National College of Ireland

Exploiting and Preventing DoS attacks via Race Conditions

MSc Research Project

MSc in Cyber Security

Sahil Das

Student ID: 22211446

School of Computing

National College of Ireland

Supervisor: Mark Monaghan

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Sahil Das……
Student ID:	22211446……
Programme:	MSc in Cyber Security…… **Year:** 2024……
Module:	Practicum……
Supervisor:	Mark Monaghan……
Submission Due Date:	……
Project Title:	Exploiting and Preventing DoS attack via Race conditions……
Word Count:	…… **Page Count**……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Sahil Das……

Date: ……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Exploiting and Preventing DoS attacks via Race Conditions

Sahil Das
22211446
Youtube Link:-https://youtu.be/H9eIe7y5Ng0

Abstract

DoS attacks are a common yet not preventable issue nowadays in the systems and these can cause severe disruptions in the working procedures of organisations, causing financial losses on the way too. The data present in the systems sometimes also gets corrupted due to over processing of the data. One such ignored aspect of how the DoS arises is via the Race conditions where the timing and sequence of various processes can be manipulated to crash the system in such a way that the least suspicion arises. By examining and cloning various race conditions onto web apps we can find out which techniques were mostly used to leverage the DoS attacks. Through various comprehensive, simple to understand analysis we will try to figure out how various race conditions can be leveraged to DoS attacks. Once this aspect is done, various preventive measures will also be implemented and these measures will also be tested as to which of them can be the most effective. The proposed solutions include but are not limited to secure coding practices, tweaking various aspects in the cloud, using secure deployment measures and so on. The various parameters of effectiveness are measured using tools such as Prometheus, Grafana and Google Pagespeed.

Keywords:- Race Conditions, DoS attacks, WebApps, Monitoring

1 Introduction

In the given times, the Internet and websites have become an important place for connectivity and doing business. This has increased the options for performing a bigger cyber warfare operation ground. One of the most common methodologies used in this case is the DoS attacks which technically exhausts the resources hence compromising the availability and reliability of the systems. It makes sense to believe that a denial of service (DoS) occurs soon after the attack is launched because the attacker wants to be able to access every connection on the targeted host as soon as possible. It is possible, though, that the server has already established real, continuing relationships with other clients. Those connections live until they are broken. As soon as a connection closes, the server releases the related resources, allowing clients to establish new connections. The attacker's objective is to replace all of the "just available" connections with malicious ones as a result. While doing so, there may be a race condition between the attacker and a few other real clients. In the current scenario of complexity, business logic vulnerabilities are most common victims of Race conditions. The attacks also happen at various layers of the systems but recently they have moved more towards the business layer/application layer which are much easier to execute as compared to other layers of the OSI Model. Business layer DoS attacks are more application specific and there is not much malicious traffic in the website but in other words it can be said as more bogus packets are sent to exhaust the systems. Detecting these loopholes needs a

better understanding of the web logic. Sometimes the flaws happen due to programming errors or maybe the lack of implementing a proper deployment measure and so on. Sometimes the detection of these business logic issues can be a bit difficult and a lot of automated web scanners won't even recognise them and the reason being a lot of these automated scanners do not understand the web logic of every single application they scan .
The aim of this research is to focus on the various aspects of race conditions which can be further leveraged into DoS attacks. Various types of business logic and other types of flaws in the web application will be listed which can be the starting point for these types of attacks. Once the ways of these attacks are figured out, then the preventive methods suitable to our setup will be implemented which can be load balancers, tweaking changes in code, changing cloud parameters, rate limiting. The various factors of implementing the security measures will be discussed in detail in the methodology and implementation.
This paper discusses the various aspects of how the race conditions are leveraged into DoS attacks and what can be the ways those issues can be prevented further. Also various experiments will be performed on the websites to find out which methods were effective. For that, tools such as JMeter, Scripts in Python, Turbo Intruder in BurpSuite and a few more will be used. Apart from that various tools such as Prometheus, Grafana will be deployed in parallel and connected to the websites or the servers to find out the parameters such as latency time, time to first byte, number of requests per second, cpu utilisation which can be used to determine which preventive measures are working effectively, although it is true that no measure will have the 100 percent capability to prevent the attack but we will try to improve as much as we can.

2 Related Work

For the above research question I will be going through various other research papers which will give a brief idea on how various researchers have implemented the things where there can be the shortcomings and how using my research methodologies I can attempt to fill those loopholes. These are some of the papers below which were referred

2.1 Automated testing techniques for event driven and dynamically typed software applications-Christoffer Quist Adamsen

In this paper they talk about various methods of detecting and preventing race conditions in AJAX and JavaScript web applications. In this case, Adamsen (2018) focuses on various aspects such as the race error occurrence due to the developers making false assumptions about a particular business logic. They often find that developers do not take into account the fact when the users start interacting with an application even before it has fully loaded. In order to prevent these conditions from happening, the researchers prepare a catalogue-like thing which contains various repair policies which will be providing a clear idea to the application on how to perform if any such actions happen. They also tried to implement policies which can make the application work in a single threaded mode when the race errors become too huge to handle for the repair policies. They also suggest that concurrent errors are also much easier for automatic repairs. There are some issues with doing this because while creating a catalogue for various repair policies they covered only a few aspects of the business logic errors and they didn't cover the ones which can actually happen on a rare basis. Also for extreme cases they are trying to switch the application from a multithreaded environment to a single threaded environment which can also be problematic for the efficiency of the application because if the application is meant to run multiple functions and all of a sudden it's only able to do one function then chances are the application can be rendered useless.

.

2.2 Detecting JavaScript Races-Erdal Mutlu, Serdar Tasiran, Benjamin Livshits

Here Mutlu, Tasiran (2015) talk about distinguishing between the benign race conditions and harmful race conditions. Not all race conditions can be harmful because in some cases due to the race conditions there can be minor glitches such as slowing down of some user functions or maybe some minor issues in the UI Side. Since given the forgiving nature of javascript execution where a failure in the execution, the event handlers force the schedulers to terminate the current handler and start a new one. They also focus on the data races which have a persistent state in the terms of data storage. The data race conditions in this case are achieved via the POST calls to the server. The major issue in this case is that the authors pre assumed the fact that the minor race conditions which cause glitches in the UI are not impactful. But in a lot of cases that is not true because in case of some large web apps it can be conflicting for the users and if due to the glitches the user could not perform the needed functions, then it can be harmful for the businesses operating that website as well as the users losing trust in the web app because of low quality processing.

2.3 Understanding and Detecting Concurrency Attacks- Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui

Wang, Cui, Li (2018) in this case have identified two main features that were further leveraged into attacks which could have bad consequences. The corrupted memory often resides a long time in the system memory. Out of 31 cases for concurrency attacks, around 14 were not able to be detected by the developers. This corrupted memory feature is something which is very hard to detect. Secondly the inputs that were used to induce the concurrency bugs out of which 12 cases were taken and 10 of those cases were requiring more subtle and detailed input to conduct the attacks. Of the instances, one of the Linux instances which had privilege escalation and in order to trigger the memory corruption only needed two threads which could be crafted. Also to trigger the root privilege a third socket is required to call the socket() to allocate a SELinux label structuring which needs the kmalloc32() structure. This structure should be located in the proximity of where the corrupted memory is present. The main issue was the fact that for the detection of new concurrency attacks with reasonable scalability it was difficult because the concurrency attacks samples gathered were more generic and in order to categorise them it was very hard. It was also hard to find the false negatives in this huge chunk of data also. Also for the valuation of the experimental detection very few datasets were used.

2.4 Race detection for web applications-B. Petrov, M. Vechev, M. Sridharan, J. Dolby

The researchers in the above case try to formulate a happens-before relation that captures the key ordering constraints for the most common web platform features. and they also try to define operations that define atomic execution. They also try to present a logical memory access model for the applications that abstract away from the implication details. There can be difficulties while defining the relation paths for happens-before relation because of the very vague specifics and sometimes browsers have also different levels of compatibility which might cause the specific parameters to fail. Due to failure of these parameters sometimes it can be difficult to decide whether the race condition is an actual/legit one or not.

2.5 On Race conditions in web applications-Roberto Paleari, Davide Marrone, Danilo Bruschi, and Mattia Monga

Paleri, Marrone, Monga(2008), mostly discuss the various types of impacts that a race condition can have on the website when there are frequent interactions between the website and the relational databases. They mostly focus on the dynamic detection of the loopholes because of the fact that the developer mostly ignores the underlying environment of the website which has mostly parallel execution methods. The main disadvantage of the research in this case was they mostly considered a single execution path or style for their dynamic detection, they did not take into account how the data will be retrieved from the backend. Apart from that it also does not take into account any synchronisation method that the website could take into account for its execution.

3 Research Methodology

This research paper methodology consists of several steps for the implementation of the project which includes: Setting up the server and its related components, Installing and deploying the application, Preparing the tools and scripts, Evaluating the applications and applying preventive measures if possible. For the base methodology I have considered certain aspects from the research paper titled as Race detection for the web applications-B. Petrov, M. Vechev, M. Sridharan, J. Dolby. The only issue that came up while implementing certain aspects of the paper was that a lot of the tools and libraries needed for the applications were obsolete or didn't have any support from the main organisations due to which installing them or executing them became a tough task. So I had decided to change certain areas of the methods for more improved and sophisticated testing and evaluation of the system. I also modified certain aspects of the methodology to perform the attacks on the websites.

In the first step compared to the methodologies of Maranda (2021) which is setting up the server and its related components which involves mostly selecting the right virtual machine with the needed specifications. For the case of this thesis, I will be working on mostly AWS where the Linux machines will be deployed as EC2 Instances and most of them will be attached with some basic features like the device storage, load balancers and instance health monitoring. The only things which will be manually configured are the Route Tables, Internet Gateway, IAM and subnet. Once the setup for the instance has been done, then I will work on setting up the Kubernetes cluster for the system which will act as the base for deploying multiple web applications. For that I will be installing Kubernetes using some bash scripts which contain all the commands and packages including for docker engine that will be helpful for installing the Kubernetes Engine. Before getting started with that process, first I will create IAM Roles in AWS for the master and the worker node which in this case will be the EC2 Instances. The reason for doing so is because this will allow the machines to communicate well with each other and the apps can operate within restricted boundaries. Once the Kubernetes master and worker node engines are installed they will be connected with each other using the token which will be created from the kubeadm join command. Once the installation has been done, the working of kubernetes can be verified by typing any of the commands such as kubectl get namespaces.

Coming to the second part which consists of installing and deploying the application. The applications mostly used in this case will be written in django and in order to install the libraries the Dockerfile will be used which is further containing the libraries and other commands that will help to install the base system to operate the application. Once the application image has been created then a deployment and service file will be written in the YAML Format in which the deployment file will be used to execute the image continuously and also provides a directory with storing the data. The service file helps to expose the image to the external clients which can be used for communication between the two parties. Once the images have been mentioned properly in the deployment file and the proper ports are set, then those yaml files are applied. Also various ingress files are mentioned depending on the application type for allowing the apps to be accessible from anywhere by just using a domain name or IP Address.

The third step invoices the creation of scripts in python for testing race conditions. Following the footsteps of Rothermel (2017), One of the scripts is just a basic testing one where the name of the host and the other parameters are to be directly edited in the code and the code creates a custom thread class containing HTTP POST Requests. It creates three instances of the thread to transfer the requests concurrently. Also a raw request will be constructed. The request line includes POST / HTTP/ 1.1 which means a request will be made to the root directory of the website with HTTP / 1.1 and also various parameters are also mentioned for starting the thread successfully. In another scenario a docker image will be deployed locally as well to test the system. which is mostly a simple page which displays various products. So in this case lets say a product needs to be bought which is more than twenty bucks so the manual task which can be done in order to buy it sells a product more than two times which will give enough credits. Knowing the requests which will be used to buy and sell products. The other features that will be added in this code will be an aspect that one part of the code will buy the product one time and it will be sold N number of times. Now only a function will be added which will find the first transaction id and spawn the threads. After executing this script a section will show up which will display the products bought.

Coming down to the next step which is evaluating and preventing the attacks if possible. For the evaluating section tools like Apache JMeter will be used for initial testing which will include the sending of multiple requests at the same time to see how the server handles them. This will give an idea on how the server handles the initial requests at the same time. There can be chances of server crash so in order to prevent it a load balancer will be added first and the experiment will be repeated again. If it doesn't work out, certain parameters in Kubernetes will be modified to prevent the crash. Apart from the basic JMeter tests various scripts written in python will be tested as well to find out how the things work out. For instance in the second case the issue in the docker image can be fixed by adding some changes in the python backend area which involves adding a login verification and also securing the endpoints of the POST Requests which involve limiting the number of requests.

4 Design Specification

1. Implementation Techniques
    - Programming Languages: Python(Django), YAML, JSON.
    - Coding methods used: Waterfall and Agile
    - Testing: Unit Testing

2. Architecture

   System Overview

   The system used in this case mostly consists of Self managed Kubernetes Cluster which mostly consists of AWS EC2 Instances which is covered by Load Balancer for additional protection for the systems. Alongside on the upper level also VPC is provided which will help in private communication between two machines. Also within the system Docker and Kubernetes will be installed which will help to manage various containers in the system and alongside they all will have an ingress file which will provide external access to the system.

   Data flow

   In this case the request is generated from any web browser to the domain where the request passes through the load balancer and then enters the system where the ingress file pointed to the container receives it and forwards it to the containers which further provide the needed content.

   Deployment Structure

For the thesis the majority of the deployment is done on AWS. First AWS EC2 Instances were set up and alongside with it VPC is set up and needed IAM permissions are also given. Once the setup is done, then certain bash scripts are added in the system which will install docker and kubernetes automatically and also provision the load balancer as well.

3. Diagram

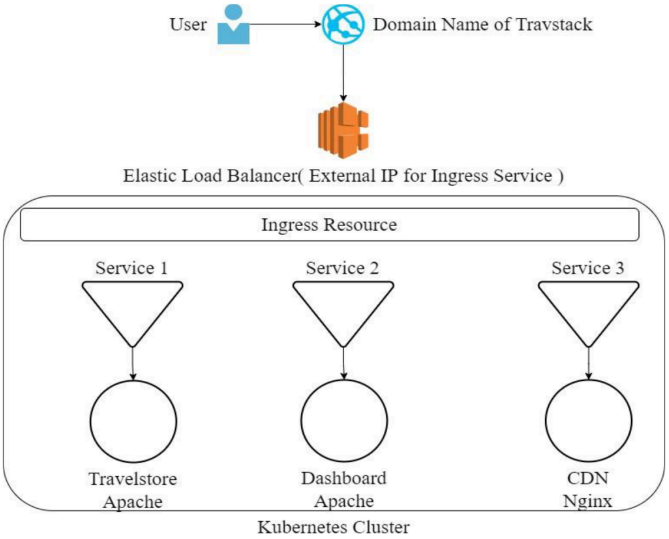

Fig 1. Kubernetes Server Architecture

5 Implementation

For the implementation stages mostly the apps were deployed both externally in Kubernetes and also a local deployment was done in docker which was further tested using various python scripts. This can be divided into various sections

Experimental Setup:

1. Target Applications: The apps targeted in this case are mostly written in Django
2. Toolset: Apache JMeter, Kuebrnetes , Docker, Burp Suite

Implementation of Exploitation Techniques

The race conditions were mostly identified via manual testing and in certain cases the static analysis of the code. For most of the cases the requests were manually generated from different sources such as sending a purchase request to the website from two different virtual machines. Also different login credentials were used at the same time using various tabs on chrome as well to see how the website responds. In the case of the simple python app which was deployed on docker, a manual review of the code was done which helped us to understand which parts of the code caused the race condition issue.

Development of the Exploit code

Once the working mechanism of the race condition in the code was understood, the needed POST Requests were also generated using python scripts and also the functions were also added which kind of mimicked the action of buying one product and selling infinite products. In the case of other websites a python script was written which will generate various types of raw requests which can be either GET or POST Requests which will try to perform various functions on the website at the same time and also the threads will respawn as well.

Implementation of the prevention measures
Code review guidelines: While sending the buy and sell requests, it was intercepted using Burp Suite which gave the exact area where the issue can be in the code. Using the method the code was reviewed which showed the loopholes and basically after doing a manual review the code mechanism was understood and it was clear which snippets needed to be added to reduce the attack surface. For the other websites which was deployed in production the tools such as JMeter and python scripts were used to check and find out where the issues came up as well

Secure coding aspects: In order to secure those areas various snippets were added in python which were used to cover up the loopholes present in the code. Also the initial protection measure of adding load balancer worked partially so certain changes were done at the ingress level of the code and also in the deployment file of the kubernetes which helped to allocate more resources to the container in order to handle more requests.

Security Testing: Once the needed measures were implemented then again the same set of tests were run to find out if the loopholes existed or not. For instance in one case there was one scenario where the website slowed down but it didn't crash completely after the implementation of the security measure at the cloud level. Also for the code deployed at the docker level, the same tests were done in two cases: the section that showed items bought were displaying partial information which indicates that the preventive measures worked successfully to a certain extent.

You will of course want to discuss the implementation of the proposed solution. Only the final stage of the implementation should be described.

It should describe the outputs produced, e.g. transformed data, code written, models developed, questionnaires administered. The description should also include what tools and languages you used to produce the outputs. This section must not contain code listing or user manual description.

6 Evaluation

For the evaluation of the attack methods alongside with which protection measures work the best various methods were tested and their results were recorded along the way on how the application reacted when the attack happened and what happened to the application when the protection measures were applied and the attack was re performed.

6.1 Experiment / Case Study 1

In the first experiment a normal travel website which has a backend that is written in django was exposed with just a simple load balancer from AWS and it didn't have any other protection. So initially Apache JMeter was configured to send multiple requests to the website. Initially the website was able to handle the requests but slowly the website became slow and also showed a 502/503 error even after allotting a load balancer. This shows the load balancer was not that successful in stopping the attack but it did slow down the attack rate initially. The race condition execution was successful to a certain extent.

6.2 Experiment / Case Study 2

In the second experiment a small shopping website was deployed on docker locally to test another race condition which was mostly related to time based execution and the buying/selling requests were intercepted by configuring burp suite with the browser and modifying the requests accordingly which helped to buy certain products without using extra credits. And now the exploit was replicated into python code and it did send requests which caused the race condition to execute and given the fact dockerise library was also added by modifying the docker image as a protection measure but the same case happened where the race conditions could not be prevented. In order to improve the protection in certain aspects the working mechanisms of the code were changed which basically locked the row in the database and deleted the item from the stock database once the purchase had been done. And again executing the attack this time the protection measures did work which shows that sometimes it's important to observe the working mechanism of the code which can be a great way to find the attack prevention measure.

6.3 Experiment / Case Study 3

In the third case an admin dashboard login was at our disposal which was related to the travel website. In order to perform the race condition various valid and invalid logins were considered and a script in python was designed to perform the same action. The various parameters were collected and set up accordingly. AWS WAF Protection was added to provide protection against the crashing of the site .Then the text file containing the username and passwords are included as well. This is similar to bruteforce but not exactly the same because in this case there is a clear idea as to which passwords are right or not. Once the script is executed it will show up the error and the successful login attempts and executing this script in a loop will sometimes cause the page to slow down or sometimes show an database error due to the fact that there is certain time difference when the requests are sent and also login page is not coded properly enough to handle the errors. This issue was solved by adding certain rate limiting logic and again the same experiment was repeated with the modified logic. This time the website slowed down a bit but didn't crash or gave many errors which shows the logic was the main issue in the code.

6.4 Experiment / Case Study 4

In the fourth case the main website was tested for the race condition to gain access to the admin page. This time a protection measure has been implemented in the ingress file of the kubernetes deployment for the app and it is just a parameter which basically provides rate limiting facilities. A python script will be written which will try all the right and the wrong credentials more or less like a bruteforce attack. When the script is executed it keeps on blocking the requests concurrently and there is a time when the backend API is unable to handle the requests but the app is still prevented from crashing due to the presence of the rate limiting configuration code added in ingress which blocks the requests after a certain threshold is reached. This shows that in certain cases a proper configuration of the deployment environment is also important alongside with writing the code keeping the race conditions in mind.

6.5 Experiment / Case Study 5

In the fifth experiment the travel website is targeted for DoS via race conditions and this time both the code has been rectified in a way to handle the multiple requests and also the ingress file parameters have been added for providing the rate limiting facilities. For additional protection the load balancer has been configured alongside. This time both the Python script and JMeter will be used

simultaneously to see how many requests the system can handle. For doing so the python script is executed on the virtual machine and the JMeter will be run as usual on the system. But this time the website slows down a bit but doesn't crash at all. Though this configuration works well it did reduce the efficiency of the website and also costed a bit high due to the usage of load balancer which may not be feasible for long run.

6.6 Discussion

Various experiments were conducted on the websites and it was found that the race conditions could mostly be prevented by a proper coding of the website but in certain cases certain aspects of the cloud deployment can be changed to provide an additional layer of security in case the code was not designed correctly. Configuring these additional configurations for the systems can be helpful because in certain cases it is evident even when the website was tested for DoS attack via the race conditions, the website did not crash or became unresponsive fully but still it just slowed down a bit. Apart from that the system was tested by providing a simple load balancer but it did not work that effectively and the malicious requests were still able to make their way through the system. There were some faults with the experiments also. The major one is just testing out the website by covering them with load balancer which kind of hindered the effectiveness of other configurations. Also most of the testing was focused on one programming language and just one aspect of the race condition which is the DoS attack whereas in reality a race condition can be used for many other purposes as well. Apart from that the testing procedures could have been improved by trying to leverage the race condition into some other form of attacks and also the defence mechanisms could have been more complicated given there were more resources. The experiment did focus on the coding aspect but the majority focus was mostly securing on the cloud side assuming the fact that in certain cases the code cannot be changed. The major improvement that could have been done is adding some detection features which could have helped to identify the loopholes quickly since in this case majority of the testing and implementation was black box based.

After analysing the data from the monitoring tools these were the parameters that show how efficient each defensive measure was. The data measurements were taken from Google PageSpeed and Prometheus

The table below shows the data for the travel website when the race condition exploits were executed

Defensive Measure	Initial Server Response time (in Milliseconds)
Load Balancer	408ms
Kubernetes Parameters	270ms
Changing Code Logic	230ms

Table 1: Response Time

Defensive Measures	Latency Time(in Milliseconds)
Load Balancer	113ms
Kubernetes Parameters	99.3ms
Changing Code Logic	82.4ms

Table 2. Latency Time

7 Conclusion and Future Work

The major aspect of exploiting and preventing the attacks was covered in this thesis displaying various experiments which showed how the attacks happened and how various protection measures worked depending on the situations. Majority aspects of the question were covered but only the detection part was not covered extensively. The main findings in the thesis was mostly about various methods of secure coding which when tweaked can give a good level of protection alongside the experiments. They also show that sometimes having a good understanding of the deployment systems would be a good idea since this will also be a blocking point for the attacks which can happen in the future. This thesis provides an overview on how the attacks can happen, and how the prevention methods can work or maybe they can be bypassed. This is an important aspect to cover because this thesis shows a way of how the attacks might evolve and how chaining other loopholes can render the defence mechanisms useless. The main limitation in this case was the majority of the race conditions were tested on the website which was coded in django and the testing was mostly done on one environment which was deployed on Kubernetes and AWS. This just limits the idea of how the race condition works and there is no guarantee the same methodologies would work. The other limitation is the majority of the focus was towards leveraging race conditions into a DoS attack which is sure a huge problem but the only issue in this case is race conditions can be leveraged into other forms of attacks such as stealing data which was not focused much. Another limitation was the majority of the detection of the race conditions was done in a manual way which was done for the purpose of studying each loophole carefully but this process could have been automated to a certain extent.

In terms of future perspective, adding various automated methods of detection of the race conditions with a high accuracy rate is something I would look for which I wasn't able to implement in the thesis due to the time restrictions and other factors. Creating an AI Model which will pickup various attack methodologies including the ones discussed in the thesis and training it accordingly. Apart from the detection phase certain modules/code snippets could be added which will provide a basic way of stopping the attack. The reason for extending the project is because detection of the points in the code is also important to find out where the issues can occur, hence saving a lot of time on the static analysis.

References

Adamsen, C.Q., 2018. Automated testing techniques for event-driven and dynamically typed software applications.

Mutlu, E., Tasiran, S. and Livshits, B., 2015, August. Detecting JavaScript races that matter. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (pp. 381-392).

Gu, R., Gan, B., Ning, Y., Cui, H. and Yang, J., 2016. Understanding and Detecting Concurrency Attacks.

Петров, Б.И., Vechev, M., Sridharan, M. and Dolby, J. (2012). Race detection for web applications. doi:https://doi.org/10.1145/2254064.2254095.

Poniszewska-Marańda, A. and Czechowska, E. (2021). Kubernetes Cluster for Automating Software Production Environment. *Sensors*, 21(5), p.1910. doi:https://doi.org/10.3390/s21051910.

Yu, T., Srisa-an, W. and Rothermel, G., 2017. An automated framework to support testing for process‐level race conditions. *Software Testing, Verification and Reliability*, *27*(4-5), p.e1634.

Paleari, R., Marrone, D., Bruschi, D. and Monga, M., 2008. On race vulnerabilities in web applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings 5* (pp. 126-142). Springer Berlin Heidelberg.