Weak Bruteforce Protections

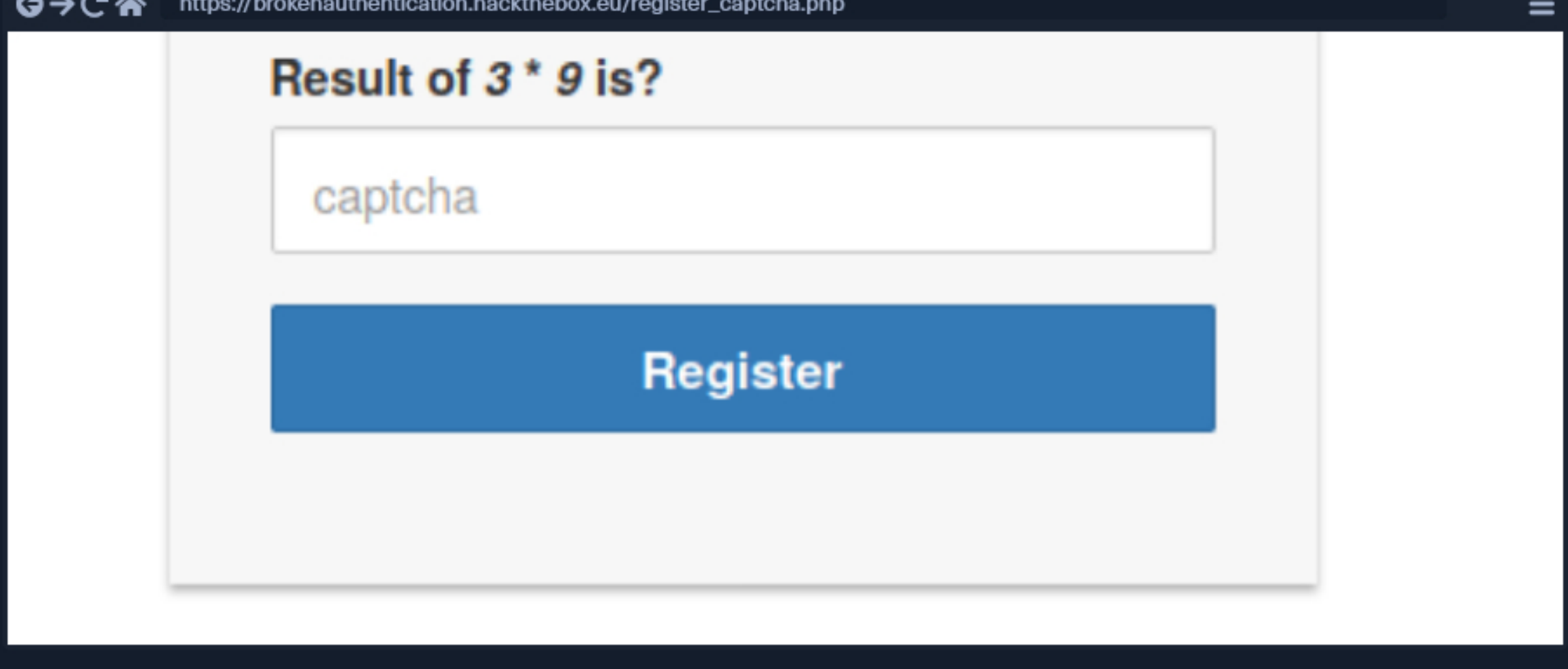
Before digging into attacks, we must understand the possible protections we could meet during our testing process. Nowadays, there are many different security mechanisms designed to prevent automated attacks. Among the most common are the following.

- `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.

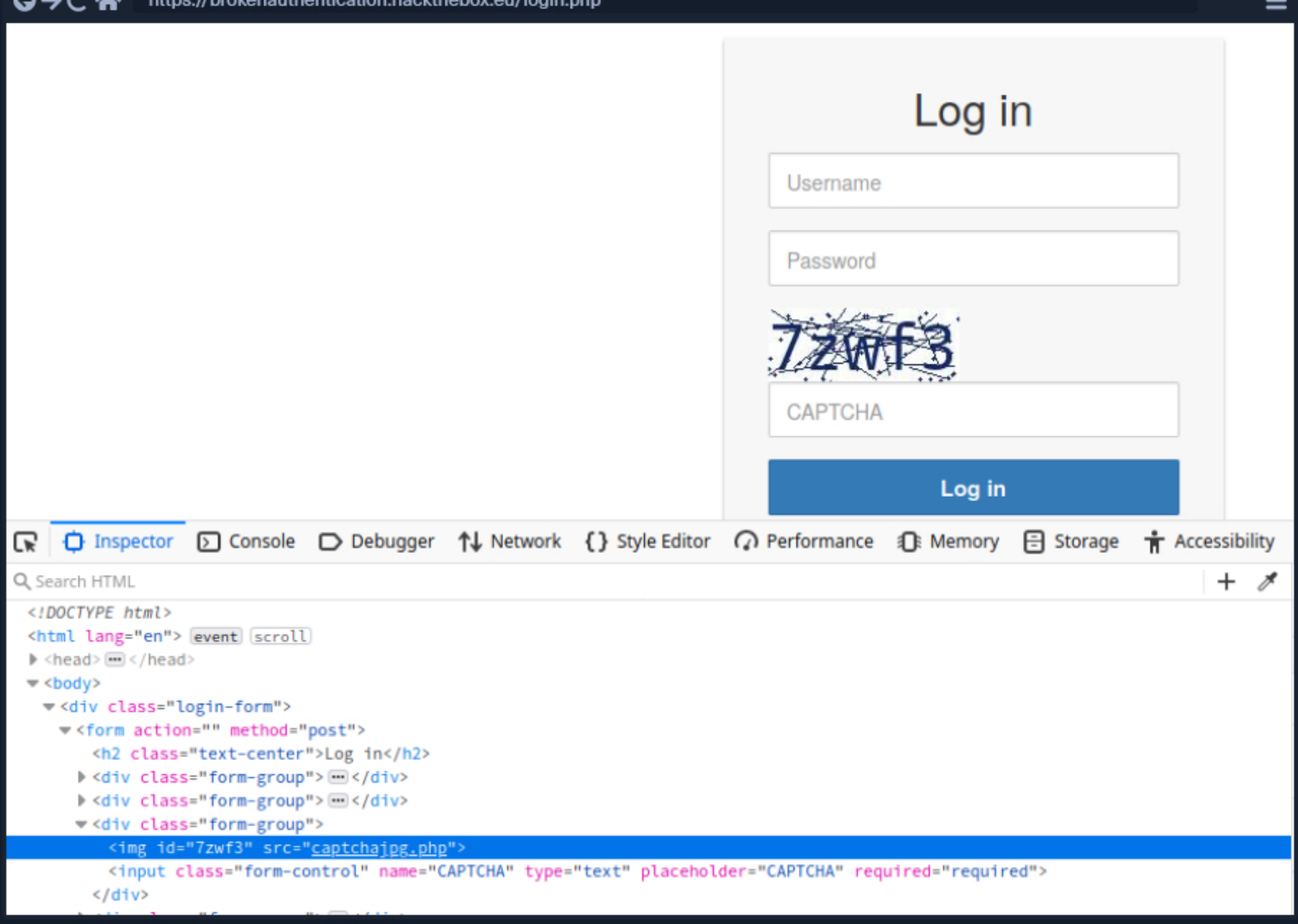
CAPTCHA

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.

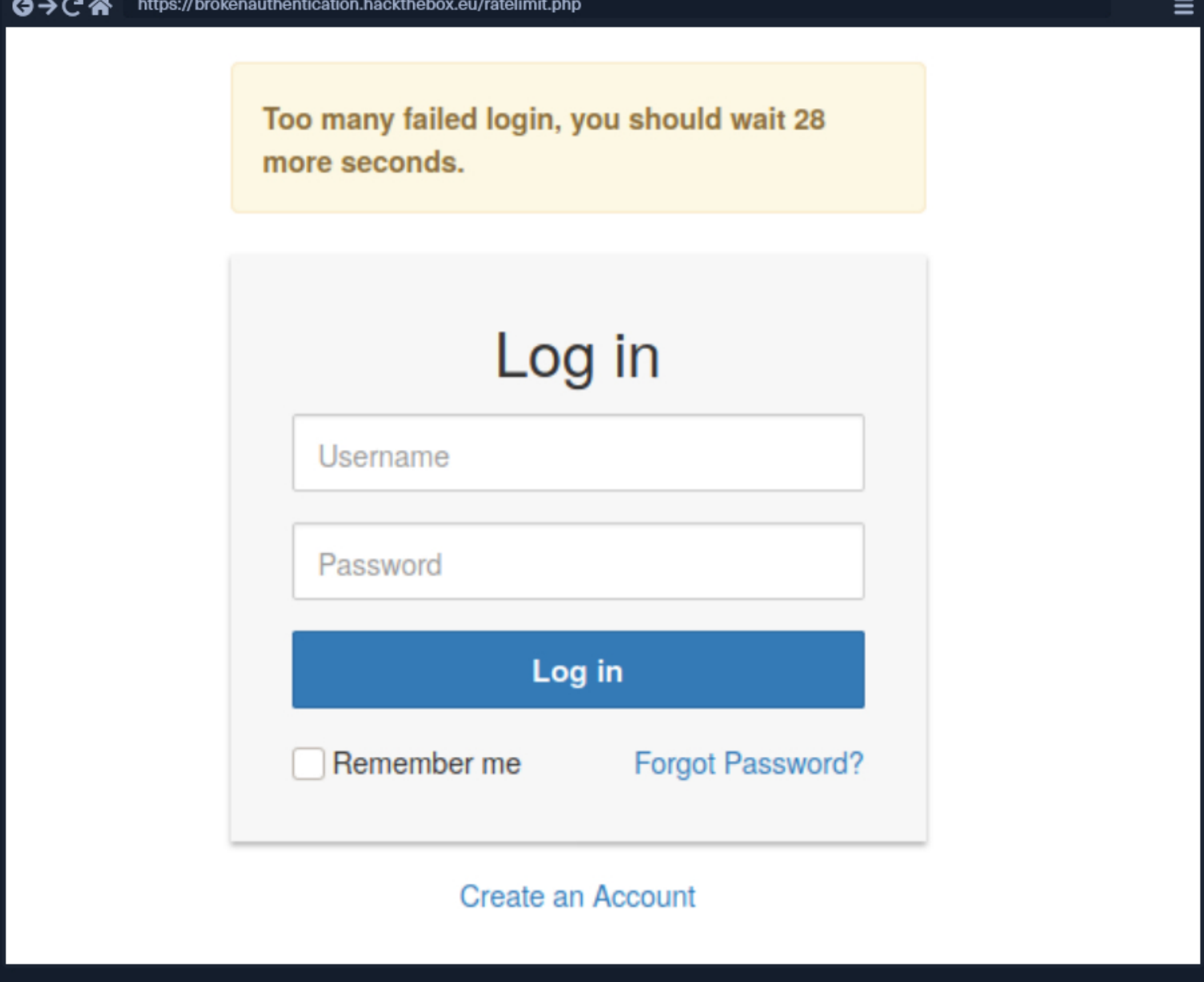


As an attacker, we can just read the page's source code to find the CAPTCHA code's value and bypass the protection. We should always read the source.

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.

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-liming 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) Chrome/88.0.4324.
  "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.

Start Instance

∞ / 1 spawns left

Waiting to start...

Questions

Answer the question(s) below to complete this Section and earn cubes!

Cheat Sheet

Target: Click here to spawn the target system!

+ 0 Observe the web application based at subdirectory /question1/ and infer rate limiting. What is the wait time imposed after an attacker hits the limit? (round to a 10-second timeframe, e.g., 10 or 20)

Submit your answer here... Submit Hint

+ 1 Work on webapp at URL /question2/ and try to bypass the login form using one of the method showed. What is the flag?

Submit your answer here... Submit Hint

Previous Next

Cheat Sheet

Resources

Go to Questions

Table of Contents

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left