriminate` closes the goal immediately. This proves that SQL injection is not a "bug" to be patched, but a property of the interface design. Sanitization attempts to restrict U to a "safe" subset, but this requires enumerating all dangerous strings in all contexts—a negative proof obligation that grows with language complexity. The only deterministic solution is to eliminate string-accepting query interfaces entirely.

1.2 The Regression of the Scripting Era

The formalization above is not novel computer science—it is the application of a foundational principle that modern systems enforce elsewhere. Operating systems evolved to strictly separate instructions from data: the NX bit, W^X (Write XOR Execute), and ASLR exist precisely because mixing code and data in memory proved catastrophic. A memory page can be writable or executable, but never both.

The database layer regressed. In the transition to web-scale architectures, the industry abandoned the strict typing of the compilation era in favor of the "flexibility" of string interpretation. We are effectively running modern data infrastructure on the equivalent of pre-protected-mode memory management—permitting users to write data that the server executes as code.

The fact that this requires formalization in 2026 is not a claim of novelty; it is an indictment of the industry's architectural standards.

2. Related Work

The Mitigation Paradigm. The dominant research trajectory has focused on retrofitting safety onto string-based protocols. Approaches include static analysis [6], runtime monitoring [7], instruction set randomization [14], and machine learning [8]. While these methods reduce risk, they accept the underlying "shotgun parser" architecture [4] as immutable—attempting to inspect toxic water rather than replacing the pipes. Our analysis of the ORM failure rate (88.5%) suggests that externalized mitigation cannot fully compensate for a non-composable wire protocol.

Type-Safe Construction. The feasibility of eliminating injection by construction is well-established. Meijer et al. [9] demonstrated in 2006 that language-integrated queries (LINQ) eliminate the hazard entirely. McClure [11] and Cook [12] formalized compile-time SQL validation. However, twenty years later, these innovations remain implemented as optional client-side abstractions. Our work provides the empirical justification—based on 1,374 CVEs—for elevating these concepts from "client conveniences" to "protocol mandates."

Historical Timeline. Forristal identified the vulnerability in 1998 [13]. Halfond et al. [3] provided the canonical taxonomy in 2006. The fact that OWASP still ranks SQL injection in the Top 3 nearly thirty years later [15] is the strongest empirical evidence that the "user education" strategy is bankrupt.

Formal Foundations. Sassaman et al. [4] established that ad-hoc input handling creates "weird machines" capable of unintended computation—the same class of vulnerability that W^X protections eliminate at the memory layer. Bratus et al. [5] demonstrated that exploitation is fundamentally a language recognition problem. Our Section 1.1 extends this by formalizing injection not as a grammar violation, but as a failure of compositional semantics—proving that string concatenation is inherently non-composable with parsing. Section 1.2 observes that the database industry has failed to adopt the code/data separation that operating systems now enforce by default.

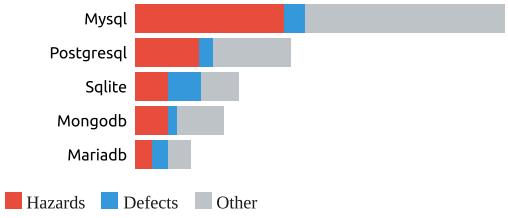
3. Methodology

3.1 Data Source

We retrieved vulnerability data from the NIST National Vulnerability Database (NVD) using the NVD API v2.0 for five widely-deployed database systems:

PRODUCT	CLASSIFIED CVES	IMPL. DEFECTS	INTERFACE HAZARDS
Mariadb	99	29 (29.3%)	30 (30.3%)
Mongodb	158	17 (10.8%)	58 (36.7%)
Mysql	656	38 (5.8%)	264 (40.2%)
Postgresql	277	26 (9.4%)	113 (40.8%)
Sqlite	184	60 (32.6%)	58 (31.5%)
Total	**1374**	**170 (12.4%)**	**523 (38.1%)**

Table 1: CVE classification by database product



3.2 Taxonomy

Implementation Defects (Vendor Error): Internal bugs such as memory corruption, race conditions, or type confusion. These are unintentional failures in the vendor's code that the vendor must fix.

Interface Hazards (Design Transfer): Risks inherent to the API contract. The database functions "as designed," but the design permits dangerous operations that require users to navigate the hazard correctly.

3.3 Note on MongoDB

While MongoDB uses structured BSON rather than SQL strings, it is included to demonstrate that Interface Hazards persist whenever an interface mixes code and data. MongoDB's `$where` operator accepts JavaScript expressions—the same class of "code-in-data" hazard, regardless of transport format.

4. Results

4.1 The Ratio of Hazard

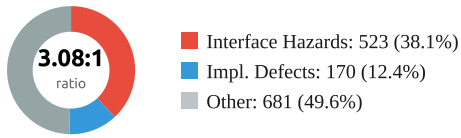


Figure 1: Primary CVE classification (n=1374)

Ratio: 3.08:1 — The database vulnerability surface is dominated not by code quality issues, but by the hazardous nature of the interface itself.

4.2 The Contagion Effect

Of the 220 identified SQL Injection vulnerabilities, **79.1%** occur downstream of the database engine.

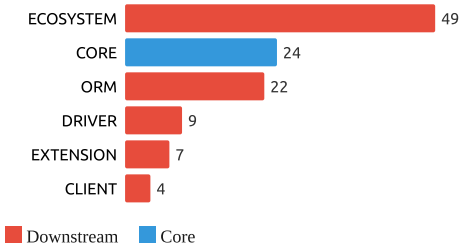


Figure 2: SQL Injection by location (n=115 known)

The root cause is not that downstream tools parse SQL poorly—most use well-tested client libraries. The problem is that the database *accepts* string queries at all. This permissiveness means that at some point in the stack, a string must be constructed. The interface permits the dangerous pattern, even when safe alternatives exist.

4.3 The Failure of ORMs

Object-Relational Mappers exist specifically to abstract query generation and prevent injection. Yet our analysis found:

Total ORM-related CVEs	**26**
ORM Interface Hazards	**23**
ORM Hazard Rate	**88.5%**

The high hazard rate within ORM failures acts as empirical proof of **Spolsky's Law of Leaky Abstractions** [16]. Because the underlying wire protocol remains string-based, the abstraction cannot seal the toxicity. When an ORM encounters complex logic and permits a "raw SQL" bypass, the underlying hazard immediately surfaces. An abstraction built on top of a defective foundation inherits the defect.

5. Discussion

5.1 The Sanitization Delusion

Despite thirty years of documentation, SQL injection remains a dominant vulnerability class (16% of dataset). This is not because developers are careless; it is because sanitization is the wrong abstraction.

Asking developers to correctly escape every input, in every context, forever, is asking them to be perfect. Humans are not perfect. In safety engineering, any control that requires 100% operator vigilance is considered a failure by design. A security model that relies on universal developer compliance is not a security model—it is a prayer.

5.2 Interface Hazards as Design Defects

The 3.08:1 prevalence of Interface Hazards confirms that database architectures systematically offload security complexity to the user. The existence of "safe" mechanisms (Prepared Statements) does not excuse the presence of the "unsafe" mechanism (String Concatenation). A type-safe interface—where injection is impossible by construction—would eliminate this vulnerability category entirely.

The database industry is distributing lead pipes and attributing the poisoning to insufficient user caution.

5.3 The Friction of Safety

Theoretically, Stored Procedures solve the injection hazard by defining logic strictly server-side. However, they introduce an architectural schism, forcing developers to maintain logic in two disparate environments (application code vs. database schema). The industry rejected this high-friction safety in favor of the agility of string-based queries.

We are currently trapped in a false dichotomy: **Safe but Rigid** (Stored Procedures) or **Agile but Toxic** (String Building). A modern architecture must offer the safety of the former with the ergonomics of the latter.

6. Conclusion

Our analysis of 1,374 CVEs provides a definitive characterization of the database vulnerability landscape:

- **3.08:1** — Interface Hazards dominate Implementation Defects
- **79.1%** — SQL injection risks transferred to the downstream ecosystem
- **88.5%** — ORM hazard rate proves mitigation layers inherit the defect
- **Section 1.1** — The machine-verified proof demonstrates that string concatenation is semantically unsound

The database industry currently operates on a model of distributing products with inherent design hazards and attributing the resulting casualties to operator negligence. It is time to stop asking developers to be perfect and start demanding protocols that are safe by construction.

To eliminate this hazard class, future database protocols must adhere to three axioms:

Axiom 1 (Atomic Transmission): The protocol must transmit query structure and data values as separate, immutable streams.

Axiom 2 (Static Resolution): The query structure must be derived from a finite set of build-time artifacts, never from runtime string evaluation.

Axiom 3 (Fail-Safe Defaults): The interface must reject any ambiguous payload by design, rather than relying on optional sanitization flags.

We do not need better documentation. We need a new wire format.

References

[1] NIST National Vulnerability Database. nvd.nist.gov

[2] MITRE Common Weakness Enumeration. cwe.mitre.org

[3] Halfond, W.G., Viegas, J., & Orso, A. (2006). A Classification of SQL Injection Attacks and Countermeasures. *IEEE ISSSE*.

[4] Sassaman, L., Patterson, M.L., Bratus, S., & Shubina, A. (2013). The Halting Problems of Network Stack Insecurity. *USENIX ;login:*.

[5] Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., & Shubina, A. (2011). Exploit Programming: From Buffer Overflows to Weird Machines. *USENIX ;login:*.

[6] Gould, C., Su, Z., & Devanbu, P. (2004). Static Checking of Dynamically Generated Queries. *ICSE 2004*.

[7] Halfond, W.G. & Orso, A. (2005). AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. *ASE 2005*.

[8] Uwagbole, S.O., Buchanan, W.J., & Fan, L. (2017). Applied Machine Learning Predictive Analytics to SQL Injection Attack Detection. *ICITST 2017*.

[9] Meijer, E., Beckman, B., & Bierman, G. (2006). LINQ: Reconciling Object, Relations and XML. *SIGMOD 2006*.

[10] jOOQ. Type Safe SQL in Java. jooq.org

[11] McClure, R. & Krüger, I. (2005). SQL DOM: Compile Time Checking of Dynamic SQL Statements. *ICSE 2005*.

[12] Cook, W.R. & Rai, S. (2005). Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. *ICSE 2005*.

[13] Forristal, J. (rain.forest.puppy) (1998). NT Web Technology Vulnerabilities. *Phrack Magazine*, Issue 54.

[14] Boyd, S.W. & Keromytis, A.D. (2004). SQLrand: Preventing SQL Injection Attacks. *ACNS 2004*.

[15] OWASP Foundation (2021). OWASP Top Ten. owasp.org/Top10

[16] Spolsky, J. (2002). The Law of Leaky Abstractions. joelonsoftware.com

[17] Borowy, P. (2026). Mechanized Proof of the Injection Condition (Coq). github.com/opoka-cloud/hazardous-interface

Cite as: Borowy, P. (2026). "The Hazardous Interface: Why String-Based Query Protocols Are Architectural Defects." *Opoka Research*.

About Opoka

Opoka builds formally verified database infrastructure with security enforced by construction. We eliminate architectural hazards at the protocol layer.

LinkedIn: Opoka Cloud