Raccoon: Automated Verification of Guarded Race Conditions in Web Applications

Simon Koch
Institute for Application Security (TU Braunschweig)
Braunschweig, Germany
simon.koch@tu-braunschweig.de

Tim Sauer
Institute for Application Security (TU Braunschweig)
Braunschweig, Germany
tim.sauer@tu-braunschweig.de

Martin Johns
Institute for Application Security (TU Braunschweig)
Braunschweig, Germany
m.johns@tu-braunschweig.de

Giancarlo Pellegrino
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
gpellegrino@cispa.saarland

Abstract

Web applications are distributed, asynchronous applications that can span multiple concurrent processes. They are intended to be used by a large amount of users at the same time. As concurrent applications, web applications have to account for *race conditions* that may occur when database access happens concurrently. Unlike vulnerability classes, such as XSS or SQL Injection, dbms based race condition flaws have received little attention even though their impact is potentially severe. In this paper, we present Raccoon, an automated approach to detect and verify race condition vulnerabilities in web application. Raccoon identifies potential race conditions through interleaving execution of user traces while tightly monitoring the resulting database activity. Based on our methodology we create a proof of concept implementation. We test four different web applications and ten use cases and discover six race conditions with security implications. Raccoon requires neither security expertise nor knowledge about implementation or database layout, while only reporting vulnerabilities, in which the tool was able to successfully replicate a practical attack. Thus, Raccoon complements previous approaches that did not verify detected possible vulnerabilities.

CCS Concepts

• **Security and privacy → Web application security**; *Information accountability and usage control.*

Keywords

Race Conditions, Web Application Security Testing

ACM Reference Format:
Simon Koch, Tim Sauer, Martin Johns, and Giancarlo Pellegrino. 2020. Raccoon: Automated Verification of Guarded Race Conditions in Web Applications. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3341105.3373855

1 Introduction

The general intention of Web applications is to accommodate a potentially large number of concurrent requests. This requirement comes with a significant challenge: maintaining shared data consistently across all processes. Such data can range from mundane things such as the amount of visits over the last hour to business critical information such as the remaining amount of a gift voucher.

To ensure consistency of data, e.g. ensure that a voucher is not used for more money than it is worth, web application developers deploy guards in form of conditional statements. Such a guard ensures that an action, e.g. a value reduction, can only occur if the current state of the data allows for it.

However, interleavings of concurrent read and write operations can circumvent that guard and leave data in an inconsistent state. These inconsistencies can be effectively exploited by attackers to perform a wide range of attacks, e.g., double spending vouchers in Instacart[1], and duplicate money transfers between gift cards in Starbucks[2]—to mention two past instances in popular websites.

As opposed to popular web vulnerabilities such as XSS and SQL injection, the detection of guarded race condition (GRC) vulnerabilities in web applications has been marginally addressed. Prior work has mainly focused on the problem of reasoning on SQL logs to detect potential vulnerabilities [1, 2].

However, detection of potential GRC is not enough due to possible protective web application logic and the underlying database structure that can prevent a detected potential GRC from manifesting. Neither the web application logic nor the complete database structure is entirely accounted for in current detection approaches and thus a detected vulnerability is possibly not exploitable after all. A usable methodology has to minimize occurrences of detected possible but actually not exploitable vulnerability candidates. The current state-of-the-art in GRC detection leaves it to the tester to manually verify that the suspected vulnerability actually manifests itself in an exploitable vulnerability.

For well-explored vulnerability classes, such as XSS or SQL injection, this process is manageable, as it is usually formulaic and independent of the semantics of the application. In contrast, manual verification of a GRC is notoriously difficult. It requires good understanding of the web application's execution model and how race conditions affect it. While the developer of the application

[1]See https://hackerone.com/reports/157996
[2]See https://sakurity.com/blog/2015/05/21/starbucks.html

most likely has the latter, they probably lack the required security expertise. A security expert should know the specifics of GRCs but rarely knows the fine-grained details of the tested application's functionality.

We now take a step forward and present a methodology that enables developers and security experts alike to automatically detect and verify GRCs in web applications. We follow up with implementing this methodology as the novel **rac**e **co**ndition verificati**on** tool Raccoon to show the practicality of the methodology.

To the best of our knowledge, Raccoon is the first tool providing a comprehensive and automated approach concerning GRCs covering all the necessary steps from detection up to verification. We report on the effectiveness of Raccoon by assessing four real-size web applications and ten use cases. We discover six GRCs with security implications using Raccoon.

Our Contributions:

- We isolate and define the programmatically verifiable guarded race conditions (GRC)
- We present the first comprehensive automated security testing methodology to detect and verify GRCs in web applications
- We implement Raccoon, a prototype implementation of our methodology[3]
- We apply Raccoon against four real-size web applications and discover six GRCs all of which have security implications

We introduce our methodology and results as followed: First we discuss the related work (Sec. 2). We continue by going into the background details of GRC and state the challenges that we addressed (Sec. 3). We follow up by stating our general approach on solving the challenges (Sec. 4) and describe how we actually implemented it (Sec. 5). We then report on the application of our implementation in a real world scenario and present the results (Sec. 6). Finally, we give a prospect of future work (Sec. 7), and a conclusion summarizing our contributions and results (Sec. 8).

2 Related Work

Related work falls into two main areas: dynamic web application testing and database based race condition detection in web applications.

Race Condition Detection in Web Applications

Race condition vulnerabilities in web applications did not yet get as much attention by the scientific community as other, easier to detect security vulnerabilities and are not listed in the most recent OWASP Top 10 report [3] despite the fact that exploitation can lead to extensive financial damages [4, 5].

Bailis et al. empirically investigated ORM-backed applications and the corresponding feral database concurrency control in Ruby on Rails applications [6]. However, this only covers the niche of possible web applications that are Ruby based. Additionally, they focused on database inconsistencies (e.g. foreign key mismatches). They did not state security concerns as an explicit focus.

Warszawski et al. propose a method for detecting race conditions using extensive modeling based on collected trace data. They generate an *abstract history graph* corresponding to a given SQL query trace and use a graph traversal algorithm to find vulnerabilities in and between requests [1]. They verified their results by hand using either brute force or inserting a small network delay in the communication between web application and database server. We extend their idea of delaying the communication by focusing the delay on only the queries of interest and using this for an automated testing tool. Additionally, Warszawski et al. reported an GRC for voucher usage in the eCommerce web application OpenCart and, thus, provided a partial ground truth for the evaluation of Raccoon.

Zheng et al. developed a context- and path-sensitive interprocedual static analysis to detect atomicity violations on shared external resources in PHP and are able to infer potential GRCs [7]. They conducted their verification of detected vulnerabilities by hand. They report on a GRC in the coupon usage in OpenCart and, thus, provided a partial ground truth for the evaluation of Raccoon.

Paleari et al. were the first to report dbms based GRCs in web applications a valid security concern in the scientific literature. In their work they propose an algorithm based on SQL query traces to detect GRCs. They introduce the notion of interdependence between SQL queries as a way to detect possible GRCs and define an algorithm to detect interdependent SQL queries [2]. Finally, they manually verified their results. Their algorithm represents the current GRCs candidate generation module of Raccoon. We, thus, extend the approach of Paleari et al. by embedding it into an automated verification facility.

The listed work is mainly concerned with a methodology for the *detection* of possible GRCs. They perform the remaining work, i.e., data gathering, testing, and verification manually. Consequently, our methodology that only requires a tester to provide user actions, a web application and allows plugging in arbitrary detection algorithms is a novel contribution to the field of race condition vulnerabilities in web applications.

Dynamic Web Application Testing using User Traces

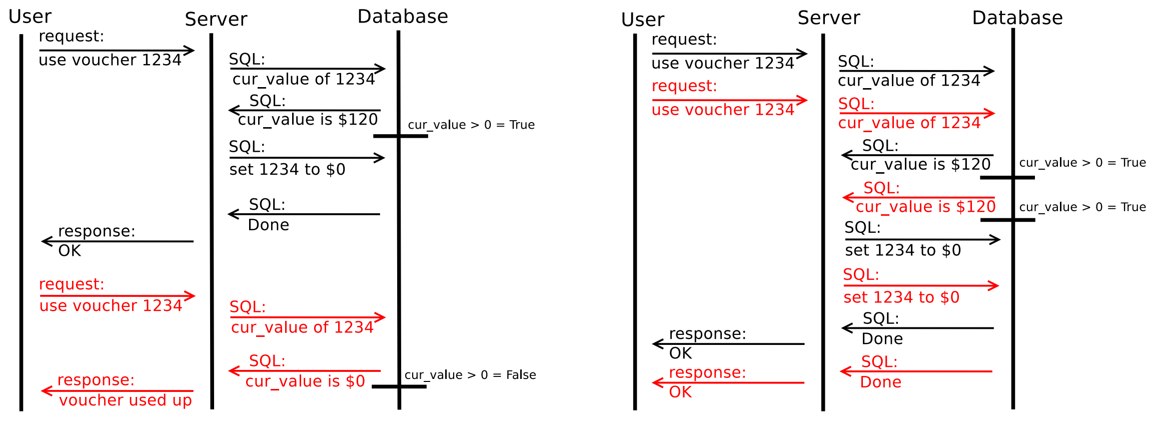
Basic web vulnerabilities can be found using automated crawling. However, crawling can only test for a small selection of possible security vulnerabilities and more complex vulnerabilities such as CSRF or logic inconsistencies are missed and a more sophisticated approach is needed. Deemon is a tool designed by Pellegrino et al. and leverages predefined user action scripts to collected dynamic execution traces, session-, and HTTP-communication. Based on the collected data they build a *deep model* of the underlying functionality to detect aCSRF vulnerabilities [8].

McAllister et al. also leverage user interactions and collect HTTP-communication to improve the testing depth for specific use cases of a given web application. Using this methodology they are able to improve the detection rate for reflected and stored XSS vulnerabilities [9].

Pellegrino et al. address logic vulnerabilities in web applications by utilizing captured network traces to identify underlying application behavior using heuristics and then to be tested using specifically generated automated tests [10].

All the listed publications and tools on automated testing share an automated approach for detection and testing of vulnerabilities with Raccoon. Additionally, leveraging user actions to improve testing depth as well as to achieve automation is an approach that Raccoon shares with the work. However, none test for GRC related vulnerabilities and, thus, Raccoon stands apart in this aspect.

[3]Source code:https://github.com/simkoc/raccoon



(a) (b)

Figure 1: (a) The process as expected by the developer: The first request reduces the voucher amount to $0 and the second request fails when reaching the check. (b) The behavior in the presence of insufficient synchronization. Both concurrent requests for the voucher usage go through without problem. The black horizontal bar shows the point where the "*if cur_value >* 0" guard is reached and it becomes obvious that, opposed to (a), both processes reach the guard with a value of $120 due to interleaving.

3 GRC in Web Applications

We start this section with an overview of the mismatch between the expected execution of a code piece by the developer and the deployment reality of web application code that leads to the vulnerability of concern. Consecutively, we follow up with a precise description of the class of race condition we want to detect and verify.

3.1 The Web's Hidden Concurrency

In the context of developing web applications, a clash exists between the dominant programming model, that creates an illusion of sequentiallity, and the execution environment of web applications, that generally is highly concurrent and hides potential side effects of simultaneous access to the database layer. In this section, we explore this general mismatch of the web's programming and execution model. Then, in the subsequent section, we show how unhandled concurrency can potentially lead to inconsistencies in the application's data.

Expectation of Synchronicity Most web applications are run on a highly concurrent execution platform. A web server is designed to handle as many concurrently incoming HTTP requests as possible. With the notable exception to the single-threaded Node.js, the majority of web servers realize this through spawning multiple workers, each handling a single incoming HTTP request. Workers operate in the same shared environment and have access to a set of shared resources, such as the server's file system and connected database(s).

These highly concurrent execution characteristics of web applications are not reflected by the dominant programming model: Most web applications are written in scripting languages, such as PHP or Python, often using application frameworks, such as Symfony[4] or Zend[5] that abstract away the program flow even more. Writing classic web code using these tools creates the illusion that the current script runs in isolation – an incoming HTTP request triggers the execution of a given web script, that is single-threaded, running from the beginning of the script until its termination, while accessing the server's storage resources sequentially in the process.

Nothing in the code suggests, that the currently running script is *not* the only code executed at this moment and during development and testing time this assumption is correct. Hence, the expectation of the execution of a piece of web application code is different to its productive execution: most web servers allow concurrent interaction with the same use case (i.e., the same code segment) at the same time.

Transparent Handling of Concurrency Server-side programming languages do not have primitives to handle concurrency transparently. They hand concurrency handling or the lack thereof to the developer. Integrity and atomicity when accessing resources has to be guaranteed by the developer using mechanisms that are not language dependent, but depend on the resource.

Most modern SQL databases provide a construct called *transaction* that allows grouping multiple queries into an atomically executed sequence. They often also consider each query to be atomic as long as the query is not otherwise included in a transaction. Consequently, errors due to concurrent query execution conflicting do not occur. However, the order of execution of concurrent queries not in transactions is not guaranteed by the database. This may create the false idea that concurrency is not a problem as reasoning about possible different sequentializations of concurrent database queries and their outcome is highly unintuitive.

The described circumstances lead to web applications that do not take care of concurrent database accesses properly. This can lead to requests to the same resource having unexpected interleavings of their read and write queries that leave the affected database entity in a valid yet unexpected state. Such phenomenons may represent exploitable behavior with security implications.

[4]https://symfony.com/
[5]https://framework.zend.com/

```
def use_voucher(cur_price, voucher_id):
    cur_value = get_voucher_value(voucher_id)
    if cur_value > 0:
       if cur_value >= cur_price:
          new_v = cur_value - cur_price
          new_price = 0
       else:
          new_v = 0
          new_price = cur_price - cur_value
       end
       update_voucher_value(new_v, voucher_id)
    return new_price
```

Figure 2: A simplified example of code using a voucher in a eCommerce paying process. The sub functions *get_voucher_value* and *update_voucher_value* both entail a *SELECT* and *UPDATE* query to the underlying database system respectively

3.2 Race Conditions in Web Applications

The execution of a program typically begins from a `main` function. However, web applications do not have a similar concept of main function and the web server calls specific functions upon a request. Consider for example Figure 1a representing the client-server-database interaction of the code show in Figure 2. It represents the execution of two requests to URL `https://server/use_voucher` providing a voucher id and the current shopping cart value. The request results in the execution of a function that retrieves the remaining amount of the referenced voucher, decreases the remaining amount of the voucher, and reduces the shopping cart price accordingly. Web servers can potentially receive thousands of such requests at the same time resulting in multiple concurrent execution of the use voucher function.

As detailed above, a typical web server receives each such concurrent request and delegates the execution and response to independent workers. Usually no means of synchronization between workers exists and concurrent access to data stored in a database eventually occurs. This is the point where problems may arise as soon as concurrent requests read and write the same resource in the database. If the processing logic or database schema does not enforce synchronization different interleavings of the concurrent reading and writings can lead to different yet still valid results in the database. Depending on the context this presents vulnerable behavior. Figure 1b presents such a context.

If the server processes the requests concurrently it might happen that all requests reach the functionality to read the voucher value before any of the other processes reach the functionality to update its value. Thus, all processes work with the full initial value and the voucher is applied multiple times, possibly in excess to its value.

This goes against the expectations of the developer as the trivial and expected requests result is that server processes the requests sequentially resulting in an appropriately reduced end price of the shopping carts and a reduction of the voucher not exceeding its initial value.

This mismatch between expectations and reality occurs as the guard protecting the application logic from overspending depends on the value of the reading query. However, the reading query for the guard assumes that there is no other code about to write to the value, which in this circumstance is false as multiple other processes are in the guarded code segment. The result is a *guarded race condition* vulnerability.

3.2.1 Guarded Race Condition: With respect to the example we define a guarded race condition (GRC) as:

(1) A non-atomic sequence of reading and writing of the same entity in a database.
(2) Triggered by a single HTTP request done multiple times concurrently
(3) The application logic guards the writing of the entity based on a decision on the return value of the reading query

Even though guarded race conditions do not represent all possible race conditions they are nontrivial and security relevant as our results in Section 6.4 show. Race condition vulnerabilities that do not fit our definition are out of scope. We consider verification of them future work.

3.3 Detection alone is not enough

Previous approaches such as from Paleari et al. [2] and Warszawski et al. [1] only addressed the detection. Either approach utilizes limited or no knowledge about the database schema and a log of the executed queries. This is not sufficient to ensure that a detected vulnerability is actually a vulnerability. The underlying database schema could provide a working protection using unique keys or triggers. Additionally, the web application logic by itself could prevent the vulnerability, e.g., by utilizing file handlers, which usually have unique access locks, effectively providing a working protection against any race condition during the time the file is open. Yet, such indirect protection is highly complex to infer and not considered in the existing approaches. Consequently, a tester has to verify the detected vulnerabilities themselves, which they currently have to do by hand as no methodology for automatic verification exists.

3.4 Verification Challenges

GRCs are notoriously difficult to verify at run-time as timing issues make testing vulnerability candidates error prone and unreliable. Those timing issues arise due to the narrow time window between queries and the random delays introduced by the transportation layer between client and server. Consequently, conducting verification tests by hand is a time-consuming and error prone activity. Consequently we need to develop a reliable automation solution.

Additionally, even assuming the existence of a reliable exploitation method we still have to solve the question on how to distinguish between resulting anomalous behavior and non-anomalous behavior. Previous work relied on visual confirmation for verification. This is not a feasible approach for an automated solution. Consequently, we also need to develop an automatic verification solution.

A final challenge in the detection of GRCs comes with the observation that a tester needs both, a thorough understanding of the use cases of a given web applications (e.g., coupon usage limits) and a thorough understanding of GRCs, to be able to successfully identify and test critical segments of a business process. Either knowledge base is limited to distinct sets of people, the developers of a given web application and security experts. However, the intersection between both sets is sparse if not disjunct. Thus, it is important to find a testing solution that eliminates one of the two prerequisites to enable and increase future GRCs testing.

4 Methodology Overview

In this section we describe our novel methodology to automatically detect and verify GRCs with minimal security understanding and input required from a potential user. We intend the methodology to be used alongside continuous integration testing. A tester may simply reuse the user traces for the already tested use cases and leave everything else to the computer running our methodology.

We separate our methodology into four distinct phases that are logically separate and are performed in succession:

(1) Setup – preparing the web application for the test process
(2) Data Gathering – gathering the data required for detection
(3) Data Analysis – evaluating the gathered data to detect GRC vulnerabilities
(4) Verification – verifying a vulnerability against the running web application

A web applications usually contains multiple use cases with their own user traces to be tested. The same setup is reusable for multiple tests. Consequently, a future user only has to execute the *Setup Phase* once on the web application. The remaining three phases (data gathering, data analysis, verification) however have to be done for each distinct use case in form of user traces. The remainder of this section is structured in the same order as the listed phases lead by a section describing the inputs required.

4.1 Inputs

Our approach requires as input a server hosting the web application as well as a user traces.

User traces: A user trace is a collection of user actions (e.g., a click, filling in a form field) that are commonly used in security testing [11]. A user trace is a step by step description of a use case (e.g., login) against the web application's GUI as displayed in the browser. Previous automated security testing solutions demonstrated the advantage of leveraging such traces for security testing [8, 11]. A tester records the user actions of a user trace and the actions can be exactly repeated without additional user input required.

Repeatable trace-driven testing: To enable reliable, side-effect free and repeatable security testing, it is important that we execute the recorded traces on a web application with a well defined state, that is the same for all relevant test runs. We solve this problem by running the web application's server-side portion in a virtual machine with the ability to take *snapshots*. Through restoring the application under test to a specific, clean snapshot before running a test, we make sure that the results of the test can be cleanly compared to the outcome of previous tests and all observed, server-side state changes between the test-runs correspond directly to the undertaken test actions.

4.2 Setup

Before we start any testing on the web application, we have to configure the web application in a way that allows us to collect SQL query logs on a request by request base. After we finish the configuration we take a *testing* snapshot, representing the running web application.

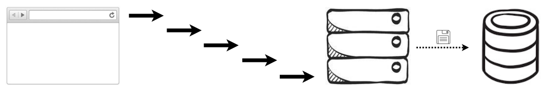


Figure 3: The user action chain is executed sequentially against the web application and the occuring sql log data is collected.

4.3 Data Gathering

We need to get a baseline of the applications default behavior if no GRC is triggered. To achieve this we take the provided user trace as well as the web application (in the *testing* state) and run the user trace consecutively multiple times against the GUI without restoring the snapshot in between. We extract the SQL query logs and store them on a request by request base.

4.4 Data Analysis

GRCs are based on requests and the interaction of the same SQL query pairs (a reading and a writing query) in the different requests. Consequently, we choose HTTP requests and the corresponding SQL query logs as the inputs for a used detection algorithm. This also matches with the already proposed detection approaches by Warszawski et al. and Paleari et al. [1, 2]. The output expected from a chosen detection algorithm is a list of tuples containing the URL of the request as well as the query pair forming a possible GRC.

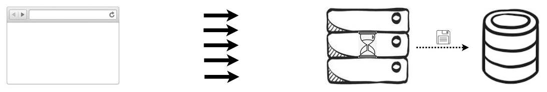


Figure 4: Running multiple concurrent user traces against the web application. An active query delay is active to force the exploitative interleaving needed to validate the GRC. The occuring sql log data is collected

4.5 Verification

We now have a set of detected potential vulnerabilities that need verification. This process requires triggering an anomaly using a detected vulnerability and finally verify or reject the vulnerability based on the gathered data.

We first describe how we *trigger the potential vulnerability* and then how we *verify* the result.

Triggering the Vulnerability: Triggering of GRCs entails running the user trace concurrently multiple times against the GUI in separate browser instances to trigger the potential vulnerability. However, as browser automation is coarse in controlling the execution times of steps and GRCs are hard to exploit reliably we cannot expect to trivially and reliably achieve the required interleaving due to the tight timing window for GRCs.

To reliably achieve interleaving we programmatically introduce a delay before the writing query of the suspected SQL query pair.

This increases the time between the reading and writing query. Thus, opening up a large time window for the interleaving to occur in contrast to the minimal time window usually encountered. This ensures that exploiting a vulnerability becomes feasible up to being unavoidable.

Due to our previous configuration the server collects SQL query logs during this process on a request by request basis. We want to stress that extending the time window only increases the likelihood of achieving interleaving but does not open up a vulnerability if none existed before (e.g., due to unique keys or file handlers).

Verification Process The verification process requires an algorithm – the oracle – that either confirms or rejects the vulnerability based on the gathered data.

After we tried to trigger a single vulnerability the oracle gets the data gathered and the potential vulnerability. The oracle then tries to verify the detected vulnerability and either outputs success or failure.

Oracle: Web applications heavily depend on their underlying database. They require the database to maintain a state across requests and constantly read and write data from the database based on the processing logic for a request. We rely on this dependence for our oracle that verifies if a detected GRC is exploitable.

Recalling our definition of GRCs, a GRC vulnerability requires a writing query whose execution is guarded by the result of the corresponding reading query. Thus, if a previous writing query changes the value to a value that triggers the guard, no further writing query is executed. Resulting in further requests not encountering the writing query in their log. This is the behavior encountered during consecutive execution. Hence, if the query is executed more often when we forced interleaving during concurrent execution we can conclude that the vulnerability is verified.

Molding this into a process leads to an oracle that counts the occurrences of the guarded writing query in both the consecutive execution and in the concurrent execution with forced interleaving. If the concurrent execution leads to a higher count our oracle considers this a verified GRC.

Splitting this into a step by step process means that the oracle takes the SQL query logs gathered during the data gathering phase, the SQL query logs gathered during the attempt on triggering the vulnerability, and the detected potential vulnerability in form of the corresponding request URL and the query pair consisting of a reading and writing query. The steps of the oracle then are:

(1) The oracle reduces the set of query logs to the logs corresponding to the request that contained the detected potential vulnerability.
(2) The oracle counts the amount of occurrences of the writing query in the subset extracted from the consecutive and the concurrent execution data.
(3) The oracle compares both counts to each other and confirms a vulnerability if the count in the concurrent execution is higher than in the data set collected during consecutive execution.

Intuitively, a higher count during the attempt to trigger the vulnerability means, that a guard that works in case of consecutive execution does not reliably work under concurrent execution and thus matches our definition of a GRC.

5 Implementation

To show that our presented methodology is feasible we implemented a prototype *Raccoon* (**rac**e **co**ndition verificati**on**). As already indicated in the previous section we intended Raccoon to be integrate into existing testing procedures. A tester may simply use the user traces for the continuous integration tests already existing and hand them over to Raccoon to perform the testing.

Our proposed methodology is applicable to any web application deployment stack that uses a database to store permanent data. However, our implementation focuses on web applications based on the LAMP-Stack (Linux, Apache, MySQL, PHP) as this represents a popular web application deployment method [12–15].

We discuss the implementation of Raccoon along the same outline we presented our methodology in. We start with the input requirements (5.1), continue with the setup (5.2), data gathering (5.3), data analysis (5.4), and conclude with the verification (5.5) that includes a detailed description of the testing and the oracle.

5.1 Input

As defined in our approach the input to Raccoon are user traces and restorable web applications.

User Traces We chose Selenium as our automation technique as it provides a convenient interface for trace recording [16]. A tester can use the Selenium IDE, perform the required steps for the use case of interest on the web application and record them without any understanding of Selenium or browser automation [17]. The tester then saves the recorded steps in a file via the IDE and passes them on to Raccoon.

In case the application logic requires different input values for form fields (e.g., user login values) to run the same action chains concurrently on the same web application (e.g. to test the checkout processes) the tester has to adapt the files appropriately. During this process it is important to only change inputs that are nonessential to the testing. E.g. if one tests the amount of login attempts a user has it does not make sense to use different users. If one tests whether the same voucher can be overspent some applications require to use multiple different users – for example if the application ties the shopping baskets to a user and not the session.

Web Applications Our approach requires web applications to be restorable to previous states. We chose to use Virtual Machines (VMs) to achieve this. Our chosen hypervisor is Virtualbox[6] that supports all requirements previously stated. The used Virtual Machines have the LAMP stack and the web application already installed.

5.2 Setup

During setup Raccoon enables Xdebug in the Apache Server. Dumps generated by Xdebug contain every function called during the execution of a PHP script including their parameters. Thus, they provide a request-by-request log of all executed SQL queries. The Apache Server ties the Xdebug dumps to specific requests using unique ids and a log that links each id to the corresponding request URL.

Additionally, Raccoon inserts the MySQL-proxy[7] in between the web application and the MySQL server instance. This proxy

[6]https://virtualbox.com
[7]https://github.com/mysql/mysql-proxy

becomes relevant during the triggering of the potential vulnerability but lays dormant during the data gathering.

To conclude the setup Raccoon takes a *testing* snapshot representing the fully configured state of the machine.

5.3 Data Gathering

To gather all the required data Raccoon executes the recorded Selenium user traces against the web application GUI, retrieves and stores all collected data (i.e., Xdebug dumps, Apache id logs).

Raccoon first executes the given user action chain two times against the web application GUI restoring the *testing* snapshot after each execution[8].

Raccoon then executes the given user action chain 7 times[9] consecutively against the web application GUI without restoring the *testing* snapshot in between. Then Raccoon restores the *testing* snapshot.

5.4 Data Analysis

Equipped with the gathered data Raccoon can now approach the detection of GRCs. We chose the algorithm proposed by Paleari et al. for Raccoon. As we designed the detection interface of our approach around the two already existing approaches, the chosen approach works trivially with our collected data.

The algorithm proposed by Paleari et al. takes a list of SQL queries and searches for pairs of queries whose read and written columns intersect. They call such queries interdependent and consider them to be possible race condition vulnerabilities. To reduce the amount of falsely detected vulnerabilities Paleari et al. also propose a post processing. The post processing tries to infer whether data sets affected by either query actually intersect. This can only be done for simple cases such as mutually exclusive *WHERE* clauses and is not implemented by us as we rely on our verification to remove/detect falsely detected vulnerabilities. For a more in-depth explanation of the algorithm we refer the reader to the original paper [2].

5.5 Verification

The verification now takes all the vulnerabilities detected by the Paleari et al. algorithm, tries to trigger them, and then to pass on the collected data to a verification oracle.

However, as we limit ourselves to the subclass of GRCs Raccoon ignores all previously detected race conditions that are not part of the defined subclass. To determine to which part of the dichotomy a detected vulnerability belongs Raccoon takes our consecutively gathered data and count whether for each consecutive execution of the user trace the writing query was executed. If this is the case it indicates that the writing query is not protected by a guard and thus Raccoon discards the detected vulnerability as it is out of scope.

Triggering the Vulnerability Raccoon now attempts to trigger each unique detected vulnerability that still remains. To achieve this Raccoon resets the machine to the *testing* state and runs the user trace concurrently against the web application.

[8]This step is not included in the methodology section as it is not strictly necessary for our methodology to work. We encountered web applications that used random values (e.g., timestamp, session) in their URL and we use the data collected during the isolated two time executions to identify random elements in URLs.

[9]We settled for 7 as any higher number ran into frequent connectivity issues with Selenium. Based on our experience this is sufficient for verification of vulnerabilities.

The interleaving required for exploitation is achieved by inserting a delay in front of the writing query of the scrutinized vulnerability using MySQL-Proxy. The MySQL-Proxy accepts a user defined Lua script that allows for the customization of behavior for each intercepted query. Thus enabling Raccoon to transparently inject a delay when needed. We chose the delay to be sufficiently large to guarantee interleaving. After the process is done Raccoon extracts the collected data is extracted and passes them on to the oracle.

The Oracle: Raccoon passes the vulnerability, the consecutively collected data set, and the data collected during the verification attempt on to the oracle which is a subsystem of Raccoon. The oracle then performs the three actions as listed in Section 4.5 to verify that a candidate is a GRC.

(1) The oracle reduces the set of queries by *bucketing* the collected query logs according to the request URLs they are associated with. This happens for both the data set of the consecutively and concurrently executed user traces.
(2) The oracle *counts* the writing query of the candidate in both the bucket for the consecutively and concurrently executed user traces.
(3) If the count of the delayed query is higher in the consecutively executed user trace the oracle confirms a GRC.

Bucketing The oracle uses the URL of an HTTP request to assign buckets to the corresponding logged SQL queries. However, some URLs may contain volatile tokens (e.g. sessions, timestamps). Due to this, SQL queries belonging to the same bucket are associated with superficially different URLs. Consequently, the oracle needs to disregard all values of volatile elements contained in a given URL.

To be able to recognize volatile URL elements Raccoon collected all the requested URLs in a special log for the two user trace executions with restoring the snapshot in between to ensure equal logs (Section 5.3). The oracle cuts each log to a format where only the part representing a HTTP-Request remains. The next step of the oracle is to compare the lines of the two logs and check for the same amount of variables.

If the same amount of parameters is present the oracle compares the parameter values name-based to check whether the variable varies in value. Thus, the oracle is able to tell whether a parameter value varies between requests. This enables the oracle to bucket queries based on request URLs.

Counting After bucketing the oracle iterates over all queries contained in the bucket belonging to the request URL associated with the scrutinized detected vulnerability. The oracle removes all assumed non-static elements of the query, the elements on the right side of the equal sign, to ensure that volatile query elements such as timestamps do not disturb the comparison. Then the oracle counts the occurrence of the delayed writing query.

6 Experiment

To show the performance of Raccoon and, thus, practicality of our proposed approach we applied Raccoon on four different web applications and ten use cases. During this we discovered six GRCs with different security implications.

We first discuss the test bed, configuration, the chosen web applications, and use cases. Consecutively, we report the results of the experiments and conclude with the lessons learned during the application of Raccoon.

6.1 Selected Web Applications

We selected a set of eCommerce and Forum web application as both contain use cases that are not supposed to be executable multiple times successfully. For each application type we chose a set of use cases to show that Raccoon is able to be applied across applications and use cases. Each use case represents an abstract action regularly performed using the web application. Additionally, each chosen action is a use case that a user should be unable to do multiple times successfully, thus, actions such as using the search or simply clicking a link were not of interest.

Not every user trace chosen by us has the potential to encountered high severity vulnerabilities. We consider not focusing on potential high severity vulnerabilities to increase our testing range to be an acceptable trade off as our focus was on validating our methodology.

6.1.1 *eCommerce*

eCommerce web applications offer potential customers a select set of goods. A user has a virtual shopping basket they can use to collect goods they would like to purchase. After finishing selecting goods the user proceeds to the checkout to pay for the goods. The eCommerce web applications we tested all required that a user has an account to finish the checkout process to provide payment details as well as shipping address. We isolated three potential use cases in an eCommerce application:

Use Case login To protect a user account against multiple illicit attempts of login (i.e., bruteforcing a password) a web application should provide an overall limit of tries a user has to enter the correct password before they have to wait for a cool down period. A malicious user being able to extend that limit is perilous as previous research has shown that a seemingly low amount of tries can already result in a high success rate [18].

Use Case Coupon/Voucher eCommerce software allows the distribution of gift vouchers or coupons. Such a voucher or coupon allows customer to reduce the payment for a shopping cart by a fixed amount. As each such voucher is paid for by either the customer in case of a purchase or by the owner of the eCommerce store in case of a promotion. As a consequence the amount of uses is limited. This means that a given voucher cannot be used for more money than its worth. Consequently, it is to be expected that in case of multiple usages that eventually exceed the worth of a voucher the web application refuses to finalize the purchase.

We selected OpenCart (Version 3.0.3.1)[10], Abantecart (Version 1.2.14)[11], and Oxid (Version 6.0.2)[12] to represent the eCommerce category.

6.1.2 *Forum*

Forum web applications offer the means for intellectual exchange between users. A forum usually has multiple different topics in which a user can create separate discussions called threads. Each thread consists of multiple posts by different users in which a user presents their opinion on the topic or responds to a previous post in text form.

[10]see https://www.opencart.com/
[11]see http://www.abantecart.com/
[12]see https://www.oxid-esales.com/

Use Case login A forum usually requires a user to log in to take part in discussions and similar to eCommerce software a forum should limit the amount of login tries to protect against bruteforcing of a password.

Use Case flooding Even when logged in, a user is usually limited in the amount of new topics he can create or PMs he can send in a given amount of time. This is meant to protect the forum from malicious users and spammers that try to dominate the list of recent topics with their spam or try to harass another user by flooding their inbox with messages. Both actions have a disruptive potential on the forum community.

We selected MyBB (Version 1.8.15)[13] to represent the forum category.

6.2 Testbed Preparation

We conducted the tests on an Ubuntu 16.04 running on an AMD Ryzen 7 Pro 1700 with 3.7 GHz, 32 GB RAM, and a SSD. As servers hosting the applications we utilized virtual machines provided by Bitnami. Bitnami provides ready-to-go VMs containing a full LAMP stack and a fully functional, pre-configured web application[14].

Prior to start testing we had to modify the Bitnami provided machines slightly to accommodate the tests done by Raccoon. We collected those modifications for the used Bitnami machines into a setup script to allow for easier reproduction.

We changed the connection configuration between the web application and the shipped dbms to inject the MySQL-Proxy. Additionally, we change the Apache configuration to give each Xdebug execution trace dump a unique id that is stored in a log associated with the corresponding request URL. Both changes are required for every web application regardless of whether it is Bitnami provided or not. A further configuration change touches the setup of the web server. We need to ensure that sufficient concurrency is possible. Apache can limit the maximum amount of concurrent requests (i.e., maximum amount of running workers) to allow for resource management on smaller machines. We increased the limit to 10 by changing the corresponding configurations for Apache and the PHP CGI module. Please note that this change does not introduce the vulnerability as it can be expected that larger set ups already have this limited extended to an even higher amount of workers as reducing the limit to one – which would negate any possible race condition vulnerability – would significantly degrade the machines performance in a high traffic context.

We additionally prepared the web application itself when necessary. This involved actions such as creating non-admin accounts, decreasing the posting limits for forums, or generating a coupon/voucher for the eCommerce web applications. Those actions were necessary as the Bitnami provided machines were blank slates, in a continuous integration environment we expect that such pre testing preparation already has been done.

6.3 Detected Vulnerabilities

The detected vulnerabilities range in severity, and we classify them as either minor, medium, or significant:

Send PM/Create New Thread We consider the flooding circumvention in MyBB to be of *minor significance* as it does enable

[13]see https://mybb.com/
[14]https://bitnami.com/

	Use Case	Generated	Tested	Oracle Data	Consecutive Data	Test Time	Pos.	GRC
	login	2	1	3 min	35 min	18 min	1	●
OpenCart (Version 3.0.3.1)	*voucher*	*42*	2	*18 min*	*244 min*	*207 min*	*1*	●
	coupon	*42*	2	*17 min*	*394 min*	*230 min*	*1*	●
	login	5	0	6 min	N/A	N/A	0	○
MyBB (Version 1.8.15)	create new thread	32	5	17 min	360 min	244 min	3	●
	send pm	66	4	14 min	313 min	171 min	2	●
	login	0	0	6 min	N/A	N/A	N/A	○*
Oxid (Version 6.0.2-0)	voucher	-	-	-	-	-	-	○**
	coupon	27	2	20 min	210 min	192	1	●
	login	0	0	8 min	N/A	N/A	N/A	○*
AbanteCart (Version 1.2.14)	voucher	-	-	-	-	-	-	○**
	coupon	61	1	30 min	395 min	24 min	0	○

Table 1: An overview showing the results. *Generated* and *Tested* refer to generated and tested candidates respectively. *Oracle Data* refers to the time required to conduct the data gathering for the oracle. *Consecutive Data* refers to the time required to conduct the consecutive data gathering. *Test Time* refers to the time needed to finish all tests. *Pos* refers to the amount of verified GRC vulnerabilities by Raccoon out of all the tests. The column *GRC* shows whether the use case was actually vulnerable. Rows completely in cursive show previously known vulnerabilities used as a ground truth.
*** The action is not protected against multiple login tries**
**** This use case did not exist**

a malicious user to flood a forum or a PM box of another user. This may disrupt the community until the culprit is banned by administrators but does not result in direct financial loss.

Login Bruteforcing a login can turn out to be significant but requires a certain amount of tries that are hard to fit within the short timing window required to exploit GRC vulnerabilities. Consequently, additional amplifying circumstances are required to turn this vulnerability into a threat model (e.g., knowledge of a small selection of possible passwords used by the user). We consider this issue to be of *medium significance.*

Coupon/Voucher The voucher/coupon overspending vulnerabilities are *highly significant.* Any exploitation has a direct financial impact for the web application owner. Additionally, the tested web applications perform the check for the voucher applicability before the complete checkout and thus an attacker is able to simply restage the attack in case of a failure without significant overhead.

The OpenCart vulnerability for both voucher and coupon were already detected by previous work and reported to the developer [1, 7]. All other vulnerabilities were reported by us to the developers.

6.4 Discussion

Raccoon verified vulnerabilities in MyBB, OpenCart, and Oxid and we give a summary of the conducted tests and related meta data in Table 1. During the tests we encountered *multiple query pairs* that exhibited anomalies during forced interleaving, and we encountered *high execution times* due to relying on Selenium.

6.4.1 Multiple Query Pairs: When looking at the listed use cases such as flooding a forum, one would expect a single query pair exhibiting anomalous behavior with the pair consisting of one query extracting the last time of the last flood protected action and one query updating that value. This was not the case for the flooding use cases in MyBB. The use case showed multiple query pairs that exhibited a higher occurrence during forced interleaving. We also encountered multiple query pairs during testing of Oxids coupon.

MyBB - Flooding: Both tests for flooding prevention had multiple queries that exhibited the hallmarks of a GRC. Not all queries in the PM flooding use case were linked to the use case at hand and we were not able to infer any security implications besides for the query pair directly related to the use case. Nonetheless, the query pairs showed the hallmarks of a GRCs and thus should be addressed by a developer.

For the creation of a new thread, however, both query pairs were related as one query pair represents the creation of the new thread and one represents the creation of the corresponding messages in the database. A delay between either query pair lead to the flooding detection being circumvented.

Oxid - Coupon: We encountered two query pairs when testing the Oxid coupon use case. One query pair was related to the Oxid coupon. The second query pair was related to Oxid internal SEO optimization. We were unable to link it to any security sensitive behavior.

6.4.2 High Execution Times: Relying on Selenium comes with the drawback that Selenium is unable to recognize when the browser performs background requests such as AJAX calls. This leads to Selenium trying to perform an action, such as clicking a button, that relies on interface elements that the browser has not yet loaded. This in turn resulted in Selenium aborting the execution. To counter this detrimental behavior we introduced long waiting periods between each step of the user trace. This ensures that all background requests had finished before performing the next step.

The result of this precaution is that depending on the user trace, Raccoon requires a significant amount of time to run a single user trace. Consequently, at a first glance, it seems that Raccoon requires a long time to complete testing a web application, possibly without actionable results.

This is a misconception. Raccoon bases its testing on the detected vulnerabilities extracted after running the user trace twice (gathering the data for the url parameter oracle). If no possible GRC is detected, Raccoon does not proceed further and the tester can proceed to the next user trace. If, however, possible race condition candidates are detected, Raccoon starts its consecutive execution data gathering to determine whether any of the detected possible vulnerabilities fit the GRC definition. Any detected candidate that does not appear to fit Raccoon eliminates as well. In the end Raccoon only tests strong candidates for GRC which reduces the overall runtime. The final runtime is admittedly not that low, but as our results show, leads to verified GRCs – some with serious financial implications.

It is possible that not all verified GRCs are security relevant. However, this assessment cannot be done by a program and even if there is no immediate security implication a GRC is still unintended behavior that a developer needs to address. Thus, time spent on verifying strong GRC candidates is time well spent in our opinion.

6.5 Lessons Learned

We tested four web applications and ten use cases. For OpenCart every single use case was vulnerable to GRC and Oxid and MyBB have at least one vulnerability. Even though we did focus on potentially vulnerable use cases, we were surprised by the high amount of vulnerabilities we detected. Additionally, researchers reported the OpenCart voucher vulnerability to the developer over a year ago [1] and the coupon vulnerability even earlier [7]. This shows that GRC are not taken as a serious threat by developers or the necessary skills to properly fix the issues are not present.

Another lesson we learned was that it is possible that multiple query pairs lead to security relevant anomalous behavior if they interleave for a single use case. We encountered such an occurrence in MyBB and a delay of the detected query pairs can lead to an exploitable vulnerability. Given that execution times for queries add up it becomes obvious that the more vulnerable query pairs there are the easier it becomes to achieve interleaving for successfully triggering the vulnerability.

7 Future Work

Applying Raccoon showed drawbacks and limitations that we intend to address in the future. This includes improving the range of testable applications.

Raccoon is limited to test GRCs (Section 3.2.1). This does not incorporate every possible race condition manifestation. Thus, extending the validation algorithm and oracle to cover every race condition type would greatly increase the usability and coverage of Raccoon.

Finally, Raccoon is currently only designed to cover LAMP based application. LAMP, however, is not the only configuration for web application deployment and a big ecosystem not relying on PHP or even relational SQL-based databases exists such as Ruby based server side code or the usage of NoSQL databases. Consequently, transferring the approach implemented by Raccoon to cover more database types (e.g., NoSQL) and different server side languages would improve the applicability of Raccoon in both research and security testing alike.

8 Conclusion

We presented an automated methodology on verifying GRCs and its implementation in form of the tool Raccoon. Our approach enables testers and web application designers alike to perform automated testing for GRCs.

Raccoon only requires a tester to provide a web application as well as the use case that they want to test in the form of a Selenium script. Thus, we extended the portfolio of existing automated web application testing tools to account for GRCs.

We applied Raccoon on four web applications and ten use cases. We were able to identify six vulnerabilities of which four where previously unknown.

Given the potentially high impact of GRCs and their apparent prevalence, it stands to reason that this vulnerability class is in dire need of increased attention.

9 Acknowledgments

This research was supported by the Lower Saxonian Ministry for Science and Culture as part of the research program MOBILISE (Mobility in Engineering and Science).

References

[1] T. Warszawski and P. Bailis, “Acidrain: Concurrency-related attacks on database-backed web applications,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064037

[2] R. Paleari, D. Marrone, D. Bruschi, and M. Monga, “On race vulnerabilities in web applications,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings*, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-70542-0_7

[3] “Owasp:top 10 project,” accessed 2019-09-05. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[4] L. Constantin, “Withdrawal vulnerabilities enable bitcoin theft from flexcoin and poloniex,” March 2014. [Online]. Available: https://www.pcworld.com/article/2104940/withdrawal-vulnerabilities-enabled-\bitcoin-theft-from-flexcoin-and-poloniex.html

[5] E. Hamokov, accessed 2019-09-05. [Online]. Available: https://sakurity.com/blog/2015/05/21/starbucks.html

[6] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Feral concurrency control: An empirical investigation of modern application integrity,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015. [Online]. Available: http://doi.acm.org/10.1145/2723372.2737784

[7] Y. Zheng and X. Zhang, “Static detection of resource contention problems in server-side scripts,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337292

[8] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow, “Deemon: Detecting csrf with dynamic analysis and property graphs,” in *Proceedings of the 2017 ACM Conference on Computer and Communications Security*, 2017.

[9] S. Mcallister, E. Kirda, and C. Kruegel, “Leveraging user interactions for in-depth testing of web applications,” in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87403-4_11

[10] G. Pellegrino and D. Balzarotti, “Toward black-box detection of logic flaws in web applications,” in *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*, 2014. [Online]. Available: http://www.eurecom.fr/publication/4207

[11] “Owasp testing guide,” accessed 2019-09-05. [Online]. Available: https://www.owasp.org/index.php/OWASP_Testing_Project

[12] “Usage of web servers for websites,” accessed 2019-09-05. [Online]. Available: https://w3techs.com/technologies/overview/web_server/all

[13] “Usage of server-side programming languages for websites,” accessed 2019-09-05. [Online]. Available: https://w3techs.com/technologies/overview/programming_language/all

[14] “Topdb top database index,” accessed 2019-09-05. [Online]. Available: https://pypl.github.io/DB.html

[15] “Usage statistics and market share of unix for websites,” accessed 2019-09-05. [Online]. Available: https://w3techs.com/technologies/details/os-unix/all/all

[16] “Selenium browser automation,” accessed 2018-03-16. [Online]. Available: https://www.seleniumhq.org/

[17] “Selenium ide,” accessed 2018-06-11. [Online]. Available: https://www.seleniumhq.org/projects/ide/

[18] D. Wang, Z. Zhang, P. Wang, J. Yan, and X. Huang, “Targeted online password guessing: An underestimated threat,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978339