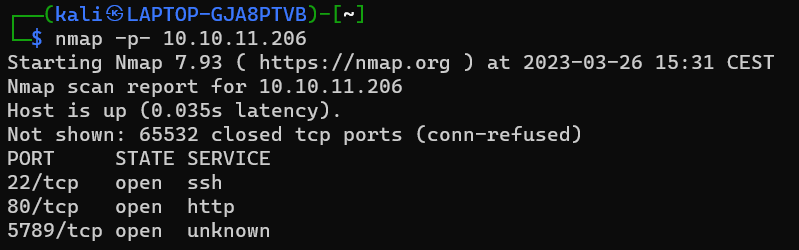
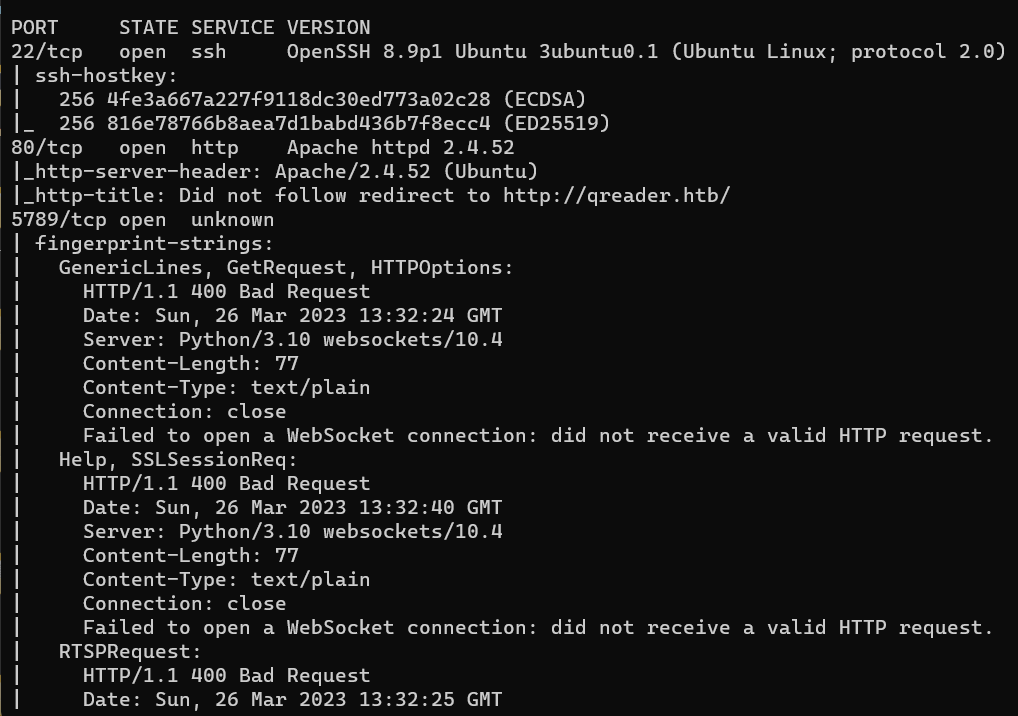
1. Pinged
2. Nmap -p- <target>

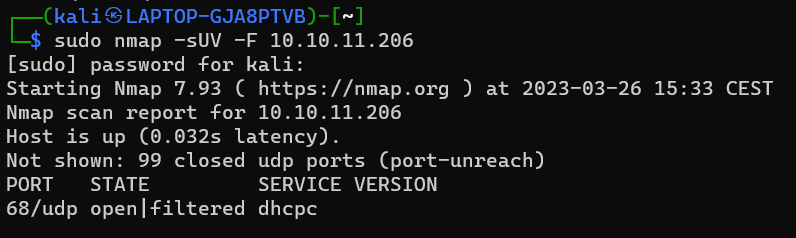


1. Sudo nmap -sS -sV -sC -O -p22,80,5789 <target>

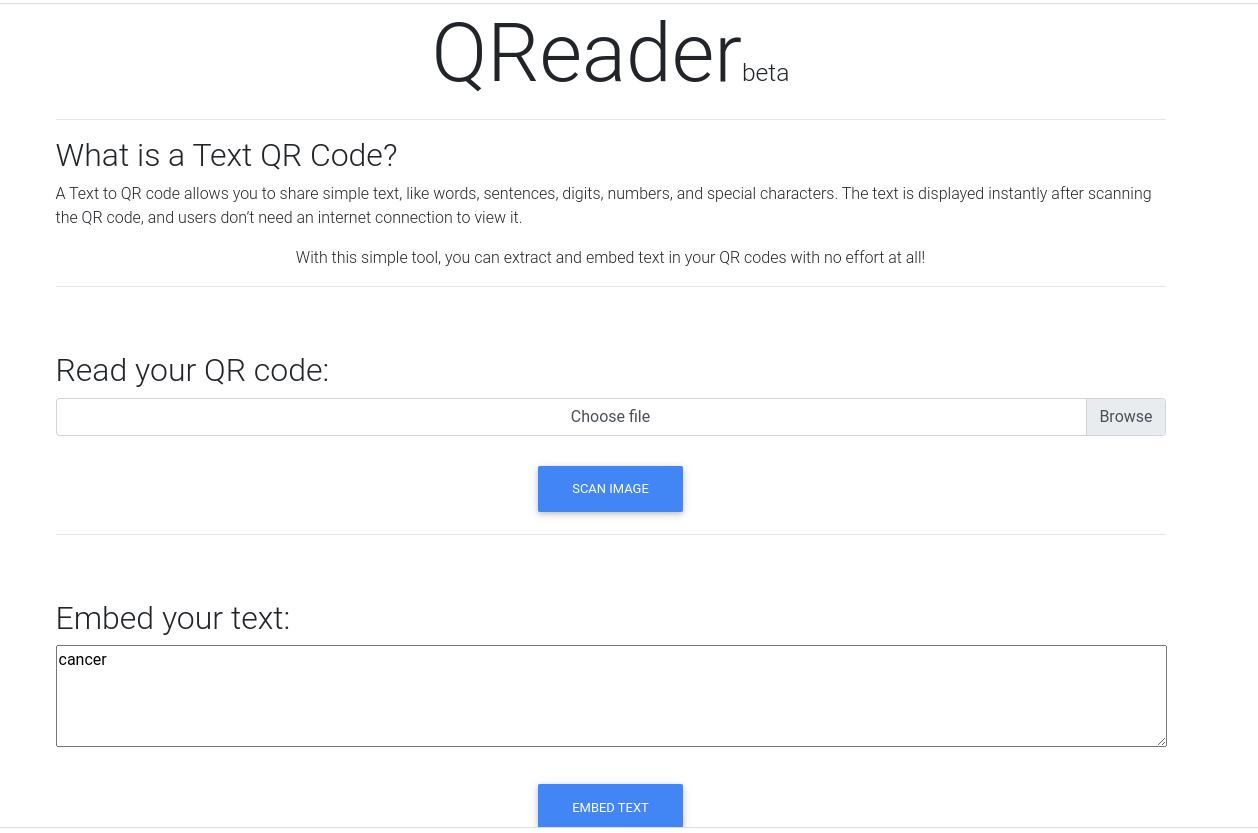


So it looks like the target is running ssh, a webserver on port 80 and some http service on port 5789 that is using python websockets version 10.4

I’ve also ran an UDP scan on the target, and surprisingly, there an udp port open:



First lets add the domain to our hosts file, and check the website out:

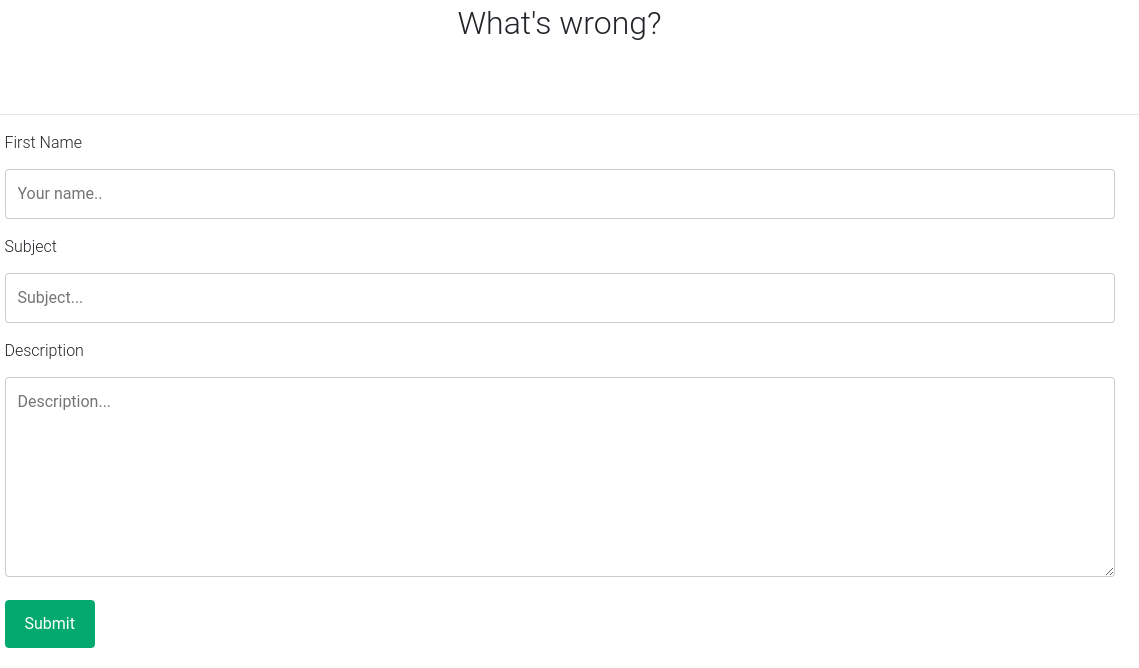


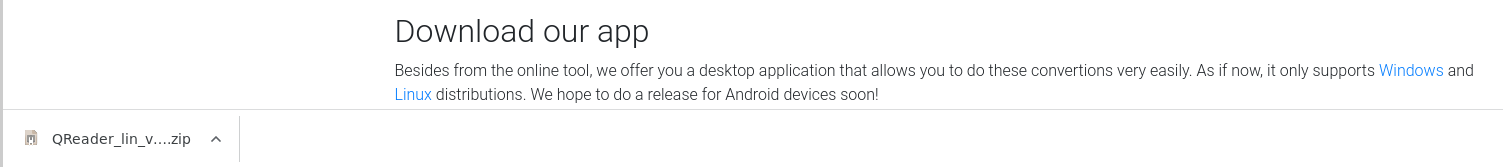
So the homepage has two functions:

1. Read text out of a qr code: allows image upload
2. Embed text into qr code: allows text input

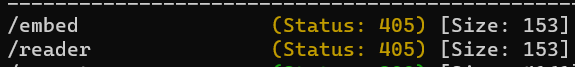
I’ve manually found two other pages on the website:

1. To report a problem: (allows text input)



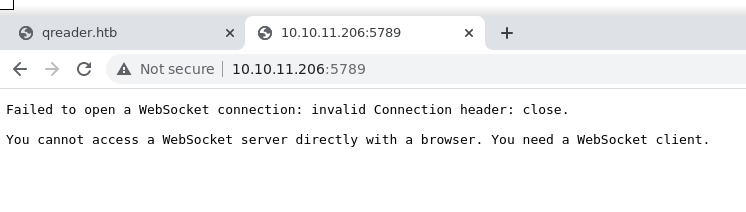
1. Download link: (gives us the whole file application to inspect)

Before we start poking, looking for vulnerabilities, I’m still going to run gobuster and nikto to see if we find more things:



It looks like there are two directories being used by the website to do the two functions of the home page.

1. Now lets visit the http service on port 5789



So as u can see, we’re working with a websocket here. What even is a websocket? A websocket is a network protocol (ws) that provides full duplex communication over a single tcp connection. It’s a communication protocol that operates over the HTTP protocol, enabling real-time communication between a client and a server. Unlike HTTP requests, which require the client to initiate a new connection with the server for each request, WebSocket establish a persistent connection between the client and the server. WebSockets are used for applications that require real-time, low-latency communication, such as online gaming, real-time chat and financial trading.

Examples:

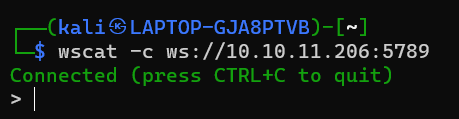
Twitter uses WebSockets to provide real-time updates to its users, such as new tweets, direct messages, and notifications.

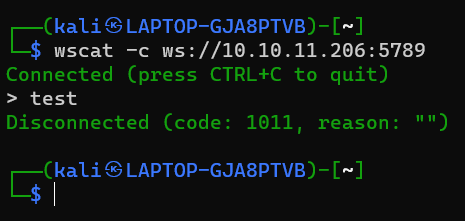
Facebook: Facebook uses WebSockets to provide real-time chat functionality to its users. The Facebook Messenger app uses WebSockets to provide a fast and responsive chat experience.

Instagram: Instagram uses WebSockets to provide real-time notifications to its users, such as new likes, comments, and direct messages.

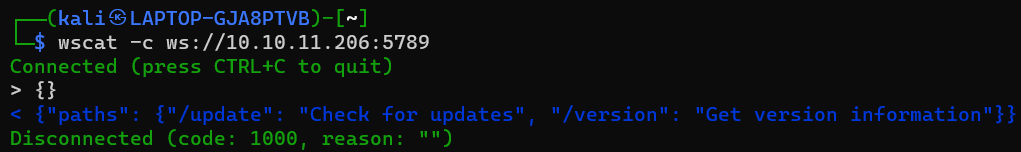
The data sent over the WebSocket connection can be in different formats, such as plain text, JSON, or binary data.

Let’s listen to the message and use a websocket client. We’re going to use wscat to connect to the ws server :



And we’re connected. Let’s pass some info with plain text:

Nothing. Lets try json:



Here we’ve got something. So we can assume the server expects json data. The empty json

payload provided us with the right paths. Let’s send some info to those paths.

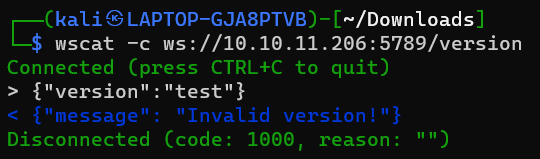
Now to find the right parameters for the content we’ve to pass in json, we have to use

PyInstallerExtractor:

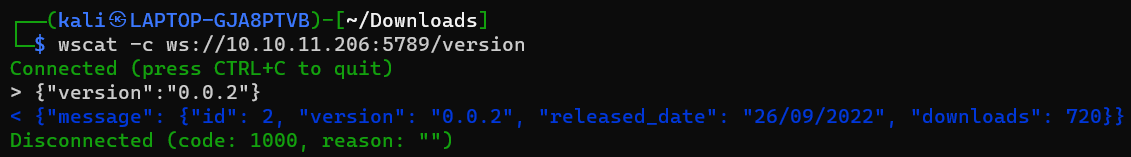
$> python3 pyinstxtractor qreader.exe

$> uncompyle6 qreader.pyc > qreader.py

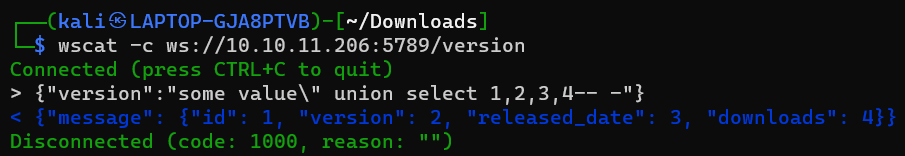
 $> cat qreader.py



Now let’s try the right version which 0.0.2, which I’ve found from the application I’ve downloaded from the website:



So it seems like the ws server is retrieving information from the database. So this is a potential sql injection point. Let’s try it:



And it works, there is an sqli in the ws server. This is how this sqli works:

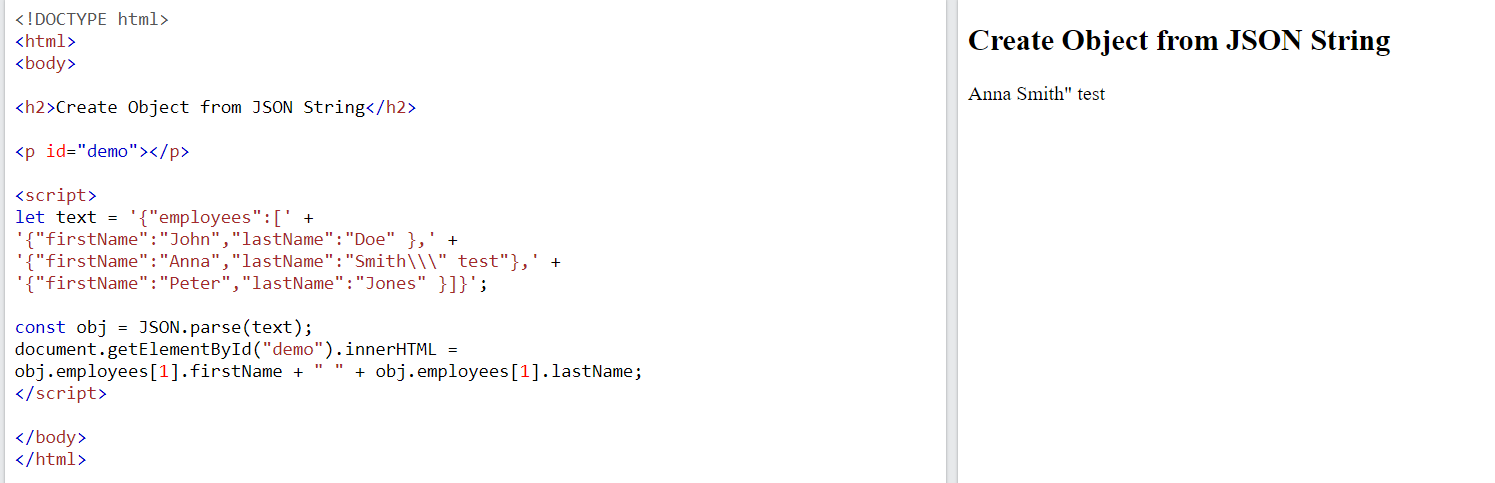
This is how the original query works:

Select \* from versions where version=”our value”, now we can’t use single quote to break out of the value to the query because it will turn out like this:

Select \* from version where version=”our value ‘our payload”

The single quote just gets included in the value. We need to somehow escape the double quotes. If we search for how to escape double quotes in json we get this answer:

With \” in .json files

With \\\” in straight queries:

So now the query looks like this:

Select \* from version where version = “our value” our payload

Now to make sure our everything after out payload doesn’t break our payload we’re gonna comment something (-)

Now u end up with this payload: union select 1,2,3,4-- -

Now that we confirmed that the ws server is vulnerable to sqli and where it’s vulnerable, there’s very beautiful script we could use to automate exploiting the ws server with sqlmap.

This is the script: <https://rayhan0x01.github.io/ctf/2021/04/02/blind-sqli-over-websocket-automation.html>

We have to use this script, since there’s no documentation about sqlmap supporting ws://

So what this script does summarized:

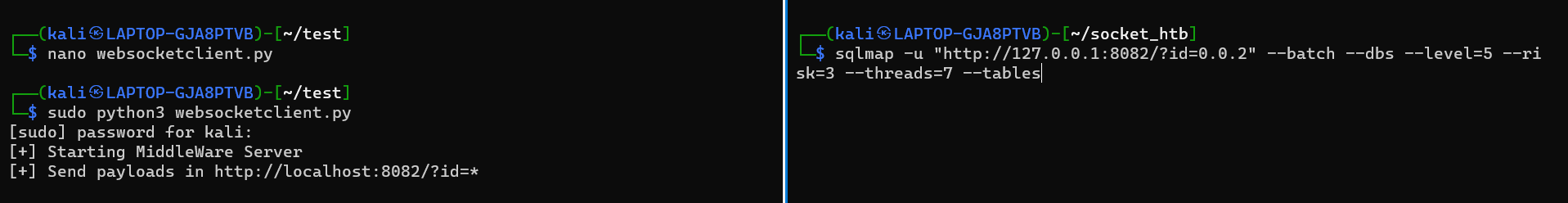
1. Defines a middleware python server on port 8081 for incoming http get requests containing an id parameter. This id parameter is gonna be used to inject sql with sqlmap.
2. The id parameter is then extracted from the query string and used to create a json message with an “EmployeeID” key-value pair.
3. The json message is sent to a websocket server at ws://ip\_address:port/dir
4. If the websocket returns a response, it is sent to the client as the http response body.

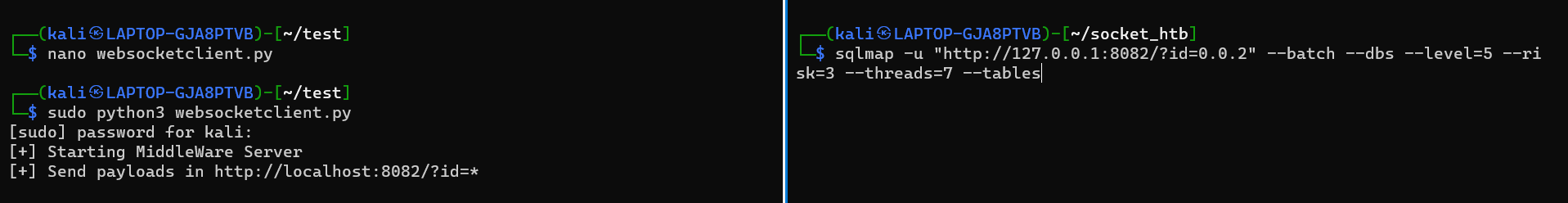
So in simple terms it turns the websocket server into an http server, which is easier to deal with with sqlmap.

Things we need to edit in the script:

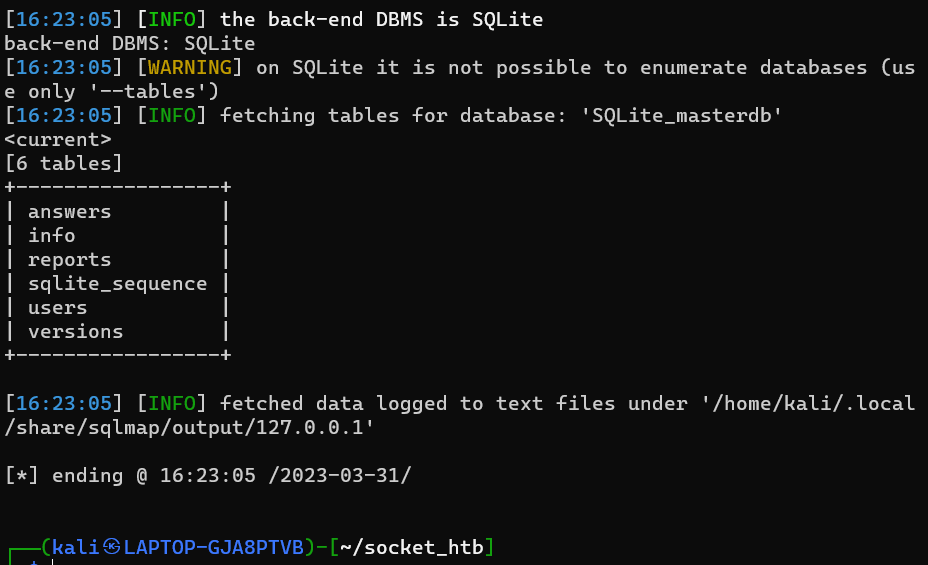
1. There is this line of code in the script: message = unquote(payload).replace('"','\''), which is used to replace all the “ in our payload (which is everything passed in id=) in single quotes, to not break the json structure. Why this I done, is because that’s how sqlmap injects sql code in http requests. It starts the payload with a single quote, to break whatever structure and execte sql code. Now we already discussed in the manual sqli, that we can not use single quotes, because it will just be seen as part of the version value. So we have to escape the double quotes. Somehow we need our payload to start with this: \\\” to break escape the double quotes. To do that we just have to replace the single quote that sqlmap uses to start a sqli, with the value: \\\”. This is easly done by just changing the code to this: unquote(payload).replace('"','\\\”').
2. There is a line of code in the script that assigns the variable data to the value: {“EmployeeID”:”%s”} % message. We need to change EmployeeID to version.
3. In the code the ws server url is assigned, we need to change that to ws://10.10.11.206:5789/version.

Now we can run the script, and run sqlmap on the middleware server, with the id as 0.0.2

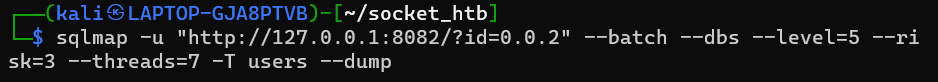




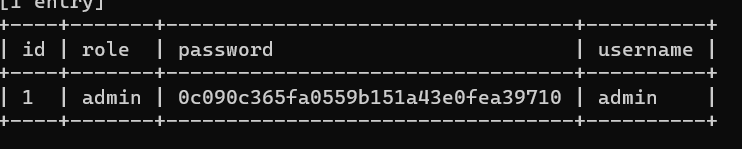
And these are the tables we got back:



Now lets dump all the data from the users table:

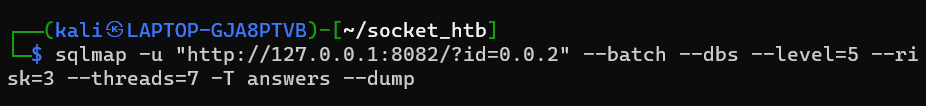


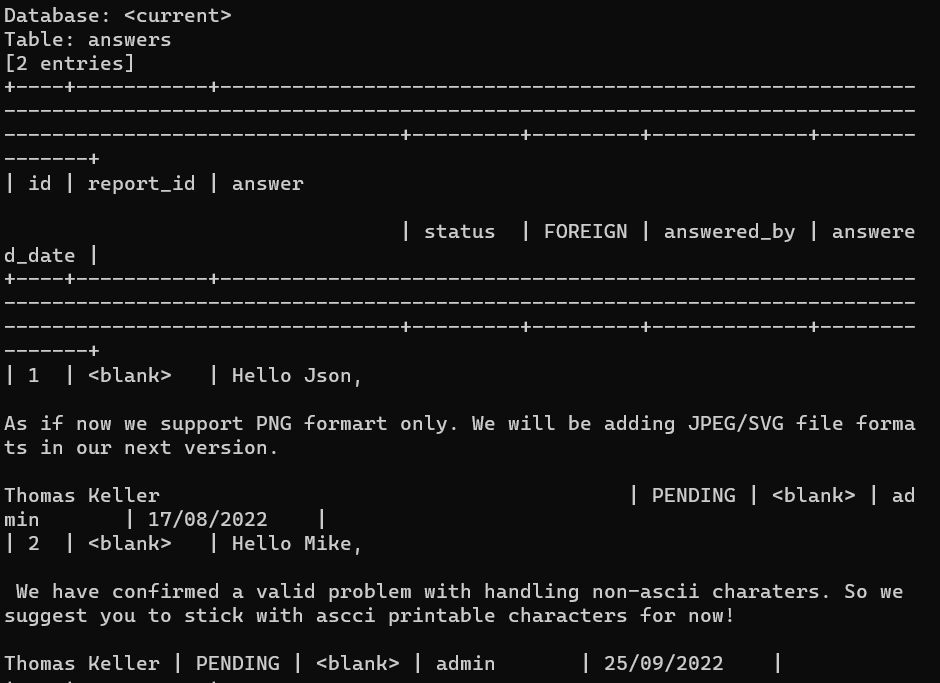
And we got a password hash without a real username:



Admin:0c090c365fa0559b151a43e0fea39710

We still need a valid username, so let’s enumerate the database, and dump everything until we find anything interesting:

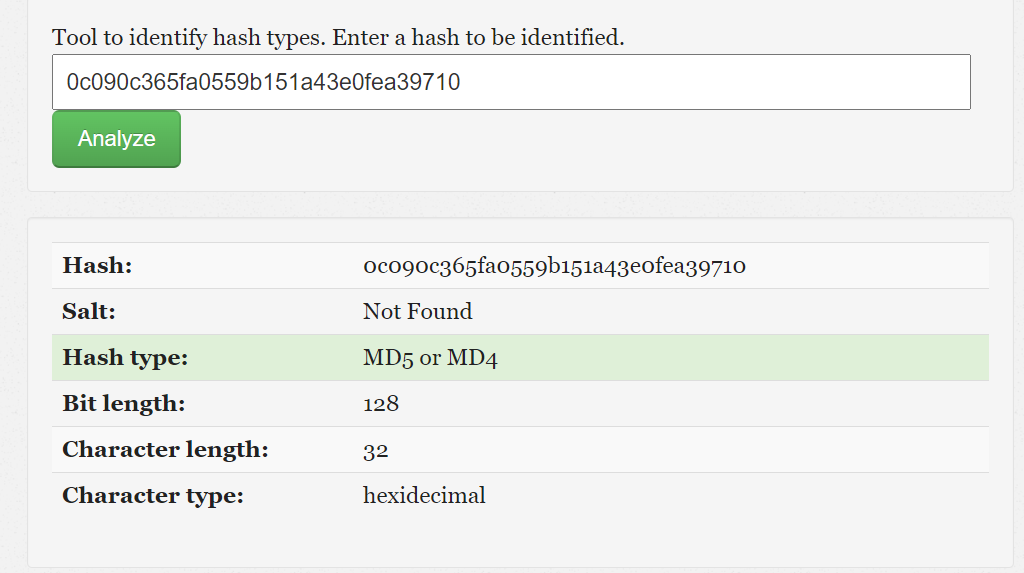




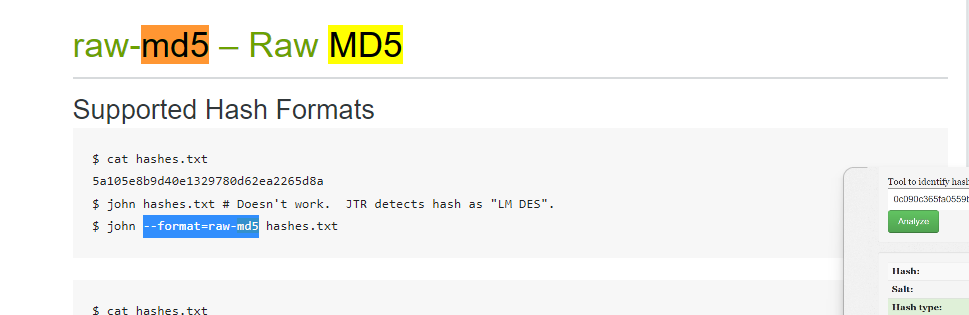
And here we’ve found a full name: Thomas Keller

Now it’s time to get the credentials:

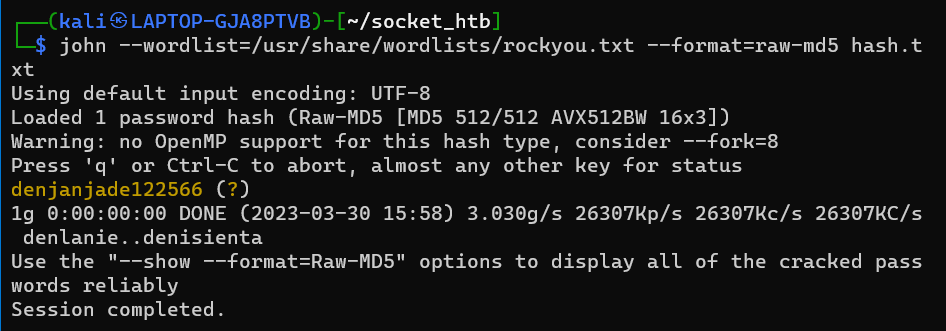
Before we start cracking the hash, lets see what type of hash it is. The source below tells us its either md5 or md4



Now lets find the supported hash formats for john the ripper.



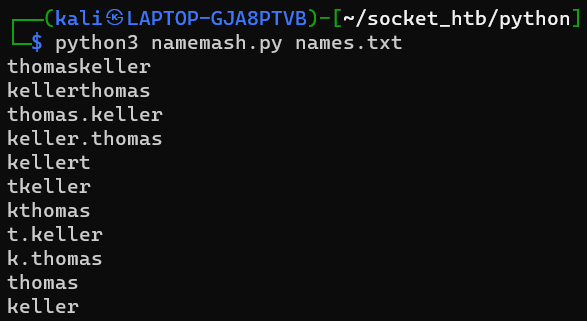
And now we can start cracking:



Password cracked: denjanjade122566

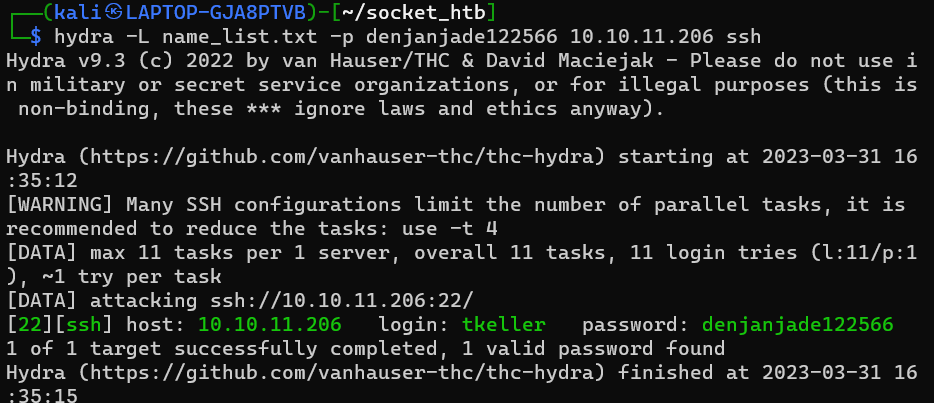
Now we need to get the username. We can use the python tool below to get a list of potential usernames based on the full name:





Now let’s put these usernames in a list, and start bruteforcing!

We’re going to use hydra, with the username list we made, and the password we cracked, to find the right username to login into the ssh server of the target:



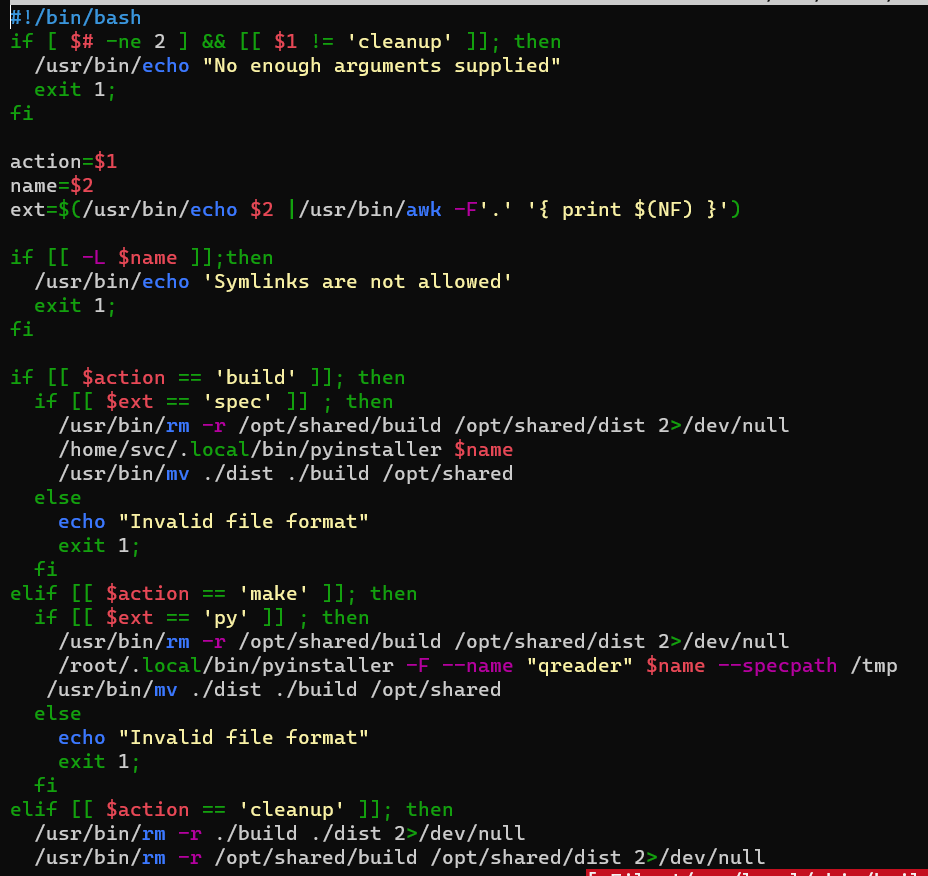
And the username is: tkeller

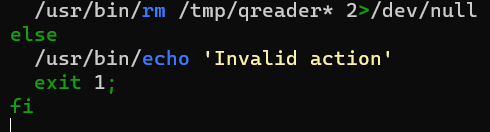
Now we can login into the ssh server:



Now it’s time for PrivEsc, before we upload anything lets try sudo -l:

(NOPASSWD) tkeller may run the following with sudo permissions: /usr/local/sbin/build-installer.sh





So this is what the code is doing:

1. It firstly checks if there 2 arguments passed, and that the first argument is NOT equal to cleanup it exists with the error: not enough arguments passed,
2. Then it assigns the variable ‘action’ to the value of the first argument, and the variable ‘name’ to the value of the second argument.
3. Then it check if the action variable is equal to build, and the extension of the name variable is equal to ‘spec’. if this is true, it then deletes the old build and dist directories in /opt/shared/. Any errors are directed to /dev/null

It then executes the name variable with the pyinstaller in root. And then it moves the dist and build directories from the current directory to the /opt/shared/ directory.

1. Then it check of the action variable is equal to make, and the extension of the name variable is equal to ‘py’. If this is true, it then deletes the old build and dist directories in /opt/shared. Anu error are throwen to /dev/null/ It then writes the code from the name variable file to the qreader file. And then it moves the dist and build directories from the current directory to the /opt/shared directory.
2. Then the last option is to pass only one argument, which is cleanup, this option cleans ups the mess by deletes files in

So obviously we’re gonna choose for the ‘build’ action, because it runs our file, while the ‘make’ action just outputs the contents of our spec file into the qreader file.

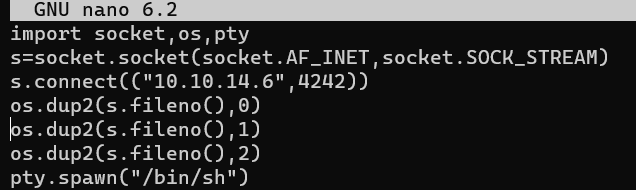
First of all we need to understand what a spec file is? A spec file tells the pyinstaller how to process your script. The spec file is actually executable python code. Pyinstaller build the app, by executing the contents of the spec file.

This means we could write some malicious python code in the file, and it gonna be executed with root privileges.

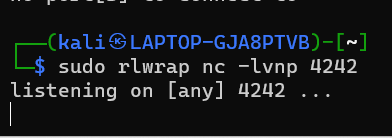
Lets create a new spec file:



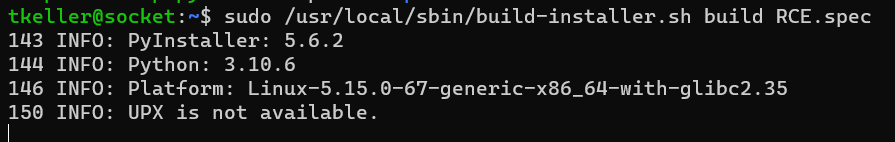
Write python code that creates a reverse shell back to our ip:



Create a listener on the right port



Execute the build-installer.sh script with sudo and build & our python file as 2 args



And here we are, we’ve got a shell, and we’re root!

