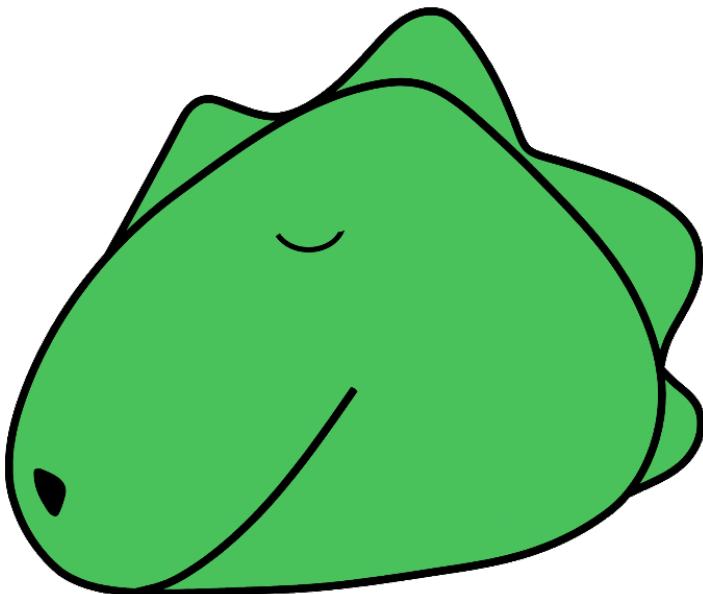


OWASP TimeGap Theory Handbook



Abhi M Balakrishnan

For my lovely wife Sarika and dear son, Josh.

[Attribution-ShareAlike 4.0 International \(CC BY-SA 4.0\)](#)

You are free:

- [To share](#) - to copy, distribute and transmit the work
- [To remix](#) - to adapt the work

Under the following conditions:

- [Attribution](#). You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work)
- [Share alike](#). If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license

Some images are subject to copyright

ISBN: 9798665526812

	Chapter 0 Welcome!	1
	Chapter 1 TOCTOU	5
	Chapter 2 Lab Setup!	11
	Chapter 3 A Quick Refresher	19
	Chapter 4 Challenge 1 - Sign Up	35
	Chapter 5 Challenge 2 - Login	51
	Chapter 6 Challenge 3 - Transfer	65

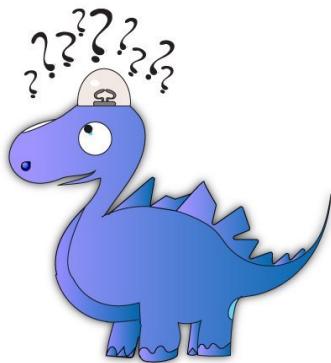
	Chapter 7	Challenge 4 - Mars	81
	Chapter 8	Challenge 5 - Tickets	87
	Chapter 9	Challenge 6 - Coupon	103
	Chapter 10	Challenge 7 - Ratings	111
	Chapter 11	More Tools	125
	Extra 0x00	Credits	135
	Extra 0x01	Addendum	139



CHAPTER 0

Welcome!

This handbook is all about TOC/TOU vulnerabilities.
Welcome aboard and let's get started!



People often ask, "Why should I bother with such a trivial vulnerability?"

- **TOC/TOU** is not in [OWASP's Top 10](#)¹
- It is not even in the CWE's Top 25 Most Dangerous Software Errors²

However, that does not make **TOCTOU** irrelevant. One open (or broken) door is enough for an attacker to break in. That open door doesn't necessarily need to be reflected in the [OWASP](#) Top 10 or CWE Top 25.

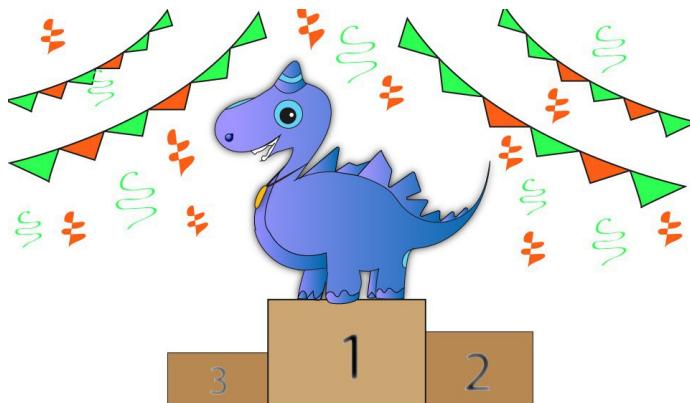
This guide is exactly about this kind of door. We are talking about [time of check to time of use](#), often abbreviated as **TOC/TOU**. Unlike cross-site scripting and SQL injection, this door is slightly harder to locate and find. But once open, it can be pretty dangerous.

People sometimes refer to **TOC/TOU** as **TOCTOU** and **TOCTTOU**. No matter how you write it, the right pronunciation is “**TOCK too.**”

This guide will walk you through seven **TOC/TOU** scenarios. Note that the application, **OWASP TimeGap Theory**, is designed to be vulnerable to **TOC/TOU**, but not all web applications are vulnerable. There are several methods to safeguard applications from **TOCTOU** issues. We won’t be discussing them here as they are beyond the scope of this handguide.⁷

However, by the end of this guide:

- You will be a **TOC/TOU** Champion
- You will be equipped with tools and techniques to check if your application is vulnerable to **TOCTOU**
- You will be able to forecast **TOCTOU** issues on application by looking at the high-level design in threat-modeling sessions
- You also will be in a position to demonstrate **TOCTOU** issues to your peers





CHAPTER 1

TOCTOU

What are these issues?

How do they become security issues?



From Wikipedia: In software development, time-of-check to time-of-use is a class of software bugs caused by a race condition involving the checking of the state of a part of a system and the use of the results of that check.³



Confused? Let us look at a scenario:

1. Your friends are at a restaurant
2. You joined them late
3. You noticed an empty chair and proceeded to sit there
4. Your friends want to shake hands with you, so you stand up again
5. Meanwhile, one of the friends pulled the chair
6. You proceeded to sit but ended up falling on the floor



There were two threads in the above scenario. One was you, and the other one was your friend. Both of you were accessing the same variable, i.e. the chair. Your friend acted on the variable and made a change. You were not aware of the change and proceeded to sit on the variable - chair.

Let us look at another example:

1. You and your brother are running a mattress store
2. A customer approaches you
3. The customer asks you if they can get two sets of pillows
4. You check if you have enough stock
5. You have exactly two sets of pillows available
6. You bill the customer
7. You go back and find only one set of pillows in stock
8. Your brother already sold one set of pillows to another customer

Do you see where this is going?

1. You did your job
2. Your brother did his job
3. However, both of you did not talk to each other
4. In the end, one customer was unhappy
5. You also were left unhappy which eventually made your brother unhappy



These scenarios can be applied to software development. You think of a problem, write a simple solution, and then deploy this code. During runtime, the code will run in parallel threads, but these threads do not talk to each other. Neither do they share a common state. They often share the same variables (storage units) in memory or in a database. What if one thread modifies these variables while the other thread is still working on it? This can cause concurrency issues.



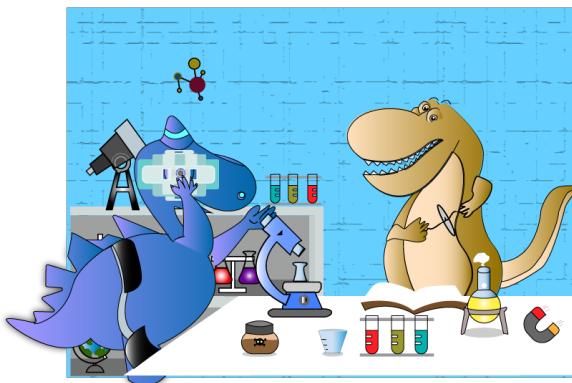
In software applications, these concurrency issues can lead to annoying bugs. Sometimes these bugs can turn into bigger security issues. Ultimately, these security issues can lead to major business risks. You might ask yourself how? We will discuss that next in this guide.



Lab Setup!

Lab setup is easy!

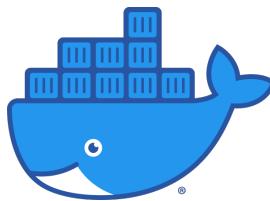
The entire lab is ready for you to download and run.



First, we need to get to the download and install some software. There are two ways forward:

1. If you're planning to launch tools or scripts against the test instance, set up a local **OWASP TimeGap Theory** lab
2. If you're planning to perform light-weight testing using a browser, use the online version

Local OWASP TimeGap Theory Lab using Docker

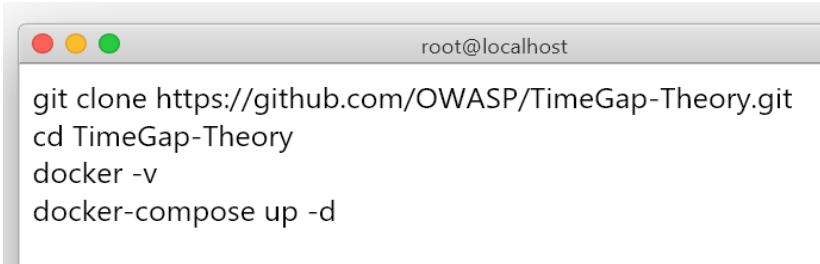


Follow the next steps if you are planning to set up a local **OWASP TimeGap Theory** lab.

- Approximate time required: 20 minutes
- Budget: **\$0, no paid software required**
- Prerequisite: **Windows Pro / Mac / Linux computer**

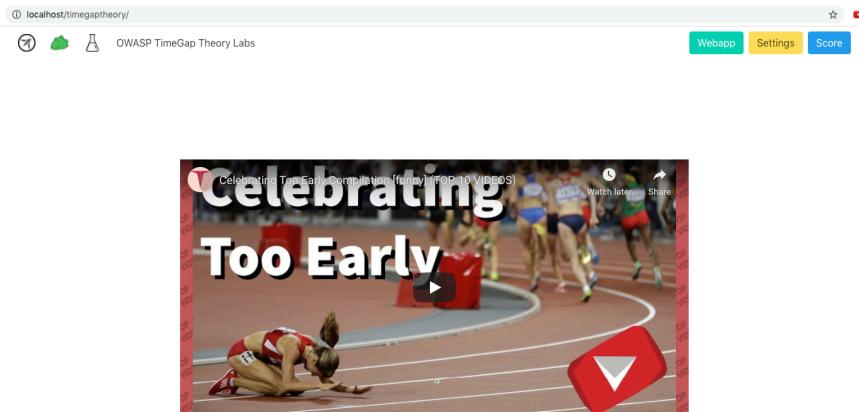
1. Install **Docker** (the free community edition is okay.)
 - a. <https://www.docker.com/>
 - b. <https://docs.docker.com/>
2. Have at least two browsers installed
 - a. **Firefox** and **Chrome** work the best
 - b. If you have only one browser, you can use an incognito/private window as the other browser
3. Download **OWASP TimeGap Theory**
 - a. Download the ZIP and extract
<https://github.com/OWASP/TimeGap-Theory/archive/master.zip>
 - b. Or type in the following in your Terminal:
`git clone https://github.com/OWASP/TimeGap-Theory.git`
4. Run Docker
5. Ensure that Docker is running
 - a. Run **docker -v** in your **Terminal** to check if **Docker** is up and running. If running, Docker will show you version information.

6. Open the command prompt/terminal
7. Change the directory to the **OWASP TimeGap Theory** directory
8. Run the following command **docker-compose up -d**



```
git clone https://github.com/OWASP/TimeGap-Theory.git
cd TimeGap-Theory
docker -v
docker-compose up -d
```

If all works as expected, you should be able to access the following URLs on your browser: <http://localhost/>



This is the homepage of your **OWASP TimeGap Theory** Lab.

The fresh installation of **OWASP TimeGap Theory** won't configure the database automatically. Go to **TimeGap Theory > WebApp > Admin** and click on the **Reset Database** button to initialize the database.

One of the common issues while setting up the Docker lab is getting an error message from CMD/Terminal - “Couldn’t connect to Docker daemon. You might need to start Docker”. This means Docker is not running. You can start Docker by launching the Docker application from the Applications/Start menu.

You can also install cURL if you want to try some advanced automation techniques for exploiting TOCTOU vulnerabilities

- If you are using Linux or Mac operating systems, cURL is already installed
- If you are using the latest version of Windows, cURL will also be installed
- If you are using an older version of Windows, you can download cURL from <https://curl.haxx.se/windows/>

Online OWASP TimeGap Theory Lab using Heroku



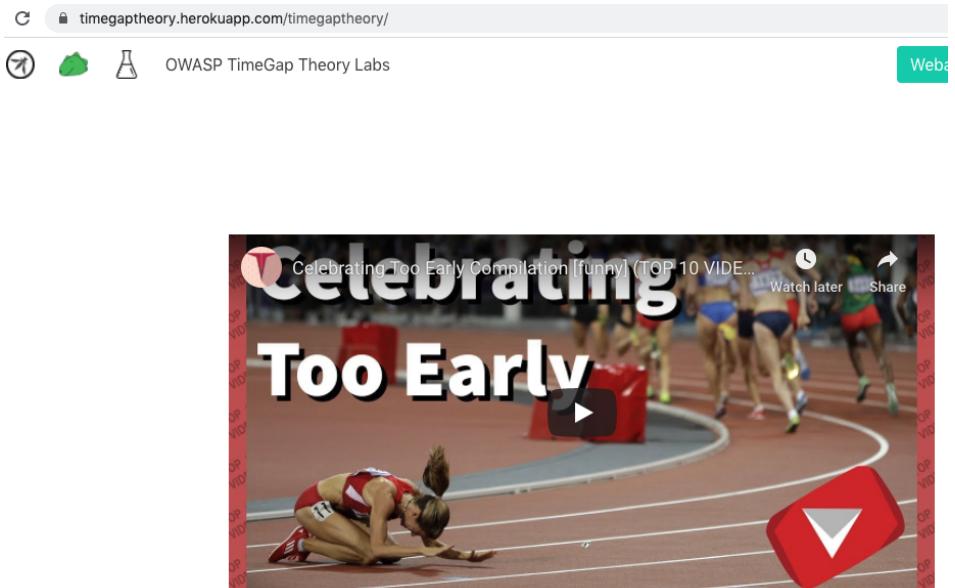
Follow the next steps if you are planning to use the online version of the **TimeGap Theory** lab.

- Approximate time required: **5 minutes**
 - Budget: **\$0, no paid software required**
1. Login to your Heroku account at <https://id.heroku.com/login>
 - a. Sign up for a free Heroku account if you do not have one already <https://signup.heroku.com>
 - b. You need to do email verification and add payment information (like credit-card) to the **Heroku** account for verification purposes.
 - c. You won't get charged if you are only running **TimeGap-Theory** on your **Heroku** account.
 2. Deploy your TimeGap Theory instance
 - a. Navigate to <https://github.com/OWASP/TimeGap-Theory>
 - b. Click on the **Deploy to Heroku** button
 - c. Choose a name for your app
 - d. Click on **Deploy app** button
 - e. Once deployed, click on **View** button

The screenshot shows the Heroku interface for creating a new app. At the top, there's a green leaf icon with the text "Deploy your own OWASP TimeGap Theory". Below it, a description reads: "A self-deployable capture-the-flag. Focusing on time-of-check to time-of-use vulnerabilities." A GitHub icon with the link "OWASP/TimeGap-Theory#master" is shown. The main form has "App name" set to "timegaptheory", which is highlighted in green with a checkmark. "timegaptheory" is also noted as available. The "Choose a region" dropdown is set to "United States". A "Add to pipeline..." button is present. Under "Add-ons", there's a ClearDB MySQL option with "Ignite" and "Free" options. A large "Deploy app" button is at the bottom.

3. Have at least two browsers installed
 - a. **Firefox** and **Chrome** work the best
 - b. If you have only one browser, you can use an incognito/private window as the other browser
4. cURL
 - a. If you are using **Linux** or **Mac** operating systems, **cURL** is already installed.
 - b. If you are using the latest version of **Windows**, **cURL** will also be installed.
 - c. If you are using an older version of **Windows**, you can download **cURL** from <https://curl.haxx.se/windows/>

And that's it. You are ready to go.



You should be able to see the homepage of your **TimeGap Theory** Lab.

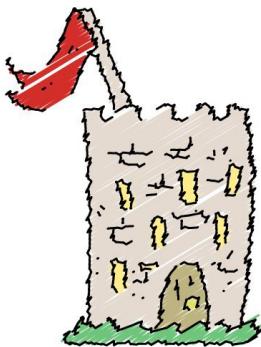
The fresh installation of OWASP TimeGap Theory won't configure the database automatically. Go to **TimeGap Theory > WebApp > Admin** and click on the **Reset Database** button to initialize the database.



CHAPTER 3

A Quick Refresher!

One more step before the adventure begins.
Let's review how the lab works.



Capture-the-Flag (CTF)

The entire lab is built like a capture-the-flag system. When you solve challenges, you earn points.

The points distribution is shown below:

Number	Challenge Name	Points
1	Sign Up	100
2	Login	100
3	Transfer	300
4	Mars	200
5	Tickets	100
6	Coupon	100
7	Ratings	100

Total = **1000 points**

You do not have to solve these challenges in any particular order. Plus, they are auto-scoring, which means no need for entering flags manually.

The OWASP TimeGap Theory application directory structure looks like:

OWASP TimeGap Theory

- Score
- Settings
- Webapp
 - o Sign Up
 - o Login
 - o Admin
 - Manage Users
 - Ticket to Mars
 - Reset Database
 - Create default users
 - o User
 - Enter coupon
 - Buy tickets
 - Transfer rewards
 - Rate the program
 - Logout

Now, let's look at some of the pages:

Settings

TimeGap Theory > Webapp > Settings

This page allows the configuration of three settings:

Main wait - how many seconds the app should wait before it writes to the database

Mars wait - how many seconds the app should wait before it writes to the database on the Ticket to Mars page

Maximum logins - how many times a user can try different passwords before their account gets locked out

The screenshot shows a web browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/settings/`. The page title is "OWASP TimeGap Theory / Settings".

Main wait
Regular database waiting period in seconds

Mars wait
Database waiting period for ticket to Mars

Maximum logins
Maximum number of login attempt before the account gets locked out

Update

Score

TimeGap Theory > Score

You can track your CTF progress here. All the challenges, points, and completion statuses are displayed on this page.

The screenshot shows a web browser window with the URL timegaptheory.herokuapp.com/timegaptheory/score/. The page title is "Score | OWASP TimeGap Theory Labs". On the right, there are three buttons: "Webapp" (green), "Settings" (yellow), and "Score" (blue). Below the title, it says "OWASP TimeGap Theory / Score". It states "You have earned 0 out of 1000." A table lists seven challenges:

Number	Title	Challenge	Points	Status
01	Sign Up	Create two users with the same email address	+ 100	To be completed
02	Login	Bruteforce login page	+ 100	To be completed
03	Transfer	Transfer more rewards than what you have	+ 300	To be completed
04	Mars	Get tickets to Mars for two people or more while only one person is eligible	+ 200	To be completed
05	Tickets	Get two tickets to the show	+ 100	To be completed
06	Coupon	Use the one-time-coupon two times or more	+ 100	To be completed
07	Ratings	Increase the love count from just one user account	+ 100	To be completed

A red "Clear" button is located at the bottom left of the table.

The clear button at the bottom of the page will clear the progress of all the levels. Use this if you would like to start the lab over again. You can also use it to keep solving the challenges over again and again.

Web App

TimeGap Theory > Webapp

This is your target. Limit all your testing activities to this directory and its subdirectories.

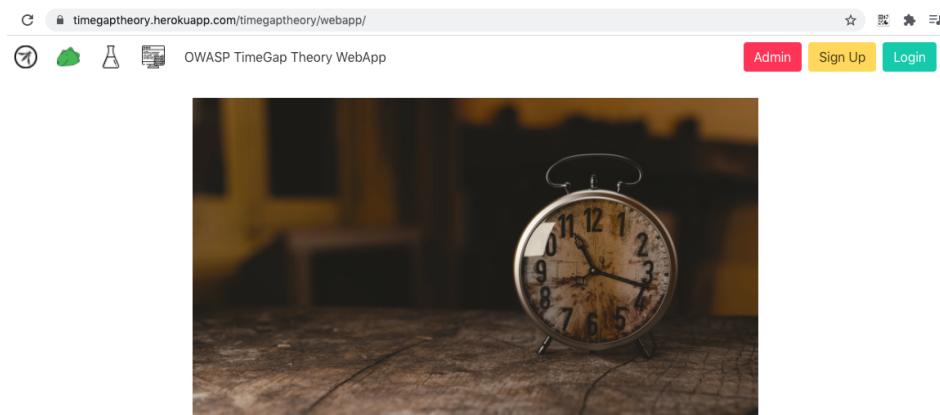


Photo by Monoar Rahman

Manage Users

TimeGap Theory > Webapp > Admin > Manage Users

Here you can view, edit, or delete user accounts you have created for TimeGap Theory WebApp.

A screenshot of a web browser showing the "Manage Users" page of the OWASP TimeGap Theory WebApp. The URL in the address bar is "localhost/timegaptheory/webapp/admin/manage-users.php". The page has a header with the same four icons as the homepage. Below the header, there is a breadcrumb navigation: "OWASP TimeGap Theory" / "WebApp" / "Administration Panel" / "Manage Users". A table lists four user accounts with columns for First Name, Email Address, Rewards, Login Attempts, Welcome Gift, and Actions. Each row includes an "Edit | Delete" link under the Actions column.

First Name	Email Address	Rewards	Login Attempts	Welcome Gift	Actions
tom	tom@timegaptheory.com	100	0	0	Edit Delete
jerry	jerry@timegaptheory.com	900	0	0	Edit Delete
spike	spike@timegaptheory.com	1000	0	0	Edit Delete
tyke	tyke@timegaptheory.com	2000	0	0	Edit Delete

Ticket to Mars

TimeGap Theory > Webapp > Admin > Ticket to Mars

This is the script an admin uses to start the **Ticket to Mars** program. Once running, the script will go through each user. If the user has at least two-thousand reward points, the script will give them a ticket to Mars. Otherwise, the app will move on to the next user.

The screenshot shows a browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/admin/ticket-to-mars.php`. The page displays four user profiles with their email addresses and status: 'tom@timegaptheory.com is NOT eligible.', 'jerry@timegaptheory.com is NOT eligible.', 'spike@timegaptheory.com is NOT eligible.', and 'tyke@timegaptheory.com is eligible. Notification email sent to the user.' Below the profiles, there are icons for a user profile, a green leaf, a flask, and a document.

[OWASP TimeGap Theory](#) / [WebApp](#) / [Administration Panel](#) / [Ticket to Mars](#)

tom@timegaptheory.com is NOT eligible.

jerry@timegaptheory.com is NOT eligible.

spike@timegaptheory.com is NOT eligible.

tyke@timegaptheory.com is eligible. Notification email sent to the user.

Reset Database

TimeGap Theory > Webapp > Admin > Reset database

In case needed, you can reset the database. Doing so will not affect your current score.

The screenshot shows a browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/admin/reset-database.php`. The page displays a single message: 'Reset completed.' Below the message, there are icons for a user profile, a green leaf, a flask, and a document.

[OWASP TimeGap Theory](#) / [WebApp](#) / [Administration Panel](#) / [Reset Database](#)

Reset completed.

Create Default Users

TimeGap Theory > Webapp > Admin > Create default users

Use this to quickly load the web app with 4 default users: Tom, Jerry, Spike, and Tyke. This saves time from manually creating test accounts through the **Sign Up** process.

→ C  timegaptheory.herokuapp.com/timegaptheory/webapp/admin/create-users.php



OWASP TimeGap Theory WebApp

[OWASP TimeGap Theory](#) / [WebApp](#) / [Administration Panel](#) / [Create users](#)

Cleared the database.

Created users.

Login

TimeGap Theory > Webapp > Login

Users can login here. By default, there are no user accounts available on **TimeGap Theory**. Users must go through **Sign Up** flow and then login. You can also create users via the **Create Default Users** option mentioned above.

→ C timegaptheory.herokuapp.com/timegaptheory/webapp/login.php



OWASP TimeGap Theory WebApp

[OWASP TimeGap Theory](#) / [WebApp](#) / Login

Tom

Jerry

Spike

Tyke

Email

you@timegaptheory.com

Password

Password

Submit

Sign Up

TimeGap Theory > Webapp > Sign up

Users can sign up for an account here. You need to supply a full name, email address, and a password to create an account.

The screenshot shows a web browser displaying the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php`. The page title is "OWASP TimeGap Theory WebApp". Below the title, there are four tabs: "Tom", "Jerry", "Spike", and "Tyke". The "Tom" tab is active. The main form has three input fields: "Name" containing "John Doe", "Password" containing "Password", and "Email" containing "you@timegaptheory.com". A blue "Submit" button is at the bottom left of the form.

→ C timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php

OWASP TimeGap Theory WebApp

Tom Jerry Spike Tyke

Name

John Doe

Password

Email

you@timegaptheory.com

Submit

Enter Coupon

TimeGap Theory > Webapp > User > Enter coupon

The application is expecting the following coupon code - **WELCOME10**. This is a one-time-use coupon. Once entered, it will give the user **200** extra reward points.

The screenshot shows a web browser window with the URL timegaptheory.herokuapp.com/timegaptheory/webapp/user/enter-coupon.php. At the top right, there is a yellow button with the text "You need to complete KYC". Below the address bar, there are four icons: a key, a green leaf, a flask, and a computer monitor. To the right of these icons, the text "OWASP TimeGap Theory WebApp" is displayed. The main content area has a breadcrumb navigation: "OWASP TimeGap Theory" / "WebApp" / "User" / "Enter coupon". Below the breadcrumb, there are two buttons: "Valid coupon" and "Invalid coupon". A message "You have **100** reward point(s)." is displayed. A form titled "Enter coupon code" contains the value "1B34SDX" in an input field, with a "Submit" button below it.

Buy Tickets

TimeGap Theory > Webapp > User > Buy Tickets

Users can buy tickets to the daily show here. The number of show tickets is limited.

→ ⌛ 🔒 timegaptheory.herokuapp.com/timegaptheory/webapp/user/buy-tickets.php

You have no tickets!

⌚ 🎟️ 💡 🖥️ OWASP TimeGap Theory WebApp

OWASP TimeGap Theory / WebApp / User
/ Buy Tickets



Photo by Kaboompics .com from Pexels

Buy One Ticket

Transfer Rewards

TimeGap Theory > Webapp > User > Transfer Rewards

Users can transfer rewards from their accounts to other accounts here.

There are two rules for each transfer rewards operations:

1. From and To accounts cannot be the same. Duh!
2. The balance on the account should be more than or equal to the rewards being transferred.

The screenshot shows a web application interface for transferring rewards. At the top, there is a header with icons for a user profile, a green leaf, a flask, and a document, followed by the text "OWASP TimeGap Theory WebApp". Below the header, a navigation bar shows the path: "OWASP TimeGap Theory / WebApp / User / Transfer Rewards". The main area contains two buttons: "\$10 from Tom to Jerry" and "\$90 from Tom to Jerry". Below these buttons are input fields for "From" and "To" accounts, both containing the email "tom@timegaptheory.com rewards(100)". There is also an "Amount" field with the value "100" and a "Transfer" button at the bottom.

→ C timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php

OWASP TimeGap Theory WebApp

OWASP TimeGap Theory / WebApp / User / Transfer Rewards

\$10 from Tom to Jerry \$90 from Tom to Jerry

From:
tom@timegaptheory.com rewards(100)

To:
tom@timegaptheory.com rewards(100)

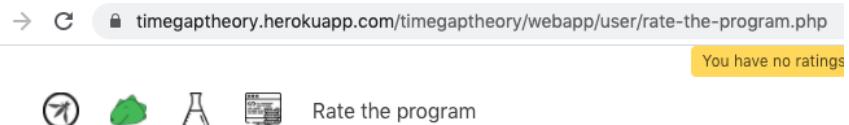
Amount:
100

Transfer

Rate the Program

TimeGap Theory > Webapp > User > Rate the program

You can rate the show here by clicking on the love button. Only one rating per user is allowed. If you click the love button once again, the app will remove your existing rating.



[OWASP TimeGap Theory](#) / [WebApp](#) / [User](#)
/ Rate the program

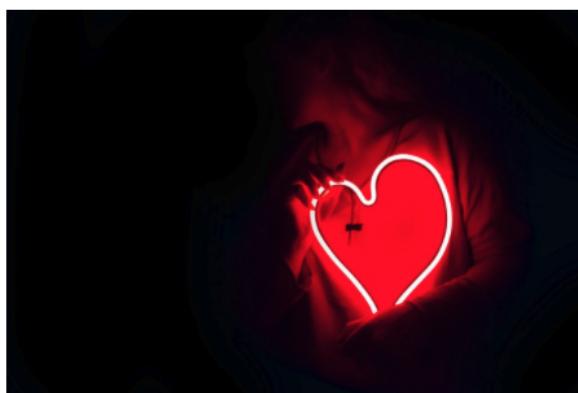


Photo by Designecologist from Pexels

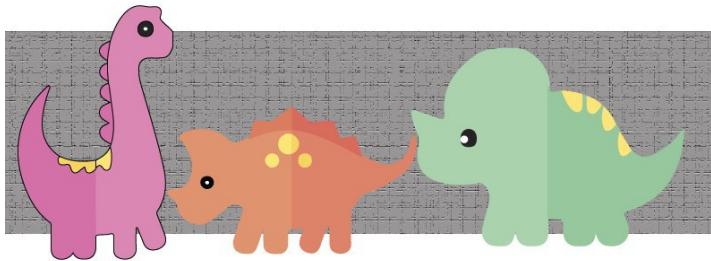
Love



CHAPTER 4

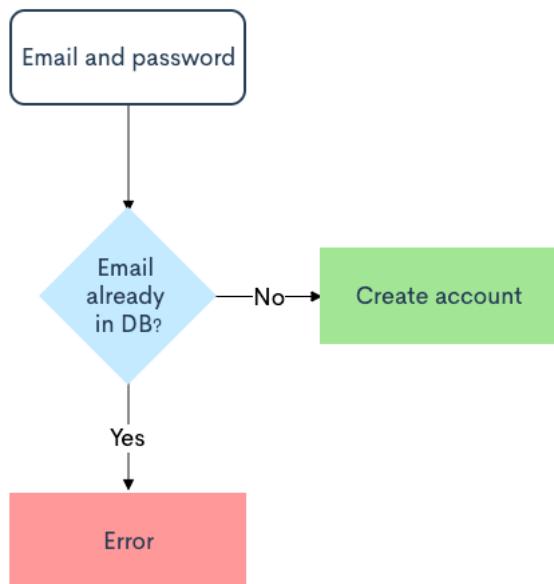
Challenge 1 – Sign Up

Sign up for an account using this page.
Only one account per email ID is allowed.

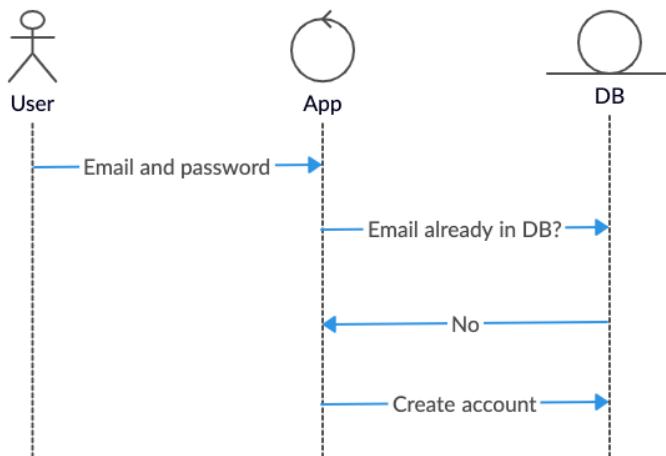


The sign up pages are pretty common in web apps. The basic functionality of these pages is simple.

- Users enter their email address and some other details
- If the email address is not already in the database, the app will create a new account
- If the email address already exists in the database, the app will show an error



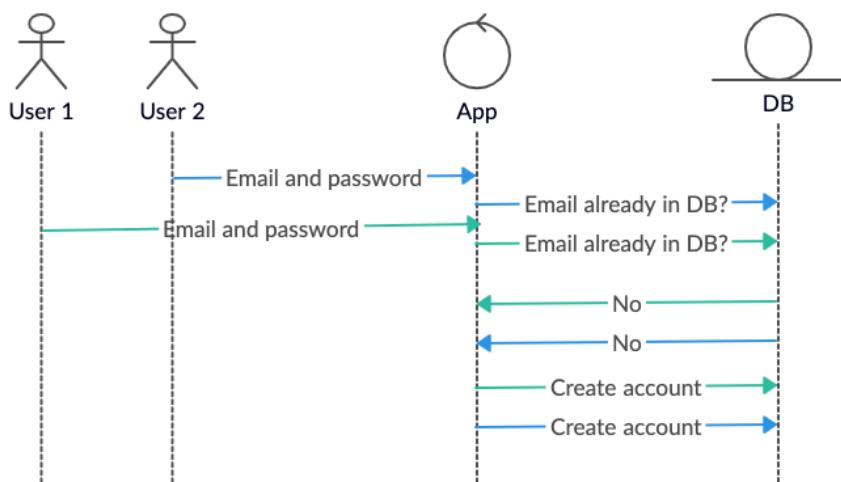
Let us visualize the happy path with the help of a sequence diagram:



That is the happy path for one user creating an account.

Next, let us reimagine the same scenario, but with two changes:

1. Two users are trying to create user accounts at the same time
2. Both users are trying to sign up with the same email address



The steps at a high level are:

1. Users fill out the **Sign up** form
2. Both users click on **Sign Up** at the same time
3. The app will create two threads
4. The threads will each check if the email exists in the database
5. Both threads will get an **email does not exist** response from the database
6. The app will proceed with account creation
7. Two accounts will be created by the app
8. Both accounts will have the same email address

Well, this looks good in theory, but is it practical? Let us find out.

Before getting started with following steps, navigate to **Admin > Reset Database** page (`webapp/admin/reset-database.php`)

First, let's go through a regular flow:

1. Open your browser
2. Navigate to the **Sign Up** page
3. Click on the first user button - this will fill the form with details of the user **Tom**
4. Click on the **Submit** button

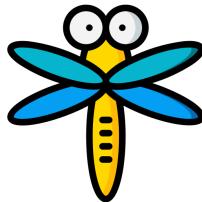
The screenshot shows a web browser displaying the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/login.php`. The page title is "OWASP TimeGap Theory WebApp". Below the title, there are four user selection buttons: "Tom", "Jerry", "Spike", and "Tyke". The "Tom" button is selected, and his details are populated into the form fields: "Email" (tom@timegaptheory.com) and "Password" (three asterisks). A green "Submit" button is at the bottom.

We have created a user account on the system now. We need to see if the app allows us to sign up for another account with the same email. Let us test that by repeating the same steps.

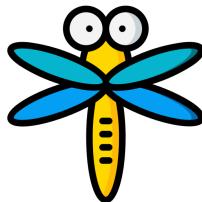
The screenshot shows a web browser displaying the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php`. The page displays the error message "account email address exists".

Uh oh! The application does not permit this account creation. However, some time has passed since you created the first account.

First request:



Second request:



We need a way to speed up the sign-up process. We want both the requests to happen more or less at the same time. But how can we do that?

Browser dev tools can be used to send multiple parallel requests to the application in a single go. We can explore that path later. For now, we need another method.

Instead of speeding up the process on our end, we can slow down the app. In real life, attackers achieve this in two ways:

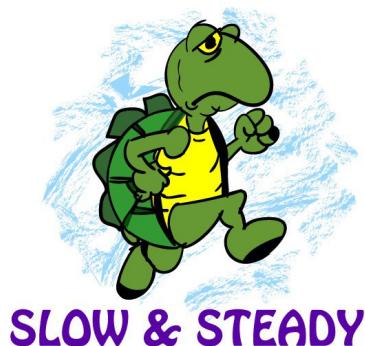
1. By launching distributed denial of service attacks against their target
OR
2. By launching their attacks during peak traffic periods

Thankfully, we do not have to do that. TimeGap theory will do that for us.

1. On your browser, navigate to the **Settings** page of **OWASP TimeGap Theory**
2. Change the **Main wait** to 5 seconds
3. Click on the **Update** button

The screenshot shows a web browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/settings/`. The page title is "OWASP TimeGap Theory / Settings". There is a section titled "Main wait" with the sub-instruction "Regular database waiting period in seconds". A text input field contains the value "5".

This will make **TimeGap Theory** wait **5** seconds before every modification operation. If you try to perform anything that requires writing to the database, you will see that it's loading slowly.



5 seconds is more than enough time for us to try out the attack scenario. We need to fire two sign-up flows within 5 seconds time and that requires some preparation.

Preparation Phase

1. Open two browsers side by side (Use private/incognito window if you do not have two browsers)
2. On both the browsers:
 - a. Navigate to the **Sign up** page
 - b. Click on the second user button this time - this will fill the form with details of the user **Jerry**



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation Phase

1. Click on the **Submit** button on the first browser
2. In the second browser, click the **Submit** button within the five seconds window



Post-exploitation phase

1. Wait for both the browsers to complete the request
2. You should see both browsers complete the requests without any errors
3. On one of the browsers, navigate to the **Manage Users** page
4. Boom! You should have two user accounts with the same email address
5. If you check your score, you should also see you now have **100** points
6. Before moving on, revert the delay back to **0** on the **Settings** page



Yes, we hear you. This attack technique is not realistic. There is no way you can ask web apps to act slowly in the real world. Launching a distributed denial of service attack is also not reliable. So, what can you do in the real world? If you can't slow down the target, you need to speed up. No, we do not mean clicking on multiple browsers at the same time. You need to automate this part.

Browser Dev Tools

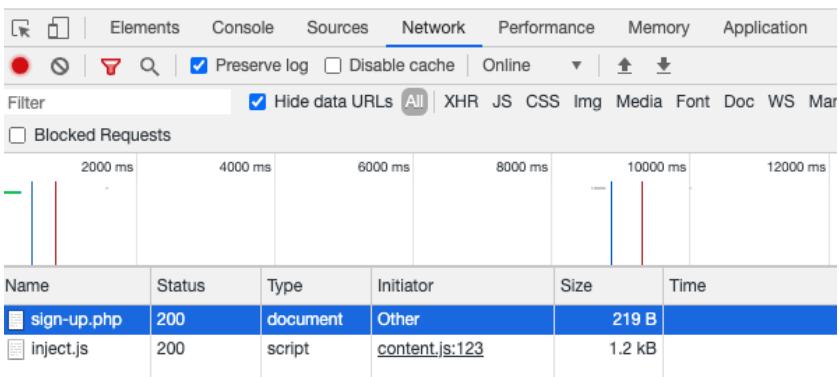
We love browser dev tools. They are useful for performing some quick security tests. These quick tests include TOC/TOU as well.⁴

First, we need a valid fetch request:

1. Open your browser ([Chrome](#) or [Firefox](#))
2. Navigate to the [Sign up](#) page
3. Click on any of the user button
4. Open up dev tools by pressing [F12](#) on the browser
 - a. On Windows, you can use [Ctrl + Shift + I](#)
 - b. On Mac, you can use [Cmd + Shift + I](#)
5. On the [Sign up](#) page, click on the [Sign Up](#) button
6. On the browser dev tools, click on the [Network](#) tab
7. Right-click on the [sign-up.php](#) request
8. Click on [Copy > Copy as fetch](#)

← → ⌂ 🔒 timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php

account email address exists



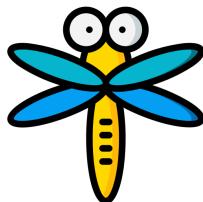
Paste this in the **Console** tab of the browser dev tools. It should look like the following:

```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php", {
  "headers": {
    "content-type": "application/x-www-form-urlencoded"
  },
  "body": "firstname=tom&password=tom&email=tom%40timegaptheory.com",
  "method": "POST"
});
```

You may see a slightly bigger fetch request. This is because of the additional headers and request parameters added by the browser. These are optional. The above example shows the bare minimum fetch request.

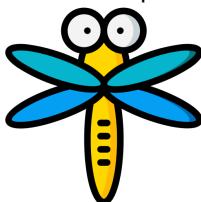
But that's just one request.

The only request:



We need two parallel fetch requests.

First request:



Second request:



This can be done by copying and pasting the fetch request without pressing the **Enter** key.

A screenshot of the Chrome DevTools interface, specifically the Network tab. At the top, there are tabs for Elements, Console, Sources, Network, Performance, Memory, and Application. The Network tab is active. Below the tabs, there is a search bar with the placeholder "top" and a "Filter" button. To the right of the search bar is a dropdown menu set to "Default levels". The main area shows a list of network requests. The first request is highlighted with a red border and shows the following code:

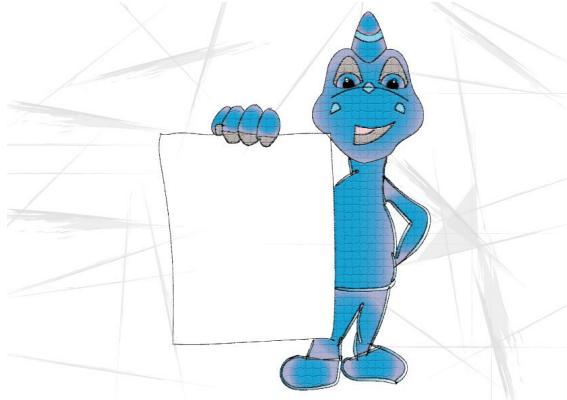
```
> fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "firstname=tom&password=tom&email=tom%40timegaptheory.com",  
  "method": "POST"  
});
```

The second request is shown below it with a similar structure:

```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/sign-up.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "firstname=tom&password=tom&email=tom%40timegaptheory.com",  
  "method": "POST"  
});
```

Before running this new command, we need to clear the slate. It involves three simple steps:

1. Ensure that there is no delay
 - a. Go to **TimeGap Theory > Settings**
 - b. Check that **Main wait** is set to **0**
2. Delete any existing **Tom** user
 - a. Go to **TimeGap Theory > Admin > Manage Users**
 - b. Delete the user **Tom**
3. Clear your current score
 - a. Go to **TimeGap Theory > Score**
 - b. Click on the **Clear** button



Alright, our slate is clear. Let us execute the attack now:

1. Enter the combined fetch requests on the **Console** tab of browser dev tools
2. Press the **Enter** key
3. Go to **TimeGap Theory > Admin > Manage Users**
4. See if two users with the same email address are created
5. Go to **TimeGap Theory > Score**
6. Check if you got points for completing the **Sign Up** challenge



First Name	Email Address	Rewards	Login Attempts
tom	tom@timegaptheory.com	100	0
tom	tom@timegaptheory.com	100	0

What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

- A legitimate user may try to register for their account in the future
- They may also get several emails asking them to confirm the account creation
- If a legitimate user confirms the account, the attacker may get a permanent backdoor to the user account
- Attackers would be able to carry out any actions on the app with all the audit trails pointing out to the legitimate user



Let's summarize what we completed:

1. The **Sign up** page allows only one user account per email address
2. You ensured the business logic is working under normal scenarios
3. You slowed down the system and bypassed the business logic
4. You bypassed the business logic by using a tool that makes concurrent requests
5. Now you know how **TOC/TOU** security issues work and how to exploit them



CHAPTER 5

Challenge 2 – Login

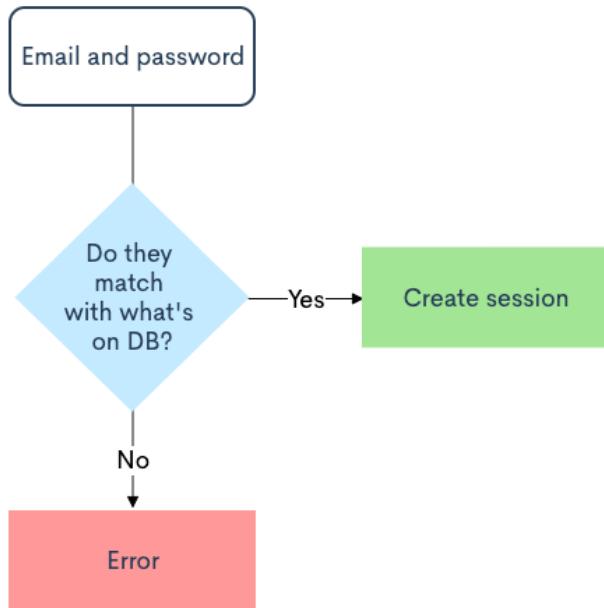
The starting point for all users.

Rate limited to avoid brute force attacks.



Login pages are the user's starting point in many web apps. They collect username/email and password and create user sessions if they are valid.

Let us look at a sample login page flow:



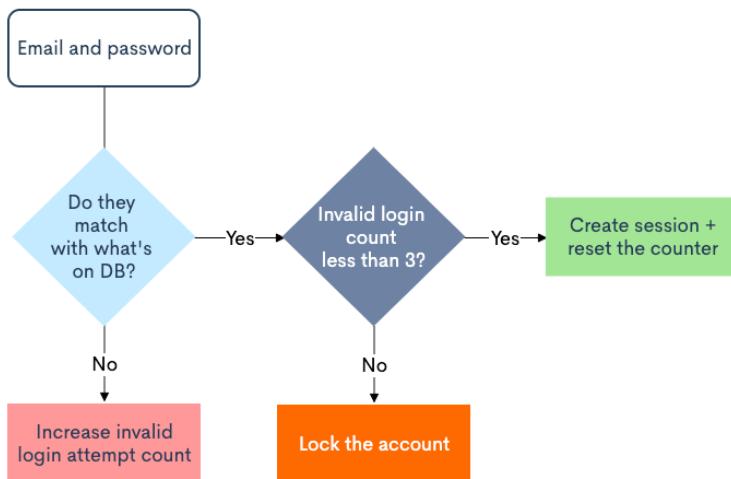
There is an obvious flaw here. Let us review it:

1. Tom is an attacker
2. Tom wants to break into Jerry's account
3. Tom tries jerry@abcd.com, jerry@efgh.com, etc
4. Tom gets account does not exist error
5. Tom tries jerry@example.com
6. Tom gets invalid password error. This means there is a user account with jerry@example.com as user name
7. Tom tries logging into the portal with jerry@example.com and 1,2,3,a,b,c etc. as the password
8. Tom gets invalid password error for all of them
9. Then Tom tries jerry as the password and Boom! Tom just logged in to the portal as Jerry



The above scenario is a major headache for all login pages. These types of attacks are known as dictionary or brute force attacks. There are multiple ways to protect login pages from brute force attacks - captchas, delays, account lockouts, generic error messages etc.

What happens if we revise the current login page design with account lockouts. If the user is not able to login successfully within a few tries, the portal will lock the user's account.



As you can see, the account lockout is achieved by storing the number of failed login attempts on the server side. For every failed login attempt, the app will increment the counter. If the value exceeds a certain point, for example 3, the portal will lock the account.

The high-level steps are:

1. User enters the correct username but the wrong password.
2. Portal increases the failed login attempt count to 1.
3. User enters the wrong password again.
4. Portal increases the failed login attempt count to 2.
5. User enters the wrong password one more time.
6. Failed login attempt count reaches 3.
7. The portal locks out the user out of their account.

Sounds good? Let us verify this in our lab:

1. Open your browser
2. Navigate to the **login** page
3. Enter tom@example.com as the username
4. Enter **abcd** as the password
5. Click on the **Login** button a few times
6. Initially, you will get an error message that goes **Login failed**
7. After a few tries, you will see a different error message - **Your account is locked out**
8. You have been locked out of the user account
9. Now, change the password from **abcd** to **tom**
10. Click on the **login** button again
11. The system will not allow you to login even with the correct password because the account is locked

What if we perform all these login attempts in a single try before the server updates the failed login attempt counter on the database? Let us check that theory.

By default, **TimeGap Theory** will lock the user account on the third wrong attempt. In order to successfully test our attack, we would need more than three browser windows. That kind of spoils the fun, doesn't it? Let us reduce the **maximum failed attempt** to **1**. This way, we just need two browser windows to test.

1. On your browser, navigate to the **Settings** page
2. Change the **login attempt count** to 1
3. Click on the **Save** button
4. Stay on the **Settings** page, we need to do one more thing

The screenshot shows a web application interface for 'timegaptheory.herokuapp.com/timegaptheory/settings/'. The page has a header with a lock icon and the URL. It contains three main configuration sections:

- Main wait**: A field labeled "Regular database waiting period in seconds" with a value of "5" in a text input field.
- Mars wait**: A field labeled "Database waiting period for ticket to Mars" with a value of "0" in a text input field.
- Maximum logins**: A field labeled "Maximum number of login attempt before the account gets locked out" with a value of "1" in a text input field.

An "Update" button is located at the bottom of the form.

Time to slow down TimeGap Theory:

1. Change the **Main wait** to 5 seconds
2. Click on the **Save** button

Now, TimeGap Theory will wait 5 seconds before every database write operation. This write operation includes **failed login attempts** values.

Preparation Phase

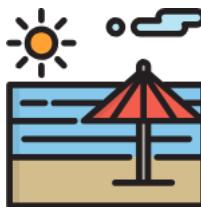
1. Open two browsers side by side (Use a private/incognito window if you do not have two browsers).
2. On both the browsers:
 - a. Navigate to the **Login** page
 - b. Click on the second user button this time - this will fill the form with details of the user **Jerry**



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation Phase

1. Click on the **Login** button on the first browser
2. In the second browser quickly also click on the **Login** button



Post-exploitation Phase

1. Wait for both the browsers to complete the request
2. You should see the login attempt was successful on the second browser
3. Check the user login attempt count is greater than **1**. Phew
4. If you check your score, you should see you have **100** points
5. Before moving on, revert the delay to **0** on the **settings** page



Automation Time

There is no way we can perform a brute force attack like this in real life. You cannot call the webmaster and ask them to delay the server. Browser **Dev Tools** to the rescue again.

First, we need a valid fetch request:

1. Open your browser (**Chrome** or **Firefox**)
2. Navigate to the **login** page
3. Click on any of the user buttons
4. Open up dev tools by pressing **F12** on the browser
 - a. On **Windows**, you can use **Ctrl + Shift + I**
 - b. On **Mac**, you can use **Cmd + Shift + I**
5. On the **login** page, click on the **Submit** button
6. On the browser dev tools, click on the **Network** tab
7. Right click on the **login.php** request
8. Click on **Copy > Copy as fetch**

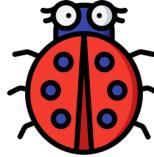
The screenshot shows the Network tab in the Chrome DevTools developer console. A request for 'login.php' is selected. A context menu is open, with 'Copy' being the active option. A submenu is displayed with several options: 'Copy link address', 'Copy request headers', 'Copy response headers', 'Copy response', and 'Copy as fetch' (which is highlighted in blue).

Paste this in the **Console** tab of the browser dev tools. It should look like the following:

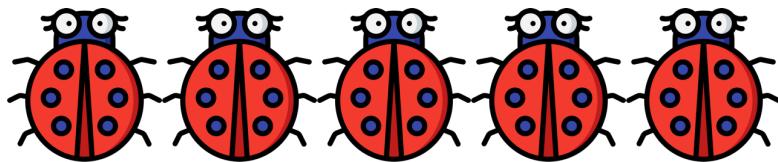
```
> fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/login.php", {
  "headers": {
    "content-type": "application/x-www-form-urlencoded"
  },
  "body": "email=tom%40timegaptheory.com&password=tom&submit=Submit",
  "method": "POST",
  "mode": "cors"
});
```

It is okay if you see a request with some more elements. We trimmed down the unwanted portions to make it look clean.

But that's just one request.



We are talking about brute forcing here. We need a minimum of 5 for this simulation.



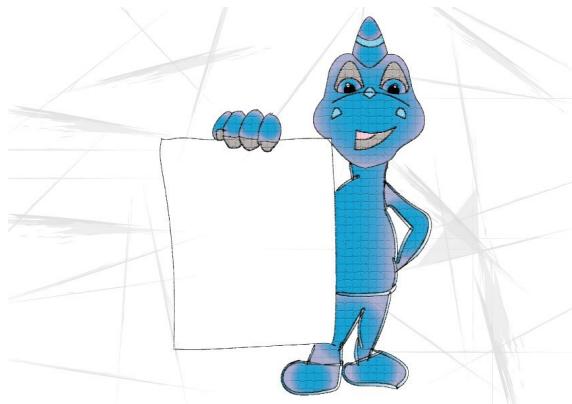
Copy and paste the fetch request without pressing the **Enter** key. However, it is a good practice to separate each request with a comma. Let us do that.

```
Elements Console Sources Network Performance Memory Application  
Default levels ▾  
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/login.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "email=tom%40timegaptheory.com&password=tom1&submit=Submit",  
  "method": "POST",  
  "mode": "cors"  
}),fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/login.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "email=tom%40timegaptheory.com&password=tom3&submit=Submit",  
  "method": "POST",  
  "mode": "cors"  
}),fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/login.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "email=tom%40timegaptheory.com&password=tom2&submit=Submit",  
  "method": "POST",  
  "mode": "cors"  
}),fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/login.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "email=tom%40timegaptheory.com&password=tom&submit=Submit",  
  "method": "POST",  
  "mode": "cors"  
});
```

We also changed the password value in each request. The first four requests have the wrong password. The last request has the correct password.

Before running this new command, we need to clear the slate. It involves three simple steps:

1. Ensure that there is no delay
 - a. Go to **TimeGap Theory > Settings**
 - b. Set the **Main wait to 0**
2. Refresh the user accounts.
 - a. Go to **TimeGap Theory > Admin**
 - b. Click on **Create Default Users**
3. Clear your current score.
 - a. Go to **TimeGap Theory > Score**
 - b. Click on the **Clear** button



Alright, our slate is clear. Let us execute the attack now:

1. Enter the combined fetch requests on the **Console** tab of browser dev tools
2. Press the **Enter** key
3. Go to the **Network** tab of the browser
4. See if you can find an **index.php** file
5. Right click on it and select **open in a new tab**
6. You are logged in
7. If you want to see which password was correct, you can check which request showed true in the **Console** tab
8. Go to **TimeGap Theory > Score**
9. You should have got **100** points for completing the **Login** challenge

Name	Status	Type	Initiator
bulmadark.css	304	stylesheet	login.php
owasp.png	304	text/plain	login.php
timegaptheor...	304	text/plain	login.php
lab.png	304	text/plain	login.php
webapp.png	304	text/plain	login.php
inject.js	200	script	content.js:123
login.php	200	fetch	VM1922:1
login.php	200	fetch	VM1922:8
login.php	200	fetch	VM1922:15
login.php	302	text/html	VM1922:22
index.php	200	fetch	login.php

What is the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. A legitimate user may keep getting their account compromised despite changing their passwords
2. Attackers may compromise the admin functionalities of web apps



Let us summarize what we completed:

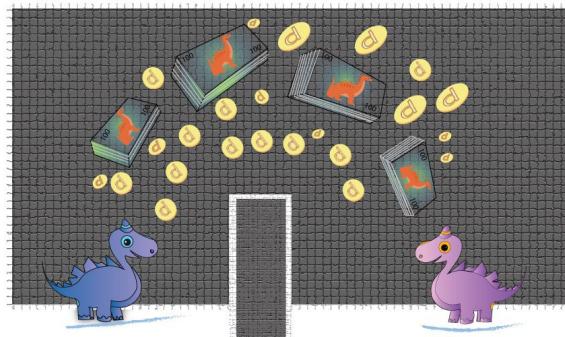
1. The login page rate limits the login attempts by locking the account
2. You analyzed this behavior by locking out one account
3. You slowed down the system and bypassed the business logic
4. You bypassed the business logic by using browser dev tools
5. Now you know how TOC/TOU security issues can affect login pages



CHAPTER 6

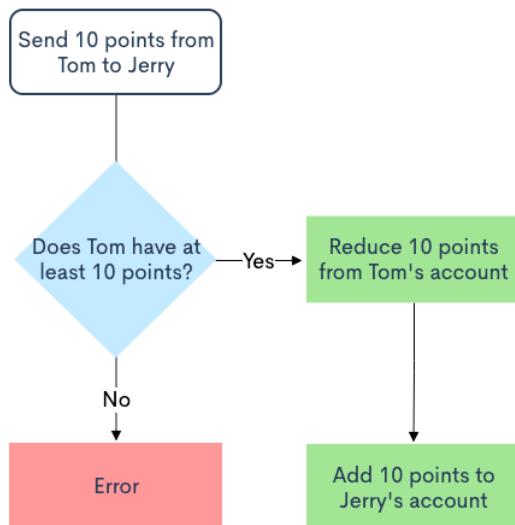
Challenge 3 – Transfer

Transfer reward points between users.



Transferring rewards is not something we do every day. But sending money is. Think of the transfer rewards as sending money. The basic functionality is identical.

Let us look at a sample transfer rewards flow:



There are three conditions to be met for the transaction to be successful:

1. Transaction amount should be a positive number
2. Payer and payee can not be same
3. Payers should have enough balance in their account

Let us see if the application is performing those checks. Go to the [Sign up](#) page and create two user accounts - **Tom** and **Jerry**

Transaction amount should be a positive number:

1. Go to the **Transfer** page
2. Put in a negative number in the amount field
3. Click on the **Transfer** button

The screenshot shows a web browser displaying the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php`. The page title is "OWASP TimeGap Theory WebApp". Below the title, there are four icons: a person, a cloud, a lab flask, and a computer monitor. The main content area shows a transfer history with two entries: "\$10 from Tom to Jerry" and "\$90 from Tom to Jerry". A "From:" field contains "tom@timegaptheory.com rewards(100)". A "To:" field contains "tom@timegaptheory.com rewards(100)". An "Amount:" field contains "-100". A red "Transfer" button is visible. A validation message box appears over the amount field, stating "Value must be greater than or equal to 10." with an exclamation mark icon.

Payer and payee can not be same:

1. Go to the **Transfer** page
2. Select the same user as payer and payee
3. Put in a **10** as the amount
4. Click on the **Transfer** button

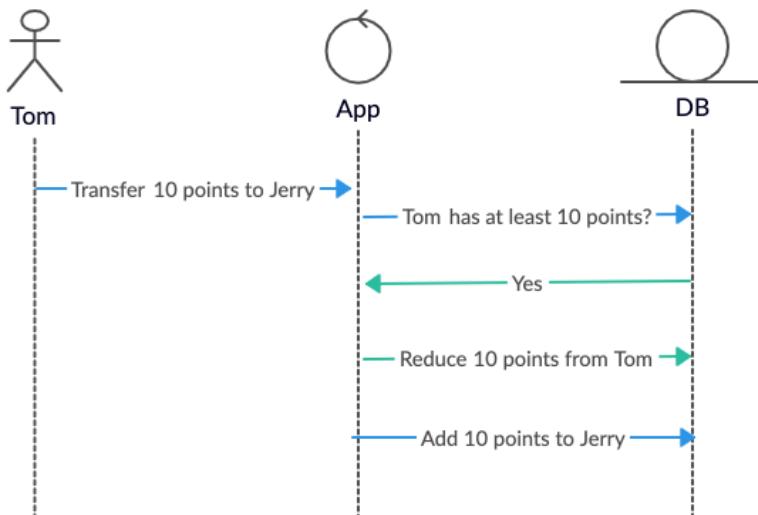
The screenshot shows a web browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php`. A red horizontal bar at the top contains the text "From and to can not be same". Below the bar, the page title is "OWASP TimeGap Theory / WebApp / User / Transfer Rewards". The transfer form fields are visible: "From:" input containing "tom@timegaptheory.com rewards(100)", "To:" input containing "tom@timegaptheory.com rewards(100)", "Amount:" input containing "100", and a green "Transfer" button.

Payer should have enough balance in their account:

1. Go to the **Transfer** page
2. Select a payer and payee
3. Put in a **300** as the amount
4. Click on the **Transfer** button

The screenshot shows a web browser window with the URL `timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php`. A red banner at the top displays the message "Not enough rewards". Below the banner, the page title is "OWASP TimeGap Theory WebApp". Underneath the title, the navigation path is shown as "OWASP TimeGap Theory / WebApp / User / Transfer Rewards". At the top right, there are four icons: a user profile, a green leaf, a lightbulb, and a computer monitor. Below the navigation, there are two buttons: "\$10 from Tom to Jerry" and "\$90 from Tom to Jerry". A "From:" field contains "tom@timegaptheory.com rewards(100)". A "To:" field contains "tom@timegaptheory.com rewards(100)". An "Amount:" field contains "100". A large green "Transfer" button is at the bottom.

Let us visualize the happy path with the help of a sequence diagram:



Our aim is to bypass the business logic and send more reward amounts than we have. How can we do that? We need to make multiple transactions before the application deducts it from the balance.

It's time to slow down **TimeGap Theory**:

1. On your browser, navigate to the **Settings** page
2. Change the time delay to **5** seconds
3. Click on the **Save** button

Now, **TimeGap Theory** will wait 5 seconds before every database write operation.

Preparation phase

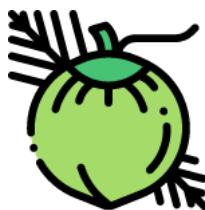
1. Open two browsers side by side (Use private/incognito window if you do not have two browsers)
2. On both the browsers:
 - a. Navigate to the **Login** page
 - b. Click on the first user button - this will fill the form with details of the user **Tom**
 - c. Navigate to the **Transfer rewards** page
 - d. If the account has **100** points as balance, put a lesser amount as the transfer value on both the browsers. Say, **90**
 - e. Select **Tom** as the payer
 - f. Select **Jerry** as the payee
 - g. Note down the balance on **Jerry**'s account (which is displayed in simple bracket)



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation phase

1. Click on the **Transfer** button on the first browser
2. Go to the second browser as fast as you can
3. Click on the **Transfer** button on the second browser



There comes our third and final phase:

Post-exploitation phase

1. Let us wait for both the browsers to complete the request
2. You can see that the transfer operations were successful on both the browsers
3. Check the balance of user **Jerry**
4. Phew. You transferred more points from **Tom**'s account that what they had
5. If you check your scores, you will see that you have got **100** points for transfer challenge
6. Don't forget to change the delay to back to **0** on the **Settings** page



Automation time

Let's try and automate this.

First, we need a valid fetch request:

1. Open your browser ([Chrome](#) or [Firefox](#))
2. Navigate to [TimeGap Theory](#) > [Login](#)
3. Click on the first user button. This will fill the user details for user [Tom](#)
4. Click on the [login](#) button
5. Navigate to [TimeGap Theory](#) > [Webapp](#) > [User](#) > [Transfer rewards](#)
6. Click on the first [\\$10 from Tom to Jerry](#) button. This will fill the form with some transfer details.
7. Open up dev tools by pressing [F12](#) on the browser
 - a. On [Windows](#), you can use [Ctrl + Shift + I](#)
 - b. On [Mac](#), you can use [Cmd + Shift + I](#)
8. Click on the [Transfer](#) button
9. On the browser dev tools, click on the [Network](#) tab
10. Right click on the [transfer-rewards.php](#) request
11. Click on [Copy](#) > [Copy as fetch](#)

The screenshot shows a browser window for the OWASP TimeGap Theory WebApp. The URL is `timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php`. The page title is "OWASP TimeGap Theory WebApp". Below the title, the navigation path is "OWASP TimeGap Theory / WebApp / User / Transfer Rewards". The main content area shows a list of network requests under the "Network" tab of the developer tools. The requests are listed in a table with columns: Name, Status, Type, and Initiator. Two requests are visible: "transfer-rewards.php" (Status 200, Type document, Initiator Other) and "owasp.png" (Status 200, Type png, Initiator transfer-rewards.php). The developer tools interface includes a timeline at the top showing request times and a detailed list of requests below.

Name	Status	Type	Initiator
transfer-rewards.php	200	document	Other
owasp.png	200	png	transfer-rewards.php

Paste that on the **Console** tab of the browser dev tools. You will get something like the following:



```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "from=tom%40timegaptheory.com&to=jerry%40timegaptheory.com&amount=90&submit=Transfer",  
  "method": "POST"  
});|
```

Again, don't worry if you see a slightly bigger request. That is just your browser being naughty. You can use that request as it is, or use the one shown above.

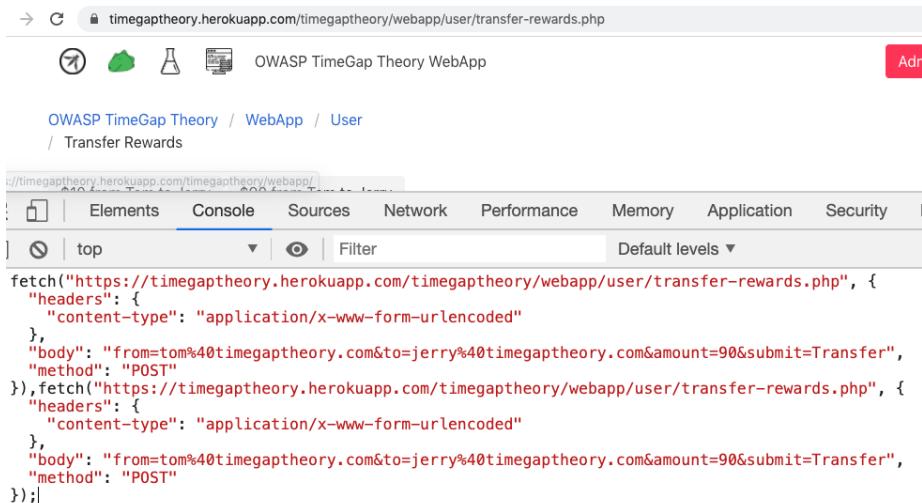
Now we have a request.



And we need two requests.



It can be done by copy-pasting the fetch request without pressing the **Enter** key. However, it's a good practice to separate each request with a comma. Let us do that

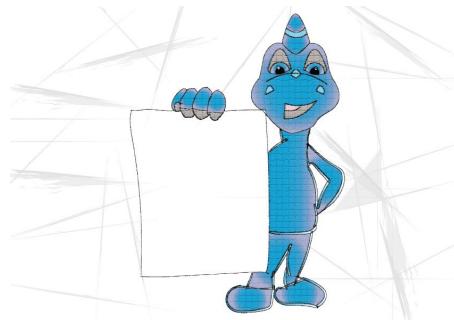


The screenshot shows a browser window for the OWASP TimeGap Theory WebApp. The URL is `https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php`. The page title is "OWASP TimeGap Theory / WebApp / User / Transfer Rewards". Below the title, there is a network monitoring interface with tabs for Elements, Console, Sources, Network, Performance, Memory, Application, Security, and I. The Network tab is selected. A single POST request is listed, showing the URL and its body. The body contains a JavaScript fetch call with two requests. The first request is to `https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php` with headers and a body containing a transfer from "tom" to "jerry" for 90 units. The second request is to the same URL with a different body, also from "tom" to "jerry" for 90 units. The entire code block is as follows:

```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php", {
  "headers": {
    "content-type": "application/x-www-form-urlencoded"
  },
  "body": "from=tom%40timegaptheory.com&to=jerry%40timegaptheory.com&amount=90&submit=Transfer",
  "method": "POST"
}),fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/transfer-rewards.php", {
  "headers": {
    "content-type": "application/x-www-form-urlencoded"
  },
  "body": "from=tom%40timegaptheory.com&to=jerry%40timegaptheory.com&amount=90&submit=Transfer",
  "method": "POST"
});|
```

Before running this new command, we need to clear the slate. It involves two simple steps:

1. Ensure that there is no delay
 - a. Go to **TimeGap Theory > Settings**
 - b. Ensure that the **Main wait** is set to **0**
2. Clear the current score
 - a. Go to **TimeGap Theory > Score**
 - b. Click on the **Clear** button



Alright, our slate is clear. Let us execute the attack now:

1. Enter the combined fetch requests on the **Console** tab of browser dev tools
2. Click on the **To** field to see the rewards on each account
3. Now you see a negative value in **Tom's** account
4. You also see corresponding change in **Jerry's** account
5. Go to **TimeGap Theory > Score**
6. Check if you got points for completing the **Transfer rewards challenge**

C timegaptheory.herokuapp.com/timegaptheory/webapp/admin/manage-users.php

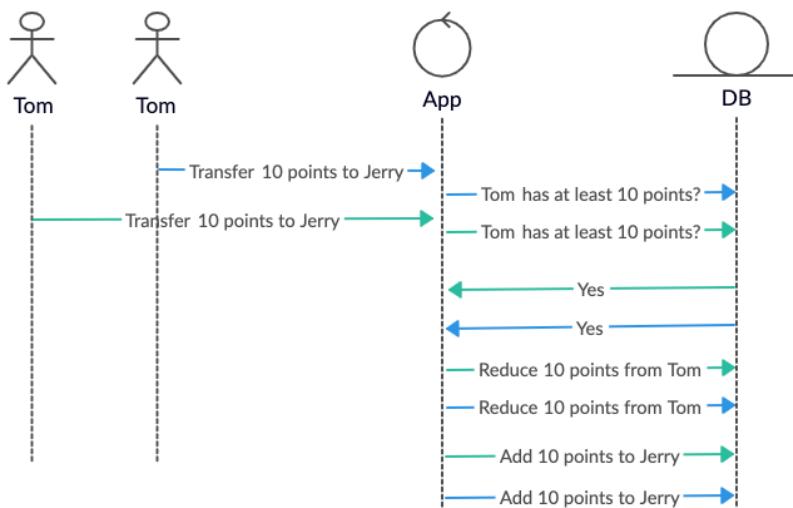
OWASP TimeGap Theory WebApp

OWASP TimeGap Theory / WebApp / Administration Panel / Manage Users

First Name	Email Address	Rewards	Login Attempts	Wel
tom	tom@timegaptheory.com	-80	0	0
jerry	jerry@timegaptheory.com	1080	0	0
spike	spike@timegaptheory.com	1000	0	0
tyke	tyke@timegaptheory.com	2000	0	0

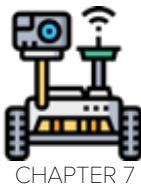
What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. A user may start a huge number of parallel requests. They may transfer this boat load of money/points to an account controlled by them or by their friends/relatives
2. Attacker may work with friends to perform these transactions back and forth infinite amount of times



Let us review what we did:

1. The transfer rewards page is performing basic checks to prevent abuse
2. One of these checks includes checking the payer's account to see if they have sufficient balance
3. You analyzed this behavior by trying to transfer more points than what you have in balance
4. First, you slowed down the system and bypassed the business logic
5. Then you bypassed the business logic by using browser dev tools
6. You must have noticed during the automation phase that accessing the transfer-rewards page does not require authentication.
 - a. If you manually enter the transfer-rewards page URL on your browser, you will be able to access this page without logging in first
 - b. This made the **TOCTOU** exploitation slightly easy. No need to worry. In upcoming chapters, you will learn how to exploit **TOCTOU** issues in authenticated pages as well.
7. Now you know how **TOC/TOU** security issues can affect money/points transfer pages



CHAPTER 7

Challenge 4 – Mars

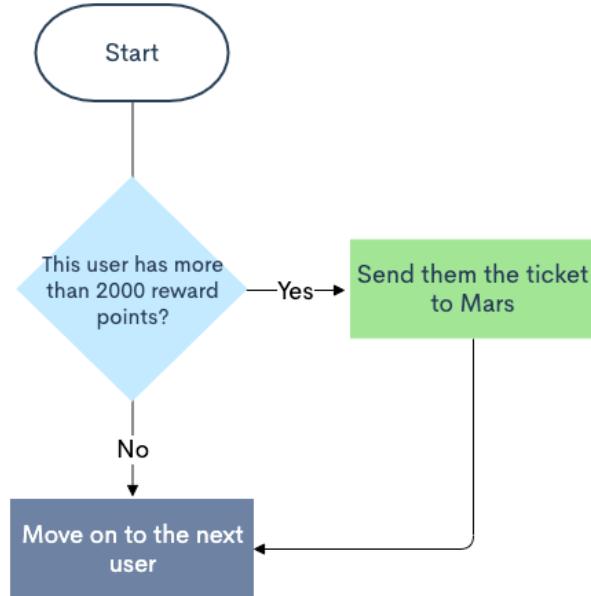
Get tickets to Mars.
Explore the possibilities.



Flying to Mars? We haven't seen such a possibility so far. We have something similar though - flying to Paris.

Despite the name, the basic functionality is simple. The page is run by admins or an automated script. Once run, it will go through each user account. If the user has more than 2000 reward points, the page will send a ticket to the user.

Let us visualize that:



There is only one condition to be met for a user to earn the ticket - their account should have more than 2000 points at the point of check.

Let us see if the application is performing those checks:

1. Go to the **Sign up** page
2. Create two user accounts - **Tom** and **Jerry**
3. Go to **Admin > Manage Users**
4. Edit user **Tom's** rewards points to **3000**
5. Navigate to **Admin > Ticket to Mars**

 timegaptheory.herokuapp.com/timegaptheory/webapp/admin/ticket-to-mars.php



OWASP TimeGap Theory WebApp

[OWASP TimeGap Theory](#) / [WebApp](#) / [Administration Panel](#) / [Ticket to Mars](#)

tom@timegaptheory.com is eligible. Notification email sent to the user.
jerry@timegaptheory.com is NOT eligible.

The page is working. App says **Tom** is eligible and **Jerry** is not.

We need to find a way to get two tickets to Mars. How can we do that?
What if we transfer points from Tom's account to Jerry's account immediately before the app checks Jerry's balance? Let us try that

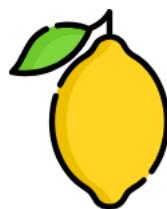
It's time to slow down TimeGap Theory:

1. On your browser, navigate to **TimeGap Theory > Settings**
2. Change the **Mars wait** to **5** seconds
3. Click on the **Save** button

Now, **TimeGap Theory** will wait 5 seconds before every database write operation.

Preparation phase

1. Open transfer rewards page
2. Click on the first \$10 from Tom to Jerry button
3. Change the transfer amount to 2000



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation phase

1. Open another browser or private/incognito window
2. On the new window, navigate to TimeGap Theory > Webapp > Admin > Ticket to Mars
3. Immediately go to the first browser window and click on the Transfer button



Let us see that worked or not

Post-exploitation phase

1. Let us wait for both the browsers to complete the request
2. You can see that both Tom and Jerry got tickets to Mars
3. If you check the balance of Tom, you will see that their balance is less than 2000
4. If you check your scores, you will see that you have got 100 points for completing the Mars challenge.
5. Don't forget to change the Main Wait to 0 on settings page.



We are skipping the automation part for this chapter. In the real world, attackers would use the same automation technique as mentioned in the transfer-rewards page. Then, they will schedule their automation program to run at a specific time they have chosen. If the attacker knows from their past experience that the **Ticket to Mars** program usually starts running at 12am on January 1st, they will schedule their attack script to trigger at the exact time.

What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. Users may keep on circling their points/money to get the benefits
2. Business will end-up spending too much on benefits compared to what they were anticipating

Let us review what we did:

1. The **Ticket to Mars** program gives some benefits to users if they have reward points more than 2000
2. You checked this functionality first
3. You slowed down the system and bypassed the business logic



CHAPTER 8

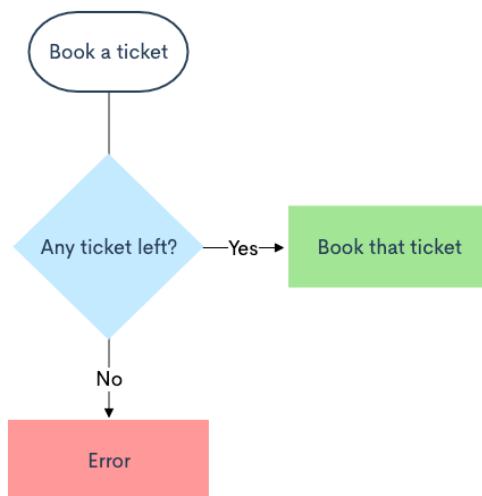
Challenge 5 – Tickets

Get tickets to the Show.

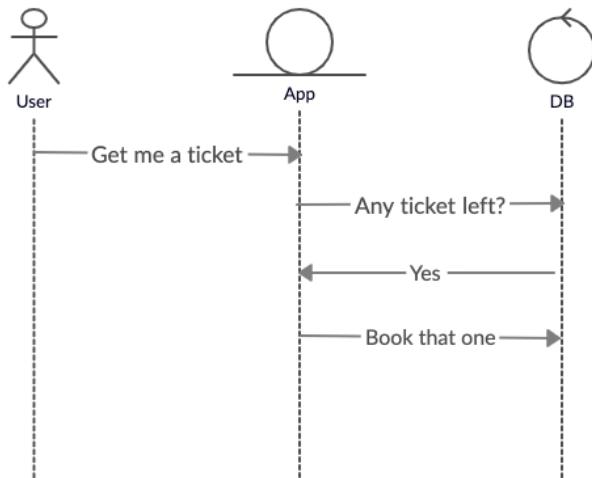
Enjoy the Show.



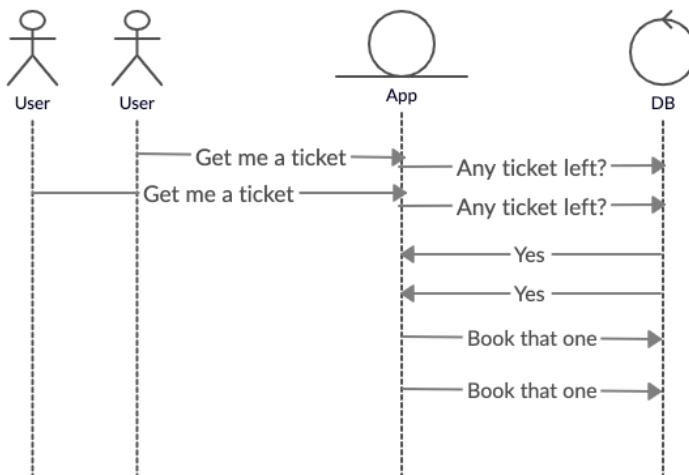
Enjoy going to the movies? This page is for booking the show. Unfortunately, there is only one ticket left. All the other tickets were sold offline. Whoever is able to complete the transaction first will get the ticket.



Let us visualize the happy path with the help of a sequence diagram:



That is just the happy path for one user purchasing a ticket. Let us reimagine the same scenario but with two users acting at the same time.



There is only one condition to be met for a user to purchase the ticket - there should be at least one ticket available in the system.

Let us see if the application is performing those checks:

1. Go to the **login** page
2. Login as any user
3. Navigate to the **buy tickets** page
4. Click on **Buy One Ticket** button
5. Observe the message **You have 1 ticket(s)** message on top of the page
6. Click on **Buy One Ticket** button again
7. See that nothing happens
8. Logout and login as another user
9. Navigate to the **buy tickets** page
10. Click on **Buy One Ticket** button
11. See that user is not able to buy tickets since there are no more tickets left

We need to find a way to purchase two tickets. How can we do that? What if we click on the buy ticket button multiple times very fast? That won't help cause the page needs to be loaded after each click.

It's time to slow down **TimeGap Theory**:

1. On your browser, navigate to the **Settings** page
2. Change the time delay to **5** seconds
3. Click on the **Save** button

Now, TimeGap Theory will wait 5 seconds before every database write operation.

Preparation phase

1. Navigate to the **Admin** page
2. Click on the **create default users** button
3. Open two browsers side by side (Use private/incognito window if you do not have two browsers)
4. Navigate to the **login** page on both the browsers
5. On the first browser:
 - a. Click on the first user button on top
 - b. This will fill the user data for user **Tom**
 - c. Submit the form to log in as **Tom**
6. On the second browser:
 - a. Click on the second user button on top
 - b. This will fill the user data for user **Jerry**
 - c. Submit the form to log in as **Jerry**



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation phase

1. On both the browsers:
 - a. Navigate to the [buy tickets](#) page
 - b. Click on the [Buy One Ticket](#) button



Let us see how that turns out.

Post-exploitation phase

1. Let us wait for both the browsers to complete the request
2. You can see that both the users have one ticket on their account
3. If you check your scores, you will see that you have got [100](#) points for completing the [buy tickets](#) challenge.
4. Don't forget to change the delay to [0](#) on the [settings](#) page.



Automation

As always, we need a way to automate this.

Pro-tip - you can skip this in real life if the webmaster is your friend and is ready to slow down the server for you.

First, we need a valid fetch request:

1. Open your browser ([Chrome](#) or [Firefox](#))
2. Navigate to [TimeGap Theory](#) > [Webapp](#) > [Login](#)
3. Click on the first user button. This will load the user data for [Tom](#)
4. Click on the [Sign in](#) button
5. Navigate to the [buy tickets](#) page
6. Open up dev tools by pressing F12 on the browser
 - a. On Windows, you can use [Ctrl + Shift + I](#)
 - b. On Mac, you can use [Cmd + Shift + I](#)
7. Click on the [Buy one ticket](#) button
8. On the browser dev tools, click on the [Network](#) tab
9. Right click on the [buy-ticket.php](#) request
10. Click on [Copy](#) > [Copy as fetch](#)

OWASP TimeGap Theory / WebApp / User
/ Buy Tickets

The screenshot shows the Network tab of a browser's developer tools. At the top, there are tabs for Elements, Console, Sources, Network (which is selected), and Performance. Below the tabs, there are filters for Preserve log (checked), Disable cache (unchecked), and Online (selected). The Network tab displays a list of requests. A red arrow points to the first request in the list, which is 'buy-tickets.php'. This request has a status of 200, a type of document, and an initiator of 'buy-tickets.php'. Other requests listed include 'owasp.png', 'timegaptheory.png', and 'lab.png', all with a status of 200 and a type of png, with initiators of 'buy-tickets.php'.

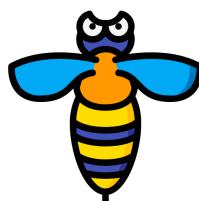
Name	Status	Type	Initiator
buy-tickets.php	200	document	buy-tickets.php
owasp.png	200	png	buy-tickets.php
timegaptheory.png	200	png	buy-tickets.php
lab.png	200	png	buy-tickets.php

Paste that on the **Console** tab of the browser dev tools. You will get something like the following:

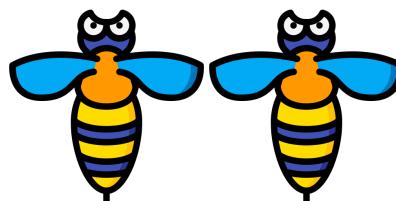
```
: ⌂ Elements Console Sources Network Performance Memory Application Se  
| ⌂ top | Filter Default levels ▾  
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/buy-tickets.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": null,  
  "method": "POST"  
});
```

Based on how playful your browser is, you may see a slightly long fetch request. Feel free to trim it down to the bare minimum version shown above.

Now we have a request.



And we need two requests.



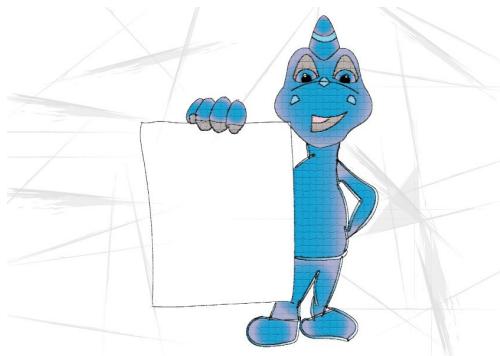
It can be done by copy-pasting the fetch request without pressing the **Enter** key. However, it's a good practice to separate each request with a comma. Let us do that



```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/buy-tickets.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": null,  
  "method": "POST"  
}),fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/buy-tickets.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": null,  
  "method": "POST"  
});
```

Before running this new command, we need to clear the slate. It involves three simple steps:

1. Ensure that there is no delay
 - a. Go to **TimeGap Theory > Settings**
 - b. Ensure that the delay is set to **0**
2. Reset database and create users
 - a. Go to **TimeGap Theory > Admin**
 - b. Click on **reset database** button
 - c. Go back to **TimeGap Theory > Admin**
 - d. Click on **create default users** button
3. Clear the current score
 - a. Go to **TimeGap Theory > Score**
 - b. Click on the **Clear** button



Alright, our slate is clear. Let us execute the attack now:

1. Navigate to the **Login** page
2. Click on the first user button on top. This will fill the user data for user **Tom**
3. Click on the **Sign In** button
4. Enter the combined fetch request on the **Console** tab of browser dev tools
5. Press the **Enter** key
6. Refresh the page
7. You will see that our attack attempt is unsuccessful.

What happened?

Two things went wrong:

1. The **Buy Tickets** page of **TimeGap Theory** is authenticated. We need to supply the cookie as well in order for this request to be successful
2. Since an authenticated session is involved, webapp is treating the executing requests one by one when it is from the same session

We need to solve both these problems for the automation to be successful.

Problem 1 - Cookie



Fetch request does not support sending cookies. We need an alternate solution. [cURL](#) supports sending cookies. We can use that.

Problem 2 - Only one request per session



This can be solved by creating two sessions for the same user. How can we do that?

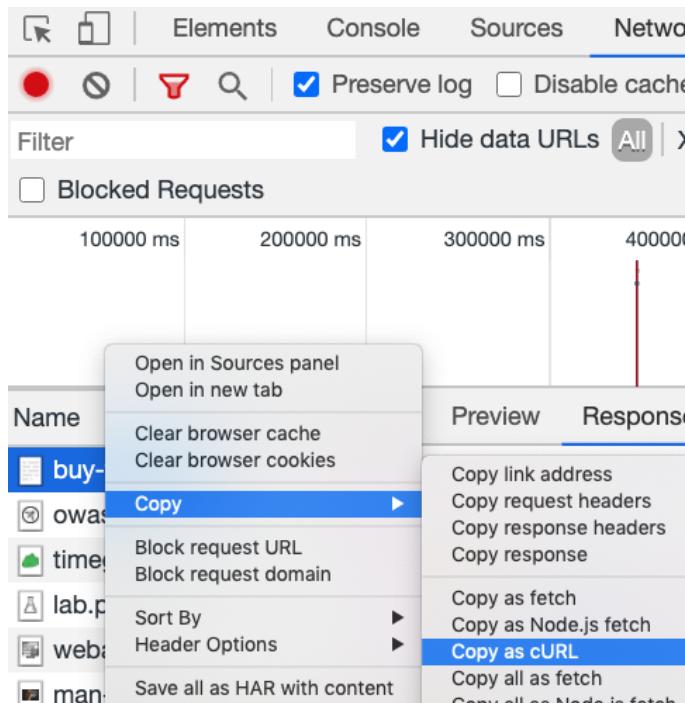
1. We will sign into [Tom](#)'s account on the first browser.
2. We will sign into [Jerry](#)'s account on the second browser.

This way we will get two separate active sessions.

Now, we need valid [cURL](#) requests with session tokens in it.

Let us obtain the first one:

1. Open your browser (**Chrome** or **Firefox**)
2. Navigate to **TimeGap Theory > Webapp > Login**
3. Click on the first user button. This will load the user data for **Tom**
4. Click on the **Sign in** button
5. Navigate to the **Buy Tickets** page
6. Open up dev tools by pressing **F12** on the browser
 - a. On Windows, you can use **Ctrl + Shift + I**
 - b. On Mac, you can use **Cmd + Shift + I**
7. Click on the **Buy One Ticket** button.
8. On the browser dev tools, click on the **Network** tab
9. Right-click on the **buy-tickets.php** request.
10. Click on **Copy > Copy as cURL**



For getting the second one:

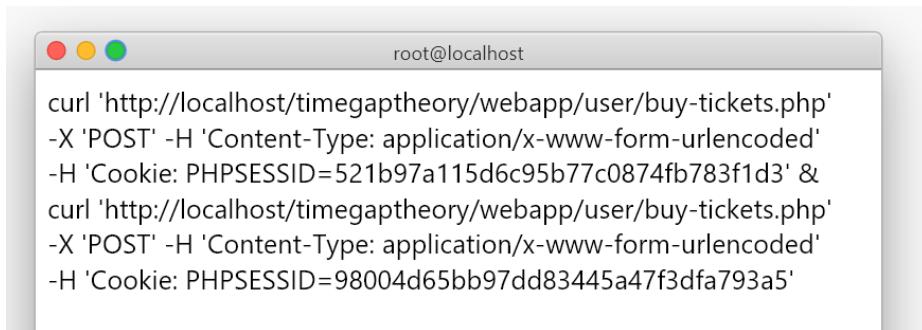
1. Open another browser (**Chrome** or **Firefox**)
2. Navigate to **TimeGap Theory > Webapp > Login**
3. Click on the second user button. This will load the user data for **Jerry**
4. Click on the **Sign in** button
5. Navigate to the **Buy Tickets** page
6. Open up dev tools by pressing **F12** on the browser
 - a. On Windows, you can use **Ctrl + Shift + I**
 - b. On Mac, you can use **Cmd + Shift + I**
7. Click on the **Buy One Ticket** button
8. On the browser dev tools, click on the **Network** tab
9. Right click on the **buy-tickets.php** request.
10. Click on **Copy > Copy as cURL**

Now that we have both the requests. Let us run them in parallel. Before doing that, we need to clear all the previous purchases of tickets. This can be done by reloading the **create-users.php** page located at **TimeGapTheory > Webapp > Admin > Create Users**.

Once you have reloaded the **create-users.php** page, follow the steps below:

1. Open your command prompt/**Terminal** window
2. If you are on Windows⁵
 - a. Type **start /b**
 - b. Put a space
3. Enter the first **cURL** request
4. Put a space
5. Enter the ampersand symbol (&)⁶
6. Put a space again
7. If you are on Windows
 - a. Type **start /b**
 - b. Put a space
8. Enter the second **cURL** request

9. If you are on Windows
 - a. Remove the following parameter from both the requests “--compress”. Some versions of **cURL** do not support this feature.
10. Now, press the **Enter/Return** key



A screenshot of a terminal window titled "root@localhost". The window contains the following command-line session:

```
curl 'http://localhost/timegaptheory/webapp/user/buy-tickets.php'  
-X 'POST' -H 'Content-Type: application/x-www-form-urlencoded'  
-H 'Cookie: PHPSESSID=521b97a115d6c95b77c0874fb783f1d3' &  
curl 'http://localhost/timegaptheory/webapp/user/buy-tickets.php'  
-X 'POST' -H 'Content-Type: application/x-www-form-urlencoded'  
-H 'Cookie: PHPSESSID=98004d65bb97dd83445a47f3dfa793a5'
```

Let us see if we are successful or not:

1. Go to the first browser window
2. Refresh the **Buy Tickets** page
3. Note the message on the top that says **You have 1 ticket(s)**
4. Go to the second browser window
5. Refresh the **Buy Tickets** page
6. Note the message on the top that says **You have 1 ticket(s)**
7. Check if you got points for completing the **Buy Tickets** challenge

What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. A legitimate user may show up for the show and realize that another user also got the same ticket
2. Business would end up having unhappy customers, giving compensations etc.

Let us review what we did:

1. The buy tickets page checks the stock of the tickets before allowing the user to book it
2. You analyzed this behavior by trying to purchase two tickets
3. First, you slowed down the system and bypassed the business logic
4. Then you tried bypassing the business logic using fetch requests on browser dev tools and failed
5. You learned that fetch requests are not useful while trying to send cookies to the server
6. You moved onto **cURL** as it supports sending cookies and you were able to purchase one ticket each for two users.
7. Now you know how **TOC/TOU** security issues can affect **buy tickets** pages



CHAPTER 9

Challenge 6 – Coupon

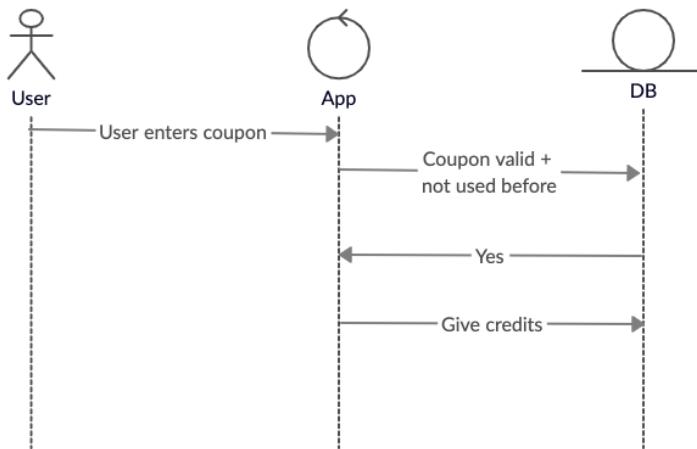
One-time use coupons.

Get extra rewards.

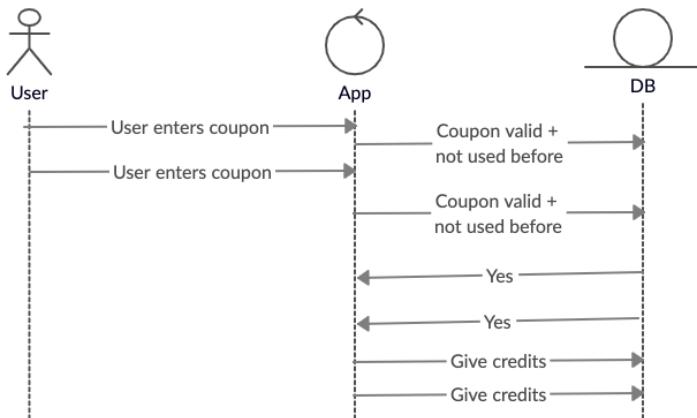


We all love coupons. Coupons help you get additional discounts or points. The only problem with them is that they can only be used once. Same is the scenario here. There is a one-time coupon which can only be used once by a customer.

Let us visualize the happy path with the help of a sequence diagram:



That is just the happy path for one user entering a valid coupon once. Let us reimagine the same scenario but with the user sending two requests at the same time with a valid coupon.



There are two conditions to be met for user to get credits from the coupon:

1. Coupon code should be valid
2. It should not have been used before

Let us see if the application is performing those checks:

1. Go to the **Admin** page
2. Click on **create default users** button
3. Navigate to the **login** page
4. Click on the first user button. This will fill in the user data for user **Tom**
5. Click on the **Sign In** button
6. Navigate to **enter coupon** page
7. Click on the **Invalid token** button on top. This will fill the coupon field with an invalid coupon code.
8. Click on the **submit** button
9. Note that the page is not accepting invalid coupons
10. Click on the **valid token** button on top. This will fill the coupon field with a valid coupon code.
11. Click on the **submit** button
12. You will see an increase in your reward points
13. You will also see **you are our preferred customer** message
14. Try repeating step number **10** and **11**
15. You will see that the application is not accepting the same coupon anymore

We need to find a way to use the coupon code twice. How can we do that?
What if we submit the valid coupon twice?

It's time to slow down **TimeGap Theory**:

1. On your browser, navigate to the **settings** page
2. Change the time delay to **5** seconds
3. Click on the **Save** button

Now, **TimeGap Theory** will wait **5** seconds before every database write operation.

Preparation phase

1. Navigate to the **Admin** page
2. Click on the **reset database** button
3. Click on the **create default users** button
4. Open two browsers side by side (Use private/incognito window if you do not have two browsers)
5. On both the browsers:
 - a. Navigate to the **login** page
 - b. Click on the first user button on top. This will fill the user data for user **Tom**
 - c. Submit the form to log in as **Tom**
 - d. Navigate to the **enter coupon** page
 - e. Note down the current reward points



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation phase

1. Click on the **Submit** button on the first browser
2. Go to the second browser as soon as you can
3. Click on the **Submit** button on the second browser



That's it. Let us see the result:

Post-exploitation phase

1. Let us wait for both the browsers to complete the request
2. Check the rewards points on **Tom**'s account
3. You will see that you got more than **500** points with the help of same coupon
4. If you check your scores, you will see that you have got **100** points for solving the **Coupon** challenge
5. Don't forget to change the delay to **0** on the settings page.



What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. Users will use one-time tokens and coupons multiple times
2. Business can suffer from financial issues

Let us review what we did:

1. The **enter coupon page** limits the number of times one-time coupons can be used
2. You analyzed this behavior by trying to use same coupon multiple times
3. You slowed down the system and bypassed the business logic
4. Now you know how **TOC/TOU** security issues can affect one-time-use coupons and tokens

We are skipping the automation part for this chapter. The steps would be the same as the **Buy Tickets** challenge but with a slight change. Since you are trying to apply the coupon to one user account multiple times, you need to get two active sessions for the same user. This can be done by:

1. Navigating the login page on the first browser and logging in as **Tom**
2. Navigating the login page on the second browser and again logging in as **Tom**

Without creating two sessions, automation attempts would fail as the application will process requests originating from the same session one by one.



CHAPTER 10

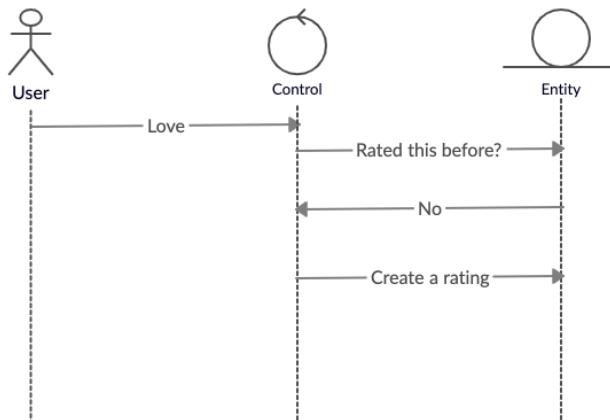
Challenge 7 – Ratings

Rate them high.
Rate them good.

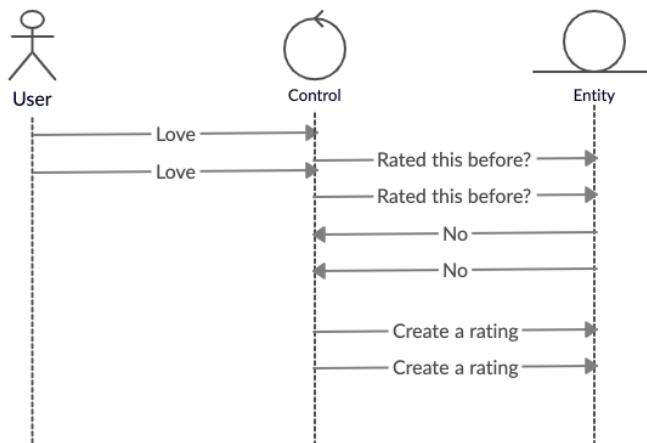


Ratings systems are one of the main features of social web apps these days. Sometimes it is a simple like/love button or it can be a full-fledged five-star rating system. A user is supposed to rate an object only once.

Let us visualize the happy path with the help of a sequence diagram:



That is just the happy path for one user rating the post. Let us reimagine the same scenario but with the same user acting at the same time from two browsers.



There is only one condition to be met for a user to rate a show - they shouldn't have rated the show before. If they have already rated the show, the app will remove the existing rating from the database.

Let us see if the application is performing those checks:

1. Go to the **Admin** page
2. Click on **reset database** button
3. Go back to the **Admin** page
4. Click on **Create default users** button
5. Navigate to the **login** page
6. Click on the first user button. This will load the user data for user **Tom**
7. Click on the **Sign in** button
8. Navigate to **rate the program** page
9. Click on the **Love** button
10. Note the message on the top that says **You have 1 rating(s)**
11. Click on the **Love** button again
12. Note the message on the top that says **You have 0 rating(s)**

We need to find a way to rate the program twice. How can we do that?
What if we submit two rating requests in parallel?

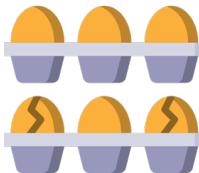
Let us slow down **TimeGap Theory**:

1. On your browser, navigate to the **Settings** page
2. Change the time delay to **5** seconds
3. Click on the **Save** button

Now, **TimeGap Theory** will wait **5** seconds before every database write operation.

Preparation phase

1. Navigate to the Admin page
2. Click on the reset database button
3. Click on the create default users button
4. Open two browsers side by side (Use private/incognito window if you do not have two browsers)
5. On both the browsers:
 - a. Navigate to the login page
 - b. Click on the first user button on top. This will fill the user data for user Tom
 - c. Submit the form to log in as Tom
 - d. Navigate to the rate the program page



Alright, the preparation is done. Here comes the exploitation phase.

Exploitation phase

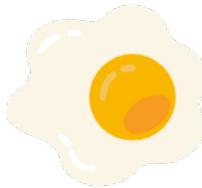
1. Click on the Love button on the first browser
2. Go to the second browser as soon as you can
3. Click on the Love button on the second browser



That's it. Let us see the result:

Post-exploitation phase

1. Let us wait for both the browsers to complete the request
2. Note the message on the top that says **You have 2 rating(s)**
3. If you check your scores, you will see that you have got **100** points for solving the **rate the program** challenge
4. Don't forget to change the delay to **0** on the **Settings** page.



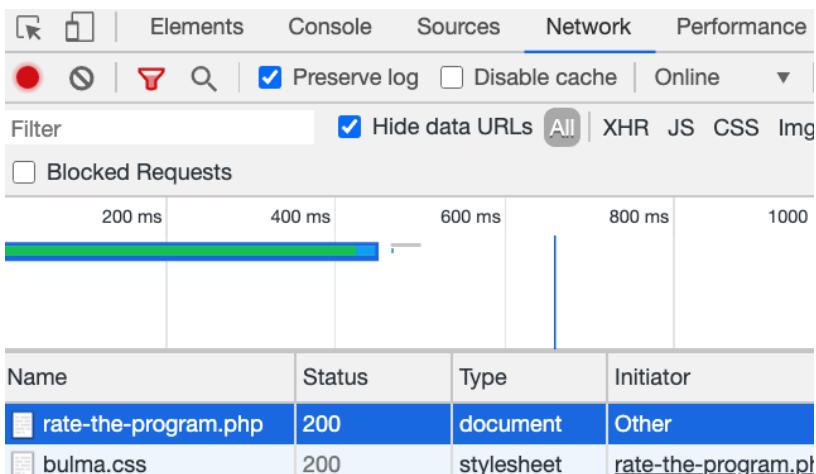
Automation time

As always, we need a way to automate this.

Pro-tip - you can skip this in real life if the webmaster is your friend and is ready to slow down the server for you.

First, we need a valid fetch request:

1. Open your browser (**Chrome or Firefox**)
2. Navigate to **TimeGap Theory > Webapp > Login**
3. Click on the first user button. This will load the user data for **Tom**
4. Click on the **Sign in** button
5. Navigate to the **rate the program** page
6. Open up dev tools by pressing **F12** on the browser
 - a. On Windows, you can use **Ctrl + Shift + I**
 - b. On Mac, you can use **Cmd + Shift + I**
7. Click on the **Love** button
8. On the browser dev tools, click on the **Network** tab
9. Right click on the **rate-the-program.php** request
10. Click on **Copy > Copy as fetch**



Paste that on the **Console** tab of the browser dev tools. You will get something like the following:

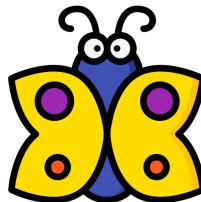
The screenshot shows the Console tab in the Chrome DevTools developer tools. At the top, there are icons for a magnifying glass, a refresh, and a square. The tabs are Elements, Console (selected), Sources, Network, Performance, Memory, Application, and Security. Below the tabs are buttons for top (selected), a dropdown arrow, and a filter input field. A 'Default levels' dropdown is also present. The console output shows a single line of code:

```
fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/rate-the-program.php", {
```

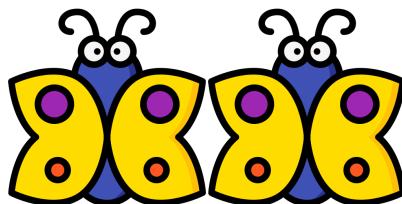
This is a partial representation of the fetch API call shown in the Network tab's table.

Based on how playful your browser is, you may see a slightly long fetch request. Feel free to trim it down to the bare minimum version shown above.

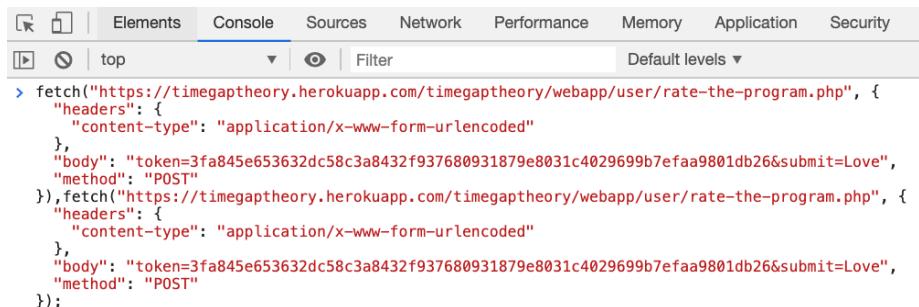
Now we have a request.



And we need two requests.



It can be done by copy-pasting the fetch request without pressing the **Enter** key. However, it is a good practice to separate each request with a comma. Let us do that

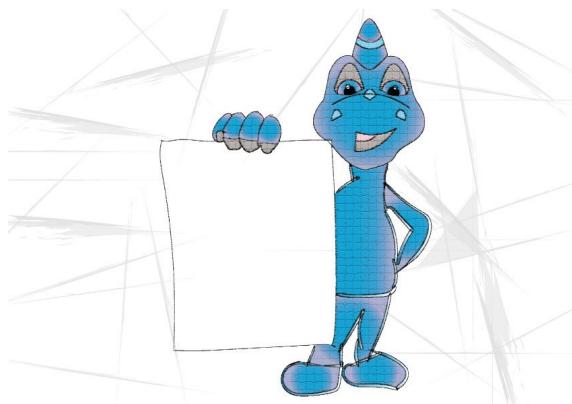
A screenshot of a browser developer tools console. The tabs at the top are Elements, Console, Sources, Network, Performance, Memory, Application, and Security. The Console tab is selected. Below the tabs, there are icons for back, forward, and search, followed by the text "top". A dropdown menu is open next to "top", and a "Filter" input field is to its right. To the far right, there is a "Default levels" dropdown. The main area of the console contains the following code:

```
> fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/rate-the-program.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "token=3fa845e653632dc58c3a8432f937680931879e8031c4029699b7efaa9801db26&submit=Love",  
  "method": "POST"  
},fetch("https://timegaptheory.herokuapp.com/timegaptheory/webapp/user/rate-the-program.php", {  
  "headers": {  
    "content-type": "application/x-www-form-urlencoded"  
  },  
  "body": "token=3fa845e653632dc58c3a8432f937680931879e8031c4029699b7efaa9801db26&submit=Love",  

```

Before running this new command, we need to clear the slate. It involves three simple steps:

1. Ensure that there is no delay
 - a. Go to **TimeGap Theory > Settings**
 - b. Ensure that the delay is set to **0**
2. Reset database and create users
 - a. Go to **TimeGap Theory > Admin**
 - b. Click on **Reset database** button
 - c. Go back to **TimeGap Theory > Admin**
 - d. Click on **Create default users** button
3. Clear the current score
 - a. Go to **TimeGap Theory > Score**
 - b. Click on the **Clear** button



Alright, our slate is clear. Let us execute the attack now:

1. Navigate to the **Sign In** page
2. Click on the first user button on top. This will fill the user data for user **Tom**
3. Click on the **Sign In** button
4. Enter the combined fetch request on the **Console** tab of browser dev tools

5. Press the **Enter** key
6. Refresh the page
7. You will see that our attack attempt is unsuccessful.

What happened?

Three things went wrong in here:

1. TimeGap theory now has a CSRF prevention token in the request.
This needs to be valid in order for the requests to be successful.
2. This page of **TimeGap Theory** is authenticated. We need to supply the cookie as well in order for this request to be successful
3. Since an authenticated session is involved, webapp is treating the executing requests one by one when it is from the same session

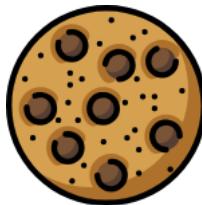
We need to solve all three problems.

Problem 1 - CSRF token



1. Open your browser and navigate to the **Rate the program** page
2. Open browser dev tools
3. Open the **Network** tab
4. Click on the **Love** button several times
5. On the dev tools, click on each request to **rate-the-program** page
6. See if we can find the nature of the anti-CSRF token
7. Note that the anti-CSRF token is same across all the requests

Problem 2 – Cookie



Fetch request does not support sending cookies. We need an alternate solution. [cURL](#) supports sending cookies. We will use that.

Problem 3 - Only one request per session



This can be solved by creating two sessions for the same user. How can we do that? We will just sign into the same account from two different browsers. Each session will be having their own anti-CSRF token as well.

First, we need two valid cURL requests.

Let us obtain the first one:

11. Open your browser (Chrome or Firefox)
12. Navigate to TimeGap Theory > Webapp > Login
13. Click on the first user button. This will load the user data for Tom
14. Click on the Sign-in button
15. Navigate to the rate the program page
16. Open up dev tools by pressing F12 on the browser
 - a. On Windows, you can use Ctrl + Shift + I
 - b. On Mac, you can use Cmd + Shift + I
17. Click on the Love button.
18. On the browser dev tools, click on the Network tab
19. Right-click on the enter-coupon.php request.
20. Click on Copy > Copy as cURL

For getting the second one:

11. Open another browser (Chrome or Firefox)
12. Navigate to TimeGap Theory > Webapp > Login
13. Click on the first user button. This will load the user data for Tom
14. Click on the Sign-in button
15. Navigate to the rate the program page
16. Open up dev tools by pressing F12 on the browser
 - a. On Windows, you can use Ctrl + Shift + I
 - b. On Mac, you can use Cmd + Shift + I
17. Click on the Love button.
18. On the browser dev tools, click on the Network tab
19. Right click on the enter-coupon.php request.
20. Click on Copy > Copy as cURL

Now that we have both the requests. Let us run them in parallel:

1. Open your command prompt/Terminal window
2. If you are on Windows
 - a. Type **start /b**
 - b. Put a space
3. Enter the first **cURL** request
4. Put a space
5. Enter the ampersand symbol (&)
6. Put a space again
7. If you are on **Windows**
 - a. Type **start /b**
 - b. Put a space
8. Enter the second **cURL** request
9. If you are on **Windows**
 - a. Remove the following parameter from both the requests -- **compress**. Some versions of **cURL** do not support this feature.
10. Now, press the **Enter/Return** key.

Let us see if we are successful or not:

1. Go back to any of your browser window
2. Refresh the **rate the program** page
3. Note the message on the top that says **You have 2 rating(s)**
4. Check if you got points for completing the **rate the program** challenge



What would be the business impact of such an attack? Depending on how the app is designed, there are several possibilities:

1. Users will be able to bombard the system with infinite number of fake ratings/reviews
2. If this is an online election system, voters would be able to make multiple votes

Let us review what we did:

1. The **rate the program** feature limits the number of times one can rate the program
2. You analyzed this behavior by trying to rate the program multiple times
3. First, you slowed down the system and bypassed the business logic
4. Then you tried bypassing the business logic by using browser dev tools
5. You learned the difficulties involved in exploiting **TOCTOU** vulnerabilities when there is an authenticated session
6. Now you know how **TOC/TOU** security issues in the real world



CHAPTER 11

More Tools

Start with whatever tools you may have.
Better tools will be found as you go.



Using a tool can help you find and exploit **TOCTOU** issues. There are a couple of such tools available in the open-source world:

Name	Notes
Browser Dev Tools	<ul style="list-style-type: none">● Very easy to find● Comes with major browsers● Does not support sending cookies
cURL	<ul style="list-style-type: none">● Easy to get● Works on almost all the platforms● Supports sending cookies● https://curl.haxx.se/

RaceTheWeb	<ul style="list-style-type: none"> ● Amazing tool geared for exploiting TOCTOU issues ● Easy to integrate in CI/CD pipeline ● Works on almost all the platforms ● https://github.com/aaronhnatiw/race-the-web
Requests Racer	<ul style="list-style-type: none"> ● Needs Python 3 to run ● https://github.com/hccgroup/requests-racer
PyRace	<ul style="list-style-type: none"> ● Needs Python 2 to run ● https://github.com/lamasoft/pyrace

We will solve the last challenge using **RaceTheWeb**. The executable files for **RaceTheWeb** can be obtained from <https://github.com/aaronhnatiw/race-the-web/releases>

Which version to download? It depends on the operating-system you are running

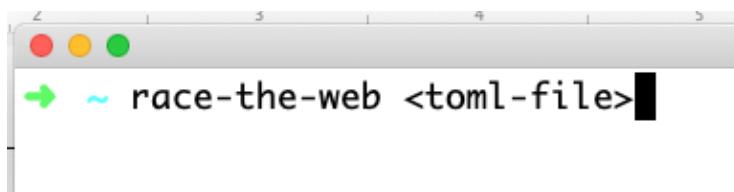
Operating System	File to be downloaded
Linux 32-bit	race-the-web_2.0.1_lin32.bin
Linux 64-bit	race-the-web_2.0.1_lin64.bin
macOS 32-bit	race-the-web_2.0.1_osx32.app.zip
macOS 64-bit	race-the-web_2.0.1_osx64.app.zip
Windows 32-bit	race-the-web_2.0.1_win32.exe
Windows 64-bit	race-the-web_2.0.1_win64.exe

On Linux and macOS machines, you need to make the file executable.

Operating System	Command to run on Terminal
race-the-web_2.0.1_lin32.bin	chmod +x race-the-web_2.0.1_lin32.bin
race-the-web_2.0.1_lin64.bin	chmod +x race-the-web_2.0.1_lin64.bin
race-the-web_2.0.1_osx32.app.zip	Extract the zip file chmod +x race-the-web_2.0.1_osx32.app
race-the-web_2.0.1_osx64.app.zip	Extract the zip file chmod +x race-the-web_2.0.1_osx64.app

Running **RacetheWeb** is easy:

1. Open command prompt/Terminal
2. Navigate to the directory where you have extracted/downloaded the executable binary file
3. Run the following command:



race-the-web portion:

Depending on the operating-system you are using, the **race-the-web** portion needs to be changed.

Operating System	race-the-web portion
Linux 32-bit	./race-the-web_2.0.1_lin32.bin
Linux 64-bit	./race-the-web_2.0.1_lin64.bin
macOS 32-bit	./race-the-web_2.0.1_osx32.app
macOS 64-bit	./race-the-web_2.0.1_osx64.app
Windows 32-bit	race-the-web_2.0.1_win32.exe
Windows 64-bit	race-the-web_2.0.1_win64.exe

<toml-file> portion:

TOML stands for **Tom's Obvious Minimal Language**. The **TOML** file supplied should be having the request details so that race-the-web can run them.

Find **TOML** file for some of the **TimeGap Theory** challenges below:

Sign Up page

```
# Transfer Rewards

count = 10
verbose = false

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/sign-up.php"
    body =
"firstname=tom&password=tom&email=tom%40example.com&rewards=100"
```

In the above **TOML** file:

- Count defines how many requests **RaceTheWeb** tools would be sending in parallel
- Verbose defines the verbosity level of output that is displayed on the screen. The value of this can either be true or false
- Method defines the type of the request. This can be GET, POST, PUT, DELETE etc.
- URL is, well, the url at which request needs to be sent
- Body of the request. You can skip this part if there is no body that needs to be submitted

Sign In page

```
# Sign In Page

count = 1
verbose = true

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/login.php"
    body = "email=tom%40sechow.com&password=1234&submit=Submit"

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/login.php"
    body = "email=tom%40sechow.com&password=password&submit=Submit"

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/login.php"
    body = "email=tom%40sechow.com&password=tom&submit=Submit"
```

In the above **TOML** file:

- Count is 1. However, there are three requests in the file. As such, **RaceTheWeb** tool will send three parallel requests
- First two requests have wrong password in the request body

Transfer rewards page

```
# Transfer Rewards

count = 10
verbose = false

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/user/transfer-rewards.php"
    body =
    "from=tom%40sechow.com&to=jerry%40sechow.com&amount=100&submit=Submit"
```

Ratings page

```
# Ratings page

count = 1
verbose = false

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/user/rate-the-program.php"
    body =
    "token=ae13e0f1df6412dc4b9e2a9a3354320b6c1f3a65160bcffb552495759870af
a3"
    cookies = ["PHPSESSID=80c3ffddbfe4771dd408b3c53d4a7a44"]

[[requests]]
    method = "POST"
    url = "http://localhost/timegaptheory/webapp/user/rate-the-program.php"
    body =
    "token=bc5091b2f60da51d203a58b2af1c8bd99a443751adb206814d22df79e335
a3e5"
    cookies = ["PHPSESSID=fab00a2f7f46e33bb57deb4e08153e52"]
```

In the above **TOML** file, we are sending a cookie as well.

Now you know:

1. Various open-source tools and techniques for finding and exploiting **TOCTOU** security issues
2. How to use **RaceTheWeb** tool for exploiting **TOCTOU** security issues
3. Writing **TOML** files for **RaceTheWeb** tool



Credits

Author

Abhi M Balakrishnan

Illustrations

Akhi Balakrishnan

Cover page and logo design

Adarsh Girijan

Technical review panel

Keith Johnson

Jennifer Diaz

Shashank Nigam

Aakash Kumar Goel

You're a TOCTOU Champion now!



- You are equipped with tools and techniques to check if your application is vulnerable to TOCTOU
- You can forecast TOCTOU issues on applications by looking at the high-level design in threat-modeling sessions
- You are also in a position to demonstrate TOCTOU issues to your peers

Visit TimeGapTheory.com to learn more



Addendum

1	OWASP Top 10	https://owasp.org/www-project-top-ten/
2	CWE's Top 25 Most Dangerous Software Errors	https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
3	Time-of-check to time-of-use	https://en.wikipedia.org/wiki/Time-of-check_to_time-of-use
4	Web app security testing with browsers	https://getmantra.com/web-app-security-testing-with-browsers/

5	How to run multiple DOS commands in parallel?	https://stackoverflow.com/questions/1010834/how-to-run-multiple-dos-commands-in-parallel#comment93535098_11011212
6	How do I use cURL to perform multiple simultaneous requests?	https://stackoverflow.com/questions/9624967/how-do-i-use-curl-to-perform-multiple-simultaneous-requests#comment104563689_58228357 https://stackoverflow.com/questions/1010834/how-to-run-multiple-dos-commands-in-parallel#comment93535098_11011212
7	Racing the Web (Aaron Hnatiw)	https://www.youtube.com/watch?v=4T99v957l0o

