Web security project

Simone Petruzzi-1811872

October 2023

1 Introduction

In today's interconnected digital landscape, web applications are at the forefront of our online experiences. They facilitate everything from e-commerce transactions to social interactions, making them essential tools for individuals and businesses alike. However, this increased reliance on web applications has also made them attractive targets for malicious actors seeking to exploit vulnerabilities and compromise data integrity and user security.

This project is dedicated to the comprehensive study of two vulnerabilities that have affected two very popular web frameworks: WordPress and Django. The types of vulnerabilities taken in address are SQL Injection (SQLi) and Cross-Site Scripting (XSS): respectively CVE-2022-34265 and CVE-2022-23988. This work aims also to represent how to exploit these two vulnerabilities and to describe how they were mitigated by the community. Eventually we'll se also a practical replication on docker of these two vulnerabilities in order to understand deeply, from a practical point of view, how to exploit them.

2 Web frameworks architecture

In this section we will deepen the architecture and main components of the web framework under consideration.

2.1 WordPress

WordPress is a popular content management system (CMS) and web framework that powers a significant portion of websites on the internet. Its architecture is built around a combination of core components and a flexible plugin and theme system.

2.1.1 Core components

- **Database**: WordPress uses a relational database, typically MySQL, to store all of its data, including posts, pages, comments, user information, and settings.
- **WordPress Core**: This is the heart of WordPress, containing the PHP code responsible for managing user requests, rendering pages, handling user authentication, and interacting with the database. The core is constantly updated and improved by the WordPress community.

- **User Interface**: WordPress provides a user-friendly admin dashboard where users can create, edit, and manage content. The UI is built using HTML, CSS, and JavaScript.

2.1.2 Other components

- **Plugins** Plugins can interact with the WordPress core, modify database tables, add new admin menu items, and execute custom code.
- **Themes** Themes control the presentation and layout of a WordPress website. They define the look and feel of a site, including fonts, colors, page templates, and responsive design.
- **Templates** WordPress follows a template hierarchy system to determine how to display different types of content. Templates are PHP files that control the rendering of pages, posts, categories, tags, and more.
- **Hooks and Actions** WordPress provides a system of hooks and actions that allow developers to add or modify functionality without directly editing core code. Hooks are specific points in the execution flow where custom code can be injected.
- **Widgets and Sidebars** Widgets and sidebars that allow users to easily customize the layout and content of their website's sidebar areas.

2.2 Django

Django is a high-level Python web framework known for its simplicity, flexibility, and rapid development capabilities. Its architecture follows the Model-View-Controller (MVC) pattern. Main three components are:

2.2.1 MVC

- **Model layer** Models represent the data structures of the application. Each model corresponds to a database table, and the fields in the model class map to the columns in the table. Django supports various types of databases including PostgreSQL, MySQL, SQLite, and Oracle
- **View layer** Views are responsible for handling incoming HTTP requests and returning HTTP responses. They act as controllers in the MVC pattern.
- **Template layer** Templates are responsible for defining the structure and presentation of the HTML content that gets sent to the client's browser.
- **Controller** Django's URL dispatcher is responsible for routing incoming requests to the appropriate view based on URL patterns.

2.2.2 Other components

- **Middleware** Middleware components are small, reusable components that process requests and responses globally for the entire application. They can perform tasks such as authentication, logging, and modifying headers before requests reach the view or after responses are generated.

- **Admin interface** Automatic admin interface that allows developers and administrators to manage application data without writing custom admin panels.
- **Auth system** Django includes built-in authentication and authorization systems, making it easy to add user registration, login, and permissions to web applications.

3 SQL Injection

SQL injection is a type of cybersecurity vulnerability and attack technique that targets web applications or databases that use SQL (Structured Query Language) to interact with their data. It's an instance of code injection vulnerability in the context of SQL.
That's how, in general, SQL injection works: an attacker must find a way to inject malicious query and force the SQL engine to respond. There are several ways in which an attacker can do this: for example it can take advantage of the **UNION** operator to extend a query by adding other malicious queries (union attack). Also an attacker could try to peform some basic attacks by exploiting the lack of input sanitization by adding for example "quote" and checking for SQL errors. Also if our database is allowing *stacking queries* an attacker can concatenate queries by using semicolon.

An attacker can perform Second-order SQL injection, that is a more advanced form of SQL injection attack. In this type of attack, the malicious payload is not directly injected into a SQL query through user inputs as in traditional SQL injection. Instead, it involves injecting malicious SQL code that gets stored in the application's database and is later executed when specific conditions are met. This is generally how it works:

- **Initial payload submission**: The attacker submits malicious input to the web application, just like in a traditional SQL injection attack. However, instead of directly affecting the SQL query, the input is stored in the application's database.
- **Lack of proper validation**: The application fails to adequately validate and sanitize the user input before storing it in the database, allowing the attacker's payload to be saved as-is.
- **Payload execution**: The stored payload remains silent until certain conditions are met. When the conditions trigger, the application retrieves the malicious payload from the database and incorporates it into a SQL query.
- **Exploitation**: At this point, the malicious SQL code stored in the database is executed as part of a legitimate SQL query, leading to unauthorized access, data leakage, or other malicious actions.

There are some mitigations in order to prevent SQL injections. First of all is a good choice to use prepared statements, such that some details of the query are known to the DB before inserting user dependent input. Also we could think of restricting the access to sensitive tables and using escape functions provided by SQL. Second-order SQL injection attacks can be more challenging to detect and mitigate because the injection point and the exploitation point are separated in time. Regular security assessments and code reviews are essential for identifying and addressing vulnerabilities that could lead to second-order SQL injection.

3.1 CVE-2022-34265

CVE-2022-34265 is a SQL injection vulnerability in Django 3.2 before 3.2.14 and 4.0 before 4.0.6. This vulnerability affected two SQL functions, namely **Trunc()** and **Extract()** functions, used for date and time manipulation, typically used in databases that support date and time data types.

The Trunc() function is used to Truncate or round off date or timestamp values to a specified level of precision. It essentially removes the fractional or less significant parts of a date or timestamp. The Extract() function is used to Extract specific components (e.g., year, month, day, hour, minute, second) from a date or timestamp value. The Trunc() and Extract() database functions are subject to SQL injection if untrusted data is used as a `kind/lookup_name` value.

The vulnerability is caused by the improper processing of the string when executing SQL for the arguments of the functions Trunc and Extract used for date data in Django. An attacker can exploit CVE-2022-34265 by crafting a malicious request that includes untrusted data in the kind or `lookup_name` argument of the Trunc() or Extract() functions. By specifying the request parameters as is in the kind argument of Trunc or the `lookup_name` argument of Extract, there is a risk that arbitrary SQL minutes can be executed.

3.1.1 Exploitation

In the model layer is used an example from Django Documentation, model for a table that records the start and end time of a date.

For what regards the model layer instead, it defines two view functions that handle HTTP GET requests, and returns JSON responses. The view function for the `extract`, receives a GET request with a query parameter `lookup_name`. It then creates a new Experiment object in the database with specific `start_datetime`, `start_date`, `end_datetime`, and `end_date fields`. The code conditionally filters the Experiment objects based on the `lookup_name` and extracts specific parts of the `end_datetime`, such as the `year`, `month`, or `day`, depending on the value of `lookup_name`.
Similarly, the view function for the `trunc` receives a GET request with a query parameter `kind`. The code conditionally filters the Experiment objects based on the `kind` and truncates the `start_datetime` to the specified `year`, `month`, or `day` depending on the value of `kind`. This is where an SQL injection could take place.

In order to verify that the injection succeed. In order to show that a potential SQL attack could be triggered, the following commands were executed:

```
curl "http://localhost:4131/extract/?lookup_name=year',␣FROM
start_datetime))␣OR␣1=1;SELECT␣PG_SLEEP(5)--"

curl "http://localhost:4131/trunc/?kind=year',␣start_datetime))
␣OR␣1=1;SELECT␣PG_SLEEP(5)--"
```

After the execution of these commands we can see really see that `PG_SLEEP` (Postgres function to pause the database's response), takes even more than 5 seconds to respond. Since arbitrary SQL statements can be executed, very dangerous attacks could take place, thanks

to the fact the the input in not correctly sanitized. Here you can see the code that has been added by the developers in order to fix the issue:

3.1.2 Mitigation

CVE-2022-34265 was mitigated in Django 3.2.14 and 4.0.6 by fixing the SQL injection vulnerability in the Trunc() and Extract() functions. The specific fix was to escape any untrusted data that is used in the `kind` or `lookup_name` argument of these functions. This means that even if an attacker is able to inject malicious SQL code into a request, it will be escaped and will not be executed by the database.

By checking the patch on git hub, we can easily note that `extract_trunc_lookup_pattern` variable is added to the code. This class variable defines a regular expression pattern used for extracting or truncating date and time values. It's compiled using `lazy_re_compile`, which is a function that compiles regular expressions lazily, improving performance. In the patch there was added also a conditional instruction that checks if `lookup_name` attribute instance matches the regular expression pattern `extract_trunc_lookup_pattern`.

Here you can see the code that has been added by the developers in order to fix the issue:

```
1 extract_trunc_lookup_pattern = _lazy_re_compile(r"[\w\-_()]+")
2 if not connection.ops.extract_trunc_lookup_pattern.fullmatch(self.lookup_name):
3             raise ValueError("Invalid␣lookup_name:␣%s" % self.lookup_name)
4 if not connection.ops.extract_trunc_lookup_pattern.fullmatch(self.kind):
5             raise ValueError("Invalid␣kind:␣%s" % self.kind)
```

In addition to the official fix in Django, users can also mitigate the affected Django's versions, by constraining the `lookup_name` and `kind` choice to a known safe list and sanitizing all user input before using it in database queries.

4 Cross-site scripting (XSS)

Cross-Site Scripting (XSS) is a type of security vulnerability and web application attack that allows attackers to inject malicious scripts into web pages accessed by other users (the malicious code is executed n the victim's browser). XSS attacks can be used to steal the victim's cookies, session tokens, or other sensitive information. They can also be used to redirect the victim to malicious websites, or to take control of the victim's browser.

There are three main types of XSS attacks:

- **Reflected XSS**: This type of attack occurs when the attacker's malicious code is reflected back to the victim in the response from the web application. For example, an attacker might submit a malicious search query to a website, and the website would then reflect the search query back to the victim in the search results page. If the website does not properly validate or sanitize the search query, the attacker's malicious code will be executed by the victim's browser when they view the search results page.
- **Stored XSS**: In this type of attack, the malicious script is permanently stored on a web server. When a user visits a page where the script is located, the script is served and executed by the user's browser. This can lead to a widespread impact as multiple users may be affected.
- **DOM-based XSS**: In this variant, the vulnerability is exploited on the client-side, within the Document Object Model (DOM) of a web page. The attacker manipulates the DOM through client-side scripting, causing the web application to execute unintended actions.

There are some precautions that can be adopted in order to mitigate XSS attacks. Most important is properly validating and sanitizing all user input, this includes input from search forms, comment forms, and other forms of user input. It is important also to properly sanitize input on server side. Other precautions consists in adopting trusted types, using escape functions, and include content security policies (CSP).

4.1 CVE-2022-23988

CVE-2022-23988 is a Cross-Site Scripting (XSS) vunerability regarding WS Form LITE and Pro plugins in WordPress. This affected the versions before 1.8.176.

This issue stems from the plugins' failure to properly sanitize and escape submitted form data. Consequently, it creates a significant security risk, as unauthenticated attackers can exploit this weakness by injecting Cross-Site Scripting (XSS) payloads into the submitted form data.

The gravity of this vulnerability becomes evident when a privileged user, such as an administrator or an authorized party, views the related submission. At this point, the XSS payloads injected by the attacker will execute within the context of the user's web browser, potentially leading to malicious code execution, data theft, unauthorized actions, or other detrimental consequences.

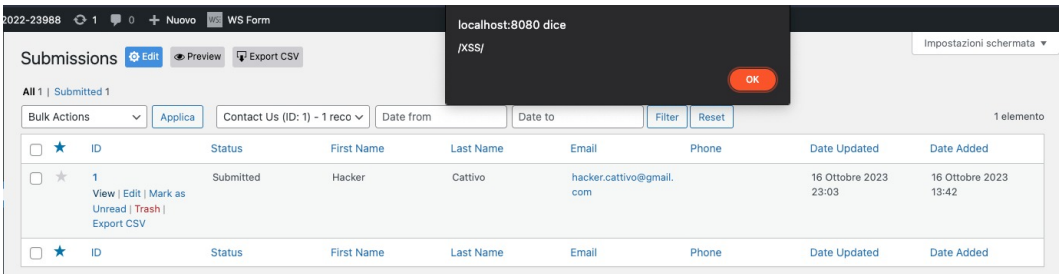
4.1.1 Exploitation

In order to exploit the vulnerability, I created a simple page in WordPress with a form, in which an attacker could inject HTML code. An unauthenticated user, that submits the form embedding a script (HTML code) in the "description" field or in the "your inquiry" field, successully injects code that is run by the admin when he checks for the submission. Indeed by going to WS form submission in the admin page, when you view the malicious submission, the script is executed inside the admin browser.

In this case the script that is injected is a simple alert in the browser showing the XSS message. The code injected is the following:

```
1          ><img src onerror=alert(/XSS/)>
```

The following picture shows the malicious code that is executed when the submission is checked in the admin panel.



4.1.2 Mitigation

CVE-2022-23988 is fixed by escaping all user input before it is displayed on the website. This means that any characters that could be interpreted as HTML code are converted to their corresponding character entities.

This escaping is done using functions provided by the programming languages or framework that is being used to develop the website. In wordpress, a plugin is a PHP file with a WordPress plugin header comment. Looking at the source code on git hub we can see that developers changed the PHP escaping function, from `e()` to `esc_attr_e()`. The main difference between `e()` and `esc_attr_e()` is that `esc_attr_e()` escapes all HTML special characters in the string that is passed to it, meanwhile `e()` does not escape any HTML special character.

In addition, the community also developed a tool to help users identify and fix websites that are vulnerable to CVE-2022-23988, this is a plugin that scans websites and checks if the current version of plugins running is safe from known problems.

You can see in the following images, the vulnerable code which was using the `e()` function in stead of `esc_attr_e()`, and how it was fixed:

wp-admin/includes/nav-menu.php

```
@@ -524,7 +524,7 @@ function wp_nav_menu_item_post_type_meta_box( $data_object, $box ) {
                        </li>
                </ul><!-- .posttype-tabs -->
-               <div id="tabs-panel-posttype-<?php echo $post_type_name; ?>-most-recent" class="tabs-panel <?php echo ( 'most-recent' === $current_tab ? 'tabs-panel-active' : 'tabs-panel-inactive' ); ?>" role="region" aria-label="<?php _e( 'Most Recent' ); ?>" tabindex="0">
+               <div id="tabs-panel-posttype-<?php echo $post_type_name; ?>-most-recent" class="tabs-panel <?php echo ( 'most-recent' === $current_tab ? 'tabs-panel-active' : 'tabs-panel-inactive' ); ?>" role="region" aria-label="<?php esc_attr_e( 'Most Recent' ); ?>" tabindex="0">
                        <ul id="<?php echo $post_type_name; ?>checklist-most-recent" class="categorychecklist form-no-clear">
                                <?php
                                $recent_args    = array_merge(
```

wp-admin/nav-menus.php

```
@@ -1040,7 +1040,7 @@ function wp_nav_menu_max_depth( $classes ) {
                                                <input type="checkbox" id="bulk-select-switcher-bottom" name="bulk-select-switcher-top" class="bulk-select-switcher">
                                                <span class="bulk-select-button-label"><?php _e( 'Bulk Select' ); ?></span>
                                        </label>
-                                       <input type="button" class="deletion menu-items-delete disabled" value="<?php _e( 'Remove Selected Items' ); ?>">
+                                       <input type="button" class="deletion menu-items-delete disabled" value="<?php esc_attr_e( 'Remove Selected Items' ); ?>">
                                        <div id="pending-menu-items-to-delete">
                                                <p><?php _e( 'List of menu items selected for deletion:' ); ?></p>
                                                <ul></ul>
```

wp-admin/widgets-form.php

```
@@ -332,7 +332,7 @@
                <div class="widget-control-actions">
                        <div class="alignleft">
                                <?php if ( ! isset( $_GET['addnew'] ) ) : ?>
-                               <input type="submit" name="removewidget" id="removewidget" class="button-link button-link-delete widget-control-remove" value="<?php _e( 'Delete' ); ?>" />
+                               <input type="submit" name="removewidget" id="removewidget" class="button-link button-link-delete widget-control-remove" value="<?php esc_attr_e( 'Delete' ); ?>" />
                                <span class="widget-control-close-wrapper">
                                        | <a href="widgets.php" class="button-link widget-control-close"><?php _e( 'Cancel' ); ?></a>
                                </span>
```