On Race Vulnerabilities in Web Applications

Roberto Paleari, Davide Marrone, Danilo Bruschi, Mattia Monga

Dipartimento di Informatica e Comunicazione,
Università degli Studi di Milano,
Milano, Italy
{roberto, davide, bruschi, monga}@security.dico.unimi.it

Abstract A web programmer often conceives its application as a sequential entity, thus neglecting the parallel nature of the underlying execution environment. In this environment, multiple instances of the same sequential code can be concurrently executed. From such unexpected parallel execution of intended sequential code, some unforeseen interactions could arise that may alter the original semantic of the application as it was intended by the programmer. Such interactions are usually known as *race conditions*.
In this paper, we discuss the impact of race condition vulnerabilities on web-based applications. In particular, we focus on those race conditions that could arise because of the interaction between a web application and an underlying relational database. We introduce a dynamic detection method that, during our experiments, led to the identification of several race condition vulnerabilities even in mature open-source projects.

1 Introduction

The overwhelming majority of new computer applications are now developed adopting the web paradigm. Communications relies on the HTTP protocol, and the computation is performed via a client-server model, where client and server are respectively represented by a web browser and a web server, appropriately augmented by extension modules which enable the execution of server-side code. The applications which satisfy these requirements are generally called "*web applications*".

Originally, these applications were implemented using simple mechanisms to create dynamic web pages. One of these technologies is the Common Gateway Interface (CGI) [1], intended to provide web-based access to legacy applications by acting like a gateway between the web server and the underlying legacy application. Today, however, the most popular approach is based on extended web servers, that provide modules that implement frameworks more suitable for the development of web-based applications. Basically, the web server is able to instantiate the virtual machine needed to interpret web application programs, that are typically written in a dynamic-typed scripting language, such as PHP, Perl, Python or Ruby. Typically, web applications rely on a three-tiers architecture (web browser/web server/database manager). A very popular platform is the

so called LAMP solution stack [2]: a **L**inux machine runs an **A**pache web server which is able to control a **M**ySQL database management system through a **P**HP script.

Web applications have been reported to be subject to different kinds of attacks, many of which are specific of the web environment [3]. Such vulnerabilities could lead to the compromise or disclosure of sensitive information. According to a recent analysis [4], more than 60% of the software vulnerabilities annually reported are specific to web applications. This is mostly due to the fact that it is often quite easy to create simple web applications, thus many of them are written by developers with low programming or security skills. Nevertheless, web applications are valuable targets for attackers, because they often interface with a back-end server that handles sensitive information as credit card numbers, e-mail addresses, financial records, etc.

The most recurrent flaws in web-based programs arise from the interaction between the application and the underlying relational database used as a long-term storage medium [5], while others depend on the incorrect handling of trust relations between clients and servers [6]. All these types of vulnerabilities can be ascribed to the lack of proper input validation: some parameters that are under the direct control of a client are not properly validated.

In this paper we will introduce and discuss a new form of vulnerability which affects web applications. Such a vulnerability emerged by observing the behavior of some web applications when forced to be executed concurrently, and it turned out that they suffer the typical race conditions symptoms. Although race conditions are a well understood problem, in this work we will show that the impact of such an issue on web applications is still largely unexplored. More precisely, most of the web applications are made of many different scripts, each performing simple and well-defined tasks, easily described by sequential code. However, it is often neglected that any time a user requires the execution of a server side script, such a script becomes the body of a new thread that is executed in a multi-threaded environment. This could lead to more application scripts instances being concurrently executed. If scripts are conceived as sequential code and if they use some shared resources (e.g., a database), the parallel execution of these multiple instances could provoke races. For example, by exploiting such concurrency problems, in our experiments we have been able to bypass brute forcing protections, exploit SMS gateways, circumvent anti-flooding mechanisms and we managed to submit multiple votes on polls where each user was constrained to vote just one time.

We further deep our analysis in order to identify detection strategies for race conditions in web applications. In particular, we are interested in the detection of those race conditions that could arise because of the interaction between a web-based application and an underlying relational database. The problem of detecting and mitigating race conditions has been extensively discussed in literature, but the literature is entirely focused on applications expressly written as concurrent. The problem we face in this paper is very different as it is related to detecting synchronization problems in sequential code which can be executed

concurrently. In other words, the problem is not, as usual, to analyze the correctness of a programmed synchronization policy, but to detect whether the implicit interprocess communication contained in a piece of sequential code could lead to security failures, when multiple instances of the code are executed concurrently.

We can summarize the key contributions of this paper as follows:

- We shed light on the impact of race conditions on web-based applications. Race conditions are a well-known problem, but the effects of those on web-based programs have not been underlined, so far.
- We propose a novel technique for the detection of race conditions that arise from the interactions between a web application and an underlying database. Our proposed method has been implemented in a prototype that led to the detection of several previously unknown vulnerabilities in mature open-source applications.
- We discuss possible countermeasures to hamper exploitation attempts.

This paper is structured as follows. In Section 2 we discuss the implications of race conditions on web-based programs and the impact of this kind of synchronization issues on real-world applications. Section 3 introduces our detection method together with some implementation details and experimental results. Possible countermeasures are analyzed during Section 4, while in Section 5 we discuss related work. Finally, Section 6 briefly concludes our paper.

2 Race Conditions in Web Applications

A *race condition* occurs when different parallel processes access shared data without proper synchronization [7]. Races are difficult to spot because the human mind is not good at extensively analyze the exceedingly high number of interleavings allowed by the operating system scheduler. Thus, concurrency is a typical source of vulnerabilities [8,9,10], and one of the oldest security problems [11]. Here we will show that the same phenomenon occurs in web applications.

Often web applications are conceived as a set of scripts that query and update an underlying database. In these situations, a programmer does not usually care about concurrency issues, and considers his scripts as executed by the web server in a strictly sequential order. So, he typically ignores the intrinsic architecture of the underlying web server, which enables multithreaded executions of code. Moreover, it is often neglected that the underlying DBMS represents a shared resource that can be concurrently accessed by multiple script instances. As we will show, by exploiting these facts, a malicious user could induce an application to behave differently from what the programmer meant.

As an example, consider the PHP script fragment depicted in Figure 1. In this example, the programmer wanted to implement an e-banking money transfer procedure: the user tries to withdraw an `amount` of money; the system checks if the user has that amount on his account (lines 1–4) and, if so, it authorizes the execution of the requested operation (line 5). Finally, the system updates the user's account by withdrawing the aforementioned amount (lines 6–7). Now

```
1  $res = mysql_query("SELECT credit FROM Users WHERE id=$id");
2
3  $row = mysql_fetch_assoc($res);
4  if($row['credit'] >= $_POST['amount']) {
5    ⟨execute the requested operation⟩
6    $new_credit = $row['credit'] − $_POST['amount'];
7    $res = mysql_query("UPDATE Users SET credit=$new_credit WHERE id=$id");
8  }
```

Figure 1. An example of a vulnerable PHP script fragment.

suppose that a script instance P executes the statement at line 3, thus retrieving from the database a tuple t of the `Users` relation. The procedure in Figure 1 would be prone to races if another script instance could obtain read or write access to t before P fully executes the query at line 7. In fact, it can be easily verified that the parallel execution of multiple instances of this script fragment on the same server could result in a violation of the precondition of line 4.

Some solutions to this classical *test & set* problem may be available, however here the main issue is that *a typical programmer does not conceive his web application as a multi-threaded or multi-process entity.*

In this paper we will focus on the detection of race condition vulnerabilities in PHP applications. However, our results are not language-dependent, and can easily be extended to other platforms, such as Perl, Python, and so on. Moreover, we also limit our analysis to the race conditions that could arise from the interactions between a web application and an underlying DBMS. It is worth noting that race conditions could derive from unmanaged access to any shared resource: a database is only an example of such a resource, even if probably the most common one.

Although not every race condition has necessarily security-relevant consequences, in our experiments we have been able to found a significant number of concurrency issues, so the overall probability of the security relevance of at least *one* of these defects is still significantly high.

We would also stress that the solution to these kind of problems cannot be delegated entirely to the DBMS implementors. Usually, even simple DBMSs do provide proper synchronization features that allow programmers to handle concurrency problems (e.g., locking statements, ACID transactions, ...) and they actually guarantee the atomic execution of each submitted query (or each submitted transaction). However, DBMSs cannot automatically recognize when a sequence of queries should be executed atomically, because this heavily depends on the application's logic. Thus, is up to programmers to properly use database-level synchronization primitives in order to avoid concurrency problems in their applications.

2.1 Case Studies

In order to verify the impact of race condition vulnerabilities on real-world web-based applications, we tried to exploit two remote closed-source commercial systems, having only access to their external interfaces. The first application is managed by an Internet service provider, while the second one belongs to a telecommunication provider. For obvious reasons, the names of the corporates involved will not be disclosed. Both applications are designed to permit users to send SMS messages through a web interface, allowing only a limited number of SMS per user, per day. For both applications, our conjecture was that, when an authenticated user tries to send a message, the application checks his account information from the database. We imagined that the program first sent the message and finally updated the user's account. This behavior is very similar to the bank example reported in Figure 1. Then, we tried to exploit the remote applications. In the first case, we sent 11 parallel SMS requests. The remote service was supposed to accept only 10 requests, but we received 11 SMS messages on our mobile phone. In the second case, we sent 10 parallel requests. The application was supposed to discard all but one request, so we were quite surprised when we received all the 10 SMS messages.

This simple experiment leads us to believe that a consistent number of commercial (and maybe critical) real-world applications are vulnerable to similar attacks.

We have also tested mature open-source applications (e.g., phpBB3 [12] and Joomla! [13]): not even a *single* application from those we analyzed was found to be free from concurrency problems. Even if many of these defects cannot lead to compromise the application's logic, some of them can actually allow a malicious user to violate the security properties of the web-based application. Nevertheless, exploiting race condition-based vulnerabilities requires some knowledge about the application's logic and thus their exploitation is surely more difficult with respect to other categories of web vulnerabilities, such as cross site scripting (XSS [6]) or SQL injection (SQLI [5]). Despite these issues, we have been able to alter the original semantics of *every* real-world application we have analyzed during our experiments.

3 Detecting Race Conditions in LAMP-like Web Applications

In this section, we propose a method based on dynamic analysis for the detection of race conditions in LAMP-like web applications. The idea is to build a system which supports a programmer during the development of a web application, and which is able to automatically locate suspicious query sequences. Such an approach has been implemented in an experimental prototype.

We focused on those race conditions that arise from the interactions between an application and the underlying SQL-enabled relational database. Moreover, we are interested in the detection of those issues that could result from the

execution of multiple instances of the *same* web application script. We leave the detection of inter-module race conditions for future work.

Our detection strategy is formed by the following components:

1. a *SQL-query logger*, which monitors a concrete execution of the web application to be analyzed and logs each query that the application submits to the DBMS;
2. an *off-line analyzer*, which examines the log files that have been produced by the SQL-query logger and detects the potential dangerous queries. Such a component is realized by two modules: the first one searches the log files for query interdependencies that could be considered as a symptom of the presence of a race condition; the second module refines the results obtained so far, by removing query pairs that are guaranteed to be race-free.

3.1 SQL-query logger

There are many different methods that can be employed in order to log database queries: we could for example intercept them at the DBMS level, or we could modify the module used by the application interpreter to interact with the underlying database; alternatively, we could intercept SQL queries at the application level, by hooking database-related functions. As we discuss in more detail in a following section, our current prototype implements the latter approach. Thus, at runtime, each time the application invokes a database-related function to submit a SQL statement to the DBMS, our logger module intercepts the query string. Then, each query that has been intercepted is recorded into a text file for the subsequent off-line analysis.

3.2 Off-line analyzer: basic approach

Once database queries have been collected, the resulting log files are examined by our query analyzer. The idea behind our method is to exploit query interdependencies so that likely race conditions can be detected. More precisely, let $q = \{s_1, s_2, \ldots, s_n\}$ be a query, where s_i denotes the schema objects (attributes or relations) referred to by q. We consider a schema object to be *used* by a query when its value is read. An attribute is *defined* by a query when it is altered by the execution of such a statement. Instead, we consider a relation to be *defined* by a query if it modifies the total number of tuples in that relation. As an example, a **DELETE** statement *defines* the relation that appears in its **FROM** clause, while it *uses* every schema object that appears in its **WHERE** clause.

Given a query q, we define *use(q)* and *def(q)* as the sets of schema objects that are respectively used and defined by q. Thus, we can formalize the notion of *interdependence* with the following definition:

Definition 1. *Let (p, q) be a pair of SQL queries. Then, (p, q) are said to be* interdependent *if $use(p) \cap def(q) \neq \emptyset$.*

Input: $Q = \{q_i, i = 1, 2, \ldots, n\}$, a list of SQL queries.
Output: $R = \{(p, q), p \in Q \wedge q \in Q\}$, a list of paired SQL queries that suggest possible race conditions.

```
R = ∅
for i = 1, 2, ..., n do
    q = Q[i]
    D = def(q)
    for j = i − 1, i − 2, ..., 1 do
        p = Q[j]
        U = use(p)
        if D ∩ U then
            R = R ∪ {(p, q)}
```

Figure 2. Pseudo-code for a simplified version of the detection algorithm.

Our observation is that interdependent queries could give rise to race conditions. Thus, our detection strategy consists in determining a set containing every pair of interdependent queries.

Definition 2. *Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of SQL queries. We define a total ordering relation $<$ on its elements, such that $\forall q_i, q_j \in Q, q_i < q_j$ if and only if $i < j$, i.e., q_i appears* before *q_j in the query log.*

Definition 3. *The set R of* interdependent query pairs *is defined as:*

$$R = \{(q_i, q_j) \in Q \times Q \mid (q_i < q_j) \wedge (use(q_i) \cap def(q_j) \neq \emptyset)\}.$$

The algorithm reported in Figure 2 formalizes these notions. The algorithm receives as input a list of SQL statements, gathered dynamically by the query logger and outputs a set of interdependent SQL query pairs (p, q). From each of these SQL query pairs, a race condition could arise.

3.3 Off-line analyzer: further heuristics

Some of the query pairs collected with the approach sketched above may represent false positives. Thus, we developed a further module to remove those pairs that are guaranteed to be race-free. Such a module is based on the following heuristics.

WHERE clauses A significant source of false positives are interdependent queries whose relative **WHERE** clauses always identify disjoint sets of rows. As an example, consider the SQL queries reported in Figure 3: here the application extracts from the Sessions relation the user IDs not yet expired; afterwards, the application removes stale sessions from the database. Apparently, a race is possible between the two queries, because the first one uses the Sessions relation

```
SELECT user_id
FROM Sessions
WHERE expiry_time >= 1195745465;

DELETE FROM Sessions
WHERE expiry_time < 1195745465;
```

Figure 3. An example of two conflicting SQL queries with disjoint **WHERE** clauses.

(as well as the user_id attribute) while the second statement defines it. This is however a false positive, because the intersection of the sets of rows selected by the two statements always corresponds to the empty set. To address this problem we need a method that allow us to assert when two **WHERE** clauses identify disjoint sets of rows. In such a situation, no race condition could occur between the two queries, even if they are interdependent. In the following discussion, we assume that two queries q_1, q_2 share the same **FROM** clause f but have (possibly) different **WHERE** clauses w_1, w_2.

A viable approach is to exploit the possibility of dynamically querying the DBMS. Every time we need to assert the disjunction between the sets of rows identified by w_1 and w_2 we can build the statement:

$$\textbf{SELECT} * \textbf{FROM } f \textbf{ WHERE } w_1 \textbf{ AND } w_2$$

If the set of rows returned by such a statement is not empty, then we can assert that there can be a race between queries q_1 and q_2. It is worth nothing that if an empty set is returned, then we can only state that no race can occur in the *current* database instance (i.e. tuples currently contained in each database relation), but we cannot be sure that no race could *ever* happen.

An alternative approach consists of employing a decision procedure to assert if the sets of rows identified by w_1 and w_2 are actually disjoint. Such a method has the obvious advantage to be able to reason about *any* possible database instance, and not only about the current one, thus overcoming the major drawback of the previous approach. However, such a method would also introduce a significant overhead due to the use of an external constraint solver. Moreover, it is important to note that the constraint solver would probably not be able to handle some particular SQL constructs, such as **LIKE** expressions or nested queries. In these situations, the constraint solver would have to behave conservatively, thus reporting that the analyzed queries are not guaranteed to be independent. Nevertheless, in many practical situations this method is still effective.

Note that the constraint solver-based approach and the dynamic query approach are complementary rather than alternative: these two methods can be employed together in order to combine the efficiency of direct DBMS queries with the conservativism of the constraint solver. Every time a race condition is detected, the DBMS can be dynamically queried in order to verify if, in the current database instance, the sets of rows selected by the two **WHERE** clauses are not

disjoint. If these sets of rows turn out to be disjoint in the current database instance, then we can fall back to the less efficient constraint solver-based method, in order to obtain a sound answer.

Attribute-relation bindings Another significant source of false positives is due to the fact that we cannot always accurately deduce the relation an attribute belongs to, by only observing a single SQL query statement. Consider the query **SELECT** a_1, a_2 **FROM** T_1, T_2. The a_1 attribute could belong either to the T_1 relation or to the T_2 relation. The only thing we can do is to conservatively assume that each attribute could belong to any relation used by the analyzed query. Clearly, this could introduce a number of false positives during race detection. To overcome this limitation, in these cases we allow our race detector to actively query the application database to determine to which relation attributes belong. In the above **SELECT** statement, our race detector would actively query the underlying DBMS to determine the attributes of the T_1 and T_2 relations, in order to discover which one contains a_1 or a_2.

Annotations Finally, it is worth nothing that the algorithm presented above does not take into account any explicit synchronization attempt. We discuss this particular design choice during Section 3.5. The main consequence of such a limitation is that our detector will report a race condition even when a solution has been coded around it. Of course, such a behavior would seriously limit the employment of our proposed detection method in the development cycle of real-world applications. The accurate detection of every possible synchronization attempt, without relying on the knowledge of the particular set of synchronization primitives being used, is a complex task. In particular, in our situation this task becomes completely not feasible because we are only observing the interactions between the application and the database, without taking into account any information that could be extracted from the web application's code. For these reasons, we allow the programmer to explicitly specify that the race condition between a pair of SQL queries has already been fixed and should not be reported anymore by including appropriate annotations into those queries. Every annotation starts with the `"#!"` prefix. The `'#'` character indicates that the current line contains a comment[1], so that our annotations will not be processed by the underlying DBMS. The `'!'` character allows our race detector to discern annotations from normal comments. We support two different annotation types:

`TAG` <***name***> an annotation of this type allows the programmer to unambiguously define a name for a particular SQL query;

`SAFE` <***name***> this annotation type specifies that a race condition between this query and the query with name <*name*> should not be reported.

[1] This is true for MySQL. Should this assumption not be true, it is only a matter of changing the comment character being used.

```
#! TAG get_all_ids
SELECT user_id
FROM Users;

#! SAFE get_all_ids
DELETE FROM Users
WHERE user_id = 10;
```

Figure 4. An example of SQL queries annotated for suppressing race reports.

As an example, consider the queries reported in Figure 4. The programmer has assigned to the **SELECT** statement the name get_all_ids by using the annotation TAG. Then, the report of every race condition that could occur between the first query and the second one has been suppressed with a SAFE annotation. With such annotations, a programmer can easily test his web application with our detector module plugged in, fix the race conditions that are detected and then annotate the concerning queries so that the same races will not be reported again.

3.4 Implementation

We have implemented our detection method in a prototype that handles PHP applications and assumes the MySQL DBMS as their back-end.

The implementation of the query logger module consists of a PHP wrapper procedure around the **mysql_query**() function, so the only preliminary operation required to analyze a web application consists in replacing every call to **mysql_query**() with a call to our **mysql_query_wrapper**() function. Many web applications include a class that provides methods for submitting queries to the underlying database, so abstracting the caller from the particular DBMS being used. Thus, in order to integrate the query logger module, only a very limited number of these methods needs to be modified. Notice that even this operation is made completely automatic by a simple script. Our wrapper function logs into a text file every query that has been submitted to the DBMS together with some meta-information, such as the name of the script that issued that query and a dump of the interpreter's call stack. When a race is detected, such meta-information could help the programmer to easily locate the problem.

After the queries generated by a web application have been logged, the log files are sent to our query analyzer module in order to spot possible race conditions. Our query analyzer module consists in roughly 2000 lines of Python code and it implements the detection model discussed in Section 3.2. Our current prototype only lacks of the constraint solver-based method for determining if the sets of rows identified by two database queries are guaranteed to be disjoint. For parsing MySQL statements, the query analyzer leverages the `DBIx::MyParsePP` PERL module.

3.5 Discussion

The proposed detection algorithm has still some limitations, that can be summarized in the following points:

- Our approach is *completely dynamic*, so it can only reason about a *specific* execution path, i.e. the one that has been covered during the observed execution.
- We have no information about the application's semantics other than the query statements submitted to the DBMS. For example, we do not take into account how data retrieved from the database is manipulated by the application.
- Our detection algorithm does not take into account any synchronization method that the application could adopt in order to avoid concurrency problems.

The first limitation could only be overcomed with the application of static or hybrid analysis techniques over the program's source code: by using static or hybrid methods we would be able to reason about the *whole* application rather than a single execution path. Unfortunately, the application of static program analysis methods to an interpreted, object-oriented and dynamic-typed scripting language like PHP is far than easy and it would require to deal with very hard problems, as mentioned in [14]. For example, the analysis of PHP applications requires to perform points-to and alias analyses, that are, in general, undecidable problems [15]. The use of program analysis techniques would also allow us to obtain more information about an application's semantics, thus overcoming our second limitation. We leave such improvements for future work.

The last limitation of our detection algorithm concerns the lack of support for synchronization primitives. Rather than a real limitation, this is an explicit design choice. First, at the application level, to the best of our knowledge, PHP does not provide portable synchronization primitives that are suitable for our needs. For example, PHP supports the **flock**() function that implements a portable file locking mechanism, that can be used for synchronization and mutual exclusion purposes. However, as stated in the PHP manual [16], on some operating systems **flock**() is implemented at the process level, and, on multi-threaded web servers such as Apache, multiple PHP requests can be executed as multiple threads of the *same* process, so making **flock**() completely ineffective. Moreover, **flock**() blocks the caller until the file lock is released unless the `LOCK_NB` flag is specified. However, this option is not currently supported on Windows systems. PHP does provide wrappers for the System V IPC functions, but this feature is not enabled by default and is not available at all on Windows platforms. Second, at the database level, the available synchronization primitives are highly DBMS-dependent and often too coarse grained. As an example, MySQL, probably the most widely used open source database, supports **LOCK** and **UNLOCK** statements that provide a relation-level locking mechanism. However, such a granularity is often too coarse to be adopted in heavy-loaded web applications. MySQL also supports ACID transactions with row-level locking, but this feature is not

Application	Category	Queries	Time	FP	TP
Joomla! 1.5RC4	CMS	4086	90.92 s	0	55 (2)
phpBB 3.0.0	forum	2236	43.09 s	0	35 (4)
WordPress 2.3.2	blog/CMS	3638	47.04 s	0	47 (4)
Zen Cart 1.3.8a	shopping cart	35194	1622.39 s	0	46 (1)

Table 1. Evaluation of the detection method. FP: False Positives; TP: True Positives (security relevant true positives are reported in brackets). Note that these results have been obtained without using the annotations supported by our system.

available when using `MyISAM`, the default storage engine. A more suitable mechanism is the GET_LOCK() function[17]: it can be used to simulate record locks by creating named locks. If a name has been locked by one client, GET_LOCK() blocks any request by another client for a lock with the same name. This allows clients that agree on a given lock name to use the name to perform *cooperative* advisory locking. Locks maybe released by calling RELEASE_LOCK().

While there are suitable solutions, these can be used only by programmers that are conscious of the concurrency issue and they require them to code explicitly a synchronization policy. Moreover, the penalty due to the use of synchronization constructs is not always acceptable, because it could drastically reduce the performances of the web application. Thus, we found that synchronization primitives are rarely used in web applications and in our experiments the lack of support for them has not raised the false positive rate. Moreover, the few synchronization attempts we found at the PHP level, have actually been made completely ineffective by the underlying storage engine.

3.6 Evaluation

To prove the effectiveness of our approach in detecting vulnerabilities, we ran our prototype tool on some real word open-source web applications. Of course, the main problem in evaluating our prototype (as with any other dynamic analysis tool) concerns gathering relevant execution traces: the ability of our approach to detect previously unknown race conditions heavily depends on the path coverage rate obtained during the query logging phase. In our experiments, we tried to stimulate the web applications being analyzed as if it was used by a typical user. For example, with forum applications we tried to login by supplying both correct and wrong credentials, we added new users, read some topics, created new topics and polls, sent instant messages to other users, and so on.

We ran our detector on a Linux machine with a dual-core 2.0 GHz Pentium processor and 1GB RAM. In Table 1 we summarize some of the results we obtained during our experiments. The time required to analyze the application's log file of queries is very large, but more than 95% of the whole execution time is spent while parsing SQL statements. Such an overhead is primarily due to the inter-process interactions between our Python detector and the external Perl

SQL parser. The runtime overhead for logging SQL queries is negligible and not reported in Table 1. As we already discussed during previous sections, not every race condition we found was actually security relevant. However, we believe the number of security relevant races we found together with the absence of false positives prove the effectiveness of our detection method.

We can briefly summarize some of the vulnerabilities we run into in the following categories:

multiple users Almost every application we analyzed was found to be vulnerable to a race condition on user uniqueness: a malicious user could register multiple accounts with the same username, thus bypassing application's checks. Of course, the security impact of this vulnerability is highly application-dependent. As an example, it could allow an attacker to take advantage multiple times of a one-time bonus granted by a unique token.

brute forcing Some applications (e.g., phpBB3), in order to prevent brute forcing attacks, check if the user that is trying to log in has already performed too many login attempts. The procedure used to perform this operation contains a race condition vulnerability that could allow a malicious user to bypass the application's attempts to limit brute forcing password attacks. Depending on the application's logic, such a vulnerability could allow an attacker to perform just a limited number of additional attempts (e.g., when the application ensures that `tries` $\leq$ `MAX_TRIES`), or to completely circumvent application's checks (e.g., when a brute force attack is reported only if `tries` = `MAX_TRIES`).

multiple poll votes Web forums and CMSs often implement polls. The applications try to assure that each user does not submit multiple votes to the same poll, but every program was found to be subject to a race that allows a user to vote multiple times by submitting parallel vote requests.

topic flooding phpBB3 and WordPress include an anti-flooding feature that forces a user that has just submitted a message to wait a couple of seconds before writing another post. Unfortunately, even this control can be easily circumvented by an attacker because of a synchronization issue.

It worths noting that in the web applications we analyzed we have met very few synchronization attempts. Unfortunately, even in these cases we have been able to find concurrency problems. This confirms that programmers are not aware of the actual impact of race conditions on web-based applications.

4 Countermeasures

Before concluding our paper, in this section we introduce some countermeasures a programmer could employ in order to hamper exploitation attempts.

Probably, the most obvious solution is to completely prevent any concurrency issue by forcing the web server to serve just one client request at a time. Unfortunately, such an approach is typically too drastic and not applicable at all, as it seriously limits the overall efficiency of the whole web-based application.

Another approach consists in employing some application-level or database-level synchronization primitives in order to explicitly serialize the accesses to an application's critical regions. As we already discussed, many of these primitives often hide some subtle platform-specific details that a programmer should accurately consider before deploying his web-based application; otherwise just the migration of the application towards a different server could alter his behavior and introduce new vulnerabilities. During Section 3.5 we pointed out the limitations of PHP/MySQL environments. Obviously, different frameworks could surely offer more efficient and fine-grained locking statements (e.g., row-level database relation locking), but this typically comes at the cost of less efficiency or more resource-consumption.

For example, a table-level locking solution will surely be too coarse grained if applied to the code snippet reported in Figure 1: here the requested transaction could take some time to be executed, thus the application cannot be constrained to serve just a single client for all that time. In this situation, an alternative solution that does not require fine-grained locking primitives consists in moving the **UPDATE** statement just before the execution of the requested transaction, then lock the table before the first **SELECT** query and unlock it both after the **UPDATE** statement and in the *else* branch. This solution is a simple two-phase commit algorithm that requires an additional error handling procedure: the credit is immediately withdrawn from the balance and must be restored if the transaction fails.

Thus, the effectiveness and the efficiency of a synchronization solution is highly application dependent. Automatically fixing race conditions by introducing appropriate locking statements, without affecting the efficiency of the whole application, is surely a rather complex task. In fact, it would be quite simple to blindly insert locking statements around a supposed critical region, but it would be significantly harder to do so also avoiding deadlocks and without reducing the performances of the web-based application. We plan to investigate on similar automatic techniques in future work.

5 Related Work

Race conditions are probably one of the oldest software problems and their implications have extensively been discussed in literature [7]. There has been a substantial amount of research work on the detection of this kind of concurrency problem, both for debugging and for security purposes. To the best of our knowledge, this paper is the first one to focus on the implications of race conditions on web-based application, so in the present section we will discuss alternative solutions directed toward traditional (i.e. non web-based) applications.

Static analysis. Many static race detectors perform compile-time analyses over a program's source code in order to detect if a race condition could occur in *any* possible program execution [18,19]. Other approaches [20,21] modify a programming language's type system so that the resulting language is guaranteed to be

race-free. Usually, the major drawback of these tools is an high false positive rate: by reasoning over an application's source code without running it, these approaches are often forced to make some conservative assumptions about possible thread interleavings that could occur at run-time. Moreover, often static methods require a substantial amount of annotation code in order to suppress false positives.

Dynamic analysis. Dynamic methods work by instrumenting and executing a program. These tools are typically easy to use and are more accurate than static methods, as they can observe a concrete execution of the application. On the other side, they are not sound: dynamic approaches can only assert the presence of a synchronization issue on a program path that has been executed, but they cannot prove the absence of race conditions. Several methods [22,23] are based on the dynamic computation of Lamport's *happens-before* relation [24], that outputs a partial ordering on program statements. Other methods [25,26] use *lockset*-based analysis [27], that stem from the assumption that race conditions occur because a programmer forgot to protect a shared variable with an appropriate lock. Basically, each shared variable is associated with a *lockset* that contains locks held during accesses to this variable; if a lockset becomes empty, then a race condition could occur. Some approaches [28,29] have also been proposed that blend together the advantages of both these techniques. Finally, another dynamic method [30] aims to prevent the exploitation of race condition vulnerabilities on filesystem operations, by keeping track of possible interferences between the actions performed by different processes: if a filesystem operation is found to be interfering with another one, then the corresponding process is temporarily suspended.

Model checking. Model checking is a powerful formal verification technique that has also been applied to the detection of concurrency problems [31]. A model checker receives as input a simplified version of an application's source code and exhaustively explores its execution states, searching for possible violations of some asserted conditions. For example, some model checking tools have already been proposed to analyze concurrent Java programs for synchronization issues [32]. Unfortunately, the application of model checking to large software systems is still problematic. Moreover, often a significant effort is required in order to build the simplified model to be supplied to the analysis tool.

Our proposed detection strategy can surely be classified as a completely dynamic detection method. However, the web environment shows some peculiarities that lead to rather different problems than the ones discussed in the aforementioned works. In fact, currently web programmers are not aware of the implications of the lack of proper synchronization on their applications, while traditional concurrent programs are actually written with synchronization in mind. Thus, the approaches discussed above are mainly focused on analyzing the correctness of a programmed synchronization policy. Instead, our work aims to make explicit

the implicit interactions among different instances of a sequential code that can be executed concurrently.

6 Conclusions

In this paper we discussed race conditions in web applications. Race conditions are a well-known security problem, but their impact on web-based programs has not been explored sufficiently. We showed that, by exploiting unforeseen interactions between different script instances, a malicious user could be able to alter the behavior of a web application as it was intended by the programmer. We further deep our analysis in order to investigate concurrency issues that could arise because of the interactions between different instances of the same application script when accessing to a SQL-enabled relational database. We proposed a dynamic detection method that allowed us to locate several security-relevant race conditions even in mature and well-tested web applications.

In the future, we plan to refine our detection method by considering how instances of different web application scripts could affect each others. Moreover, we will improve our detection strategy by extracting some additional information from the application through the employment of more sophisticated program analysis techniques, thus overcoming some of the limitations discussed during Section 3.5. Finally, as web programmers will get aware about concurrency problems, they will surely start to try to solve these issues by using some synchronization primitives. So, we plan to improve our analyses to include support for validating their use.

7 Acknowledgments

The authors would like to thank Lorenzo Cavallaro and the anonymous reviewers for their useful suggestions and comments on this paper.

References

1. NCSA Software Development Group: The Common Gateway Interface. (1995)
2. Kunze, M.: Let there be light. LAMP: Freeware web publishing system with database support. c't **12** (1998) 230
3. Cova, M., Felmetsger, V., Vigna, G.: Vulnerability Analysis of Web Applications. In Baresi, L., Dinitto, E., eds.: Testing and Analysis of Web Services. Springer (2007)
4. Symantec Inc.: Symantec internet security threat report: Volume XII. Technical report, Symantec Inc. (sep 2007)
5. Halfond, W.G., Viegas, J., Orso, A.: A Classification of SQL-Injection Attacks and Countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA (March 2006)
6. CERT: Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests (2002)

7. Netzer, R.H.B., Miller, B.P.: What are Race Conditions?: Some Issues and Formalizations. ACM Letters on Programming Languages and Systems **1**(1) (1992) 74–88
8. Dean, D., Hu, A.J.: Fixing races for fun and profit: How to use access(2). In: Proceedings of the 13th conference on USENIX Security Symposium. (2004)
9. Borisov, N., Johnson, R., Sastry, N., Wagner, D.: Fixing races for fun and profit: How to abuse atime. In: Proceedings of the 14th conference on USENIX Security Symposium. (2005)
10. Bishop, M., Dilger, M.: Checking for race conditions in file accesses. Computing Systems **2**(2) (1996) 131–152
11. Abbott, R.P., Chin, J.S., Donnelley, J.E., Konigsford, W.L., Tokubo, S., Webb, D.A.: Security analysis and enhancements of computer operating systems
12. phpBB Group: phpBB
13. Joomla! Core Team: Joomla!
14. Jovanovic, N.: Web Application Security. PhD thesis, Technical University of Vienna (July 2007)
15. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01). (2001)
16. PHP Documentation Group: PHP Manual. [Online; accessed 23-November-2007].
17. MySQL AB: MySQL Reference Manual. Online at http://dev.mysql.com/doc/refman/5.0.
18. Sterling, N.: WARLOCK - A static data race analysis tool. In: Proceedings of the Usenix Winter 1993 Technical Conference. (1993) 97–106
19. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. (2003) 237–252
20. Flanagan, C., Freund, S.N.: Type-based race detection for Java. ACM SIGPLAN Notices **35**(5) (2000) 219–232
21. Boyapati, C., Rinard, M.: A parameterized type system for race-free java programs. In: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications. (2001) 56–69
22. Dinning, A., Schonberg, E.: An empirical comparison of monitoring algorithms for access anomaly detection. In: Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming. (1990) 1–10
23. Ronsse, M., Bosschere, K.D.: RecPlay: A fully integrated practical record/replay system. ACM Transactions Computer Systems **17**(2) (1999) 133–152
24. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7) (July 1978) 558–565
25. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. ACM SIGPLAN Notices **37**(5) (May 2002) 258–269
26. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in Cilk programs that use locks. In: Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures. (1998) 298–309
27. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15**(4) (1997) 391–411
28. Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient detection of data race conditions via adaptive tracking. Technical report, Microsoft Research (April 2005)

29. Pozniansky, E., Schuster, A.: Efficient on-the-fly data race detection in multi-threaded C++ programs. ACM SIGPLAN Notices **38**(10) (October 2003) 179–190
30. Tsyrklevich, E., Yee, B.: Dynamic detection and prevention of race conditions in file accesses. In: Proceedings of the 12th USENIX Security Symposium. (August 2003)
31. Chamillard, A.T., Clarke, L.A., Avrunin, G.S.: An empirical comparison of static concurrency analysis techniques (July 23 1996)
32. Visser, W., Havelund, K., Brat, G., Park, S.J.: Model checking programs. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering. (September 2000)