**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 oftentimes 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.

A screenshot of a computer

Description automatically generated

For example, on [Hack The Box](https://hackthebox.eu), 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](https://core.trac.wordpress.org/ticket/3708) for the entire story

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

A screenshot of a login box

Description automatically generated

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.

A screenshot of a login box

Description automatically generated

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](https://github.com/danielmiessler/SecLists/tree/master/Usernames) 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.

A screenshot of a computer

Description automatically generated

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.

**WFuzz - Unknown Username**

WFuzz - Unknown Username

yovecio@htb[/htb]$ wfuzz -c -z file,/opt/useful/SecLists/Usernames/top-usernames-shortlist.txt -d "Username=FUZZ&Password=dummypass" --hs "Unknown username" http://brokenauthentication.hackthebox.eu/user\_unknown.php

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Wfuzz 3.1.0 - The Web Fuzzer \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Target: http://brokenauthentication.hackthebox.eu/user\_unknown.php

Total requests: 17

===================================================================

ID Response Lines Word Chars Payload

===================================================================

000000002: 200 56 L 143 W 1984 Ch "admin"

Total time: 0.017432

Processed Requests: 17

Filtered Requests: 16

Requests/sec.: 975.1927

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.

A screenshot of a login form

Description automatically generated

**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:

A screenshot of a login form

Description automatically generated

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:

A screenshot of a log in form

Description automatically generated

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

// 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 (so no users)

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](https://academy.hackthebox.com/storage/modules/80/scripts/timing_py.txt) to witness these types of timing differences and run it against an example web application ([timing.php](https://academy.hackthebox.com/storage/modules/80/scripts/timing_php.txt)) that uses bcrypt.

**Timing Attack - Timing.py**

Timing Attack - Timing.py

yovecio@htb[/htb]$ python3 timing.py /opt/useful/SecLists/Usernames/top-usernames-shortlist.txt

[+] user root took 0.003

[+] user admin took 0.263

[+] user test took 0.005

[+] user guest took 0.003

[+] user info took 0.001

[+] user adm took 0.001

[+] user mysql took 0.001

[+] user user took 0.001

[+] user administrator took 0.001

[+] user oracle took 0.001

[+] user ftp took 0.001

[+] user pi took 0.001

[+] user puppet took 0.001

[+] user ansible took 0.001

[+] user ec2-user took 0.001

[+] user vagrant took 0.001

[+] user azureuser took 0.001

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](https://en.wikipedia.org/wiki/2012_LinkedIn_hack) 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](https://www.tarsnap.com/scrypt.html), [bcrypt](https://en.wikipedia.org/wiki/Bcrypt) or [PBKDF](https://en.wikipedia.org/wiki/Pbkdf2) (or [argon2](https://en.wikipedia.org/wiki/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

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.

**Python - Hashtime.py**

Python - Hashtime.py

yovecio@htb[/htb]$ python3 hashtime.py

sha1: 0:00:00.000082

scrypt: 0:00:03.907575

bcrypt: 0:00:22.660548

Let us add some context by going over a rough example:

* LinkedIn has ~200M daily users, which means ~24 logins per second (we are not excluding users with a remember-me token).

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](https://tools.ietf.org/html/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.

A screenshot of a login form

Description automatically generated

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.