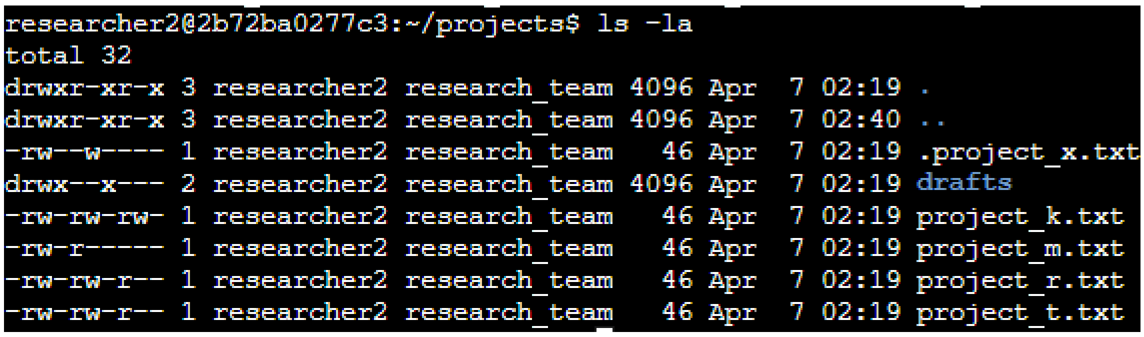
Project1 - File permissions in Linux

Project description

In an organization, directories, files, and documents must be securely protected. To minimize security risks, access to these archives should be restricted to specific authorized users. Allowing unrestricted access can lead to serious vulnerabilities. Linux is one of the key tools used by cybersecurity analysts to manage file permissions, as it offers powerful commands to view, assign, and modify access rights. This helps ensure the integrity and confidentiality of critical data.

Check file and directory details

The image below shows the command I used in Linux to display the types of permissions assigned to the files in the directory.



```
researcher2@2b72ba0277c3:~/projects$ ls -la
total 32
drwxr-xr-x 3 researcher2 research_team 4096 Apr  7 02:19 .
drwxr-xr-x 3 researcher2 research_team 4096 Apr  7 02:40 ..
-rw--w---- 1 researcher2 research_team   46 Apr  7 02:19 .project_x.txt
drwx--x--- 2 researcher2 research_team 4096 Apr  7 02:19 drafts
-rw-rw-rw- 1 researcher2 research_team   46 Apr  7 02:19 project_k.txt
-rw-r----- 1 researcher2 research_team   46 Apr  7 02:19 project_m.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr  7 02:19 project_r.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr  7 02:19 project_t.txt
```

In the screenshot above, the command **ls -la** is used in the first line. The output displays a list of files and their associated permissions within the project directory. Among the contents, there is a directory named **drafts** and a hidden file called **.project_x.txt**. The 10-character sequence at the beginning of each line provides detailed information about the permissions set for each file or directory.

Describe the permissions string

In Linux, there are three basic types of permissions: **read (r)**, **write (w)**, and **execute (x)**. These permissions are represented in a 10-character string, such as **drwxrwxrwx**. The first character indicates the file type, that means **d** for a directory file and **-** for a regular file. The remaining nine characters are divided into three groups of three, representing permissions for the **user (owner)**, **group**, and **others**, respectively.

Each group of three characters indicates which permissions are granted:

- **r** stands for read
- **w** stands for write
- **x** stands for execute
- **-** indicates that a permission is not granted

Here's a breakdown of the lines in the screenshot based on this format:

Lines 1 and 2:
These entries are directories (indicated by **d**).

- **User** has **read, write, and execute** permissions (**rwx**)
- **Group** and **others** have **read and execute** permissions (**r-x**) but **no write** permission (**-**)

Line 3:
This is a regular file (indicated by hyphen **"-"**).

- **User** has **read and write** permissions (**rw-**)
- **Group** has only **write** permission (**-w-**)
- **Others** have **no permissions** (**---**)

Line 4:
This is a directory file (**d**).

- **User** has **read, write, and execute** permissions (**rwx**)
- **Group** has only **execute** permission (**--x**)
- **Others** have **no permissions** (**---**)

Line 5:
This is a regular file (**-**).

- **User**, **group**, and **others** all have **read and write** permissions (**rw-**)
- None have **execute** permissions

Line 6:
This is a regular file (**-**).

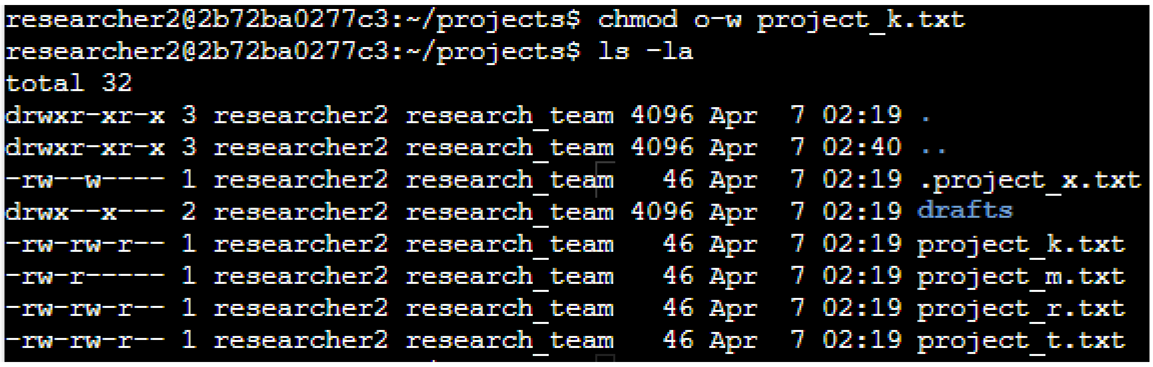
- **User** has **read and write** permissions (**rw-**)
- **Group** has only **read** permission (**r--**)
- **Others** have **no permissions** (**---**)

Lines 7 and 8:
These are regular files (**-**).

- **User** and **group** have **read and write** permissions (**rw-**)
- **Others** have only **read** permission (**r--**)
- No execute permissions are granted to any group

Change file permissions

Updating file permissions is crucial for maintaining an organization's security. It ensures consistent control over who can access the file system and helps prevent data loss or unauthorized modifications. The image below demonstrates the command I used to restrict write access for others on the file **.project_k.txt.**

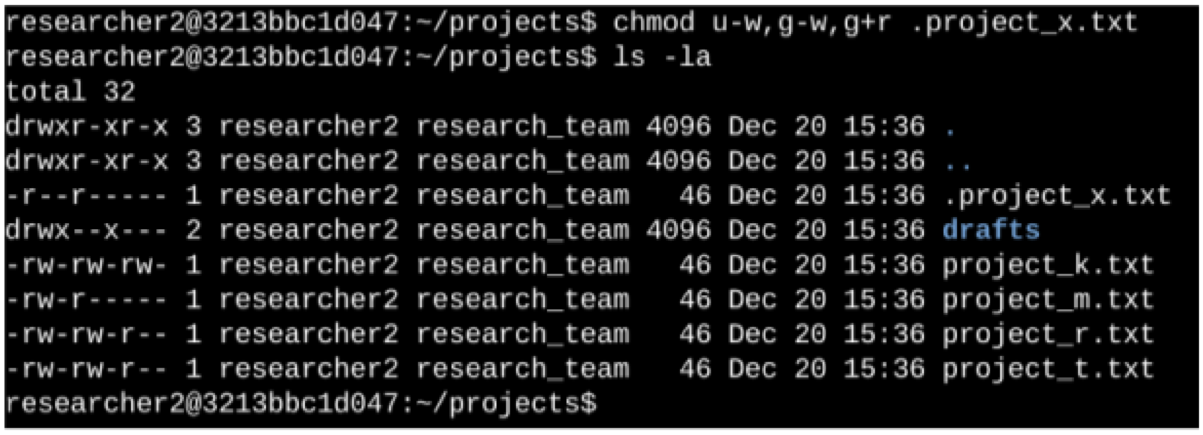


```
researcher2@2b72ba0277c3:~/projects$ chmod o-w project_k.txt
researcher2@2b72ba0277c3:~/projects$ ls -la
total 32
drwxr-xr-x 3 researcher2 research_team 4096 Apr  7 02:19 .
drwxr-xr-x 3 researcher2 research_team 4096 Apr  7 02:40 ..
-rw--w---- 1 researcher2 research_team   46 Apr  7 02:19 .project_x.txt
drwx--x--- 2 researcher2 research_team 4096 Apr  7 02:19 drafts
-rw-rw-r-- 1 researcher2 research_team   46 Apr  7 02:19 project_k.txt
-rw-r----- 1 researcher2 research_team   46 Apr  7 02:19 project_m.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr  7 02:19 project_r.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr  7 02:19 project_t.txt
```

The first two lines in the screenshot show the commands I entered, while the remaining lines display the output of the second command. The **chmod** command is used to modify permissions on files and directories. The first argument defines which permissions to change, and the second specifies the target file or directory. In this case, I removed the write permission for "others" on the **project_k.txt** file. I then used **ls -la** to verify the updated permissions.

Change file permissions on a hidden file

The research team at my organization recently archived **project_x.txt**. To protect the file, they want to ensure that no one has write access, while still allowing the user and group to have read access. The following code shows how I used Linux commands to update the file permissions accordingly:



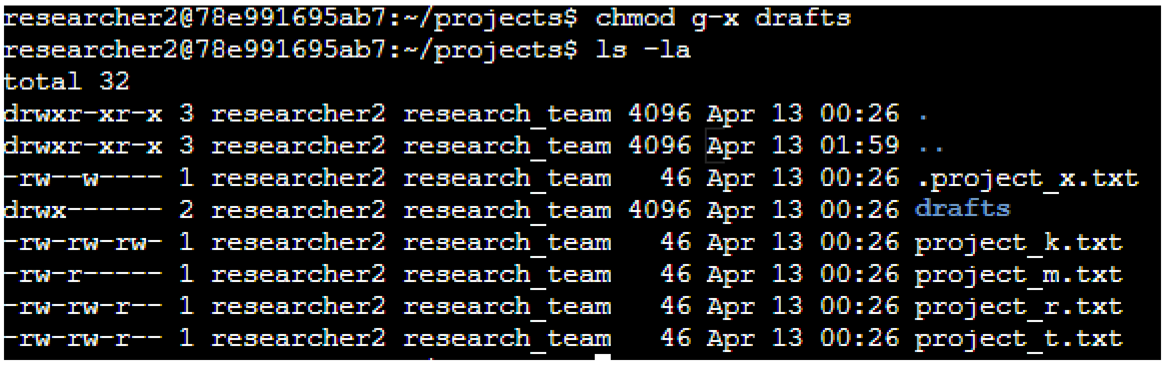
```
researcher2@3213bbc1d047:~/projects$ chmod u-w,g-w,g+r .project_x.txt
researcher2@3213bbc1d047:~/projects$ ls -la
total 32
drwxr-xr-x 3 researcher2 research_team 4096 Dec 20 15:36 .
drwxr-xr-x 3 researcher2 research_team 4096 Dec 20 15:36 ..
-r--r----- 1 researcher2 research_team   46 Dec 20 15:36 .project_x.txt
drwx--x--- 2 researcher2 research_team 4096 Dec 20 15:36 drafts
-rw-rw-rw- 1 researcher2 research_team   46 Dec 20 15:36 project_k.txt
-rw-r----- 1 researcher2 research_team   46 Dec 20 15:36 project_m.txt
-rw-rw-r-- 1 researcher2 research_team   46 Dec 20 15:36 project_r.txt
-rw-rw-r-- 1 researcher2 research_team   46 Dec 20 15:36 project_t.txt
researcher2@3213bbc1d047:~/projects$
```

In Linux, hidden files always begin with a dot (.). To modify the permissions of the hidden file **.project_x.txt**, I used the following command: **chmod u-w, g-w, g+r .project_x.txt.** The **chmod** command is used to change file and directory permissions. In this case, **u-w** removes write permission from the users, **g-w** removes write permission from the group, and **g+r** grants read permission to the group.

This ensures that the **.project_x.txt** file is protected from modification, while still allowing the group to read its contents.

Change directory permissions

For the **drafts** directory, my company decided to grant access exclusively to **researcher2**. This means that only the owner should have execute permission.
The following command demonstrates how I used Linux to update the directory's permissions accordingly:



```
researcher2@78e991695ab7:~/projects$ chmod g-x drafts
researcher2@78e991695ab7:~/projects$ ls -la
total 32
drwxr-xr-x 3 researcher2 research_team 4096 Apr 13 00:26 .
drwxr-xr-x 3 researcher2 research_team 4096 Apr 13 01:59 ..
-rw--w---- 1 researcher2 research_team   46 Apr 13 00:26 .project_x.txt
drwx------ 2 researcher2 research_team 4096 Apr 13 00:26 drafts
-rw-rw-rw- 1 researcher2 research_team   46 Apr 13 00:26 project_k.txt
-rw-r----- 1 researcher2 research_team   46 Apr 13 00:26 project_m.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr 13 00:26 project_r.txt
-rw-rw-r-- 1 researcher2 research_team   46 Apr 13 00:26 project_t.txt
```

To change the permissions for the **drafts** directory, I used the following command: **chmod g-x drafts**. The **chmod** command is used to modify permissions on files and directories. In this case, **g** refers to the group category and **-x** removes execute permission.

This command ensures that the group no longer has execute access to the **drafts** directory.

Summary

As a security analyst, managing directory and file permissions is often a key responsibility. The **ls -la** command allows you to inspect detailed permission settings for files and directories. The **chmod** command enables you to modify these permissions, helping ensure they align with the principle of least privilege.

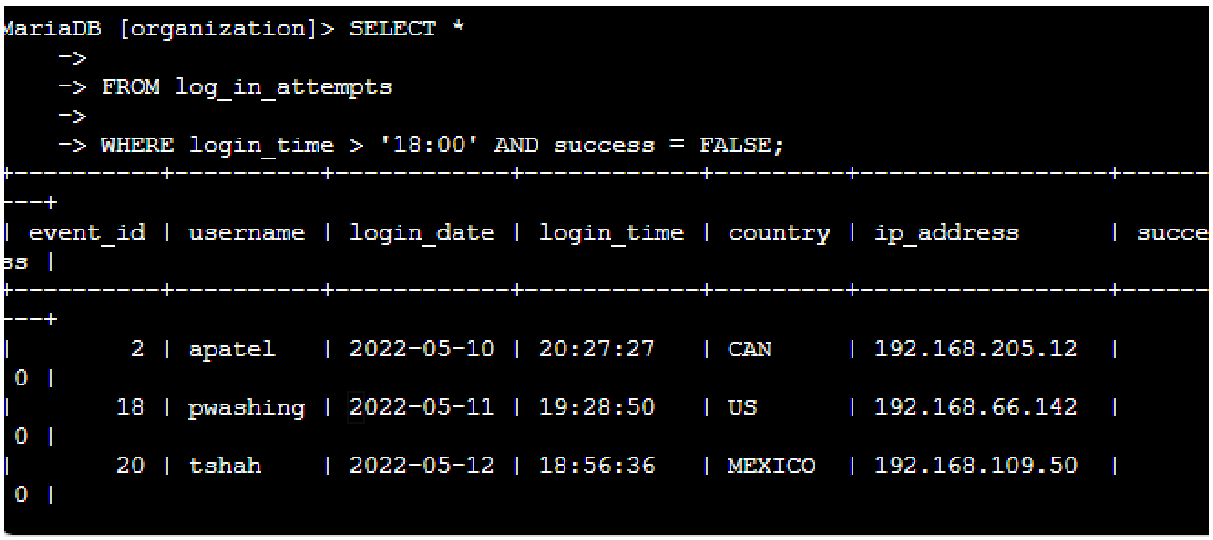
Project 2 - Apply filters to SQL queries

Project description

Structured Query Language (SQL) is one of the most common programming languages used in data analysis to create, interact with, and query information from a database, as it is widely supported by a variety of data products. In this project, I will be using some SQL filters using Linux Bash Shell to retrieve records from different datasets and investigate potential security risks.

Retrieve after hours failed login attempts

Security incidents often occur outside of regular business hours, as hackers frequently target these times to attempt attacks on an organization's servers and extract sensitive information. As a result, organizations routinely investigate login activity during off-hours.
The image below shows an SQL query used to retrieve all failed login attempts that occurred after business hours.

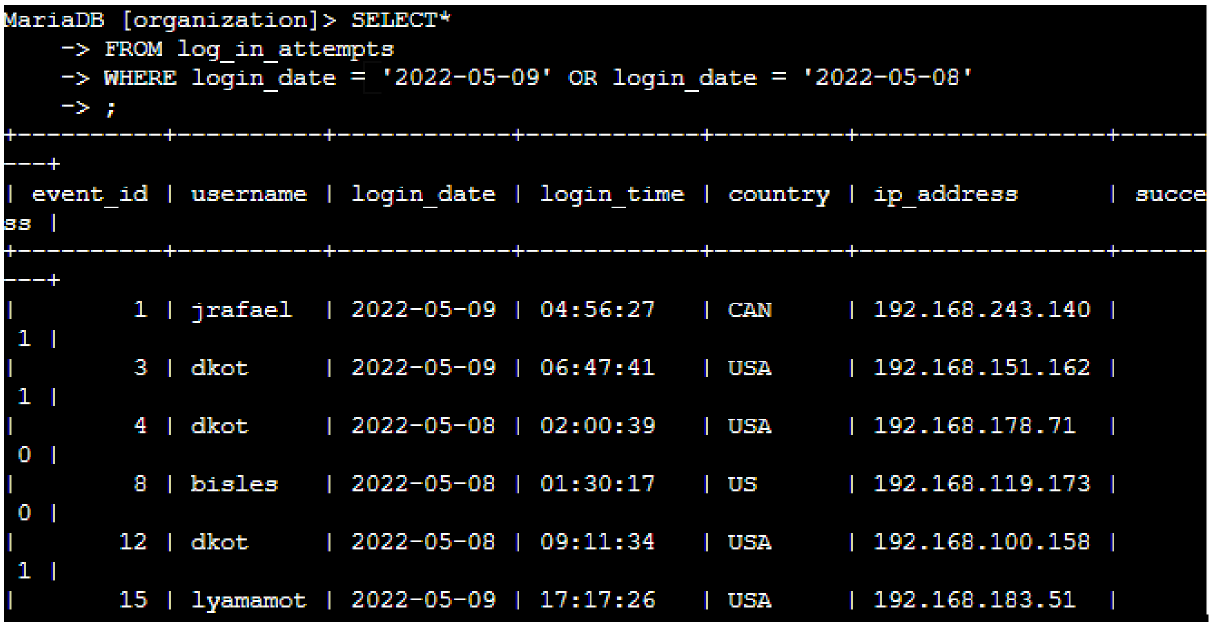


```
lariaDB [organization]> SELECT *
    ->
    -> FROM log_in_attempts
    ->
    -> WHERE login_time > '18:00' AND success = FALSE;
+----------+----------+------------+------------+---------+-----------------+------
---+
| event_id | username | login_date | login_time | country | ip_address      | succe
ss |
+----------+----------+------------+------------+---------+-----------------+------
---+
|        2 | apatel   | 2022-05-10 | 20:27:27   | CAN     | 192.168.205.12  |
 0 |
|       18 | pwashing | 2022-05-11 | 19:28:50   | US      | 192.168.66.142  |
 0 |
|       20 | tshah    | 2022-05-12 | 18:56:36   | MEXICO  | 192.168.109.50  |
 0 |
```

To investigate all unsuccessful login attempts after business hours, I used the following coding: **SELECT***, this command with an asterisk, displays all columns of a table. **FROM log_in_attempts**, this command line specifies the name of the table from which I wanted to retrieve my information. **WHERE login_time > '18:00' AND success = FALSE**, in these sequences, I queried all logins made after 18:00 that were not successful.

Retrieve login attempts on specific dates

Sometimes, it's not necessary to examine the entire dataset—only a specific portion may be relevant for analysis. To focus on particular data, we can specify the exact information we want to retrieve.
The screenshot below displays selected entries from the **login_time** column in the **log_in_attempts** table, which I am analyzing as part of the investigation.

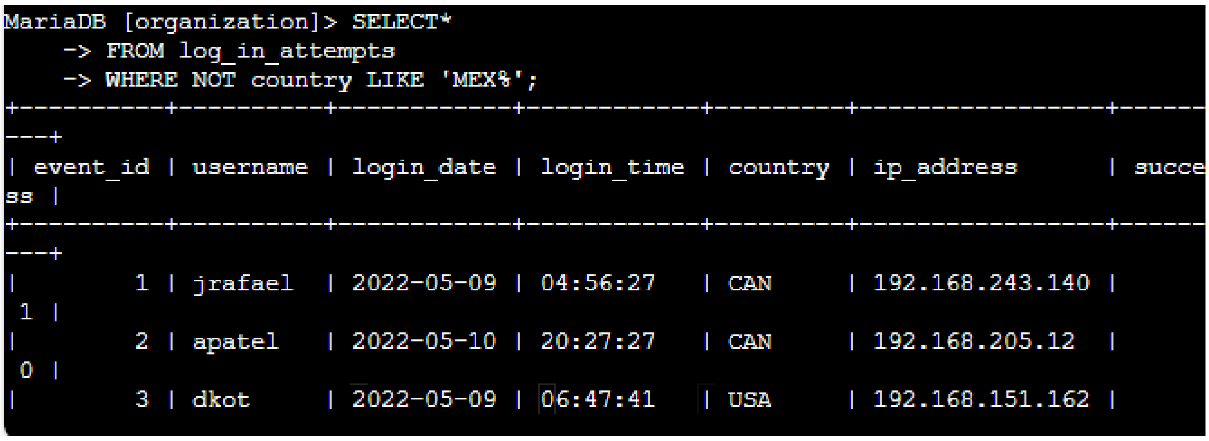


```
MariaDB [organization]> SELECT*
    -> FROM log_in_attempts
    -> WHERE login_date = '2022-05-09' OR login_date = '2022-05-08'
    -> ;
+----------+----------+------------+------------+---------+-----------------+------
---+
| event_id | username | login_date | login_time | country | ip_address      | succe
ss |
+----------+----------+------------+------------+---------+-----------------+------
---+
|        1 | jrafael  | 2022-05-09 | 04:56:27   | CAN     | 192.168.243.140 |
 1 |
|        3 | dkot     | 2022-05-09 | 06:47:41   | USA     | 192.168.151.162 |
 1 |
|        4 | dkot     | 2022-05-08 | 02:00:39   | USA     | 192.168.178.71  |
 0 |
|        8 | bisles   | 2022-05-08 | 01:30:17   | US      | 192.168.119.173 |
 0 |
|       12 | dkot     | 2022-05-08 | 09:11:34   | USA     | 192.168.100.158 |
 1 |
|       15 | lyamamot | 2022-05-09 | 17:17:26   | USA     | 192.168.183.51  |
```

Initially, I used **SELECT *** to display all columns from the table. In the next line, I specified the data source using **FROM log_in_attempts**. Then, in the third line, I applied the WHERE clause with the **OR** operator to filter and display records from the **login_time** column for either **05/09/2022** or **05/08/2022**.

Retrieve login attempts outside of Mexico

When a security incident or suspicious activity takes place, one of the first questions we ask is, “Where did the event occur?” Identifying the location helps narrow the investigation to a specific region.
The illustration below shows the SQL query I used to display all login attempts that did not originate from Mexico.

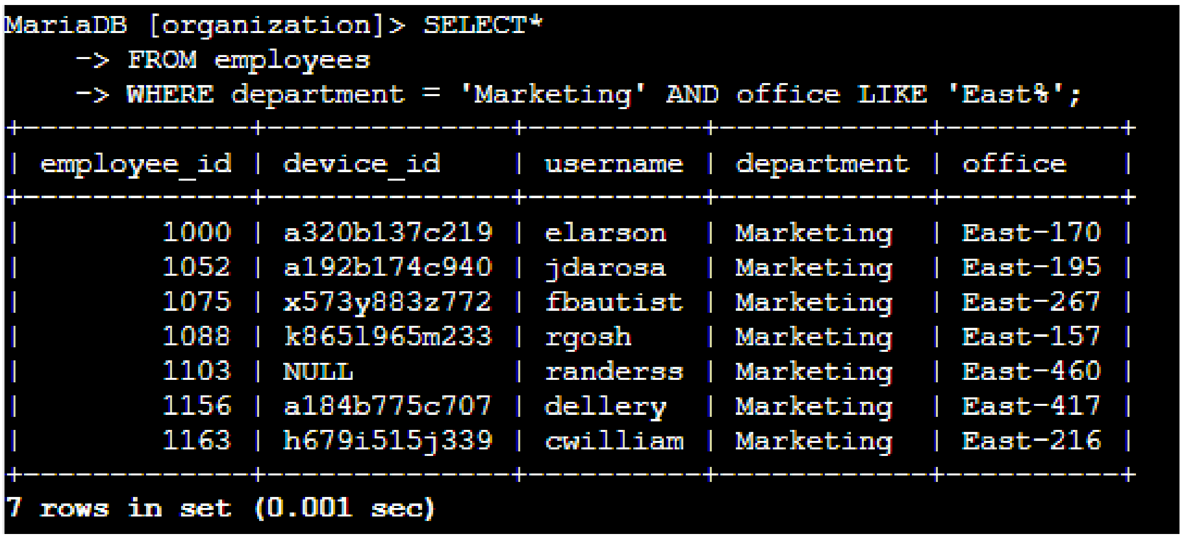


```
MariaDB [organization]> SELECT*
    -> FROM log_in_attempts
    -> WHERE NOT country LIKE 'MEX%';
+----------+----------+------------+------------+---------+-----------------+------
---+
| event_id | username | login_date | login_time | country | ip_address      | succe
ss |
+----------+----------+------------+------------+---------+-----------------+------
---+
|        1 | jrafael  | 2022-05-09 | 04:56:27   | CAN     | 192.168.243.140 |
 1 |
|        2 | apatel   | 2022-05-10 | 20:27:27   | CAN     | 192.168.205.12  |
 0 |
|        3 | dkot     | 2022-05-09 | 06:47:41   | USA     | 192.168.151.162 |
```

In the first line, I used the **SELECT *** command to display all columns from the table in the output. In the second line, I called the table using **FROM log_in_attempts**. In the third line, I applied the **WHERE** clause with **NOT** to filter and display all suspicious login attempts that did **not** originate from Mexico. Since the dataset may represent Mexico in different formats (e.g., "MEX" or "MEXICO"), I used the **LIKE** clause with **%MEX** to capture both variations. The percentage symbol (**%**) acts as a wildcard, representing any number of unspecified characters when used with **LIKE**.

Retrieve employees in Marketing

Organizations often consist of multiple departments, but sometimes the information you need relates to just one of them. To retrieve specific data, you can apply a filter to display only the relevant records.
For example, the image below shows a SQL query I used to filter and display **employees** who work in the **Marketing** department and are located in the **East Building** office.



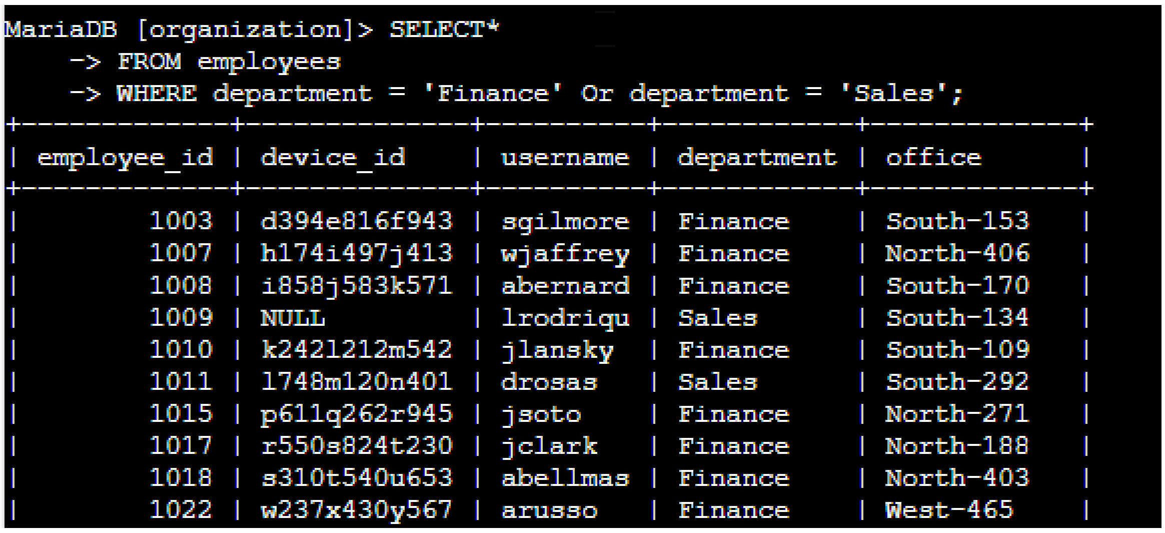
```
MariaDB [organization]> SELECT*
    -> FROM employees
    -> WHERE department = 'Marketing' AND office LIKE 'East%';
+-------------+---------------+----------+------------+----------+
| employee_id | device_id     | username | department | office   |
+-------------+---------------+----------+------------+----------+
|        1000 | a320b137c219  | elarson  | Marketing  | East-170 |
|        1052 | a192b174c940  | jdarosa  | Marketing  | East-195 |
|        1075 | x573y883z772  | fbautist | Marketing  | East-267 |
|        1088 | k8651965m233  | rgosh    | Marketing  | East-157 |
|        1103 | NULL          | randerss | Marketing  | East-460 |
|        1156 | a184b775c707  | dellery  | Marketing  | East-417 |
|        1163 | h679i515j339  | cwilliam | Marketing  | East-216 |
+-------------+---------------+----------+------------+----------+
7 rows in set (0.001 sec)
```

To retrieve a list of employees from the Marketing department, I began with the **SELECT *** command to display all columns from the **employees** table, specified using **FROM employees**. Next, I applied the **WHERE** clause with the **AND** operator to filter the data for employees who work in the **Marketing** department and are located in the **East Building**. The condition **department = 'Marketing'** filters employees by department, and the **LIKE 'East%'** clause is used to match all office locations that begin with "East", effectively identifying those in the East Building.

Retrieve employees in Finance or Sales

SQL filter commands are highly valuable because they produce precise and targeted results. They are commonly used to perform quick queries and retrieve specific information.

The illustration below demonstrates the filter command I used to display all employees who belong to either the **Finance** or **Sales** department.



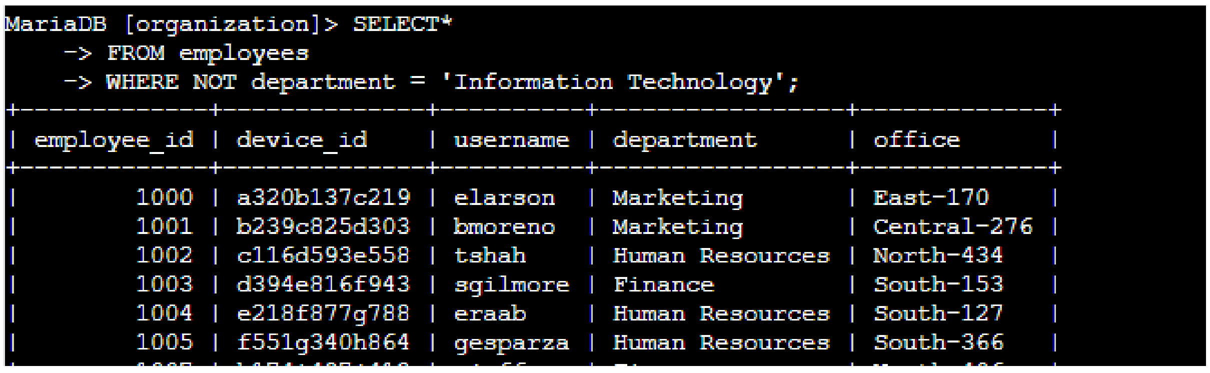
```
MariaDB [organization]> SELECT*
    -> FROM employees
    -> WHERE department = 'Finance' Or department = 'Sales';
+-------------+---------------+----------+------------+-------------+
| employee_id | device_id     | username | department | office      |
+-------------+---------------+----------+------------+-------------+
|        1003 | d394e816f943  | sgilmore | Finance    | South-153   |
|        1007 | h174i497j413  | wjaffrey | Finance    | North-406   |
|        1008 | i858j583k571  | abernard | Finance    | South-170   |
|        1009 | NULL          | lrodriqu | Sales      | South-134   |
|        1010 | k2421212m542  | jlansky  | Finance    | South-109   |
|        1011 | 1748m120n401  | drosas   | Sales      | South-292   |
|        1015 | p611q262r945  | jsoto    | Finance    | North-271   |
|        1017 | r550s824t230  | jclark   | Finance    | North-188   |
|        1018 | s310t540u653  | abellmas | Finance    | North-403   |
|        1022 | w237x430y567  | arusso   | Finance    | West-465    |
```

I used the SELECT * command to retrieve all columns from the **employees** table, specified by the **FROM employees** clause. Then, I applied the **WHERE** clause with the **OR** operator to filter and display employees who work in either the **Finance** or **Sales** department. The condition **department = 'Finance'** filters employees in the Finance department, while **department = 'Sales'** filters those in the Sales department. If either condition is true, the corresponding employee records are included in the results.

Retrieve all employees not in IT

Depending on the specific information you need from a table, there are various ways to retrieve it. For instance, the **WHERE** clause can be used with different operators to filter data according to your requirements.

Refer to the screenshot below for another example of how command filters can be used effectively.



```
MariaDB [organization]> SELECT*
    -> FROM employees
    -> WHERE NOT department = 'Information Technology';
+-------------+---------------+----------+-----------------+-------------+
| employee_id | device_id     | username | department      | office      |
+-------------+---------------+----------+-----------------+-------------+
|        1000 | a320b137c219  | elarson  | Marketing       | East-170    |
|        1001 | b239c825d303  | bmoreno  | Marketing       | Central-276 |
|        1002 | c116d593e558  | tshah    | Human Resources | North-434   |
|        1003 | d394e816f943  | sgilmore | Finance         | South-153   |
|        1004 | e218f877g788  | eraab    | Human Resources | South-127   |
|        1005 | f551g340h864  | gesparza | Human Resources | South-366   |
```

To retrieve employees who are not part of the IT department, I began by using **SELECT *** in the first line to display all columns from the **employee** table. In the second line, I specified the table using the **FROM** employees. In the final line, I applied the **WHERE** clause with the **NOT** operator to exclude employees from the IT department. The condition **department = 'Information Technology'** identifies IT employees, and **NOT** ensures that only those outside this department are included in the results.

Summary

In this project, I applied various filters to retrieve information from the **log_in_attempts** and **employees** tables. I used the **WHERE** clause in combination with different types of operators, including Boolean values like **FALSE**, logical operators such as **OR** and **NOT**, and comparison operators like > and =. Additionally, I used the **LIKE** keyword along with the **%** wildcard to filter data based on specific patterns.

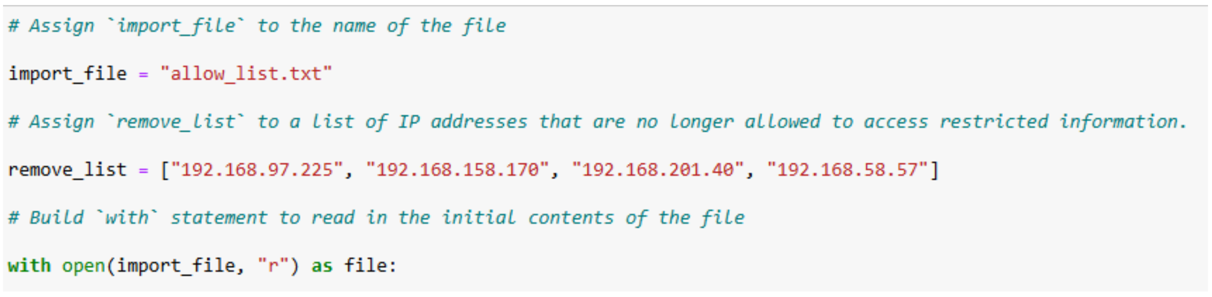
Project 3 - Algorithm for file updates in Python

Project description

One of the key responsibilities of a security analyst is to safeguard an organization’s data. This involves continuously auditing and analyzing data to defend against potential threats and attacks. A crucial aspect of data protection is controlling access, as unauthorized access to sensitive information can severely damage an organization’s reputation. Parsing files enables security analysts to read and modify their contents as needed. Python is a valuable tool in this process, allowing analysts to create algorithms that automate file analysis and ensure data remains current and secure.

Open the file that contains the allow list

The screenshot below shows the Python code I used to open and access the contents of a file named **allow_list.txt**.



```python
# Assign `import_file` to the name of the file

import_file = "allow_list.txt"

# Assign `remove_list` to a list of IP addresses that are no longer allowed to access restricted information.

remove_list = ["192.168.97.225", "192.168.158.170", "192.168.201.40", "192.168.58.57"]

# Build `with` statement to read in the initial contents of the file

with open(import_file, "r") as file:
```

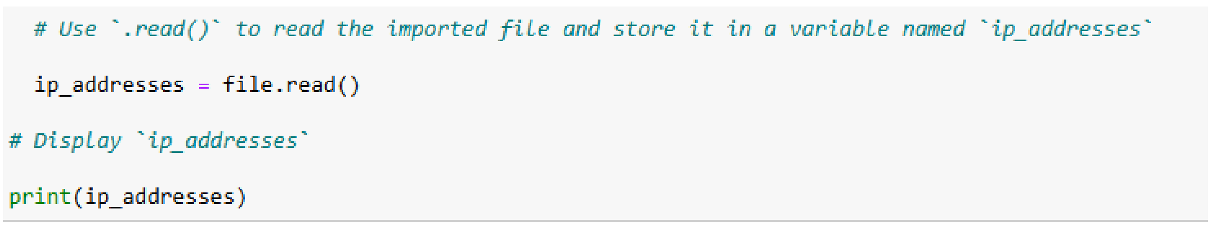
In the first line of the program, I assigned the file **allow_list.txt** to a variable named **import_file**. In the second line, I created a list of IP addresses and stored it in the variable **remove_list**, which I will use later.

And On the third line, I used the **with** keyword along with the **open()** function and the **as** keyword to handle the file. The **with** statement is useful for managing external resources and handling errors automatically. In this case, it's used to open the file. The first argument passed to **open()** is the file name, which I stored in the **import_file** variable, and the second argument, **"r"**, specifies that the file should be opened in read mode.

The **as** keyword assigns the opened file to a variable named **file**, allowing us to reference it within the indented block.

Read the file contents

The image below illustrates the Python code used to read data from the **allow_list.txt** file.

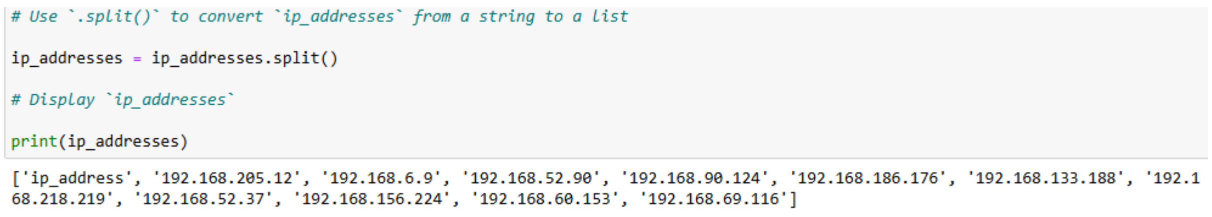


```
  # Use `.read()` to read the imported file and store it in a variable named `ip_addresses`

  ip_addresses = file.read()

# Display `ip_addresses`

print(ip_addresses)
```

I used **file.read()** to read the contents of the file and stored the result in a variable called **ip_addresses**. The **.read()** method converts the file content into a string. Finally, I used **print(ip_addresses)** to display the file contents.

Convert the string into a list

The Illustration demonstrates the python code that I used to convert **ip_addresses** from a string to a list.



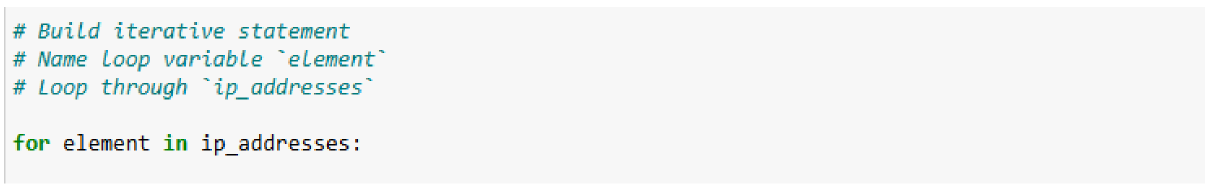
```
# Use `.split()` to convert `ip_addresses` from a string to a list

ip_addresses = ip_addresses.split()

# Display `ip_addresses`

print(ip_addresses)
```

```
['ip_address', '192.168.205.12', '192.168.6.9', '192.168.52.90', '192.168.90.124', '192.168.186.176', '192.168.133.188', '192.1
68.218.219', '192.168.52.37', '192.168.156.224', '192.168.60.153', '192.168.69.116']
```

To convert **ip_addresses** from a string to a list, I used the code **ip_addresses.split()** and stored the result in the same variable **ip_addresses**. The **.split()** method breaks a string into a list, and in this case, it uses a comma to separate each element.

Iterate through the remove list

In programming, iterative statements are especially useful when you want to display a range of elements. The **for loop** is commonly used for this purpose. The example below demonstrates the code I used to create an iterative statement that verifies each element of **ip_addresses**.

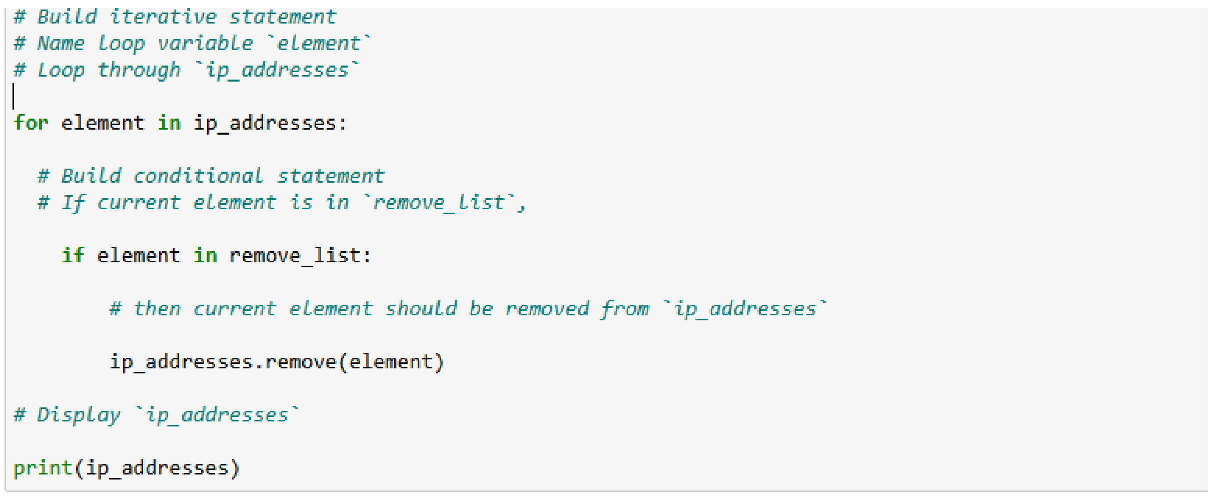


```
# Build iterative statement
# Name loop variable `element`
# Loop through `ip_addresses`

for element in ip_addresses:
```

To create the iterative statement, I used the following code: **for element in ip_addresses:**. The keyword **for** indicates the start of a for loop. The variable **element** serves as the **loop variable**, controlling each iteration. The in operator is used before the sequence to tell Python to execute the loop once for each item in the **ip_addresses** list.

Remove IP addresses that are on the remove list

The screenshot below shows the combination of an iterative statement and a conditional statement that I used to remove IP addresses found in the removal list.

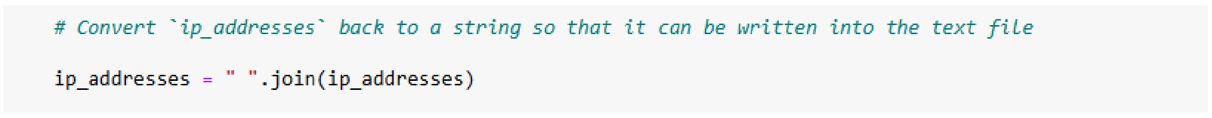


```
# Build iterative statement
# Name loop variable `element`
# Loop through `ip_addresses`

for element in ip_addresses:

  # Build conditional statement
  # If current element is in `remove_list`,

    if element in remove_list:

        # then current element should be removed from `ip_addresses`

        ip_addresses.remove(element)

# Display `ip_addresses`

print(ip_addresses)
```

After the iterative statement, I added a conditional statement using the following code: **if element in remove_list:**. In an **if** statement, the keyword **if** is followed by a condition—in this case, checking whether the current **element** exists in the **remove_list**. If the condition is true, the element is removed from **ip_addresses** using **ip_addresses.remove(element)**.

Update the file with the revised list of IP addresses

As the final step in my algorithm, I needed to update the allow list file with the revised list of IP addresses. To do this, I first converted the list back into a string using the **.join()** method:



```
# Convert `ip_addresses` back to a string so that it can be written into the text file

ip_addresses = " ".join(ip_addresses)
```

The **.join()** method combines all the items in an iterable into a single string. It is called on a string that specifies the separator to be used between the elements. In this algorithm, I used **.join()** to convert the **ip_addresses** list into a string, which I then passed as an argument to the **.write()** method when updating the "allow_list.txt" file. I used **"\n"** as the separator to ensure that each IP address appears on a new line.

Next, I used another **with** statement along with the **.write()** method to update the file:

```
# Build `with` statement to rewrite the original file

with open(import_file, "w") as file:

    # Rewrite the file, replacing its contents with `ip_addresses`

    file.write(ip_addresses)
```

This time, I used **"w"**as the second argument in the **open()** function within my **with** statement. This argument tells Python to open the file in write mode, meaning any existing content will be overwritten. With the file open in this mode, I used the **.write()** method to write string data to the file, replacing its previous contents.

In this case, I wanted to write the updated allow list as a string to the file **"allow_list.txt"**. This ensures that any IP addresses removed from the list will no longer have access to the restricted content. To overwrite the file, I used the **.write()** method on the file object defined in the **with** statement. I passed the **ip_addresses** variable as the argument, so the file's contents would be replaced with the updated data from that variable.

Summary

I developed an algorithm to remove IP addresses listed in the **remove_list** variable from the **"allow_list.txt"** file, which contains approved IP addresses. The algorithm begins by opening the file and reading its contents as a string. This string is then converted into a list and stored in the **ip_addresses** variable. Next, I iterated through each IP address in **remove_list**. For each one, I checked whether it existed in the ip_addresses list. If it did, I used the .remove() method to delete it from the list. After completing the removals, I used the **.join()** method to convert the updated **ip_addresses** list back into a string. Finally, I overwrote the contents of **"allow_list.txt"** with this updated string to reflect the revised list of approved IP addresses.