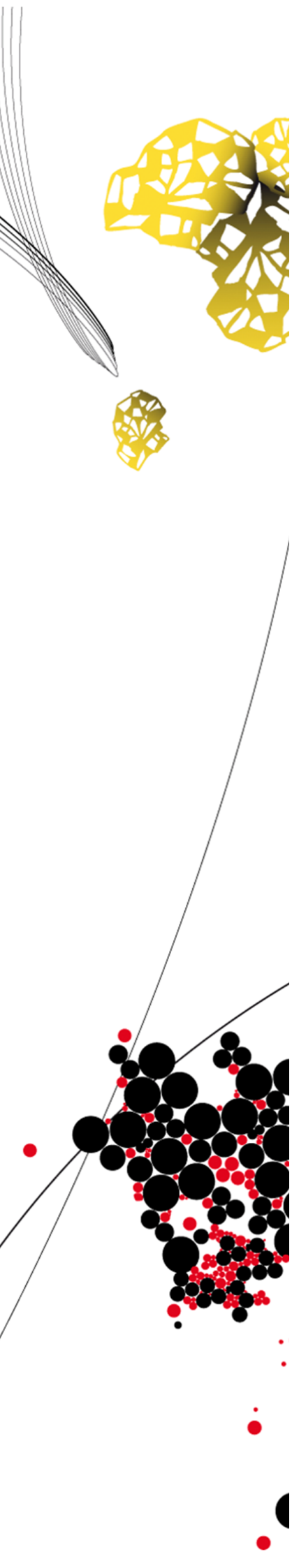


UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Towards Systematic Black-Box Testing for Exploitable Race Conditions in Web Apps

Rob J. van Emous
r.j.vanemous@student.utwente.nl

Master Thesis
June 2019

Supervisors:
prof. dr. M. Huisman
dr. ing. E. Tews
(Computest) M.Sc. D. Keuper

Formal Methods and Tools Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

ABSTRACT

As web applications get more complicated and more integrated into our daily lives, securing them against known cyber attacks is of great importance. Performing security tests is the primary way to discover the current risks and act accordingly. In order to perform these test in a systematic way, testers create and use numerous vulnerability lists and testing guidelines. Often, dedicated security institutions like Escal Institute of Advanced Technologies (SANS), MITRE, Certified Secure, and the Open Web Application Security Project (OWASP) create these guidelines. These lists are not meant to be exhaustive, but as the introduction in the Common Weakness Enumeration (CWE) of MITRE and SANS by Martin et al. (2011) puts it: they *"(..) evaluated each weakness based on prevalence, importance, and the likelihood of exploit"*. This seems like a reasonable way of thinking, but as we have shown in this research, it can also lead to oversight of essential, but stealthy security issues.

In this research, we will focus on one such stealthy issue in web apps called a race condition. Race conditions are known for a very long time as research by Abbott et al. (1976) shows. Still, they are a type of security vulnerability that is often not included in these lists as it is challenging to test for and also is not often exploited. Based on the lack of research in this field, we argue that especially in the web-environment, it has resulted in an underestimation of the risks involved. The races continue to show up in web apps and when exploited, could have a significant impact as a recent security blog by Jadon (2018) shows. This impact ranges from circumventing any limited-usage functionality like coupon redemption to enabling other types of security vulnerabilities like privilege escalation or a Denial of Service (DoS).

That is why, in this research, we developed the first systematic method to test for race conditions in web apps from a black-box perspective. We also built a tool to support the exploitation and evaluated both in comparison with related tools.

1. **Methodology** - we have devised the first method for systematically testing for race conditions in web apps from a black-box perspective. Most importantly,

this method contains a list of common race condition vulnerabilities in web apps and a detailed strategy of how to test for these items.

2. **Toolset** - Next to this, we have developed the toolset called CompuRacer to support the tester in the execution of this systematic test. It supports the gathering of HTTP requests of interest, the parallel sending of these requests and guided evaluation of responses.

3. **Evaluation - toolset** - Subsequently, we have evaluated both the toolset and method. In order to do this, we compared the toolset to three related toolsets on a functional-, usability- and performance-level. For the performance evaluation, we used the tools in a real-life setup on a self-developed web app that is vulnerable to race conditions. In this evaluation, we tested the raw speed of sending parallel requests and the ability to exploit race conditions using the appropriate statistical tests. Regarding all of these metrics, the toolset is shown to be equal or better than all other tools.

4. **Evaluation - method** - Finally, the method and toolset are evaluated together on seven web apps ranging from e-commerce platforms to blogs and wikis. We were able to find much minor race condition related issues in these platforms, but more importantly, for two e-commerce platforms, a severe vulnerability has been found and reported which has a significant financial impact.

Based on this, we conclude that we have successfully created a method and toolset that are sufficient for security testing. We are also aware that much more research is required to expand upon these findings. Still, we hereby achieved the first step towards systematic testing for race conditions in web apps, and by that, we hope that this will have a positive effect on software quality in the future.

PREFACE

In my heart, I am both a meticulous software developer and a curious seeker of truth. In academics, but also in other aspects of life, I experience that this goes hand in hand perfectly. I remember that often when a professor teaching calculus would make a remark about the computational complexity of a function, I would instantly be prompted to write some code to verify his claims. They were always right.

Likewise, when creating software for other reasons like the automation of processes and exploration of ideas, it feels like both an amazing privilege and a significant responsibility. That is why I pursue to develop and encourage well written and secure software. This often requires countless hours of designing, building, testing, and building some more, but the end result is worth it. So it has been with my thesis. *"Failures, repeated failures, are finger posts on the road to achievement. One fails forward toward success."* as C.S. Lewis would cleverly put it. The research was challenging but practical enough to fulfil my creative desire while hopefully still being a truthful enrichment to the academic world.

ACKNOWLEDGEMENTS

There would be no way I could have achieved this result on my own. Firstly, I would like to thank my parents, brother and girlfriend for their loving support. Secondly, my committee members, each of whom has been patient with me in times of doubt and provided me with the necessary guidance throughout the process. Finally, as a follower of Christ, I would like to thank God for the love and support I believe He provides for me in life, but especially during this intense final period of my studies.

"For I can do everything through Christ, who gives me strength."
Philippians 4:13 - The Holy Bible (NLT)

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CI	Continuous Integration
CIA	confidentiality, integrity and availability
CLI	Command Line Interface
CMS	Content Management System
CSS	Cascading Style Sheets
CWE	Common Weakness Enumeration
DBMS	DataBase Management System
DOM	Document Object Model
DoS	Denial of Service
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol (OSI level 7)
HTTP/1.x	HTTP version 1.0 or 1.1
HTTP/2	HTTP version 2
HTTPS	HTTP over TLS
IP	Internet Protocol
JSON	JavaScript Object Notation
MitM	Man-in-the-Middle
OSI	Open Systems Interconnection (model)

OWASP Open Web Application Security Project

PHP Hypertext Preprocessor

RFC Request For Comments

ROI Return On Investment

SANS Escal Institute of Advanced Technologies

SIGSEGV Signal Segmentation Violation

SQL Structured Query Language

SSRC Server-Side Race Condition

SSRCs Server-Side Race Conditions

TCP Transmission Control Protocol (OSI level 6)

TOCTOU time of check to time of use

UDP User Datagram Protocol

WebRTC Web Real-Time Communication

WS WebSocket (OSI level 7)

WSS WS over TLS

XHR XMLHttpRequest

ZAP Zed Attack Proxy

Chapter 1

Introduction

1.1 Problem description & motivation

1.1.1 Importance of secure web apps

In modern society, we have fully integrated web applications into our everyday life. They are used in all parts of life, including education, entertainment, taxes, shopping and banking. As we trust these applications with our most important data, it is essential that these applications are secure and available at all times. Both when normal users use the web app (during peak hours), but especially when a malicious user tries to attack it. Preferably, it should not be feasible for an attacker to extract or destroy private information, act on our behalf, or disturb the availability of these applications. In other words, the software quality should be at an appropriate level. The question is, how do we make sure that is the case?

As we know from very early research regarding software testing, finding all flaws in software is undecidable (Howden, 1976). To our best knowledge, this statement still holds today. The best possible alternative can be found in a level of software quality and maturity that is sufficient given the risks involved. In this case, 'risk' is defined by (McGraw, 2006, pp. 144-146) as the probability that an asset will suffer an event of a given negative impact. High risk can, therefore, result from both a high probability of the adverse event and also from a potentially catastrophic impact.

For example, if the application is a single page web app that only displays static information, exploitation is not deemed to result in much negative impact and is also not very likely to be interesting for attackers. Thus, low risks are involved, and the level of software quality is allowed to be on the lower end of the spectrum. On the contrary, for a high-valued banking application, it is very interesting for attackers,

and the financial impact could be significant. Therefore, this poses high risks and requires an equally high-security standard.

In order to create such a secure software system, it should be designed and implemented with security in mind. Even this is not enough as people are known to make mistakes which could lead to security bugs. Next to this, developers often expect educated and benign users to use their application. Not all users are educated users, or benign. Any user might be an attacker that tries to penetrate the application's safety measures and wreck havoc. Therefore, during and after development, we also need to perform a variety of automated and manual testing to make sure we meet our functional and security requirements.

In this thesis, the focus is on the security testing of web applications in order to improve software quality. More concretely, it first points out a lack of knowledge regarding security testing in one specific area. Many manuals exist on how to perform a black-box security test in a systematic way. However, we will show that an important issue is consistently lacking in attention both in theory and in practice. This issue is called a race condition. We will show that exploitation of this issue can significantly impact the security of a modern web application, but that testers also often neglect it. Historically, testing for race conditions has received attention, but we show that this effort is far from mature and cannot easily be applied to the complex multi-part web applications of today. That is why we will investigate how systematic testing of this specific issue can be introduced into the skillset of a security tester.

1.1.2 Danger of race conditions explained

Race conditions are a family of bugs that is timing-related. The Common Weakness Enumeration regarding software systems by the SANS Institute and the MITRE Corporation (2019) lists this issue in the *'time and state'* category (CWE-361) under the specific item: *'Concurrent Execution using Shared Resource with Improper Synchronisation'* (CWE-362). In other words, it is an issue that can occur when accesses to a common variable for multiple processes are not properly managed. Providing several examples might be the easiest way to show how this issue occurs in practice and how it can have a significant impact on web app security.

For instance, one process might increment a variable with value five by one, while another process tries to decrement the same variable by one. When the processor executes these actions sequentially, we will end up with the original value. However, when these actions are executed in parallel, as is rather normal using the multi-core processors of today, one might overwrite the result of the other. We illustrate this

race condition between two threads in figure 1.1. In this case, the outcome of the incrementing process (T1) gets overwritten by the decrementing process (T2).

The example does not sound too dangerous, but what if the affected value is the available money on a bank account? Alternatively, what if the true/-false value that indicates whether a user has access to some secret information? Instead, what if it might push the system in an unexpected state after which it crashes? These are severe consequences to the integrity, confidentiality and availability of the application. If one or both of the involved concurrent processes are controllable by an attacker, it poses a real security threat. Next, we give two examples of a race condition bug and its consequences in order to show that these theoretical issues also occur in practice.

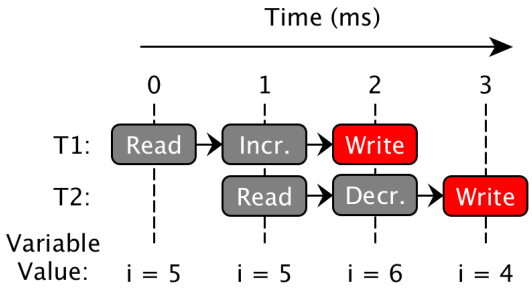


Figure 1.1: *The figure illustrates two threads T1 and T2 that perform an update action to variable i. T1 increments the value by one and T2 decrements the value by one. As both threads read the same value of $i = 5$, the result of the race condition is that T2 overwrites the result of T1.*

Example 1 - Duplicate account creation

We found a concrete example of an unexpected state that crashed web app functionality during the exploration phase of this research. In this phase, we performed a test for race conditions on the OWASP WebGoat, which is a deliberately insecure e-learning web application based on the Spring MVC framework. It is supposed to be insecure in some aspects, but not regarding race conditions, and that made it an attractive target.

In this case, due to a race condition in account creation, multiple accounts with the same username could be created while this should have been impossible. The result was that the newly created accounts could only log in and log out, but all other user-specific functionality no longer worked. A more far-reaching consequence was that the general scoreboard did no longer work for any current or future user. As this web app could be used for Capture The Flag (CTF) contests, this would be a very undesirable impact.

We found two root issues:

1. The first issue is the fact that the developers did not mark the username as a 'unique' field in the 'user_tracker' table in the database. This is shown in the WebGoat database diagram in figure 1.2.

2. The second issue is found in a TOCTOU or RCA - race condition (see definitions in section 4.1) in the app server with regards to the check to add only unique users. In listing 1 a non-synchronised method in the `RegistrationController` class is shown that is responsible for adding new users to the database. At line 4, both the 'Read' and 'Check' acts are performed to validate whether this is a new user and at line 8, the 'Act' is performed by adding the new user. The time between executing line 4 and line 8 is the race window.

These issues made it possible to successfully add two accounts with the same username in parallel. The 'user_tracker' table is often read using a Spring function that expects only one result. As it would get two (or more) results when loading any user-based functionality, this function would throw the error: `javax persistence NonUniqueResultException` with the explanatory text: `result returns more than one elements` and the functionality would fail to load. The issue described here is a clear case of an exploited race condition that impacts the availability of the application as a whole.

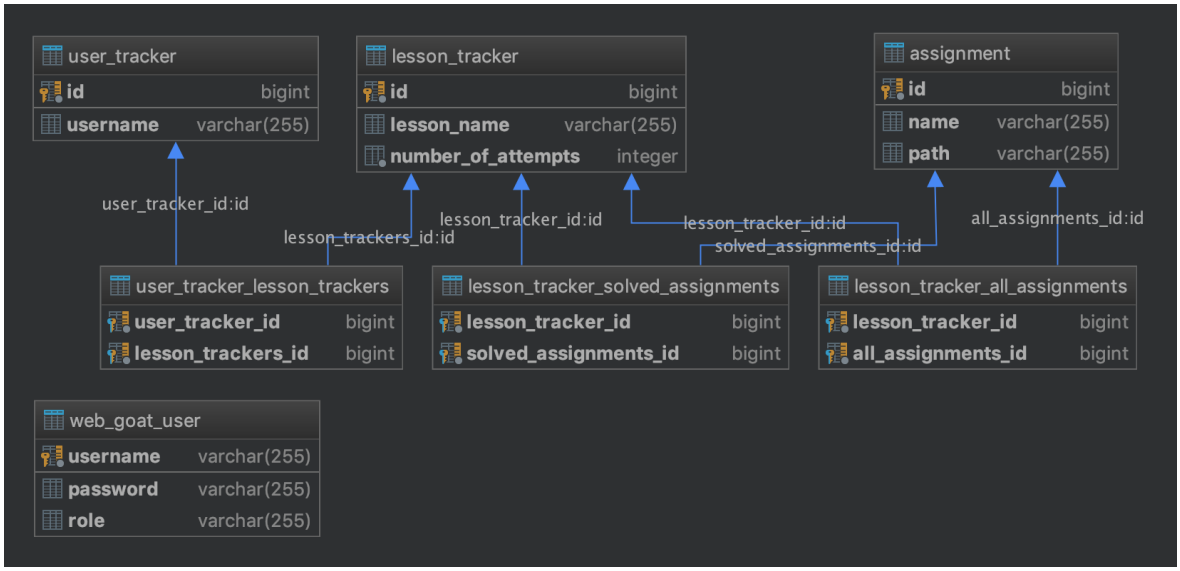


Figure 1.2: *The figure shows the flow from web app functionality to potential race condition vulnerabilities. For every vulnerability, the expected impact is also indicated using the appropriate coloured circles.*

```
@PostMapping("/register.mvc")
@SneakyThrows
public String registration(@ModelAttribute("userForm") @Valid UserForm
 ↪  userForm, BindingResult bindingResult, HttpServletRequest request) {
    userValidator.validate(userForm, bindingResult);
    if (bindingResult.hasErrors()) {
        return "registration";
    }
    userService.addUser(userForm.getUsername(), userForm.getPassword());
    request.login(userForm.getUsername(), userForm.getPassword());
    return "redirect:/attack";
}
```

Listing 1: *The listing shows to method that is called when a new WebGoat user tries to register.*

Example 2 - Session puzzling

To further motivate the difficulty in testing for race conditions in web apps, let us provide a second illustrative example of a race condition vulnerability in a web app. Imagine a web application that provides access to public and private forums about all kinds of subjects. The user has created an account and is using the forum now for some time. One day when he logs in and the application redirects him to the profile page; he receives the information of someone else. All details are there, and he can view the contents of all private forums that this user has access to. Being a nice person, he decides not to abuse the issue and log out. Later, when he logs in again, he is back to his own page. How is this possible?

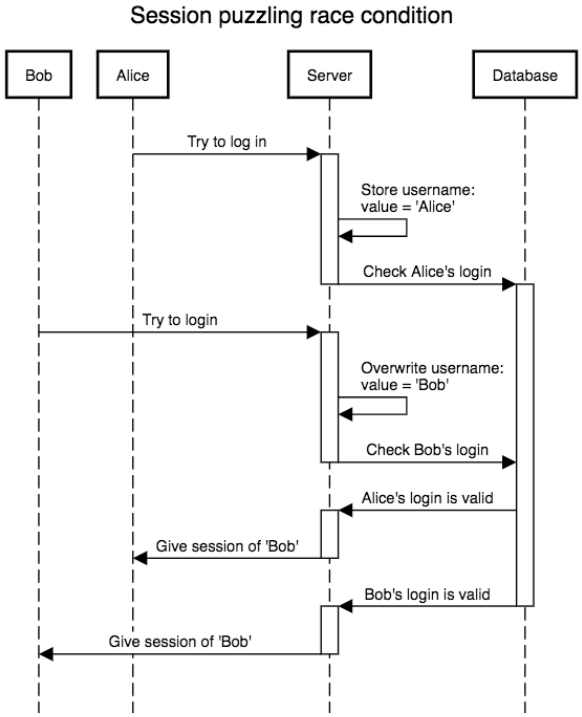


Figure 1.3: *A sequence diagram showing interaction with a server vulnerable to session puzzling that triggers the race condition.*

The issue described above is not just a hypothetical situation, but it is a real session puzzling race condition that is introduced in the Web Application Hacker's Handbook (edition 2) written by (Stuttard and Pinto, 2011, pp. 426-427). One of

the authors had personally run into this specific issue. In that instance, the login process required several steps. This is normal in, for instance, multi-factor authentication. During the process, the server locally stored the username of the user that was logging in. This username would be used in the end to send the correct session information to the client of the user. When two users would log in at roughly the same time, thus within the race condition window, one would occasionally get the session information for the account of the other. Thus, the sessions would be mixed up or puzzled.

Even when a login consists of just a single request by a user and the server stores the username while verifying the credentials using the database, this issue is still possible. The sequence diagram in figure 1.3 illustrates this issue. Between the moment Alice's username is stored at the server and the response of the server with her session information, Bob could also try to log in, accidentally overwrite the username and unintendedly give Alice full access to his account.

1.1.3 Difficulties in testing for race conditions

Most web application tests do not formally include the security issue of race conditions. Therefore, any reported race often falls in the category of unexpected findings. Even then, the issue will be difficult to reproduce and therefore, difficult to understand or resolve. In the case that a tester specifically tests for races, depending on the individual, it does not go beyond some limited effort of the tester to perform a particular request several times in parallel. For instance, trying to use a discount code twice. Incidental testing for an issue can hardly be called a thorough security assessment but is the current state of affairs. The reason behind this lack of thorough testing for race conditions is that they are relatively difficult to test for. The difficulty exists because race conditions are probabilistic events that are state and time-related. They often need a specific program state to become a possible outcome and even then can only be triggered during a minimal time-frame. This period is called the race window. In the context of remote security tests, finding and exploiting these issues becomes even more difficult as security tests mostly happen without access to the internals of servers or full knowledge of the source code. Black- or grey-box testing is a common name for testing with (very) limited knowledge. In this case, as (OWASP, 2009) mentions, only the responses from the server can be monitored, and this increases the difficulty of verifying suspected issues. For this research, the focus is on black-box testing.

As both attackers and testers do not seem to have a systematic method of finding these issues, their discovery and exploitation remain rare for now. That, in turn, justifies only limited time and resources to improve on this situation and thus the problem remains. As we have shown in the examples, incidental findings do show that race conditions are prevalent in web apps, and therefore the current lack of systematic exploitation is not a good indicator that this will remain the case in the future. The MITRE Corporation (2019) page for race conditions also confirms this by stating *"Race conditions in web applications are under-studied and probably under-reported (..)"*. That is why the addressing of issue is very important so that testers will develop systematic testing for race conditions before the attackers do.

Despite the lack of systematic testing for race conditions in web apps, after conducting a thorough search, we could find some effort in this direction in testing manuals of security organisations and academic writing. We discuss these efforts below.

1.1.4 Overview of current web app testing methods

For the reasons mentioned before, race conditions have never been a central part of the main security test manuals of the most renowned security institutions. More specifically, it is not included in the OWASP top ten lists of 2010, 2013 or 2017 (OWASP community, 2017, p. 6), or the Certified Secure (2018) checklists for server and client testing. Only the extended weaknesses list of the MITRE Corporation (2011) contains race conditions on position 33. However, when we dive into the most elaborate security testing guides for web apps, some pages devoted to race conditions can be found.

For instance, in the 300+ page OWASP testing guide by (Meucci et al., 2008, pp. 144-146), it is said that race conditions *(..) may occur when a process is critically or unexpectedly dependent on the sequence or timings of other events*. The authors explain dangerous areas as places where there is a difference between the time of check to time of use (TOCTOU) of resources. In other words, they regard any non-atomic access to a local variable or database as a potential race condition vulnerability worthy of attention. Interestingly, in the latest edition of this document (version 4) by (Muller et al., 2013, pp. 40,98), even less information can be found regarding race conditions. There is one link to a blog by Chen (2011) about session puzzling by using race conditions. This issue is equivalent to our illustrative example. It is an actual vulnerability as it uses extra server load to make temporary (admin) session values last long enough to use them to access sensitive information.

There does exist some effort in the literature and in practice to resolve this lack of systematic testing for race conditions in web apps. However, most of these efforts are only focused on functional or visual issues that occur on the client- or server-side due to races. Research that seems to focus on exploitable race conditions in web app logic or database interactions is minimal. The rarely found, related blogs of security testers seem to be the only recent evidence that people are looking into the security implications of race conditions in web apps. We will discuss all of these efforts will in more detail in chapter 3 on state of the art.

Although several tools do exist like Sakurity racer, Race the web, Turbo Intruder and netCloneFuzzer that target precisely this, in later sections, we will show these have significant limitations. For instance, these tools only make the exploitation of a known simple attack surface, like an account creation page, easier. As already explained, duplicate accounts can often be created due to race conditions. Without the knowledge of where to look for race conditions issues in web apps, the tools have only minimal use. Next to this, they are only able to send a single request in parallel. The lack of support for different requests or send delays is insufficient when complex configurations of requests are required to trigger a race. Finally, they provide insufficient help in interpreting the server responses. When a tester has to verify issues or classify the associated risks, (semi)automated interpretation of responses is essential. We discuss these tools more in-depth in section 3.2.3 on related work and will also compare them to our product in chapter 6 on the evaluation of the testing method and the toolset.

1.1.5 Classical race conditions tests for single-tier applications

In order to solve this lack of expertise, one might be tempted to look at methods of development and testing that people have used historically. It used to be much more common to create a monolithic or single-tier application. As Smith et al. (1998) state, a single-tier application is a system in which the user interface, program logic and storage interface is all contained in one place and often in one language. In these rather simple software applications, we can see that Abbott et al. (1976) already mention concurrency issues and that makes it one of the oldest security problems.

The development of web applications started at a later point in time, but these could also be considered sing-tier applications. All logic and storage were located at the server, and the client browser only had to display the static webpage. If the browser is not considered to be a different program or just a user interface but is included

in the total client-server system as an active entity, the system could also be regarded as a two-tier application like (Tanenbaum and Van Steen, 2007, pp. 549-551) does.

Many tools like (Flanagan and Freund, 2009, 2010; Wilcox et al., 2018) have been designed over the years to help the developer of single-tier applications to deal with race conditions. Some of these tools are discussed here. These tools started as static checks that just looked at the source code without running the application. They verified the correctness of the application: that the application used proper synchronisation functions like fork-join pairs, locks, semaphores and mutexes (mutually exclusive access). Later, developers also built dynamic tools that looked at the application when it was actively running. For instance, FastTrack developed by Flanagan and Freund (2009) and VerifiedFT developed by Wilcox et al. (2018) were designed to verify that all concurrent sections either affect each other only temporarily or when they do share some memory or other logic, that they are serializable. In other words, whether these concurrent statements can be converted to a single series of non-concurrent statements. These tools have an excellent performance, even in complex single-tier applications, but when the topology of the system increases to a multi-tier system with many involved technologies and languages, they are no longer applicable.

Unfortunately, web applications of today also fall in the category of multi-tier applications. At least three tiers can be distinguished:

1. **App server** – This is the heart of the application. It receives and processes all user requests, communicates with the database, and provides response pages to the user. A plethora of frameworks, languages and database types are available to create the app server logic.
2. **Web server** – This is the entry point of the application for the user and securely forwards all traffic between the app server and the user.
3. **Browser** – This shows the web page, but also contains the logic to respond to user actions, interact with the server and to update the interface.

The intricate architecture of current day web apps clearly shows that we cannot directly transfer the knowledge of how to test single-tier applications to this domain. Testing individual parts of these apps for race conditions might still be possible using traditional methods, but these tests cannot take into account the complex interaction of different parts and the different languages involved. Next to this, the usage of event-based languages like JavaScript at the client- as well as the server-side that have almost no built-in concurrency control, impacts how testing must be performed (Hong et al., 2014; Ide et al., 2009; Wang, 2017; Wang et al., 2017). Finally, as

security testers often only access a web app from a grey- or black-box perspective, this approach is incompatible with the full source code access that most non-web app tools require. Based on all this, we must devise entirely different methods to test for race conditions in web apps.

1.2 Research questions and methodology

It is clear that race conditions can pose a serious threat to the security of modern web applications and exploitation can have a big (financial) impact. At the same time, there is both a lack of awareness of these issues and a lack of capability to efficiently test for them. Existing knowledge on how to test for race conditions is shown to be not directly applicable as it mostly targets single-tier applications. When it does focus on web applications, it does not look at the security of the application as a whole. The handful of tools that does exist appears to be too simple to fit the current strategy of security testers.

At the beginning of the chapter, it was stated that the thesis would investigate how systematic testing of this specific issue can be introduced into the skillset of a security tester. We can now conclude that at least two hurdles have to be taken to make this possible: 1) Find out in what places race conditions could occur in web apps, and 2) explore how these places can be systematically detected and exploited from a black-box perspective. With this in mind, we propose the following main question for this research:

How can we perform systematic black-box testing for exploitable race conditions in web apps?

In order to answer this question, we need an appropriate methodology. We found a similar type of security testing research, and this inspired our methodology. The research was done by Kuosmanen et al. (2016) with regards to automated security testing of WebSocket implementations. They first tried to understand the technology and common security vulnerabilities. Then, they investigated how to test for these issues and finally, they created an automated testing tool.

When we apply this to our research, we first need to know where Server-Side Race Condition (SSRC) vulnerabilities typically occur in web apps and investigate what impact they have. Then, we investigate how to exploit them from the client-side in a general and systematic way. We use these findings to create a semi-automatic testing tool. Finally, this tool should be applied to some real web apps to test the performance and usability.

We can rewrite this procedure as three research questions:

- **RQ1** - In what parts of web app functionality do SSRC vulnerabilities occur?
- **RQ2** - How can we make SSRCs more likely to happen from a black-box perspective?
- **RQ3** - How can we develop an effective tool to help find and exploit SSRCs from a black-box perspective?

Chronologically, we will start by investigating RQ1 and RQ2, but this will overlap with the creation and testing process of the tool of RQ3. The reason behind this is the fact that the knowledge of race conditions in web apps is currently rather poor. Therefore, we will gain much information about where to find these issues and how to exploit them by applying the tool in practice. Subsequently, this knowledge can be used to add information to RQ1 and RQ2. Based on a combination of the knowledge about how to test for race conditions in a systematic way and the usability and performance of the final tool, we can answer the main research question.

The value of the research is significant because, to the best of our knowledge, no research exists that looks into systematic black-box testing for exploitable race conditions. If the results are positive, our research makes a significant leap in closing this gap of knowledge. If we would have to answer the main question in the negative, the gained knowledge about the location, impact and exploitation of race conditions in web apps is still a valuable addition to the current lack of research about this subject.

1.3 Contributions

Concretely, the following contributions of the thesis can be discerned:

1. **Methodology development** - A preliminary systematic methodology for testing a web app regarding race conditions from a black-box perspective. The method includes an answer to the questions of how to discover race conditions (RQ1), how to combine requests to trigger race conditions, and how to evaluate whether the race happened (RQ2). This methodology both includes a general solution to these issues and an answer specific to particular web app types like blogs, wikis and e-learning platforms.
2. **Toolset creation** - A toolset that can support the tester in following the methodology as described before (RQ3). This toolset integrates well with the current security testing practices by importing requests of interest from existing testing

tools. Next to this, it is capable of supporting complex combinations of requests that can be sent in parallel to trigger race conditions. Last, it supports the tester in evaluating whether the exploitation was successful.

3. **Evaluation of methodology and toolset** - The existence of a testing method and a toolset are already an improvement upon the current situation, but without any evaluation regarding the effectiveness of the proposed method and tool, the validity of the answers to all research questions diminishes greatly. Therefore, an evaluation of the effectiveness of these contributions is added as well (RQ1, RQ2, RQ3). This evaluation covers the functionality, usability and performance of the toolset as compared to similar tools put forward by related work. Next to this, it contains a practical application of the method and the toolset to test for race conditions in several web apps. After this application, we can easily evaluate the effectiveness of the method and toolset combined.

When we obtain the results regarding all research questions, this should lead to a satisfying answer to the main research question as well. Thus, having the first version of a systematic way to perform black-box testing for exploitable race conditions in web apps.

1.4 Commissioner

As ready indicated in the motivation, during security assessments, testers often run into a range of particularities of web applications that seemed to be timing-related. Some of these issues were shown to be exploitable, but a systematic way to find them was not yet employed. Some research showed that not only Computest, but most of the companies in this field do not systematically test for race conditions in the web apps they test. With the knowledge of the possible impact of race conditions on the security of web apps, Computest decided that this was a perfect topic for a master thesis.

We conducted the research at a Computest under the guidance of MSc. Daan Keuper at the R&D department. They also graciously provided us with a professional version of the advanced security testing tool called Burp Suite (Portswigger, 2018a). This was heavily used in the development and application of our toolset. Computest is a software testing company that is located in the Netherlands and has about 120 employees. It offers consultancy and training in integrated quality assurance and supplies services in the areas of performance, security and functional testing. The end goal is an improvement of software quality to ensure that applications and in-

frastructures work as well as possible (Computest, 2019). This end-goal of software quality improvement perfectly matches the general goal we have also stated at the beginning of this chapter.

1.5 Structure of the work

Next, we give an in-depth background of race conditions, web applications and the testing of software systems in chapter 2. The information should give the reader enough knowledge of the context to understand the actuality and difficulty of the problem and the importance of the solution that this thesis tries to provide. In chapter 3, the state of the art research in the detection and exploitation of race conditions in web apps is given.

Chapter 4 starts with a continuance of our earlier treatment of race conditions based on our findings. Then it addresses the creation of the first systematic method to test for race conditions in web apps. Then, in chapter 5, we discuss the requirements, design and implementation of CompuRacer. This is a software toolset we created to accompany the systematic method. In chapter 6 the effectiveness of the toolset and systematic method are evaluated based on their functionality, usability and performance.

Based on this work, the research questions as posed in section 1.2 are answered in chapter 7. Then, future work to extend on this research is enumerated in chapter 8. Last, the bibliography is listed, and the appendices, including a detailed manual of the CompuRacer toolset, finalise the thesis.

Chapter 2

Background

In the introduction, we have only briefly introduced several important concepts. More in-depth knowledge is deemed necessary for the reader to understand the reason behind different steps that we will take in the thesis. In this chapter, we set out to do this.

First, we will give a more detailed description of what is already known about race conditions in general section 2.1. Thus, how they can occur and what types of impact are expected for web applications. At the beginning of the section 4.1, we will continue this explanation of race conditions with our definitions and findings regarding race conditions in web apps. Next, in section 2.2, the essential elements that web apps most commonly consist of are explained for the reader to get a better grasp of where race conditions could occur in these web apps. Finally, a background on software testing, in general, is given in section 2.3. In this section, in line with the central questions of the thesis, we will pay special attention to how security assessments and penetration testing of web apps is performed from a black-box perspective.

2.1 Race conditions

In this research, we will focus on a family of bugs called race conditions. These bugs are caused by an unpredictable ordering of (atomic) events in which at least one sequence results in unwanted behaviour of the application. The unpredictable order is caused by any form of concurrent execution without proper synchronisation techniques. More specifically, most races are either caused by TOCTOU bugs or by read-and-then-write bugs, as Northcutt (2007) explains.

Root of the issue Most software is based on assumptions of a happens-before relation between events like creating a file before writing to it (TOCTOU), storing the settings of a user before closing the application, or reading from and writing to a database entry without any other concurrent reads or writes to this entry (read-and-then-write) (Dean and Hu, 2004). The program will often fail when these assumptions are violated. As already mentioned in the introduction, these bugs are notoriously hard to debug as they are timing related. Thus, any change to the execution order or speed of the program to pinpoint the bug could also influence or temporarily mask the bug itself. As we have shown in chapter 1, these bugs occur both in single-tier applications but also in more complex web apps (at client- and server-side).

Impact of exploitation We can divide the impact of race conditions in web apps, when there is any impact, into three main types: 'visual', 'functional' and 'security.' The overarching consequences can also be expressed financially or as reputational damage. Reputational damage or distrust by the clients is often only of concern because of the financial impact. For this thesis, we are primarily interested in race conditions that result in security issues, but to clearly be able to identify this kind of impact, we will also discuss the other types of impact:

1. **Financial** - The total impact can also be measured financially, but as this requires a lot more knowledge about the (specific) involved business processes, we do not concern ourselves with this measure in this thesis. However, as the actions of most businesses revolve around the financial Return On Investment (ROI), a financial incentive for investing in cybersecurity is very important in practice. Unfortunately, often, companies only see the value of these types of investments after a catastrophic failure. For reasons behind this largely ignorance-based issue and appropriate solutions, we refer the reader to the research on the economics of cybersecurity by Moore (2010).
2. **Visual** - Visual consequences are about any change in the visual appearance of a webpage (at the client-side) that is not desired or intended by the developers, but also does not affect the functionality or security of the application (Mutlu et al., 2014). This issue can sometimes be observed when a webpage loads without any styling because the Cascading Style Sheets (CSS) fails to download or parse in a reasonable amount of time.
3. **Functional** - Functional race condition bugs are related to any aspect of the web app not functioning as designed. This malfunctioning could lead to an exploitable security vulnerability when an attacker can influence the issues. If

not, the issues could still cause unwanted behaviour or crashes. For instance, multiple background cleanup job of a web server might race and as a consequence, leave the database in an inconsistent state. This issue does not seem to be influenceable by an attacker, but could still cause crashes in the application.

4. **Security** - If, however, the erratic behaviour can be exploited by an attacker resulting in significant impact, it can be considered a security vulnerability. Examples of this kind of impact are shown in the introduction chapter (multiple account creation and session puzzling). The impact of security vulnerabilities can be further subdivided in confidentiality, integrity and availability (CIA) (Techrepublic, 2019). We will further explain these types of security impact in section 4.2.

In section 4.1, we will continue with our findings regarding race condition types in web apps. We will mainly focus on explaining how race conditions start at interleavings of single actions in source code and how these issues propagate to a series of actions that can eventually also impact the application as a whole.

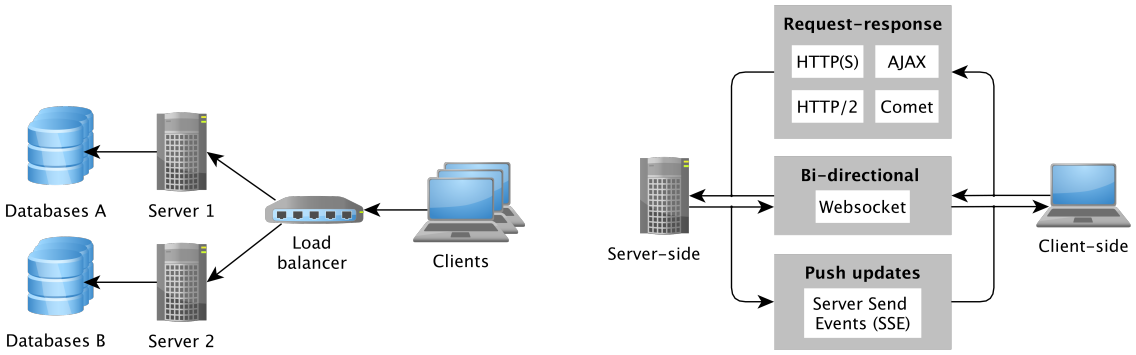
2.2 Web applications

To discover in what functional parts of a web app certain race conditions could occur and what kind of impact is expected, it is essential to have a basic knowledge of how a web app is designed. We will first elaborate on the network setup of a web app. Then, the communication techniques between client and server are discussed. Finally, the specific technologies, libraries and tools are discussed that are commonly used for developing web applications. For all aspects, we will briefly discuss the impact of certain design choices on the expected prevalence of race conditions.

2.2.1 Structure

In this section, the network setup of web apps is discussed. As already mentioned in the introduction, in the past, web applications have changed from simple single-tier (or two-tier) applications to complex multi-tier applications. For our thesis, we focus on state of the art regarding web applications, and that is why only the multi-tier web applications are discussed. Still, the foundational aspects of the web have not changed, and that is why we also refer to relatively mature and but well-established sources.

As you can see in figure 2.1a, a multi-tier web app can be divided in a client-side and a server-side. The client-side is comprised of the Hypertext Markup Language (HTML), CSS and JavaScript files rendered in a browser (Tanenbaum and Van Steen, 2007, chap. 12.1-12.2). The HTML files describe the general structure and static content of the web app, the CSS describes the look and feel of the application and the JavaScript files control the dynamic behaviour. The server side can just be a single web server, application (app) server and a database, but in every medium-to-large web app, the network structure is much more complex.



(a) The network components involved in hosting a typical web app

(b) The client-server communication techniques often used in web apps

Figure 2.1: *Web app network components and communication techniques*

Often, a load balancer distributes the traffic between multiple servers at multiple locations. The web server receives the client-requests, responds with the static content and forwards requests for dynamic content to the application server. The app server then fetches information from a database or (internal) cache, creates a response and sends it back to the client via the web server. Web apps are not limited to one database which can only scale 'vertically' by using better hardware but also scale 'horizontally' by adding more database replica's.

The purposes of multiple replicated databases are indicated by Özsu and Valduriez (2011) to be the following: 1) system availability, 2) performance, 3) scalability, and 4) diverse application requirements like for backups. The availability is increased by having more available database instances than necessary where one could take over from the other in case of a failure. For further reading with regards to novel ways to make applications scale using microservices (a separation of every functional part into its own independent subsystem), we would like to refer the reader to Dragoni et al. (2017). Additional databases can also be used to separate the data and workload evenly for more performance. Scalability is supported because these multiple replicas can be placed in different geographical areas to support a wide variety of users spread across the globe.

As both the number of components of web apps and the performance is increased, we also expect more options to be available for race conditions to occur. Within single-tier applications, traditional types of race conditions occur between multiple processes. In multiple duplicated components or separate services, these race conditions could also occur between these components during the mutual usage, editing and synchronisation of data. In chapter 4, we will further elaborate on the potential race conditions between web app components.

2.2.2 Communication

Next to the network setup of a web app, the communication protocols between client and server also differ per web app, and most web apps use a combination of techniques for different parts of functionality. From the black-box testing perspective, having in-depth knowledge about these communication techniques is very important as these are the only means by which race conditions might be triggered at the server from the client. In this thesis, only the request-response type of communication is taken into account, but in the section 8.2.1 on the future work we will also discuss race condition exploitation options using the other two techniques. In figure 2.1b, the three main types of communication are listed. We will explain these techniques below.

1. **Request-response** - the client initiates the request for data. In normal Hypertext Transfer Protocol (OSI level 7) (HTTP), a client first creates a TCP connection to the server and then (re)loads a full page via several HTTP requests (Tanenbaum and Van Steen, 2007, chap. 12.3). In contrast, in Asynchronous JavaScript And XML (AJAX) or Comet, this behaviour is changed. These techniques still rely on the creation of a TCP request but are much more versatile.

   AJAX is the name of all techniques that allow the client (via JavaScript XMLHttpRequests) to ask the server asynchronously for an update to a part of the webpage without a page-reload (Woychowsky, 2007). Comet or long-polling is a specific type of AJAX in which the asynchronous request blocks until there actually is an update (Carbou, 2019). This avoids some of the unnecessary polling by temporarily moving the initiative to the server.

2. **Push updates** - Contrary to the standard HTTP requests and AJAX and continuing in the direction of Comet, the initiative has been fully moved from the client to the server in this branch of techniques. The only responsibility of the client is to subscribe to the types of events (Server-Sent Events) she is interested in. When the server encounters an event of this type, it will update the

client until she unsubscribes. Server-Sent Events (SSE) can be sent using the Comet technique (Pohja, 2009).

Officially, SSE are part of the HTTP version 2 (HTTP/2) specification and are one of the most distinguishing new features. HTTP/2 is an update to the old HTTP version 1.0 or 1.1 (HTTP/1.x) Standard in which Transmission Control Protocol (OSI level 6) (TCP) streams can be shared between HTTP requests (speedup), binary data is transferred instead of text strings, some headers are made optional, and headers can be compressed. SSE also allows for the server to send several files like scripts, pictures or CSS alongside an HTTP response page. It already knows that the client requires them eventually and therefore avoids multiple request-response cycles to load a single page.

3. **Bi-directional** - Especially, for live chats, web-based games, media streaming, live statistics of sports or stock prices, long polling techniques or SSE might still not be sufficiently flexible. Also, the overhead per request is still significant. Improvement is found in using a real-time web protocol like WebSocket. Websockets are part of the HTML5 specification and were fully supported by most browsers at around 2011.

   It is a protocol at the same Open Systems Interconnection (model) (OSI) level as HTTP and HTTP over TLS (HTTPS) called WebSocket (OSI level 7) (WS) and WS over TLS (WSS) respectively, and therefore the initial HTTP protocol needs to be upgraded (changed) when a client wants to communicate to a server using this technology. As the final Request For Comments (RFC) by Fette and Melnikov (2011) shows, WebSockets allow for full-duplex real-time communication between the client and server side of web apps. Thus, the server, as well as the client, can push updated data to each other at any moment in time with a minimal overhead compared to HTTP.

2.2.3 Technologies

Next to the network setup and communication protocols used in a web app, we are also interested in the technologies that are currently in use to develop web apps. How these technologies support developers in creating secure web apps greatly influence the generalisability of any race condition findings in a specific web app.

When most of the web application logic is self-created by the developers, most findings of bugs will be specific to this web app. On the other end of the spectrum, we see Content Management Systems (CMSs) like the favoured Wordpress CMS that only requires minor configuration from the developers. Therefore, most findings will

be equivalent to all web apps built using these systems. The middle ground is covered by web apps supported by (elaborate) frameworks like Python Django.

The listing of commonly used web app technologies in figure 2.2 is based on the popularity of the languages and techniques as indicated by the global survey of Stack overflow (2018), the popularity ranking of databases by DB-Engines (2018), the suggestions of blogs by web development companies like B. and V. (2018) of RubyGarage and Lozinsky (2017) of WebiNerds, and the advice of security testers at Computest. Both SQL and NoSQL databases are included.

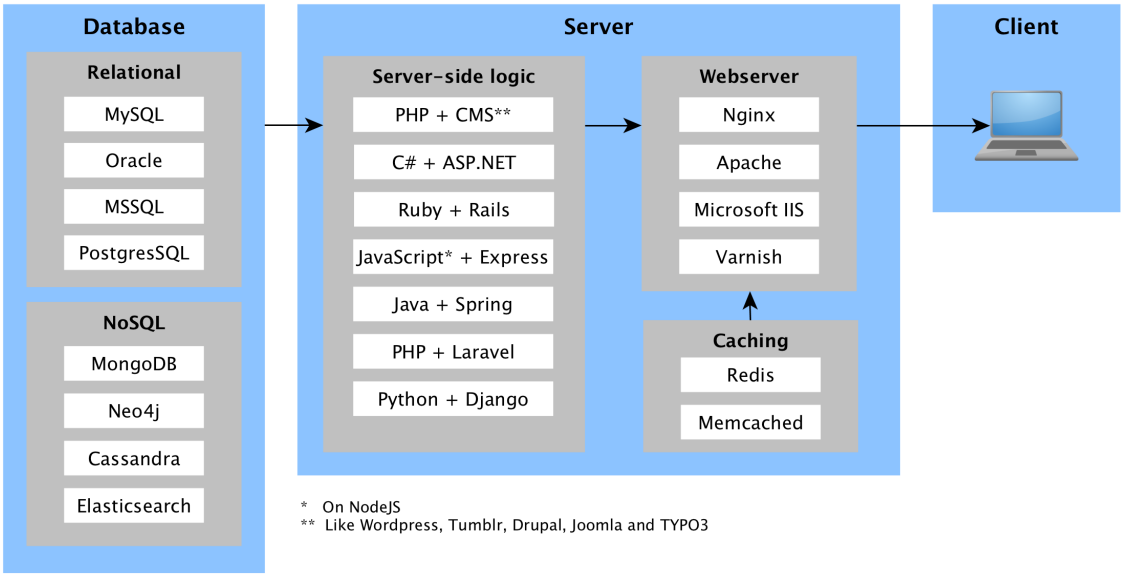


Figure 2.2: *A listing of some of the most common web app structure and software components.*

As race conditions are explained above to require parallelly accessed storage, the way the database copes with parallel transactions greatly influences the existence and exploitability of race conditions in the web app as a whole. The way a database can deal with this is described in its ACID compliance. The ACIDity of databases is as (Özsu and Valduriez, 2011, pp 21-23) explains, the extent to which different parallel transactions are capable of influencing each other. As you might expect, complying to the ACID standards becomes more complicated when databases are also allowed to be (partially) replicated and distributed geographically. Especially now that more NoSQL databases like MongoDB become at least document-level ACID compliant and stable at massive scales, as Vial (2018) shows, there is a move from traditional relational databases to NoSQL databases. Still, both types are used in practice and therefore are included in the figure.

Redis is also a NoSQL database, but it is more commonly used as a key-value caching solution and is therefore placed in this block together with the alternative

called Memcached. The server-side logic is shown as a programming language + web development framework because these components are often inseparable. However, for some languages, more framework options are available like the Pylons Project (2019) which is web framework for Python or the Meteor framework for JavaScript (Vanian, 2019).

The reader should now have a more in-depth idea of how web apps are developed nowadays. In this thesis, we will not focus on researching race conditions in specific technologies, but rather on the user-faced functionality (of any technology) that a tester can actually influence from a black-box perspective. However, during the selection of web apps for practical testing in section 6.3, we have tried to find a set of web apps that, when combined, use most of the technologies discussed here.

2.3 Software testing

Companies that build web app software are focused on delivering fast, feature-packed and high-quality software. For about 20 years, companies are no longer using static 'waterfall' methods of design, development and testing, but are using various methods of Agile development and Continuous Integration (CI) (Kurapati et al., 2012). This results in a development process in which every day (or week) a slightly improved (nightly) version of the system is built and tested.

It is clear that the methods of testing have considerably evolved, but the statement by Hetzel (1984) still holds: *"Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results."*. Next to this, the goal of the most mature type of testing is defined by Beizer (2003) to be to create a serious mental discipline to ensure software quality. This idea is a significant improvement over the novice mindset: 'debug it until it compiles' and is a more foundational goal than: 'test software to reduce failure-related risks'. At the same time, it is something that is known to be very hard to realise in practice.

As already mentioned in the introduction, the topic of this thesis is one small part of this overall goal of ensuring software quality. In this section, we will show where black-box security testing for race conditions in web apps could integrate with existing security testing. To do this, first, four major dimensions to software testing are explained. After this, zoom into how security testing is performed, and finally, we try to pinpoint the location along each of these axes of our topic of black-box testing for race conditions.

2.3.1 Essential software testing dimensions

In this section, we will discuss four major dimensions (or axis) of software testing, as shown in figure 2.3. We would like to stress that not every option along every axis is compatible with every option along a different axis. That is why, for every axis, the compatibility with the other axis is discussed. For instance, static performance testing or black-box functional testing are not feasible options.

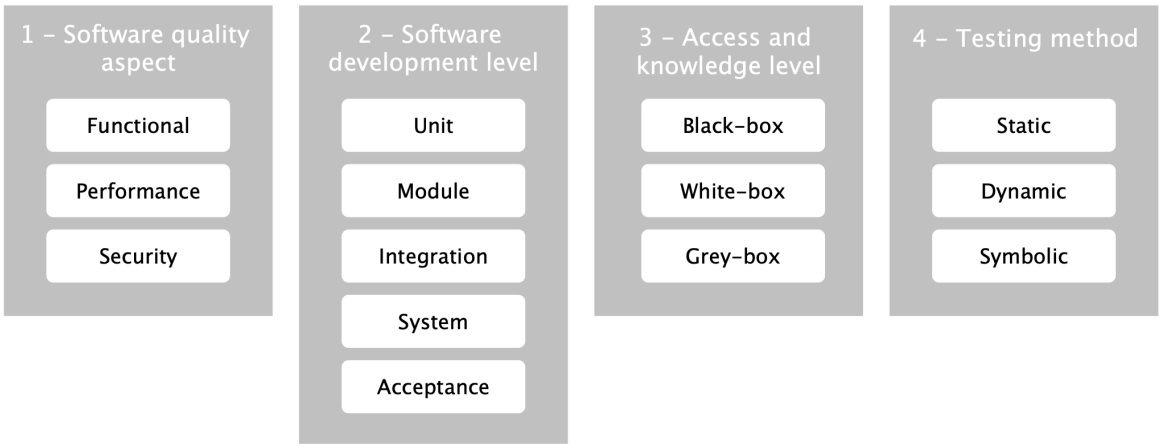


Figure 2.3: *The figure shows all aspects of the four major dimensions of software testing that are discussed in this section.*

The first axis is based on the three different aspects of software quality that could be tested: *functional*, *performance* or *security* (Moilanen et al., 2015). You could also consider adding the legal requirements to software quality as a fourth aspect, but we do not know of any legal requirements to software quality that cannot be assigned to either of the three aspects already mentioned.

- **Functional** - Whether every aspect of the system is implemented correctly according to the requirements. This results in a system that works as expected during a typical run.
- **Performance** - Whether the system is robust enough to be able to reliably support the expected (peak) load without failure.
- **Security** - Whether the risk that the system can be made to function in a not-intended way resulting in a negative impact is of an acceptable level.

Interestingly, improvements to any software based on functional and performance tests can also improve the security of the application as functional bugs or performance issues could be misused by an attacker. However, this does not mean that the specific security aspect of software testing can be omitted. One reason for this

is that the testers need to adopt an adversary mindset for most security testers. In other words, he needs to think like a hacker who wants to get in using any possible method (outside of the exact specification and design) instead of a developer who tends to only check conventional methods.

Although functional and performance tests do require manual labour to set up, after this phase, it can often be run automatically. For security testing, we can automatically check for most known vulnerabilities in software and dependencies, but these checks are neither durable nor conclusive. On the one hand, this is because new security bugs are found, and any listing of potential vulnerabilities quickly becomes outdated. It is also due to the very versatile and creative interaction with the system that is required to perform thorough security testing. Any determined hacker would perform such manual labour as well, and he would only need one way in. On the contrary, a security tester often tries to discover every way in – the whole attack surface.

The second axis corresponds to the five software development levels where testing could take place: *unit*, *module*, i, *system* and *acceptance* as (Ammann and Offutt, 2016, chap. 1.1) show. One might be tempted to include regression testing as well, but this is not an additional level. It is the act of comparing the software product as a whole between different versions as a part of maintenance. The in-depth explanation of every level is shown below and is adapted from (Ammann and Offutt, 2016, chap. 1.1):

- **Unit testing** - This often targets the smallest amount of code at the level of functions and methods. The method is tested with different kinds of expected and not expected input data to confirm a correct implementation.
- **Module testing** - This level targets a combination of methods and functions with the same purpose. In Object-Oriented (OO) programming, this is a class. Else, it could refer to a single file. The test assesses whether the methods within the class work together to abide by the detailed design of the application properly.
- **Integration testing** - This encompasses testing a combination of classes that form a single subsystem of an application. Based on the subsystem design, the tester assesses whether the structure and behaviour of this part match the design.
- **System testing** - This level targets the assembly of the different subsystems and their connectors into the complete system. The tester evaluates whether

the system matches the requirements to the system as a whole. This is the 'architectural design'.

- **Acceptance testing** - This highest level of testing is also about the complete system. However, now, the tester determines whether the actor-needs as described in the requirements are fulfilled. It requires users with the appropriate domain knowledge as it evaluates whether the system does what the users themselves want.

Both the functional and the security aspects of software quality, as mentioned above, can be tested at all levels. Performance tests, however, are often based on specific user-activity based requirements and can, therefore, only be performed as system or acceptance testing.

The third axis is the access and knowledge level of the tester regarding the internal working of the system under test: *black-box*, *white-box* or *grey-box* (Acharya and Pandya, 2012). The exact definition of these terms does not seem to be used consistently throughout literature. We have used the most common way to explain them. In most cases, regardless of this level, the tester does know the user-requirements of system or how it should behave.

1. **Black-box** - The tester known next to nothing about the internal workings of the system (like statements and branches) and cannot view the source code. He does know how the system should behave according to its requirements, he can run the system, and interact with it from the outside to evaluate the consistency of this expected behaviour. In this case, the tester does not have significantly more information than a typical user or attacker of a system has. Thus, a manual and a compiled version of a piece of software or the Internet Protocol (IP) addresses of the web app servers.
2. **White-box** - Next to the abilities and knowledge of the black-box tester, this tester also has full knowledge of the design and internal workings of the system and can view the source code and any related documentation. The test can, therefore, be executed from the inside of the application either by a developer or an independent tester. The advantage of this technique is that a tester can reliably test almost all execution paths of a system. This allows for the most thorough testing.
3. **Grey-box** - This is a hybrid form of the types described above. As Acharya and Pandya (2012) describes, in this case, the tester knows some things about the internal workings of the system and might have access to parts of the

documentation or source code. Both the advantages and the disadvantages of the other types could apply to this technique dependant on the specific case.

Interestingly, in contrast to the other work by Acharya and Pandya cited earlier, (Ammann and Offutt, 2016, chap. 1.4) claim that the test distinctions made above have become obsolete as several new types of abstract software models are a more powerful way to describe this aspect of testing. However, to the best of our knowledge, these distinctions are still in common use, and we also deem them sufficient for explaining the topic of this thesis. That is why we have decided to stick by these terms.

Regarding the first axis, functional tests are often executed from a white-box perspective, while performance and security tests can be performed both from a black- or grey-box perspective. Any part of the second axis, as described before, can be tested using the white-box approach. Contrarily, only system and acceptance testing seem to be possible using a black-box method. A grey-box approach might be able to target all levels depending on the specific case.

The fourth axis is the testing method: *static*, *symbolic* or *dynamic* (Godefroid et al., 2008). This indicates how the software is analysed to perform the tests. This analysis could be purely based on the source code itself, or it could require the tester to execute the software.

1. **Static** - This method could be considered the oldest form of testing. It requires access to the source code of the application and at least verifies whether the code abides by the syntax of the language itself. Any other checks that do not require knowledge of the value of variables or other execution-related behaviours can also be considered static testing. Often, these kinds of checkers are already built into Integrated Development Environments (IDEs) to quickly spot errors while developing an application.
2. **Dynamic** - This method runs the application using predefined or automatically generated input and tries to discover whether all reachable program states are valid and do not result in errors. Contrary to static analysis, it requires a certain knowledge of valid inputs and outputs of the program, which is not always desirable. It does, however, have the power to show that a system not only works in theory (static) but also in practice. As the system must be successfully compiled and executed before dynamic testing is possible, a dynamic test following and complementing a static test is a common procedure.

3. **Symbolic** - Just like dynamic testing, this method tries to infer the run-time behaviour of the application. However, it does not do this by simply running the application itself, but by also assigning a symbolic value to each variable. Based on the requirements of the system, it can then calculate what ranges of different variable values will occur during program execution. It does not require knowledge of valid inputs like dynamic testing does. Unfortunately, exhaustive testing in large systems is often not possible due to 'path explosion'. This is the name for an exponentially large number of paths through the program, or states, due to branches (like if, while, for, fork statements) that have to be explored. Concolic testing tries to resolve this issue by combining symbolic analysis with concrete (dynamic) testing using actual inputs to a running program to quickly rule out (prune) unfeasible paths and thereby resulting in a more manageable state-space (Sen et al., 2005).

Regarding the first and second axes, we estimate that they are compatible with all testing methods described above. However, with regards to the third axis, only the dynamic method seems to apply to black- and grey-box testing. Static and symbolic testing requires full access to the source code that is only possible when the access and knowledge of the tester are of the white-box level.

2.3.2 Security testing

As mentioned before, only the security aspect of software quality is targetted in the thesis. That is why we will explain particular types of security testing and some available tools in more detail.

Security testing types Security testing can be divided into three main types: penetration testing, vulnerability assessments and security auditing. The first type is similar to what a malicious hacker would do. Without much knowledge about the internal workings of the completed system (black- or grey-box), the tester tries to find a way into the application and continue as far as possible. The final target is an essential asset like a particular company document, passwords, or other secret information.

During vulnerability assessments, however, the tester tries to cover the whole attack surface of an application without going further than a proof of concept for every vulnerability found. This is mostly done from a black-box or grey-box perspective and is a breadth-first test compared to the depth-first penetration test. After performing a

vulnerability assessment, the findings can also be used for a subsequent penetration test as Goel and Mehtre (2015) indicate.

Security auditing is a white-box testing method in which the security tester has access to the entire source code of the running system while, just like with a vulnerability assessment, he tries to cover the whole attack surface.

Security testing tools Although most security testing is done manually, there are still a large number of tools available to support the testing. We will not go into tools that are designed to exploit a specific vulnerability. Instead, two general toolsets will be discussed that support web app security testing. The first tool is the Zed Attack Proxy (ZAP) by OWASP (2018). It is free, open-source and actively maintained by hundreds of developers. The second tool is called Burp Suite and is paid proprietary software made by Portswigger (2018a).

Both tools act as a Man-in-the-Middle (MitM) proxy to capture all internet traffic of interest and then allow for active and passive scanning to build a sitemap (graph of functionality: attack surface) of the web app. Next to this, HTTP requests can be replayed with (automated) alterations to headers, tokens, passwords and other data. Finally, they can both actively test the web app for common security bugs that can be found in the testing lists mentioned before.

2.3.3 Location of race condition testing

Based on the definition of the four dimensions of software testing, we can now locate the testing goal of this thesis on these axes. For web apps, we test 1) the security aspect (race conditions) 2) on the level of system and acceptance (and regression) 3) from a black-box perspective 4) using dynamic analysis. Next to this, based on the security testing types as defined above, black-box race condition testing seems to fit best into vulnerability assessments. Finally, regarding the available general testing tools, both are applicable, but we will only use the Burp Suite.

In this chapter, we have defined clearly what the race conditions entail, how a web app is designed, and what encompasses software testing. Hopefully, this has resulted in the essential background knowledge to understand the further contents and decisions put forward in the thesis. In the next chapter, we will describe state of the art with regards to security testing for race conditions in web apps.

Chapter 3

State of the art

The most crucial requirement to be able to place the research and its added value in context is by looking at state of the art. State of the art comprises all research that looks at the testing for race conditions regarding the detection, repair and exploitation of these vulnerabilities. We can differentiate between client-side and server-side races. For every category, we evaluate the maturity of the state of the art research regarding race condition testing. Next to this, we discuss the differences and similarities between the existing work and the thesis.

3.1 Client-side race conditions

Although this research is not about the functional or visual testing of web applications, for several reasons, research about client-side race conditions is still included. As the field of race condition testing in web apps is dominated by tools to detect the races that happen at the client-side, and this is still a very closely related topic, it is still considered a valuable source from which to learn.

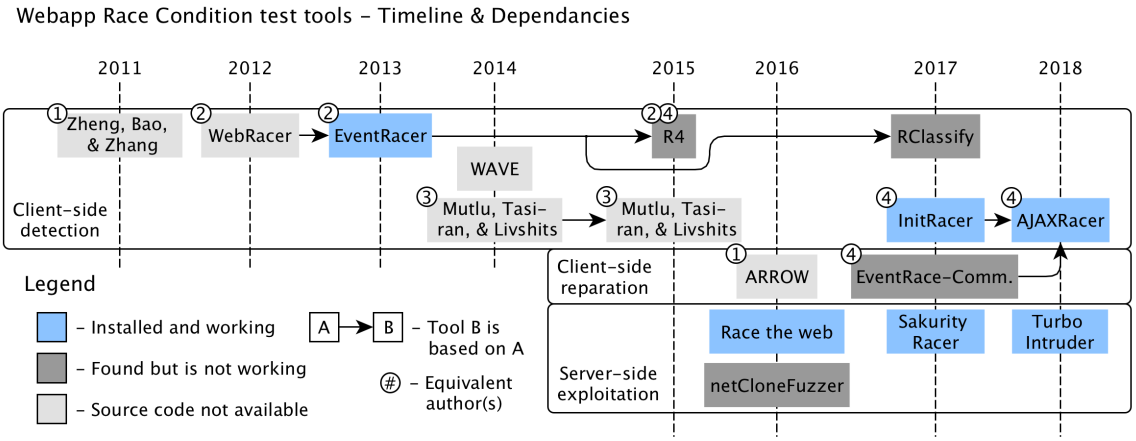


Figure 3.1: *An overview of a number of race condition testing tools over time and their relations.*

In figure 3.1, we have listed all tools for client- and server-side detection, reparation and exploitation that were considered. When a tool was not named, we have listed the authors instead. Most tools target the client-side detection or reparation of races, and only four tools try to exploit races at the server-side. We will present the exploitation tools Sakurity racer, Race the web, Turbo Intruder and netCloneFuzzer in the next section on server-side race conditions. In order to properly understand the workings of the client-side tools, and to what extent they target exploitable race conditions, we tried to acquire the source code.

For every tool displayed in blue and dark grey, this proved to be successful, and we include the links to the sources in appendix A. For the dark grey tools, however, we either did not manage to successfully install and run the tool after several days or weeks of effort, or we did not use it for other reasons. R4 was too difficult to set up due to software incompatibilities and old age. RClassify was never meant to be open-source, was scarcely documented and even after significant effort, it did not fully work in the end. We decided not to use **ARROW** by Wang et al. (2016) and **EventRace-Commander** by Adamsen et al. (2017a), because they are race-repair tools and therefore not comparable to the other tools.

For the light grey tools, acquiring the source code was not possible. We discuss these tools below.

- **Zheng et al. (2011)** - The static JavaScript analysis tool by these authors looks at atomicity violation problems in AJAX. It requires access to the server-side Hypertext Preprocessor (PHP) code to extract all JavaScript code. Based on generated call and control flow graphs, it tries to infer possible race conditions. The researcher has not open-sourced the tool.

- **WebRacer** by Petrov et al. (2012) - This is the first dynamic race detector for web apps. The research does not point to any sources, and this is also confirmed by a Computer Science reproducibility research led by Collberg (2014) in which the authors of the tool respond that the sources are not available. Its techniques will be discussed later in comparison to its successor: EventRacer.
- **WAVE** by Hong et al. (2014) - This performs simple event handler inference of client-side JavaScript to detect races. It, however, cannot point to the primary cause of race conditions like, for instance, R4 does. The paper contains a link to the sources, but these are no longer available at this website.
- **Mutlu et al. (2014)** - The tool by these authors focuses entirely on visually observable race conditions. By randomly changing the order and adding delay to XMLHttpRequest (XHR) requests and responses, the tool tries to trigger race conditions at the client-side. The authors do not mention the sources of their tools.
- **Mutlu et al. (2015)** - In an extension to their research as cited above, these authors use the same delay techniques, but the focus is shifted entirely to finding race conditions that affect the persistent client state. The paper provides a link to the source code, but we cannot resolve this link.

Therefore, we are left with three detection tools that we could successfully use, and they will be explained more in-depth: EventRacer, InitRacer, AJAXRacer. Next to this, we will explain R4 and RClassify to more accurately show the improvements of the race condition detection tools over time.

- **EventRacer** by Petrov et al. (2012), is based on WebRacer which uses an instrumented version of the WebKit browser to track and record most Document Object Model (DOM) load/update and JavaScript events that happen during the (automated) exploration of a webpage. This recording is then used to create the happens-before graph of events. Events that access similar memory parts and are not in a specific order (thus not included in the happens-before graph) are expected to be prone to race conditions. By using heuristics about where races can and cannot take place at the client, EventRacer improves on WebRacer and filters out a significant number of false positives.
- **R4** by Jensen et al. (2015), continues where EventRacer left of by including a harmfulness classification to every race condition. It can differentiate between temporal or non-functional race conditions and race conditions that result in changes to JavaScript variables, session information or cookies. Next to this, it tries to improve its true positive rate by exploring new paths in the web app exploration that were not triggered during the (automated) exploration.

- **RClassify** by Zhang and Wang (2017), also builds on the results of EventRacer but takes every different direction than R4 in improving on this. It requires the tester to have a local version of the web app client-side code and then adds additional instrumentation to both the JavaScript and HTML files. This instrumentation will be used to approximately replay the race condition prone sections that were found by EventRacer. By replaying the run with and without the specific race, it verifies whether the race condition impacts the web app client state (JavaScript variables, session information or cookies).
- **InitRacer** by Adamsen et al. (2017b), is an entirely new kind of tool in multiple ways. Instead of changing the browser as EventRacer and derivatives do, it injects instrumentation in every script and HTML file that the website loads using a MitM proxy by Mitmproxy (2018). This instrumentation makes sure that the tool can record every event and subsequently, that it can arbitrarily trigger these events using Protractor. Protractor is a Selenium-like end to end test framework for Angular by Google (2018). Another unique feature is that it specifically looks at races that happen during the first couple of seconds when loading (initialising) a webpage.
- **AJAXRacer** by Adamsen et al. (2018), is made by the same author as InitRacer but targets a very different part of the web app functionality. As the name indicates, it focusses on race conditions in AJAX communication. AJAX is the name for all client-side technologies that enable asynchronous communication between client and server. As this involves requests with non-deterministic callback timing and ordering from the server, this could result in races. Just like InitRacer, it uses a proxy and Protractor browser automation to inject instrumentation and control the browser. Unlike InitRacer, it requires the tester or an automated spider to run through the web app in order to obtain an event-graph of AJAX-requests and responses. Then, it uses this graph to create and trigger pairs of AJAX-events to see whether this has a visual impact on the webpage.

Conclusions Ten tools were discussed, of which five tools could successfully be installed and run. Overall, based on the results in the related papers, this research does seem to find and report visual or functional race conditions in every part of the client-side due to internal-, user-, or communication-events like AJAX. However, it fails to look at the security implications of these races. Our research differs from these tools as it does not target client-side race conditions and especially looks at the security implications of race conditions. Next to this, our research does not only result in a toolset that supports security testers, but it also results in a full

methodology that guides the user in the security test itself. This method helps to discover where the race conditions occur and how they can be triggered.

3.2 Detection of server-side race conditions

Contrary to the client-side tools, some research does focus on the security impact of race conditions in web apps. As such, they are a great starting point for the exploration of systematic testing of this security issue.

3.2.1 Published work

Only four works were found that focus on server-side race conditions. They are discussed below. Conclusions regarding the impact of their research on our work are discussed at the end of the section:

- The work by Paleari et al. (2008) seems to be the first research in this direction. They note that concurrency is one of the oldest security problems, because it is both hard to detect and because *"(..) a typical programmer does not conceive his web application as a multi-threaded or multi-process entity"*. A lot has changed in web development since this research was conducted, but these statements still hold. The research focuses on race conditions in the interactions between a multi-threaded web app and the underlying DataBase Management System (DBMS).

  They test the following popular Content Management System (CMS): Joomla, Wordpress and phpBB. The test does not include the popular Drupal CMS. The test itself encompasses capturing the requests from the application to the database and forwarding these in parallel to the database. The parallel requests should result in race conditions at the database. They especially look for security relevant findings, and this resulted in the following insights.

  In the applications that they tested, they were able to register multiple users with the same name, perform more login attempts, cast more votes, and send more messages than was officially allowed due to races in the logic that checked for these conditions. Obviously, having more login attempts makes brute forcing attacks easier, but the other findings are only a security issue when it results in a (financial) advantage like multiple one-time bonuses, results in an unexpected state of the application or helps in certain privilege escalation or DoS attacks.

- The second work we would like to discuss is a section of the book by (Stuttard and Pinto, 2011, pp. 426-427). Chapter 11 in this book is about attacking the application logic of web apps and ends with a discussion of a number of previously executed attacks that were done by the authors. The 12th example is discussed in our introductory chapter in section 1.1.2. This example ends with a number of steps that a tester or hacker can take in order to perform a black-box test for race conditions.

  They state that this kind of test is not straightforward and that *"It should be regarded as a specialised undertaking, probably necessary only in the most security-critical of applications"*[1]. We agree that, currently, it is a specialised undertaking, but when a better testing methodology and supporting toolset is created, this issue should be partially alleviated. The authors then list several hacking steps that boil down to:

  1. **Functionality** - Select web app parts with the most important functionality like the login, password change, or funds transfer.
  2. **Requests**- Select a limited number of requests to perform one action within the functionality and define a means to verify that the attack is successful.
  3. **Attack** - Select high-spec machines and send these requests in parallel on behalf of different users.
  4. **Evaluation** - Evaluate the results and filter out the potentially the large number of false-positives as this load-test type of attack could result in a lot of non-related anomalies.

  Together with tips and hacking steps found in other sources, we will use these steps as a start for our systematic methodology for testing race conditions in web apps (see section 4.2).

- The third research is executed by Zheng and Zhang (2012) who perform static analysis of the interaction between a web server and its external resources from a white-box perspective in order to find race conditions. They state that most other research only seems to focus on in-memory race conditions, but fail to also look at resource contention in external. The research explicitly refers back to the research by Paleari et al. discussed above and distanced itself from it in several ways. It is similar in that both look at calls from PHP scripts to external resources, but the work by Paleari et al. is dynamic in nature and

[1] In the remainder of this chapter, we will show that this issue is far more prevalent and has more impact than these authors suspect.

therefore requires concrete database interaction traces to work with. It also does not look at the program semantics. This research converts the PHP code to C and has developed a *"(..) context- and path-sensitive interprocedural static analysis to automatically detect atomicity violations on shared external resources in PHP code"*. They found 113 errors in real web applications, of which some are security issues with financial impact.

- The fourth research is executed by Billes et al. (2017) is about the automatic detection of race conditions in collaborative web applications like Google Docs. As these applications are meant to be used with multiple users at the same time, race conditions between updates to the same data are likely to occur and can result in diminished user experience. That is why they set out to create a black-box visual analysis tool that first learns potential behaviour from recorded user interactions and then replays these sequences to find conflicts automatically. They could successfully find several functional issues in collaborative web applications. The approach seems similar to what Mutlu et al. (2014) have performed, but instead of replaying actions using recorded XHR requests, they record and replay user interaction using the Selenium WebDriver. This test method is also comparable to how Adamsen et al. (2017b) try to find client-side race conditions.

3.2.2 Articles and blogs

Next to this research, we found multiple blogs of primarily security researchers and testers explaining certain race condition exploitation techniques. Sometimes, they even include a proof of concept tool. As findings written in blogs cannot be trusted with the same confidence as academic and peer-reviewed papers, these sources were only included based on the following requirements.

The blog had to be written by a recognised security specialist or be published by an established security institution or company. Next to this, the focus of the blog should be descriptive of the tests that they conducted and easily verifiable. The latter requirement can either be fulfilled by the availability of a proof of concept tool, or a proof that developers of a vulnerable application acknowledged the issue. Six blogs that seem to abide by these requirements and display the most significant insights and findings were selected, and we discuss them in chronological order.

SANS security blog In the SANS security laboratory: Methods of Attack Series by Northcutt (2007), one entry is about race conditions. He describes races as *"(..) that small window of time between when a security control is applied and when the service is used"*. This is similar to the TOCTOU type of race condition as described before. However, the main focus of this blog is not on web apps but native low-level languages like C++. So, unless the developers of a web app have also written the server-side in C++, this approach is not directly comparable to our research.

Defuse security blog Another blog about practical race condition vulnerabilities in web apps by Defuse Security (2011) is much more similar to our research. The proof of concept target is a multi-process bank account web app written in PHP and run on an Apache server. It contains a typical TOCTOU bug as the database is first queried to check the balance of the user, and if allowed, it withdraws the desired amount. Many concurrent requests are shown to withdraw more money than originally available: an exploitable race condition. Instead of using transactions or *SELECT FOR UPDATE* Structured Query Language (SQL) methods to avoid these races, the author proposes to use System V-like semaphores or use locking files to synchronise the requests within the PHP language.

McAfee security blog We also look at a McAfee blog by Pandey (2016). Interestingly, he also notices that security tests often omit race conditions: *"The general consensus is that race-condition attacks are unreliable and cannot be identified using the black-box/grey-box approach."*. At the same time, they conclude that modern tools have a real potential to identify race condition from the tester perspective. Similar to the banking test app, a web app is written in Java with Servlets (framework for Java web development) on Apache. Using the Intruder tool in Burp, they concurrently replayed the request to transfer money between accounts for 25 times and this resulted in the source account having a negative balance.

Independent hacker blog Another blog written by Franjković (2016) is dedicated to *"(..) raise awareness about race condition attacks in both developers and security people (..)"*. This bug hunter was able to find and report all kinds of bugs related to race conditions. He found a race in the voucher redeem functionality in Facebook, Mega and DigitalOcean that allowed for redeeming the vouchers multiple times. Another exciting part of the blog is the bugfix for a race condition in Keybase invites-system that created a new race condition bug. The first bug made it possible to send out more than the allowed single invite, but the fix allowed for using a single

invite multiple times. Both resulted in the same issue of allowing a user to invite more users than he is permitted to invite.

The blog contains a link to writeups of all issues, but no proof of concept tool is included to verify the results. Finally, the author encountered another fascinating and complex race condition bug in Facebook account registration, where he could confirm an email address that he did not own. After creating an account, but not yet verifying his email, he sent several parallel requests to change his email address to both the email under attack and his own. The requests would trigger the sending of a new email-address-verification email to his original address, but this would often contain the token to validate the email address under attack (instead of his own).

Lightning security hacker blog The next blog is written by Cable (2017), a hacker at Lightning Security. This is the only blog that contains a live example of a race condition. The example, written in NodeJS, simulates the transfer of money between banking accounts with a TOCTOU bug. It allows for transferring more money than available by quickly clicking the send button multiple times. Next to this, he discusses some real race condition hacks like a Bitcoin transfer site where he could act similar to the banking example and a HackerOne report on single-use vouchers that could be redeemed multiple times. The suggested solution is using locks in the database or back-end language. Regarding the detection of race conditions he concludes that *"(..) you should look for race conditions whenever a one-time action occurs, whether sending money, redeeming coupons, or casting a vote."*

Second independent hacker blog The last blog we would like to discuss is written by professional bug hunter Jadon (2018). It uses the examples from the McAfee blog and then shows how he used Burp to trigger a race condition as well, but now on a real web app. This web app provided free console (command-line interface) service, but this was limited to two consoles per user. By using 100+ intruder requests to add consoles while he also removed consoles manually, he was able to get three consoles. He concludes with the suggestion that a tester should make changes to a source while sending concurrent requests as *"(..) there are higher chances you get it executed"*.

Conclusions Based on the published research regarding race conditions, we see that in the first and third work, static and dynamic white-box analysis of race conditions in external resource access of PHP scripts can yield valuable results. These

works are more similar to our purposes than the race condition tools for single-tier applications that were described in the introduction. In fact, we see no reasons why they could not become a part of the white-box aspect of system testing or security audits of source code in the future. However, our work is focused on black-box testing of web applications and any source code analysis is not possible. Both techniques, when used together, could improve the security test as a whole, but in and of itself, this work cannot use any of their methods.

Regarding the second work, as said before, the concrete 'hacking' steps for black-box race condition testing will be used as a starting point for our systematic methodology. Finally, regarding the fourth work, this is the most similar to our research. The differences lie in the fact that they created an automatic tool for testing for race conditions in collaborative web applications only, where we actually have created a systematic method that educates the tester himself to find all kinds of race conditions in web apps using our toolset. Next to this, their research only targets on functional race conditions where we primarily target intentionally adverse attacker behaviour that impacts the security of applications. That is why their toolset does not seem to be specifically tailored for triggering race conditions using high-speed parallel requests and might only be usable when the race window is rather high.

Based on the six security blogs, we can make the following conclusions. First, race conditions in web apps are not limited to a single language, framework, or server type but occur across the whole spectrum and can often be feasibly exploited. Next to this, most sources admit that the awareness of the widespread existence and consequences of these bugs is significantly below par. Furthermore, The primary bugs that are found in practice are vouchers, links or functionality that the tester can use more times than allowed. These bugs could cause direct financial damage, but do not interfere with the confidentiality, integrity, or availability of the web app.

Sometimes a more sophisticated race bug is found that requires several steps, but could actually in a confidentiality or integrity infringement. The first type could be considered the low-hanging fruit regarding race conditions, and the more complex variant the hard-to-reach fruit. It is clear from these blogs that none of these types is tested for systematically either by developers or security testers. Not only that, but the sources also confirm that there is no systematic method available to them regarding the testing for race conditions.

Based on these sources, the thesis tries to improve on the current situation in two ways. Primarily, by creating the first effort towards a systematic security testing method for race condition vulnerabilities in web apps and also, by creating an elaborate tool to support this.

3.2.3 Open source tools

Next, four open source tools will be discussed that target general server-side race condition bugs in web apps. The research behind these tools is the most closely related to this thesis.

- **Turbo Intruder** by Kettle (2019) is an extension for the popular security tester tool called Burp Suite. It is created during the execution of this research and therefore, it was initially not considered. The primary goal of the tool is to be able to send brute force requests as fast as possible. Although the developers have not explicitly made the tool for testing race conditions, its versatility still makes this possible.

  It is written in Java and uses a self-made HTTP stack that can send requests very quickly. It can use all the advantages of Burp regarding request gathering and the viewing of results. In this research, we have considered version 1.0.9 (latest). This version was released at 29-03-2019.

- **Sakurity Racer** by Sakurity (2017), is a Chrome browser plugin and intercepting proxy written in JavaScript and NodeJS made by researchers at Sakurity. According to the designers, it is applicable for all simple web app actions that the user is supposed to do a limited amount of times. For instance: performing financial or trade transfers and using vouchers and discount codes.

  When enabled, the plugin will block and forward all whitelisted HTTP requests of the web app in the active tab for three seconds. The extension forwards the requests to a NodeJS which will concurrently repeat every request five times (default value). The tester must manually verify the possible races. In this research, we have considered the latest version of this software. It has no version number attached, but it is last updated at 22-09-2017.

- **Race the web** by Hnatiw (2016), is an application written in Go, is presented in the Hackfest conference of 2016 and is written about in this blog by Security Compass (2016). Contrary to the Sakurity Racer, it is not able to gather interesting requests by recording the actions of the tester but requires him to supply the HTTP requests manually. After setting the number of concurrent requests it has to perform, it can be started. The tester can also set up and execute a test via API calls to the included server, and therefore, the whole process can be integrated more easily into other automated tests.

  Where Sakurity Racer requires the tester to look at the results manually, this tool automatically tries to match all equivalent responses to makes anomalies more evident. Whether the races have caused these anomalies instead of

some randomness, an overloaded server or other bugs must still be looked into manually. In this research, we have considered version 2.0.1 (latest). This version was released at 3-10-2017.

- **netCloneFuzzer** by Jans (2016), presents itself as a website race-condition tester on live HTTP and HTTPS events for Windows 7 and 10. Just like Sakurity Racer, it acts as an intercepting proxy, but it also has a simple Graphical User Interface (GUI). It is a .NET application written in C# which is based on the HTTP proxy server library called Fiddler.

  By using the GUI, intercepted messages can be manually changed or updated based on regular expression rules. netCloneFuzzer will hold the HTTP requests until they are prepared and then sends them all at once. Unlike Race the web, it supports no grouping of response messages, and evaluation must be done manually. In this research, we have considered the latest version of this software. It has no version number attached, but it is last updated at 09-08-2017.

3.2.4 Testing the open source tools

Installing tools As a part of finding out to what extent server-side race condition testing tools like Sakurity racer, Race the web, Turbo Intruder and netCloneFuzzer already fulfil the requirements of effectively exploiting Server-Side Race Conditions (SSRCs), the tools have been installed and set up. Unfortunately, the developers of netCloneFuzzer have only tested it on the Windows 7 and 10 operating systems, and on macOS, it currently crashes with a Signal Segmentation Violation (SIGSEGV) on startup. Therefore, it is excluded from this test. As Sakurity Racer does not group the response output in any way, we have made an effort to add a table of HTTP response codes, message lengths and contents to the output. Next to this, an HTTP response code counter is added to spot anomalies faster.

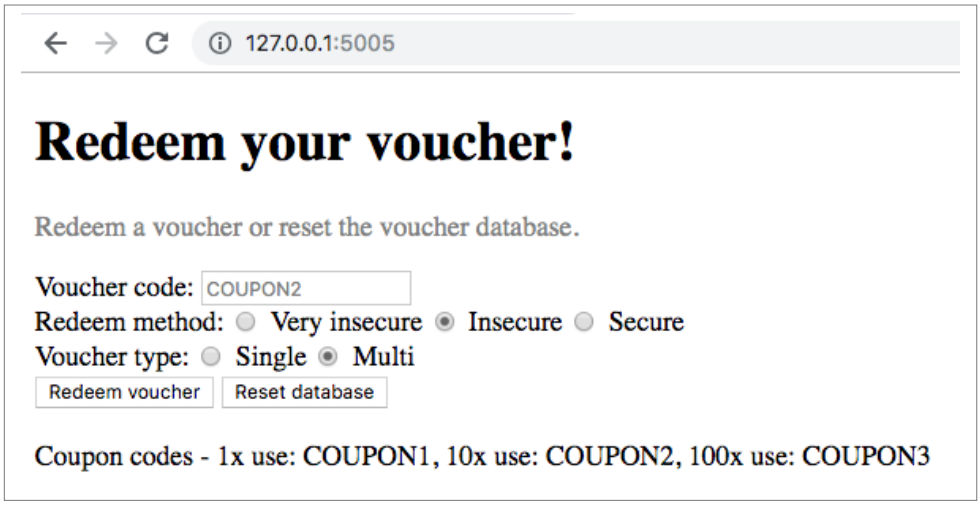


Figure 3.2: *A view on the Flask testing web app in a browser.*

Development of vulnerable web app In order to test their functionality, a deliberately insecure voucher-redeem web app is written in Python with the Flask framework and uWSGI on an NGINX server with a MariaDB database. The web app sources can be found on the public GitHub page of Computest[2] under the folder `TestWebAppVouchers`. In this web app, we spawn multiple processes to handle requests concurrently.

It supports redeeming two types of vouchers at three security levels. These options can be selected using the radio buttons as visible in figure 3.2. The 'Redeem voucher' button is used to redeem a voucher according to the selected settings. The 'Reset database' button will reset all voucher usages. The grey text-area is used to communicate actions and responses to the user. For a redeemed voucher, it shows the redeem-time at the application server and the number of vouchers left (including the just-redeemed one). We explain the three security modes below:

1. **Secure** - The first option does not contain a race condition. It is made fully thread-safe by using a single database transaction to check whether the voucher is unused and if so, removes it from the database and returns a success code.
2. **Insecure** - The second option contains a race condition that is difficult to exploit. It uses queries in two separate database transactions to create a TOCTOU race condition.
3. **Very insecure** - The third option contains a race condition that is easy to exploit. It works similar to the *Insecure* option, but also sleeps for 3 seconds between the two transactions to make the race almost certain.

[2]Link to the GitHub page: `https://github.com/computestdev/CompuRacer`

It supports single- and multi-use types of vouchers. We explain these types below:

1. **Single-use voucher** - This type of voucher can only be redeemed once and resembles a gift card. As shown at the bottom of figure 3.2, the code `COUPON1` can be used to redeem a voucher of this type.
2. **Multi-use voucher** - This type of voucher be redeemed multiple times and resembles a discount coupon that a company can send to multiple users. As shown at the bottom of figure 3.2, the codes `COUPON2` and `COUPON3` can be used to redeem vouchers of this type.

Note a limitation to the tool: the behaviour is undefined when the user tries to redeem a voucher that is not equal to the selected voucher type.

Results & conclusions All tools could be used to trigger the *Insecure* and *Very insecure* option, but the responses of the server were difficult to manually process using the Sakurity Racer or Race the web tools as Sakurity Racer just printed everything on the command line and the response-grouping functionality of Race the web did not always work. Turbo Intruder, however, did show a tabulated result view of the responses that was easy to process.

The tools have a very different method of request gathering (hardcoded, config-file and proxy), are written in very different programming languages and have different methods of checking whether a race condition took place. The most important limitations of these tools are the limited ability to import requests from different sources, the lack of support for sending sequences of different parallel requests (possibly after logging in) and finally, only a limited ability to check server responses for success. Despite the shortcomings, these tools are still useful as a starting point for this research.

Three of these tools have been developed in 2016 and 2017 and do not seem to be actively maintained. The Turbo Intruder, however, is still under active development in the spring of 2019. The fact that these tools are recently developed shows that both the issue of race conditions in web apps and the creation of toolsets for targetting these issues is still very active. As the goal of these tools is so closely related to this research, during the evaluation in chapter 6, they will be directly compared to our toolset.

In the next chapters, the core elements of the thesis will be discussed, starting with the systematic testing method in chapter 4.

Chapter 4

Creating a systematic method for web app testing

In this chapter, a systematic method is developed for performing security tests on web apps regarding race conditions. As no equivalent method exists, this method has to be designed from the ground up. First, a more elaborate definition is given of the race condition bug in section 4.1. Based on this definition, the detailed methodology is designed in section 4.2. This method contains all steps between the mapping of race condition-prone web app functionality and the successful exploitation of all race conditions that were found.

4.1 Definition of a race condition

In this section, we will try to define the race condition bug at the most basic level. Using this definition, we will try to fill the gap between the theoretical knowledge to the practical application of security testing for this issue.

On the most basic level, a race condition is unsynchronised access of multiple parallel processes to a variable when at least one of the accesses is a write. The fact that it is not synchronised makes it uncertain to the developer in what order the threads will access the variable. This is only a problem when at least one of the threads update the variable, for this makes the parallel actions non-commutative. In other words, a different order is likely to give a different outcome in program state. Depending on whether a thread reads the variable before or after the update action, it will read a different value and all subsequent actions based on the value will be different.

When we zoom out from single read/write actions, we see that these single actions are part of a series of actions that often appear as one action from the developer side. The non-deterministic interleaving of processes violates the apparent atomicity or non-interruptability of these series of actions. For instance, when a variable is incremented by one, a read, an update and a write action are executed. It is often assumed that no writes will happen to the variable between the read and write action as these writes would be overwritten. Also, in an if- or while-statement, a read, check and action are performed. It is assumed that no writes will happen to the variable that would result in a different outcome to the check and the subsequent action.

We have defined what the race condition, but for it to be an issue of interest for security testers, it must both be exploitable and have an impact. If there is no way to influence the race condition or it has no noticeable impact on the program, it is of little concern. Therefore, a tester must either be able to influence one of the involved threads or add a thread in which he performs a write-action on the shared variable. Next to this, after the race happens, the subsequent actions of a thread must be influenced and have a lasting impact on variable values or the flow of the application.

Regarding the impact of race conditions on web applications, we get a rough impression by looking at the CWE-362 that was already mentioned in the introductory chapter (MITRE Corporation, 2019). Next to an overview of the vulnerability and how to prevent, detect and exploit it, the document also lists three different types of exploitation impact:

1. **Resource exhaustion** - The *"(..) race condition makes it possible to bypass a resource cleanup routine or trigger multiple initialisation routines (..)"*. Thereby, exploitation results in using (far) more resources than necessary or expected, possibly resulting in a Denial of Service (DoS). Therefore, this type of impact affects the availability of the application.
2. **Unexpected states** - The *"(..) race condition allows multiple control flows to access a resource simultaneously (..)"*. This uncontrolled parallel access might result in unexpected control flows within an application and thereby result in crashes. Again, this type of impact affects the availability of the application.
3. **Confidence breach** - The *"(..) race condition is combined with predictable resource names and loose permissions (..)"*. In this case, an attacker might be able to read or overwrite confidential data.

We would also like to suggest that the second issue of altering control flows could also result in the third issue of viewing or overwriting confidential data. As we will be testing applications from a black-box perspective, these impacts have to be de-

tectable from this point of view. The first two issues might be detectable by looking at the response time of the server or when we are getting frequent error-responses from the server. The last issue can be detected by trying to access functionality that we should not have access to.

Taking all of this into account, we can differentiate between two types of subsequent actions that have an impact: writing a value or performing a different action. This leads to two different series of actions that are prone to race conditions. We have named them accordingly: 'Read Check Action' (RCA) and 'Read Update Write' (RUW) race conditions. These types are shown in figure 4.1.

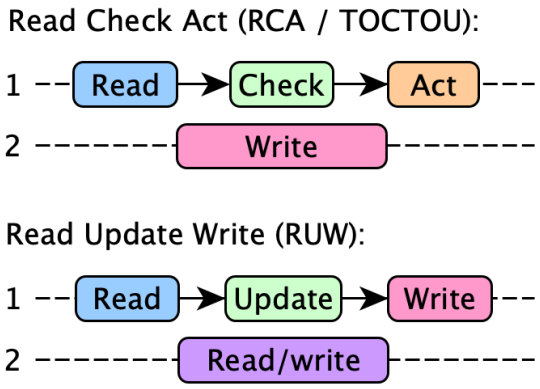


Figure 4.1: *The figure shows the two basic series of actions that are prone to race conditions. For both examples, two parallel processes are shown (vertically) in which the second process displays the interfering read/write action that would result in a race condition.*

The figure also shows what kind of action would influence or be influenced by the series if it would occur in parallel. During the RUW series, any read-action would be influenced, and a write action would be overwritten. The RCA series itself could be influenced by a write that happens in parallel. In the larger picture of two parallel series, the two RUW series could interfere with each other, and the RUW and RCA series could interfere with each other.

4.2 Development of methodology

In this section, an initial systematic testing methodology for race conditions in web apps is developed. The security test is supposed to be executed as a system, acceptance or regression test of a mostly completed software product (see background on software testing in 2.3). Next to this, the method assumes the test is dynamic, and the tester only has black-box knowledge and access to a web application. We have based our methodology on the simple hacking steps that were listed in section 3.2.1 of the related work and expanded upon that based on our own research.

Just like most security testing methodologies, it starts at the web app under test and ends with a list of suspected vulnerabilities. Between these two points, we can

discern at least five steps:

1. **Map website functionality** - How can we map website functionality from a black-box perspective?
2. **Functionality to race conditions** - Where to look for race conditions per discovered functionality type?
3. **Select HTTP requests** - How to select HTTP requests for triggering a suspected race condition?
4. **Send HTTP requests** - How can we send the requests to make the race condition likely to happen?
5. **Evaluate attack** - How can we evaluate whether a race condition occurred?

These steps are illustrated in figure 4.2. After completing one iteration of the five steps, the new findings might show more potential race conditions. In this case, the process should be repeated from step 3. After several iterations, the found vulnerabilities can be reported.

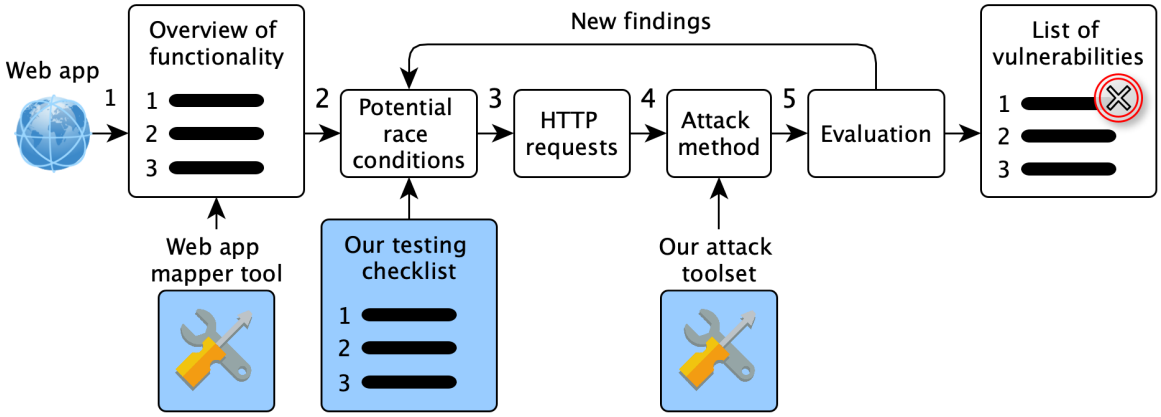


Figure 4.2: *The figure shows the flow, and aspects of the methodology for testing race conditions in web apps.*

Next to these five steps, the figure shows that that the method also requires three additional items: 1) a web app functionality mapper tool, 2) a testing checklist that indicates the potential issues in certain functionality and 3) a toolset to support the attack. In this section, each of the steps and the usage of the additional items are explained. This should result in a sufficient initial method for systematic testing for race conditions in web apps.

4.2.1 How can we map website functionality from a black-box perspective?

The first step is common to most security testing of web applications and therefore, can be re-used for race condition testing as well. Typically, a security test starts with a mapping of the application as (Stuttard and Pinto, 2011, p. 73) indicate. During this period, the tester clicks through most of the functionality of the web application while a proxy tool is used to capture the requests between the client and the application (see 'Web app mapper tool' in figure 4.2).

This tool can use the requests to map the logical (the URL structure), functional (what part does what) and authentication (which user can do what) structure of a web app. Often, this tool can also extend the manual research by using a spider that automatically crawls any additional URLs within the testing scope that were found in response content. Using some heuristics, it can also try out some paths that can reasonably be expected in any web app (like a configuration or admin page).

The result is a sufficiently complete mapping of website functionality. This is also called the attack surface. From here, the known functionality can be tested for security vulnerabilities. In this research, the Burp toolset that was intruded before is used as a website-functionality mapper, but any other mapper like the OWASP ZAP is also applicable. Both tools were introduced in section 2.3.

4.2.2 Where to look for race conditions per discovered functionality type?

The second step is to compose a list of possible race conditions that could exist for each functionality type that was found. For every item in this list, it is also important to estimate what kind of impact the exploitation of the race condition would have. To the best of our knowledge, there does not exist a direct mapping between the way to test for other security vulnerabilities and testing for race conditions regarding this step. That is why a new method must be devised. As it is unfeasible to create a mapping between any web app functionality and possible race conditions, a combination of two approaches is suggested:

1. **Checklist** - The tester should consult a basic checklist of race conditions for general types of functionality in web apps. The place of the 'testing checklist' in the methodology is shown in figure 4.2. The essential checklist of race conditions in web app functionality is not found in testing manuals or literature. That is why we have created these items ourselves based on a combination

of the limited suggestions of different sources and our own experience with testing for race conditions.

2. **Extension strategy** - The tester should be aware of and apply a general strategy for extending this list for the specific web app under test to cover all functionality. The most basic strategy is determining for every part of the unlisted web app functionality whether a particular action should only be performed a limited number of times and what would happen if this restriction is circumvented by a race condition. If this might result in a security impact, effort should be made to exploit the race condition and add the result to the other findings.

Basic race condition testing checklist

As this is the first checklist for race conditions in web apps, it was kept simple and compiled with only a few types of web apps in mind. For three particular types of web apps, the specific attack surface is considered when creating the checklist. Based on these examples, a tester is encouraged to apply the extension strategy as discussed before during the testing of a specific web app.

The considered types are blogs, wikis and e-commerce web apps. The reasons behind the choice for these application types are twofold. Firstly, the research tries to target as much concrete and open source web apps as possible. By using types of web apps that are very common and in widespread use without much difference between setups, the findings will be more generally applicable. Platforms for chats, forums, games and private clouds were also considered, but time constraints did not allow for adding these types as well. Secondly, for testing, easy setup is desirable, and these common types with many implementations are more likely to be available in a packaged format like Docker containers (Merkel, 2014).

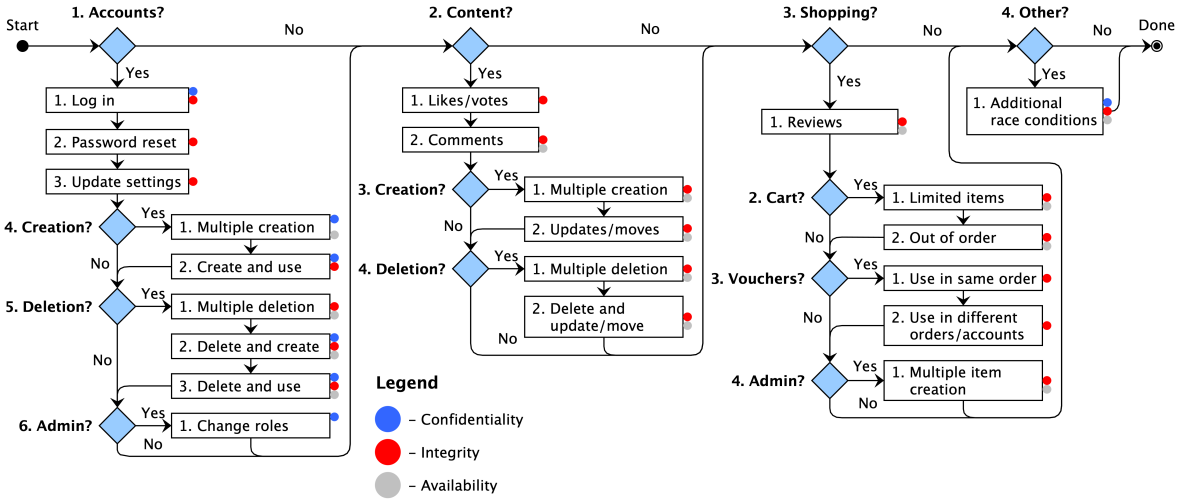


Figure 4.3: *The figure shows the flow from web app functionality to potential race condition vulnerabilities. For every vulnerability, the expected impact is also indicated using the appropriate coloured circles.*

Based on these types, the functional categories of 'Accounts', 'Content', 'Shopping' and 'Other' are defined. For each category, several subcategories are established regarding the type of functional action like 'Creation', 'Deletion', 'Vouchers' or 'Admin'. Finally, the actual race condition items are listed, including an estimation of the impact. This impact is indicated using the well-known CIA system: Confidentiality: viewing what you should not view, Integrity: changing what you should not change and Availability: limiting the access to the content for other users. This estimation of impact is in line with the impact types that we discussed in section 4.1.

Each (sub)category or item might or might not be present in a specific web and could, therefore, be skipped. The last category can be used to hold all race condition findings that are not included in the checklist but were added based on the application of the extension strategy. In figure 4.3, the checklist is included in the form of an action diagram that a tester can follow to test a web app for race conditions.

Testing checklist items explained

For every race condition item in figure 4.3, we will indicate from what access level (guest, normal user or admin) a tester should try to perform a certain action. We also explain why this functionality could contain race conditions and what kind of exploitation impact is to be expected.

1. **Accounts** - These items depend on the existence, creation, deletion and administration of user accounts in the web app.

   1.1. **(1) Log in** - As a guest, try to log in to one or multiple accounts in parallel. If the session state is not properly synchronised by the server, parallel login actions might influence each other. For two different accounts, this could lead to puzzled sessions, which is a confidentiality issue (see example 2 in section 1.1.2). For the same account, it might lead to immediate integrity issues, or when further (parallel) actions are taken on behalf of two different sessions, these actions might interfere at a later point in time. Parallel log out actions can also be tested, but no real-life exploitation of this functionality is known to the author.

   1.2. **(2) Password reset** - As a user, try to reset the password in parallel using 'Forgot password' functionality. Often, this reset includes an email with a new password or an access token. Resetting the password in parallel could lead to multiple emails with different passwords or tokens that create integrity issues when subsequently, more than one of them is used. Next to this, using a single token in parallel might also result in integrity issues.

   1.3. **(3) Update settings** - As a user, try to update user account settings in parallel. If any limited or unique fields are included like an email address, changing this value might lead to integrity issues in the application. If the email address can be changed in parallel to the same value as a victim user who changes this email address, this could also result in access to his account. This is a severe confidentiality issue, but it is difficult to set up as it requires some form of surveillance of the victim's actions and still involves much luck.

   1.4. **Creation**

      1.4.1. **(4) Multiple creation** - As a guest, try to create multiple new accounts in parallel using the same identifier (ID). If this ID should be unique, this limitation might be circumvented by a race condition. In this case, it could result in unexpected duplicate accounts that influence the stability (availability) when this results in errors, but also influence the integrity when it results in data loss in the application.

      1.4.2. **(5) Create and use** - As a guest, try to create a new account while at the same time logging in and using this account to perform an action. If the application first assigns default values to a new account and overwrites these values at some point during the creation process, these actions might be interruptable. If the default values,

for instance, assign too many rights to the user accounts, we might be able to act using these default rights before they are overwritten. These activities could involve viewing or changing data that we should not have access to resulting in confidentiality and integrity issues in the application.

1.5. **Deletion**

1.5.1. **(6) Multiple deletion** - As a user, try to delete a single account in parallel. Due to the interference, it might only delete the account partially, or delete more than one account. In both cases, the integrity of the application is compromised. The invalid data could subsequently lead to unexpected states in the server application resulting in crashes and a lack of availability.

1.5.2. **(7) Delete and create** - As a user, try to delete a single account while also adding the same or a different account. Based on the results of item four and six above, both multiple creation and deletion could lead to invalid data and unexpected states in the program. By combining these issues into one test, the creation of a partially deleted account or the deletion of a partially created account can be tested. The issue influences both the integrity and availability of the web app. Finally, if another user deletes his account while we create an account with the same identifier, it might be possible to get access to his (partially deleted) data resulting in a confidentiality breach.

1.5.3. **(8) Delete and use** - As a user, try to delete and use an account in parallel. Similar to item five above, the rights or restrictions of a user might already be deleted before the account as a whole becomes unavailable. During this race window, the user might be able to access or change data that should be restricted impacting both the integrity and confidentiality of the application. Just like in item six and seven, the invalid data could subsequently lead to unexpected states in the server application resulting in crashes and a lack of availability.

1.6. **Admin**

1.6.1. **(9) Change roles** - As an admin, try to change the roles and access rights of other users and admins in parallel. This might lead to partially implemented access rights for users which suddenly can or cannot access some confidential information. This vulnerability can only be executed by someone who already has admin rights and therefore might be less interesting from a security tester perspective as this

person can already perform much more damaging actions. Still, as an attack method for an insider, it is a very stealthy way to provide a partner with access rights without the logging file showing such an action (it just shows a simple change in roles).

2. **Content** - These items depend on the existence, creation and deletion of content in the web app. This content could be a blog item, a forum topic, a chat message, a photo or other types of storage and media.

   2.1. **(10) Likes/votes** - As a guest or user, try to like or vote for an item in parallel. Often, a limit of one like or vote per user is maintained. A race condition might circumvent this limit and allow for multiple votes by one user. Often, removing one vote resets this limit again providing the attacker with virtually unlimited votes. An attacker could use this to skew voting results in his advantage, impairing the integrity of the application.

   2.2. **(11) Comments** - As a guest or user, try to add a comment in parallel, or edit / delete in parallel. Just like items three up to eight, this could lead to inconsistent or partially deleted comments (compromised integrity) resulting in errors and a lack of availability of the application.

   2.3. **Creation**

      2.3.1. **(12) Multiple creation** - As a guest or user, try to create a certain unit of content with a unique identifier in parallel. Just like item four, this could lead to unexpected duplicate (supposedly unique) identifiers compromising the integrity of the application and lead to errors (impacting availability) when other users want to view or edit this content.

      2.3.2. **(13) Updates/moves** - As a guest or user, try to update or move existing units of content in parallel. By updating it to different values in parallel, the result might be inconsistent if no thread-safe data update merging methods are used. Similarly, moving the data in parallel might move it partially or fully to both locations creating inconsistent data or invalid duplicates. Again, both integrity and availability might be influenced by the exploitation of these issues.

   2.4. **Deletion**

      2.4.1. **(14) Multiple deletion** - As a guest or user, try to delete the same or different content in parallel. When automatic counters are used to identify this data, these counters might not be reset correctly, resulting in a value that is either too low or too high. In the former case,

newly created content will conflict with or overwrite existing content (impacting integrity). Also, partial deletion can occur that also renders a parent-object, like a website root for a blog or forum, invalid resulting in a lack of availability of a larger number of components.

2.4.2. **(15) Delete and update/move** - As a guest or user, try to delete content while updating or moving it. Just like items thirteen and fourteen, this could result in partially, updated, moved or deleted data impacting the integrity and availability of all content that is dependant upon this particular item.

3. **Shopping** - These items depend on the ability to buy or sell products using the web app. A web app that supports other types of (financial) transactions could be tested similar to item thirteen where content is moved in a parallel, and an equivalent (financial) impact is to be expected.

3.1. **(16) Reviews** - As a user, try to create, update or delete a review about a product in parallel. Just like item three up to eight and eleven, this could lead to inconsistent or partially deleted reviews (compromised integrity) resulting in errors and a lack of availability of the application. Next to this, reviews should often only be done once per user, and just like item ten, this limit might be circumvented by a race condition. This results in a skewed review consensus in the advantage of an attacker and thereby another breach of application integrity.

3.2. **Cart**

3.2.1. **(17) Limited items** - As a guest or user, try to add more items than allowed to the shopping cart using parallel add-requests. These items might be limited per user during a discount-period to avoid people buying in large volumes for profitable resale. The tester should try to add these items to the cart in every possible way. Not only using the normal 'Add to chart' request, but also, if present, the request that is used for the '+' or 'change-amount' functionality from within the shopping cart. Additionally, he should try to finalise the order in parallel or change the contents in parallel with the submission to circumvent any checks. This could result in an integrity issue with direct financial consequences for the web application owner.

3.2.2. **(18) Out of order** - As a guest or user, try to add more items than available to the shopping cart using parallel add-requests. Similar to item seventeen, the limit imposed on the number of items that are available to a user might be circumvented by a race condition. In

this case, it might result in serious unexpected situations in the order-packaging process. If this process is (fully) automated, having the requirement of a larger amount in an order than is available could result in costly delays and also impact the availability of the application.

3.3. **Vouchers**

3.3.1. **(19) Use in same order** - As a guest or user, try to redeem a voucher in parallel within the same order. The check for the availability of the voucher might contain a race condition and result in multiple redemption's of the voucher and more discount. This is an integrity issue that, just like item seventeen, results in direct financial consequences for the shop owner. This issue becomes even more interesting when the voucher is a gift card that can be partially spent. In this case, adding it multiple times might only result in partially spending one of these duplicated cards.

3.3.2. **(20) Use in different orders/accounts** - As a guest or user, try to redeem a voucher in parallel using different orders from the same or different accounts. Just like item seventeen and nineteen, this results in direct financial consequences for the shop owner. This strategy looks similar to item nineteen but is a valuable addition as using the same voucher multiple times within a single order (or shopping cart) is often blocked. This was the case for both Platform A and B that we tested. This alternative method of adding the voucher once to different orders and placing these orders in parallel still accomplishes the result of spending the voucher more times than allowed. In this case, a partially spendable voucher might even duplicate after using it in multiple different orders as the remaining amount might be returned once per order through a new voucher. For both new vouchers, this attack could be repeated to result in an exponential increase in the total spendable voucher amount.

3.4. **Admin**

3.4.1. **(21) Multiple item creation** - As an admin, try to add, update or delete a shop item in parallel. Just like items 12 up to 15, this could lead to duplicate 'identifiers', partially deleted items and overwriting other items. This results in integrity issues regarding the items of the other sellers and availability issues for the users of the shop.

4. **Other** - Here, the tester should add any other race conditions that were found in the web app when using the extension strategy.

4.1. **(22) Additional race conditions** - As an admin, user or guest, try to exploit any functionality that imposes a limit on the user which might be circumvented by race conditions. Alternatively, race conditions can also be exploited by using a (complex) combination of the items that are already listed above.

4.2.3 How to select HTTP requests for triggering a suspected race condition?

Based on the definition of race condition types at the start of this chapter, we have shown in figure 4.1 that both elemental race condition types require at least one parallel write action to take place. The HTTP-level black-box test-equivalent of a write action is an HTTP request that explicitly changes something at the server. Therefore, it is crucial during a race condition test to use at least one state-changing HTTP request. According to the official HTTP/1.1 RFC by (Fielding et al., 1999, pp. 51-57), this excludes safe methods like GET, HEAD, OPTIONS and TRACE as can be seen in table 4.1. Idempotent requests like DELETE and PUT only *work* once and every additional request should be ignored. Therefore, we are left with POST and PATCH as possible race-prone request types. As Sturgeon (2016) explains: where POST fully replaces content, PATCH is only meant to edit it.

However, it is essential to be aware that developers do not always strictly abide by these standards and therefore, a non-safe GET or non-idempotent PUT are still very likely to be used in practice. That is why we have marked the idempotence (and safety) of the GET, DELETE and PUT methods in table 4.1.

Table 4.1: *A listing of common HTTP methods and whether they are idempotent or safe. A star indicates that this protocol requirement is often violated in practice.*

HTTP Method	**Idempotent**	**Safe**
CONNECT	Yes	Yes
GET	Yes*	Yes*
HEAD	Yes	Yes
OPTIONS	Yes	Yes
TRACE	Yes	Yes
DELETE	Yes*	No
PUT	Yes*	No
PATCH	No	No
POST	No	No

For instance, in e-commerce platform A, which is a web app that we tested, a GET request was used to add a new item to the webshop while a POST request would have been appropriate. Also, for HTML 5 forms, only the GET and POST methods are natively available and this is not likely to change (Faulkner et al., 2017, Section 4.10.18.6.). So, when a developer creates a 'delete item X' web form, he can only use a POST request. Only by using asyn-

chronous XHR requests as explained in section 2.2, a developer can send PUT, PATCH and DELETE requests. In general, any state-changing HTTP request which is sent in parallel with any other type of HTTP request could hypothetically trigger a race condition.

For each testing item, using the 'Web app mapper tool' as explained before, the tester should monitor what kind of HTTP requests are sent during the regular usage of a particular web app function. During the monitoring process, any state-changing usage of requests which are not meant for this purpose should also show-up, and the tester can act appropriately. Then, he should select a minimal subset of these requests to be used in the test setup.

How much parallel requests is enough? To be more confident that requests of interest arrive at the server in parallel, more than one request per type should be sent. For instance, if two different requests A and B should arrive at the server in parallel, both A and B could be sent five times for a total of ten parallel requests. Adding parallel requests results in an exponential increase in the chance of success. The mathematical basis for this increase is found in the birthday problem that was first discussed by H. Davenport (Ball, 1960).

For the birthday problem, the chance of a collision in birthday increases somewhat exponentially (following a sigmoid pattern) and is asymptotically close to one when we have a group of about 60 people. Thus, when approximately 20% of the day-options are covered. The specific function for the probability $p(n)$ that at least 2 persons have the same birthday, where n is the number of people, is given below:

$$p(n) = 1 - \frac{365!}{365^n(365-n)!}$$

As only the event that two requests arrive at the server within the race window (a collision) matters (not the absolute arrival time), and we have a limited range of options, this principle is equivalent to the birthday problem. When sending an HTTP request, we can define the range of possibilities as the twice the connection jitter value (absolute variance) divided by the race window. If we take a jitter value of 50 ms and a race window of 2 ms (thus 25 time-slots), we would, based on the function above, need about 15 parallel requests to have more than 99.5% chance of collision.

Unfortunately, the race window is not known from a black-box perspective, so we cannot perform this calculation in practice. Also, parallel requests are still no guarantee that a race condition will occur as this is also influenced by, for instance, the server state and the database configuration. Nevertheless, the principle shows

that relatively few parallel requests are required to result in a high chance of success.

4.2.4 How can we send the requests to make the race condition likely to happen?

The critical element of exploiting race conditions is speed. The parallel requests should be able to arrive at the server within the race window, and this window might be very tiny. As far as we know, this cannot be done manually and should be supported by a software tool. As indicated in the introductory chapter, several tools exist that explicitly or optionally support this process. In chapter 5, we develop our extensive toolset to support and in chapter 6, we evaluate this toolset in comparison with the other available tools.

A number of aspects regarding the web app functionality and server state should also be taken into account when trying to make race conditions likely to happen. These aspects influence the ability to send parallel requests, the level of automation and also influence the size of the race window. They are listed below:

- **Proximity to target web app** - When testing for race conditions in a web app, the parallel requests that are sent from the client are also supposed to arrive at the server at roughly the same time. When the client is located far away from the server both regarding the absolute distance and the number of network node hops, this affects the time between requests in unpredictable ways. For every kilometre, the light speed adds some absolute delay to every request and for every node hop (router, firewall, load balancer or server) some unpredictable processing time is added.

  The absolute time added to every request (latency), is not essential as long as this value is consistent for all parallel requests. The variance in this latency (jitter) has a much more negative impact on the ability to stay within the race condition window. That is why we suggest the tester to place the exploiting client as close to the target web app as possible. Preferably within the same local network, but at least within the same country. Often, ping requests can be used to estimate the latency and jitter of the connection to the server.

- **The load on the server** - During a race condition test, the time it takes for a server to process a request fully has a direct impact on the race condition window. When the component that is prone to the race condition experiences more traffic load during the test, the race condition window is expected to in-

crease. Also, the requests of the test are expected to be better masked by the other traffic. Both of these changes are good, but with the increase in load, the variability of the race window and the number of (time-out) errors is expected to increase as well. Therefore, we refrain from a general suggestion regarding the preferred load on the server during a test. However, a tester should undoubtedly take this factor into account, and when the exploitation of an expected race condition does not work, this might be traced back to a very low (small race window) or very high (highly unpredictable race window) server load.

- **Component duplication and load balancing** - As explained in section 2.2 on the background of web apps, in order to scale a web app to support more concurrent users, a combination of component duplication and load balancing is used. For instance, a single web app uses five instances of the database are used alongside three web servers. A load balancer equally divides the incoming requests between the web servers and database servers. When a tester then wants to test a race condition and sends a request in parallel, these requests might end up at a different web server and database. Depending on the place where the race condition could occur, this would make exploitation much harder.

  If the race happens at the memory of the web server, we would require at least four parallel requests for two of them to inevitably hit the same server. This is called the pigeonhole principle: if n items are put in m containers with $n > m$, then at least one container must contain more than one item. In the case of a race condition within the database, it would even require six requests to make sure two of them hit the same database server. Preferably, the tester knows whether these techniques are used for the web app under test and would adapt his exploitation strategy accordingly. If not, the tester could also try to get this information by assessing the difference in the result between sending for instance 5 or 25 parallel requests to the server.

- **Database type and isolation level** - The isolation level of a database changes the extent to which separate parallel transactions influence each other (concurrency side-effects). However, as Milener et al. (2018) indicate, it will only change the amount of isolation between a write and read action and not the isolation between two write-actions. The latter actions are always fully isolated. Therefore, the setting is not interesting for a race condition between two write actions, but it does affect the two parallel series of actions that contain both reads and writes as introduced in section 4.1.

Four isolation settings are often supported: 1) Read uncommitted, 2) Read committed, 3) Repeatable read and 4) Serializable. In the Serializable case, the transactions are fully isolated. In the Repeatable read case, the phantom read phenomenon may occur. This is the case when one transaction adds rows to a table while another transaction (partially) reads these rows. In the Read committed case, both phantom and Non-repeatable reads can occur. Non-repeatable reads happen when two read actions (within the same transaction) during a write transaction result in the return of different row-values. Finally, in the Read uncommitted case, phantom, Non-repeatable and dirty reads can occur. Dirty reads are similar to Non-repeatable reads, but now the different read values also occur before the write transaction is committed. All of the phenomena discussed above can result in inconsistencies between different parallel reads and writes and result in a negative impact.

Still, this setting can often not be accounted for during a black-box race condition test as the tester has no knowledge of this value and its default value varies wildly between databases. Only when the tester invents a way for the specific application under test to see whether the associated phenomena are present, this setting might be discoverable from a black-box perspective. If this value is known, a tester might use the knowledge of the related phenomena when testing for race conditions.

- **CAPCHA's in web forms** - These are challenges or puzzles added to web forms that require human interaction. The name comes from *'Completely Automated Public Turing test to tell Computers and Humans Apart'*. The form cannot be submitted successfully until the challenge is completed.

  Impact: The usage of these challenges limits our ability to automatically fetch, fill in and submit forms, but this is probably not an issue. We often already perform the fetching and filling in of forms by hand and only automate the submission of the webforms. When a web app accepts identical but verified CAPCHA's, there is no problem. Only when it requires unique verified CAPCHA's per form-submission, this is an issue. This functionality could also contain a race condition and would be an interesting addition to the security test. A solution to this issue is to gather several verified CAPCHA's beforehand and use them in the parallel requests.

- **Cross-Site Request Forging (CSRF) tokens in forms** - These tokens are attached to forms in web apps to avoid Cross-Site Request Forging (CSRF). This is an attack where a user that is logged in to a particular application A visits a malicious website B. This website B makes a malicious request that

involves submitting a form to the application A using the active session of the user. When CSRF tokens are used, a random number is generated when fetching the form on the web page using the conventional method, and this needs to be added to the form submission. The malicious user cannot easily discover the value of this number, and the user is now protected.

Impact: Just like with CAPCHA's, we often want to submit forms in parallel. We first visit the page and get the CSRF token before we submit the web form, but if the application requires a unique token for every form submission, this no longer works. A workaround would be to first fetch a unique CSRF token for every parallel request that we want to send. In testing of the web apps that are mentioned in the thesis, we did not stumble upon unique CSRFs per form submission. We only saw rather constant CSRFs that were tied to the form type or the user session. Therefore, it seems that currently, there is no problem here.

- **Rate limiters and bans** - Sometimes, websites employ rate limiters or serve bans to avoid the brute forcing of user names and passwords or inhibit spam regarding comments and messages. Initially, the server often uses some form of rate limiting (or tar pitting) that temporarily blocks access. For every subsequent block, it increases in duration and eventually, the client gets banned permanently.

  How clients are identified in order to apply temporary or permanent blocks varies greatly in practice. For spam caused by logged in users, a block can directly be linked to the account, but for unknown guests, identification is much harder. When a login fails a few times, the account that is tried could be blocked based on the filled in username, but this is discouraged because it essentially enables a denial of service (DoS). An attacker could create a script to regularly perform many failed login attempts for an existing user and this result in a permanent block.

  A better alternative is the tracking of the client IP address or other heuristics, but this is both not fully effective and results in the same denial of service possibility. The attacker could use a proxy to circumvent an IP address-based block, and could also spoof someones IP address to get them blocked. To the best of our knowledge, the preferred option is to show a CAPCHA to the user after several failed login attempts. Assuming CAPCHA's cannot be automatically filled in, this essentially resolves the issue without creating the DoS issue.

  Impact: - In our case, we do send multiple requests to all kinds of functionality, including logins and content creation forms. We often do not require more than

a few requests to test functionality for race conditions, so it seems like this is not a problem. Only when the web app or connection is so unreliable that chances of success are meagre, and we need tens of requests, this might be a problem. Still, connection reliability can be improved when we run an application locally or rent a server at the same location as the target web app (see the first enumeration point). Also, even when we get blocked, all requests up to the point that the block became active, and the current test might still succeed.

4.2.5 How can we evaluate whether a race condition occurred?

As this research focuses on black-box testing, networking setup or source code is expected to be unknown during a test. Also, no specific debugging information is supposed to be available. Fortunately, at least two types of information sources are still available:

1. **Direct responses** - The requests that are used to trigger a race condition will also receive HTTP responses. Although any debugging information is not expected to be present, both the response code and the response timing are valuable metrics. When the normal response code to a request (when sending once) results in a certain response code, say 200-code (success), and an error results in a 404-code (resource not found), the tester can check whether this is still the case when requests are sent in parallel. When the parallel requests, however, get two or more 200-codes as responses, this indicates a race condition might have occurred at the server. Next to this, the response timing can be used similarly. When there is a difference between the time it takes to process a successful request and a faulty request, we can use this to our advantage to detect race conditions.
2. **Expected state** - The alternative source of information is found in an expected state of the application. For instance, when we test for item 4 (Multiple account creation), we can evaluate whether we created a duplicate account by logging in, changing the 'unique' identifier (like the email address), log out and log in again using the original identifier. This should not work if only one account was created but is likely to work when two accounts were created (and therefore only one email address was changed).

Based on a combination of the methods described above, the tester should be able to verify whether a race condition occurred and report the results.

4.3 Conclusions

In this chapter, we have successfully developed a method to perform sufficient systematic security testing for race conditions in web applications. The method discusses all aspects starting at the web app under test, then discovering race condition-prone functionality, attacking this functionality and finally verifying the discovered race condition vulnerabilities. As indicated in figure 4.2, additional items were required for this methodology. The web app mapper tools were already shown to exist, and we devised the testing checklist ourselves in this chapter. The only requirement we are still missing is a toolset to help the tester in the practical attack. This tool will be designed and implemented in the next chapter.

Chapter 5

Developing the CompuRacer toolset

In this chapter, we discuss the development of the CompuRacer toolset in detail. The name for the toolset is a portmanteau of the commissioning company 'Computest' and 'racer' which refers to something that exploits race conditions. The project is fully open sourced, including a README and an elaborate manual. The sources can be found on the public GitHub page of Computest[1].

First, the requirements for the toolset are explained in section 5.1. Then, the high-level design based on the requirements is shown in section 5.2. Last, the concrete implementation of the tool is discussed in section 5.3. In other words, we first answer the question: *What should the toolset be able to do?*, then: *How will the toolset be organised to fulfil the requirements?* and lastly: *How is this design implemented concretely into a software product?*.

5.1 Requirements

The requirements to the toolset have to be defined first in order to design and implement a toolset that supports systematic black-box testing for SSRCs. These requirements flow from the introduction in chapter 1, where it was stated that there did not seem to exist a sufficient tool to support in testing for race conditions. The requirements are created for three different phases: obtaining requests, composing & sending requests and handling responses. For every phase, functional & usability requirements are described. Next to this, for the second phase: composing & sending requests, performance requirements are stated as well.

[1] Link to the GitHub page: `https://github.com/computestdev/CompuRacer`

5.1.1 Gathering of HTTP requests

The tool should be able to support the easy gathering of HTTP requests of interest in order to use these for triggering race conditions at a target web app. How this is done should integrate well with the current practices of security testers. These desires lead to the following requirements regarding the functionality & usability:

1. **Straightforward importing of HTTP requests** - It should be possible to import HTTP requests into the tool using a method that is quicker than manually copy-pasting requests to a file or the CLI. This method should integrate into the normal testing process of security testers. This includes the ability to easily use the requests gathered with the Burp Suite or using a browser. It is expected that this results in multiple components that work together to gather and use requests.
2. **Aliveness checks between components** - Based on the point above, the toolset is expected to consist of multiple components. Each component could independently fail from the others and result in issues. For instance, a gathering tool that is supposed to forward requests to the exploitation tool should make sure the following tool is alive. Otherwise, forwarded requests could be lost. That is why the gathering tools should only forward requests when the receiver is deemed alive and indicate this state to the tester.
3. **Request correctness checks** - To avoid unexpected issues when sending requests to the target web app, requests that are imported into the toolset should be checked their correctness. Even when the requests are forwarded from other tools like the Burp suite, issues like malformed HTTP headers, unsupported body content or incorrect body length could be present and should be checked for.
4. **Persistent request storage** - Requests should be stored in the toolset to allow for re-use within a test. This request-storage should not only be in memory, but also on a medium that persists its state after restarting the testing environment on the software- or hardware-level.

5.1.2 Composing and sending of HTTP requests

The essential part of the toolset functionality is to compose and send HTTP requests in such a way as to trigger race conditions at the server of a web app. Therefore, the selection of requests of interest, the configuration of how they are sent, and then sending them should work smoothly. Triggering race conditions requires the

requests to arrive at the server with a time difference that is as small as possible to stay within the race window, so this should be pursued. These desires lead to the following requirements:

Functional & Usability

1. **Creation of batches of parallel requests** - As stated in the section 4.1, a test for race conditions always contains at least two actions that interfere with each other. When testing an application from the client-side, these actions take the form of HTTP requests. The tool should, therefore, allow the tester to combine any similar or different HTTP requests in a batch and send this batch at once.
2. **Configuring time delays** - The tool should allow for setting the time offset between sending different requests in a batch. This makes it possible to send some sequential requests in advance of the actual parallel requests to set up a certain state like logging in, changing settings, or adding products to a shopping basket.
3. **Sending requests sequentially** - To verify whether a race condition occurred or to reset the state during the test, the tester might want to send a request multiple times sequentially. Therefore, the tool should support this option for any request in a batch.
4. **Automatic batch creation** - Next to the ability to create batches of requests manually, it should be possible to redirect requests that are added to the toolset directly to a batch. This batch could be an existing batch or a newly created batch. It should also be possible to let the tool automatically send this newly created batch after a certain delay. This functionality helps in the integration of the tool used in the testing process. When the tool is set up and configured to create and send batches automatically, the tester can quickly test several different request combinations and only needs to check the results using the interface of the toolset.
5. **Comparison of requests** - Adding the right requests to a batch when requests are almost equal can be difficult. Also, similar requests forwarded from different sources might be slightly different. To aid the tester in comparing requests in these cases, the tool should include a powerful field by field or line by line comparison method that highlights differences between requests.

Performance

1. **High raw speed** - As already stated at the start of this subsection, the essential part of the tool is to trigger race conditions at the server. The raw speed at which multiple requests are sent in parallel can be used to estimate this performance. Concretely, the tool should have an equal or lower time difference between parallelly send requests than other comparable tools currently in existence.
2. **Effectiveness in triggering races** - This requirement is comparable to the first, but it targets the actual ability of the tool to trigger race conditions. The tool should be able to trigger as much as possible race conditions at a web app that could feasibly be triggered from a black-box perspective. This is the case when the race window is at least some (tens of) milliseconds bigger than the expected deviation in latency between the client and the server (jitter).

5.1.3 Handling of HTTP responses

After sending a batch of HTTP requests, it is vital to be able to validate whether this has caused a race condition. Thus, do the response contents or the timing show any anomalies that indicate successful exploitation? If so, what request triggered it, at what moment, and how did the server respond? If not, what happened instead? These desires lead to the following requirements regarding the functionality & usability:

1. **Storing of responses per batch** - To evaluate whether responses are interesting, these responses need to be stored in the same batch that sent the requests. This should be done in a way that allows for the tester to investigate what requests created an anomalous response and at what time this occurred.
2. **Aggregation of response content** - Storing all responses in a batch allows for the tester to evaluate anomalies, but this is not straightforward. When for instance, many requests are sent or when large and mostly equivalent responses are received, the tester has to look through an unfeasible amount of data. Therefore, the tool should support the aggregation of responses that already takes care of sorting out the simple cases. For instance, empty responses due to timeouts can often be ignored. Also, the same responses only need to be shown to the tester once.
3. **Summary of differences** - Next to a grouping of responses as described above, several aggregated aspects of the responses should also be calculated

automatically. For instance, a count of the different response codes, number of headers and body lengths (in bytes) can quickly show the tester what kind of anomalies can be expected in the responses. The tool should support this.

4. **Parsing of result-bodies** - Next to the aggregation of responses and the creation of aspect summaries, investigating the bodies of the requests themselves should also be supported. For instance, a single- or multi-part form, but also a JSON string could be parsed and be shown in a prettified way with one key-value pair per line. Next to this, an HTML response is much more informative when it is parsed and shown in a browser. The tool should support easy viewing for these four types of responses.
5. **Comparison of responses** - Similar to the comparison of requests described in section 5.1.2 above, the tester should also be able to compare responses field by field or line by line while the differences are shown. More specifically, the tool should let the tester pick any group from the aggregated results of the same request and perform this powerful comparison.

5.2 Design

In this section, based on the requirements laid out in section 5.1, the design of the CompuRacer toolset created and explained. First, based on the requirements, it becomes apparent that two types of components are required: a type that gathers HTTP requests and a type that uses the requests to trigger race conditions and to evaluate results. We have decided to split the toolset into three separate parts or tools: the Core application, Burp extension and browser extensions (Chrome & Firefox). All extensions are used to gather and alter requests of interest. These requests are subsequently sent to the Core application. From the Core, the requests can be sent to trigger a race condition at the target web app, and it can also be used to analyse the responses afterwards.

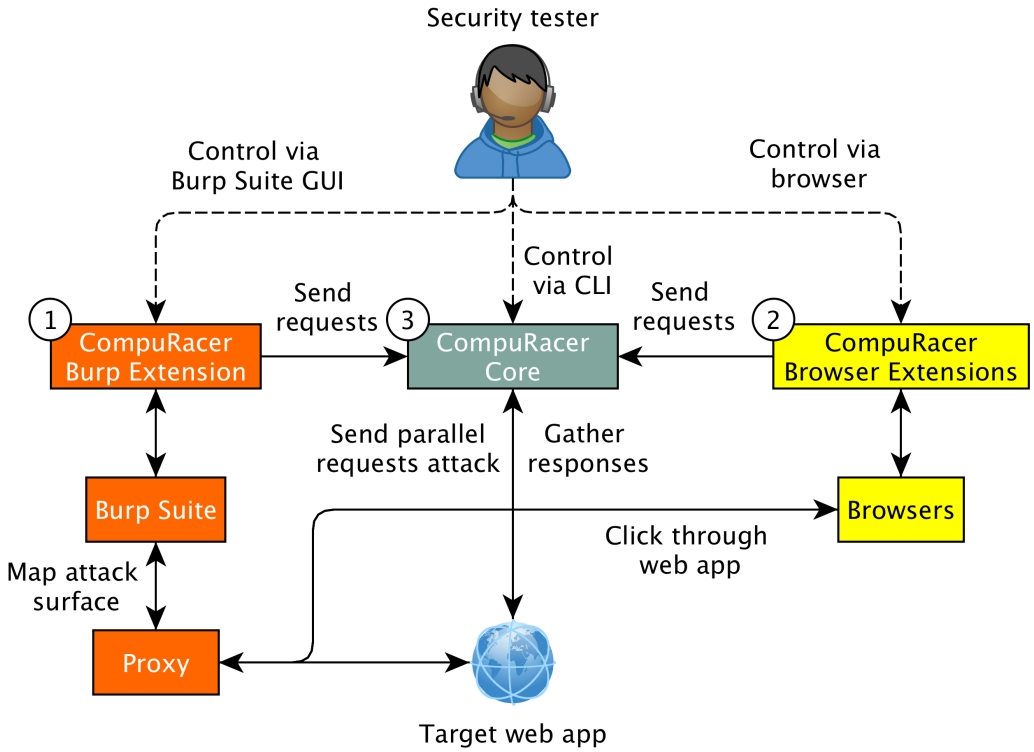


Figure 5.1: *The figure shows how the toolset fits within the normal activity of a security tester.*

Figure 5.1 illustrates the main network structure of the CompuRacer toolset. On the left, in orange, all parts that are related to the Burp Suite are displayed. On the right, in yellow, all parts that concern the browser are displayed. In grey, the Core element of the toolset is shown. The user controls the CompuRacer toolset via the CLI of the Core, the Graphical User Interface (GUI) of Burp Suite and the browsers. Next to this general structure, a more detailed description is given of the design of the three sub-components. For every tool, where applicable, it is described how the tool should fulfil the requirements defined before.

5.2.1 Core

The Core is the part of the toolset that contains most of the functionality and hereby also fulfils most requirements. More specifically, the Core is supposed to take care of all requirements of the second and third phase: 'Composing and sending of HTTP requests' and the 'Handling of HTTP responses'. Next to this, regarding the first phase: 'Gathering of HTTP requests', it is responsible for receiving requests from gathering-tools and responding to their aliveness checks (requirements 1 and 2). It is fully responsible for requirements 3 and 4.

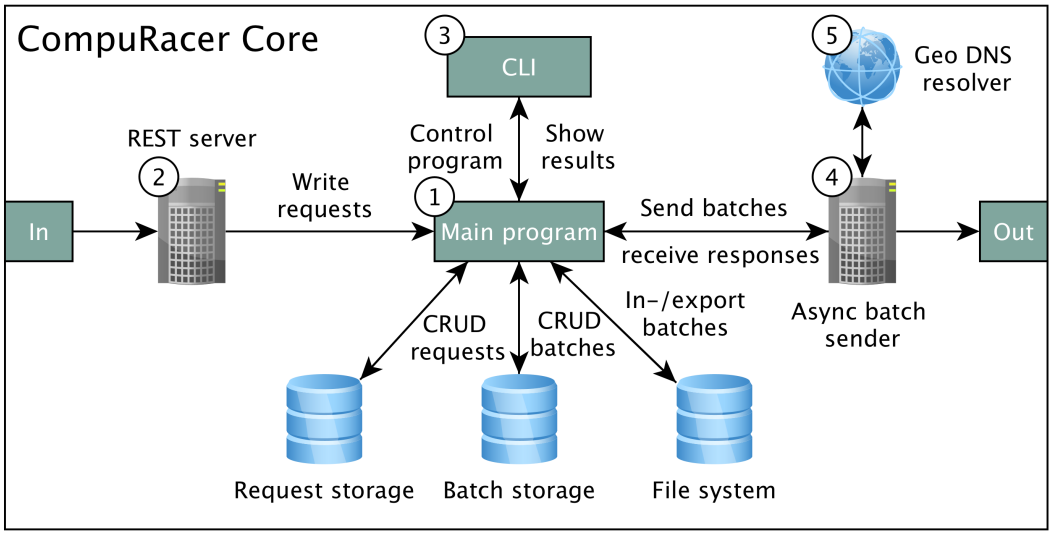


Figure 5.2: *The figure shows how the Core application is designed.*

In figure 5.2, the design of the Core application is shown. HTTP requests enter the Core via a REST server (2). The main program (1) receives the HTTP requests from the server and stores them in its local storage. Via the CLI (3), the main program can be controlled to create and store batches of requests that can later be sent on command via the async batch sender (4) to the target web app. The async batch sender will forward all responses back to the main program for further automatic and manual analysis directed by commands from the CLI. In this section, we will not go into the implementation of these specific components. However, the design of some concepts for the CompuRacer toolset as a whole will be discussed instead.

Batches of requests

This design concept is about the parallel and sequential requests and their delays within the batches. The batch is a concept used in the CompuRacer to hold a configuration of requests that can be sent at once. Every request is added to this batch with the number of parallel and sequential duplication's. The sending part of the toolset can use the batch object as an accurate and complete description of how the attack should be executed.

Request receive modes

This design concept is about different modes of the Core that determine what it will do when a new request is added via the extensions. These modes create a level of automation that allows the tester to circumvent the normal steps required to create and send a batch of requests. Next, we will explain in detail how every mode influences the behaviour of the Core after adding requests[2].

- **On** - In this mode, all requests that are received within a small interval will together be added to a specially reserved batch. The mode uses its own sequential and parallel duplication settings to add the requests. Then, the batch is sent automatically. Just like the normal batch, the results can be viewed, and it can be copied to a regular batch.
- **Curr** - In this mode, all requests that are received are added to the current batch. It uses the same sequential, and parallel duplication settings as the 'on' mode does. In contrast to the 'on' mode, will not send the batch automatically. As the normal flow of activities always includes creating a batch and adding stored requests, this mode removes the second step and makes the process more user-friendly.
- **Off** - In this mode, no additional action will be taken when a request is received.

Last-byte synchronisation in sending

This design concept is about the synchronised sending of the last byte of the body of parallel requests. As long HTTP requests tend to be split into several TCP/IP packets when transmitted, the low local time difference between requests could significantly grow in unexpected ways when the packets reach the web server. As web servers often wait for the entire HTTP request headers and body to be received, synchronising the last TCP/IP packet between parallel requests could severely reduce the unexpected time delays.

The idea originates from two tools that support this functionality. The first is a very basic race condition testing tool by Riancho (2019) which is available on GitHub. It was not discussed earlier because it is largely undocumented, difficult to, and the author could not get the tool to work. In `rc_exploit/utils/create_threads.py` on the lines 52 to 55, it is clear that last-byte-synchronisation is used. Next to this, in the Turbo Intruder tool that was introduced in section 3.2.3, similar functionality is

[2]although duplicate requests will not be added to the total request list, they will trigger the mode actions with the current request as an argument.

found. The Turbo Intruder developers have added this feature to ensure *"(..) all your requests hit the target in as small a window as possible, which can be done by queuing all your requests before starting the request engine."* (Kettle, 2019).

Presentation of results

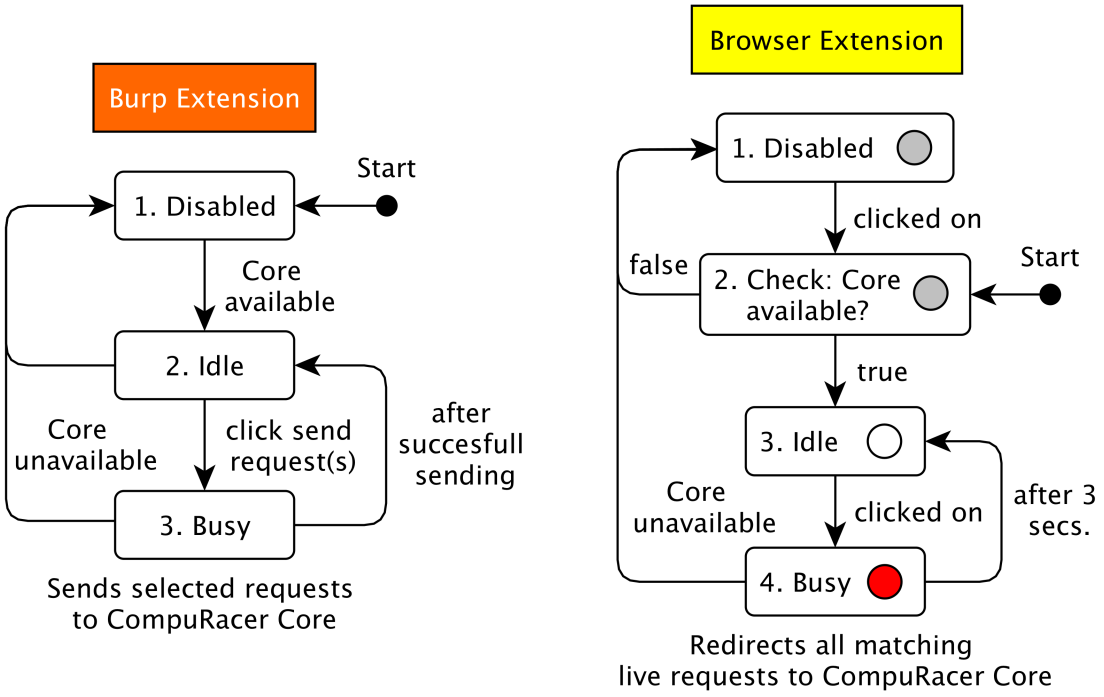
This design concept is about the presentation of the results in a concise manner. This presentation contains several summary tables for a quick overview and a grouping of unique responses for detailed analysis. These tables will show a count of the unique status codes, body length (bytes), number of headers and the header length (bytes). This allows the tester to spot any anomalies in the responses quickly. The grouping of unique responses is done based on the HTTP headers and body content, which removes uninteresting clutter in the results. The grouping helps the tester to more easily spot differences in the results and determine whether a race condition has occurred. For both the send and response times of all responses in a single group, only the minimum and maximum times are shown. The timing information within the headers like the 'Date' and 'E-tag' fields are largely non-informative and therefore are ignored in the grouping process.

5.2.2 Extensions

The extensions are the part of the toolset meant for gathering requests. As the tester might use different browsers or tools for their work, multiple ways have been designed to gather requests. This should result in the process of testing for race conditions that easily integrates with the normal security testing processes and thereby fulfils requirement 1 of the first phase. Therefore, both an extension is created for the popular testing tool called Burp Suite, and also for both the Chrome and Firefox browsers.

In figure 5.3, the design of the lifecycle of all extensions is shown. This lifecycle contains two important elements the aliveness check of the Core application and the forwarding of requests. The aliveness check fulfils requirement 2 of the first category.

Differences between the Burp and browser extensions Although both types of extensions have a similar purpose, they work in slightly different ways. Firstly, the browser extensions can only forward live requests while the Burp extension can also forward historic requests from storage. Also, the browser extension has to be



(a) Lifecycle of the Burp extension

(b) Lifecycle of the browser extensions

Figure 5.3: *The figure shows the the different states and transitions within the lifecycles of the Burp Suite and browser extensions. For the browser extensions, the color of the button for each state is shown as well. The event-based language of the browser extensions requires an additional click-action of the user.*

written in event-based asynchronous JavaScript and the Burp extension either in the mostly synchronous Python or the Java language. Thus, the former can only respond to external events, and the latter can keep running in a loop and actively check for conditions. That is why state two is unique to the browser extensions: they require the user to activate a single check for server aliveness. Other than this additional state, the extensions work in similar ways.

The added value of the extensions We have chosen to develop both the Burp and browser extensions for multiple reasons. As the Chrome browser extension in its purest form was part of the Sakurity Racer tool, this was a very reasonable basis to start. Only later, an effort was also made to create the Burp extension, and at that moment, the Chrome browser extension was already completed. Next to this, the extension was also ported to Firefox as this required only a few configuration changes. However, even with the Burp extension in place, the browser extensions have not become obsolete. They fulfil the unique purpose of supporting the testing of visual components, like submitting a form or clicking a button, directly from the browser

without having to look up the accompanying request in Burp (or an equivalent tool). Also, when Burp is not used, it provides a quick way to test some functionality in the web app for race conditions.

The primary advantage of the Burp extension is that it is already integrated into the regular testing flow of a security tester. The tester usually first clicks through the application to map the attack surface, and from then on only uses the Burp Suite and other tools. At any point in performing tests with the data gathered in Burp, the HTTP requests can also be sent to the CompuRacer Core for testing race conditions, and this provides for a smooth testing experience.

5.3 Implementation

In this section, the implementation according to the high-level design laid out in section 5.2 is described. Both the Core, the extensions and their sub-components will each be discussed, including the functional, performance and security considerations when applicable. This chapter only describes how the CompuRacer is implemented but does not contain a manual on how to use the toolset. The manual can be found in appendix C.

5.3.1 Core - Main class

The Core is implemented as a Python program spanning about 4000 lines and is divided over six class-files and a separate file with some utility functions. Five of these classes match the five functional components in the Core design, and next to this, a class is created specifically for holding and manipulating all batch information and responses. All six classes and their contribution to the Core is discussed separately.

The first and foremost aspect of the Core is the main class. It is responsible for both the startup and the shutdown sequence of the Core. Next to this, it regulates all communication between different components during the runtime.

Startup sequence

On startup, it will first try to load the settings file `state/state.json`. If it is not found, a new settings file will be created according to the preferences of the user. Then, all stored requests and batches (from the files in the `state/batches/` folder)

will also be loaded if present. Finally, it initialises all other parts and sets up the necessary communication mechanisms between these parts. Concretely, it loads all commands and associating functions into the Command Processor class, initialises the REST server and finally starts the Command Processor.

Runtime

The functions of the main program during runtime are mostly passive. It contains all functions that are linked to the CLI commands; it will receive requests from the REST server, will read and store files and will forward batches to be sent by the batch sender.

Shutdown sequence

When the user has initiated the showdown sequence using the appropriate command, or by killing the program directly, the main program will first shut down the REST server and the Command Processor. Then, any unsaved part of the state of the program, the requests and the batches are saved to disk and the program exits.

5.3.2 Core - REST server

The second aspect of the Core that is discussed is the RESTfull server. According to the design, a server class is implemented in the toolset using the Flask microframework Pallets Team (2010). This is a straightforward server framework that is meant for development and experimentation purposes. Every endpoint of the API is linked to a function in this class by using Python annotations.

Functionality

The endpoint that is used by all extensions to check for the liveliness of the server is `GET /`. The endpoint to add single requests is `POST /add_request` and for multiple requests `POST /add_requests` can be used. The tool will then extract the requests from the JSON body and parse them. Subsequently, the requests are sent via a thread-safe queue to the main part of the Core in order to be added to the total request list. HTML responses to parallelly send requests are also hosted by this server and can be accessed by name using `GET /responses/<string:filename>`.

Performance and security considerations

The Flask framework, which was created almost ten years ago, is neither designed to be secure nor fast. Extensions like FlaskSecurity made by Wright (2012) try to add security mechanisms and as shown by Jones (2017), the asynchronous Asyncio library can be used to improve performance. However, this is not necessary for the current application as ultimate performance is not demanded and this server is only locally accessible to the tester to forward requests from one application, being Burp or a browser, to the other being the Core. That is why, to our knowledge, there is no way to intercept these requests or to access the server without already having compromised the host itself.

Multi-threading challenges

This server runs on and blocks the thread in which it is started, and therefore has to be started in its thread to allow for other functionality of the Core to execute concurrently. By default, the server will log every request to the standard output (command-line), and this interferes with the interaction of the user with the CLI. By running it in a separate process which has its standard output, this issue can be resolved. Unfortunately, programs that consist of multiple processes cannot be reliably debugged in Python, and the ability to perform debugging is an essential part of a research-grade toolset. Therefore, the decision is made to disable all logging of the server instead. If at any point in time, server debugging should take place, the logging can be enabled again by switching one boolean value.

5.3.3 Core - CLI

The third aspect of Core is the Command Processor. It is used to receive commands from the user via a standard input like a terminal. As said before, the commands are first added by the main part of the program. When it starts the Command processor, this class takes control of all standard input and output of the program.

The class has been self-written by the authors of this research. We had decided not to use either of the available CLI libraries for the following reasons. First, we already had a simple self-created CLI class available from different research projects, and after some alterations, it also seemed to fit this project perfectly. Next to this, the libraries provide such a scale of additional functionality that it would require some

further learning to be able to use it appropriately. In section 8.2 on the future work, we do propose the use of dedicated CLI libraries to fulfil this function.

Command formatting

The CLI supports commands and sub-commands, followed by several arguments. These arguments must be provided in a predefined order, but optional arguments may be omitted. When a command has multiple optional arguments, and either one of them could be omitted, we cannot differentiate between these cases. That is why if the user wants to omit the second to last optional argument, he must also omit the last optional argument. The arguments can currently be of type int, float, boolean, string and enum. When arguments contain spaces, the arguments must be enclosed in double quotes. The first section in the manual in appendix C contains a more elaborate explanation of the formatting and usage of the supported commands.

Basic functionality

It supports all basic commands that a user would expect from a CLI like a help command (with search options) that lists all functionality, an executed-commands history and detailed error reporting on failed commands. Next to this, the CLI is loaded by the main class with five additional commands regarding general behaviour, mode changes, requests, batches and the current batch. These types of commands are explained below:

1. **General** - These commands are related to the general workings of the application. They encompass changing the startup message, the enabling of the coloured output of the application and the saving/exiting of the application.
2. **Mode** - These commands are related to the mode of the application. They can be used to change the mode to one of the three states: 'off', 'curr' or 'on' as explained before. Next to that, the parallel and sequential duplication settings of the mode can be altered, and the user can control whether the immediate mode prints its results.
3. **Requests** - These commands are related to viewing, comparing and removing requests. They can be used to view the summary of all gathered requests, show the details of one specific request, compare two requests line by line and remove a request. Finally, the internal ID's of requests by which the user can reference them can be re-calculated when necessary. This update is also

applied to all batches and results that reference the requests. As every new request gets an ID that is the highest existing ID plus one, frequent removal and addition can result in ID value fragmentation (high ID values, but few requests). This command can undo the ID fragmentation.

4. **Batches** - These commands are related to creating, viewing, updating, copying, comparing importing/exporting and removing of batches. They can be used to send a certain batch, create a new batch by name, and list a summary of all stored batches. A certain batch can also be marked as the current batch. Details about the configuration or of the results (if present) of a single batch can be requested, and just like the requests, responses to the same request within a batch can be compared line by line.

   Batches can also be renamed, completely removed, and the configuration and results copied to a new batch. Batches can also be exported a `JSON` file for manual inspection or backup purposes. These files can be imported back into the application at a later time. Finally, the grouping of results within batches can be re-executed at command. This is useful when batches with ungrouped results are imported or when any changes to the source code have been made that influence the grouping behaviour.

5. **Current-batch** - These commands are related to changing, viewing, comparing and removing the current batch. The current batch is a batch that can be accessed using shorter (easier) commands, and only the current batch can be edited. Just like the general commands for the batches, the user can view the content, results and compare responses.

   Contrary to the other batches, the user can now also add and remove requests to/from the batch and update the configuration of a request that is already added. Next to this, the commands can be used to change the sending behaviour of the batch. First, the user can change whether redirects are followed. Second, whether the last-byte-sync behaviour is enabled and last, what HTTP header fields (or the body contents) are ignored when responses are grouped. The latter functionality is useful to exclude fields that are meant to change between responses like counters or time fields. A number of header fields like 'Date' or the 'E-tag' are already ignored by default.

5.3.4 Core - Batch

The batch class of the Core is responsible for holding the information of one batch. Next to this, it manages and groups the response data. It can hold an unlimited number of requests and configurations, but it only stores the result of one send action. If the batch is sent again, these results are overwritten.

For the responses, a summary is created based on all responses to the same requests. This summary contains a count of the unique status codes, body length (bytes), number of headers and the header length (bytes). For every aspect, this data is shown in a table. As stated in the subsection above on the CLI of the Core, it will also group responses based on equality in header fields and body content. This is implemented as follows, for every batch that is sent the responses to the same type of request are added to the same list. For every element in this list, it is compared field by field to the representative of every existing group of similar requests. If only the ignored fields do not match, the request is added to the group.

5.3.5 Core - Async Batch sender

This is the part of the toolset responsible for sending batches of requests. It has to abide by both functional and performance requirements. The functional requirements encompass that it has to be able to send any number of requests with parallel and sequential duplication and the requested delay. The performance requirement entails that it tries to send parallel requests as fast as possible.

The main functionality of the component is sending the parallel requests as fast as possible, but to the best of our knowledge, there are not many examples available on how to approach this using the Python language. There were plenty of documented ways to host high-performance servers, but not high-performance clients. This situation was to be expected as we can imagine only some special Python-based password bruteforcing tools have similar performance requirements. Therefore, this was an involved trial-and-error type of process. We can discern four stages in which the performance of the batch sender has been improved.

1. **Multi-threading** - Initially, it seemed best to use the default Python HTTP library called 'Requests' and spawn multiple worker threads that all send requests in parallel. However, it turned out that using multiple threads would not be an improvement after all because the Python documentation shows that the Global Interpreter Lock (GIL) does not allow multiple threads to access python objects at once (Wouters, 2017). The GIL is a mutex (MUTual EXclu-

sive access) that prevents multiple threads from executing Python bytecode at once and is required because the CPython memory management is not thread-safe. It mentions that long-running I/O operations could happen outside the GIL, but the functions in the Requests library as a whole will still run sequentially. Therefore, this option was abandoned.

2. **Multi-processing** - To avoid the GIL threading issues, the alternative was sought in multiple Python worker processes. These processes do not share normal variable values and therefore, can execute in parallel. A bi-directional process-safe queue was used to pass the request to the process and return the result. This worked, but unfortunately, this option also greatly disturbed the Python debugging ability. Apparently, the Python debugger was not designed to work with multiple processes, and when the debugger would encounter one exiting process, it would think the whole program had shut down and would exit as well. By deciding not to debug this part of the program, the developer can avoid this.

   At this point, using a network protocol analyser called Wireshark (Wireshark, 2019), the raw performance of the tool was compared to the performance of Sakurity Racer when sending a single request in parallel. The raw performance was measured concerning the average time difference between two consecutive requests. For this test, the account registration request to the OWASP WebGoat of the first example in section 1.1.2 was used. The results are shown in figure 5.4 and figure 5.5. As clearly shown, the time-differences between requests made by Sakurity Racer are about 100 times smaller than the time-differences between requests made by the CompuRacer. Also, because of the much slower performance, the CompuRacer was not able to trigger the account-creation race condition. Therefore, this option was abandoned, as well.

Time	Source	Destination	Protocol	Length	Info
0.000078	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000082	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000088	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000084	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000080	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000078	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc
0.000079	127.0.0.1	127.0.0.1	HTTP	761	POST /WebGoat/register.mvc

Figure 5.4: *The figure shows multiple parallel requests captured with Wireshark which were made by the Sakurity Racer tool. The time-column shows the difference in seconds at microsecond resolution between two subsequent HTTP requests.*

Time	Source	Destination	Protocol	Length	Info
0.008078	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.007654	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.003707	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.002763	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.008727	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.008268	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.004618	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc

Figure 5.5: *The figure shows multiple parallel requests captured with Wireshark which were made by the CompuRacer toolset when using multiple processes and the 'requests' library for sending. The time-column shows the difference in seconds at microsecond resolution between two subsequent HTTP requests.*

3. **Asynchronous** - The next option that was considered is sending the requests using an asynchronous library called 'aiohttp' [ref] (instead of the very bulky 'Requests' library). Since Python 3, it supports both synchronous and asynchronous (event-based) programming. This method does not execute code purely sequentially or in parallel but uses a hybrid of the two options.

   An event-loop is used to sequentially queue and process actions, but all actions that cannot be executed immediately, like network requests, are handed over to a parallel process (from a pool of available processes). The event-loop then continues to process the next action while the parallel process executes the delegated action. When the delegated action is completed, the callback with the result is added back to the event-loop to be processed sequentially. We used a faster version of the built-in 'AsyncIO' event-loop called 'uvloop' [ref] which can achieve a 2 to 3 times speedup.

Time	Source	Destination	Protocol	Length	Info
0.000016	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000009	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000009	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000009	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000008	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000010	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc
0.000008	127.0.0.1	127.0.0.1	HTTP	125	POST /WebGoat/register.mvc

Figure 5.6: *The figure shows the time difference in seconds at microsecond resolution between two HTTP requests for the CompuRacer toolset when using the asynchronous 'Aiohttp' and 'uvloop' libraries for sending.*

Using this new method, the same test, as described above, is executed again to evaluate the improved performance. The results are shown in figure 5.6. This is a major speedup of about 800 times, which results in a performance that is about eight times better than that of Sakurity Racer.

4. **Last-byte-synchronisation** - The last improvement to the tool was made using a method called last-byte-synchronisation. We altered the source code of the 'aiohttp' to include the ability to start sending some packets in parallel and then synchronise the sending of the last byte of the body. This last byte is sent at a specific point in time for all parallel requests. A limitation of this method is the fact that requests without a body cannot be synchronised. As only HTTP requests without a body but with huge headers are expected to be split over multiple TCP/IP packets, this seems to be a minor issue.

Next to these clear performance stages, some other performance measures can also be discerned. First, before sending, all request-objects are pre-created, and the processing of responses is also postponed until all responses are received. Both measures are meant to avoid any processing interference.

Next to this, when the goal is for instance to trigger a race condition between the login and shopping based functionality, we might decide to send ten copies of two different requests (like a login and a 'put product x in my shopping basket') in parallel. However, as the tool would originally first create 25 asynchronous sending tasks for the first request and then 25 tasks for the other, the time difference between the first copy of the first request and the first (or last) copy of the second request would often be much higher than the optimal performance of the tool promises. In our case, at around 25 or more duplicated requests per type and using two or more different types, a significant delay was often observed. To solve this issue, the order in which asynchronous sending tasks for different types of requests are created, is now randomised and this seemed to solve the issue.

5.3.6 Burp extension

The burp extension is written in about 250 lines of Python code within a single file. The Burp Suite natively only supports Java plugins, but as the extension Application Programming Interface (API) indicates (Portswigger, 2018b), the Jython runtime can also be used which converts the Python code to Java bytecode. It allows for selecting one or more requests from any message viewer or editor within the Burp Suite interface, packages these requests in a JSON object and adds this as a body of a new POST request. It then sends this request to REST server endpoint `/add_requests` at the Core. It uses a separate thread to check whether the Core is alive by sending a GET request to the root of the REST server. If it responds with a 200-code, the Core is assumed to be alive. It does not allow sending requests when the Core is not alive.

5.3.7 Browser extensions

The browser extensions are written in about 250 lines of JavaScript code within a single file. As indicated in the design of the extensions, it can capture requests and forward them the CompuRacer Core. It uses the WebRequest API of the browser to block all requests of interest, package them in the JSON body of a new POST request and send this to REST server endpoint `/add_request` at the Core. Next to this, the extension is also able to check whether the Core is alive and will only forward requests if this is the case. It uses a GET request to the root of the REST server, and if it responds with a 200-code, the Core is assumed to be alive.

Contrary to the Burp extension, the user cannot specify which requests to forward, and that is why the extension has to contain some programming logic to decide what requests are interesting. These requests are selected based on several hardcoded rules. First, the requests have to be of the HTTP or HTTPS (secure) type. Second, all requests with the OPTIONS, CONNECT and TRACE HTTP methods are ignored because we have shown in section 4.2.3 that these requests are not state-changing and therefore not interesting for race conditions. Third, all GET requests of an image, script, CSS and font are ignored as these requests are not state-changing.

5.4 Conclusions

In this section, we have created implemented a toolset that supports the security tester in finding and exploiting race conditions in web applications. In the next chapter, we will test both the toolset and the methodology created in chapter 4 to evaluate whether they actually fulfil the requirement of supporting the first systematic method in testing for race conditions in web apps as set out in the introduction of the thesis.

Chapter 6

Evaluation of toolset and testing methodology

In this chapter, we will evaluate both the testing toolset and the testing methodology. For the tool, per the main question of the research, we will evaluate the extent to which it adequately supports the tester when he is testing for race conditions in web apps from a black-box perspective. The tool needs to contain the necessary functionality, needs to be easily usable by the tester, and needs to perform well enough to exploit race conditions from a black-box perspective successfully. For the methodology, it needs to provide the tester with a complete and sufficient guideline to perform a systematic race condition test on a web app. The method is developed with these requirements in mind, but we will execute a practical test to verify these claims. The toolset will be evaluated first as we need a working toolset to use and assess the testing methodology. Based on the requirements to the toolset as defined in section 5.1, we have defined ten metrics regarding the functionality and usability of the toolset in section 6.1 and we have identified four metrics regarding the performance of the toolset in section 6.2.

Based on the metrics, we have gathered the necessary data to rate the toolset and make the appropriate conclusions. In this evaluation, when applicable, the results of the CompuRacer toolset are compared to three existing race condition testing tools. The tools used for comparison are the Sakurity Racer, Race the web and Turbo intruder which we introduced in section 3.2.3. As the Netclonefuzzer tool could not be made to work successfully on macOS, we have excluded it. Next to this, for the performance test, we used two versions of the CompuRacer. One version with last-byte-synchronisation enabled and once with the functionality disabled (see point 4 in section 5.3.5).

Next, according to the methodology, the toolset is used to test seven web apps regarding race condition vulnerabilities. The web applications we tested fall in the three web app categories that were also considered when creating the testing methodology in section 4.2. The web apps were tested regarding all applicable items as specified in figure 4.3 (page 49). Finally, the findings are discussed, and we draw the appropriate conclusions.

6.1 Evaluation - Toolset functionality & usability

The toolset should be easy to use by a security tester and provide all the necessary functionality to support a race condition test. In other words, the functionality and usability of the toolset are essential. In order to compare the functionality and usability of the tools, we have defined ten metrics. The metrics cover both the preparation of the tool before a tester can use it, the usage itself and how future-proof it is estimated to be.

We can objectively rate tools according to purely functional metrics, but the usability metrics will involve more subjective evaluations. Therefore, for every metric, not only the scores are given but also an argument for why the tools receive a particular score.

In this section, first, every metric is explained, and the scoring system is laid out. Then, we will rate every tool according to the system, and explain the scores. Finally, based on the data, conclusions are drawn.

6.1.1 Definition of metrics and scores

Below, every metric is defined, and we explain how a good and bad score can be distinguished. These definitions and scores are summarised in table 6.1.

Relative metric weight distribution Regarding the weights of the individual metrics, we have decided to divide them into three groups. The first three metrics are about the setup of the tool (group 1), the next six metrics are about the usage of the tool (group 2), and the final metric is about the continual use of the tool (group 3). The metrics in all groups are important, but regarding the most essential impact on the functionality and efficiency of the tool, we can make a distinction.

The first group only once takes a specific unit of time for a tester, and the second group takes an amount of time for every test or test case. Therefore, the time-usage for the second group has the most substantial impact on the total adoption and usage time. Next to this, the second group addresses the most essential functional aspects of the tools. That is why these six metrics are regarded as the most important and used as a tiebreaker when two or more tools would otherwise get the same score.

1. **Instructions** - the availability of a guideline on how to install and use a tool. Having no instructions results in the minimum score and a complete manual on how to install and use it will yield the maximal score.
2. **Installing tool** - the complexity and the number of steps required to install a tool. If a tester should follow many complicated steps for the installation, it gets the minimum score. When a tester, however, only needs a download and a single-click installer, it gets a maximal score.
3. **Configuring tool** - any configuration required after installation and before we can use the tool to test for race conditions. Similar to the installation of the tool, a straightforward process will yield the maximal score, and a complex process involving many steps will result in a minimum score.
4. **Control options** - the way the tool can be controlled by the tester to perform race condition tests. When any alterations require source code updates, it gets the lowest score. On the other hand, when not only a human (graphical or command line) interface is available, but also an API is available for automatic or remote test integration, it will receive the highest score.
5. **Importing requests** - the process of getting HTTP requests of interest into the tool for further configuration and using them in a test. When only manual and repeated copy-pasting of raw HTTP packet contents are supported, this is very inconvenient and will yield the worst score. A much better way is the ability to dynamically push requests via multiple kinds of sources like proxies and browsers that were already meant for gathering HTTP requests. These abilities result in good integration and easy usage and will result in the best score.
6. **Storing requests** - the ability to not only import and use requests but also store them conveniently for later use. If this is not supported at all, the tool will get the lowest score. To the contrary, if both storage and well-designed searching through these requests are supported, the highest score is awarded.
7. **Storing configuration** - when a tool can be set to work in different ways de-

pending on the type of race condition test, it should support storing this configuration to avoid repeated work. When not only the storage of settings but also importing and exporting of settings for different tests are supported, we hand out the maximal score. When no such storage is possible, the tool gets the lowest score.

8. **Sending requests** - the most essential part of a tool is the parallel sending of requests. In order to trigger complex race conditions, the tester often requires multiple parallel requests and delicate timings. If any combination of requests and timings is supported, the tool gets the highest score. If only the bare essential functionality of sending a single request multiple times is supported, we give the minimum score.

9. **Aggregation of responses** - verification of the occurrence of a race condition is the second-most important functionality of the tool. When the tool prints all HTTP responses to the parallel requests to the standard output, analysing this becomes very cumbersome and therefore yields the lowest score. When advanced and possibly automatic aggregation of responses for quick verification of success is supported, the tool gets the highest score.

10. **Future proof** - for most research-grade tools, acquiring and particularly using more antiquated tools can be very challenging. An open source and actively maintained tool with updates in the past six months will probably work most conveniently and will be usable for a longer period in the future. Therefore, such a tool will receive the maximal score. On the other hand, a closed source tool that does not seem to be actively maintained hence will receive the lowest score.

Table 6.1: *According to the scoring laid out before, this table shows the concrete mapping between metrics and scores that is used when rating the tools.*

Metric	Worst: - -	Bad: -	Good: +	Best: ++
Instructions	None	Very limited instructions	Instructions on, install, config and run	+ elaborate manual on how to use the tool
Installing tool	Complex and time-consuming: many steps / requirements / bugs		Download, some config and start installation	
Configuring tool	Complex and time-consuming		Quick config	No config required
Control options	Via source code changes	Via config files	Via a GUI / CLI	+ via and API
Importing requests	Manual using copy paste	Manual using files	Dynamic via API	+ multiple dynamic methods
Storing requests	No	Yes		+ automatic / searchable
Storing configuration	No	Yes		+ import / export
Sending requests	One request in parallel and no timing config		Different requests / timing configs	+ any request and timing combination
Response-aggregation	No	Yes, simple grouping / filtering / highlighting		+ more advanced options
Future-proof	Closed source & abandoned	Closed source & active / open source & abandoned		Open source & active

6.1.2 Rating the tools according to metrics

Based on the ten metrics and the scores defined before, every tool is rated. The scores are shown in table 6.2. Then, for every metric, an argument is given for why the tool received this score.

Table 6.2: *The table shows the scores of every tool for the metrics defined before.*

Metric	CompuRacer	Turbo Intruder	Sakurity Racer	Race the web
Instructions	++	+	+/-	+
Installing tool	+	++	+	+/-
Configuring tool	++	+/-	++	++
Control options	+	+	- -	++
Importing requests	++	+	+	+
Storing requests	++	++	- -	+/-
Storing configuration	++	++	-	+
Sending requests	+	+	-	+
Response-aggregation	+	++	- -	-
Future-proof	++	++	+	+

1. **Instructions**

   - The CompuRacer gets the best possible score because it comes both with a README [ref] on how to install, configuration and run the tool, and also with a manual [ref] on how to use the tool on an included vulnerable web application.
   - The publication of the Turbo intruder also includes a README on Github [ref], a talk on a security conference [ref] and an entry in a blog [ref] on how it works in general. These documents do not equal a full-blown manual, and therefore the score is not maximal.
   - The Sakurity Racer only contains basic instructions on its Github page [ref] and shows its usage in a security blog [ref] and therefore gets a limited score.
   - Finally, Race the web is presented in the Hackfest infinity security conference [ref] and contains an elaborate README on its Github page [ref].

This document gives the tool the second to highest score.

2. **Installing tool**

   - The CompuRacer requires the user to download the sources from Github, install Python 3.7 and the dependencies, and to load the browser and the Burp extensions. These are quite some steps, but as every step itself is easy, it scores well.
   - Turbo Intruder only needs to be installed via the Extension tab of the Burp Suite and then is ready to use. This simple method yields a maximal score.
   - Sakurity Racer requires the user to download the source from Github, install NodeJS and the dependencies, start the server and load the extension in the Chrome browser. It is a similar process as for the CompuRacer tool and therefore, the score is the same.
   - Race the web needs to be downloaded from Github, then the tester must install Go, compile the program and install the dependencies. These steps are the most complicated of all tools, and therefore, it gets an average score.

3. **Configuring tool**

   - CompuRacer requires no setup after every part is installed and therefore gets the maximal score.
   - Turbo Intruder requires no setup when a simple brute force tactic executed, but when testing for race conditions, it requires the user to get a different execution script. As this requires some knowledge about the internal workings of the tool, this is not a trivial task. That is why it gets an average score.
   - Sakurity racer and Race the web both require absolutely no configuration before requests can be added to the tool and therefore get the best possible score as well.

4. **Control options**

   - The CompuRacer can be fully controlled and customised using its elaborate Command Line Interface (CLI) and therefore gets a good score.
   - The Turbo Intruder can both be used within the GUI of the Burp Suite and outside using a CLI. As its integration with Burp makes the advanced control and aggregation possible, its functionality is reduced greatly to a

level equivalent to the Sakurity Racer when we use this mode. That is why it does not get the maximal score for this metric.

- The tester can only start the Sakurity Racer and forward requests received from the browser in parallel in a fixed way. The browser extension does send the parallelisation value to the NodeJS server, but the tester cannot change this value other than by updating the extension source code. Therefore, the tool gets the lowest score.
- The behaviour of Race the web can be changed a bit more easily using a configuration file and therefore it gets the second to the lowest value.

5. **Importing requests**

   - The CompuRacer supports receiving requests via the Burp Suite, the Firefox and Chrome extensions and even by adding the request-content it to the JSON state file of the tool. Consequently, it gets the maximum score.
   - Similar to the CompuRacer, the Turbo Intruder supports adding requests via any request-view in the Burp Suite interface. However, there are no other methods to add requests. Thus, it does not get the maximum score.
   - Sakurity Racer also has one dynamic-method of adding requests using the browser extension and therefore gets the same score as the Turbo Intruder.
   - Race the web, on the other hand, supports two methods of adding requests. Manually via a configuration file and also via the API. As it does not support multiple dynamic methods, it only gets the second to highest score.

6. **Storing requests**

   - Both the CompuRacer and the Turbo Intruder support storing requests, sorted viewing and comparing of stored requests and therefore get the maximal score.
   - For the Sakurity Racer, only the lowest score is applicable as it immediately forwards requests and does not store them in any way.
   - Race the web uses configuration files for adding requests, and we can view these files as straightforward not-searchable storage of requests. The requests that the tester sends via the API will not be stored. Therefore, it gets an average score.

7. **Storing configuration**

   - These results are similar to the scores to the 'Storing requests' metric. Both the CompuRacer and the Turbo Intruder support advanced configuration of the tool behaviour and this behaviour can easily be imported and exported for different tests. Thus, they receive the best score.
   - The configuration of the Sakurity Racer is hardcoded in the application but can be altered by changing certain variable values in the source code. Therefore, it only gets the second to lowest score.
   - Race the web is configurable using its configuration files, but as these same files also contain the requests of the attack, importing and exporting settings between tests is not trivial. Therefore, it does not get the highest score.

8. **Sending requests**

   - CompuRacer supports the creation of batches of requests that the tester can send at once. The tester can also configure the tool to sent requests with arbitrary parallel and sequential duplication, and also delay requests from the start of the attack with millisecond precision. By altering the asynchronous library used to send requests, it supports the option to synchronise the sending of the last byte of the body of parallel requests. This alteration improves the synchronisation of the arrival of parallel requests at the server-side, especially when the connection to this server is slow or unstable. Therefore, we award the tool with the maximal score.
   - Turbo Intruder contains the same last-byte-sync functionality, and by using the configuration scripts, the tester can also arbitrarily delay requests. Finally, the request headers and body can be altered to some extent using these scripts. As it does not support sending different requests that were gathered by the Burp Suite in parallel, it only gets the second to highest score.
   - Sakurity Racer can only send the same request in parallel without any delays or additional last-byte-sync and therefore gets the second to lowest score.
   - Race the web can send multiple different requests in parallel, but no timing options or last-byte-sync are available. It, therefore, gets the same score as Turbo Intruder.

9. **Aggregation of responses**

   - For every test, CompuRacer will calculate the response code, header and body length statistics for all responses to quickly spot obvious anomalies. Next to this, the tool groups all responses by header and body similarities. The grouping-behaviour can be changed, and the tester can also use the tool to compare two groups field-by-field. In these groups, response-codes are highlighted using colour-codes and HTML bodies are made available for inspection via the built-in server. As this behaviour is only customizable in a limited way, it yields no more than the second to highest score.
   - For the Turbo Intruder, Python scripts can be used for endless options for display and aggregation of responses of interest. Next to this, it also has a built-in algorithm for showing only results of interest. These results can be sorted and viewed afterwards. We can only award the highest score to this tool because of the massive amounts of options alongside great defaults.
   - Sakurity Racer has not aggregation at all and just prints all responses to the terminal output. Therefore, it gets the lowest score.
   - Race the web has a basic method of aggregation that groups all duplicate responses. We found that the grouping function does not take into account the undefined ordering of the Go dictionary used for the HTTP headers. Therefore, it is only able to group duplicate responses by chance. Thus, we can give no better than the second to lowest score.

10. **Future proof**

    All tools are open source and available on GitHub and therefore obtain a good score.

    - However, only the CompuRacer and the Turbo Intruder have been updated in the last six months. Actually, the developers of both of these tools have also created them in the last six months. As both the CompuRacer and the Turbo Intruder are developed by employees of security testing companies (Computest[1] resp. PortSwigger) and published on their GitHub accounts, we estimate that both will be well-maintained.
    - Sakurity, a security company, created and published Sakurity Racer, but provided the last updates only two years ago.

[1] The author of this research has developed CompuRacer, but it has also been audited by a senior security tester at Computest regarding security vulnerabilities.

- For Race the web, also no significant updates seem to have been made in the last two years.

6.1.3 Conclusions

Based on the functionality and usability scores of the tools, we can spot some trends. For one, the CompuRacer and Turbo Intruder tools score much better than the other two tools according to most metrics, and the Sakurity Racer gets the lowest scores.

The reasons why a tester could prefer the Sakurity racer over the other tools are for requiring no configuration and the ability to easily send some parallel requests directly from the browser to a web app. As the CompuRacer has an improved version of the same functionality, this is not a real advantage. Race the web gets moderate to good scores and is the only tool that has advanced API integration. A tester could, therefore, use it in automated tests, and this is an advantage worth mentioning. As most race conditions often require some manual effort to discover, trigger and exploit, this only seems usable for already known race conditions and therefore is deemed to be of limited value during security tests.

The Turbo Intruder comes much closer to the CompuRacer when comparing the scores. No real difference can be observed in the scores, even when we use the weights of the different metrics to differentiate between them. The Turbo Intruder seems to be the better option when a single-click installation, full integration with the Burp Suite and advanced aggregation of responses is valued highly. CompuRacer scores better regarding its elaborate instructions, easy configuration, the number of different sources to get requests from and the ability to send multiple different requests in parallel. As both tools still get a good score when the other tool outperforms it, we cannot point to an absolute winner. Based on the strengths of tools and the particular use case, a tester should make the final decision regarding the preferred tool.

6.2 Evaluation - Toolset performance

The toolset should have a good and consistent request-sending performance in order to provide the security tester with the highest possible chances to trigger and exploit race conditions. To compare the performance of the tools, we have defined four metrics and set up a practical test. The first two metrics cover the raw performance (speed), and the other two metrics cover the real-life performance (triggering race conditions). For the practical test, we will target the voucher app that was introduced in section 3.2.3. The web app is run using four different configurations: locally, with and without a proxy to simulate real-world latency, and also remotely on an Amazon EC2 microserver in Paris. A script is created to support the gathering of data for the performance metrics.

In this section, first, we define the test metrics. Then, we describe the test setup and script. Last, we show the results, discuss them, and draw the conclusions.

6.2.1 Definition of metrics and scores

Below, we will elaborate on what the metrics consist of, why they are important to evaluate performance, and how to evaluate them on a tool.

1. **Raw performance**

   Here, we look at the speed at which a tool can deliver parallel requests to the application. This data is valuable because the parallel requests have to arrive at the application within the race window in order to cause a race condition to occur. Therefore, a robust tool should consistently deliver parallel requests with a time difference that is as short as possible.

   As tools might not construct TCP packets and HTTP requests in the same way, this could result in different handling of these packets between the client and the target application. That is why two metrics are defined that both measure the time difference between requests at a distinct measuring place:

   1.1. **Client** - The time between packets as measured when they arrive at the network interface that is used to send the packets to the target application.

   1.2. **Application** - The time between packets when the web application processes them.

   Our experience shows that an acceptable time difference for staying within the race window varies between applications. In a complex and busy application

where the request involves external database access, the value is much higher (order of 100's of milliseconds) than for a simple and almost idle application that only accesses some internal storage (order of 1 millisecond or less). Regardless, a lower time difference is likely to yield better results. Therefore, for this metric, a good score would be given to a tool that delivers a significant portion of its parallel requests to the web application logic within a short amount of time and a low variety.

2. **Real life performance**

   Here, we look at the actual number of race conditions that a tool can trigger at the application. This aspect, naturally, is essential to a race condition testing tool, and it should give consistent and good results. As too quickly sending requests can also result in Denial of Service (DoS) - related errors at the server-side, we should also look into this aspect. That is why two metrics have been defined that measure the voucher usage ratio and the number of success codes:

   2.1. **Voucher usage ratio** - This is the number of vouchers that are used compared to the total number of successful requests. More specifically, this is the number of successful requests divided by the number of used vouchers. If this value is one, no race conditions occurred, and every voucher is just redeemed once. If this value is two, every voucher is redeemed twice on average due to race conditions.

   2.2. **Number of success codes** - The is the total number of success codes that are returned by the application. As we will always send the same number of requests, this value directly indicates how many errors occurred at the server side.

   This metric is evaluated based on the number of race conditions the tool causes to occur at the web application compared to the total number of parallel requests: the voucher usage ratio. As requests that trigger unexpected error states of the application are not desirable (and could be part of an attack) but are also not directly related to the ability to cause race conditions, we measure these error-causing requests separately and will exclude them from the calculation of the ratio.

   We expect that there is a strong correlation between the scores of the tools regarding this metric and the raw performance metrics. However, we still include this metric, because the expected correlation is not proven or fully understood and therefore, it is an adequate verification method in itself. We award an excellent score when a tool has a high voucher usage ratio with a low variability

while causing as little errors on the server side as possible.

Notes For the first metric, the target web application must support logging of the time when a request is handled by the application logic handles. For the second metric, it should be possible to decide from responses how many race conditions occurred at the server. For both metrics, the target web app should not only be run at the local host as this is not a very realistic situation. Additional tests should be executed remotely or by using a proxy that causes a realistic amount of latency between the client and the server.

6.2.2 Performance test setup

In order to rate the tools according to the metrics, a test setup is required to acquire the necessary amount of data. First, we will elaborate on how the tools and the target web app is set up. Then, we explain the proxy that we use between the client and the server, and we show how the test automation script works. The complete test setup is illustrated in figure 6.1.

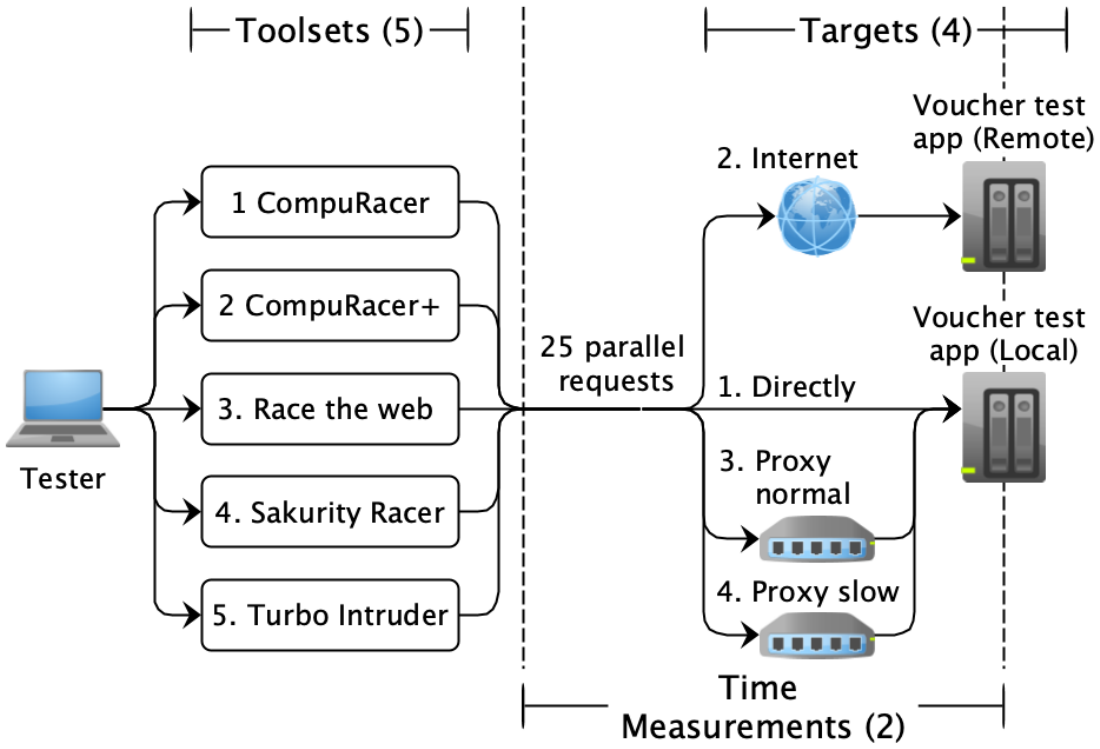


Figure 6.1: *The figure shows the complete test network setup using five tools and four environments (targets). It also indicates the two places where the time-difference measurements are computed.*

Setup of tools and target web app

Both the four tools and the target web app will be run locally on a MacBook Pro from 2013 (macOS High Sierra). CompuRacer will be tested twice for every set setup. Once with the last-byte-synchronisation enabled (CompuRacer+, CR+ or CR_lbs) and once with this function disabled (CompuRacer or CR). This difference allows us to test the effectiveness of this particular function as well and results in five tools in total. In the results, we abbreviate Turbo Intruder to TI, Race the web to RTW and Sakurity Racer to SR.

As only the CompuRacer and RTW support the sending different requests in parallel, we only tested the parallel sending of the same request. This request is first added to the tool under test and then is sent 25 times in parallel. We record the time-differences between requests at the client and the web server inside the application. We repeat this process 15 times per tool. Between tests, we restart the application and reset its database to avoid any interference. In the next paragraphs, we discuss the process to add and send requests with the RTW, SR and TI tools.

Race the web For this tool, we have created a TOML (Tom's Obvious, Minimal Language) configuration file for all setups. The file contains the parallel duplication amount and the request-content itself. We start the tool with this file as the only argument, and then it sends the parallel requests.

Sakurity Racer For this tool, there is no configuration file. The parallel duplication amount is hardcoded into the sending application. For every test, the Chrome extension has to be used to intercept the request of interest and forward it to the sending application. This application then immediately sends the requests in parallel to the target web app.

Turbo Intruder For this tool, we should first load it with a script that optimises the sending for triggering race conditions. In this script, the parallel duplication amount is also indicated. Then, the requests of interest should be intercepted using the Burp Suite proxy. From the proxy history, we send the request to the TI extension. The extension can then send this request at the press of a button.

Target functionality in web app We test the tools on the self-developed vulnerable voucher app. The most interesting redeem functionality is targeted: redeeming

a multi-use coupon (code: COUPON3) multiple times in the 'insecure' setting. This coupon can be redeemed 100 times. The 'insecure' setting contains a TOCTOU bug, but the race window is tiny as there is almost no logic between check-transaction whether a voucher is usable and act-transaction that uses the voucher. As this coupon will be redeemed 25 times, when no race condition occurred, this would result in 75 leftover usages. When more than 75 usages are left, and no errors occurred, we know that a race condition in the voucher app has taken place.

Note about test interference We keep an eye on the CPU and RAM usage of the MacBook during the three local tests. This check is necessary because, as already mentioned before, a race condition is exceedingly dependant on the exact timing of requests. Slow performance of the testing system will influence both the tools and the test web app in unexpected ways, and we will, therefore, monitor it. When the CPU usage rises above 50% on average, the particular test is postponed or re-executed.

Setup of proxy and remote server

As mentioned in the metric notes above, a test that only consists of a local setup with a server and client on one system or one local network is not very realistic. In a real-life setup with the server being in another city, country or continent, latency and packet loss become significant factors that we also have to include in the test.

That is why, after the test is executed once with a local server; both a configuration using a random-delay-proxy called ToxiProxy (version 2.1.4) developed by Shopify (2019) is set up, and also a configuration using an Amazon EC2 microserver instance in Paris. Note that each test is executed from the west of the Netherlands and all relative latencies are also calculated from this location. More concretely, using the remote server and the proxy three additional tests are added which adds up to 4 test environments in total. In figure 6.1, the test network setup is illustrated and below, every environment is described in more details:

1. **Local server** - A server on the same local network is simulated: a delay of 1 ms or less and jitter at microsecond-level is to be expected.
2. **Remote server** - An Ubuntu microserver in Paris is used: a delay of about 20 ms and jitter of at most 5 ms is to be expected for every packet up- and downstream as Reinheimer and Roberts (2019a) indicate. This results in a latency of 15 to 25 ms.

3. **Proxy normal** - A server in Los Angeles is simulated: a delay of 250 ms and jitter of at most 50 ms is added to every upstream request. So every request will have a latency of 200 to 300 ms as Reinheimer and Roberts (2019b) indicate.
4. **Proxy slow** - A horrible environment is simulated: a delay of 1500 ms and jitter of at most 500 ms. So every request will have a latency of 1000 to 2000 ms [2].

As exploiting race conditions is not expected to depend on downstream latency (server -> client), this latency is not included for the proxy setups. The following commands were used to start Toxyproxi, create the two different proxy setups to the testing web app (at `127.0.0.1:5005`) and set the 'toxics' (latency and jitter) of these proxy setups:

```
$>  toxiproxy-server &
$>  toxiproxy-cli create delay_normal -l 127.0.0.1:5006
    ↪ -u 127.0.0.1:5005
$>  toxiproxy-cli create delay_high -l 127.0.0.1:5007 -u
    ↪ 127.0.0.1:5005
$>  toxiproxy-cli toxic add delay_normal -t latency -a
    ↪ latency=250 -a jitter=50 -u
$>  toxiproxy-cli toxic add delay_high -t latency -a
    ↪ latency=1500 -a jitter=500 -u
```

Setup of test automation script

As shown in figure 6.1, the test as a whole encompasses using five different tools to send 25 requests in parallel via four different (proxy) setups, and the complete test is repeated 15 times. This results in 20 different setups repeated 15 times to result in 300 individual tests. For every test, the time differences between requests have to be gathered from two different locations and also the number of used vouchers, and error responses have to be saved. Next to this, the logging, database and application have to be reset between tests. As it would require significant effort to gather this data by hand, most aspects of this process have been automated using a Python script.

The source code of the script can be found on a GitHub repository of the author[3].

[2] Under normal circumstances, these characteristics are not found between any two servers on earth, but an awful network setup is still likely to yield these results.

[3] Link to the repository: `https://github.com/RobvEmous/WebAppRaceConditionTesters`

The script contains four stages per test and repeats this 15 times until all results are gathered. Then, it creates a summary file with all time-differences, statistics and redemption results. The script should be re-run for all tools and different proxy configurations. The different stages are described below. The output of the tool for one illustrative test is shown in figure 6.2

```
Provide input <name> <type> (type = f/n/s) TEST n

> Did not find result input 1 -> perform new test
> Restarting app and clearing logs.. Done.
> Press ENTER, wait for text: "Capturing on 'Loopback'" and then send all parallel requests.

> Recording traffic for 8 seconds..
Capturing on 'Loopback'
2 packets captured
Done.
> Reading app logs.. Done.
> Parsing and filtering pcap..
reading from PCAP-NG file csvs/TEST/n/TEST_n_1_rec.pcapng
Done.
> Results: total 20X return codes? 20
> Results: total vouchers used? 10
> Writing results input file.. Done.
```

Figure 6.2: *The console output for running the test script once with illustrative data. All input is coloured green and all output from the Wireshark recorder and converter is coloured red.*

Configuration stage The user first indicates the name of the tool and the proxy configuration that will be tested. For the proxy configuration, 'f' (fast, no proxy), 'r' (remote), 'n' (normal proxy) and 's' (slow proxy) can be used. The local request recorder uses this information to optionally filter any requests from the proxy to the web app. Only the time-differences of requests from the tool to the proxy are measured. Then, the script will first create the necessary folders for result storage.

Preparatory stage After the setup, several commands are executed on the server of the voucher app via SSH. These commands empty the log files of the web server and the application, restart the application and reset the database. Then, the script waits for an ENTER-press. During this time, the user should manually prepare the tool for sending parallel requests via the current proxy configuration.

Execution stage Then, he presses ENTER, and the script will start a Wireshark recorder to save the request differences at the client for 8 seconds. When the recording is started, it will indicate this fact. Only after this indication, the user can safely start the parallel sending of requests. All requests send before this moment or after 8 seconds will not be recorded.

Evaluation stage After 8 seconds, the script will stop the recorder, save the results and again access the server of the voucher app via SSH to read the log files. It will extract the arrival-timestamps of the 25 requests at the web server and the application. After this, it will ask the user how much success codes were returned and how much vouchers were redeemed. This data is stored, and the tool returns to the preparatory stage for the second test.

6.2.3 Results

According to the test setup laid out before, the test is executed, and the results are gathered. According to the metrics as laid out before, we are primarily interested in the expected value (mean or median) and the variance of the performance regarding the two metrics as compared between the five tools.

Presentation of results for both metrics First, in order to get a general idea of the answers to the metrics, histograms were created of all results. For metric 1, the time-difference between requests at the client and the application are plotted. For these differences, smaller is better. The distribution of these time-differences is assumed to be log-normal as most differences will be tiny: in the order of milliseconds or microseconds, but several outliers were also found of up to multiple seconds in duration. That is why we have transformed this data by taking the base-10 logarithm of all data points. The log-normality assumption is not validated because it is only used for a more comfortable visual comparison of the results. For metric 2, the success codes and ratios are plotted on a histogram with regular bin-sizes.

In the histograms, a number of additional statistics are listed. They indicate the expected value using the mean and median, and the variance using the standard deviation of the mean, and the first and third quartile (Q1 and Q3) of the data. Unfortunately, these statistics can only give us a hint as to which tool performs better, but proper statistical tests are required to make any firm claims about the performance. We will explain these tests below. Note that afterwards, as encouraged by any good resource on statistics, we verified whether the histograms, means, medians, variance, etc. matched the conclusions indicated by the output of the statistical tests. No anomalies were spotted, and that is why these individual comparisons will not be discussed.

Note that that the results for all five tools are plotted in histograms, but as this resulted in page-size figures, only the overview-histograms are shown here and the full figures are moved to the appendices. In the remainder of this section, we will

first show the overview-histograms for all setups of all test metrics, and secondly, discuss the executed statistical tests and their results. Lastly, the results are summarised.

Additional statistics for both metrics As stated above, several statistical tests were conducted on the data to compare the results of all tools and draw appropriate conclusions. As every test has specific requirements with regards to the data, and the point in the process it is used, this is not a simple process. The appropriate process that is followed by us in the statistical analysis contains four sets of tests and is illustrated in figure 6.3. We explain each step below. For all tests, the significance value is set at $a = 0.05$, as this is a commonly used value.

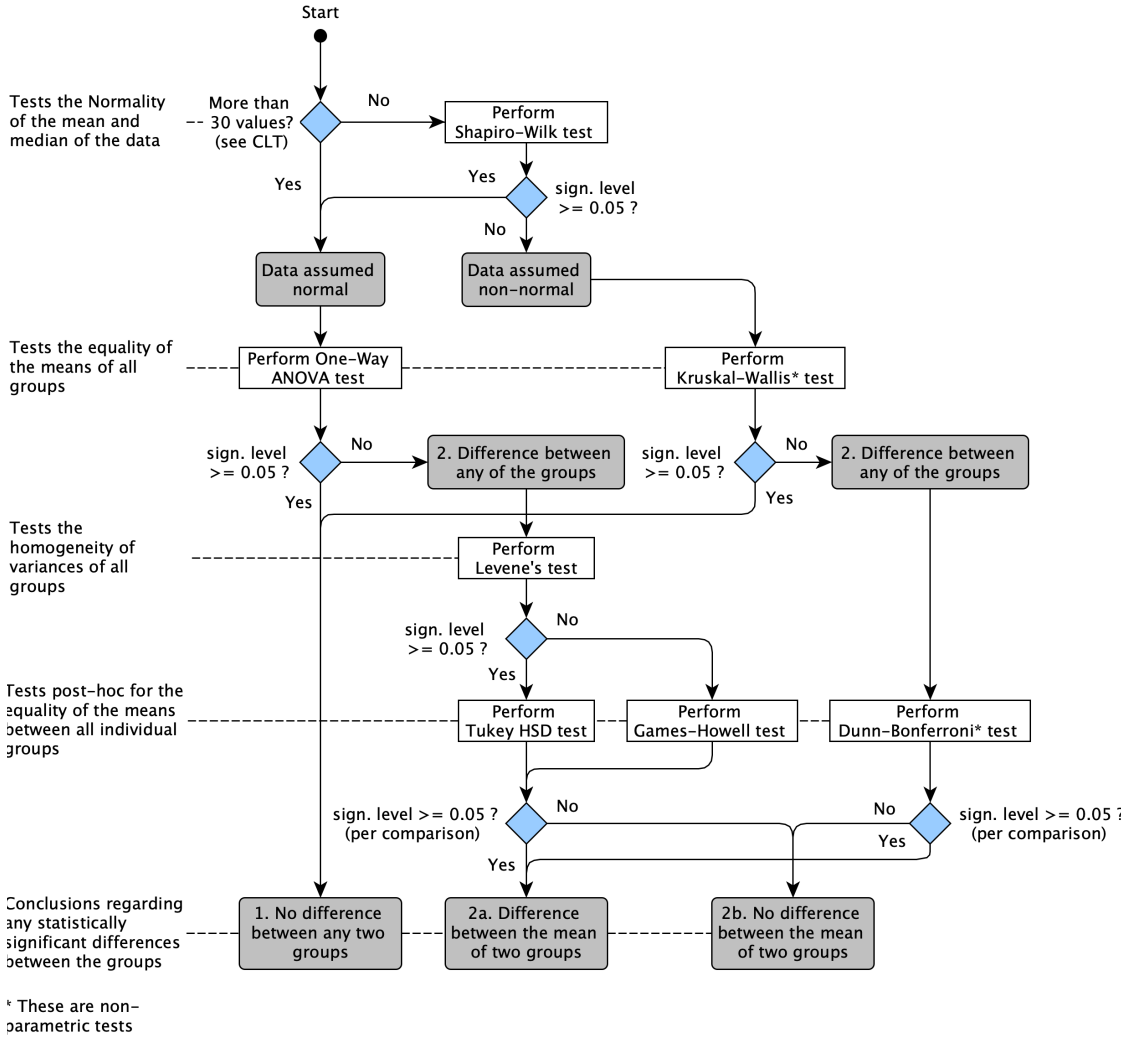


Figure 6.3: *The figure shows the process of using statistical tests to compare the results of the tools.*

1. **Test for normality** - As most tests assume the normality of the mean and median of the data, we first test this quality. Based on the Central Limit Theorem (CLT), we can already assume that these values are normally distributed when more than about 30 samples are used per group (tool). For the time-difference measurements for metric 1, this requirement easily holds as we collected 360 measurements per tool (15 tests of 24 time-differences). However, for the voucher redeem results for metric 2, we have only collected 15 samples per tool. That is why we have to use a specific test for the normality of the data in this case.

   As Kar and Ramalingam (2013) point out, 30 samples is not a magic number that works in all cases. However, for our test, we either fail or greatly surpass this minimal requirement. That is why this value, often cited in works on this subject, is still used and deemed sufficiently valid.

   We use the Shapiro-Wilk test, which is specifically designed for estimating the distribution of small sample sizes of $N < 50$ (Shapiro and Wilk, 1965). The Kolmogorov-Smirnov Test (or KS test) could also be used for this purpose as according to (Halley and Inchausti, 2002, p. 521), this test *"measures the maximum departure of the data from a theoretical cumulative distribution function"* and this can be used for an evaluation of the 'goodness of fit' of the normality of the results (Ball, 1960). This version of the KS-test is very similar to the Lilliefors test (Lilliefors, 1967). However, it is less focused on small sample sizes and, therefore, the Shapiro-Wilk test seems to be the better option in our case.

2. **Test for equality of means of all groups** - Now, we can perform a test that estimates whether the means of-of the groups are the same (hypothesis 0), or at least one group-mean is different (hypothesis 1). If the outcome of the normality test is positive (h0), we use the One-Way ANOVA test (Kim, 2014) and otherwise, we use the non-parametric Kruskal-Wallis test (Breslow, 1970). Although the One-Way ANOVA is quite robust to some deviations to the normality of the data, the latter test does not require this assumption at all and therefore seems to be a better fit data. The One-Way ANOVA test also requires the variances of the groups to be equal, but this requirement is relaxed when the group-sizes are roughly equal. This is the case for our tests.

   If this test returns positive, no group-mean is said to be statistically different from the others, and we are done. Otherwise, we continue with the next steps to find out which groups in specific deviate from the rest.

3. **Test for homogeneity of variances** In the case of normally distributed data, we need to know whether the variances of the different groups are equal (ho-

mogeneous) to be able to select the appropriate next test. We use Levene's test for this end, which tries to confirm the hypothesis that all variances are equal.

4. **Post hoc test for equality of means between groups** We now know that at least one-group mean is different from the others, but we do not know what groups(s) are different. That is why the fourth set of tests is required.

   - The Turnkey HSD test requires normally distributed data and homogeneous variances between groups (Tukey et al., 1949).
   - The Games-Howell test requires normally distributed data, allows for differences in variance (Games et al., 1981), but requires at least 15 samples (Shingala and Rajyaguru, 2015) to make accurate predictions. Note that the latter requirement is always satisfied with our dataset.
   - The non-parametric Dunn-Bonferroni test does not have any requirements regarding the distribution of the data (Dunn, 1961).

   Although the last-mentioned test could hypothetically be used for all data, the relative statistical strength of these tests varies greatly (Ruxton and Beauchamp, 2008). That is why we will always use the strongest applicable test in each case (they are listed from strongest to weakest).

Based on these tests, we can see that figure 6.3 concludes with two primary outcomes for a metric: 1) all groups (tools) have the same mean, or 2) at least one group has a different mean. Within the secondary outcome, we will pursue to find out what groups differ from the others, make a ranking of the means (lowest to highest) and make appropriate conclusions.

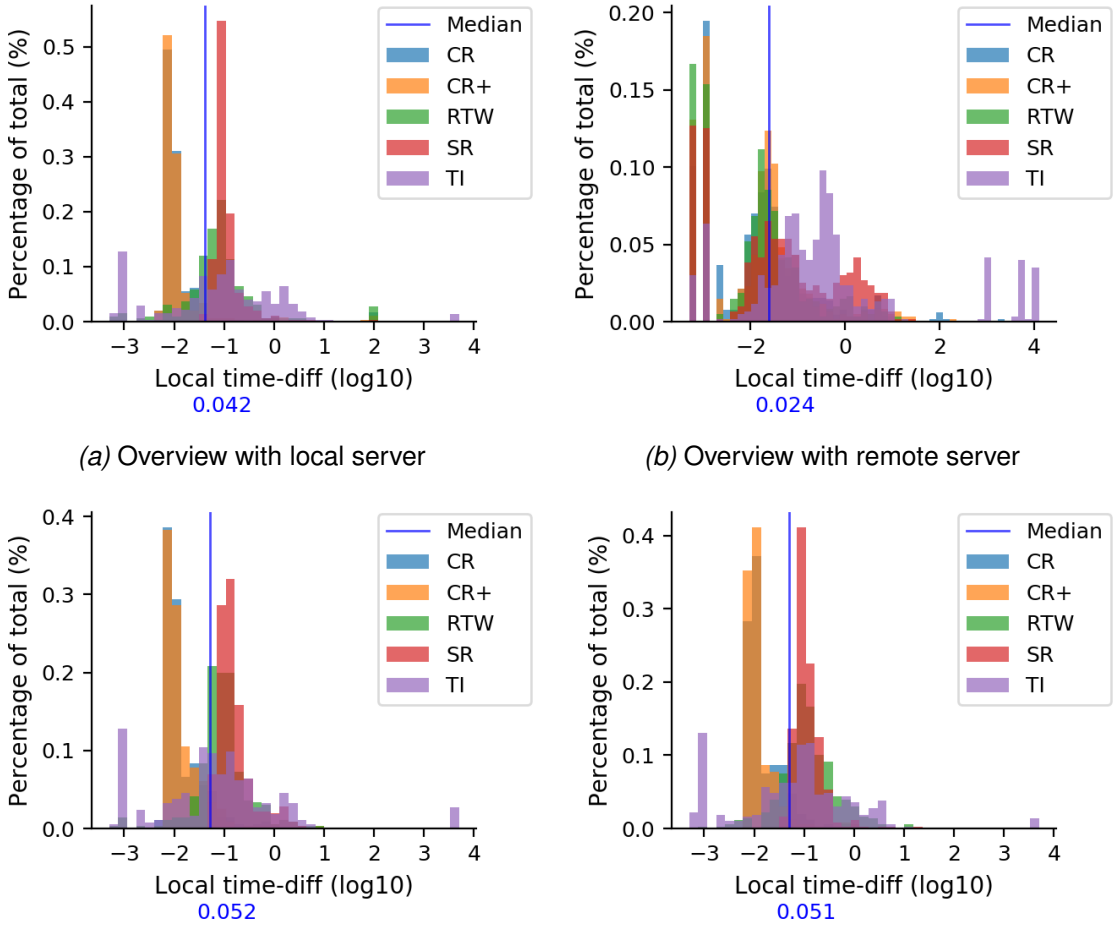
Next, we will show and discuss the histograms and the statistical tests for each of the four (sub) metrics separately. The raw results to all tests that we performed and the complete results of the statistical analysis can be found on the earlier mentioned GitHub repository of the author[4].

Metric 1 - Test 1 - local time-difference

Here, the results of metric one, test one are shown. In this case, the time-differences are measured between the requests at the client. This measurement is done before any proxy can interfere, and that is why we expect that this will result in very similar data for all three proxy configurations. However, we will still show the results of all

[4]Link to the repository: `https://github.com/RobvEmous/WebAppRaceConditionTesters`

four configurations. The combined histograms for all tests are shown in figure 6.4. A full display of the individual histograms and statistics can be found in the appendices in section D.1 (page 179). We can make several observations:



(a) Overview with local server

(b) Overview with remote server

(c) Overview with normal proxy

(d) Overview with slow proxy

Figure 6.4: *The figure shows the overview histograms of the* ***local time-differences*** *between requests of all tools for all four configurations. The median of the complete datasets is calculated and indicated in base-10. Please note that the x- and y-scale of the figures are not necessarily equal.*

- Firstly, it is clear that the results of all setups are very similar except for the one with the remote server in figure 6.4b.
- Secondly, for the three setups with similar results, the results between the tools themselves vary greatly (visually), while this difference is almost completely gone in figure 6.4b.
- Thirdly, the mean and variance of TI and RTW, and CR and CR+ seem to be similar to each other in all setups.
- Finally, regarding the CPU usage of the client system, this value was observed to be much lower during the *Remote server* test than the other tests, this is not

a surprise as for these tests, not only the toolset ran on the client, but also the server itself.

Next, we will discuss the results of the individual tests more in-depth, and finally, the conclusions are drawn.

1. **Test for normality** - For every tool and setup, we have performed 15 tests in which we gathered 24 time-differences resulting in 360 data points. Based on the CLT, we can assume that the mean of this data is normally distributed for all setups and tools. That is why no statistical tests are required (see table 6.3, row 2).
2. **Test for equality of means of all groups** - As all data is normally distributed, we can use the One-Way ANOVA test to find out whether there is a statistical difference between means of the results of the tools. In table 6.3, row 3, the results of this test are shown. For all setups, the test results show statistically significant differences.
3. **Test for homogeneity of variances** - As the One-Way ANOVA test showed significant differences between the results of all groups in all setups, we performed Levene's test for the homogeneity of variances. In table 6.3, row 4, the results of this test are shown. It shows statistically significant differences in the variances for all setups.
4. **Post hoc test for equality of means between groups** - As Levene's test resulted in different variances between the groups for all setups, we used the Games-Howell post hoc test to group the results based on statistically significant differences between their means. In table 6.3, row 5, the results of this test are shown.

Table 6.3: *The table shows the performance test results for the* ***local time-differences****. The degrees of freedom for all tests are included in the first row. The results to the normality, equal means and equal variances tests are formatted as follows: '[H0 confirmed]: [Test name] = [F/statistic], [p-value]*[5]*. In the last row, we indicate the post hoc test used, list the groups and associated p-values. We sorted the groups (top to bottom) and the tools (left to right) in ascending order based on their mean values.*

	Local server	**Remote server**	**Normal proxy**	**Slow proxy**
Df	4, 1759	4, 1794	4, 1795	4, 1795
Normal	Yes	Yes	Yes	Yes
Equal means	No: A = 148.628, 0.000	No: A = 107.811, 0.000	No: A = 231.286, 0.000	No: A = 230.365, 0.000
Equal variances	No: L = 93.851, 0.000	No: L = 11.556, 0.000	No: L = 121.294, 0.000	No: L = 176.929, 0.000
Post hoc grouping	G - 3 groups: (CR+, CR) 0.999 (TI, RTW) 0.720 (SR) 1.000	G - 3 groups: (CR, CR+, RTW) 0.764 (SR) 1.000 (TI) 1.000	G - 4 groups: (CR, CR+) 1.000 (TI) 1.000 (RTW) 1.000 (SR) 1.000	G - 4 groups: (CR+) 1.000 (CR) 1.000 (TI) 1.000 (RTW, SR) 0.986

Conclusions and discussion Based on the results as discussed above, we can draw the following conclusions regarding the local time-difference of the tools.

- **Group means** - Based on the results, we confirm that the means of the groups for all setups are not equal. Also, the mean of the performance of both CR and CR+ statistically significantly higher than the performance of the other tools except in the *Remote server* setup. In that case, RTW has the same mean. Next to this, the means of CR and CR+ are the same except in the *Slow proxy* setup. This outcome was to be expected as the only difference between these tools is the moment the last byte is sent to the server. That means that regarding metric 1.1, we can conclude that both CR and CR+ score similar or better than all other tools.
- **Local server interference** - As already indicated at the beginning of the section based on figure 6.4, there is a big difference between the results of the *Remote server* setup and the other setups for all tools except for the TI. This difference cannot be traced back to any difference in the test setup itself except for the location of the target server. In the first test, both the tool and the target

[5]The test names are abbreviated as follows. Normality test: S = Shapiro-Wilk; Equal means tests: A = One-Way ANOVA, K = Kruskal-Wallis; Equal variances test: L = Levene; Post hoc tests: T = Tukey HSD post hoc, G = Games-Howell, D = Dunn-Bonferroni (never used).

server are run on the same system and in the second test, the server is run on a remote server. Next to this, we observed a higher CPU usage during the *Local server* test. Therefore, we conclude that a locally run server interferes with the performance of the tools in such a way that the tools perform less and the time-differences increase accordingly. Apparently, this interference does not affect the tools equally. Therefore, the results of the *Remote server* are expected to be more reliable.

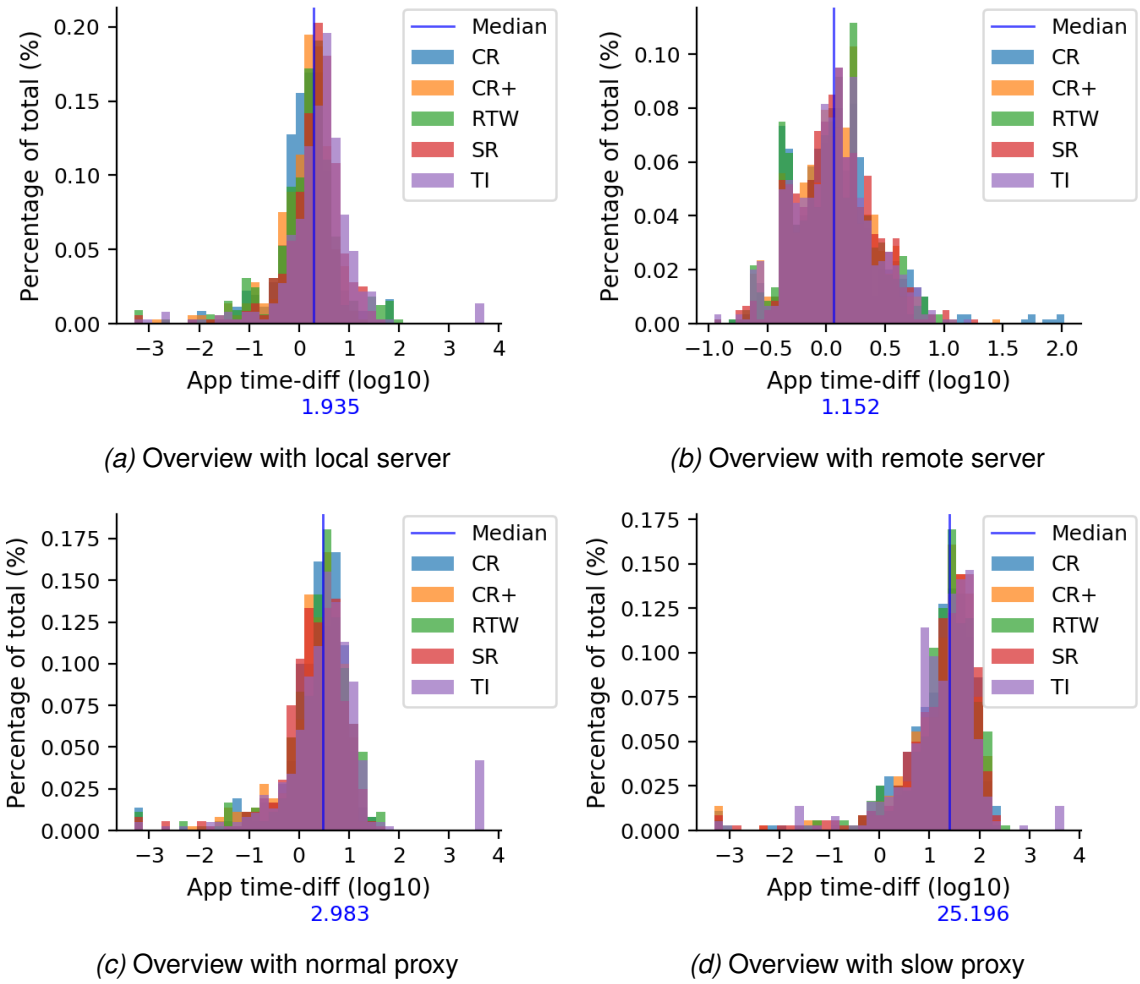
Metric 1 - Test 2 - Application time-difference

Here, the results of metric one, test two are shown. In this case, the time-differences are measured between the requests at the application. This measurement is done after the interference of latencies and jitter on the connection and any delay in the web server. That is why we expect that this will result in time-differences between requests that are much higher than for metric 1.1. The combined histograms for all test setups are shown in figure 6.5. A full display of the individual histograms and statistics can be found in the appendices in section D.2 (page 182). We can make several observations:

- Firstly, just like for the local time-differences, it is clear that the results of all setups are similar except for the one with the remote server in figure 6.4b.
- Secondly, for all tools in the overview histograms, the results are visually very similar to each other (high overlaps). As the scale is logarithmic, this does not mean that the results actually the same for all tools. The statistical tests will verify this.
- Thirdly, the mean and median of the different setups greatly differs with the *Remote server* being the lowest and, as expected, the *Slow proxy* being the highest.
- Finally, the variance is similar for all tests except for the *Remote server* test, which reports a variance (in the mean value) that is between two and three times lower.

Next, we will discuss the results of the individual tests more in-depth, and finally, the conclusions are drawn.

1. **Test for normality** - Just like for the local time-differences, we gathered 360 data points per group (tool). Based on the CLT, we can assume that the mean of this data is normally distributed for all setups and tools. That is why no statistical tests are required (see table 6.4, row 2).



(a) Overview with local server

(b) Overview with remote server

(c) Overview with normal proxy

(d) Overview with slow proxy

Figure 6.5: *The figure shows the overview histograms of the* ***application time-differences*** *between requests of all tools for all four configurations. The median of the complete datasets is calculated and indicated in base-10. Please note that the x- and y-scale of the figures are not necessarily equal.*

2. **Test for equality of means of all groups** - As all data is normally distributed, we can use the One-Way ANOVA test to find out whether there is a statistical difference between means of the results of the tools. In table 6.4, row 3, the results of this test are shown. For the *Local server* and *Normal proxy* setups, it shows statistically significant differences between the means of the groups. This is not the case for the other setups and, therefore, we assume no differences exist between the means of the groups.

3. **Test for homogeneity of variances** - For the *Local server* and *Normal proxy* setups, the One-Way ANOVA test showed significant differences between the results of the groups. That is why we performed Levene's test for the homogeneity of variances for the results of these setups. In table 6.4, row 4, the results of this test are shown.

4. **Post hoc test for equality of means between groups** - As Levene's test resulted in different variances between the groups for these setups, we used the Games-Howell post hoc test to group the results based on statistically significant differences between their means. In table 6.4, row 5, the results of this test are shown.

Table 6.4: *The table shows the performance test results for the* ***app time-differences****. The formatting of the table and the meaning of the contents is the same as in table 6.3.*

	Local server	**Remote server**	**Normal proxy**	**Slow proxy**
Df	4, 1755	4, 1787	4, 1795	4, 1795
Normal	Yes	Yes	Yes	Yes
Equal means	No: A = 16.645, 0.000	Yes: A = 0.641, 0.633	No: A = 2.647, 0.032	Yes: A = 0.522, 0.720
Equal variances	No: L = 2.668, 0.031	N/A	Yes: L = 0.117, 0.976	N/A
Post hoc grouping	G - 2 groups: (CR, CR+, RTW) 0.898 (SR, TI) 0.480	N/A	T - 2 groups: (**CR+**, CR, RTW, SR) 0.317 (CR, RTW, SR, **TI**) 0.133	N/A

Conclusions and discussion Based on the results as discussed above, we can draw the following conclusions regarding the application time-difference of the tools.

- **Group means** - Based on the results, there is no statistically significant difference between the group means in the *Remote server* and *Slow proxy* setups. In these cases, all tools perform equivalently. For the *Local server* and *Normal proxy* setups, two groups can be formed in which for both setups, CR, CR+ and RTW are in the group with the lowest mean values. That means that regarding metric 1.2, and similar to our answer to metric 1.1, we can conclude that both CR and CR+ score identical or better than all other tools.

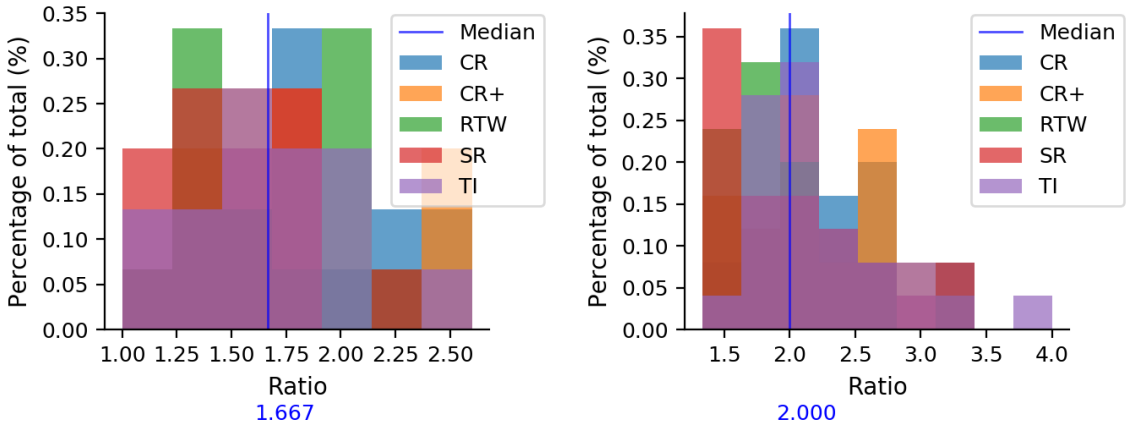
  Also, we estimated that the time-difference at the application is correlated with the ability to exploit race conditions. We will verify this claim by comparing these results to the results of metric 2.1 and 2.2.

- **Local server interference** - Just like for the local time-differences, there is a big difference in mean and variance between the results of the *Remote server* setup and the other setups. Based on the delay between client and server, the *Remote server* setup should perform anywhere in between the performance of the *Local server* and *Normal proxy*, but this is not the case. We estimate

that is, again, caused by client-server interference when both systems are run on the same system.

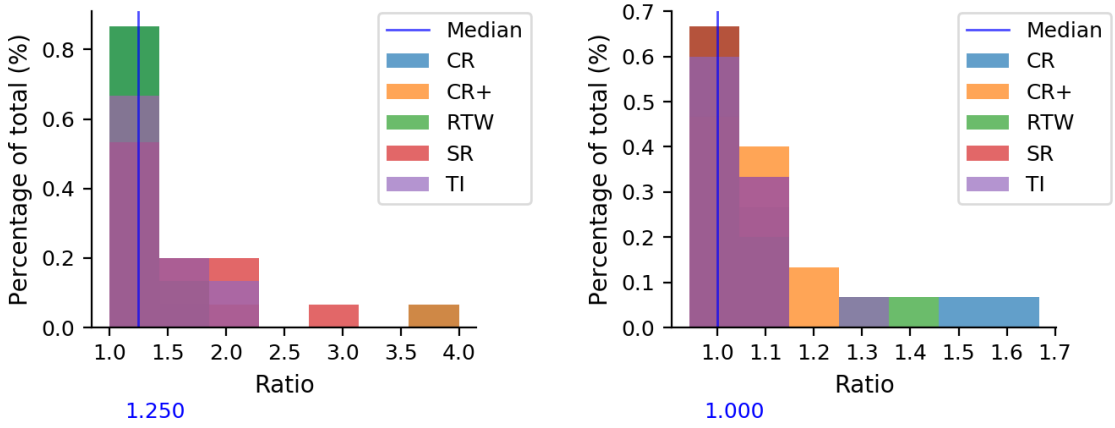
Metric 2 - Test 1 - Voucher usage ratio

Here, the results of metric two, test one are shown. In this case, we measured the voucher usage ratio: the total number of success codes divided by the number of vouchers used. A low number of vouchers used while the same number of success codes are returned indicates that more race conditions are triggered. Therefore, a higher value is better. The combined histograms for all test setups are shown in figure 6.6. A full display of the individual histograms and statistics can be found in the appendices in section D.3 (page 184). We can make several observations:



(a) Overview with local server

(b) Overview with remote server



(c) Overview with normal proxy

(d) Overview with slow proxy

Figure 6.6: *The figure shows the overview histograms of the* ***voucher usage ratio*** *of all tools for all four configurations. The median of the complete datasets is calculated and indicated in base-10. Please note that the x- and y-scale of the figures are not necessarily equal.*

- Firstly, the figures show a big difference between different setups. Except for the *Remote server* setup, the ratio seems to scale evenly with the latency and jitter of the setups. In the best setup, the median ratio is two, while for the worst setup, it is one. In the first case, every voucher was redeemed twice while in the second case, the vouchers were redeemed only once (no race conditions).
- Secondly, it seems like there are quite some differences between the performance of the tools. In the *Normal proxy* and *Slow proxy* setups, this difference seems rather small, but for the other two setups, the difference seems to be significant. In these setups, CR, CR+ and TI seem to perform better than RTW and SR. As the number of samples per tool is quite small at 15 we are not sure whether these observed differences are actually significant.

Next, we will discuss the results of the individual tests more in-depth, and finally, the conclusions are drawn.

1. **Test for normality** - As we gathered only 15 data points per group (tool). The CLT does not apply, and we need to perform the Shapiro-Wilk test to verify whether the results are normally distributed. The results of this test are shown in table 6.5, row 2). Based on this, we can conclude that only the results to the *Local server* setup are normally distributed.
2. **Test for equality of means of all groups** - For the *Local server* setup, we can use the One-Way ANOVA test to find out whether there is a statistical difference between means of the results of the tools. For the other setups, we have to use the non-parametric Kruskal-Wallis test. In table 6.5, row 3, the results of this test are shown. For all setups, it shows that there are no statistically significant differences between the means of the groups.

   This means that no other tests need to be performed, and we can conclude that for this test and for all setups, the groups have an equal voucher redeem ratio.

Table 6.5: *The table shows the performance test results for the* ***voucher redeem ratio****. The formatting of the table and the meaning of the contents is the same as in table 6.3.*

	Local server	**Remote server**	**Normal proxy**	**Slow proxy**
Df	4, 70	4, 70	4, 70	4, 70
Normal	Yes: S $\geq$ 0.890, 0.068	No: S $\geq$ 0.868, 0.032	No: S $\geq$ 0.475, 0.000	No: S $\geq$ 0.588, 0.000
Equal means	Yes: K = N/A, 0.281	Yes: K = N/A, 0.116	Yes: K = N/A, 0.078	Yes: K = N/A, 0.840
Equal variances	N/A	N/A	N/A	N/A
Post-hoc grouping	N/A	N/A	N/A	N/A

Conclusions and discussion Based on the results as discussed above, we can draw the following conclusions regarding the voucher redeem ratio of the tools.

- **Group means** - Based on the results, there is no statistically significant difference between the group means for any setup. This means that regarding the voucher redeem ratio, no tool seems to perform better than the other tools concerning metric 2.1.
- **Correlation between metric 1 and 2** - As the local and app time-differences (metrics 1.1 and 1.2) did show significant differences between the tools for most setups, it seems like the correlation between HTTP request time-differences and the race condition exploitation rate is not as strong as expected. We estimate that this is primarily due to other not-measured factors that also play a role in this process.

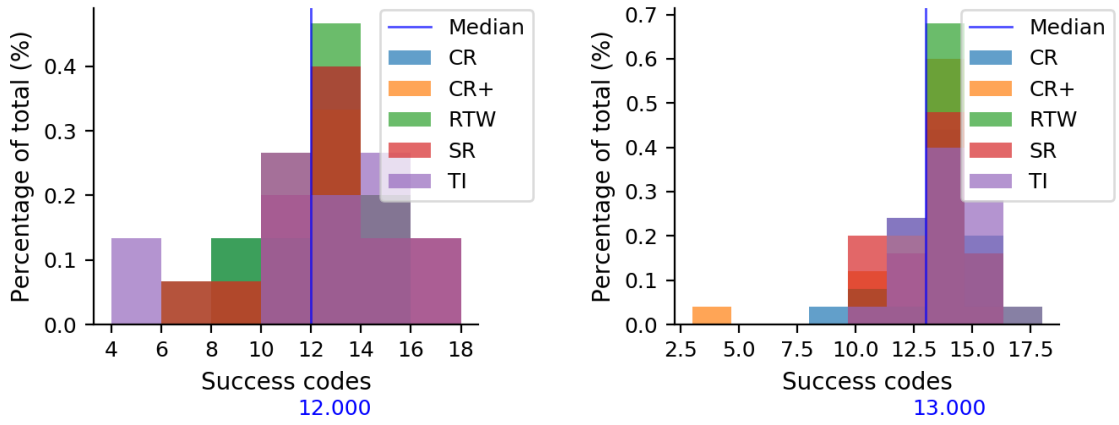
  For instance, we did not compare variation in the 24 measured time-differences within a single attack between tools. When the time-difference is low on average for both tools but varies more between requests for one tool, the slow requests might not trigger a race condition at all. The error rate, the busyness of the server or the implementation of the TCP connection might also be factors that influence the results. More research is required to validate this.

Metric 2 - Test 2 - Number of success codes

Here, the results of metric two, test two are shown. In this case, we measured only the number of success codes returned by the application. As 25 requests are sent, at most 25 success codes can be returned in which higher is better. This element is

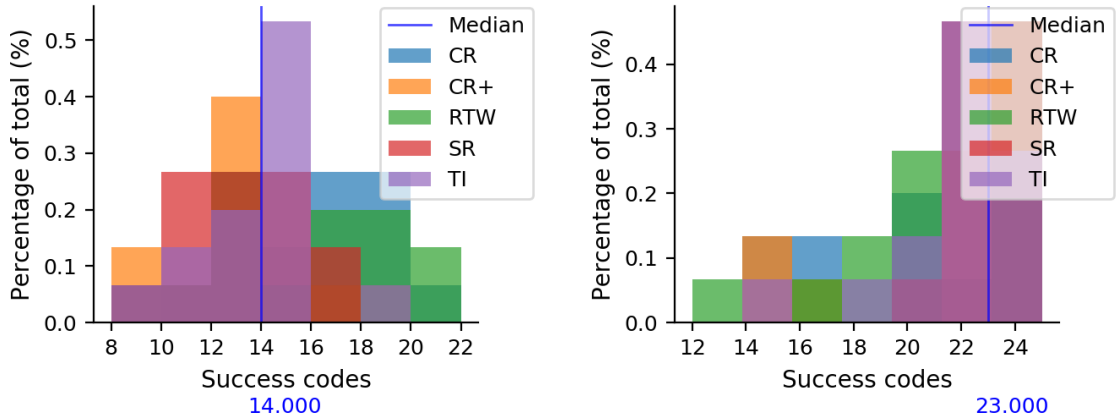
a part of the already measured voucher redeem ratio. The reason we also include the separate measurement of this factor is to distinguish between a low ratio due to fewer race conditions or a low ratio due to lots of server-side errors. Both factors are negative, but as the relation between the errors and the race conditions is not known, this measurement could provide valuable additional insights.

The combined histograms for all test setups are shown in figure 6.7. A full display of the individual histograms and statistics can be found in the appendices in section D.4 (page 187). We can make a number of observations:



(a) Overview with local server

(b) Overview with remote server



(c) Overview with normal proxy

(d) Overview with slow proxy

Figure 6.7: *The figure shows the overview histograms of the* ***number of success codes*** *obtained by all tools for all four configurations. The median of the complete datasets is calculated and indicated in base-10. Please note that the x- and y-scale of the figures are not necessarily equal.*

- Firstly, it seems like the average of the number of success codes is inversely related to the stability of the connection (in the different setups). Almost no errors were returned in the *Slow proxy* setup and about half the requests returned an error for the *Local server* setup. As we have seen in metric 1.2, this

stability greatly influences the time between requests at the server. When the time-difference is low, the server experiences a greater peak-load, and that could be the reason for more errors. Alternatively, the race conditions themselves that occurred in the application could also be responsible for the errors.

- Secondly, for the *Remote server* setup, the variance in success codes seems to be less than for the other setups. This behaviour is in line with the results to the other metrics where the *Remote server* setup also stood out with different and often more table results.
- Finally, the differences between tools for different setups seems to be marginal. Only for the *Normal proxy* setup, the CR, TI and RTW seem to have a larger average than the other tools. Further tests will have to confirm whether this difference is statistically significant.

Next, we will discuss the results of the individual tests more in-depth, and finally, the conclusions are drawn.

1. **Test for normality** - Just like for metric 2.1, we gathered only 15 data points per group (tool). Therefore, the CLT does not apply, and we need to perform the Shapiro-Wilk test to verify whether the results are normally distributed. The results of this test are shown in table 6.6, row 2). Based on this, we can conclude that only the results to the *Local server*, and *Normal proxy* setups are normally distributed.
2. **Test for equality of means of all groups** - For the *Local server* and *Normal proxy* setups, we can use the One-Way ANOVA test to find out whether there is a statistical difference between means of the results of the tools. For the other setups, we have to use the non-parametric Kruskal-Wallis test. In table 6.6, row 3, the results of this test are shown. Only for the *Normal proxy* setup, it shows that there is a statistically significant difference between the means of the groups.

   This means that only for this setup, we will perform further tests regarding the individual group means. For the other setups, we can conclude that the number of success codes is equal.

3. **Test for homogeneity of variances** - For the *Normal proxy* setup, the One-Way ANOVA test showed significant differences between the results of the groups. That is why we performed Levene's test for the homogeneity of variances for the results of this setup. In table 6.6, row 4, the results of this test are shown. Based on this, the variances of the groups are assumed to be equal.
4. **Post hoc test for equality of means between groups** - As Levene's test resulted in equal variances between the groups for this setup, we used the Tukey HSD post hoc test to group the results based on statistically significant differences between their means. In table 6.6, row 5, the results of this test are shown.

Table 6.6: *The table shows the performance test results for the* ***number of success codes****. The formatting of the table and the meaning of the contents is the same as in table 6.3.*

	Local server	**Remote server**	**Normal proxy**	**Slow proxy**
Df	4, 70	4, 70	4, 70	4, 70
Normal	Yes: S $\geq$ 0.890, 0.068	No: S $\geq$ 0.637, 0.000	No: S $\geq$ 0.475, 0.213	Yes: S $\geq$ 0.588, 0.001
Equal means	Yes: A = 0.465, 0.761	Yes: K = N/A, 0.793	Yes: A = 4.126, 0.005	Yes: K = N/A, 0.634
Equal variances	N/A	N/A	Yes: L = 1.982, 0.107	N/A
Post hoc grouping	N/A	N/A	T - 2 groups: (**SR, CR+**, TI, RTW) 0.115 (TI, RTW, **CR**) 0.063	N/A

Conclusions and discussion Based on the results as discussed above, we can draw the following conclusions regarding the number of success codes of the tools.

- **Group means** - Based on the results, there is no statistically significant difference between the group means for any setup except for the *Normal proxy* setup. In this setup, we have shown that two groupings of tools can be made of which the difference in means is statistically significant. As higher is better in for metric 2.1, the last group consisting of TI, RTW and CR has the best results. Interestingly, the two groups overlap as TI and RTW are part of both groups. For SR and CR+, however, they only belong to the worst performing group.
- **Relation between metric 2.1 and 2.2** - As we stated in the discussion above about the overview histograms in figure 6.7, this metric is interesting as it could point to reasons why no statistically significant differences in voucher redeem

ratio (metric 2.1) were observed earlier. If the difference in the number of success codes for the *Normal proxy* setup would have been smaller, the differences in the voucher redeem ratio, might have become significant. As the difference in voucher redeem ratio between the tools during the *Normal proxy* setup was almost statistically significant at a value of 0.078 (see table 6.5, row 3), this is not a far-fetched assumption. Further research is required to validate this.

6.2.4 Conclusions

In this section, the performance of the CompuRacer toolset has been compared with the other toolsets based on four (sub)metrics. Based on the results, we ran statistic tests to discover per metric whether any statistically significant differences could be found between the tools.

- Firstly, regarding metric 1.1 (local time-differences) and 1.2 (app time-differences), this resulted in the conclusion that the CompuRacer toolset, both with last-byte-sync enabled (CR+) and disabled (CR), always forms better than or equal to the other tools.
- Secondly, regarding metric 2.1 (voucher redeem ratio), no statistically significant differences were found between the mean-performance of the tools. This is a remarkable and unexpected result as the TI, and CR+ with last-byte-sync were expected to perform better in high-latency and jitter situations. Further research would have to look into this result to verify whether the outcome is actually consistent in other situations, or is an anomaly based on other factors we did not take into account here.
- Finally, regarding metric 2.2 (success codes), only for the *Normal proxy* setup, statistically significant differences were found. In this case, we got two different groupings of tools of which CR belongs to the group with the highest mean and CR+ to the group with the lowest mean. For this metric, higher is better, so CR+ does not seem to perform that well. We guess that the last-byte-sync function of CR+, which should reduce the time difference between the HTTP requests at the server also causes more errors responses. Further research would have to look into why the TI, which also has last-byte-sync, does not have this issue as well.

Based on the answers to our four metrics, we like to answer the question we postulated at the beginning of the performance evaluation: Whether the CompuRacer (and CR+) toolset has a good and consistent request-sending performance or not.

Our answer is as follows. Our toolset performs better than or equal to the performance of the other toolsets (as desired), and the last-byte-sync addition of the CR+ toolset does not seem to have significantly improved its performance.

6.3 Evaluation - Testing methodology

In this section, the methodology and CompuRacer toolset are combined and are both applied to the security testing of the web apps. This practical test targets the effectiveness and completeness of the methodology and the toolset in practice. Afterwards, we will evaluate this concerning the ease of use of the method and toolset and concerning the vulnerabilities found.

Regarding the target web apps, WebGoat and JuiceShop and also to several web apps within the categories: blogs, wikis and e-commerce (see section 4.2) are tested. For every category, one or more applications have been tested according to the methodology and using the toolset.

Note: Unfixed exploitable web apps censored For two e-commerce platforms, the testing resulted in the discovery of an exploitable race condition with significant financial impact. This race condition makes it possible to use any gift card and discount voucher more times than allowed in these web apps. For both security vulnerabilities, the developers of the platforms were informed using a responsible disclosure report. These reports are included in appendix E.

Unfortunately, the security and development teams of both platforms were not able to fix these issues before the submission date of this research. That is why these software products and companies will not be mentioned by name in the thesis, and any information about the issues that could reasonably be traced back to the specific software platform is omitted. When the information is allowed to be disclosed, an uncensored version of the responsible disclosure reports in the appendices is added as an addendum on the official institutional page of the thesis[6].

6.3.1 Tested web apps

In this subsection, the tested web apps are listed. To speed up the setup of the web application before testing, we decided that only platforms that are available as

[6]This page can be found at: `https://fmt.ewi.utwente.nl/education/master/322/`

a Docker image are included.

Docker is a very light-weight containerisation system similar to virtual machine images: only a download and startup is required to run the web app. The critical difference is that these containers only contain specific libraries and installed software, but share a kernel and operating system. This allows for using much less storage compared to virtual machine images.

For every web app, we give a short introduction, indicate why it was chosen, which sources and versions were used, and how it was set up.

1. **Testing apps: WebGoat (WeGo)** - The OWASP Webgoat, already introduced in example 1 of section 1.1.2, is a deliberately insecure web app written using Java Spring MVC. The web app has a login functionality and contains a large number of web security-related challenges that can be solved to score points[7]. These points are indicated in a global leaderboard.

    - **Why chosen** - It was chosen based on the fact that it is open source, multi-process, easy to set up, contains interesting functionality like accounts and leader boards and is not meant to contain race conditions. This makes it a perfect candidate for quickly testing a web app for race conditions.

    - **Version and source** - 8.0.0.M21 released on the 18th of January 2019[8]. Both the built-in HSQLDB, an in-memory relational database, and PostgreSQL version 11 is used.

    - **Setup** - It was run on a Tomcat server using the default configuration that can be found on the linked Github page.

2. **Testing apps: JuiceShop (JuSh)** - The OWASP Juice shop is a web application with an equivalent goal as the Webgoat: provide a learning experience to security testers. However, this application contains vulnerabilities from start to finish and does not contain them in clearly recognisable challenges. It is configured as an e-commerce application in which all kinds of juice can be bought. Again, no specific race condition challenges were found. The application is written in NodeJS using the Express framework, an Angular front-end, and an SQLite database.

    - **Why chosen** - It was chosen based on the fact that just like the WebGoat, it is open source, multi-process, easy to set up, contains interesting functionality like accounts and the e-commerce buying process and is not

[7] Unfortunately, as expected, no race condition-related challenges were found.

[8] Downloaded from: `https://github.com/WebGoat/WebGoat/releases/tag/v8.0.0.M21`

meant to contain race conditions. Next to this, as indicated on its GitHub page, it is based on a very modern set of technologies: "Probably the most modern and sophisticated insecure web application". Therefore, it is a good candidate for testing a new web app for race conditions.

- **Version and source** - 7.5.1 released on the 24th of September 2018[9].
- **Setup** - It was run on an NGINX server using the default configuration that can be found on the linked GitHub page.

3. **Blog: Puput CMS (Pupu)** - This rather new CMS is built on the popular Wagtail CMS, which is build using the Python Django web framework. It runs on a MySQL database. The system helps the user in easily creating complex and nested blogging pages with rich media content.
    - **Why chosen** - Firstly, because the design architecture is completely different from Java and NodeJS for the former two web apps and that made it an attractive target. Next to this, the many software layers with overlapping management and admin interfaces could pose interesting race condition issues. Finally, the blogging functionality of creating, updating and deleting blogs and media and posting comments all seemed to be possible race condition targets.
    - **Version and source** - 0.5 released on the 2nd of September 2016[10].
    - **Setup** - As it was run using a Docker container, no setup was required. It runs on an Apache server.
4. **Wiki: MediaWiki (MeWi)** - This wiki CMS is based on PHP and commonly runs on a MySQL database. It is a rather mature platform that supports most open source wikis, including the famous Wikipedia.
    - **Why chosen** - It is chosen because it is a Wiki platform that is very popular and mature. There are over 100,000 active websites that actively use the platform, according to BuiltWith (2019a). Next to this, and just like the blogging web app, it supports the creation of complex web content. Contrary to Puput, it also supports accounts and the creation of content by guests. All of this could contain interesting race conditions.
    - **Version and source** - 1.31.1 released on the 20th of September 2018[11].

[9] Downloaded from: `https://github.com/bkimminich/juice-shop/releases/tag/v7.5.1`
[10] Downloaded from: `https://github.com/APSL/docker-puput/`
[11] Downloaded from: `https://hub.docker.com/_/mediawiki`

- **Setup** - As it was run using a Docker container, no setup was required other than creating and linking a MySQL database. This process is explained on the webpage of the Docker hub link.

5. **E-commerce: osCommerce (osCo)** - This is a webshop platform written in PHP that is not very mature and its look and feel are a bit outdated.
   - **Why chosen** - The platform is chosen as a more easy target (because of its immaturity) that still has over 100,000 active sites that make use of it according to BuiltWith (2019b). Next to this, it contains account and shop functionality like products, shopping carts, verification emails and transactions which could contain interesting race conditions. We tried to add voucher redemption functionality to the platform, but as this also required manually updating the shopping-card source files, it was considered unfeasible given the available time.
   - **Version and source** - 2.3.4.1 released on the 18th August 2017[12].
   - **Setup** - As the web app source did not come with its own Docker container, we created a Docker container ourselves. The content of the Dockerfile, as shown in listing 2, can be used to re-create and run this container. The default content of the shop was used.

```
1  FROM greyltc/lamp
2  MAINTAINER Rob van Emous r.j.vanemous@student.utwente.nl
3  RUN pacman -S git --noconfirm
4
5  COPY oscommerce-2.3.4.1/catalog/ /srv/html/
6
7  RUN chmod 777 /srv/html/includes/configure.php
8  RUN chmod 777 /srv/html/admin/includes/configure.php
9
10 EXPOSE 80
11 #
```

Listing 2: *The listing shows the contents of the Dockerfile for setting up the osCommerce web app in Docker.*

6. **E-commerce: Platform A & B (Pl-A & Pl-B)** - These are the e-commerce webshops which are censored. The webshops are both build in PHP, use a MariaDB or MySQL database and are very popular.

[12]Downloaded from: `https://github.com/osCommerce/oscommerce2/releases/tag/v2.3.4.1`

- **Why chosen** - They are chosen because these platforms are prevalent and mature with over 50,000[13] websites that actively use it. Next to this, and just like osCommerce, all account and shopping related behaviour like vouchers, transactions, verification emails, and all kinds of buy-limitations pose interesting targets for race condition testing.
- **Version and source** - For both platforms, the last or second-to-last versions are used.
- **Setup** - Both platforms were available as easy to setup Docker containers, and the GitHub pages provided enough information to set up the environment and testing data.

6.3.2 Test results

In this subsection, the results of testing the web apps are shown. For every web app, the testing methodology that we created in chapter 4 is used. Regarding the issues that we tested for, we used the checklist as shown in figure 4.3 as a guideline. In table 6.7, we indicate the findings per web app. The names of the tested web apps are listed in abbreviated form. For every item in this guide, we indicate the testing result using the following codes:

- **Flag:** N/A - In this case, we did not test the item for the web app either because the whole category (like 'shopping') is missing or because the specific functionality is not present. For every webshop, we explain why items with this flag could not be tested.
- **Flag:** PASS - In this case, we tested the item for the web app, and it was not found to contain any race conditions. The test was at least performed twice using 10 parallel requests per test and resulted in no anomalous behaviour. This is not a guarantee that the issue is not present, but it is certainly a reliable indicator that an attacker cannot feasibly exploit this item from a black-box perspective.
- **Flag:** FAIL - In this case, we tested the item for the web app, and it was found to contain one or more race conditions with a significant security impact. For every webshop, we explain the process of finding the vulnerability of the failed items and what kind of impact is expected. As the tests are executed using a synthetic test setup, we cannot guarantee that this issue can also be exploited

[13] Very approximate as more exact numbers would result in easy identification of the platforms.

in a production environment. Therefore, the issue should be separately verified for any real web app.

- **Flag:** IND - In this case, we tested the item for the web app and some issues related to race conditions were found. These issues were not severe enough to warrant a complete fail, but also not minor enough to ignore them. For every webshop, we explain the process of finding the issues of the indeterminate items and what kind of impact is expected.

Table 6.7: *The table lists all race condition security test items that were tested in the seven web apps.*

#	Index	Name	WeGo	JuSh	Pupu	MeWi	osCo	Pl-A	Pl-B
1	1.1	Log in	PASS	IND	N/A	PASS	N/A	N/A	N/A
2	1.2	Password reset	N/A	N/A	N/A	N/A	PASS	PASS	PASS
3	1.3	Update settings	PASS	N/A	N/A	N/A	N/A	FAIL	FAIL
4	1.4.1	Multiple creation	FAIL	IND	N/A	PASS	IND	FAIL	PASS
5	1.4.2	Creation and use	PASS	PASS	N/A	N/A	N/A	N/A	N/A
6	1.5.1	Multiple deletion	N/A	N/A	N/A	N/A	N/A	N/A	N/A
7	1.5.2	Delete and create	N/A	N/A	N/A	N/A	N/A	N/A	N/A
8	1.5.3	Delete and use	N/A	N/A	N/A	N/A	N/A	N/A	N/A
9	1.6.1	Change roles	N/A	N/A	N/A	N/A	N/A	N/A	N/A
10	2.1	Likes/votes	N/A	N/A	N/A	N/A	N/A	N/A	N/A
11	2.2	Comments	N/A	N/A	PASS	N/A	N/A	N/A	PASS
12	2.3.1	Multiple creation	N/A	N/A	IND	PASS	N/A	N/A	N/A
13	2.3.2	Updates/moves	N/A	N/A	PASS	PASS	N/A	N/A	N/A
14	2.4.1	Multiple deletion	N/A	N/A	FAIL	N/A	N/A	N/A	N/A
15	2.4.2	Delete and update/move	N/A	N/A	FAIL	N/A	N/A	N/A	N/A
16	3.1	Reviews	N/A	N/A	N/A	N/A	N/A	N/A	N/A
17	3.2.1	Limited items	N/A	PASS	N/A	N/A	PASS	PASS	FAIL
18	3.2.2	Out of order	N/A	PASS	N/A	N/A	FAIL	PASS	FAIL
19	3.3.1	Use in same order	N/A	PASS	N/A	N/A	N/A	FAIL	PASS
20	3.3.2	Use in different orders/accounts	N/A	PASS	N/A	N/A	N/A	FAIL	FAIL
21	3.4.1	Multiple item creation	N/A	N/A	N/A	N/A	PASS	PASS	N/A
22	4.1	1. Additional race conditions	FAIL	N/A	N/A	N/A	N/A	PASS	PASS

We give a more in-depth explanation of skipped items, minor issues and severe issues below. For every issue, we try to connect it to one of the two race condition definitions that we created in section 4.1 and we state the expected security impact of every item. For the first application, we decided to add a white-box examination of the issues that were found to gain more insight into why these issues are present

in the web app. That is why, in these cases, we can list the expected root cases of the issues at the code-level of the software.

1. **E-learning: WebGoat (WeGo)**

   - FAIL **4: Multiple creation** - By creating an account in parallel (as a guest), multiple accounts will be created with the same username. This username is supposed to be unique. When using the built-in HSQLDB, the application is vulnerable to this issue. However, when the database is swapped with a PostgreSQL database, the issue disappears. We estimate that the default synchronisation and isolation rules are different between these databases. This is not a far fetched conclusion as the HSQLDB is only an in-memory database meant for testing purposes. The consequence for the duplicate user account is that it cannot solve any challenges. Next to this, for all users, the general scoreboard is no longer accessible. The vulnerability is worked out in more detail, in example 1 of section 1.1.2.

   - FAIL **22: Additional race conditions** - A user can submit different solutions to security challenges in parallel, and this will only be counted as one try (or as fewer tries). In our testing setup with a locally run server, we were repeatedly able to successfully submit 100 solutions while the counter was only incremented by two or three. As these counters are used in the general scoreboard to differentiate between user scores, this race condition makes it possible for users to cheat.

     At the code-level, as shown in listing 3, we have depicted the method that handles the submission of challenge answers. The method `trackProgress` is not synchronised and can, therefore, be accessed in parallel. Here we can see that the root issue is a RUW race condition (see definitions in section 4.1). The 'Read' happens at line 2 or 4, the 'Update' at line 7 or 9 and the 'Write' at line 11. The race window is as large as the time between the read and the write actions. The update actions happen one layer deeper in the methods `assignmentSolved` and `assignmentFailed` shown in listing 4. The `incrementAttempts()` calls (lines 3 and 9) post-increment the counter variable: `numberOfAttempts++;`. When two or more answers arrive at roughly the same time, all of the handling threads could read the same `UserTracker` object, increment the counter by one and overwrite the data of the other threads. The result is an incrementation of only one (or a small amount).

- N/A - As WebGoat is an e-learning application for security tester testing, some types of functionality are missing, and these test items were therefore not applicable. More concretely, there is no password reset or admin role functionality, no deletion of accounts, no content or shopping functionality.

```
protected AttackResult trackProgress(AttackResult attackResult) {
    UserTracker userTracker =
     ↪  userTrackerRepository.findByUser(webSession.getUserName());
    if (userTracker == null) {
        userTracker = new UserTracker(webSession.getUserName());
    }
    if (attackResult.assignmentSolved()) {
        userTracker.assignmentSolved(webSession.getCurrentLesson(),
         ↪  this.getClass().getSimpleName());
    } else {
        userTracker.assignmentFailed(webSession.getCurrentLesson());
    }
    userTrackerRepository.save(userTracker);
    return attackResult;
}
```

Listing 3: *The listing shows the a method in the class `AssignmentMethod.java` that contains the root cause of a race condition in the WebGoat challenge answer submission.*

```
    public void assignmentSolved(AbstractLesson lesson, String
     ↪  assignmentName) {
        LessonTracker lessonTracker = getLessonTracker(lesson);
        lessonTracker.incrementAttempts();
        lessonTracker.assignmentSolved(assignmentName);
    }

    public void assignmentFailed(AbstractLesson lesson) {
        LessonTracker lessonTracker = getLessonTracker(lesson);
        lessonTracker.incrementAttempts();
    }
```

Listing 4: *The listing shows two methods in the class `UserTracker.java` that handle updating the user tracker state when an assignment answer is submitted.*

2. **E-commerce: JuiceShop (JuSh)**

   - IND **1: Log in** - By logging in to the same account in parallel (as a guest), the application returns an HTTP internal server error-code (500). The server returns the following errors:

     – `SequelizeTimeoutError: SQLITE_BUSY: database is locked`

     – `Error: commit has been called on this transaction(..), you can no longer use it`

     According to the first error, the parallel traffic of the test has caused a timeout in the database queries. This is to be expected and is no real issue. For the second error, a database connection seems to be used after the SQL transaction is committed. This is illegal behaviour, which is probably due to accidental connection re-use between different parallel execution threads. We guess that the root issue is a race condition between the usage and the renewal of a database connection object. We are not sure whether an attack with significant negative impact could be constructed from this issue, and that is why we did not fail this item.

   - IND **4: Multiple creation** - By creating multiple accounts in parallel with the same username (as a guest), the application returns the same HTTP internal server error-code (500) as in the issue above. The created account did function normally, just like any other account, and no other functionality seems to be affected. That is why we did not fail this item as well.

   - N/A - As JuiceShop is both an e-learning (like WebGoat) and e-commerce application meant for security tester training, some types of functionality are missing, and these test items were therefore not applicable. More concretely, there is no password reset or admin role functionality, no deletion of accounts, no content or shopping-item creation functionality. Next to this, we could not test whether a multiple use-voucher could be used more than allowed because this type of voucher was not available to us. As testing the unlimited-use voucher did not show any signs of race conditions, we do not expect this to be an issue, but we are not sure.

3. **Blog: Puput CMS (Pupu)**

   - IND **12: Multiple creation** - By creating a new blog in parallel (as a user), multiple entries will be created that are reachable via the same URL (same slug). This field is supposed to be unique, but a TOCTOU / RCP race condition circumvents this rule. Subsequently requesting the newly created blog(s) results in an HTTP 500 error-code with the follow-

ing text (when debugging is enabled): `get() returned more than one Page, it returned 2!`. As this only impacts, the new blog of the current user, the integrity or availability of the application as a whole is not affected. This issue does seem to point to deeper problems with regards to the thread-safety of content creation code, and that is why the indeterminate-flag is used.

- FAIL **14: Multiple deletion** - By deleting a blog in parallel (as a user), the blog will only be partially removed due to a RUW-type race condition. After deletion, at least the 'path' value of this blog (which is expected to be unique) remains in the database, and this results in the following issues. Firstly, no new sibling pages[14] can be created as the automatically generated 'path' value will collide with the remainder of the deleted page.

  Secondly, this also renders the existing sibling pages inaccessible. The only solution is to delete (and re-create) the parent page and all siblings. As sibling pages could be owned by other users, this is a significant impact. In the case that the parent-page is the root of the web app, the consequences are even more severe as this page cannot be re-created from the user or admin interface. Only a complete re-install of the database solves this issue, and that is a serious availability issue.

- FAIL **15: Delete and update/move** - By moving and deleting a blog in parallel (as a user), the blog will only be partially moved and deleted due to a RUW-type race condition. This issue seems to have the same impact as deleting a page in parallel. Further testing is required to verify the exact difference between the impact of item fourteen and fifteen.

- N/A - As Puput is only a simple blog CMS, there is no account creation, deletion, password reset or admin role functionality. Also, there is no shopping functionality. Finally, regarding the content, no votes or likes can be awarded.

4. **Wiki: MediaWiki (MeWi)**

   - PASS - No race condition vulnerabilities were found.

   - N/A - As Mediawiki has no shopping purpose, all shopping-related items cannot be tested. Next to this, there is no account deletion, password reset or admin role functionality. Regarding the content, no votes or likes can be awarded.

[14] Two pages are called siblings when only the part after the last slash in the URLs is different.

5. **E-commerce: osCommerce (osCo)**
    - IND **4: Multiple creation** - By creating multiple accounts in parallel with the same username (as a guest), a TOCTOU / RCP race condition circumvents the unique-email limitation, and multiple accounts are created. This does not result in any errors within the application, and only one account (with the lowest ID: first created) is accessible until its email address is changed. As this is a circumvention of a rule results in unexpected duplicate database entries but does not result in a direct security impact, we have flagged this issue as indeterminate.
    - FAIL **18: Out of order** - By adding multiple items to the shopping cart in parallel, it was possible to add more items than were actually in-stock. When the in-stock limit is not rigidly employed by the shop owner, the text 'Items might not be available' is shown with every order. In this case, it is not a security issue. If the limit is employed, this race condition is still present en could result in a negative (security) impact of availability and packaging issues regarding these products.
    - N/A - There is no account settings, deletion, admin role functionality or content. Regarding the content, no votes or likes can be awarded, and no vouchers were supported. Vouchers could be added by manually installing plugins for both the vouchers and an update to the shopping cart, but this option was abandoned when it turned out to be a very time-consuming process.
6. **E-commerce: Platform A (censored) (Pl-A)**
    - FAIL **3: Update settings** - By changing the email address of two accounts from two different email addresses to the same email address (in parallel), the uniqueness requirement for the email address is violated. This is a TOCTOU / RCP race condition. A possible attack could be executed by using social engineering to make someone change their email to a certain value while the attacker does the same on his account that was created at a later moment (higher user ID). If successful, the issue results in the attacker getting access to the account of the victim without knowing their password, and this would be a confidentiality breach. By using two local accounts, we were able to perform this attack. However, this attack would be very complex and hard to perform in real life as the race window seems to be only about 100-200 ms (in our local setup), but it is still possible.

- FAIL **4: Multiple creation** - By creating multiple accounts in parallel with the same username (as a guest), a TOCTOU / RCP race condition circumvents the unique-email limitation, and multiple accounts are created. Hypothetically, the same attack could be devised for this race condition as the one discussed above for 'Update settings' which would again result in a confidentiality breach. However, in this case, no sensitive data is yet put into the victim account. The attacker would have to devise a way to maintain access to the account while the victim adds his sensitive information.
- FAIL **19: Use in same order** - By sending the same order in parallel, limited-use vouchers can be used more times than allowed. This is a TOCTOU / RCP race condition with significant financial (integrity) impact. Next to this, as the remainder of partially used gift cards gets emailed to the user in the form of a new voucher, we received a new voucher for every parallel redemption of the original voucher. Unfortunately, only the first one of these vouchers could be used to repeat the attack.
- FAIL **20: Use in different orders/accounts** - By sending an order with the same limited-use voucher from multiple accounts in parallel, this voucher can be used more times than allowed. The issue has the same consequence as for item nineteen.
- N/A - In Platform A, there is no account settings, deletion, admin role functionality, or content. Regarding the content, no votes or likes can be awarded.

7. **E-commerce: Platform B (censored) (PI-B)**

   - FAIL **3: Update settings** - By changing the email address of two accounts from two different email addresses to the same email address (in parallel), the uniqueness requirement for the email address is violated. This is a TOCTOU / RCP race condition. An attack similar to the attack for the item 3 in Platform A could be devised here as well to exploit this race condition for informational gain. Interestingly, the platform seems to have been defended against the 'Multiple creation' race condition (item 4), but this defence is not extended to this part of the functionality. No confidentiality issues were found.
   - FAIL **17: Limited items** - By using parallel product add-requests (as a user or guest), we can add more units of one product to the shopping cart than is allowed for one user. This is a TOCTOU / RCP race condition. The impact is that the attacker could use this advantage to buy

additional products for resale. This also negatively affects the availability of the products for other legitimate users.

- FAIL **18: Out of order** - By using parallel product add-requests (as a user or guest), we can add more units of one product to the shopping cart that are available. This is a TOCTOU / RCP race condition. The impact is that any automated packaging system could end up in an unexpected state where an order cannot be packaged successfully. Other than that, the attacker is not likely to be able to use this issue into any financial gain.
- FAIL **20: Use in different orders/accounts** - By sending an order with the same limited-use voucher from multiple accounts in parallel, this voucher can be used more times than allowed. Contrary to Platform A, this platform does not support partial-use vouchers, and therefore, this exploitation only once results in financial gain. Also, we were not able to redeem vouchers more times than allowed by sending parallel orders within one user account (item 19).
- N/A - In Platform B, there is no account settings, deletion, admin role functionality, or content. Regarding the content, no votes or likes can be awarded.

6.3.3 Conclusions

The evaluation does not include any formal metrics but should estimate the effectiveness and completeness of the method and toolset from a practical perspective. That is why, below, we will discuss our own experience and results after using the method and toolset:

1. **Regarding the methodology** - This method provided the necessary guidance to know for each type of web app we focused on where to look for race conditions, how to select requests, send them in parallel and verify the results. Next to this, it provided an easy way to group and present the issues found regarding race conditions in web apps as can be seen above.

   Unfortunately, for several aspects of the tested webshops, a single test had to be added or removed from the total test. This shows that the method still requires the tester to look into the specific functions of a particular web app for each test, and that adds time-consuming complexity to a test. However, as the current situation is a lack of any methodology for testing web apps regarding race conditions, this can still be considered a significant improvement.

2. **Regarding the toolset** - This was of great help both during the discovery phase of a test, but also during the exploitation and verification of a suspected issue. The ease at which requests could be added from Burp or the browser while testing was great. Also, the ability to send multiple different requests in parallel was used more than once. Next to this, the raw speed of the toolset made it possible only to use a few parallel requests to exploit issues consistently. Finally, the way the tool shows the results was of great help. The summary tables were always considered first and often helped in the initial prediction whether a race was triggered or not. After this, using the advanced grouping and comparing behaviour alongside the parsing and viewing of results, we made sure that the validation of the prediction could reliably be confirmed or denied.

3. **Regarding the vulnerabilities found** - The ability to find issues is an essential aspect of the toolset and methodology. In all tested web apps except for MediaWiki, some interesting exploitable race condition vulnerabilities have been found. For all of these web apps except for JuiceShop, exploiting these vulnerabilities could result in a significant security impact. Only for E-commerce platforms A and B, these race conditions pose a high-risk security vulnerability with a significant financial impact.

   3.1. **WebGoat** - Exploiting items 4 and 22 could result in cheating and disturbing the stability of the platform as a whole.

   3.2. **JuiceShop** - Exploiting items 1 and 4 could result in server-side errors, but any significant impact is not expected.

   3.3. **Puput CMS** - Exploiting item 12 only has an impact on the current (malicious) user and therefore, is not deemed dangerous. Items 14 and 15, on the other hand, could result in the destruction of the running platform as a whole, but these issues can only be exploited from an admin- or user-perspective.

   3.4. **MediaWiki** - No issues.

   3.5. **osCommerce** - Exploiting item 4 could result in instability of the application, but any significant impact is not expected. However, item 18 could impact the availability of the application and result in packaging issues regarding ordered products.

   3.6. **Platform A** - Exploiting items 3 and 4 could lead to a serious confidentiality breach for existing users, but as exploitation requires social engineering and very delicate timing, this is not a very likely scenario. Items 19

and 20, however, are not difficult to exploit and could result in a significant financial impact.

3.7. **Platform B** - Exploiting item 3 could lead to a serious confidentiality breach for existing users, but like for item 3 in Platform A exploitation seems unfeasible for all but the most high-value accounts (like celebrities or site-admins). Item 17 could result in resale competition and impact product availability and thereby, customer satisfaction. Item 18 could impact the availability of the application and result in packaging issues regarding ordered products. Item 20 is not difficult to exploit and could result in a significant financial impact.

Taking into account the limited experience of the author with security testing in practice, this is a very positive result. Based on these results and the fact that systematic security testing for these issues is currently not performed in practice, we estimate that a professional security tester that would employ our method and toolset would be able to find numerous security issues in existing web apps.

In the next chapter, we reiterate the discoveries and conclusions made in this chapter next to the findings in chapters 4 and 5, and use them to answer to main research questions of the thesis.

Chapter 7

Conclusions

In the exploratory phase of the research in chapters 1, 2 and 3, we have shown that race conditions are a prevalent issue among a variety of web app platforms. However, systematic security testing for these race conditions is also shown not to occur from a black-box perspective, and existing toolsets are not mature enough to efficiently support the tester in this area. Similar to the approach to testing other security issues, systematic testing for the existence of race conditions requires a methodology of where to search for the issue, and how to exploit it.

That is why the central aspect of this research revolved around the question: *how can we perform systematic black-box testing for exploitable race conditions in web apps?* We have set out to answer this broad main question by finding the answers to three sub-questions: 1) at what places in web app functionality do race conditions occur, 2) how can we trigger them from a black-box perspective and 3) how can we create a tool to support this testing process?

We found the answers to the first two questions by developing an appropriate testing methodology in chapter 4. As a part of this method, we provided a detailed mapping between web app functionality and potentially exploitable race conditions in section 4.2.2. This answers our first question. Also, a strategy is created that explains how HTTP requests should be selected and sent in ways to maximise the chances of triggering race conditions in sections 4.2.3 up to 4.2.5. This answers our second question. We can answer the third question after having created a toolset for testing race conditions from a black-box perspective (see chapter 5). The toolset is designed based on the requirements to it that flowed from the creation of the methodology and by the limitations found after executing a preliminary test on the existing tools in section 3.2.4.

Any new academic proposals for a methodology or application should also try to compare it with existing work to validate its added value. That is why we performed an extensive evaluation of both the effectiveness of the toolset compared to other tools and the practical usage of the methodology and toolset in chapter 6.

Based on section 6.1, we can conclude that both the CompuRacer and the Turbo Intruder provide, by far, the best functionality to support a race condition test. The results in section 6.2 show us that the performance of the CompuRacer (and CR+) is better than or equal to the performance of the other toolsets. It can remotely cause race conditions even with high latency and jitter on the connection between the tool and the target web app. This is a very positive result. The results in section 6.3 indicate that both the methodology and the toolset perform well in practice and provide a systematic overview of race condition vulnerabilities per web app. The practical test further proved its effectiveness because it led to finding two race condition related security issues in two well-used e-commerce platforms.

Based on these achievements, we can conclude that we have successfully taken the initial step towards systematic black-box testing for exploitable race conditions in web apps. We now know how to discover and exploit race conditions in a systematic way from a black-box perspective, and this answers our main research question. At the same time, at every step in this process we became more aware we only scratched the surface of what can be known about black-box testing for race conditions in web apps and that much more research is needed in this area. That is why we have also dedicated a significant portion of our time to formulate concrete starting points for future work in chapter 8.

We envision that race conditions, due to their probabilistic nature, might never be fully tamed, but at the same time, we have discovered in this research that, given enough dedication, much improvement is possible in this area. Systematic testing for race conditions and fixing the found issues could have a direct impact on software quality, and that has been our primary goal. We hope that other researchers will respond to our call and further develop the methodology and toolset until they have achieved a level of maturity that can successfully penetrate the broader academic and business world.

Chapter 8

Future work

In this chapter, we will iterate over all parts of the research that could be improved upon in later research. The contents are broken down into the improvements to each main chapter of the thesis document: the systematic testing method in section 8.1, the toolset in section 8.2, and the evaluation 8.1. For the toolset, the possible improvements are split into scientific research challenges and engineering improvements.

8.1 Methodology improvements

The methodology, as described in chapter 4, currently contains all aspects of interest to perform a sufficient systematic test for race conditions in web apps. However, as this method, to our best knowledge, is the first method devised in academic literature, we regard it as still being in its infancy. Based on the limitations we encountered during the creation and evaluation of this method, we have defined the following concrete improvements.

Fundamental web app types The first improvement we like to state is the establishment of a more complete basis of web app types for the checklist of the methodology. For our methodology, we only sampled from the functionality usually found in wikis, blogs and e-commerce webshops to create the list of possible vulnerabilities. However, in this section, we already stated that it would be advantageous to examine the functionality of even more web app types like platforms for chats, forums, e-learning, games, and private clouds.
These additional types are expected to yield more options for race conditions to manifest themselves when the proper parallel requests are sent to the application.

For instance, all rights-related race conditions that exist or existed at the file system level of an operating system could also be present in a cloud platform [ref to related work]. Also, it might be possible to cheat in online games and e-learning platforms by exploiting race conditions in the scoring system. Last, forums or chat systems may also be vulnerable to race conditions regarding the sending of messages and attachments. These findings could then be added to the checklist.

Integration with existing methods The second improvement to the methodology can be found in better integration with existing methods for security testing. Currently, the process is sufficient in itself, but in practice, it will probably be executed as a part of a complete vulnerability assessment or penetration test (see section 2.3). As it requires information about the functionality of the web app, it cannot happen before the data gathering or exploration phase of a vulnerability assessment. Also, as the found race condition vulnerabilities (like duplicate account names) often need to be combined with other vulnerabilities for successful exploitation, it should also happen at a specific moment during a penetration test.

More research is needed to find out at what moment during the complete test, different aspects of the test for race conditions can best be added. Also, it is uncertain what information can be exchanged between this test and other tests for efficiency reasons. We suggest that a solution should be sought in combining the tests in security areas like authentication, identity management, input validation, error handling and cryptography with the specific potential race condition vulnerabilities in these areas for the most efficient combination. This assumption is still to be validated by future work.

8.2 Toolset improvements

The toolset, as described in chapter 5 currently fulfils all requirements to be of efficient support to the methodology. However, based on our experience with the toolset and the elaborate evaluation of the toolset that we executed in chapter 6, several improvements to the toolset were discovered and are discussed below.

8.2.1 Scientific research challenges

This topic is about scientific challenges regarding the toolset that are not yet solved in this research or in other literature, and require further investigation.

Geographical race conditions In the current methodology, we (implicitly) only concern ourselves with the different ways one or more clients can send parallel requests to one specific server to exploit race condition vulnerabilities. As stated in section 2.2, web apps often use multiple synchronised web servers in parallel to host their service with improved performance or reliability.

An interesting research question in this direction is: 'To what extent do different synchronised back-ends to a web app increase the number of race conditions and the race window of different parts of the application?' In other words, what happens when we send requests in parallel to different or even geographically spaced servers? If the mutual synchronisation between servers is lacking, it is expected that new kinds of exploitable race conditions can be found. This test setup can be seen in figure 8.1.

Other application-level protocols In this research, only race conditions regarding the application-level protocols HTTP and HTTPS over transport-level protocol TCP have been considered. However, as indicated in section 2.2, other application-level protocols like WebSockets (WS) or Secure WebSockets (WSS) are also used in web applications. Compared to HTTP, this protocol is rather new as it became mainstream in browsers between 2011 and 2013. It is used in real-time communication data streaming in videos, games or chats applications.

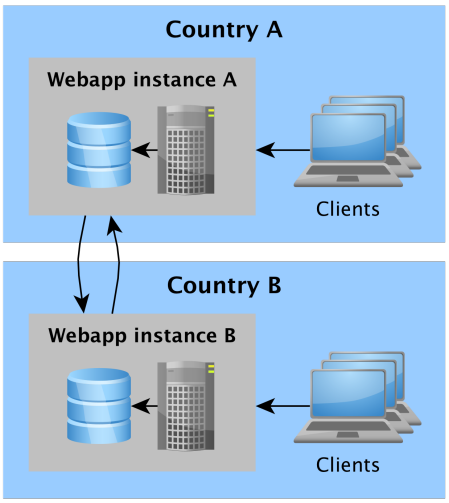


Figure 8.1: *Load balancing a web app via synchronising instances in every country. This could result in additional race conditions or an increased race window.*

Just like most other guidelines regarding security testing mentioned in chapter 1, we have not found any research that connects this technology with potential race condition vulnerabilities. More specifically, both Fette and Melnikov (2011) and Koch (2013) (already mentioned in section 1.2) look into security testing of the WebSocket protocol, but fail to mention race conditions. That is why further research is required to find out to what extent race conditions can also be found and exploited in WebSockets. It is expected that requests for the switch between HTTP and WSS using this protocol.

Other transport-level protocols Next to WebSockets for real-time communication, protocols like Web Real-Time Communication (WebRTC) over User Datagram Protocol (UDP) streams also seem to be a useful alternative as Fiedler (2017) shows. This author made a simple protocol implementation called Netcode.io. This functionality is not yet built into browsers, but it can already be tried out via a plugin for Chrome and Firefox made by Rhodes and Reinstein (2017). UDP as the underlying layer is much faster than TCP and when a web app like an online game makes use of this, exploiting possible race conditions in this part of web app functionality is expected to be easier. Further research is required to verify this claim.

Improved parallelisation As indicated in section 5.3.5 on the implementation of the toolset, four sequential steps were taken to increase the performance of the toolset. To our best knowledge, there is no potential improvement to the performance without redesigning the asynchronous HTTP libraries that the CompuRacer toolset uses. Still, as the evaluation in section 6.2 shows, the performance of all tools is severely lacking when there are significant latency and jitter in the used connection. This was the case for the 'Normal' and 'Slow' proxy tests.

Therefore, more research is required to find out what kind of techniques can be employed to ensure that even when using a bad connection, the requests arrive at the server with the lowest possible time-difference. Solutions might be found in a redesign of the entire IP/TCP stack, just like the developers of the Turbo Intruder performed so that every part is tuned for synchronised delivery of HTTP requests. Also, the usage of multiple network cards might allow for improved parallelisation.

8.2.2 Engineering improvements

This topic is about potential engineering improvements to the toolset. These solutions are not necessarily novel in an of themselves, but they would primarily be engineering challenges that make the toolset more useful in practice.

Geographic DNS resolver In line with the future work proposed in section 8.2.1, the toolset should also support sending requests to different physical servers that host the same web app. We have already implemented a simple version of this functionality, and this can be found in the sources on the CompuRacer toolset GitHub in the source file `geo_dns_resolver.py`. The script used dozen's of DNS servers from all over the world to find all (most) IP addresses that the domain of the web app resolves to. Then, it uses a public API to find out the physical location of all

servers. Based on the expected latency between these servers, the tester can, for instance, pick a very remote server pair for an attack to maximise the synchronisation time and thereby also maximise the race window. Executing the latter example from the CLI of the toolset is not yet supported and is still to be implemented in the future.

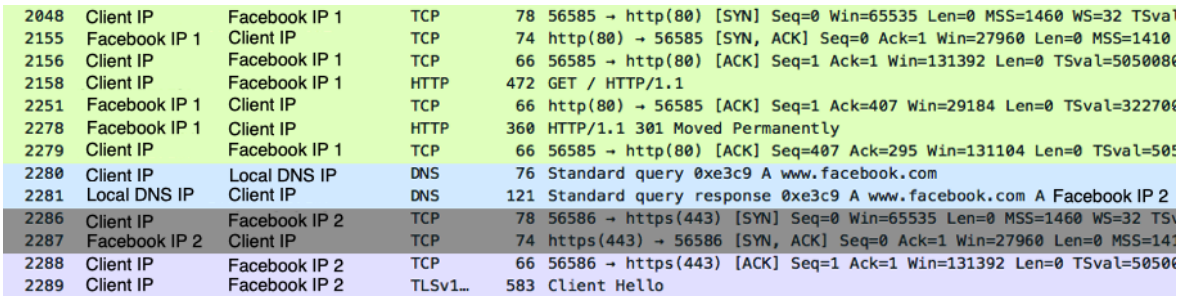


Figure 8.2: *The figure shows a Wireshark capture of the TCP and HTTP requests while directly connecting to a Facebook server in Toronto (Canada) from the west of the Netherlands. The connection gets redirected to a local Facebook server.*

Initial testing with our current implementation showed that this process is not as easy as expected because web apps often only allow specific source IP addresses to access certain physical servers (Geo-fencing). In figure 8.2 the Wireshark capture is shown of trying to connect to a Facebook server in Toronto (Canada) from the West of the Netherlands. As expected, first, a TCP handshake is performed, and then, the client issues an HTTP GET request for the Facebook web app. However, the server responds with a status code `Removed Permanently (302)` that contains the general `www.facebook.com` domain. This then causes the client to perform a local DNS lookup of this domain. The local DNS server responds with the IP address of a local Facebook server, and the client finally connects to this server instead. A potential solution to the geo-fencing could be to access these different servers via proxy servers in these countries. Further research is required to verify this claim.

Usage of CLI libraries Currently, the toolset uses a self-written Command Line Interface (CLI) which proved to be the best option given the limited time available. However, the functionality is somewhat limited, and future work should include the adoption of dedicated and advanced libraries to replace this functionality. This should make the addition of additional advanced functionality like command completion, output redirection and pipes possible. More importantly, the maintenance of this part of the toolset would then be moved to the party, which saves time for the maintainer of the CompuRacer toolset.

For instance, CDM2 by the Python-CMD2 Team (2019), or the even more advanced Cement library by Data Folk Labs (2018), could be used for creating advanced CLIs. PyInquirer made by CITGuru (2018) could be used alongside either of these libraries or the self-written CLI specifically for asking (multiple choice) questions to the user. Unfortunately, this library does not work in the modified terminal of the PyCharm Integrated Development Environment (IDE). As this is a very popular IDE for Python developers as (Reitz and Schlusser, 2016, p. 29) indicate, and this tool is still in the development stage, compatibility with this environment was considered very important. That is why we do not suggest the use of this library. Future work should investigate further whether these or other CLI or question-asking libraries might provide a valuable addition to the toolset.

Testing assistance Currently, the toolset can only be used to test for race conditions in a manual way. There is no assistance regarding the selection of requests of interest, the creation of batches, and the interpretation of results.

- The selection of requests of interest is already naively done by the browser plugins as described in section 5.3.7 and these checks could also be built into the CompuRacer Core and extended upon.
- The automatic creation of batches requires the concept of web app types (blogs, wikis, e-commerce, etc.), functionality (login, shopping, content creation, etc.), and the associated vulnerabilities as described in section 4.2.2 to be included into the toolset.
- Finally, based on the evaluation methods described in section 4.2.5, we think that the interpretation of results can be supported by allowing the tester to provide either the expected number of certain HTTP response codes or to select a test-request with an expected result that can be used to validate whether a race condition occurred automatically.

Further work is required to build the various assistance options into the toolset.

Generating reports Next to automating the detection and exploitation process, it would be useful to be able to export the issues that are found. This would help in the integration with the current testing process and the ease of use. When a particular batch of requests has resulted in a race condition at the target website, it should be possible to generate a report of this. For this to work, target websites and scopes should be added to the tool, and the tester should be able to indicate what kind of race condition has occurred.

Integration with existing methods The toolset now runs separately from the extensions in the Burp Suite and the browser. Via the extension, it can use the history of requests from Burp or the browser, but cannot be controlled via these extensions or send any results back. This would be a valuable addition to the toolset as it would integrate even better in existing manual and automated testing methods.

The benefit to manual testing is that instead of having to use a separate tool, security testers only have to get used to a new extension in, for instance, Burp. This extension would contain the same functionality as the CompuRacer Core CLI now does. It can also make use of the GUI elements that Burp makes available to extensions. For automated testing, it could include sending some requests via the CompuRacer Core to the target web app. Then, get back the results to validate certain race conditions are not present in the web app.

To make this possible, the CompuRacer Core logic should be fully controllable via an API. Then, all extensions would be able to both control the Core and show results to the user. As it would then be fully controllable via both the API and the CLI, the developer has to think through how different concurrent commands will be managed. As the sending logic works best when it has as much CPU power and bandwidth as possible, this part cannot easily be made concurrently accessible. A solution would be to buffer the sending of batches and execute them sequentially.

The Core could also be fully moved in the Burp extension, for instance. The downside to this is that it loses its independence. When you just want to test a single HTTP request and forward it using a browser extension, it would be better not to require Burp. Also, when another in-flow method of requests would be added like the ZAP proxy (OWASP, 2018), the Postman API tester (Postman Inc., 2019), or another toolset, the ability to independently run the CompuRacer Core is essential.

More work needs to be done to validate the need for this integration and subsequently upgrade the toolset accordingly.

8.3 Evaluation improvements

The evaluation, as described in chapter 6 currently fulfils the purpose of evaluating the effectiveness of the methodology and toolset, but several improvements to this evaluation can still be discerned. In this final section of the future work, we propose extensions to our affords to evaluate the methodology and the toolset. For section 6.1 on functionality & usability evaluation, no improvements were discovered. For the other two sections, the improvements are discussed separately.

8.3.1 Performance evaluation

In this section, improvements to the performance evaluation in section 6.2 are discussed.

Number of target web apps In the performance evaluation, four setups were tested for all tools on a single web app. As this web app was specifically built by us to contain race conditions, this is not an entirely realistic target. In future work, it is recommended to execute the performance evaluation on multiple types of web apps where certain vulnerabilities within different functionality are targeted. It is expected that including these web apps will result in different results regarding the second test of the first metric (time-differences at the application), and both tests of the second metric (exploitation success and number of errors). This would result both in a better-grounded conclusion regarding the performance of the tools and would also yield valuable insights regarding the behaviour of different web apps when a race condition test is performed.

More remote targets Next to this, the results showed that having both the client and the server on the same system can influence the results compared to using a remote server. It would have been better to not only include one distant target but to host multiple target web apps at different geographical locations on the earth. This would both remove the influence factor in the results and also result in more insights regarding the relationship between geographic distance and the extent to which requests can be made to arrive within the race window.

A testing platform hosting web apps types Based on our experience, we think it would be highly valuable future work to create an independent platform that focuses on providing packaged versions of all kinds of web apps (grouped by type or functionality) which are configured and ready to use for research and testing purposes. The most obvious method would be to package these setups using collections of Docker containers or virtual machines (VMs). It would be best when a community of researchers and testers would also be able to add packaged web apps themselves. In this case, researchers could also point to these packages within their research for greatly improved reproducibility and extensibility of their results.

This suggestion is not closely related to our research topic itself but is a practical issue we continuously ran into while testing out toolset and methodology. It turned out that it is a rather involved process to set up a web app from scratch to test it. As

we have shown, this process involves finding a recent Docker image of the platform, installing and configuring the platform, linking a compatible database and filling the database with appropriate testing data. As specifics of this process are often completely different between individual platforms, for us, this process took several days for some web apps. Every single researcher or tester has to go through these steps themselves, introducing a massive amount of duplicate work in the field of computer science. The creation of a platform, as described above, would reduce this work to a single community effort. We have not looked at the drawbacks regarding security and the process of keeping packages up-to-date, but we still deem this as an exciting research direction.

8.3.2 Practical evaluation

In this section, improvements to the practical evaluation of the methodology and toolset in section 6.3 are discussed.

Increase number of web apps tested For the practical evaluation of the methodology and toolset, we have currently tested only seven distinct web applications. In future work, we would recommend the researcher to both add more web apps of the current types (wikis, blogs and e-commerce), and also to add more types of web apps in line with the extension proposed in the first paragraph of section 8.1. This addition would both create a more elaborate evaluation of the methodology and toolset, but it would also have the potential to result in more found vulnerabilities. In two of the seven applications we tested, a severe vulnerability was found, and therefore, we estimate that a more extensive test would likely result in several additional interesting findings.

Remote exploitation and the race window Next to the extension in the number of tested web apps as proposed above, future work should also include a test of remotely hosted applications. As already indicated in section 8.3.1 above, distant targets might be harder to exploit and therefore provide a test environment that is more comparable to an actual web app security test.

Bibliography

R. Abbott, J. Chin, J. Donnelley, W. Konigsford, S. Tokubo, and D. Webb, “Security Analysis and Enhancements of Computer Operating Systems,” National Bureau of standards Washington, D.C., Technical report, 1976.

S. Acharya and V. Pandya, “Bridge between Black Box and White Box–Gray Box Testing Technique,” *International Journal of Electronics and Computer Science Engineering*, vol. 2, no. 1, pp. 175–185, 2012.

C. Adamsen, A. Møller, R. Karim, M. Sridharan, F. Tip, and K. Sen, “Repairing Event Race Errors by Controlling Nondeterminism,” *IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

C. Adamsen, A. Møller, and F. Tip, “Practical initialization race detection for JavaScript web applications,” *Proceedings of the ACM on Programming Languages*, vol. 1, pp. 1–22, 2017.

C. Adamsen, A. Møller, S. Alimadadi, and F. Tip, “Practical AJAX Race Detection for JavaScript Web Applications,” in *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018.

P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2016.

G. B. and V. V. (2018) How to Choose a Technology Stack for Web Application Development. [Online]. Available: https://rubygarage.org/blog/technology-stack-for-web-development

W. W. R. Ball, “Other Questions on Probability,” *Mathematical Recreations and Essays*, vol. 45, 1960.

B. Beizer, *Software Testing Techniques*. Dreamtech Press, 2003.

M. Billes, A. Møller, and M. Pradel, “Systematic black-box analysis of collaborative web applications,” *ACM SIGPLAN Notices*, vol. 52, pp. 171–184, 2017.

N. Breslow, “A generalized Kruskal-Wallis test for comparing K samples subject to unequal patterns of censorship,” *Biometrika*, vol. 57, no. 3, pp. 579–594, 1970.

BuiltWith. (2019) Mediawiki usage statistics. [Online]. Available: https://trends.builtwith.com/shop/MediaWiki

——. (2019) oscommerce usage statistics. [Online]. Available: https://trends.builtwith.com/shop/osCommerce

J. Cable. (2017) Exploiting and Protecting Against Race Conditions. [Online]. Available: https://lightningsecurity.io/blog/race-conditions/

M. Carbou. (2019) Reverse Ajax, Part 1: Introduction to Comet. [Online]. Available: https://www.ibm.com/developerworks/web/library/wa-reverseajax1/index.html

Certified Secure. (2018) Certified Secure Checklists. [Online]. Available: https://www.certifiedsecure.com/checklists/

S. Chen. (2011) Session Puzzling and Session Race Conditions. [Online]. Available: http://sectooladdict.blogspot.com/2011/09/session-puzzling-and-session-race.html

CITGuru. (2018) PyInquirer: A Python module for common interactive command line user interfaces. [Online]. Available: https://github.com/CITGuru/PyInquirer

C. Collberg. (2014) Examining "Reproducibility in Computer Science". [Online]. Available: http://cs.brown.edu/~sk/Memos/Examining-Reproducibility/

Computest. (2019) IT Security, Performance and Test automation. [Online]. Available: https://www.computest.nl/en/

Data Folk Labs. (2018) Cement: Application framework for python. [Online]. Available: https://github.com/datafolklabs/cement

DB-Engines. (2018) DB-Engines Ranking. [Online]. Available: https://db-engines.com/en/ranking

D. Dean and A. J. Hu, “Fixing Races for Fun and Profit: How to Use access (2).” in *USENIX Security Symposium*, 2004, pp. 195–206.

Defuse Security. (2011) Practical Race Condition Vulnerabilities in Web Applications. [Online]. Available: https://defuse.ca/race-conditions-in-web-applications.htm

N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, “Microservices: How To Make Your Application Scale,” in *International Andrei Ershov*

Memorial Conference on Perspectives of System Informatics. Springer, 2017, pp. 95–104.

O. J. Dunn, "Multiple Comparisons Among Means," *Journal of the American statistical association*, vol. 56, no. 293, pp. 52–64, 1961.

S. Faulkner, A. Eicholz, T. Leithead, A. Danilo, and S. Moon, "HTML 5.2," *W3C. Retrieved January*, vol. 17, p. 2018, 2017.

I. Fette and A. Melnikov, "The WebSocket Protocol - 6455," Internet Engineering Task Force, RFC, 2011.

G. Fiedler. (2017) Why can't I send UDP packets from a browser? A solution for enabling UDP in the web. [Online]. Available: https://gafferongames.com/post/why_cant_i_send_udp_packets_from_a_browser/

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1 - 2616," Internet Engineering Task Force, RFC, 1999.

C. Flanagan and S. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," *ACM SIGPLAN Notices*, vol. 44, p. 121, 2009.

C. Flanagan and S. N. Freund, "The ROADRUNNER Dynamic Analysis Framework for Concurrent Programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010, pp. 1–8.

J. Franjković. (2016) Race conditions on the web. [Online]. Available: https://www.josipfranjkovic.com/blog/race-conditions-on-web

P. A. Games, H. Keselman, and J. C. Rogan, "Simultaneous pairwise multiple comparison procedures for means when sample sizes are unequal." *Psychological Bulletin*, vol. 90, no. 3, p. 594, 1981.

P. Godefroid, P. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin, "Automating Software Testing Using Program Analysis," *IEEE software*, vol. 25, no. 5, pp. 30–37, 2008.

J. N. Goel and B. Mehtre, "Vulnerability Assessment & Penetration Testing as a Cyber Defence Technology," *Procedia Computer Science*, vol. 57, pp. 710–715, 2015.

Google. (2018) Protractor end to end testing for Angular. [Online]. Available: https://www.protractortest.org

J. Halley and P. Inchausti, “Lognormality in ecological time series,” *Oikos*, vol. 99, no. 3, pp. 518–530, 2002.

W. Hetzel, “The Complete Guide to Software Testing,” *QED Information Sciences*, 1984.

A. Hnatiw. (2016) Race The Web (RTW). [Online]. Available: https://github.com/insp3ctre/race-the-web

S. Hong, Y. Park, and K. Moonzoo, “Detecting concurrency errors in client-side java script web applications,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2014, pp. 61–70.

W. E. Howden, “Reliability of the Path Analysis Testing Strategy,” *IEEE Transactions on Software Engineering*, no. 3, pp. 208–215, 1976.

J. Ide, R. Bodik, and D. Kimelman, “Concurrency Concerns in Rich Internet Applications,” 2009.

M. Jadon. (2018) Race Condition Bug In Web App: A Use Case. [Online]. Available: https://medium.com/@ciph3r7r0ll/race-condition-bug-in-web-app-a-use-case-21fd4df71f0e

M. Jans. (2016) netCloneFuzzer. [Online]. Available: https://github.com/snapo/netCloneFuzzer

C. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, “Stateless model checking of event-driven applications,” *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 57–73, 2015.

P. Jones. (2017) 3x faster than flask – hacker noon. [Online]. Available: https://hackernoon.com/3x-faster-than-flask-8e89bfbe8e4f

S. S. Kar and A. Ramalingam, “Is 30 the magic number? issues in sample size estimation,” *National Journal of Community Medicine*, vol. 4, no. 1, pp. 175–179, 2013.

J. Kettle. (2019) Turbo Intruder: Embracing the billion-request attack | Blog. [Online]. Available: https://portswigger.net/blog/turbo-intruder-embracing-the-billion-request-attack

H.-Y. Kim, “Analysis of variance (ANOVA) comparing means of more than two groups,” *Restorative dentistry & endodontics*, vol. 39, no. 1, pp. 74–77, 2014.

R. Koch, “On WebSockets in penetration testing,” Master’s thesis, Vienna University of Technology, 2013.

H. Kuosmanen *et al.*, “Security Testing of WebSockets,” Master’s thesis, JAMK University of Applied Sciences, 2016.

N. Kurapati, V. S. C. Manyam, and K. Petersen, “Agile Software Development Practice Adoption Survey,” in *International Conference on Agile Software Development*. Springer, 2012, pp. 16–30.

H. W. Lilliefors, “On the Kolmogorov-Smirnov Test for Normality with Mean and Variance Unknown,” *Journal of the American statistical Association*, vol. 62, no. 318, pp. 399–402, 1967.

Y. Lozinsky. (2017) 6 Web Development Stacks To Try In 2017. [Online]. Available: https://webinerds.com/6-web-development-stacks-try-2017/

B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, “2011 CWE/SANS top 25 most dangerous software errors,” *Common Weakness Enumeration*, vol. 7515, 2011.

G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006, vol. 1.

D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

M. Meucci, E. Keary, and D. Cuthbert, “OWASP Testing Guide v3,” OWASP Foundation, Testing guide, 2008.

G. Milener, J. Roth, C. Malhotra, and C. Guyer. (2018) Understanding isolation levels - sql server | microsoft docs. [Online]. Available: https://docs.microsoft.com/en-us/sql/connect/jdbc/understanding-isolation-levels?view=sql-server-2017

Mitmproxy. (2018) A free and open source interactive HTTPS proxy. [Online]. Available: https://mitmproxy.org/

MITRE Corporation. (2019) CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (’Race Condition’). [Online]. Available: https://cwe.mitre.org/data/definitions/362.html

——. (2011) On the Cusp: Other Weaknesses to Consider. [Online]. Available: https://cwe.mitre.org/top25/archive/2011/2011_onthecusp.html

J. Moilanen, M. Jeskanen *et al.*, “Non-functional testing: security and performance testing,” 2015.

T. Moore, “The Economics of Cybersecurity: Principles and Policy options,” *International Journal of Critical Infrastructure Protection*, vol. 3, no. 3-4, pp. 103–117, 2010.

A. Muller, M. Meucci, E. Keary, and D. Cuthbert, “OWASP testing guide 4.0.” OWASP Foundation, Testing guide, 2013.

E. Mutlu, S. Tasiran, and B. Livshits, “I know it when I see it: Observable races in JavaScript applications,” in *Proceedings of the Workshop on Dynamic Languages and Applications*. ACM, 2014, pp. 1–7.

——, “Detecting JavaScript races that matter,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 381–392.

S. Northcutt. (2007) Security Laboratory: Methods of Attack Series - Race Conditions. [Online]. Available: https://www.sans.edu/cyber-research/security-laboratory/article/race-cndtns

OWASP. (2018) OWASP Zed Attack Proxy Project. [Online]. Available: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project#tab=Main

OWASP community, “OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks,” OWASP, Tech. Rep., 2017.

M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer Science & Business Media, 2011.

R. Paleari, D. Marrone, D. Bruschi, and M. Monga, “On Race Vulnerabilities in Web Applications,” *Lecture Notes in Computer Science*, pp. 126,142, 2008.

Pallets Team. (2010) Flask - A Python Microframework. [Online]. Available: http://flask.pocoo.org/

S. Pandey. (2016) Testing Race Conditions in Web Applications. [Online]. Available: https://securingtomorrow.mcafee.com/business/testing-race-conditions-web-applications/

B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, “Race Detection for Web Applications,” *ACM SIGPLAN Notices*, vol. 47, pp. 251,262, 2012.

M. Pohja, “Server Push with Instant Messaging,” in *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 2009, pp. 653–658.

Portswigger. (2018) Burp Suite Editions. [Online]. Available: https://portswigger.net/burp

——. (2018) Extensibility. [Online]. Available: https://portswigger.net/burp/extender/

Postman Inc. (2019) Postman API Development Environment. [Online]. Available: https://www.getpostman.com/

Pylons Project. (2019) Welcome to the Pylons Project. [Online]. Available: https://pylonsproject.org/

Python-CMD2 Team. (2019) CMD2 - quickly build feature-rich and user-friendly interactive command line applications in Python. [Online]. Available: https://github.com/python-cmd2/cmd2

P. Reinheimer and W. Roberts. (2019) WonderNetwork - Ping time between Amsterdam and Los Angeles. [Online]. Available: https://wondernetwork.com/pings/Amsterdam/Los+Angeles

——. (2019) WonderNetwork - Ping time between Amsterdam and Paris. [Online]. Available: https://wondernetwork.com/pings/Amsterdam/Paris

K. Reitz and T. Schlusser, *The Hitchhiker's Guide to Python: Best Practices for Development*. " O'Reilly Media, Inc.", 2016.

J. Rhodes and M. Reinstein. (2017) Browser extensions for netcode.io. [Online]. Available: https://github.com/RedpointGames/netcode.io-browser

A. Riancho. (2019) Race-condition-exploit: Tool to help with the exploitation of web application race conditions. [Online]. Available: https://github.com/andresriancho/race-condition-exploit

G. D. Ruxton and G. Beauchamp, "Time for some a priori thinking about post hoc testing," *Behavioral ecology*, vol. 19, no. 3, pp. 690–693, 2008.

Sakurity. (2017) Sakurity Racer. [Online]. Available: https://github.com/sakurity/racer

Security Compass. (2016) Moving Beyond The OWASP Top 10, Part 1: Race Conditions. [Online]. Available: https://blog.securitycompass.com/moving-beyond-the-owasp-top-10-part-1-race-conditions-912dccbb7c14

K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

S. S. Shapiro and M. B. Wilk, "An Analysis of Variance Test for Normality (Complete Samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

M. C. Shingala and A. Rajyaguru, "Comparison of Post Hoc Tests for Unequal Variance," *International Journal of New Technologies in Science and Engineering*, vol. 2, no. 5, pp. 22–33, 2015.

Shopify. (2019) Shopify/toxiproxy: A TCP proxy to simulate network and

system conditions for chaos and resiliency testing. [Online]. Available: https://github.com/Shopify/toxiproxy/releases

R. Smith, R. Harrison, S. Wood, D. Sussman, A. Fedorov, S. Murphy *et al.*, *Professional Active Server Pages 2.0*. Wrox Press Ltd., 1998.

Stack overflow. (2018) Developer Survey Results. [Online]. Available: https://insights.stackoverflow.com/survey/2018/#technology

P. Sturgeon. (2016) PUT vs PATCH vs JSON-PATCH. [Online]. Available: https://philsturgeon.uk/api/2016/05/03/put-vs-patch-vs-json-patch/

D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, 2011.

A. S. Tanenbaum and M. Van Steen, *Distributed systems: Principles and Paradigms*. Prentice-Hall, 2007.

Techrepublic. (2019) The CIA Triad. [Online]. Available: https://www.techrepublic.com/blog/it-security/the-cia-triad/

J. W. Tukey *et al.*, "Comparing Individual Means in the Analysis of Variance," *Biometrics*, vol. 5, no. 2, pp. 99–114, 1949.

J. Vanian. (2019) Meteor wants to be the warp drive for building real-time apps. [Online]. Available: https://gigaom.com/2014/12/27/meteor-wants-to-be-the-warp-drive-for-building-real-time-apps/

G. Vial, "Different Databases for Different Strokes," *IEEE Software*, vol. 35, no. 2, pp. 80–85, 2018.

J. Wang, "Characterizing and taming non-deterministic bugs in Javascript applications," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, 2017, pp. 1006–1009.

J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in Node.js," in *roceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 2017, pp. 520–531.

W. Wang, Y. Zheng, P. Liu, L. Xu, X. Zhang, and P. Eugster, "ARROW: automated repair of races on client-side web pages," *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016.

J. Wilcox, C. Flanagan, and S. Freund, "VerifiedFT: A Verified, High-Performance Precise Dynamic Race Detector," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2018, pp. 354–367.

Wireshark. (2019) About Wireshark. [Online]. Available: https://www.wireshark.org/

T. Wouters. (2017) Globalinterpreterlock - python wiki. [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

E. Woychowsky, *AJAX: Creating web pages with asynchronous JavaScript and XML*. Prentice Hall, 2007.

M. Wright. (2012) Flask-Security 3.0.0 documentation. [Online]. Available: https://pythonhosted.org/Flask-Security/

L. Zhang and C. Wang, "RClassify: Classifying Race Conditions in Web Applications via Deterministic Replay," *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

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*. IEEE Press, 2012, pp. 584–594.

Y. Zheng, T. Bao, and X. Zhang, "Statically locating web application bugs caused by asynchronous calls," in *Proceedings of the 20th international conference on World wide web*. ACM Press, 2011, pp. 805–814.

Appendices

Appendix A

Race condition testing tools sources

In this appendix, we have listed the links to the source code for all race conditions testing tools that we have considered in this research in table A.1. These tools are first listed in chapter 3. Obtaining the links to these sources often proved to be rather difficult due to obscure references in related works or dead links. That is why we have included these links in the document for efficient reproducibility of the results.

Table A.1: *The table shows the links to the sources of all considered race condition testing tools.*

Year	Name	Sources

2013	EventRacer	`https://github.com/eth-sri/EventRacer`
		`https://github.com/eth-sri/webkit`
		(dependancy: altered version of WebKit)

2015	R4	`https://github.com/eth-sri/R4`
2016	netCloneFuzzer	`https://github.com/snapo/netCloneFuzzer/tree/master/NetCloneFuzzer`
	Race the web	`https://github.com/insp3ctre/race-the-web`
2017	EventRace-Commander	`https://github.com/cs-au-dk/EventRaceCommander`
	InitRacer	`https://github.com/cs-au-dk/initracer`
	RClassify	On request, we received the sources directly from the first author. It is not publicly available.
	SakurityRacer	`https://github.com/sakurity/racer`
2018	AJAXRacer	`https://github.com/cs-au-dk/ajaxracer`
	TurboIntruder	`https://github.com/PortSwigger/turbo-intruder`

Appendix B

CompuRacer toolset – README

The README shows how to install, setup and run the CompuRacer toolset.

Recommended software versions

The toolset has been tested with Python 3.7, Firefox v. 65, Chrome v. 72, Burp Suite Professional v1.7.37, Vagrant 2.1.5 and Git 2.21.0. It is run on a MacBook Pro (late 2013) running macOS High Sierra. Every single tool is expected to be compatible with Linux and Windows as well, but this is not tested. The plugin is also likely to work in Burp Suite CE.

Installation

- Clone the repository:
  `$ git clone https://github.com/rvemous/CompuRacer`
- Install CompuRacer Core dependencies
  - Go to the `CompuRacer_Core/` folder.
  - Run: `$ pip install -r requirements.txt`
- Install CompuRacer Firefox extension
  Firefox does not support adding extensions permanently if they are not signed by Mozilla. You can add it temporarily (until the next restart), using the following method:
  - In Firefox, go to: `Settings > Add-ons`.

  - Click the gear icon and select: `Debug Add-ons`.
  - Go the `CompuRacer_Extensions/Browser/Firefox/` folder and select: `manifest.json`.

- Install CompuRacer Chrome extension
  Note that due to recent changes in Chrome (after version 71), this extension will no longer send most of the headers to the CompuRacer. Therefore, in any authenticated session, it no longer works. You can add the extension using the following method:
  - In Chrome, go to: `Settings > More Tools > Extensions`.
  - Click: `Load unpacked`.
  - Select the `CompuRacer_Extensions/Browser/Chrome/` folder.
- Install CompuRacer Burp Suite extension
  - In the Burp Suite, go to: `Extender > Add`.
  - Select `Python` as the extension type.
  - Go to the `CompuRacer_Extensions/Burp/` folder and select: `compu_racer_extension_burp.py`.
  - Click `next` and after loading the extension, close the window.
- Install test web app for voucher redemption
  - In a terminal, go to the `TestWebAppVouchers/app/` folder.
  - Run the following command: `vagrant up`.

Configuration

The Firefox, Chrome, Burp Suite extensions and test web app do not need any configuration and are ready to use. The Computest Core will create the necessary folders and settings-files on the first startup. Make sure it has full read/write access rights in this folder.

Running

The Firefox, Chrome, Burp Suite extensions and test web app are already started after the install. The CompuRacer Core can be started by running the following command within the `CompuRacer_Core` folder:

```
$ python3 main.py
```

Troubleshooting

All extensions can be reloaded (or reinstalled) if they stop working for any reason. All platforms support some form of (live) debugging of extensions.

Appendix C

CompuRacer toolset – Manual

The manual contains a guide on how to use the toolset for exploiting race conditions. It covers using the extensions to add requests to the Core, the composing of batches of requests in different modes, sending the batches and interpreting the results. Throughout this process, some powerful built-in view and compare tools are also used to support the process. The manual assumes both the toolset and the test web app have been installed successfully according to the README.[1]

Command formatting In the manual, every command is shown on a new line and can be used in the Command Line Interface (CLI) of the CompuRacer Core. As already mentioned before, the terminal of a MacBook Pro running macOS High Sierra is used. It is configured to look similar to the JetBrains PyCharm terminal using the 'Homebrew' theme with the colour of `Text` and `Bold Text` set to white. If a command argument of a string-like type contains a space, for instance when it is a name, the argument must be enclosed in double quotes (""). When an argument is a boolean, the true and false values can be abbreviated to 't' and 'f'. Commands used in this manual are formatted as follows:

```
racer> command (abbreviation) <required argument> [optional
    ↪ argument]
```

In this manual, we will not explain every part of the functionality exhaustively. Instead, the manual will go through most functionality just like a tester would when he is using the tool. For this to be exactly reproducible, we take the included web

[1] As already indicated in the README enclosed with the toolset, the Chrome extension cannot send some headers anymore, and this breaks most security testing activities. That is why we will only use Firefox in this manual.

app for voucher redeeming as a concrete example for all detection and exploitation related functionality. Some commands have different behaviours defined when optional arguments are provided or omitted. The help command can be used to view the details of commands and their behaviour or to search for a command:

```
racer> help [search term]
```

C.1 How to add HTTP requests of interest to the tool?

In this section, we will show how to add HTTP requests to the tool from the extensions, or by manually adding a batch to the tool by creating a JSON file. Last, we will explain how to use the Core action modes that help make the testing process more efficient. To view stored requests, the tester can use the first command below. If the tester wants to remove one or more requests (all ids between the first and second id) from the complete list, use the second command below:

```
racer> reqs
racer> rm reqs [first request id] [last request id]
```

C.1.1 Send it from the browser using the Firefox extension

One option for adding requests to the Core is by using the Firefox extension. When the Firefox extension is loaded, it will show a grey circle as its icon as long as it is not connected to the CompuRacer Core. You can click the icon to try to reconnect. When it shows a white circle, the connection is made successfully. If we click it again, it will show a red circle for three seconds. During this time, any request of interest that is sent to the domain of the current tab is forwarded to the CompuRacer Core. When the tester hovers over the button, the extension state is also listed in a tool-tip.

Example We want to add the POST request of redeeming `COUPON1` in the very insecure way to the CompuRacer Core. In figure C.1, you can see the web app just after activating the extension (see red circle).

After this, we click the `Redeem single` button. This sends the request to the CompuRacer Core. When it is received successfully, we will get output similar to the cyan

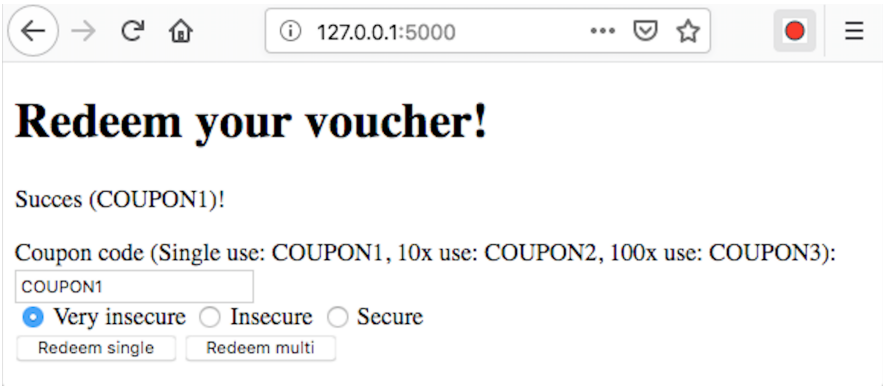


Figure C.1: *The figure shows the activated Firefox extension in the test web app just before redeeming a voucher.*

text in figure C.2 that shows the summary of the request. Then, in the same figure, the command `req 0` is executed, which shows the details of the request with id '0'.

```
racer>
INFO: Added new request:
       ID  Timestamp                  Method    URL                                                  Body Length
  --  ----  -------------------------  --------  ---------------------------------------------------  -------------
   0     0  2019-03-21 16:49:54.471755  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0

Total number: 1

racer> req 0
INFO: Request '0':
  {'id': '0'}
  {'timestamp': '2019-03-21 16:49:54.471755'}
  {'method': 'POST'}
  {'url': 'http://127.0.0.1:5000/redeem/very_insecure/COUPON1'}
  {'headers': {'Accept': 'application/json, text/javascript, */*; q=0.01',
               'Accept-Encoding': 'gzip, deflate',
               'Accept-Language': 'en-US,en;q=0.5',
               'Connection': 'keep-alive',
               'Host': '127.0.0.1:5000',
               'Referer': 'http://127.0.0.1:5000/',
               'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; '
                             'rv:65.0) Gecko/20100101 Firefox/65.0',
               'X-Requested-With': 'XMLHttpRequest'}}
```

Figure C.2: *The figure shows the output of the CompuRacer core when a new request is added.*

C.1.2 Send it from the Burp Suite using the Burp extension

The second option for adding requests is by using the Burp extension. We first need to configure the Burp proxy to capture all requests as we often do not want the original request to arrive at the target server. When the request of interest is sent, it will first appear in the Proxy tab, and then we can forward it to the CompuRacer Core using a button in the context menu. This button will be greyed-out when it is busy sending requests or when the Core not available. This is indicated clearly.

Example We want to add the same POST request as earlier to the CompuRacer Core, but now using the Burp extension. We set the proxy in capturing mode and in the web app, we click the button to redeem the voucher. After this, it pops up at the proxy tab. By selecting it, we can then click the button in the context menu to forward it to the CompuRacer Core as figure C.3 shows. The tester can also select multiple requests from any part of the Burp Suite interface and send these via the extension to the Core.

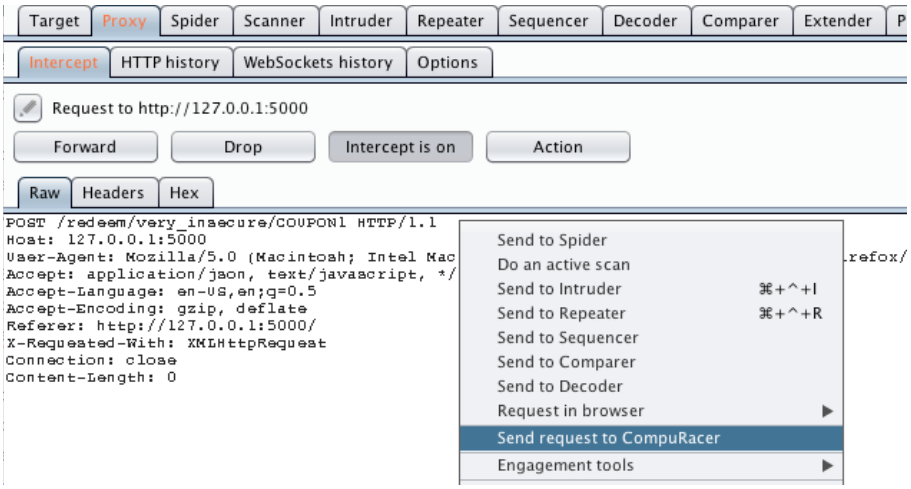


Figure C.3: *The figure shows the Burp Suite after capturing a request. From the context menu,we can decide to forward it.*

After this, we then receive the request in the CompuRacer Core. We expected to receive a duplicate request, and this should be rejected, but it was not the case, and we got a message equivalent to the one in figure C.2. When two requests appear to be the same, but the tester does not want to check the details line-by-line, the built-in compare function can be used. It compares every header field and the body of two requests and displays the changes. As you can see in figure C.4, we executed the command `comp reqs 0 1` and got a precise indication of the differences.

```
racer> comp reqs 0 1
INFO: Comparison of requests
      with id '0' and '1':

 fail: {
    normal: {
        Connection: {
    - keep-alive
    + close
        }
    }
    custom: {
    }
    missing: {
        Content-Length: {
            [None, '0']
        }
    }
 }
```

Figure C.4: *The figure shows the output of the CompuRacer core when comparing the requests from Firefox (id=0) and Burp (id=1). Only the 'Connection' and 'Content-Length' headers differ between them.*

The command works as follows. When a header is present in both requests, but the values differ based on a string-comparison, it is added to the 'normal' differences. When instead, a custom compare function is used on header values, and this fails, it is added to the 'custom' differences. Finally, when the header is missing altogether from one of the requests, it is added to the 'missing' differences. Appar-

ently, the Burp Suite adds a `Content-Length` header with a value of zero to all POST requests even when they do not have a body. Also, it changes the `Connection` header value from 'keep-alive' to 'close'.

Finally, to show what would happen when a duplicate request would arrive, we send the request again from Burp to the Core. The resulting message is shown in figure C.5.

```
racer>
WARN: New request is not added, it already exists:
        ID  Timestamp                  Method    URL                                                  Body Length
  --  ----  -------------------------  --------  ---------------------------------------------------  -------------
   0     2  2019-03-21 18:03:58.046942  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0
```

Figure C.5: *The figure shows the output of the CompuRacer core when a duplicate request is added.*

C.1.3 Add it manually using the correct JSON format

The third option is to add a request manually. This is not officially supported, but still possible. As it alters the `state.json` JSON file, the CompuRacer Core should not be running when adding a request in this way. In this file, the current settings of the Core are stored alongside the complete list of requests. When we would like to add a request, the request headers and content should be added to the dictionary-value of the `requests` key with a request key that is unique.

Example For this final example, we want to add almost the same HTTP request as in the examples earlier. The only changes are in the following HTTP headers: `Connection` will have the value `close` instead of `keep-alive`, and `Accept-Language` is changed from English to Dutch. It will get id '1' as '0' is already taken by the request added earlier. All of this can be done by adding the following entry to this JSON file, as shown in listing 5.

C.2 How to compose a batch of HTTP requests?

In this section, we will show how to add stored HTTP requests to a batch. A batch is a collection of requests that are sent with several individually defined settings. The request can be sent after a specific delay (in ms) from the start of sending the batch, and it can be sent multiple times in parallel and in sequence. This provides for almost unlimited options to prepare combinations of requests to trigger race conditions.

```
"1": {
    "body": "",
    "headers": {
        "Accept": "application/json, text/javascript, */*; q=0.01",
        "Accept-Encoding": "gzip, deflate",
        "Accept-Language": "nl-NL,nl;q=0.5",
        "Connection": "close",
        "Content-Length": "0",
        "Host": "127.0.0.1:5000",
        "Referer": "http://127.0.0.1:5000/",
        "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13;
        ↪  rv:65.0) Gecko/20100101 Firefox/65.0",
        "X-Requested-With": "XMLHttpRequest"
    },
    "method": "POST",
    "timestamp": "2019-03-21 18:01:43.019910",
    "url": "http://127.0.0.1:5000/redeem/very_insecure/COUPON1",
    "id": "1"
},
```

Listing 5: *Adding a request manually to the CompuRacer Core state file: 'state.json'*

Batches can be created from the CLI, and be filled with requests manually or automatically based on newly added requests. Batches that have been exported can also be imported back into the tool. Finally, batches can be added manually using JSON files similar to how requests were added. The tester can list all stored batches using the first command below. The second command can be used to remove one or more batches (all ids between the first and second id) from the complete list of batches. The third command is used to remove a request from the current batch by id and delay:

```
racer> batches (bss)
racer> rm batches (bss) [first batch id] [last batch id]
racer> rm [request id] [delay]
```

C.2.1 Creating it manually and adding requests

The first option is to add a batch and its requests via the CLI. We create a batch with the name `voucher_redeem_single_very_insecure` using the following command:

```
racer> add batch (bss) <batch name>
```

This does not only create a batch, but it also makes it our 'current batch'. This is the one specific batch that is selected to be edited, viewed and sent easily. Access to other batches requires more effort (longer command, and requires you to provide the batch id) and these batches cannot be edited. The current batch is marked in the listing of all batches. The current batch can be changed using the first command below. The new batch does not hold any requests yet. We can add the two requests from the previous section with ids '0' and '1', no delay, 5x parallel and no sequential duplication using the second command. Finally, the contents of the current batch can be shown using the third command:

```
racer> set curr <batch id>
racer>
    add <request id> [delay] [num parallel] [num sequential]
racer> cont
```

Figure C.6 shows the whole process of creating a batch, adding requests to the batch and listing the contents.

```
racer> add batch voucher_redeem_single_very_insecure
INFO: Created a new batch:
name = 'voucher_redeem_single_very_insecure'
allow_redirects = 'False'
sync_last_byte = 'False'
    Empty.

INFO: Set current batch to batch with name 'voucher_redeem_single_very_insecure'.
racer> add 1 0 5 1
INFO: The request was added to the current batch:
('1', 0) -> [5, 1]
racer> add 0 0 5 1
INFO: The request was added to the current batch:
('0', 0) -> [5, 1]
racer> cont
INFO: Batch contents:
name = 'voucher_redeem_single_very_insecure'
allow_redirects = 'False'

(request_id, wait_time) -> (parallel, sequential)
('1', 0) -> [5, 1]
('0', 0) -> [5, 1]

INFO: The matching request(s):
      ID  Timestamp                  Method    URL                                                  Body Length
--  ----  -------------------------  --------  ---------------------------------------------------  -------------
 0     0  2019-03-21 16:49:54.471755  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0
 1     1  2019-03-21 18:01:43.019910  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0

racer> batches
INFO: Table of batches info:
    Name                                Items        Requests  Has results
--  ----------------------------------  ----------  ----------  -------------
 0  voucher_redeem_single_very_insecure  ['1', '0']          10  {}
```

Figure C.6: *The figure shows creating a new batch via the CLI, adding two requests and showing the contents.*

C.2.2 Creating a batch using the automated modes

The CompuRacer Core can be set to three different action modes that control whether to create batches automatically when a request is added to the Core via the REST server. The tester can set the mode using the first command below and change the mode settings using the second command:

```
racer> mode [on/curr/off]
racer> set mode [num parallel] [num sequential]
```

Next, we will explain in detail how every mode influences the behaviour of the Core after adding requests.

- **On** – In this mode, all requests that are received within a 3-second interval together will be added to a special batch with the name 'Imm'. The mode uses its own sequential and parallel duplication settings to add the requests. Then, the batch is sent automatically. If the batch already exists, its contents and results are overwritten. Note that this batch can only be viewed to avoid interference and race conditions, but not be altered or sent by the user himself. If more control is required or the user wants to preserve the contents, it must be copied to a new user-controlled batch. It is useful to test forwarded requests without requiring any user interaction quickly.
- **Curr** – In this mode, all requests that are received are added to the current batch. It uses the same sequential, and parallel duplication settings as the 'on' mode does. In contrast to the 'on' mode, will not send the batch automatically. As the normal flow of activities always includes creating a batch and adding stored requests, this mode removes the second step and makes the process more user-friendly.
- **Off** – In this mode, no additional action will be taken when a request is received.

Note: although duplicate requests will not be added to the total request list, they will trigger the mode actions with the current request as an argument.

C.2.3 Add it manually using the correct JSON format

In the `state/batches/` folder, all batches of the Core are stored in their own JSON file. These batches are only loaded at startup, so after adding a batch using this method, the CompuRacer Core must be restarted for it to take effect.

Example Let us say we want to add a batch called 'get_voucher_page' that only contains the GET request for the main voucher page. This GET request has id '1'. Finally, it should be sent after a delay of 100ms for five times in parallel and two times sequentially. This can be done by adding a JSON file with contents, as shown in listing 6. The other keys in the batch JSON file are used for the batch-specific settings:

- **allow_redirects** - if this value is true, when the batch is sent, it will follow redirect (302) HTTP responses.
- **custom_comparing** - this would contain a dictionary of HTTP headers that should be ignored or be compared differently when making result groups. Result groups are explained later.
- **results** - this would contain the results when the batch has been sent.

```
{
    "name": "get_voucher_page",
    "items": [
        {
            "key": ["1", 100],
            "value": [5, 2]
        }
    ],
    "allow_redirects": false,
    "custom_comparing": null,
    "results": null
}
```

Listing 6: *JSON file of a CompuRacer batch: 'get_voucher_page.json'*

C.2.4 Import an exported batch

The final option to get batches into the Core is to import already exported batches. Just like all other storage, these are JSON files as well. By default, they are stored in the `CompuRacer_Core/exp_files/` folder. Contrary to the files that are used to store the internal batches, exported batches contain an extra key `requests` which contains the contents of all requests that are referenced in the batch contents or the results. The requests in the contents and the results might be different when requests are added to a batch after it has been sent. Adding the requests makes

it possible to import and use a batch just like it was meant to be, even when the matching requests have been changed or removed from the Core in the meantime. Only unique requests are imported back again. As requests contain a unique id, when a request is imported, it gets a new id that is the highest request id plus one. The tester can export one or more batches (all ids between the first and second id) using the following command:

```
racer> exp batches (bss) [first batch id] [last batch id]
```

Importing a batch requires a slightly different approach. As exported batches can be stored all over the system, it does not require any arguments, but it opens a file-picker dialog to the default storage location. The tester can select one or more valid exported-batch files and import them back. The tester can import a batch using the following command:

```
racer> imp batches (bss)
```

Example As we have not exported a batch before, we should first do this. We export the only batch that is currently in storage `voucher_redeem_single_very_insecure` with id '0'. Then, we remove the batch and one of its two requests from the Core itself and then import the exported batch again. It is shown that it only imports one of the requests again. This request gets the id '5' as the highest request id used to be '4'. The process can be seen in figure C.7. The import file-picker dialog is not shown.

C.3 How to send a batch and interpret the results?

In this section, we will show how to send the batches of requests and how to view and evaluate the results. Sending a batch is rather straightforward. There are two ways to do it. We can send a self-created batch, or we can allow the immediate mode to send the automatically created batch. The effects on the target web app should be the same. By using the following command, we can send a batch:

```
racer> exp bss 0
INFO: Exporting batch 'voucher_redeem_multi_insecure'..
INFO: Batch exported successfully to 'exp_files/voucher_redeem_single_very_insecure.json'
racer> rm bss 0
WARN: Are you sure you want to remove the batch with name 'voucher_redeem_single_very_insecure'?
This is the current batch! [y(es)/n(o)]
racer> y
INFO: Batch with name 'voucher_redeem_single_very_insecure' is removed.
racer> rm reqs 0
WARN: Are you sure you want to remove the request with id '0'? [y(es)/n(o)]
racer> y
INFO: Request with id '0' is removed
INFO: Removal of 1 request(s) successful.
racer> imp bss
INFO: Importing batch file '/CompuRacer/CompuRacer_Core/exp_files/voucher_redeem_single_very_insecure.json'..
INFO: Importing requests..
WARN: New request is not added, it already exists:
      ID  Timestamp                  Method    URL                                                 Body Length
  --  ----  -------------------------  --------  --------------------------------------------------  -------------
   0     1  2019-03-21 18:01:43.019910  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0

INFO: Added new request:
 {'id': '5'}
 {'timestamp': '2019-03-21 16:49:54.471755'}
 {'method': 'POST'}
 {'url': 'http://127.0.0.1:5000/redeem/very_insecure/COUPON1'}
 {'headers': {'Accept': 'application/json, text/javascript, */*; q=0.01',›
              'Accept-Encoding': 'gzip, deflate',
              'Accept-Language': 'en-US,en;q=0.5',
              'Connection': 'keep-alive',
              'Host': '127.0.0.1:5000',
              'Referer': 'http://127.0.0.1:5000/',
              'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; '
                            'rv:65.0) Gecko/20100101 Firefox/65.0',
              'X-Requested-With': 'XMLHttpRequest'}}
Total number: 5
INFO: Batch 'voucher_redeem_single_very_insecure' with 2 requests imported successfully.
INFO: Importing of 1 batches(s) successful.
```

Figure C.7: *The figure shows exporting the 'voucher_redeem_single_very_insecure' batch, removing the batch and one of its requests and importing it again.*

```
racer> go [batch id]
```

After sending the batch, we get several responses back. The CompuRacer Core does not just print all these results as it will often be very much and not very informative. As a tester, we want to be able to quickly spot and investigate anomalies in the responses that indicate a race condition was triggered in the target web app. In order to make this possible, the results of different request are separated. Next to this, for every request, the results that are shown can be divided into three categories: overview tables, a summary of the grouped responses, and the groups themselves. We can view the results of the current batch using the following command:

```
racer> res [show overview tables] [show grouped responses]
```

C.3.1 Overview tables

Every overview table checks just one characteristic of the responses and counts every different value. Currently, the following four characteristics are checked: the sta-

tus codes, body lengths (bytes), numbers of header and the header lengths (bytes). If the table has just one row, all responses are the same regarding the metric. Otherwise, it might be interesting to investigate it further.

For instance, if the status code is 403 (forbidden) for 18 packets and 302 (redirect) for two packets, then we know that something was allowed for two times. If that something is a function that should only be used once, we might have found a race condition.

C.3.2 Grouped responses

All responses are grouped based on the headers and the body. Some time and tag headers that are likely to change between responses, but are not informative, are ignored. First, it is shown how much groups we have. If this is more than one group (everything is the same) and less than the total number of parallel requests (everything is different), some responses are equal and might be interesting. Next to this, the fields that always match between responses, the fields that never match and the ignored fields are listed.

The always- and never-matched fields are especially interesting when we want to be more/less strict about some differences to create more/fewer groups. The grouping behaviour can be changed for the current batch by ignoring more or fewer fields in the grouping process. This can be altered at runtime. For instance, when we get too many groups, we might want to add a never-matched field to the ignore list. This can be done using the first command below. Viewing the ignored fields for the current batch can be done using the second command. Finally, resetting the ignored fields to the default is one using the third command:

```
racer> add ignore (ign) [field name (case sensitive)]
racer> get ignore (ign)
racer> res ignore (ign)
```

For every change to the ignored fields, the groups will be re-created automatically. When something seems off regarding the grouping after a crash, or randomly, the tester can run the following command to re-create the groups for all batches. For many batches, this could take some time:

```
racer> regroup batches (regr bss)
```

Example of sending a batch

As an example, we will send the batch created earlier called `voucher_redeem_single_very_insecure` to the server of the test app. The batch is supposed to trigger the following TOCTOU race condition and redeem more vouchers than available.

The race would work as follows. When a user tries to redeem a voucher using the 'very insecure' method, it uses two transactions instead of one and also sleeps for 3 seconds between the two transactions. One transaction to check whether the voucher can still be used and one transaction to reduce the available amount by one. In this case, we use a 'single' voucher that can be used only once. When the two transactions are executed in parallel with two other transactions, they might both read a voucher-availability of 1 and reduce this to 0, while this should not be possible. The test app will send the left-over amount back to the client, so a race condition should be easy to spot. If we get more than one success response with an amount of 1, a race condition has occurred.

It is probable that all five parallel requests of the batch will succeed to redeem the single-use voucher as we have a huge race window of 3 seconds. Therefore, for this example, we first change the parallel duplication of the request in the batch to 10 requests using the command below:

```
racer> update (upd) [request id] [delay] [num parallel]
   ↪ [num sequential]
```

Now, we send the batch. The required commands and the results of sending the batch are shown in figure C.8. If we would want to view these results again, we use the command as shown before, where both boolean arguments are set to 'true':

```
racer> res t t
```

In figure C.8, it shows that the tool has sent the ten requests and 20 seconds later (indicated by the send and end time), the results are back. These 20 seconds are not coincidental. This is the timeout that is used by default. Two responses were not back in time to meet this requirement and have been ignored in the results.

The results show three groups. One with six success responses (200), one not-found response (404) and two internal server errors (500). From this, we can conclude that we were successful in triggering the voucher-redeem race: the single-use

voucher was redeemed six times.

```
racer> upd 0 0 10
INFO: The request was updated in the current batch:
Old: ('0', 0) -> [5, 1]
New: ('0', 0) -> [10, 1]

racer> go
INFO: Sending the batch with name 'voucher_redeem_single_very_insecure'..
Start sending time: 2019-04-04 20:48:23
Receiving: 100%|████████████████████████████████████████████████████████| 10/10

Error in sending request 5 :
  [('0', 0), TimeoutError()].
Error in sending request 8 :
  [('0', 0), TimeoutError()]
INFO: The batch is sent successfully.
Results:
Batch results:
  Send time: 2019-04-04 20:48:23:
  End time:  2019-04-04 20:48:43:

  Request id '0':

     Group 0 - 6 item(s):
        {'send_time_min': '2019-04-04 20:48:23.000082'}
        {'send_time_max': '2019-04-04 20:48:23.007559'}
        {'response_time_min': '2019-04-04 20:48:26.279034'}
        {'response_time_max': '2019-04-04 20:48:26.320966'}
        {'status_code': 200}
        {'headers_length': '224 bytes'}
        {'headers': {'Cache-Control': 'no-store, no-cache',
                     'Connection': 'close',
                     'Content-Length': '48',
                     'Content-Type': 'application/json',
                     'Date': 'Thu, 04 Apr 2019 18:48:26 GMT',
                     'Server': 'nginx/1.14.0 (Ubuntu)'}}
        {'body_length': '48 bytes'}
        {'body': {'count': 1, 'time': '2019-04-04 18:48:23.252420'}}
     Group 1 - 1 item(s):
        {'send_time': '2019-04-04 20:48:23.006701'}
        {'response_time': '2019-04-04 20:48:26.261353'}
        {'status_code': 404}
        {'headers_length': '181 bytes'}
        {'headers': {'Connection': 'close',
                     'Content-Length': '48',
                     'Content-Type': 'application/json',
                     'Date': 'Thu, 04 Apr 2019 18:48:26 GMT',
                     'Server': 'nginx/1.14.0 (Ubuntu)'}}
        {'body_length': '48 bytes'}
        {'body': {'count': 0, 'time': '2019-04-04 18:48:23.267498'}}
    Group 2 - 1 item(s):
       {'send_time': '2019-04-04 20:48:23.001110'}
       {'response_time': '2019-04-04 20:48:23.349994'}
       {'status_code': 500}
       {'headers_length': '181 bytes'}
       {'headers': {'Connection': 'close',
                    'Content-Length': '38',
                    'Content-Type': 'application/json',
                    'Date': 'Thu, 04 Apr 2019 18:48:23 GMT',
                    'Server': 'nginx/1.14.0 (Ubuntu)'}}
       {'body_length': '38 bytes'}
       {'body': {'time': '2019-04-04 18:48:23.264756'}}

Request id (continued) '0':

    Status code      Amount
    -------------  --------
    200                   6
    404                   1
    500                   1
    Total                 8

    Body length (bytes)      Amount
    ---------------------  --------
    38                            1
    48                            7
    Total                         8

    Headers      Amount
    ---------  --------
    5                 2
    6                 6
    Total             8

    Headers bytes      Amount
    ---------------  --------
    181                     2
    224                     6
    Total                   8

    Number of groups: 3
    Ignored:      ['body', 'Date']
    Always match: ['Connection', 'Content-Type', 'Server']
    Never match:  ['Cache-Control', 'status_code']
```

Figure C.8: *The figure shows sending of the 'voucher_redeem_single_very_insecure' batch, and getting the results.*

If there were many equivalent result-groups, it would have been advantageous to automatically compare the contents just like we compared two requests figure C.4. Comparing two result groups can be done using the following command:

```
racer> comp [result id 1] [result id 2]
```

Figure C.9 shows the results of comparing the first group '0' of success responses with the second group '1' of a not-found response. It highlights the response code difference, the difference in body content, send and receive times (omitted) and finally indicates that that the `Cache-Control` header is missing from the second result group.

```
racer> comp 0 1
INFO: Comparison of result groups 0 and 1 in request '0'
      of batch 'voucher_redeem_single_very_insecure':

 fail: {
    normal: {
       status_code: {
    - 200
    + 404
       }
       body: {
      {
    -     "count": 1,
    ?             ^
    +     "count": 0,
    ?             ^
    -     "time": "2019-04-04 18:48:23.252420"
    ?                                 ^^ ^^
    +     "time": "2019-04-04 18:48:23.267498"
    ?                                 ^^ ^^
      }
       }
       headers_length: {
    - 224 bytes
    + 181 bytes
       }
    ** timing differences omitted **
    }
    custom: {
    }
    missing: {
       Cache-Control: {
          ['no-store, no-cache', None]
       }
    }
 }
```

Figure C.9: *The figure shows the comparison of the first two result groups after sending the 'voucher_redeem_single_very_insecure' batch.*

Appendix D

Toolset performance result histograms

In this appendix, the histograms of the results in section 6.2 are listed. These histograms are rather large, and therefore, we have decided only include only the coloured summary histograms in the referenced section and moved the composite figures to this appendix.

D.1 Metric 1 - Test 1 - Local time-difference

For all figures in this section, the y-axis shows the percentage of requests and the x-axis the time-difference using a log10 scale. A lower time-difference is likely to result in more race conditions, so lower is better. The calculation of the mean (and standard deviation between brackets), median, the first and third percentile of the data is shown as well (in regular base 10).

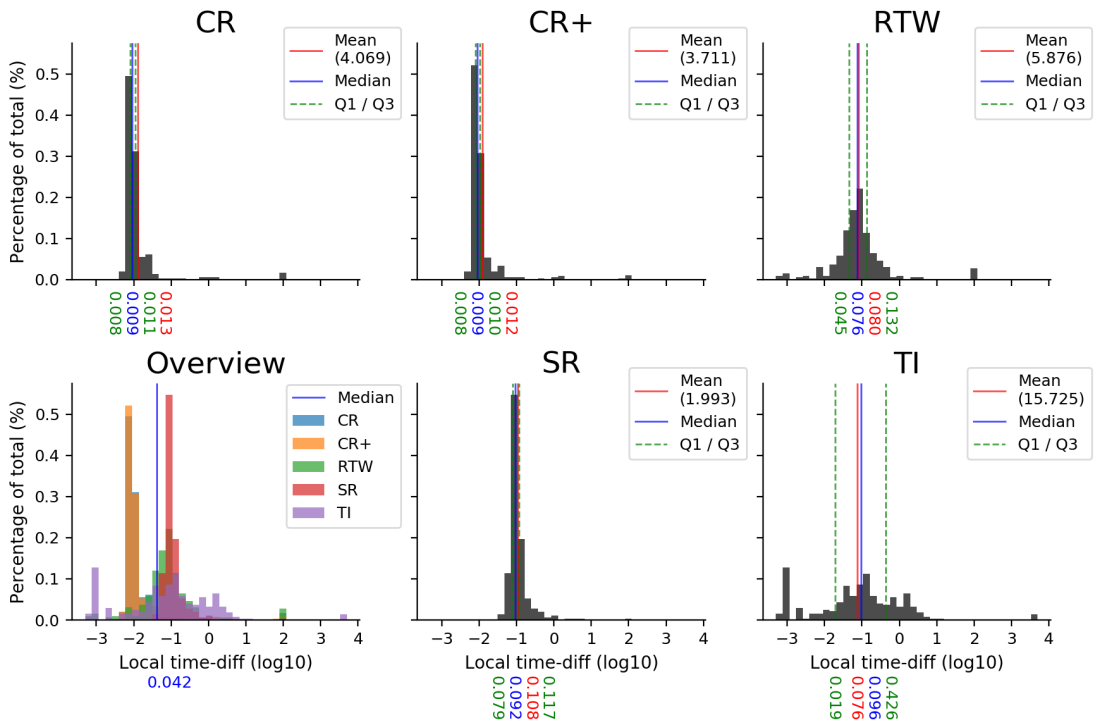


Figure D.1: *The figure shows the histograms of the* ***local time-differences*** *between requests of all tools when a* ***local server*** *is used.*

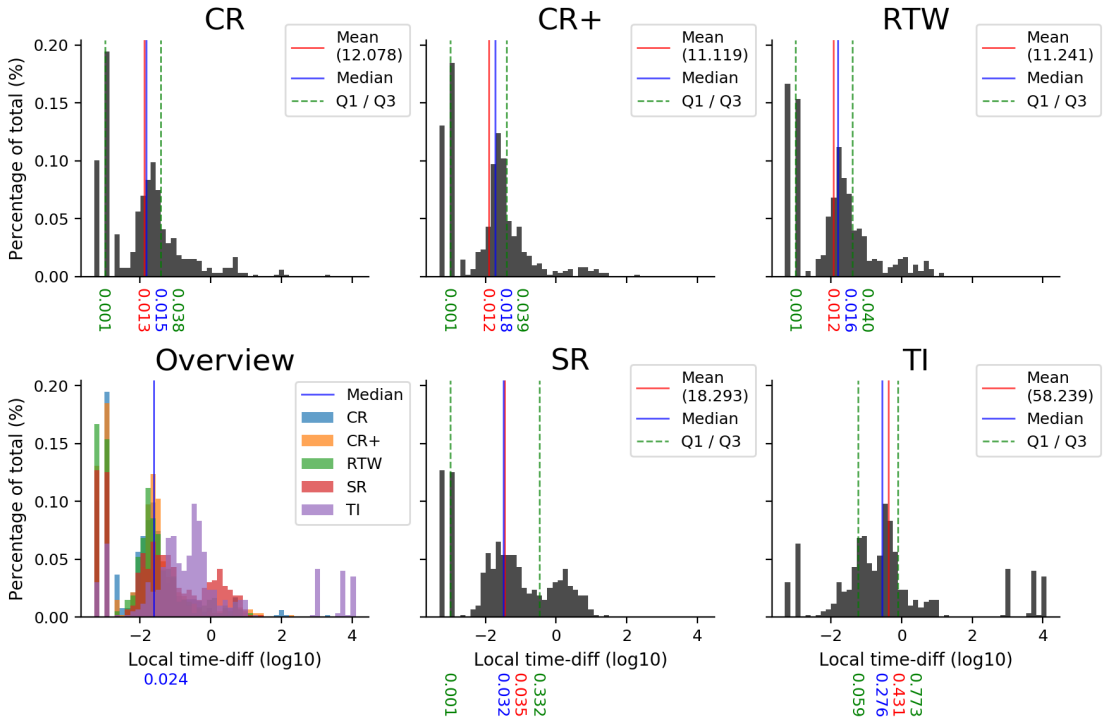


Figure D.2: *The figure shows the histograms of the* ***local time-differences*** *between requests of all tools when the* ***remote server*** *is used.*

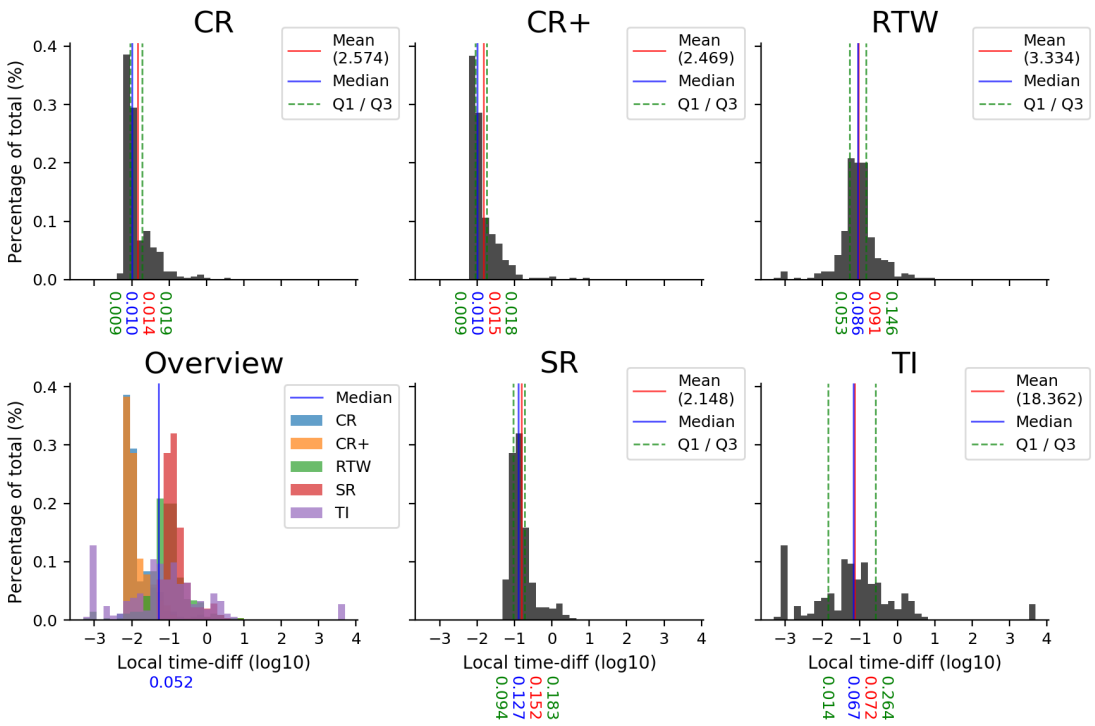


Figure D.3: *The figure shows the histograms of the* ***local time-differences*** *between requests of all tools when a* ***normal proxy*** *is used.*

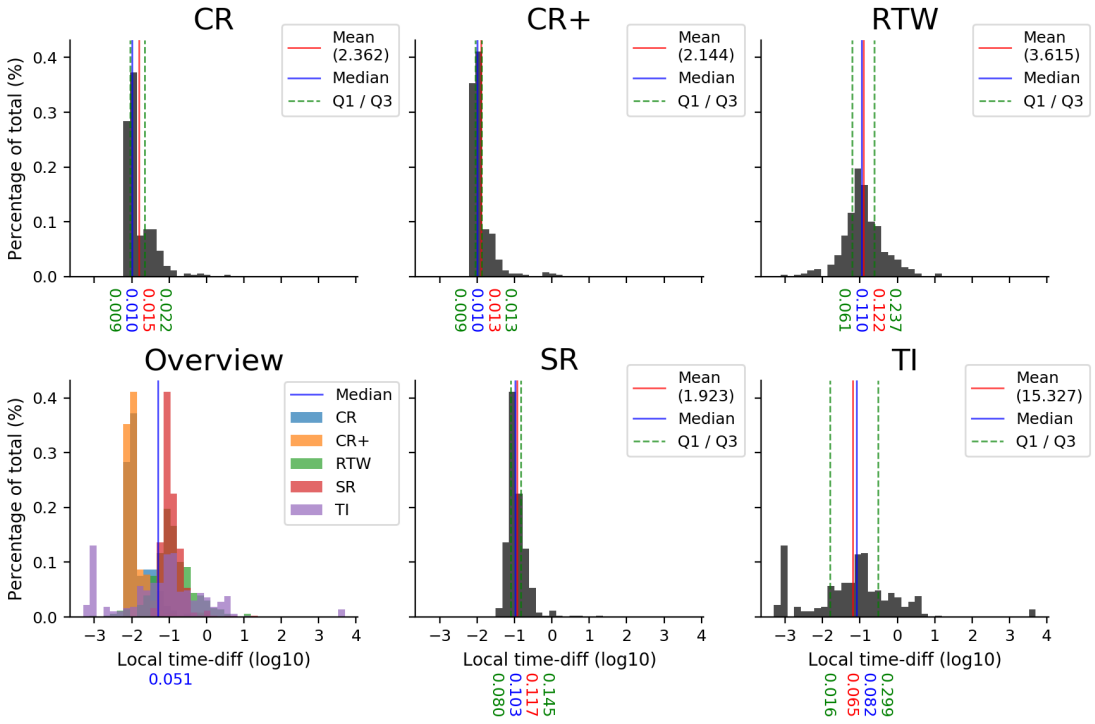


Figure D.4: *The figure shows the histograms of the* ***local time-differences*** *between requests of all tools when the* ***slow proxy*** *is used.*

D.2 Metric 1 - Test 2 - Application time-difference

For all figures in this section, the y-axis shows the percentage of requests and the x-axis the time-difference using a log10 scale. A lower time-difference is likely to result in more race conditions, so lower is better. The calculation of the mean (and standard deviation between brackets), median, the first and third percentile of the data is shown as well (in regular base 10).

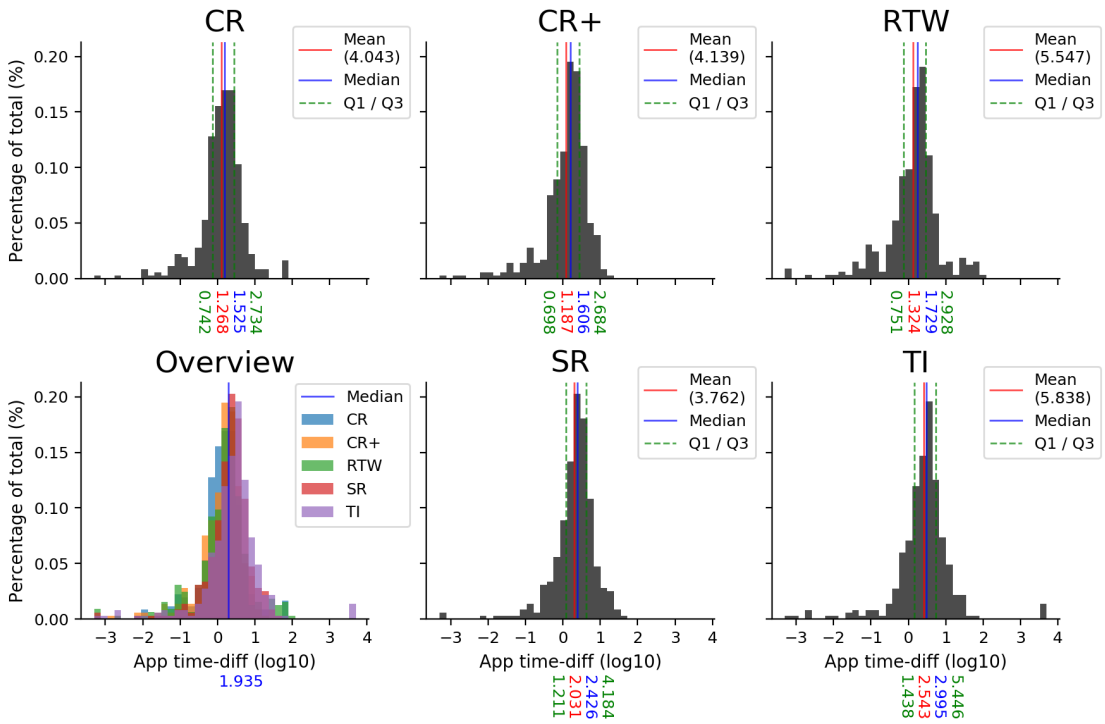


Figure D.5: *The figure shows the histograms of the* ***application time-differences*** *between requests of all tools when a* ***local server*** *is used.*

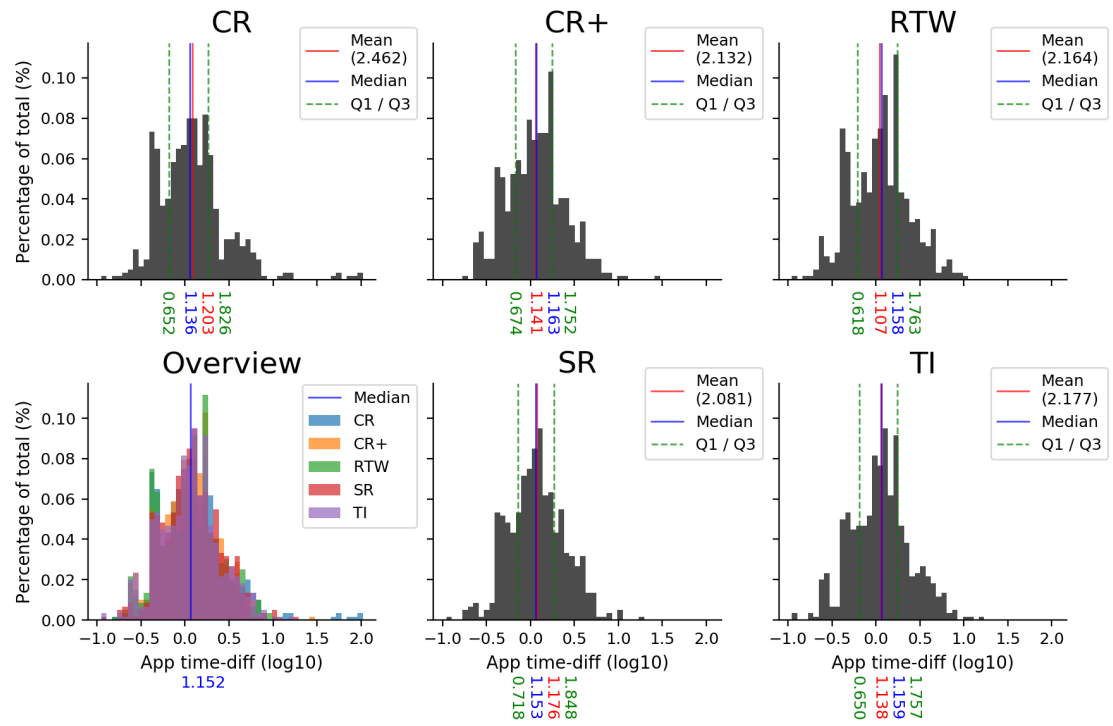


Figure D.6: *The figure shows the histograms of the* ***application time-differences*** *between requests of all tools when the* ***remote server*** *is used.*

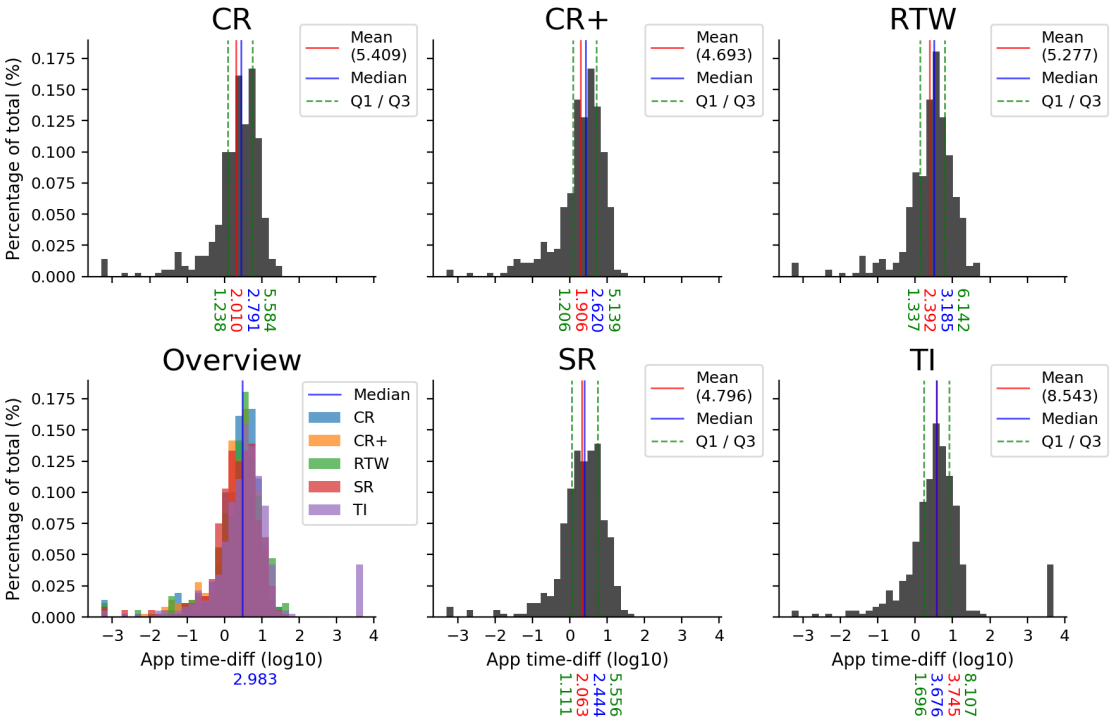


Figure D.7: *The figure shows the histograms of the* ***application time-differences*** *between requests of all tools when the* ***normal proxy*** *is used.*

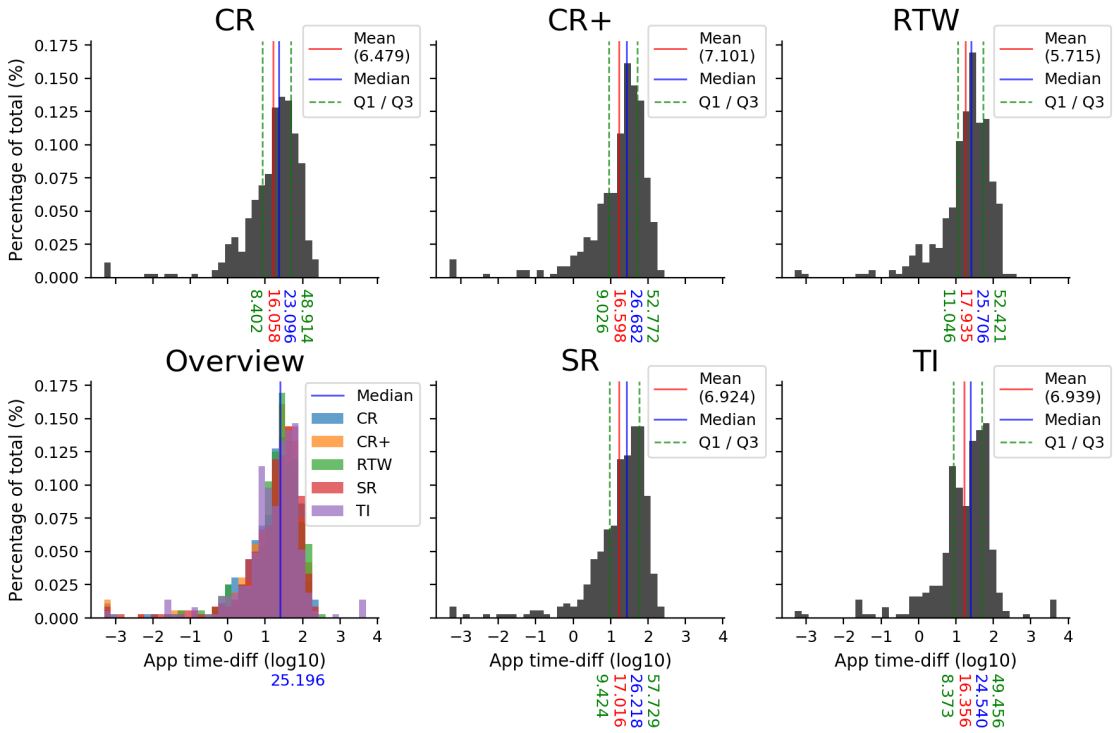


Figure D.8: *The figure shows the histograms of the* ***application time-differences*** *between requests of all tools when the* ***slow proxy*** *is used.*

D.3 Metric 2 - Test 1 - Voucher usage ratio

For all figures in this section, the y-axis shows the percentage of the 15 tests and the x-axis shows the total number of success codes divided by the number of vouchers used. A low number of vouchers used while the same number of success codes are returned indicates that more race conditions are triggered. Therefore, a higher value is better. The calculation of the mean (and standard deviation between brackets), median, the first and third percentile of the data is shown as well (in regular base 10).

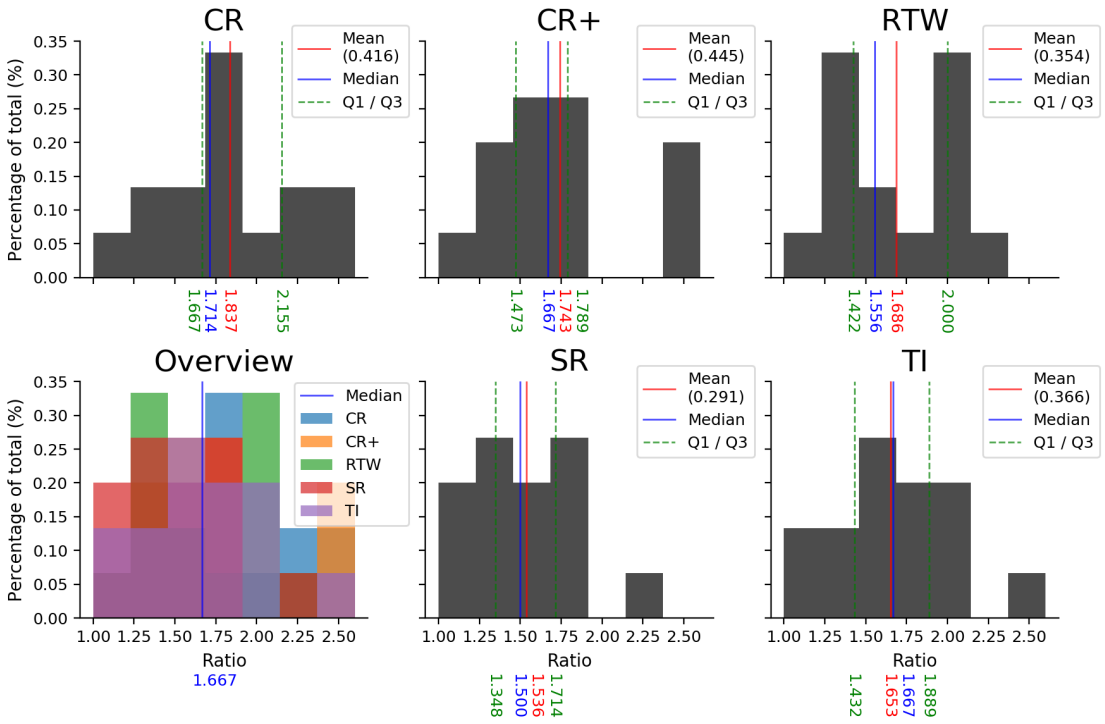


Figure D.9: *The figure shows the histograms of the* ***voucher usage ratio*** *of all tools when the* ***local server*** *is used.*

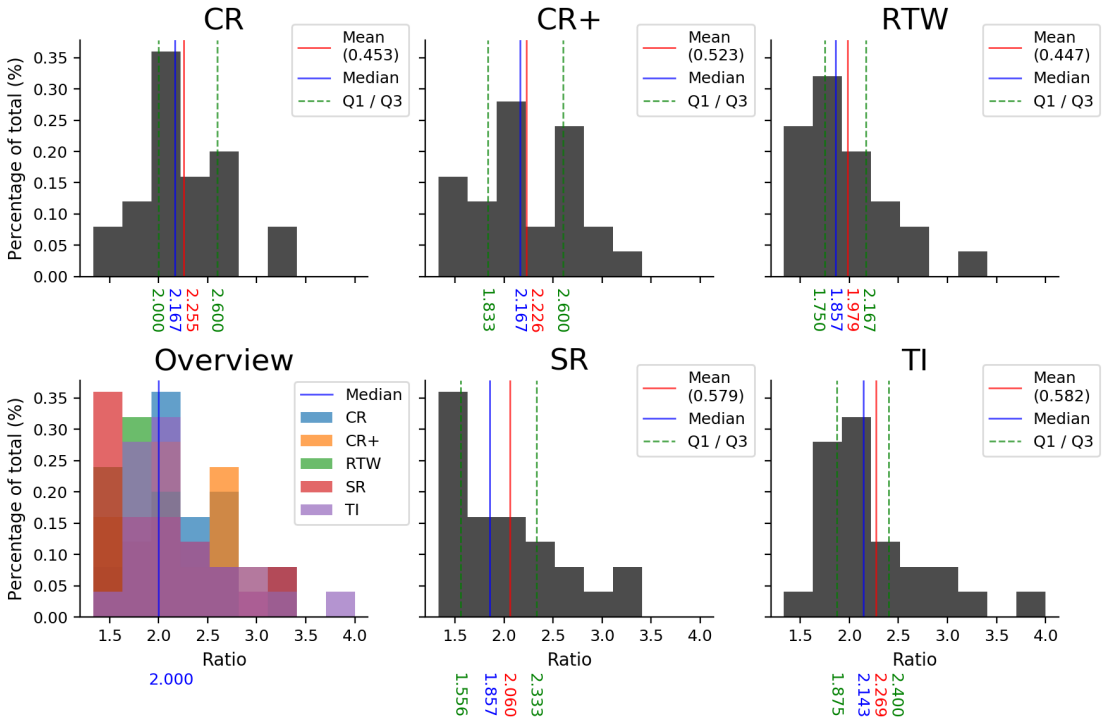


Figure D.10: *The figure shows the histograms of the* ***voucher usage ratio*** *of all tools when the* ***remote server*** *is used.*

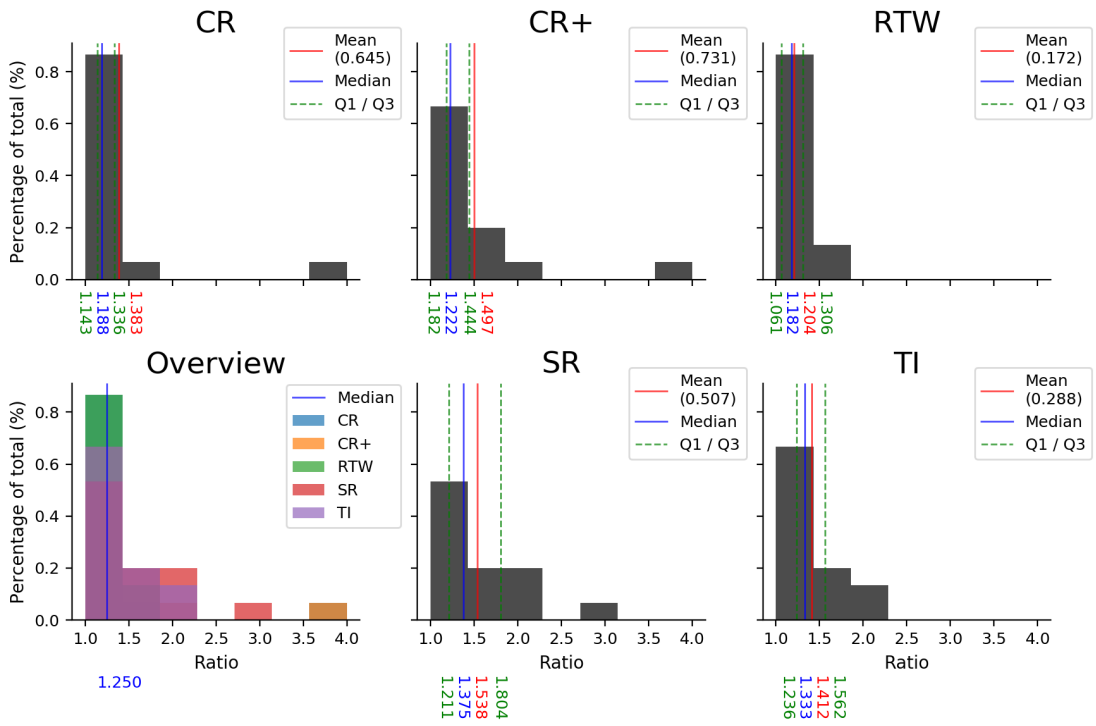


Figure D.11: *The figure shows the histograms of the* ***voucher usage ratio*** *of all tools when the* ***normal proxy server*** *is used.*

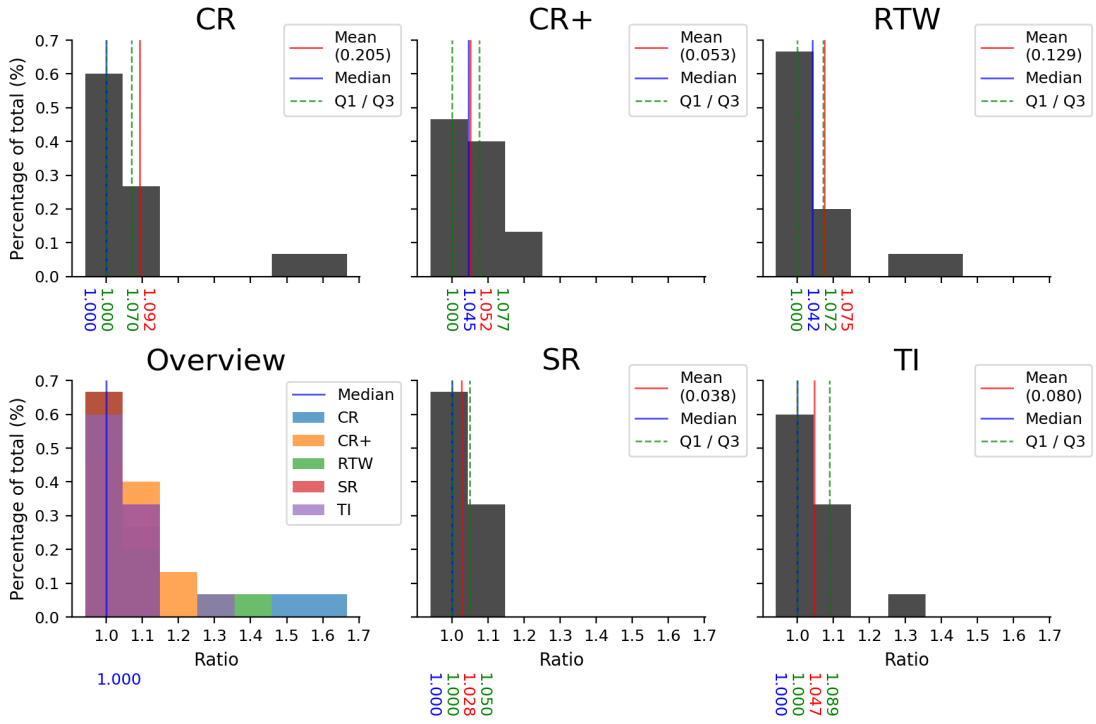


Figure D.12: *The figure shows the histograms of the* ***voucher usage ratio*** *of all tools when the* ***slow proxy*** *is used.*

D.4 Metric 2 - Test 2 - Number of success codes

For all figures in this section, the y-axis shows the percentage of the 15 tests and the x-axis shows the total number of success codes that were returned. A high number of success codes indicates that more race conditions could have occurred (as successful voucher redemption returns a success-code). A higher value also indicates that the attack was more stealthy as fewer errors occurred at the server. Therefore, a higher value is better. The calculation of the mean (and standard deviation between brackets), median, the first and third percentile of the data is shown as well (in regular base 10).

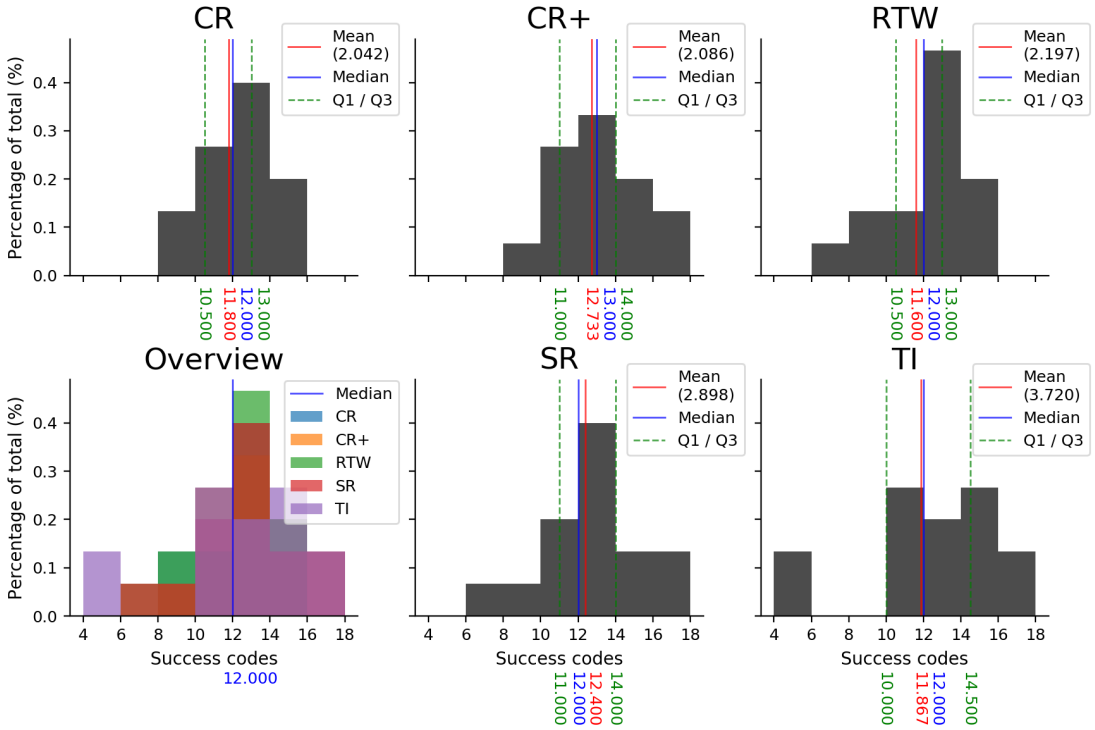


Figure D.13: *The figure shows the histograms of the* ***number of success codes*** *of all tools when the* ***local server*** *is used.*

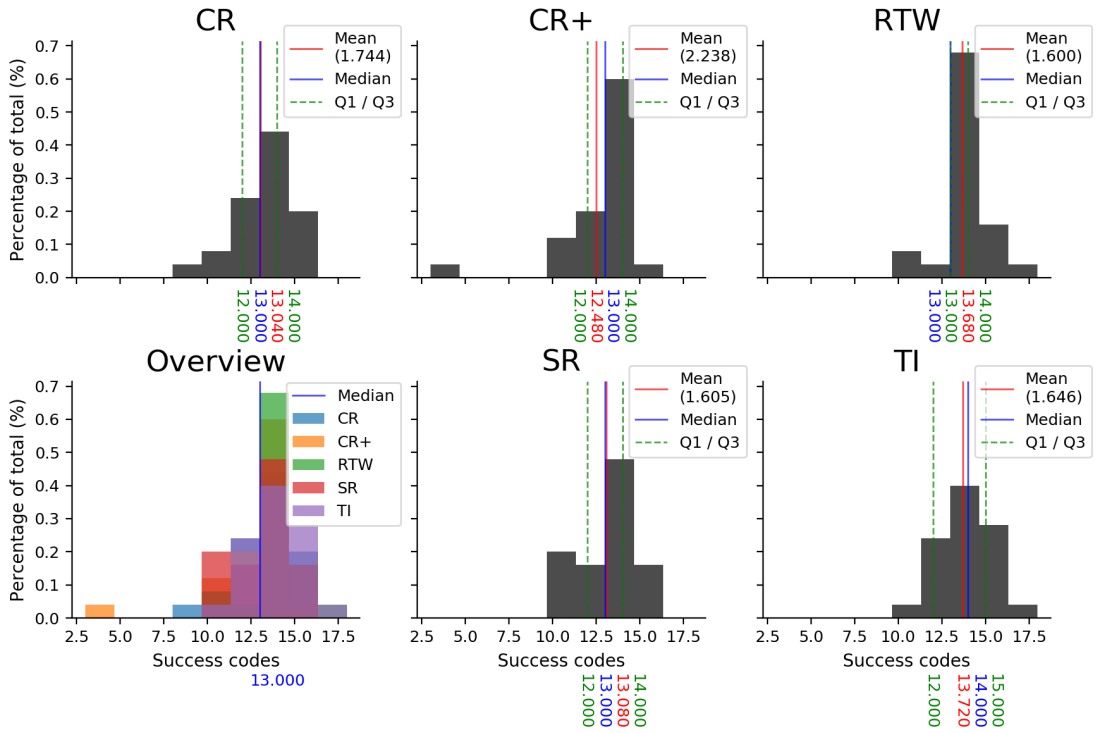


Figure D.14: *The figure shows the histograms of the* ***number of success codes*** *of all tools when the* ***remote server*** *is used.*

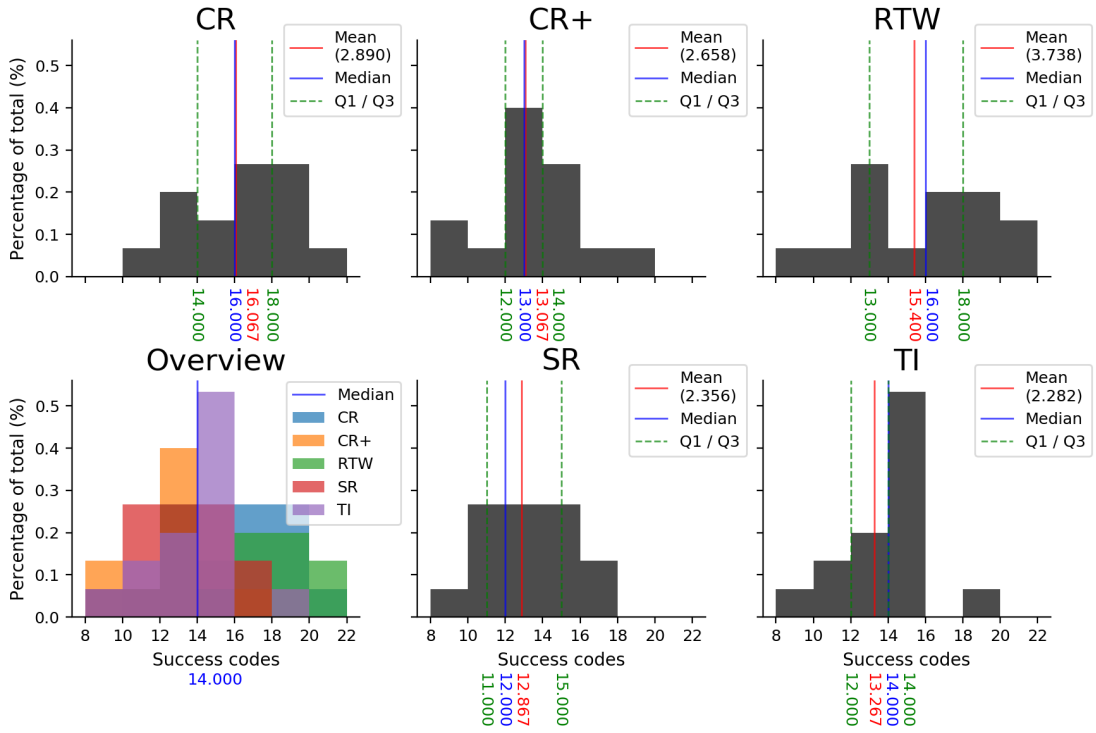


Figure D.15: *The figure shows the histograms of the* ***number of success codes*** *of all tools when the* ***normal proxy*** *is used.*

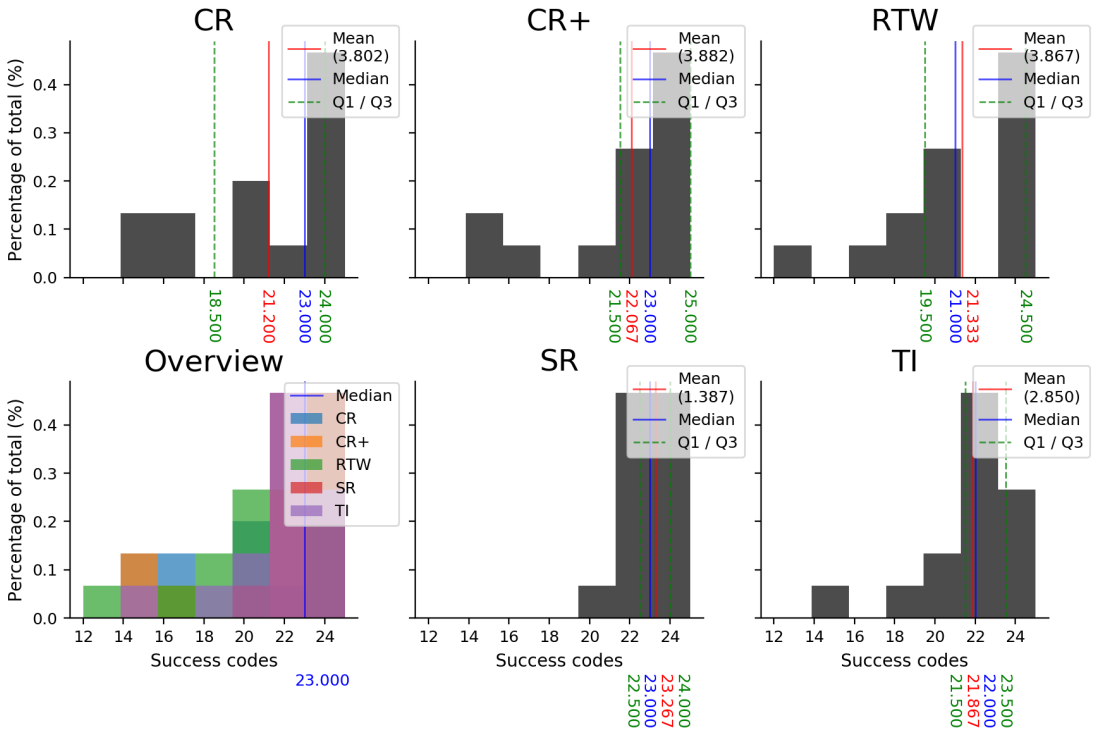


Figure D.16: *The figure shows the histograms of the* ***number of success codes*** *of all tools when the* ***slow proxy*** *is used.*

Appendix E

Responsible disclosure reports

In this appendix, the responsible disclosure reports regarding a race condition in the voucher redemption of two e-commerce webshop platforms: [redacted](A) and [redacted](B) are included. The first report (A) was sent to the security team of the affected e-commerce platform and the second report (B) was submitted to the responsible disclosure page on HackerOne of the company behind the platform[1]. Both reports were made in the name of Computest, the commissioner of the thesis. For both reports, at least the following parts were included: a summary, estimation of impact, setup regarding sources and versions used, guide on how to reproduce, estimation of the root issue and possible solutions.

Note about censored content As the issues were not solved before the publication date of the thesis, the details about the affected companies, software platforms and specific files in which the issues occur have been censored. For this reason, we were also not yet able to link to a published report on HackerOne. When the information is allowed to be disclosed, an uncensored version of this chapter will be added as an addendum on the official institutional page of the thesis[2].

Dear security team at [redacted](A), March 6th, 2019

As agreed upon, we have not encrypted this message.

Summary During research conducted for a Masters Thesis, we encountered an issue with the built-in cart rules of the [redacted]. Due to a race condition in

[1] This is where the link to the published HackerOne report will be.

[2] This page can be found at: `https://fmt.ewi.utwente.nl/education/master/322/`

the check whether a cart rule (discount percentage, or amount) has already been used, we can redeem them more times than allowed. This results in multiple successful orders that all use the same single-use cart rule, which should not be possible.

Impact Any user can buy and use his gift cards and discount codes multiple times at every webshop that uses the [redacted] CMS. Although the issue is rather obscure and exploitation not completely stealthy, with a possibly big financial impact and limited mitigation options, the risk of exploitation is still deemed rather large.

Setup We used a fresh install of version [redacted] on a Docker container downloaded from [1]. We did not change any default setting and used the sample shop contents when placing an order.

How to reproduce

1. Create a new cart rule (percentage or amount) from the [redacted] back-office which can only be used once in total: [redacted]
2. Add the new cart rule to a single shopping cart with at least one item to apply the discount. Go through the order process pages up to the last page and select 'Pay by bank wire'*.
3. Press the 'Order with obligation to pay' button and capture the HTTP POST request that places the order before it arrives at the [redacted] server-side.
4. Then, duplicate the request ten times (for instance) and send the requests at once to the server.

By using a self-developed tool, we were able to perform the last two steps. For a quick reproduction, we recommend using the simple open-source tool called Sakurity Racer [2].

Source of the issue The source of the issue is a Time Of Check - Time Of Use (TOCTOU) bug that occurs between the check: "Can I use this discount?" and actually applying the discount to the order and reducing the available number of discounts. Currently, this is not an atomic (uninterruptible) action, and when

the server runs at least two processes to serve the clients, this results in race conditions.

It seems that both when adding a cart rule to a cart [3], and when submitting an order [4], using the [redacted] function in [redacted].php, it is checked whether the voucher can actually be used. Among other things, it checks whether the left-over usage amount is larger than zero and that this user has not used the rule more often than allowed [5]. After this check, at a final stage in the order process and for every used voucher, its discount is applied, and the available amount is reduced by one [6].

Possible solutions We have to make sure that the check and application is one atomic (uninterruptible) action. A solution would be to use a 'SELECT FOR UPDATE' SQL statement when checking the availability of a cart rule. This locks the selected part of the database until the update (decrementing the available amount) is carried out.

Further steps For any questions regarding the disclosure, please do not hesitate to contact us. We maintain a 90-day disclosure deadline policy after which we will make the security report public. Next to this, we would like to be kept posted about the progress of verifying and solving this issue.

Best regards, Computest

* We are not sure whether other payment modules like credit cards or PayPal also show the issue.

[1]: [redacted]

[2]: `https://github.com/sakurity/racer`

[3]: at line 257 in function [redacted] in controllers/[redacted].php

[4]: at line 300 in function [redacted] in classes/[redacted].php

[5]: around line 639 and 658 in [redacted].php

[6]: at line 1172 in function [redacted] in classes/[redacted].php

Dear security team at [redacted](B), March 12th, 2019

Summary An issue was found in the voucher functionality of the [redacted]. Due to a race condition in checking whether a voucher has already been used, we were able to redeem a voucher more times than allowed consistently. This results in multiple successful orders that all use the same single-use voucher, which should not be possible.

Impact The security impact is a financial and possibly reputational loss for a **Wordpress** webshop that runs on the [redacted] plugin. The attacker only needs to buy one gift card or voucher and create multiple accounts. Then, he is able to use the voucher once for every account.

Setup We used a fresh install of version [redacted] of [redacted] and version [redacted] of [redacted] on a Docker container setup downloaded from the dockerhub [1]. It uses an Apache 2 web server and a MariaDB database. We then updated the [redacted] plugin to the latest version [redacted], installed the [redacted] theme and used the sample shop contents from [redacted] [2]. The server is ran and tested locally on a laptop (1).

Discovery process In an older version of [redacted], we were able to exploit this issue within a single user account by submitting an order with an attached voucher multiple times in parallel. This allowed for using a single voucher up to 10 times or more. In the newest version [redacted], submitting multiple orders in parallel was no longer possible. However, when we create multiple accounts and use the same single-use voucher in placing an order with all accounts, we could still use the voucher multiple times.

How to reproduce

1. Either buy a gift card or as an admin of the web app, create a single-use voucher in the back-office:
   - Go to: [redacted]
   - Fill in a coupon code, select any discount type and an amount > 0.

- Finally, in "Usage limits", fill in "Usage limit per coupon" = 1 and "Usage limit per user" = 1.

2. Go to the target webshop and create two accounts (2). Using two different browsers or the incognito mode of a browser, log in to both accounts.
3. For both accounts, add at least one product to the shopping cart, go to the checkout, add the newly created single-use voucher and fill in the order details.
4. Click 'Place order' for both accounts and capture the two HTTP POST requests that the browser sends to place the order on the server.
5. Send the two POST requests in parallel to the server.
6. Go to [redacted] in the back-office to verify the single-use voucher has been used two times.

By using a self-developed tool, we were able to perform the last two steps. This tool is not yet ready to be open-sourced. For a quick reproduction, we recommend using the simple open-source tool called Sakurity Racer [3].

Source of the issue The source of the issue is probably a Time Of Check - Time Of Use (TOCTOU) bug that occurs between the check: "Can I use this discount?" and actually applying the discount to the order and reducing the number of available discounts. Currently, this is not an atomic (uninterruptible) action, and when the server runs at least two processes to serve the clients, this results in race conditions. The same issue also makes it possible to order more items than in stock (when backorders are disallowed) using two orders from different accounts with the cumulative number of items that is more than the stock quantity. This shows that there is a general atomicity issue in processing orders.

Possible solutions We have to make sure that the check and application of a voucher is one atomic (uninterruptible) action. A solution would be to use a 'SELECT FOR UPDATE' SQL statement when checking the availability of a voucher. This locks the selected part of the database until the update (decrementing the available amount) is carried out.

Related minor issue Another race condition is found in the change email functionality of the accounts. When creating accounts, we cannot use an email address of an existing account. Even when two accounts are created in parallel,

this fails. However, when two existing accounts (with different email addresses) change their email address to the same new value in parallel. This works. It causes the second account in the database (by ID/creation date) to no longer be accessible. Only the first account can now be used to log in until he changes his email address again. Although this issue cannot be exploited, it is still a bug.

(1) We are aware that getting requests to arrive in parallel at a locally running web server is easier than when the web server is, for instance, on another continent. In that case, the unpredictable latency is massive. However, as most web apps are currently run on large hosting providers like Azure or AWS, any attacker could just rent a small machine in the same hosting location and attack from there reducing the latency significantly.

(2): If the webshop only allows for creating accounts when placing an order, skip this step and create the two accounts when placing the orders.

[1]: [redacted]

[2]: [redacted]

[3]: https://github.com/sakurity/racer