WebSecurity Project

Nicola Iommazzo

September 23, 2022

Contents

Abstract

This report is related to the *WebSecurity and Privacy* course, from the *Engineering in Computer Science* Master Degree, in *La Sapienza*. I will consider two well known vulnerabilities exposed in the **Django framework** in its version **4.0**, explaining what they are, how they work, how can they be exploited, and how they have been patched. At the end I will also run a minimal setup on Docker that will prove directly how to lead an attack. These vulnerabilities have been fixed with later security patches, so they can not be exploited in anymore, aside from using a not *up-to-date* version of the framework.

1 Django Framework

*"**Django** is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source."*

This is the descrition we read on the homepage of the Django website, and it is true. It is a really simple to create a very basic web application by using this framework. Instead of handling directly low level every server mechanism on python , the Django framework will do it by itself. Companies as Instagram, Spotify, Mozilla and National Geographic used this framework to develop completely, or just a part, of their application. Just to list some advantages of this framework, it is **fast** and **simple** in terms of pluggability and reusability, letting the programmer to use the same codebase for different purposes; the **python dependability** is optimal for code readability and simplicity; it is **comprehensive** of the majority tools needed for HTTP libraries, authentication, multi-site support, Django ORM for database interaction, and more; being an open souce framework it is constantly updated with **security** patches that fix newly discovered vulnerabilities; **scalability** is provided by clustering or load-balancing.

Django uses a particular version of the Model-View-Controller architecture, called **Model-Template-Controller (MTC)**, really similar to the first one. A standard Django web application, indeed, groups the code in different files.

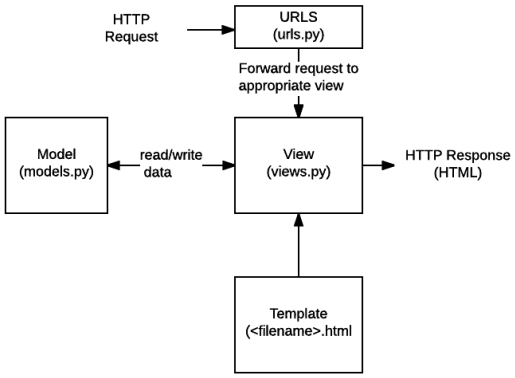


Figure 1: Django Model-Template-Controller Architecture

- **URLs**: A URL mapper is used to redirect HTTP requests to the appropriate view based on the request URL.
- **View**: A view is a request handler function, which receives HTTP requests and returns HTTP responses. Views access the data needed to satisfy requests via models, and delegate the formatting of the response to templates.
- **Models**: Models are Python objects that define the structure of an application's data, and provide mechanisms to manage (add, modify, delete) and query records in the database.
- **Templates**: A template is a text file defining the structure or layout of a file (such as an HTML page), with placeholders used to represent actual content. A view can dynamically create an HTML page using an HTML template, populating it with data from a model.

2 XSS: Cross-site Scripting

Cross-site scripting, commonly referred as **XSS**, is an injection type vulnerability. The injected objects are scripts that can be executed client-side when rendering the web page. This vulnerability is mainly given by two factors:

1. Data sent to a web application through an uncontrolled web request.
2. Data dynamically included in the page without being validated properly.

The injected content might be directly JavaScript code, but also HTML or CSS code that browsers evaluate and execute to properly render the web page.

There are multiple ways of injecting malicious code when visiting a Website. For this reason they can be categorized into different types of XSS attacks:

- **Reflected**
- **Stored**
- **DOM-based**

2.0.1 Reflected XSS

In a **Reflected XSS** attack, the injected malicious code is directly *reflected* and executed when rendering the web page. The code can appear on the client-side for example through an error message, a search result, or any other response that include all or part of the input sent as request. The most common strategy to deliver a Reflected XSS attack to a victim is from crafted requests placed into an e-mail message or some other website. When the victim is tricked to perform that request, the malicious code in the request travels to the vulrnerable website and the browser of the victim executes the code reflected in the rendered webpage.

2.0.2 Stored XSS

In a **Stored XSS** attack, the injected malicious code is permanently *stored* on the vulnerable server. They can be stored in databases, forum messages, logs, comment fields. When the victim is requesting to view the malicious crafted content, the browser will also execute the code inside that content.

Blind XSS is a particular form of Stored XSS. It is called blind because an attacker cannot directly observe the response from the website. In this type of XSS the malicious code is stored on the server, as any other stored XSS attack, but the attacker has no visibility of the response generated. Generally the response in these cases can only be accessed by a user from the backend application. A common attack is through feedback forms. The attacker can craft a malicious code, sending it as a "feedback form" that we can commonly find on a website. When the backend victim will open the content, the malicious code will also be executed.

2.0.3 DOM-based XSS

In a **DOM-based XSS** attack, the malicious code is injected in sensitive sinks, and executed as result of modifying the DOM structure of the Webpage when rendered by the client browser. Sensitive sinks can be functions that allow to modify the DOM structure of the webpage or the execution of code. A simple example is the *innerHTML* property of some JavaScript objects. The DOM structure of the page can be modified with this property in the case the input is not properly sanitized. For example values from the request URL might be evaluated and directly embedded in the webpage.

2.1 Django vulnerability: CVE-2022-22818

Description
The {% debug %} template tag in Django 2.2 before 2.2.27, 3.2 before 3.2.12, and 4.0 before 4.0.2 does not properly encode the current context. This may lead to XSS.

A **template tag** is a particular way to represent data using the Django framework. As explained in the previous section, Django adopt a *Model-Template-Controller* architecture. The response for a specific request is crafted by combining templates and model into views. In particular these template tags can be used in templates file to define the general structure that the final reponse view will have. *{% debug %}* is one of the built-in templates tag really usefull for debugging purposes. This tag indeed will be replaced at runtime by debugging information in the final response (including current context and imported modules). The current context is the set of variables shared between view and template. As the description of the vulnerability says *The {% debug %} does not properly encode the concurrent context.* This means that every variable assuming the form of an executable script is not encoded properly when afterwards is showed in the page, letting the browser to execute the script instead of simply printing its text content.

Anyway this vulnerability is limited to this particular template tag and it is very uncommon to use it out of debugging scopes, or generally out of the very early stages when developing an application.

2.2 Vulnerability impact

As expressed by the CVE, *This may lead to XSS*. Since the content of the current context variables is not properly encoded, a variable containing a JavaScript script or any other form of executable code, will be directly printed on the final view of the page. If the server is not corretly set, a simple script can access important data, like *section cookies* or *csrf tokens*.

The *Set-Cookie* header does not have the property *HttpOnly*, that means JavaScript can easily access the cookies with the *document.cookie* expression. In this particular case then it will be very easy to leak csrf tokens.

2.3 Security Release

The Django version 4.0.0 was released on 7 December 2021. The security release 4.0.2 patches this vulnerability on 1 February 2022. Changes have been adopted on the usage of the **{% debug %}** template tag, in particular this template tag no longer outputs information when the **DEBUG** variable in the *setting.py* file is set to *FALSE* (ignored previously), and in the case this variable is set to *TRUE* all the context variables are correctly encoded.

3 SQL Injection

SQL injection, is another injection type vulnerability. The injected objects in this case are strings that assume the form of *SQL statements*. These specific strings, inserted in web pages input forms, might communicate with the database in non-standard ways, starting from getting the access to sensitive data, or modifying the structure of the database itself. Obviosly there exist a lot of different Database Management Systems, and for this reason the injected code has to be carefully crafted in order to communicate with the database system.

Suppose in a web page there is a form asking for *username* and *password* to access a particular web service. Those fields will be sent to the database management system and it will check if there is a user with those credentials. The resulting (very simple) query structure might be the following one:

```
SELECT user.id
FROM users
WHERE user.username = "admin" AND user.password = "admin"
```

A possible SQL injection attack might be lead in this way. As username field we can insert a string like *" OR 1=1 –*, as password it really does not care. The – express a comment for what is coming after that line. The resulting query will be:

```
SELECT user.id
FROM user
WHERE user.username = "" OR 1=1 -- AND user.password = "admin"
```

This is a really simple example, but exists a lot of different SQL injection types. They can be classified as:

- **First-Order SQL Injection**
- **Second-Order SQL Injection**
- **Blind SQL Injection**

3.0.1 First-Order SQL Injection

This is the simplest kind of SQL injection. The injected code in fact is directly interpretated by the DBMS and the action is performed. The above example is an SQL injection of this kind. Even if they seems to be relatevely simple, attacks of these type can lead to destructive actions, as deleting tables, updating existing tuples, or creating new ones.

3.0.2 Second-Order SQL Injection

This is the case in which the injected code is not directly executed but it is firstly stored in the database, and only used after in some other query, by using the previsouly injected code. For example suppose we create a new user in the web application with username *"'; UPDATE TABLE users SET password='newpassword' WHERE user='admin'– "*. The username value will be stored in the database and used like any other username in the application. Suppose now the application executes a query to retrieve our informations with the following query:

```
SELECT *
FROM informations,
WHERE informations.username = ''; UPDATE TABLE user SET password='newpassword'
    WHERE username='admin'-- '
```

The result will be destructive for the admin since the query will modify its password.

3.0.3 Blind SQL Injection

This is the case when we have no details on the query performed on the database. So in this sense we are "blind" and can only deduct different behaviour of the application on different query executions. Suppose we want to explore a database and want to leak the password of the *admin* user. We can craft a query that will return a binary response, and then proceed letter by binary search letter by letter. The finally crafted query would be:

```
SELECT user.id
FROM user
WHERE user.username = 'admin'
  AND
```

```
SUBSTRING(
   (SELECT user.password
   FROM user
   WHERE user.username='admin'),
   1, 1) > 'k')
```

Here we are going to select the password field of the admin user, extracting a single letter from that string, and checking if it is greater then the letter 'k'. Then we can proceed by checking with a stricter condition. In few words a standard technique to approach to blind SQL injections is forcing the application to a conditional behaviour.

3.1 Django vulnerability: CVE-2022-28346

Description
An issue was discovered in Django 2.2 before 2.2.28, 3.2 before 3.2.13, and 4.0 before 4.0.4. QuerySet.annotate(), aggregate(), and extra() methods are subject to SQL injection in column aliases via a crafted dictionary (with dictionary expansion) as the passed **kwargs.

QuerySet.**aggregate()**,**annotate()** and **extra()** are methods of the class **QuerySet**. A QuerySet object in Django is constructed everytime the application asks for data coming from the database. The most simple case is when we wants to retrieve all tuples from a table. As the signature specify, these methods accept as argument a dictionary ***kwargs* containing multiple parameters. A simple usage might be:

```
data = {
   # values
}
query_response = Item.objects.method(**data)
```

Through the use of dictionary expansions we can retrieve values of variables from the user input, and then use them in the above methods. If the dictionary is carefully crafted, the method can lead to SQL injection, by manipulating the values of column aliases.

3.1.1 QuerySet.aggregate(*args, **kwargs)

This method returns a dictionary of aggregate values, calculated over the QuerySet Object. Aggregate values are those values computed on groups of items, such as *averages, sums, min, max*. As argument this method can accept a dictionary that will rename the final column, through column alias, of the aggregate value.

```
data = {
   column_alias : Max('age'),
}
q = Item.objects.aggregate(**data)
> {column_alias: 99}
```

This method is directly translated into SQL syntax as follow:

```
SELECT MAX("Item"."value") as column_alias
FROM "sql_user"
```

The value of the *column_ alias* can be altered in order to perform an SQL injection attack. For practical example see Section 5.0.

3.1.2 QuerySet.annotate(*args, **kwargs)

This method performs an annotation between each object in the QuerySet and the provided list of query expressions. Suppose in our database we have two tables: *Student* and *Exam*. Each student can be linked to more exams that has passed. If for each Exam we want to know how many students have passed it, the Django syntax would be:

```
data = {
   column_alias : Count('student'),
}
q = Exam.objects.annotate(**data)
```

When we pass the dictionary as argument to this method, we can alterate the resulting SQL syntax of the query, by changing the value of the variable "column_alias". For practical example see Section 5.0.

3.1.3 QuerySet.extra(select=None, where=None, params=None, tables=None, order_by=None, select_params=None)

This is a method introducted in the QuerySet API, by compensating the lack of other APIs to handle particular edge cases query. The Django 4.0 documentation itself explains that this method should never be used if the query can be expressed with other more secure methods. In particular this method offers so much freedom to the user in order to execute even the most complex query. In the arguments of the methods can be specified the main contructs of a SQL query *(select, table, where)*. In particular the fields in the *select* dictionary express the extra columns that we would like to add to the resulting query. Notice that the already existing columns are kept in the final query. A demonstrative example is the following:

```
data = {
   "select": {column_alias: "name"},
   "tables": {"sql_user"},
   "where": ["age > 45"]
}
query_response = User.objects.extra(**data)
```

The SQL syntax translation is:

```
SELECT (name) AS column_alias, "sql_user"."id", "sql_user"."name",
    "sql_user"."age"
FROM "sql_user"
WHERE (age > 45)
```

Even in this case is evident that we can manipulate the *column_alias* variable in order to perform an SQL injection attack. For practical example see Section 5.0.

3.2 Vulnerability impact

We have alredy said this vulnerability leave space to SQL injection attacks. In particular the type of harm of an SQL injection vulnerability is potentially huge. An SQL injection attack might leak sensitive data from the database, or modify the database structure itself. A lot depends on how the database has been created, on the permissions needed to modify tables, or the contrains the DBMS mantains among the tables. Sometimes DBMS will not let us to delete some tuples because they will bring to a loss of integrity in the entire database. All of these are *In-depth mechanism defense* that will reduce the harm provocated by a SQL injection attack. Another defense against SQL injections attacks is the sanitization of the user inputs embedded in the final query construction. Double checking if the input contains particular symbols like dots, quotes, double quotes or dashes can prevent attacks of these type.

3.3 Security Patch

The Django version 4.0.0 was released on 7 December 2021. The security release 4.0.4 patches this vulnerability on 11 April 2022. More tests have been introducted on the column_alias expression. In particular has been added a new function *test_alias_sql_injection()* that will abort the execution of the query in case the alias contains whitespace characters, quotation marks, semicolons, or SQL comments. Django GitHub Commit

4 Docker demo

As practical demonstration of possible attacks that might be lead exploiting these vulrnerabilities, I prepared a docker image running a really simple Django webserver, using the version 4.0.0. The image runs two containers, in one will run the django application, in the other one an instance of postgres. The Django project is comprising of two applications, the first one is prepared to exploit the CVE-2022-22818 (XSS) vulnerability, the second one prepared to exploit the CVE-2022-28346 (SQLi) vulnerability.

4.0.1 Source code

The entire source code for the whole web application is available at https://github.com/NicolaIomm/ws-django_vulnerable. The repository contains all the instructions to run the project under the docker environment. It is necessary having installed both Docker and Docker-Compose on the machine. Here are the steps needed to run the web service:

1. Clone the repository with:
   $ git clone https:github.com/NicolaIomm/ws-django_vulnerable
2. Extract the "data.zip" file with:
   $ unzip data.zip
3. Run:
   $ docker compose build
   $ docker compose up

Note: The first time you will run the image you will get an error caused by the postgres container in order to initializate database. The second time it will work fine.

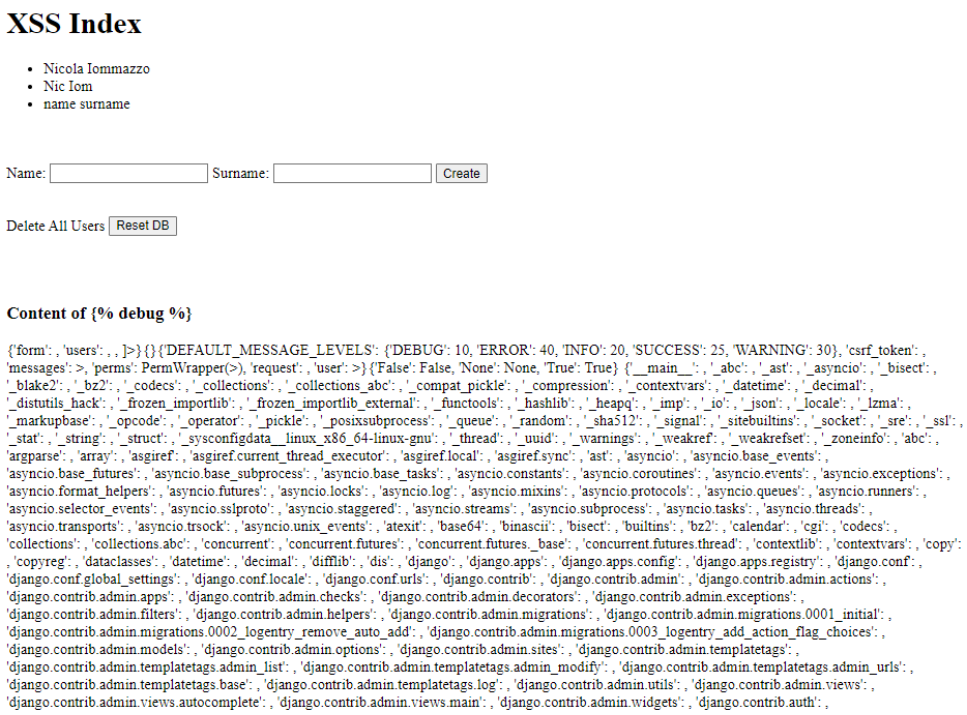
Once the web app is started there will be two URLs available, **localhost:8000/xss** and **localhost:8000/sql**, one for each vulnerability tested.

4.1 Demo/xss Application

Navigating to the page at http://localhost:8000/xss we are shown a really simple webpage:

- An unordered list containing all the users in the database.
- A form to create a new user.
- A form to delete all the users in the database.
- The value of the debugging information printed at the bottom.

The resulting page at *localhost:8000/xss* looks like this:



Here is reported the template used for the construction on the view:

```
<html>
    <head>
        <title>XSS Index</title>
    </head>

    <body>
        <h1>XSS Index</h1>

        {% if user %}
            <ul>
            {% for u in users %}
                <li> <a> {{ u }}    </a> </li>
            {% endfor %}
            </ul>
        {% else %}
            <p> No users found. </p>
        {% endif %}

        <br><br>

        <form action="" method="post">
            {% csrf_token %}
            {{ form }}
            <input type="submit" value="Create">
        </form>

        <br>

        <form action="/xss/reset" method="post">
            {% csrf_token %}
            <label for="html">Delete All Users</label>
            <input type="submit" name="delete" value="Reset DB">
        </form>

        <br><br>

        <h3>Content of &#123;&#37; debug &#37;&#125;</h3>

        {% debug %}

    </body>
</html>
```

Figure 2: Template file in *./xss/templates/xss/index.html*

In this snippet are highlighted the relevant portions of code:

- The first one is a combination of Django template tags, in particular a **{% if ... %}** and a **{% for ... in ...%}**. This portion of code will print on the HTML page all the values in the variable *user*, in form of unordered list. The content of this variable is obtained by the current context, passed from the *views.py* to the *template.py*.
- The second portion of code will simply print a form on the HTML page. The form structure is defined in the file *form.py*. In this case the form is simply asking for name and surname values. Once the user clicks on the *Create* button, the data are sent in a **POST** request at the same endpoint, so /**xss**. Notice that the Django framework needs by default the *csrf_token* in order to denote the form as valid.
- The third highlighted portion is another form. This time the only element in the form is a button that will reset the database of users. The button issues a **POST** request to the path /**xss**/**reset**.
- The last portion is a simple line where we use the **{% debug %}** template tag to print debugging information on the page.

This template is invoked by the *views.py* file, reported in the following picture:

```python
from django.shortcuts import render

from django.http import HttpResponse
from django.http import HttpResponseRedirect
from django.template import loader

from .forms import CreateUserForm
from .models import User

def index(request):

    if request.method == 'POST':
        userForm = CreateUserForm(request.POST)
        if userForm.is_valid():
            user = User(name=request.POST['name'], surname=request.POST['surname'])
            user.save()

    users = User.objects.all()

    userForm = CreateUserForm()

    template = loader.get_template('xss/index.html')
    context = {'users' : users,
               'form': userForm }

    return HttpResponse(template.render(context, request))

def reset(request):
    if request.method == 'POST':

        users = User.objects.all().delete()

        return HttpResponseRedirect('/xss')

```

Figure 3: Template file in *./xss/views.py*

Here we can find two python functions:

- The first one is referred to the page at *localhost:8000/xss/*. If we get this page through a POST method, coming from the form in the second portion of code in the template file, then we have to instantiate a new user inside the database. Then we retrieve all the users from the database, create a *userForm* and build the response starting from the template we have seen before. Notice that we pass the current context to the template, in particular both the variables *users* and *form*.
- The second one is referred to the page at *localhost:8000/xss/reset*. This function describe what we have to do when we receive a POST request. Wee simply want to retrieve all the users in the database and delete them. Then we redirect the navigation to the *localhost:8000/xss* page.

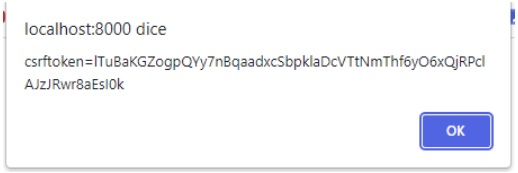
4.1.1 A possible attack

An exploit we can obtain for this really simple application is the following. Since we know that the user we create in the first form are then shows in the list, we can try to create a new user with *name* = *<script>alert(1)</script>*. As we can see nothing happens. The new user is correctly shown in the list. This means the Django framework is parsing our input in order to correctly print it on the html page.

What we also notice is that in the debugging information the content of the values in current context are not shown. By looking at the source page we see that the browser obtains the information from Django, but it is trying to interpretate the *<CreateUserForm>* and *<QuerySet>* objects as HTML tags.

To exploit this behaviour we can try to brutely close the *QuerySet* tag by adding at the start of our *name* the sequence **"]>}"**. In particular we are closing every parenthesis already opened due to object representation. Then we can concatenate that string with our script, obtaining **name = "]>} <script>alert(document.cookie)</script>"**.

If we create a new user with this specific name and whatever as surname, what we obtain is the following:

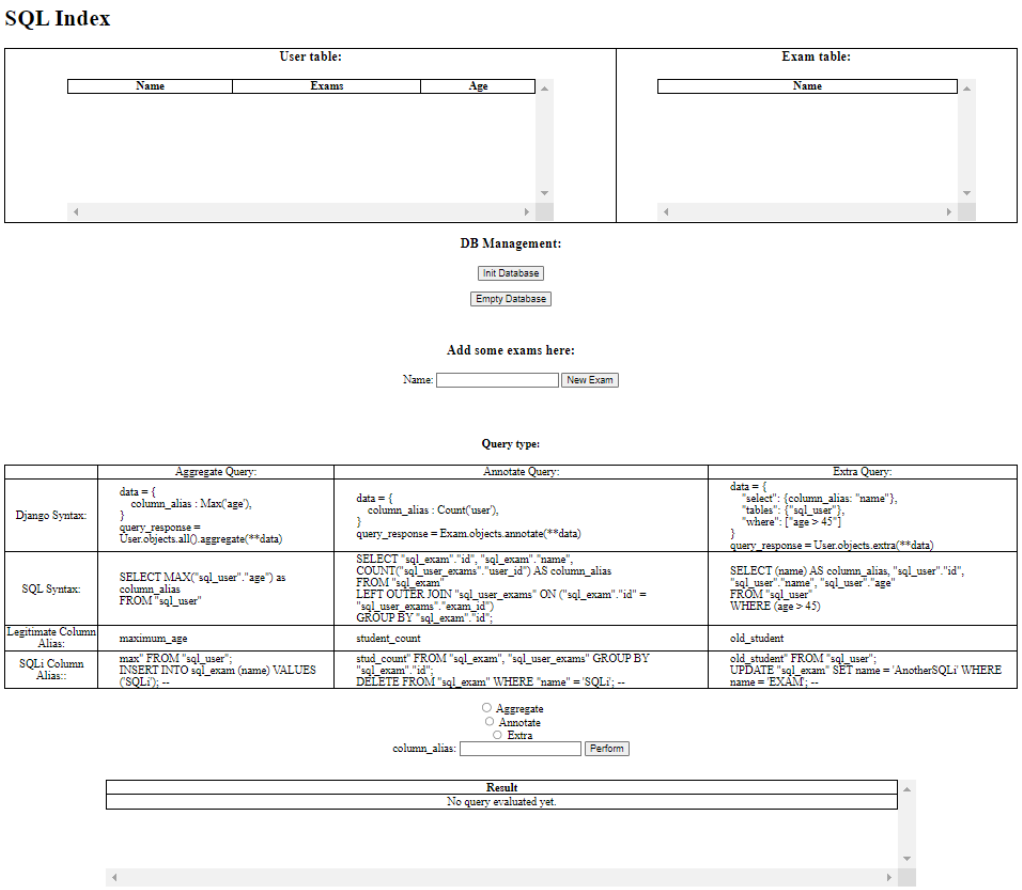


The browser has executed our script by spawning an alert box with the *document.cookie* variable. Technically what happened is that thanks to a variable stored in the database we achieved to modify the DOM structure of the page, by interrupting the *QuerySet* object list and the *User* object. By looking at the DOM we have broken the list and injected our script.

In conclusion we have exploited a vulnerable sink, that was the **{% debug %}** template tag in Django 4.0.0, by inserting a custom value in the database, in order to manipulate the DOM structure of the final page. This is an example of a mixed type of XSS, both stored and DOM-based injection.

4.2 Demo/sql Application

Navigating to the page at http://localhost:8000/sql we can find a demonstrative Django application. The page we see will be this:



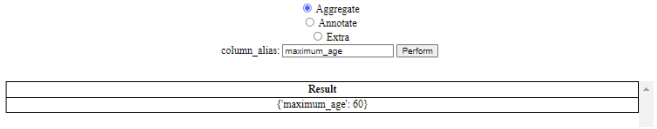
The page is obtained using the same method of the *xss* application, so just playing around with templates, forms, models and views. I will bypass any detail on the construction of this page. It is sufficient to know that the application uses two models: **User** with attributes *name* and *age*, and **Exam** with attribute *name*. Every User is linked with many Exams, representing the exams sustained by the user.

The web page structure is:

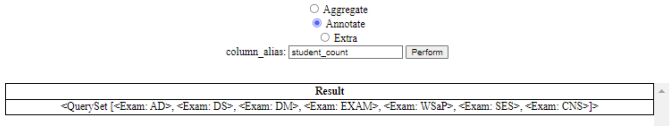
- A table representing the values stored in the database. On the left side we have the User table, on the right side we have the Exam table.
- A simple form with two buttons. The first one will randomly add new entries to the User and Exam tables. The second one will basically delete every entry from both the tables.
- A simple form to add a new Exam to the database, it is suffiecient to insert the name attribute in the apposite field.
- A table explaining the three different methods we have seen so far, that are vulnerable to SQL injection. I have reported the Django syntax (how to execute the query in python), the SQL Syntax translation of the query executed by the DBMS, a legitimate value that we can use for the column alias, and finally a crafted string that lead to SQL injection if inserted as column alias value.
- Directly below we have a form thanks to we can choose which from the above query execute, and a column alias field to customize the query.
- At the end a simple table showing the result of the query that we have chosen to execute.

Starting from an empty database, we can just initialize a new database with the apposite button. As we can see the table will be filled with twenty user, with random age, seven exams. Each user is linked randomly to many exams. For now we can just execute the queries in the legitimate way. So:

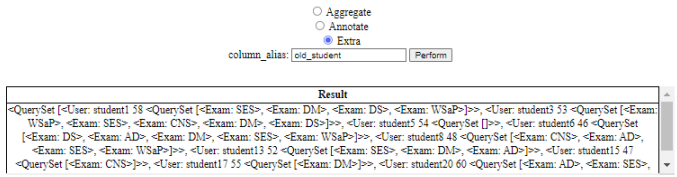
- **Aggregate query**: The query is the one showed in the web page table. The objective is to retrieve the max age from the user table. As column alias we can insert *maximum_age*. As result we will obtain a single tuple with the column alias we have chosen and the max age among all users.



- **Annotate query**: The query is the one showed in the web page table. The objective is to retrieve how many users have sustained that specific exam. As column alias we can insert *student_count*. As result we will obtain the list of all exams. In particular we will not get any value of the query we have done, because the annotate() query will create a new field in the Exam object with attribute equal to the column alias we have chosen. So in this case if we retrieve the value "*query_response[0].student_count* we will get our result. We do not really need the value, it is sufficient that the query is being executed correctly.



- **Extra query**: The query is the one showed in the web page table. The objective is to retrieve all users with age greater than 45. As column alias we can insert *old_student*. As result we will obtain the list of all users filtered with age greater then 45:



4.2.1 A possible attack

Now we analyze possible payloads in order to exploit the vulnerability found and lead an SQL injection attack. For semplicity I reported the SQL Syntax of every query executed. This has been easier by adding few logging directives in the *settings.py* file in the Django project folder. In this way I was able keep track of the queries executed from the DBMS and look at them directly printed on the console.I have also reported in the table a possible column alias value to execute a SQL injection. You can just copy and paste it in the column alias field. Notice that any change that will infer with the integrity of the database imposed from the constraints of the relations between User and Exam will be blocked from the DBMS. For this reason we will not be able to delete an existing Exam already linked to a User, but we can still add new Exams, delete unlinked Exams or modify the name of linked Exams.

- **Aggregate query**
  For this query we can break the SQL syntax right after the *column_alias* value. We still need to create a legit query, otherwise the DBMS will give error and will not continue to execute our query. For this reason we can just complete the existing query with "*max' FROM "sql_user";*", and only after that add the new query we would like to execute. In this case we execute "*INSERT INTO sql_exam (name) VALUES ('SQLi');*". At the end we need to stop the query sequence here by adding the comment symbol "–". What is following our query will be ignored. The result will be:

```
SELECT MAX("sql_user"."age") as 'max' FROM "sql_user";
INSERT INTO sql_exam (name) VALUES ('SQLi');
-- FROM "sql_user"
```

  We will notice a new value in the Exam table with name "SQLi". Obviosuly we will not get any response value from the query since we have broke the syntax with the comment symbol.

- **Annotate query**:
  As before we can break the SQL syntax right after the *column_alias* value. In this case the query is a bit more complex because it will need to perform a *Join* operation to retrieve both Users and Exams, and then perform a *Group By* operation. But we can still perform a legit query, and then add our query. So in the first part we have "*SELECT "sql_exam"."id", "sql_exam"."name", COUNT("sql_user_exams"."user_id") AS 'stud_count' FROM "sql_exam", "sql_user_exams" GROUP BY "sql_exam"."id";*", and in the second part we can perform a delete operation with "*DELETE FROM "sql_exam" WHERE "name" = 'SQLi'; –*", by taking care of commenting what comes after. At the end the following query will be executed:

```
SELECT "sql_exam"."id", "sql_exam"."name",
    COUNT("sql_user_exams"."user_id") AS "stud_count"
FROM "sql_exam", "sql_user_exams"
GROUP BY "sql_exam"."id";
DELETE FROM "sql_exam" WHERE "name" = 'SQLi';
-- FROM "sql_exam"
LEFT OUTER JOIN "sql_user_exams" ON ("sql_exam"."id" =
    "sql_user_exams"."exam_id")
GROUP BY "sql_exam"."id";
```

  As result we will notice the new value "SQLi" has been just deleted. We will not get any response value from the query since we have broke the syntax with the comment symbol.

- **Extra query**:
  Also for this query we can break the SQL syntax right after the *column_alias* value. As before we can continue the existing line with a legit query like "*old_student" FROM "sql_user";*". Then we can add the new query we want to perform. In this case we execute "*UPDATE "sql_exam" SET name = 'AnotherSQLi' WHERE name = 'EXAM'; –*". At the end the following query will be executed:

```
SELECT (name) AS old_student" FROM "sql_user";
UPDATE "sql_exam" SET name = 'AnotherSQLi' WHERE name = 'EXAM';
-- , "sql_user"."id", "sql_user"."name", "sql_user"."age"
FROM "sql_user"
WHERE (age > 45)
```

As result we will notice the value of the Exam with name "EXAM" has just been updated with a new value "AnotherSQLi". Even in this case we will not get any response value from the query since we have broke the syntax with the comment symbol.

5 References

- Django Website: https://www.djangoproject.com/
- Django Advantages: https://www.trio.dev/blog/django-applications
- Django MTC Architecture: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction
- Cross-site Scripting: https://owasp.org/www-community/attacks/xss/
- Blind XSS: https://owasp.org/www-community/attacks/DOM_Based_XSS
- Vulnerability CVE-2022-22818: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22818
- Django: Built-in template tags https://docs.djangoproject.com/en/4.0/ref/templates/builtins/
- Django security release 4.0.2: https://www.djangoproject.com/weblog/2022/feb/01/security-releases/
- Vulnerability CVE-2022-28346: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-28346
- Django: Queryset API: https://docs.djangoproject.com/en/4.0/ref/models/querysets/
- Django: Aggregation: https://docs.djangoproject.com/en/4.0/topics/db/aggregation/
- Django Docs: https://docs.djangoproject.com/en/4.0/intro/
- Docker Docs with Django Projects: https://docs.docker.com/samples/django/