Poster: The Risk of Insufficient Isolation of Database Transactions in Web Applications

Simon Koch*
TU Braunschweig
simon.koch@tu-bs.de

Malte Wessels*
TU Braunschweig
malte.wessels@tu-bs.de

David Klein
TU Braunschweig
david.klein@tu-bs.de

Martin Johns
TU Braunschweig
m.johns@tu-bs.de

ABSTRACT

Web applications utilizing databases for persistence frequently expose security flaws due to race conditions. The commonly accepted remedy to this problem is to envelope related database operations in transactions. Unfortunately, sole trust in transactions to isolate competing sets of database interactions is often misplaced. While the precise isolation properties of transactions depend on the configuration of the database management system (DBMS), the default configuration of common DBMS exposes transactions to anomalies that render their protection worthless.

We give a comprehensive overview on the behavior of common DBMSes with respect to transactions and show that their default settings are insufficient to provide comprehensive protection. Furthermore we conduct a preliminary study on how commonly transactions and isolation configuration adjustments are deployed across 4.222 open source PHP applications that use SQL, finding 2.789 transactions and only 418 isolation adjustments indicators.

Our findings indicate that race conditions are an underappreciated vulnerability class and adjustments are too rare to for transactions to reliably provide sufficient protection.

CCS CONCEPTS

• **Security and privacy → Web application security**; • **Information systems → Data locking**; Structured Query Language.

KEYWORDS

DBMS, Race Condition, Transaction, Web Application, Isolation Level

ACM Reference Format:
Simon Koch, Malte Wessels, David Klein, and Martin Johns. 2023. Poster: The Risk of Insufficient Isolation of Database Transactions in Web Applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3576915.3624394

*Both authors contributed equally to this research.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3624394

1 INTRODUCTION

Web applications are used to realize ever more complex use cases. To do so, they typically keep persistent state, most commonly achieved via a relational database management system (DBMS).

The execution model of Web Applications is multi-threaded by nature: at any moment in time a multitude of users may interact with the application concurrently. Thus, anomalies due to interleaving database queries, better known as *race conditions*, are a problem. Meanwhile reasoning about interleaving is difficult, especially as Web Applications are commonly tested with low user load and without interleaving in mind. Race conditions in Web Applications are known to cause financial loss if successfully exploited and have been studied by both static [2, 8] and dynamic [5–7] analysis research.

The best known defense against race condition are transactions. The PHP documentation describes the effect of a transaction as "any work carried out in a transaction, even if it is carried out in stages, is guaranteed to be applied to the database safely, and without interference from other connections, when it is committed." [1]. Consequently, all queries wrapped in a transaction can be assumed to be protected from race conditions. This assumption is wrong!

The reality of modern DBMSes is more nuanced: Naive approaches to achieve strong isolation properties, namely global locks, introduce punishing performance bottlenecks, undesirable for latency sensitive applications. For this reason, modern DBMSes provide more finely grained methods, allowing to balance between isolation properties and performance: Isolation Levels.

An Isolation Level, configurable by the developer, defines the allowed parallelism of transactions. The ANSI SQL standard defines multiple Isolation Levels. The strongest is Serializable, requiring that transactions executed concurrently shall behave the same as if they were executed consecutively. As such it is functionally similar to a global lock on the affected tables. Trivially, Serializable ensures that no race conditions can take place and is defined as the default Isolation Level by the SQL standard [4].

However, **all** modern DBMS that we tested deviate from this requirement, leaving Web Applications susceptible to race conditions even when diligently using transactions.

2 DATABASE BASED RACE CONDITIONS

Interaction with relational databases, i.e., storing and retrieving data, is typically done using SQL queries. There is no limit on the amount of concurrent interactions a database can handle and multiple interactions may touch the same entities at the same time.

```
SELECT value FROM coupons WHERE id = #1;

UPDATE coupons SET value = 0 WHERE id = #1;
```

Figure 1: Two consecutive, interdependent queries exposed to a race condition if #1 references the same id.

It is the responsibility of the DBMS to ensure that no interaction leaves the stored data unreadable for future interactions. However, logical coherence of the stored data is usually ensured via the application logic accessing the database.

Figure 1 shows an example of two queries operating on the same entity. Such query pairs in applications are candidates for race conditions. We base our example on a web shop that uses coupons for price reductions of purchases. The first query reads the *value* column from table *coupons* for the entity, e.g. with the *id* 1. The underlying code uses this first query to determine how much remaining value a coupon has. This remaining value is then applied and subsequently removed from the stored coupon. In our example the coupon has been used up. Consequently, the writing query sets the value for the coupon to 0.

Let us assume two different users trying to apply the full coupon to their purchase simultaneously. The execution order of those two pairs of queries (one pair per user) becomes vitally important. If one pair executes before the second, the remaining coupon value drops to zero and the second user does not get any price reduction. However, if, for both executions, the reading query is executed before the writing query, both checkout processes assume that the coupon still has value. Thus, both customers can claim a discount on the product leading to doubling the overall price reduction, causing a monetary loss for the shop owner. This anomalous behavior is called a Lost Update but there is more than one type of such race condition that can happen. Database literature defines six major ones [3, 4]: Dirty Write, Dirty Read, Non Repeatable Read, Lost Update, Write Skew, and Phantom.

SQL DBMS are expected to adhere to the ACID properties and should consequently be able to prevent such race conditions when used properly. The ACID properties guarantee that a transaction, i.e., an interaction with the database, is either executed completely or not at all (**A**tomicity), cannot leave the database in an inconsistent state (**C**onsistency), concurrent transactions are isolated from each (**I**solation), and finally that changes by finished transactions are stored durably (**D**urability). A transaction in this context can span any number of queries and is delimited by special start and end markers.

Based on this definition a programmer should be able to trust the DBMS to ensure, that, if multiple transactions access the same entities timing based difference are impossible, as transactions are supposed to be isolated. This assumption, however, is wrong as DBMS provide multiple different isolation levels each with its own caveats concerning isolation.

The ANSI specification defines four isolation levels (Read Uncommitted, Read Committed, Repeatable Read and Serializable) and Berenson et al. describe two further isolation levels (Cursor Stability and Snapshot Isolation) [3]. Read Uncommitted, Read Committed, Cursor Stability, and Repeatable Read are defined by the race conditions they prevent. Snapshot Isolation is a technique where a transaction works on a snapshot of the data, possibly requiring merging back into the database. Only Serializable, the strongest Isolation Level, consistently provides the protection expected [3].

3 DBMS ISOLATION LEVELS AND THEIR USAGE

We discussed Isolation Levels and how the lower levels may allow for the manifestation of race condition even if transactions are used. In this section, we examine to which degree this theoretical problem manifests itself in running PHP code. For this purpose, we design a specific test suite to test for the race condition variants. This way, we evaluate the isolation levels of different popular database management systems and their effects on race conditions using the PHP provided main APIs `PDO` and `mysqli`.

Each individual test case is designed to cause a specific race condition. In particular, we test for race conditions based on Dirty Reads, Non Repeatable Reads, Lost Updates, and Phantom Reads. Additionally, we test for two custom cases based on Phantom Reads, that we discovered during our initial manual discovery phase: Phantom Append and Phantom Delete, a programming pattern where a set is first counted and then appended to or deleted from. When applicable, we also test the same pattern with a special *SELECT ... FOR UPDATE* clause that is supposed to prevent race conditions.We also test the behavior of seemingly "nested" transaction, i.e., what happens if the application begins a new transaction while another one is currently active. The SQL standard forbids such nesting of transactions. Finally, we also check for differences between options of controlling transactions: Using the "raw" APIs to pass a query starting a transaction as string, e.g., `BEGIN TRANSACTION` vs using API functions to control transactions, such as `mysqli_begin` and PDO's function `PDO::beginTransaction`.

Our testing setup consists of one main script automatically requesting other worker scripts concurrently. Generally, every test first resets the SQL table we are testing against. Then it requests two hosted scripts at the same time. These scripts take the role of concurrent requests and are designed to trigger the race condition. The first step of every worker script is to optionally begin a new transaction either via API or raw SQL. For every failed query (including those controlling transactions) we check if a serialization error occurred as this also signals a prevented race condition. We evaluated our testbed against multiple popular DBMSs. We installed PHP 7.4 as well as MySQL, PostgreSQL, SQLite, Microsoft SQL Server (MSSQL) in Ubuntu 18.04 Docker containers and ran the aforementioned proof of concept scripts to test whether race conditions occur at a given isolation level.

Results: MySQL doesn't stop Lost Updates from happening in Repeatable Read, but is still adhering to the original SQL standard which defined Repeatable Read by the Anomalies it prevents: Dirty Read, Non Repeatable Read and Phantom Read. Lost Update was only introduced by [3] after the standard was formalized. The FOR UPDATE clause stopped any anomalies from occurring.

For PostgreSQL the FOR UPDATE clause only stopped Lost Update from occurring, but not Phantom Append.

MSSQL stopped Lost Update from occurring in Repeatable Read, which is stronger than the standard requires.

Table 1: Which non serializable patterns are prevented at given isolation levels

	MySQL				PostgreSQL			MSSQL					SQLite		
Isolation Level	RU	RC	**RR†**	S	**RC†**	SI	S	RU	**RC†**	RR	SI	S	RU	SI	**S†**
Dirty Read	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
Non Repeatable Read	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
Phantom Read	✗	✗	✓	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓
Lost Update	✗	✗	✗	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓
Lost Update w/ FOR UPDATE	✓	✓	✓	✓	✓	✓	✓	-	-	-	-	-	-	-	-
Phantom Append	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
Phantom Append w/ FOR UPDATE	✓	✓	✓	✓	✗	✗	✓	-	-	-	-	-	-	-	-
Phantom Delete	✗	✗	✗	✓	✗	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓

RU: Read Uncommitted, RC: Read Committed, RR: Repeatable Read, SI: Snapshot Isolation, S: Serializable, †: Default

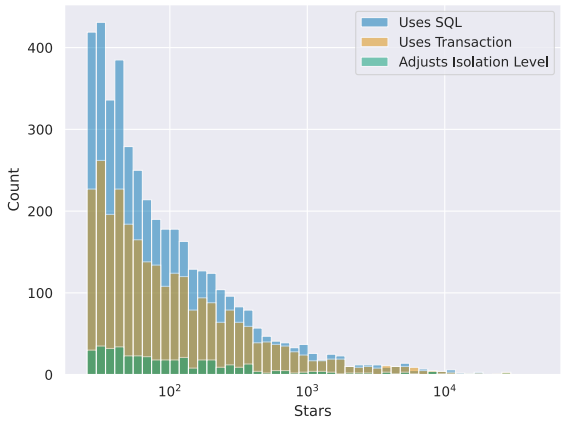

Figure 2: Public PHP GitHub projects containing keywords indicating the usage of SQL, the use of Transactions, and the adjustment of Isolation Levels.

MSSQL's Snapshot Isolation disallowed Phantoms but Phantom Append still occurred which makes sense as Snapshot Isolation makes a snapshot of the database state before the transaction starts.

We discovered different behavior for MySQL when used via the `mysqli` or the `PDO` API. The `mysqli` API implicitly commits the first transaction on a second begin, as documented in its official documentation.`PDO` on the other hand throws an Exception if a new transaction began while another one is still running, as documented in the documentationThe ANSI SQL standard states "If a ⟨start transaction statement⟩ statement is executed when an SQL-transaction is currently active, then an exception condition is raised"[4, p. 698], i.e., the `PDO` API adheres closer to the ANSI standard than MySQL itself. MySQL, PostgreSQL, and MSSQL all implicitly commit the first transaction while using "raw" SQL to control transactions and do not successfully commit either transaction when `PDO`'s transaction control is used. This difference in behavior makes it impossible to design consistent high-level database interaction across different backend DBMS as soon as nested transactions are used.

4 WERE TO GO FROM HERE?

It is obvious that the relationship between race conditions and Isolation Levels is complex, indicating a knowledge gap with significant security risk. We confirmed this suspicion by conducting a preliminary study on 30.868 open source PHP projects by searching for keywords indicating SQL or transaction usage and Isolation Level adjustments.

Only 4.222 and 2.789 projects contained keywords indicating SQL or transaction usage, respectively. An even smaller number, only 418, used isolation level adjustments related keywords.Based on those numbers we suspect a large amount of Web Applications to contain race conditions and even if they follow best practices, i.e., use transactions, to still be affected. Figure 2 plots the results in relation to the GitHub Stars of the corresponding project.

Consequently, race conditions are here to stay as a vulnerability class. We want to advocate towards developing a methodology that is able to extract transactions and identify race conditions on a large scale.

ACKNOWLEDGEMENTS

This research was funded by the European Union's Horizon 2020 grant agreement No 101019206 and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - grant agreement No 390781972.

REFERENCES

[1] 2023. *Transactions and auto-commit.* Retrieved 2023-08-18 from https://www.php.net/manual/en/pdo.transactions.php
[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *ACM International Conference on Management of Data.*
[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. *SIGMOD Rec.* (1995).
[4] ISO/IEC JTC 1/SC 21/WG 3 1994. *(ISO-ANSI Working Draft) Database Language SQL (SQL3).* Standard. International Organization for Standardization, Geneva, CH.
[5] S. Koch, T. Sauer, G. Pellegrino, and M. Johns. 2020. Raccoon: Verifying Race Conditions in Web Applications. In *ACM Symposium on Applied Computing.*
[6] R. Paleari, D. Marrone, D. Bruschi, and M. Monga. 2008. On Race Vulnerabilities in Web Applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.*
[7] T. Warszawski and P. Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *ACM International Conference on Management of Data.*
[8] Y. Zheng and X. Zhang. 2012. Static Detection of Resource Contention Problems in Server-side Scripts. In *International Conference on Software Engineering.*