

UNIVERSITY OF BUCHAREST

FACULTY OF MATHEMATICS AND INFORMATICS

SECURITY AND APPLIED LOGIC

DISSERTATION PAPER

Python Web Scanner

SCIENTIFIC COORDINATOR: GRADUATE:

LECT.DR. SILVIU LAURENȚIU VASILE NĂIBOIU TEODOR

BUCHAREST, SEPTEMBER 2024

**CONTENTS**

[I. Introduction 7](#_Toc175244397)

[I.1 Motivation 7](#_Toc175244398)

[I.2 Personal contribution 8](#_Toc175244399)

[I.3 Structure 9](#_Toc175244400)

[II. Technologies 10](#_Toc175244401)

[II.1 Python 10](#_Toc175244402)

[II.2 Requests 10](#_Toc175244403)

[II.3 Urllib 11](#_Toc175244404)

[II.3.1 Urllib.parse 11](#_Toc175244405)

[II.3.2 Urllib.requests 11](#_Toc175244406)

[II.4 BeautifulSoup 12](#_Toc175244407)

[II.5 Threading 12](#_Toc175244408)

[II.6 OS 12](#_Toc175244409)

[II.7 ConfigParser 12](#_Toc175244410)

[II.8 RE 13](#_Toc175244411)

[II.9 ArgParse 13](#_Toc175244412)

[II.10 HTML 13](#_Toc175244413)

[II.11 JINJA2 14](#_Toc175244414)

[II.12 Nvdlib 14](#_Toc175244415)

[II.13 Wappalyzer 14](#_Toc175244416)

[II.14 DVWA 15](#_Toc175244417)

[II.15 Bee-Box 15](#_Toc175244418)

[II.16 Other technologies 15](#_Toc175244419)

[III. Vulnerabilities and Misconfigurations 17](#_Toc175244420)

[III.1 Injections 17](#_Toc175244421)

[III.1.1 SQL Injections 18](#_Toc175244422)

[III.1.2 XSS Injections 18](#_Toc175244423)

[III.1.3 HTML Injections 19](#_Toc175244424)

[III.1.4 Command Execution Injections 19](#_Toc175244425)

[III.1.5 iFrame Injections 19](#_Toc175244426)

[III.1.6 Javascript Injections 20](#_Toc175244427)

[III.1.7 IDOR Injections 20](#_Toc175244428)

[III.1.8 LFI Injections 20](#_Toc175244429)

[III.1.9 PHP Injections 21](#_Toc175244430)

[III.1.10 SSI Injections 21](#_Toc175244431)

[III.1.11 SSRF Injections 21](#_Toc175244432)

[III.1.12 XML Injections 22](#_Toc175244433)

[III.2 Security Misconfigurations 22](#_Toc175244434)

[III.2.1 Weak Browser Cache Configurations 23](#_Toc175244435)

[III.2.2 CORS 23](#_Toc175244436)

[III.2.3 Host Header 23](#_Toc175244437)

[III.2.4 HSTS 24](#_Toc175244438)

[III.2.5 RIA 24](#_Toc175244439)

[III.2.6 Robots.txt 24](#_Toc175244440)

[III.2.7 Cookies 24](#_Toc175244441)

[III.3 CVEs 25](#_Toc175244442)

[IV. Application Architecture and Implementation 26](#_Toc175244443)

[IV.1 The configuration files 27](#_Toc175244444)

[IV.2 The Payload files 28](#_Toc175244445)

[IV.3 The Report files 28](#_Toc175244446)

[IV.4 The Core files 29](#_Toc175244447)

[IV.4.1 Confidence 29](#_Toc175244448)

[IV.4.2 DataStorage 30](#_Toc175244449)

[IV.4.3 ScanConfig 32](#_Toc175244450)

[IV.4.4 Utilities 33](#_Toc175244451)

[IV.5 The Test files 38](#_Toc175244452)

[IV.5.1 SQL Tests 38](#_Toc175244453)

[IV.5.2 XSS Tests 41](#_Toc175244454)

[IV.5.3 HTML Tests 42](#_Toc175244455)

[IV.5.4 Code Execution Tests 42](#_Toc175244456)

[IV.5.5 iFrame Tests 43](#_Toc175244457)

[IV.5.6 Javascript Test 43](#_Toc175244458)

[IV.5.7 IDOR Tests 44](#_Toc175244459)

[IV.5.8 LFI Tests 45](#_Toc175244460)

[IV.5.9 PHP Tests 45](#_Toc175244461)

[IV.5.10 SSI Tests 45](#_Toc175244462)

[IV.5.11 SSRF Tests 45](#_Toc175244463)

[IV.5.12 XML Tests 46](#_Toc175244464)

[IV.5.13 Misconfiguration Tests 46](#_Toc175244465)

[IV.5.14 Other detections and features 47](#_Toc175244466)

[IV.6 Report Generation 52](#_Toc175244467)

[IV.7 Application usage 56](#_Toc175244468)

[V. Conclusions, comparisons and future enhancements 57](#_Toc175244469)

**Abstract**

The following project represents the methods and techniques used for the implementation of a scanner application written in Python programming language, designed to aid developers, white hat hackers (security researchers, red-teamers, bug-bounty hunters) and other cybersecurity enthusiast in detecting vulnerabilities and assessing the attack surface and risk exposure of a Web Application. This new implementation is a redesigned and enhanced version of my previous Web Application Vulnerability Scanner software presented in my 2021 Bachelor's Thesis [1]. This new implementation is designed for Linux ***O****perating* ***S****ystem* (OS) usage only. Users can interact with the application through the ***C****ommand* ***L****ine* ***I****nterface* (CLI) and by providing the scanning arguments such as the target website, scan type and login details.

The new scanner functionality has been enhanced by refining more accurate detections and by adding new features, such as rogue comment identification, hierarchical site map graph, external ***U****niform* ***R****esource* ***L****ocators* (URLs) references, hidden URL paths and ***C****ommon* ***V****ulnerabilities and* ***E****xposures* (CVE) detection. The reporting method has also been changed to a ***H****yper****T****ext* ***M****arkup* ***L****anguage* (HTML) report. The new version has been built for scalability and ease of access, as the number of prerequisite configurations has been decreased compared to the previous version. Additionally, the majority of ‘future enhancements’ objectives mentioned in the previous thesis were implemented in this new version.

# Introduction

*“*Cybersecurity is the art of protecting networks, devices, and data from unauthorized access or criminal use*”* [2]. This formal definition of cybersecurity can be visualized as a tug-of-war battle between two opposing forces whose common purpose is to dominate each other’s’ territory. Over the last years, the need of understanding and educating people on cyber security has been increasing. Nowadays, not only people working in the **I**nformation **T**echnology (IT) industry are impacted by digital data polluted with malicious additions, but any other person with a smart device connected to the internet can be a victim of the attackers’ bad intentions, either directly or, as most of the time, coincidentally.

Currently, more and more solutions are transitioning to web applications designed to provide the users with a fast resolution for everyday activities (e.g., buying a bus ticket, ordering a meal from the local restaurant, tracking your parcels or accessing your bank balance). “On average every day, approximately 30000 new websites fall to hacking attacks” [3]. As the numbers of websites is increasing due to demand and digitalization, so are the attacks aimed at stealing data by exploiting misconfigurations, vulnerabilities or by simple social engineering attacks facilitated by lack of access restrictions. Due to this reason, numerous websites are storing sensitive user data that can be worth lots of money in the hands of the wrong people. So, protecting the web applications and their data has become one of the most important security challenges of our times, as today, compromising such an application can provide valuable data for further elaborated attacks.

## Motivation

I have decided to continue my previous implementation of the web application vulnerability scanner and to enhance its functionality due to my ongoing passion for web application security and for my personal projects. “A web application is a computer program stored on a remote server and run by its users via a web browser” [4]. “A web browser is a computer program used for accessing sites or information on a network (such as the World Wide Web)”[5]. As the web application is designed to be publicly accessible, attackers have an easy target available to lock on their scanners to and wait for immediate and easy to reach security weaknesses. Due to this reason, I have aimed to develop a scanner tool that simulates attackers’ tactics to proactively identify the latest emerging threats.

As information is constantly changing, so are software solutions and their configurations. Due to the ongoing modifications in web application service configurations and their increased complexity, researchers and ethical hackers can have difficulties in detecting vulnerabilities in a timely manner. This also causes automated vulnerability detection tools to become deprecated really fast. As my project is an automated detection tools as well, I have decided to further continue its development and update the detection capabilities to keep up with the latest security risks and recommendations while also keeping track of scalability and performance. In the following pages, I will present the new features of this tool, as well as the implementation methods I have chosen for this new and enhanced web applications scanner.

## Personal contribution

While the previous version was much more focused on the ***O****pen* ***W****orldwide* ***A****pplication* ***S****ecurity* ***P****roject* (OWASP) Top 10 Standards [6], the new version is based on my work experience as a Security Analyst and on ***C****apture* ***t****he* ***F****lag* (CTF) competitions and observations, over the last 3 years.

“CTFs are gamified competitive cybersecurity events that are based on different challenges or aspects of information security” [7]. The focus was shifted from a OWASP Top 10 coverage approach to more subjective means of detections based on my past observations from live models of vulnerable web applications. Almost all of the detection methods and capabilities were implemented based on my best judgement instead of generic approaches. This was done to further increase the accuracy of detections and to decrease the chance of false-negative findings. Due to this approach, there might be a significant number of false-positive findings as well, hence the scanner results should be confirmed by manual replication.

Furthermore, I have decided to bring this scanner up to a competitive level by lowering the number of prerequisite configurations needed, and by providing a human readable report with dynamic and intuitive ***U****ser* ***I****nterface* (UI).

## Structure

The structure of this paper consists of five chapters in which I have described all the technologies, concepts and practical approaches used for the implementation of this Python Web Application Scanner. In the following chapters I will be describing the technologies used to build the scanner application, the vulnerabilities and misconfigurations the scanner is designed to detect, the practical approach and the implementation of the testing methods, the report generation logic and a usage example. In the final chapter I will also be comparing this project with other similar testing tools and I will be listing some possible future enhancement ideas.

# Technologies

Compared to the previous scanner version which was running Python version 3.2, the new version was implemented in Python version 3.9. Python is one of the best programming languages to use for automated sequential or parallel instructions, since it is easy to learn, to read and to understand [8]. Additionally, what makes Python great for automations is the big number of modules and libraries available and easily compatible with numerous software applicability (web and desktop applications, data structures and analysis, scripts, etc.).

For this project, the most important modules and components are the ones used to communicate with the website, such as *beautifulsoup* [9], *requests* [10], *urllib* [11]. In the following pages I will be describing the modules among other technologies and solutions used to build, run and test the new version of the scanner.

## Python

“Python is an interpreted, object-oriented, high-level programming language with dynamic semantics” [12]. Due to the great, easy to read and write syntax, Python is widely used for complex automated actions among cybersecurity enthusiasts. In projects similar with this one, Python is the best choice of any individual interested in ease of use and versatility. Web applications are constantly changing their structure and code, due to this reason, automated scanning tools need to be implemented in easy to maintain solutions that can handle large amount of data and also be compatible with other tools and technologies [13].

## Requests

The *requests* module is used for transferring data from a client (*Web Browser*) to a *Web Server* using the ***H****yper****t****ext* ***T****ransfer* ***P****rotocol* (HTTP) protocol and ***U****niform* ***R****esource* ***L****ocators* (URLs) [10]. There are a total of nine requests type (*GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH*). During the development of this project, I have used four types of these requests:

**GET** – One of the principal types of requests as it represents the action of requesting data from the server. This request type is used for receiving data according to the provided parameters.

**POST** – Another principal request method used to send data to the server. This method can be used for creating new data on the server and for updating already existing data.

**PUT** – Method used for creating or changing the representation of a resource according to the data provided in the request payload.

**TRACE** – Method generally used for debugging purposes. This method provides information on the path the request takes to the target destination.

The *requests* module is not only used for exchanging data with the server, but is also used for creating the session to maintain persistent communication with the web server during the testing procedures. This module also provides custom request creation functionalities and HTTP error handling methods.

## Urllib

Urllib module is used generally for URL manipulation [11]. From this module, I have used two functionalities:

### Urllib.parse

The *.parse* module from *Urllib* is generally used for combining strings back into URL format. In this project, the module has been used for combining actions, extracting certain strings of interest and fragment parts of URLs into variables [14].

### Urllib.requests

The *.requests* module from *Urllib* is similar with the *requests* module. The difference between the two is that this module provides some more advanced HTTP actions. In this project, this module was used for opening and reading URLs [15].

## BeautifulSoup

One of the most popular modules used for interaction with ***H****yper****t****ext* ***M****arkup* ***L****angua*ge (HTML), ***E****xtensible* ***M****arkup* ***L****anguage* (XML) components and ***D****ocument* ***O****bject* ***M****odel* (DOM) of a page. It provides a wide array of functions that help with pulling data from web pages, navigation, modifying and parsing different values available on a website [9]. This module is responsible for any interaction between the scanner and the targeted web page content.

## Threading

Threading is used for changing the flow of execution from a sequential flow to a parallel flow by creating a separate line of execution that can run simultaneously. Threading is useful for applications that handle a large number of independent instructions and in which execution time is also important. The CPU is used for allocating divisions of the physical core into virtual components or codes, these virtual components are called *threads* [16]. Threading was implemented for decreasing the execution time of the project by running multiple tests for multiple URLs in parallel.

## OS

The ***O****perating* ***S****ystem* (OS) module is used in any interaction with the operating system. Actions such as creating, reading, modifying, opening and closing files and directories are performed through the usage of this module [17]. This module can also be used for more advanced interactions with the OS such as accessing environments variables, process information, managing registries and other system related functions. In this project, the module is responsible for file interactions.

## ConfigParser

Configuration files are a necessary part of an application that interacts with multiple pieces of data and complex logic. Configuration files are code structures with the *.ini* file extension that can be easily modified by experienced and beginner users without the need of any application architecture knowledge. A configuration file contains custom variables that are usually depended on the execution environment or purpose of the application [18].

The *ConfigParser* module is responsible with the interaction between the configuration file and the Python code. This module is used to access and import the configuration values in the code and execution logic.

## RE

***R****egular* ***E****xpressions* (RE or Regex) are recognition rules based on symbols and characters that form search patterns in order to detect any certain characters or string of characters inside other strings based on that searching rule [19]. Once a pattern is defined, an identified string can be manipulated in any way the user wants, those include edits, deletions, word substitutions, character(s) extractions from any string of characters that contains that pattern.

The RE module is the Python implementation of Regex functionality. Regex has been extensively used across this project for vulnerability detection checks, result validations, URL manipulation and other.

## ArgParse

ArgParse is a module used for Python applications that require user interaction through the ***C****ommand* ***L****ine* ***I****nterface* (CLI). ArgParse is the link between *sys.argv* (the list of commands passed to the script using the CLI) and the Python code [20]. The advantage of using ArgParse is that it provides a user-friendly CLI while also automatically generating the *–help* menu used for displaying all the available arguments. From a programming point of view, this module also provides an easier method to manipulate the arguments passed via CLI into the code logic. Since this new scanner version is based on CLI interaction, this module is needed for parsing the user arguments to the application logic.

## HTML

This is a simple module used for manipulating HTML entities. The submodules used from HTML are the modules that *escape* and *unescape* characters. An *escaped* character is a character that uses an alternative interpretation of the same character [21]. For web application scanning, certain characters need to be escaped when a set of data is sent to the website and others need to be unescaped when data is received. Across this project, data sent and received to, respectively from the application, needs to be escaped for further processing down the test logic.

## JINJA2

The Jinja2 module is responsible for HTML template creation and manipulation through Python code [22]. This module is used for creating the final HTML report file based on an already created template which serves as a starting point for the report output data and format. All scanning results are parsed through Jinja2 to the templating functions that handle the data. The data is then added to the report, through Jinja2 methods, along with the HTML tags required for the HTML page to be correctly interpreted and displayed.

## Nvdlib

“NVDLib is a Python ***A****pplication* ***P****rogramming* ***I****nterface* (API) wrapper utilizing the ***RE****presentational* ***S****tate* ***T****ransfer* (REST) API provided by ***N****ational* ***I****nstitute of* ***S****tandards and* ***T****echnology* (NIST) for the ***N****ational* ***V****ulnerability* ***D****atabase*” (NVD) [23]. In this project, this module was used for interacting with the NIST CVE database. This module is responsible for retrieving all the required data to search for CVEs based on the identified technologies and their versions.

## Wappalyzer

Wappalyzer is a web application built for extracting and identifying technologies running on websites. The Wappalyzer module used in this project is the Python implementation of this online tool functionality. This module was used through the implementation of the scanner for CVE identification functionality. This module is responsible for identifying the running technologies and their versions to be later used for identifying the necessary information to work with the CVE NIST API module (nvdlib) [24].

## DVWA

*“****D****amn* ***V****ulnerable* ***W****eb* ***A****pplication* (DVWA) is a PHP/MySQL web application that is left intentionally vulnerable with the main goal of helping security enthusiasts, web developers, students and teachers to use test their tools, practice their cybersecurity skills and understand web application security better, in a controlled, isolated and legal environment” [25].

This application is one of the two web servers used in this project for testing the scanner and simulating real life scenarios. This web server contains numerous different vulnerabilities and misconfigurations which allow a comprehensive coverage of the most frequently found vulnerabilities in the wild. This application was hosted in a controlled virtual environment using the Kali Linux Distribution.

## Bee-Box

Bee-Box or BWapp is also a PHP/MySQL web application left intentionally vulnerable that follows the same purpose as DVWA. Bee-Box contains over one hundred web vulnerabilities and it covers all the major known web weaknesses and bugs along with the ones from OWASP Top 10 project [26].

This is the second web server used for testing the scanner progress and detections in this project. This web server was hosted in a controlled virtual environment using the Ubuntu Linux Distribution.

## Other technologies

Below are some of the less utilized but still important modules that were used for the implementation of the web application scanner.

“The *random* module is used for generating pseudo-random numbers or choices for various purposes” [27]. In this project, the module was used for choosing a pseudo-random web user agent from a list. A web user agent is a characteristic string of data that helps servers identify the application, operating system and vendor of the requesting source [28].

The *datetime* module is used for accessing the calendar’s date and time. Using this module, I have provided date and time details in the resulting report with the format *YYYY:MM:DD HH:MM:SS* [29]*.*

“The *math* module provides access to mathematical operations and functions” [30]. In this project, the module was used to perform rounding on list length divisions used to split the data for threading purposes.

The *colorama* module is used to color the output text to terminal [31]. In this project the module was used to color the output text when the error file is not available due to any reason.

The *json* module is used to import ***J****ava****S****cript* ***O****bject* ***N****otation* (JSON) data interpretation. JSON is a standardized format for data that can be easily interpreted and read both by humans and computers [32]. In this project, the *json* module was used for creating and manipulating the URL graph map.

The *warning* module provides warning control functions. Warnings are issued in situations in which the script does not raise a condition that terminates the program, but the user should be aware of the event. In this project, the *warning* module was used only to suppress character encoding warnings for HTML received data [33].

# Vulnerabilities and Misconfigurations

This chapter is necessary for better understanding vulnerabilities and common misconfigurations a web application usually has. This chapter covers the theoretical implementation of the scanner. The following few pages present all the information necessary for understanding the detection capabilities of the scanner and its report.

“A vulnerability, is a flaw in code or design that creates a potential point of security compromise for an endpoint or network”[33]*.* Usually, thereason a vulnerability exists is due to a misconfiguration. These misconfigurations are usually incorrect setups of data in the system or the environment. Misconfigurations can be caused by lots of variables, both from a technical point of view to business logic.

Vulnerabilities represent attack opportunities for adversaries. There are multiple methods of abusing vulnerabilities for gaining access to the weakened system. Depending on the type and motives, attackers can leverage them for unauthorized access in the environment. Both *white-hat hackers* and *black-hat hackers* are constantly searching for these vulnerabilities as they represent the initial key of access to any system. The notion of *white-hat hacker* is used to refer to a person motived by legal and ethical reasons to find and report such misconfigurations. The *black-hat hacker* term is used to describe people motivated by and not limited to financial gain or by seeking reputation from their community while disregarding the legal and moral constrictions [34].

In the following pages I will be describing some of the vulnerabilities and misconfigurations that can be detected using this scanner, with the mention that the new scanner version is no longer following the OSWASP Top 10 Standards, hence the vulnerabilities will be more biased towards personal criteria of exploitability, severity and importance.

## Injections

“Injections based vulnerabilities are still one of the most abused and widely spread ones, due to this reason, injections are considered a critical threat to cybersecurity” [35]. Injections rely on user input fields. The provided data always needs to be interpreted by the application; hence the application is susceptible to any malicious instruction if the input field is not properly sanitized. There are multiple types of injection attacks, with the majority of them depending on the software solutions running on the application server.

### SQL Injections

*“****S****tructured* ***Q****uery* ***L****anguage* (SQL) is the most common language for interacting with data stored in a relational database” [36]. These injections rely on misconfigurations in the input fields connected to the SQL Database running in the back-end of the web application. Attackers can leverage these vulnerabilities to arbitrarily tamper with any piece of data stored in the application. SQL Injections can be blind or reflected. With reflected SQL Injections, feedback is provided back to the client according to the provided set of SQL data. Blind injections do not provide any feedback back to the client; hence, are more difficult to detect.

### XSS Injections

*“****Cross****-****S****ite* ***S****cripting* (XSS) injections refer to the practice of sending malicious scripts, in the form of a browser side script to a target victim using a web application” [37]. The victim’s web browser will execute the code as part of the legitimate website. XSS attackers are used when attackers want to execute code on behalf of another user in the browser. The executed code will have all the privileges and access rights the victim has at the moment the malicious script is executed. There are multiple types of XSS attacks.

#### Reflected XSS

Reflected XSS attacks are the most common XSS attack in the wild. This type of attack relies on single execution of the malicious script and requires user interaction. The malicious code is not stored in the browser and can only be executed once, this applies to both requests and response transmitted malicious code [38].

#### Stored XSS

Stored XSS is one of the most dangerous XSS attack as it does not require any user interaction. This attack is based on injecting XSS code directly into the database through the vulnerable inputs. The malicious code gets stored on the affected page and gets executed each time the page is accessed, hence affecting all the users visiting it [39].

### HTML Injections

This injection is a type of attack in which HTML tags are injected directly in the DOM of the website. This attack is different than others as it does not exploit the back-end of the server, but the structure and content of web page. Similar with the XSS, this type of injection can also have a single time execution (Reflected HTML Injection) or a persistent injection (Stored HTML Injection) [40].

### Command Execution Injections

“Also called ***R****emote* ***C****ode* ***E****xecution* (RCE), code injection is a method that allows attackers to remotely execute unauthorized code on targeted systems with the end goal of executing commands”. The targeted server executes OS commands received through the web application inputs. Similar with the SQL injection, Command injection can also be reflected or blind.

### iFrame Injections

“iFrame is short for inline frame and is used to embed content from another website into the current page” [41]. iFrames injection is the practice of injecting malicious code into a webpage in the form of an iFrame element in the HTML page. Once an iFrame is injected, attackers can add their own controlled domain snippet to the targeted app, this can cause numerous security issues as iFrames provide full view of the embedded domain to the body of the compromised webpage.

### Javascript Injections

“JavaScript is the most powerful and versatile web programming language used for building interactive websites through implementation of animations, interactive forms and dynamic content to web pages” [42]. Javascript injections are a type of injections in which attackers inject malicious Javascript code into the client side of the application, usually a web browser. Through this attack, adversaries can steal sensitive information such as personal user data, credentials, sessions, cookie settings and any other data available in the target’s browser. They can also manipulate the targeted website, redirect the users to other attacker-controlled domains or install malicious extensions in the victim’s browser.

### IDOR Injections

“***I****nsecure* ***D****irect* ***O****bject* ***R****eference* (IDOR) is a vulnerability that arises when attackers can access or modify objects by manipulating parameters used in a web application” [43]. This type of attack allows adversaries to access unauthorized data through URL tampering or DOM body manipulation by modifying the vulnerable fields with custom values that link different objects in the back-end or database. This vulnerability is most commonly abused for horizontal privilege escalation, but it can be used for vertical privilege escalation as well. The term of horizontal privilege escalation refers to the practice of tampering with data of users with the same access level as the attacker’s. Vertical privilege escalation refers to the practice of accessing a higher level of privilege than the one they initially possess.

### LFI Injections

***“L****ocal* ***F****ile* ***I****nclusion* (LFI) allows an attacker to include files on a server through the web browser” [44]. Local files are files that are already present on the server that is running the web application. Accessing such files can lead to information disclosure, RCE or XSS. This vulnerability can be easily abused by providing filenames or file paths as inputs in the URL parameters or in the body of a website. Using this vulnerability, an attacker can perform a Directory Transversal attack and access any file from the server. Path or Directory transversal is a type of vulnerability in which the attacker is navigating the file hierarchy on the targeted system through CLI compatible commands such as *‘../../etc/passwd*’ used for navigating two folders back and accessing the account passwords located in the ‘*etc’* folder in UNIX based systems.

### PHP Injections

“PHP (recursive acronym for PHP: Hypertext Preprocessor) is a widely-used open-source, general-purpose scripting language that is especially suited for web development and that can be embedded into HTML” [45]. PHP Injections are very similar to Javascript or Code Injections as it requires an attacker to send malicious PHP code to an application running PHP. The provided code then gets executed successfully on the back-end server.

### SSI Injections

*“****S****erver-****s****ide* ***I****ncludes* (SSIs) are directives found within a web application’s HTML code and are used to provide an HTML page with dynamic content***.*** SSI injections are a server-side exploit that enables an attacker to inject code into a web server and execute it upon the page load” [46]. The SSI syntax is ‘*<! --#directive parameter=value parameter=value -->’.* Directives are instructions that can be interpreted by the web server for providing dynamically generated content on a web page.

### SSRF Injections

“**S**erver-**S**ide **R**equest **F**orgery (SSRF) attacks allow adversaries to trick server-side applications to allow access to the server resources. The attack can be performed if the target application supports data imports or reads data from URLs without proper validation” [47]. SSRF Injections are performed by injecting an arbitrary domain into the URL or replacing the URL altogether on the targeted web application. The resulted HTTP requests can be abused for unauthorized access or actions over the web server or any other external server communicating with the vulnerable web application.

### XML Injections

***E****xtensible* ***M****arkup* ***L****anguage* (XML) is used to store, transport and exchange data. The difference between HTML and XML is that HTML is oriented towards displaying data and can be managed only with predefined tags, whereas XML is more focus on data manipulation and allows the users to define their own tags. Due to this reason XML can be abused by adversaries by creating custom ***X****ML* ***E****xternal* ***E****ntities* (XEE) with specific tags designed to allow access to view files on the server filesystem and to interact with any other back-end or external system that the web application has access to and communicates with [48].

The majority of the above injections can be mitigated by implementing input validation and sanitization. Developers need to ensure that all available inputs are validated, sanitized and that the outputs are properly encoded. There are a multitude of methods depending on the frameworks and software solutions used by the web application that need to be properly configured, such as parameterized queries for SQL interaction, pre-compiled and stored objects for databases, restrictive access and error handling.

## Security Misconfigurations

“Security misconfigurations represent any error or vulnerability in the configuration of the code that allows attackers to access sensitive data” [49]. Almost all vulnerabilities are, to some level, misconfigurations caused by either human or software error or due to deviations from best-practices. There are multiple types of misconfigurations that either represent full vulnerabilities or can lead to opportunities by allowing prerequisite configurations for defense impairment. Due to their nature, misconfigurations are difficult to detect due to the high number of places they can be found. It is a very difficult job for *ethical hackers* and *developers* to correctly identify and mitigate, in a timely manner, all security misconfigurations present in a web application. Due to this reason, I have decided to cover the most frequent and common misconfigurations found in the wild and some other that are not that popular. Detecting misconfigurations as soon as possible is the key to keeping attackers away and to discourage them for further targeting specific web servers.

### Weak Browser Cache Configurations

“Browsers can store information for purposes of caching and history. Caching is used to improve performance, so that previously displayed information doesn’t need to be downloaded again” [50]. Caching issues can lead to vulnerabilities such as Cache Poisoning and Cache Deception or sensitive data disclosure by abusing other vulnerabilities such as XSS along with the caching misconfiguration. Cache Poisoning refers to the practice of injecting malicious code into the cache. On requests, the malicious content gets served to other web applications users. Cache Deception is more oriented towards sensitive data access, the attacker causes the web application to store sensitive content in the cache, the content can be later access by adversary [51].

Generally, for web pages with sensitive content or information, the cache should be disabled.

### CORS

*“****C****ross-****O****rigin* ***R****esource* ***S****haring* (CORS) is a protocol that enables scripts running on a browser client to interact with resources from a different origin” [52]. CORS is usually used for web application that rely on third-party APIs or embedded content from other websites. This attack relies on the bad validation of the actual origin of the request. An attacker can create a custom script that sends requests to an application and then execute it from their controlled malicious domain. Without proper validation, the targeted application can provide the attacker with access to its resources [53].

### Host Header

“The Host header is a crucial part of the HTTP protocol and is used by web servers to determine which virtual host or website the client is requesting” [54]. The Host Header misconfiguration provides an attacker the opportunity to inject malicious code into the targeted application Host-Header. Similar with other misconfigurations, this attack also relies on the improper validation of the Host input.

### HSTS

***H****TTP* ***S****trict* ***T****ransport* ***S****ecurity* (HSTS) governs the connection method of a web browser to a web application. This configuration is responsible for ensuring the web server is only communicating with clients through the HTTPS (HTTP Secure) protocol by upgrading the connection from the HTTP protocol to HTTPS through a redirect. HTTP connections are vulnerable to ***M****an-****I****n-****T****he-****M****iddle* (MITM) attacks and other data sniffing techniques and are generally considered unsafe. HSTS flag should be generally enabled in the headers of a request to avoid the usage of HTTP communication [55].

### RIA

*“****R****ich* ***I****nternet* ***A****pplications* (RIAs) refer to web applications that offer dynamic content and interactions, often inaccessible to users who rely on screen readers or other assistive technologies” [56]. RIA uses Adobe *crossdomain.xml* to allow cross domain access to the web server resources required for RIA good functionality. Restrictive permissions need to be set up to prevent attackers from abusing RIA function to access sensitive data on the server.

### Robots.txt

“The robots.txt file tells search engine crawlers which URLs the crawler can access on a website” [57]. The file is a text base field that contains numerous web paths allowed or disallowed from search engine indexing. Due to this reason, there is a chance that sensitive paths such as */admin* or */administrator* can be accidentally leaked. As those paths are not usually designed for public access, they can lack the necessary access restriction a usual public path would have.

### Cookies

“Cookies are small files of information that a web server generates and sends to a web browser” [58]. Cookies are mostly used for tracking and tailoring the users experience over the web application. Depending on the website, cookies can be set as flags for numerous access rights or for storing relevant user data such as the session ***id****entification* (ID). If not properly secured, attackers can access the cookies and steal or modify relevant data that can lead to website compromission. Cookies can be frequently misconfigured by not protecting the user’s session cookies with appropriate attributes and by permitting stored cookies to persist indefinitely in the user's browser [59].

## CVEs

*“****C****ommon* ***V****ulnerabilities and* ***E****xposures* (CVEs) is an international framework and effort to maintain an updated registry of all known computer security vulnerabilities and exposures” [60]. CVEs can be applied to any piece of software running in any environment. Modern web applications are very complex with many third-party integrations and frameworks embedded. “A framework is a collection of reusable software components that make it more efficient to develop new applications” [61]. Due to the multiple pieces of individual components that now form a web application, a potential CVE identified in one solution running on a web server can be exploited for full compromission of the whole server. It is very important for web server owners to manage and be up to date with the latest CVEs for the software running in their environment.

In this project I have also implemented methods for automatic CVE detection using the ***A****pplication* ***P****rogramming* ***I****nterface* (API) from ***N****ational* ***I****nstitute of* ***S****tandards and* ***T****echnology* (NIST) ***N****ational* ***V****ulnerability* ***D****atabase* (NVD). “An API is a collection of communication protocols and subroutines used by various programs to communicate between them” [62]. “NIST NVD is the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP)” [63].

# Application Architecture and Implementation

Unlike the previous version of this scanner, the new one has been split amongst multiple folders according to their code and purpose. The previous version was implemented on a single Python file too big to manage and difficult to keep up to date due to number of lines. The new version is easier to maintain and update due to the module-like approach. The filenames have descriptive names according to their role in the application logic.

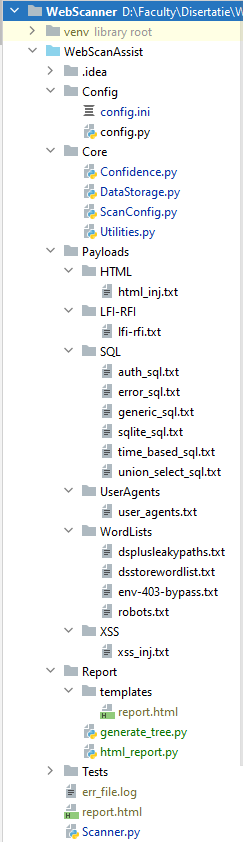


Figure 4.1. Application file hierarchy

## The configuration files

The configuration files are stored in the ‘Config’ folder. The folder contains the Python code file ‘config.py’ required for reading and parsing the configurations from the ‘config.ini’ configuration file stored under the same folder.

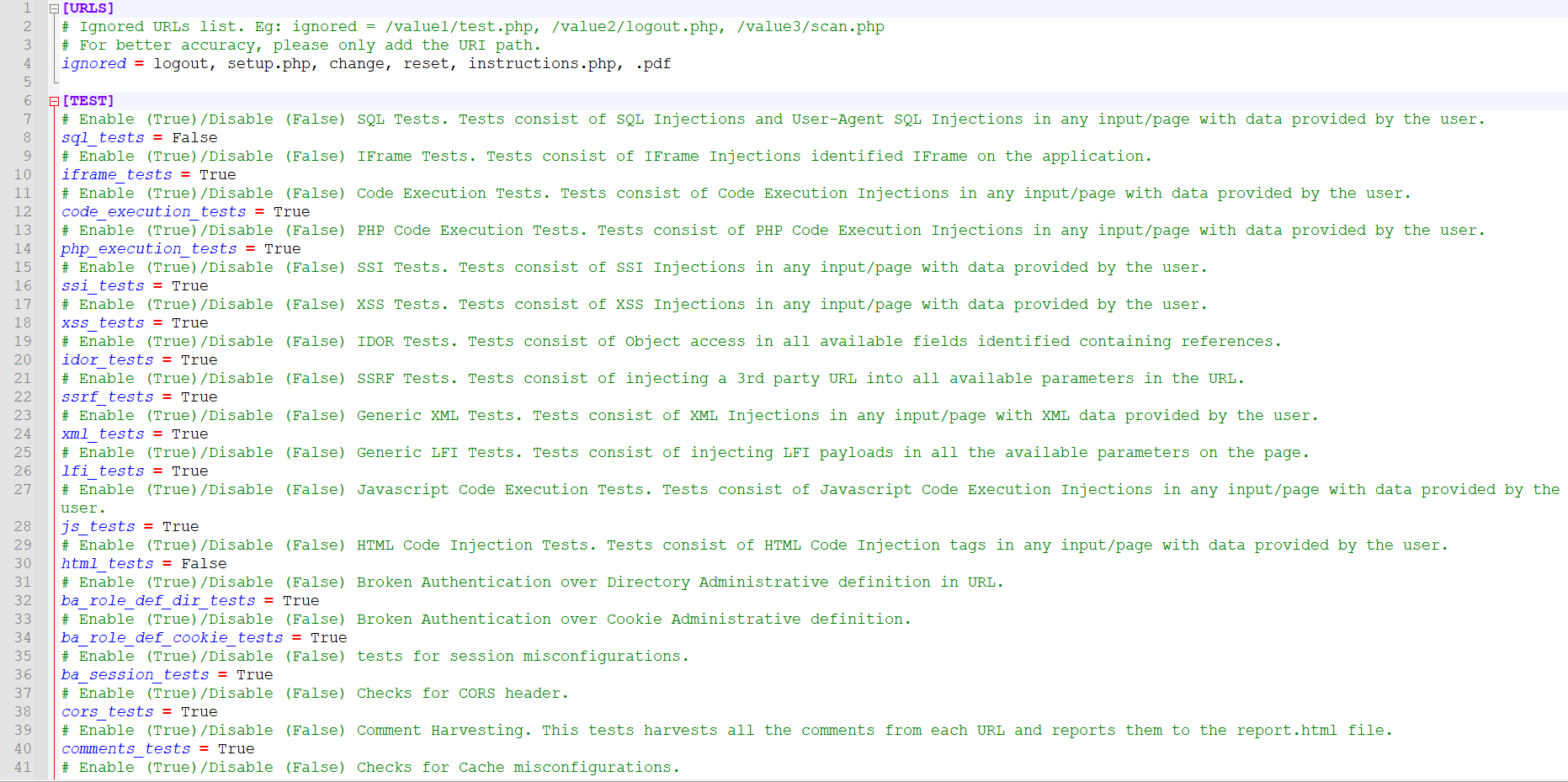
class Config:  
 def \_\_init\_\_(self):  
 try:  
 self.config\_object = ConfigParser()  
 self.config\_object.read("Config/config.ini")  
 except Exception as e:  
 print(e)

Figure 4.2. Configuration file contents (partial capture)

The configuration file ‘config.ini’ contains the following options: variables for enabling/disabling specific tests and features, the ignored URL string list required for ignoring specific URL keywords (e.g., logout.php) and the API private key required for NIST interaction for CVE identification.

## The Payload files

“A payload in cybersecurity is a code that is designed to perform malicious activities on a victim’s computer or network” [64]. In this project, the ‘Payloads’ folder contains the necessary strings and data required for testing and validating the vulnerability state of the targeted input.

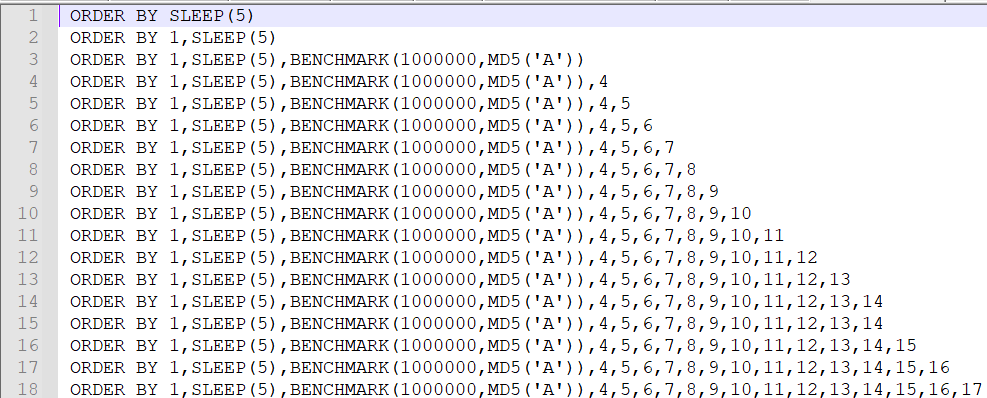
 As observed in the *Figure 4.1 Application file hierarchy* each file and folder name are descriptive for their purpose. These files are dictionary of words from the most popular and up to date payload repositories on the internet [65] [66] [67] [68] [69]. Unlike other scanner applications that rely on static and hard-coded out-of-reach dictionary of payloads, this format allows the user to provide his own dictionary of words or payloads required for testing the targeted application.

Figure 4.3. Contents of union\_select\_sql.txt (partial capture)

## The Report files

The ‘Report’ folder contains the necessary files for dynamically generating the final .HTML report file.

As also observed in *Figure 4.1 Application file hierarchy* the folder contains a subfolder named ‘templates’ with the ‘report.html’ template. This template contains the necessary HTML and JINJA2 tags required for dynamically generating and displaying the final report based on the lists of data populated by the executed testing methods results. This file contains the ***C****ascading* ***S****tyle* ***S****heet* (CSS) code required for styling the report to an easy to navigate and read format, the Python code necessary for parsing and displaying the results from the testing procedures and the Javascript code required for the generation of the hierarchical site map graph and for the animations.

The ‘generate\_tree.py’ and ‘html\_report.py’ files contain the Python side logic for forwarding the testing and the site map graph data to the HTML template. These files are required for correctly parsing and formatting the data for compatibility with the display code in the HTML file. More on the generated report and methods used in subchapter IV.1 Report Generation.

## The Core files

The Core folder contains the four main files with no testing logic, but with various methods used for interacting with the web application, generating confidence, opening and closing files, setting up prerequisites, error handling, sending requests, managing responses and other utility functions.

### Confidence

This file contains the code necessary for dynamically assigning a confidence level based on a test result. The logic relies on four arguments (*severity, past\_occurences, exploitability and impact*) provided during each test result. All arguments accept values between ‘0.1’ and ‘1’ depending on the test logic.

class Confidence:  
 def \_\_init\_\_(self, severity=0, past\_occurrences=0, exploitability=0, impact=0):  
 self.severity = severity  
 self.past\_occurrences = past\_occurrences  
 self.exploitability = exploitability  
 self.impact = impact  
  
 def add\_confidence(self, severity=0, past\_occurrences=0, exploitability=0, impact=0):  
 self.severity += severity  
 self.past\_occurrences += past\_occurrences  
 self.exploitability += exploitability  
 self.impact += impact  
  
 def calculate\_confidence(self):  
 score = (self.severity \* 0.3) + \  
 (self.past\_occurrences \* 0.3) + \  
 (self.exploitability \* 0.2) + \  
 (self.impact \* 0.1)  
  
 if score >= 0.75:  
 return "Critical"  
 elif score >= 0.5:  
 return "High"  
 elif score >= 0.25:  
 return "Medium"  
 else:  
 return "Low"

During execution of a test, the ‘*Confidence’* object is initialized with the default initial values (0). For each new payload that successfully exploits an input, the ‘*add\_confidence’* method is called and the values are increased accordingly. At the end of the test, the ‘*calculate\_confidence’* method is called and the vulnerability score is determined based on the previous values.

For example, the below vulnerability confidence logic is used for the LFI identification tests:

At the beginning of the testing function, the confidence object is created:

sql\_test\_confidence = Confidence()

Then, if the target is vulnerable to LIF, the ‘*add\_confidence’* method is called with arbitrary values dependent of the testing conditions and the number of times the application is found vulnerable to different inputs.

if "root:" in response.text.lower():  
 sql\_test\_confidence.add\_confidence(severity=0.8, past\_occurrences=0.1, exploitability=0.7, impact=0.7)  
 confidence += 1

After the true positive condition is met, the application calculates the final vulnerability score and returns it to the calling method:

if confidence >= 1:  
 if new\_url:  
 return lfi\_script, new\_url, sql\_test\_confidence.calculate\_confidence()

Each testing method has different confidence values depending on the testing logic used.

### DataStorage

The ‘*DataStorage.py*’ file contains the Python code necessary for accessing the local data in the ‘*Payloads’* folder.

The most important method is the one that accesses the relevant payloads based on the provided input:

@staticmethod  
def payloads(p\_type): *# returns a list of payloads depending on the chosen type* try:  
 *# Get the payload type from the Payload Local Repo.* if p\_type == 'SQL':  
 for filename in os.listdir(os.getcwd() + '/Payloads/SQL'):  
 with open(os.path.join(os.getcwd() + '/Payloads/SQL', filename), 'r', encoding="utf8") as f:  
 DataStorage.sql\_dict[filename.split('.')[0]] = f.read().splitlines()  
 f.close()  
 all\_sql\_values = []  
 for value in DataStorage.sql\_dict.values():  
 if isinstance(value, list):  
 all\_sql\_values.extend(value)  
 else:  
 all\_sql\_values.append(value)  
 return all\_sql\_valueselif p\_type == 'HTML':  
 for filename in os.listdir(os.getcwd() + '/Payloads/HTML'):  
 with open(os.path.join(os.getcwd() + '/Payloads/HTML', filename), 'r', encoding="utf8") as f:  
 DataStorage.html\_inj = f.readlines()  
 f.close()  
 return DataStorage.html\_injelif p\_type == 'XSS':  
 for filename in os.listdir(os.getcwd() + '/Payloads/XSS'):  
 with open(os.path.join(os.getcwd() + '/Payloads/XSS', filename), 'r', encoding="utf8") as f:  
 DataStorage.xss\_inj = f.readlines()  
 f.close()  
 return DataStorage.xss\_inj  
 elif p\_type == "WORDS":  
 for filename in os.listdir(os.getcwd() + '/Payloads/WordLists/'):  
 if 'robots' in filename:  
 continue  
 with open(os.path.join(os.getcwd() + '/Payloads/WordLists/', filename), 'r', encoding="utf8") as f:  
 DataStorage.word\_list.extend(f.readlines())  
 f.close()  
 return DataStorage.word\_list  
 elif p\_type == "LFI":  
 for filename in os.listdir(os.getcwd() + '/Payloads/LFI-RFI'):  
 with open(os.path.join(os.getcwd() + '/Payloads/LFI-RFI', filename), 'r', encoding="utf8") as f:  
 DataStorage.rfi\_inj = f.readlines()  
 f.close()  
 return DataStorage.rfi\_inj  
 elif p\_type == "ROBO":  
 with open(os.path.join(os.getcwd() + '/Payloads/WordLists/', 'robots.txt'), 'r', encoding="utf8") as f:  
 DataStorage.robots\_list = f.readlines()  
 f.close()  
 return DataStorage.robots\_list

On each call, the method accepts the payload type parameter ‘*p\_type’* and according to its value, it provides the list, set or dictionary required for test usage.

This method also contains a special function for the SQL injection payloads designed to also return the type of SQL Injection used, for better detailing in the report:

@staticmethod  
def inject\_type(p\_type):  
 try:  
 *# Based on filename, get the injection type, used for SQL.* for key, value in DataStorage.sql\_dict.items():  
 if isinstance(value, list) and p\_type in value:  
 return key  
 return None

### ScanConfig

The Scanning Configuration file ‘*ScanConfig.py*’ contains the logic for the preliminary tests and setups for the scanner. It contains the ‘*ScanConfig*’ object that once called, initializes the initial parameters (session, lists for errors, the *ignored\_links*, the DataStorage variable and the configuration variable).

class ScanConfig:  
 *# Initialize Session for current run* session = requests.Session()  
 *# Initialize the ignored links and errors file parameters* err\_file = None  
 ignored\_links = None  
 config\_params = config.Config().config\_object  
  
 def \_\_init\_\_(self, url):  
 try:  
 *# Initialize Scanner Configuration Parameters.* self.url = url  
 *# Initialize Data Storage.* self.DataStorage = DataStorage()  
 *# Empty list of links pairs used for hierarchy.* self.link\_pairs = []  
 *# Setup initial requirements and prepare files.* self.setup()

This class is also responsible for setting up the original connection and testing it in the ‘*setup()*’ method.

def setup(self):  
 try: *# Test connection* test\_get = self.session.get(self.url)  
 if test\_get.status\_code == 404: *# If page is not found, scanner will not start.* print("[Error] Bad URL provided (404). Quitting..")  
 quit()  
 except requests.ConnectionError: *# If ConnectionError identified, try to change the HTTP protocol.* try: *# Switch to HTTP if HTTPs is unsupported* self.url = self.url.replace("https://", "http://")  
 self.session.get(self.url)  
 except requests.ConnectionError: *# Switch to HTTPs if HTTP is unsupported* self.url = self.url.replace("http://", "https://")  
 self.session.get(self.url)

The setup method will also attempt correct the URL and try to switch the protocols if the provided one is unsupported. This was done in an effort to allow users to use the scanner even if the application is not publicly available yet nor has a valid ***S****ecure* ***S****ockets* ***L****ayer* (SSL) certificate. The session variable is the one that will be used across the all the scanner logic to maintain communication and interact with the application responses.

### Utilities

The ‘Utilities.py’ method is one of the most referenced methods across the application. This method contains all the major logic for interacting with the web application and all its components. It also contains methods that are used during testing procedures but are not generally applicable to one test, such as the ones used for formatting strings, URLs, accessing cookies, escaping HTML strings or processing the login sequence.

The first implemented method is the ‘*spider*’ function used for harvesting all the available URLs in the application and appending them to the relevant lists (main URL scan list, the link pair hierarchy used for creating the URL graph map and the URLs referencing external domains). At the first call, the provided URL is added to the ‘main URL scan list’ and the ‘link pair list’ as a starting point for the crawling process, for future calls, the application continues with the crawling process for the provided ‘*url*’ variable. This method is called recursively.

def spider(self, url):  
 try:  
 *# Check if this is the first call, and if it is, add the provided URL to the list.* global firstCallSpider  
 if firstCallSpider:  
 self.link\_pairs.append(['-1', url])  
 response = self.session.get(url)self.DataStorage.urls.add(url)  
 firstCallSpider = 0  
 else:  
 response = self.session.get(url)

Then, the method searches for HTML Anchor (<a>) tags and verifies if the newly identified URL is part of the target application. The verification is based on a regex expression designed to match the provided URL domain with the extracted one.

if response.status\_code == 200:  
 soup = BeautifulSoup(response.text, "html.parser")  
 *# Build up URls list by extracting all hrefs and anchors* for link in soup.find\_all('a'):  
 href = link.get('href')  
 if href and not href.startswith('#'):  
 extracted\_url = urllib.parse.urljoin(url, href)  
 *# Ensure the app does not scan other webapps by checking if the harvested URL has the same domain as the URL provided by user.* if re.search("^(?:https?:\/\/)?(?:[^@\/\n]+@)?(?:www\.)?([^:\/?\n]+)", extracted\_url).group(1) == re.search("^(?:https?:\/\/)?(?:[^@\/\n]+@)?(?:www\.)?([^:\/?\n]+)", self.url).group(1):  
 *# Adds URLs to main list object, ignore the user - ignored ones.  
 # [[a.html,b.html], [b.html,c.html], [a,d], [a,g], [g,h], [b,j]]* if extracted\_url not in self.DataStorage.urls:  
 self.link\_pairs.append([url, extracted\_url])  
 if extracted\_url not in self.DataStorage.urls and not any(ignored.replace(" ", "") in extracted\_url for ignored in self.ignored\_links):  
 self.DataStorage.urls.add(extracted\_url)  
 self.spider(extracted\_url)  
 else: html\_report.add\_external\_link(extracted\_url)

Based on the result, if the URL is internal, it is added to the ‘graph URL map list’ and to the ‘main URL to scan list’. The external URLs are added to the ‘external URL list’.

Other methods present under ‘Utilities’ class manage interactions between the scanner and the *inputs* and *forms* on the web application. The method used for extracting the *forms* uses the ‘*BeautifulSoup*’ parser to find and extract all the data under each *form* tag identified on the provided URL.

def extract\_forms(url):  
 try:  
 *# Extract all forms from an URL.* response = ScanConfig.session.get(url, timeout=25)  
 response.raise\_for\_status()  
 parsed\_html = BeautifulSoup(response.content, "html.parser")  
 return parsed\_html.findAll("form")

The same type of logic is used for extracting the *inputs* as well. The *input* tags can be under *form* tags or have no *form* parent. Due to this reason, a separate method is used for extracting all identified inputs as well.

def extract\_inputs(url):  
 try:  
 *# Extract Inputs the same way forms are extracted* response = ScanConfig.session.get(url, timeout=300)  
 response.raise\_for\_status()  
 parsed\_html = BeautifulSoup(response.content, "html.parser") *# , from\_encoding="iso-8859-1")* return parsed\_html.findAll("input")

For extracting data out of *form* tags and to determine potential injection points, few additional methods are used. This was done in an effort to ensure the scanner is able to recognize and extract as much injection opportunities as possible for better test coverage.

The scanner attempts to extract the following tags from a *form* tag: *input*, *button*, *select* and *textarea*. The method creates and returns a ‘*form\_data’* dictionary variable in which the key is the field name and the attribute is left blank (the *button* and *input* tags are additionally checked for the common values needed for web application good functionality [e.g., *submit*]).

Inputs:

input\_fields = form.find\_all('input')  
for field in input\_fields:  
 if field.get('name'):  
 if (str(field.get('name')).lower() == 'submit') or (str(field.get('type')).lower() == 'submit') or (str(field.get('name')).lower() == 'user\_token'):  
 form\_data[field.get('name')] = field.get('value')  
 continue  
 form\_data[field.get('name')] = ''

Buttons:

form\_fields = form.find\_all('button')  
for field in form\_fields:  
 if field.get('name'):  
 if (str(field.get('name')).lower() == 'submit') or (str(field.get('type')).lower() == 'submit') or (str(field.get('name')).lower() == 'user\_token'):  
 form\_data[field.get('name')] = field.get('value')  
 continue  
 form\_data[field.get('name')] = ''

Selects:

form\_option = form.find\_all('select')  
for field in form\_option:  
 if field.get('name'):  
 form\_data[field.get('name')] = ''

Textareas:

form\_textarea = form.find\_all('textarea')  
for field in form\_textarea:  
 if field.get('name'):  
 form\_data[field.get('name')] = ''

This class also contains the necessary methods used for sending data to the application. The method used for submitting the *form* with the payload data for testing is also implemented here.

def submit\_form(url, form, form\_data):  
 try:  
 *# Get the action (URL or PATH)* action = form.get("action")  
 *# Get the method (GET, POST, PUT etc.)* if form.get("method"):  
 method = form.get("method").upper()  
 else:  
 return 0  
 *# Check if action is URL or if it is path relative to URL.* if action:  
 if action.startswith('http'):  
 action\_url = action  
 else:  
 action\_url = urllib.parse.urljoin(url, action)  
 else:  
 action\_url = url  
 *# Send data accordingly to method.* if method == 'GET':  
 response = ScanConfig.session.get(action\_url, params=form\_data, timeout=30)  
 else:  
 response = ScanConfig.session.post(action\_url, data=form\_data, timeout=30)  
 response.raise\_for\_status()  
 return response

The *submit\_form* method is used for checking the request type and the destination of data based on the provided URL, then, after all the checks are performed, the payload data is sent to the application according to the identified *method* and *action*.

In certain situations, injection fields cannot be extracted from newly extracted *form* tags, so another method is used to extract them based on the *form*\_*data* already present in the scanner memory by providing it as an argument.

def extract\_injection\_fields\_from\_form(form\_data):  
 try:  
 keys\_to\_populate = []  
 *# Find the emtpy form values, meaning they await user input, add them to a list and return the list* for key, value in form\_data.items():  
 if form\_data[key]:  
 continue  
 else:  
 keys\_to\_populate.append(key)  
 return keys\_to\_populate

This method receives the *form\_data* as an argument, then it iterates through the available tags. If an empty tag is found, a list with the injection field names is populated and returned.

Another method used for attempting to find injection points is the ‘extract\_non\_form\_inputs’.

def extract\_non\_form\_inputs(self, url):  
 try:  
 name\_list = []  
 action\_urls = []  
 input\_list = self.extract\_inputs(url)  
 for input in input\_list:  
 name\_list.append(self.extract\_name\_value(input))  
 for name in name\_list:  
 action\_urls.append(str(url + '?' + str(name) + '='))  
 return action\_urls

This method is used to extract all the *input* tag names and force them into URL parameters.

The last major method of this class is the method used for extracting special *inputs* either under *form* tags or in the URL path.

def no\_form\_input\_content(url, payload):  
 try:  
 *# Extract inputs outside of form or with no method/action in form* input\_list\_fin = set()  
 form\_data = {}  
 response\_list = []  
 soup = BeautifulSoup(ScanConfig.session.get(url, timeout=300).content, 'html.parser')  
 input\_list = soup.findAll('input')  
 *# Get all inputs that have a form parent but no action or button.* for input in input\_list:  
 if input.find\_parent('form'):  
 parent\_attr = input.find\_parent('form').attrs  
 if not (('action' in parent\_attr) or ('button' in parent\_attr)):  
 input\_list\_fin.add(input)  
 else:  
 *# Get inputs with no form parent* input\_list\_fin.add(input)  
  
 if input\_list\_fin:  
 for field in input\_list\_fin:  
 if field.get('name'):  
 form\_data[field.get('name')] = payload

This method searches for inputs with a *form* parent but with no *action* or *method* defined in the HTML DOM. The inputs are added to a list and returned.

The ‘*Utilities’* class also contains more additional methods used for various purposes such as extracting the headers, preparing XML entities for injections, changing the User-Agent, extracting cookies, finding hidden inputs under DOM, printing the exception messages in color and formatting them based on time and date, extracting XML tags, escaping HTML encoded characters, creating average response time based on the request dwell time. All of these methods will be referenced in the following subchapters under each related testing procedure.

## The Test files

Each test has a designated Python file in which all the logic relevant to that test is contained. The testing methods apply the principles presented in the *Chapter III. Vulnerabilities and Misconfigurations* All the implemented testing functions apply the identification methods used by *white-hat* hackers but also by the attackers’ automated scanning tools. Each test has a primary method called *‘run()*’ used to call the testing logic for the specific file. Each file can have one or more additional methods used for various testing logic.

### SQL Tests

The SQL test file contains four methods used to identify the presence of SQL vulnerabilities. The first method is the standard method predominantly used in all future injection tests. This method contains the logic required for extracting forms and inputs from the provided URL using the ‘Utilities’ module.

def t\_i\_sql(url, form, form\_data):

injection\_keys = Utilities.extract\_injection\_fields\_from\_form(form\_data)

for sql\_payload in DataStorage.payloads("SQL"):  
 *# Populate injection keys with payloads.* for injection\_key in injection\_keys:  
 form\_data[injection\_key] = sql\_payload

Each payload from the DataStorage payload list is attempted as an test string. Depending on the response results, two types of SQL vulnerabilities can be found. The reflected type is usually containing a message with error or the injected payload itself.

if ("error" in response\_injected.text.lower() and 'error' not in ScanConfig.session.get(url)) or (  
 'error' in ScanConfig.session.get(url) and sql\_payload in response\_injected.text.lower()):  
 confidence += 1

The time-based test is done using the average response time. The testing payload contains instructions designed to make the SQL database hang execution (sleep) for minimum two seconds.

if payload\_response\_time > avg\_response\_time and payload\_response\_time > 2:  
 *# Vulnerable to Time based SQL type X, increase confidence* sql\_test\_confidence.add\_confidence(severity=0.8, past\_occurrences=0.1, exploitability=0.4, impact=0.4)  
 confidence += 1

To prevent the scanner from over-testing, a threshold of maximum four true positive reflected and one time-based attempt is conditioned as a stop flag.

if confidence == 4:  
 forms, form\_data = Utilities.extract\_from\_html\_string('form', response\_injected.text)  
 return sql\_payload, sql\_type\_list, forms, sql\_test\_confidence.calculate\_confidence()  
  
if confidence >= 1:  
 return sql\_payload, sql\_type\_list, form, sql\_test\_confidence.calculate\_confidence()

The SQL test method returns the SQL payload used, the SQL payload type, the vulnerable form and the test confidence (note: the ‘confidence’ variable is used just for threshold purposes and does not influence the actual confidence presented in the final report).

The second type of SQL testing is the one using the ‘non-form input’ testing method from the Utilities class. The same type of logic is applied here as well, with the only exception being the input fields the tests are performed for.

def t\_i\_sql\_nfi(url):

response\_injected = Utilities.no\_form\_input\_content(url, sql\_payload)  
if not response\_injected:  
 continue  
for response\_inj in response\_injected:  
 if response\_inj.elapsed.total\_seconds() > 4.5:  
 sql\_test\_confidence.add\_confidence(severity=0.8, past\_occurrences=0.1, exploitability=0.5, impact=0.7)  
 return sql\_payload, sql\_test\_confidence.calculate\_confidence()

The third SQL testing method is based on User-Agent. For this attempt, the scanner injects the testing payloads into the headers of the request. The identification conditions are the same with the above methods.

for sql\_payload in DataStorage.payloads("SQL"):  
 *# Inject headers with payloads* headers = Utilities.custom\_user\_agent(sql\_payload)  
 try:  
 response = ScanConfig.session.get(url, timeout=30, headers=headers)

The last type of SQL test method is the XML injection with a SQL Payload. For this attempt, the scanner attempts to identify and inject XXE prepared data with SQL payloads into the request. For this, the testing method uses the ‘prepare\_xml\_inj’ method from the Utilities class, this method is used only for SQL and XML injection tests. The’‘prepare\_xml\_inj’ method has two parameters (URL and payload). This method uses RegEx to attempt to extract XML object from the page (using the ‘extract\_xml\_tags()’ method from Utilities module), modify them according to the provided parameters and return them back to the SQL testing method.

Extraction:

pattern\_tags = r'xmlHttp\.send\("([^"]+)"\);'  
pattern\_uri = r'xmlHttp\.open\("POST","([^"]+)"'  
match\_tags = re.search(pattern\_tags, ScanConfig.session.get(url).text, re.DOTALL)  
match\_uri = re.search(pattern\_uri, ScanConfig.session.get(url).text, re.DOTALL)

Preparing the SQL XXE payload:

This method also handles the situation if the application requires a specific URL to send the XML data.

if extracted\_uri and extracted\_tag:  
 *# Get the first content of the first identified tag and inject it with the custom XXE variable* prepared\_tag = re.sub(r'(<[^>]+>)([^<]+)(</[^>]+>)', r'\1' + '&XXE;' + r'\3', extracted\_tag, count=1)  
 *# Prepare the payload with specific requirements such as identified tags.* xml\_payload = '''<?xml version="1.0" encoding="utf-8"?>  
 <!DOCTYPE root [<!ENTITY XXE SYSTEM "{}"> ]>  
 {}'''.format(payload, prepared\_tag)  
  
 *# Prepare URL for injection, URL consist of custom POST request of XML.* pattern\_url = r'(.\*/)[^/]+$'  
 prepared\_url = re.sub(pattern\_url, r'\1' + extracted\_uri, url)  
  
 *# Return the pre-build URL and the custom XML payload* return prepared\_url, xml\_payload

SQL method sending XXE payload to custom URL:

if prepared\_url and custom\_payload:  
 try:  
 response = ScanConfig.session.post(prepared\_url, data=custom\_payload,  
 headers={'Content-Type': 'application/xml'})

After the data is sent to the application, the same verification conditions as in the previous SQL testing method are used to validate the SQL vulnerability.

All the previous testing methods are called in the ‘run()’ method under ‘InjectionSql’ file. The ‘run()’ method also handles the call to the report file providing the necessary information according to each test.

payload, xml\_url, confidence = t\_i\_xml\_sql(url)  
if payload:  
 payload = Utilities.escape\_string\_html(encoded\_single=payload)  
 html\_report.add\_vulnerability('SQL Injection in XML tag',  
 'SQL Injection in XML tag vulnerability identified on URL: {} using custom XML tags.'.format(  
 url), confidence, payload=payload, comment="Used custom URL for XML SQL Injection URL: {}.".format(xml\_url))

### XSS Tests

The XXS tests are very similar with the SQL ones. The only difference is that XSS does uses specific XSS payloads and does not test for XXE injections with XSS payloads, as XSS is only run in the client’s browser. The XSS vulnerability validation condition is based on the existence of the XSS payload into the application response. If the application response contains the unmodified payload string, it means that the data was successfully interpreted by the HTML page, since any tag on the page is valid for the HTML interpreter. Similar with the SQL test, this tests also has a threshold of four true positive identifications.

if (str(xss\_payload).lower() in str(response\_injected.text).lower()) or (  
 str(xss\_payload).lower() in str(html.unescape(response\_injected.text.lower()))) or str(html.unescape(str(xss\_payload).lower()) in str(response\_injected.text).lower()):  
 confidence += 1

### HTML Tests

Also, very similar with the previous two testing practice, the HTML Injection is also performed on the standard inputs and only on the ‘non-form inputs’. Due to their nature, HTML injection cannot be done on User-Agents or XXE tags, so testing for those is not required. The vulnerability validation is performed the same way the XSS one is, by confirming the existence of the HTML injected payload into the application response.

if html\_payload.lower() in response\_injected.text.lower():  
 confidence += 1

The same threshold rules apply to HTML injections as well.

### Code Execution Tests

Code execution tests are performed only for standard and ‘non-form’ inputs. Unlike the previous testing methods, the code execution test does not contain a DataStorage object for storing payloads. For this test, the injected payload is a UNIX command, preceded by a pipe (|), that once executed pings the loopback address (127.0.0.1) for 3 seconds.

code\_exec\_payload = "|ping -c 3 127.0.0.1"

The pipe (|) character is used to chain a series of commands under the same CLI line.

Since it is a time-based detection, this method will ensure that both reflected and time-based Code Execution vulnerabilities are detected:

for injection\_key in injection\_keys:  
 form\_data[injection\_key] = html.unescape(code\_exec\_payload)  
response = Utilities.submit\_form(url, form, form\_data)

The vulnerability validation is performed by comparing the average response time with the injected response one.

if response.elapsed.total\_seconds() > 1.5 and response.elapsed.total\_seconds() > avg\_response\_time:  
 forms, form\_data = Utilities.extract\_from\_html\_string('form', response.text)

This testing method does not require a threshold as this is the only attempted payload for this test.

### iFrame Tests

For iFrame testing, the application attempts to extract all available iFrames from the web page using the ‘extract\_iframe()’ method.

def extract\_iframes(url):  
 *# Extract iFrames the same way forms are extracted* response = ScanConfig.session.get(url, timeout=300)  
 response.raise\_for\_status()  
 parsed\_html = BeautifulSoup(response.content, "html.parser"return parsed\_html.findAll("iframe")

From the extracted iFrame, the source value in the URL is modified to a custom one provided as payload.

If iframe[‚src’] in url:  
 url = url.replace(iframe[‚src’], payload)

The payload value is actually the ‘https://www.google.com' URL. The testing method then sends a get request to the modified URL. For validation, the iFrame payload is searched in the response body. If the value is found, the application is vulnerable to iFrame injection on that specific URL.

if iframe\_payload in ScanConfig.session.get(iframe\_url).text.lower():  
 return iframe\_url

### Javascript Test

Javascript code injection tests are very similar with the Code Execution tests, the only difference is the payload used and the method of injection. For JS Injection, the URL is brute forced with the JS payload. The JS payload is provided as a parameter in the URL either by replacing already existing parameters or by concatenating the JS payload to the URL, as the payload already contains the parameter definition symbol (?).

js\_payload = '/?javascript:alert("testedforjavascriptcodeexecutionrn3284")'

if url[-1] != '/':  
 new\_url = url + js\_payload

else:  
 new\_url = url + js\_payload[1:]

For vulnerability validation, the JS payload is searched in the response body, if the payload is identified the application is considered vulnerable.

### IDOR Tests

There are two chosen methods for IDOR testing. One simple method is to check for the existence of hidden elements within the body of the HTML DOM. This test will not always return conclusive or true positive results as many tags can be hidden by default, however, there are situations in which developers forget hidden tags with valuable object references in the body of the DOM. The hidden tag method is only searching for tags containing the ‘hidden’ keyword in their definition, then it returns them to the report as potential IDOR vulnerabilities.

for input in inputs:  
 if input['name'] in form\_data:  
 if input['type'] == 'hidden':

A more complex method of testing consists of searching for objects referenced by numbers in the URL (e.g., ?query=test&value=123&other=4). The search is based on Regex.

*# Capture everything after the last '?'*match = re.search(r'[^?]\*$', url)

Then, for the identified string, multiple other Regex rules are used to extract and modify the numbers referenced in the parameters.

int\_groups = re.findall(r'=(\d+)', str(sub\_string))  
for grp in int\_groups:  
 index\_from\_url = int(str(grp)) *# 123* sub\_string = sub\_string.replace(str(index\_from\_url), str(index\_from\_url + 5))

After the numbers are extracted and modified, a new request is done for the new modified URL. There is a maximum of only ten attempts until the testing result come back negative.

while attempts < 10:  
 try:  
 url = url.replace(original\_sub\_string, sub\_string)  
 response\_2 = ScanConfig.session.get(url)  
 if response.text != response\_2.text and str(response\_2.status\_code).startswith("2"):

If the application breaks at any point before hitting ten failed attempts, the URL is returned back and added to the report.

response\_url = t\_idor\_nfi(url)  
if response\_url:  
 html\_report.add\_vulnerability('IDOR', 'Insecure direct object reference (IDOR) vulnerability identified. URL: {}'.format(url), 'Medium', comment="Got reply from {}. If the URL is referencing objects, the application is vulnerable to this injection.".format(response\_url))

### LFI Tests

LFI testing method is very similar with the SQL and XSS tests. All available inputs and URL parameters are injected with LFI Payloads from the PayloadsAllTheThings GitHub page [69]. The LFI script used, the vulnerable form or URL and the confidence are printed back into the report.

### PHP Tests

PHP testing methods have similarities with Code and Javascript injections. During this testing procedure, the ‘%60ping%20-c%203%20127.0.0.1%60‘ command is used for all injection points available. This command is HTML encoded for better detection chance.

### SSI Tests

For SSI testing, the ‘<!--#exec cmd=netstat -->‘ directive is injected in all the extracted injection point with exception to URLs. This injection payload was chosen as it will always return the phrase ‘active internet connections’ when executed on UNIX based systems. This phrase is also the one that is searched for during test validation.

### SSRF Tests

This test consists of the injection of two payloads only in the URL parameters, after the ‘=’ character and validating the response accordingly.

ssrf\_payload = https://www.google.com’

The status code of the response is checked for validation. A response of code 200 is considered a true positive.

For the second payload, the value of ‘root’ which is always present when the injected command is executed successfully is checked for validation.

ssrf\_payload = 'file:///etc/passwd'

### XML Tests

For the XML injection test, the same method used in SQL XEE testing prepare\_xml\_inj was used here. The injection payload is also the 'file:///etc/passwd' command.

custom\_url, custom\_payload = Utilities.prepare\_xml\_inj(url, payload)  
if custom\_url and custom\_payload:  
 response = ScanConfig.session.post(custom\_url, data=custom\_payload, headers={'Content-Type': 'application/xml'})  
 custom\_payload = Utilities.escape\_string\_html(encoded\_single=custom\_payload)  
 if 'root' in response.text.lower():

### Misconfiguration Tests

For the misconfiguration tests, checking was performed on the previously mentioned data configurations in III.1 Security Misconfigurations chapter.

The cache settings in the headers were checked for the presence of the recommended values.

if "Cache-Control" in str(response.headers):  
 if (response.headers["Cache-Control"] != "no-store" and response.headers[  
 "Cache-Control"] == "no-cache, must-revalidate") or \  
 (response.headers["Cache-Control"] == "no-store" and  
 response.headers["Cache-Control"] != "no-cache, must-revalidate"):

The CORS misconfiguration is validated by reviewing the access rights for the ‘Access-Control-Allow-Origin’ header flag:

if ScanConfig.session.get(url).headers['Access-Control-Allow-Origin'] == '\*':

HHI was tested by performing an injection on the ‘Host’ and ‘X-Forwarded-Host’ flags with ‘google.com’ domain. For validation the response status code 200 is considered a true positive finding.

host = {'Host': 'google.com'}  
host\_injection = ScanConfig.session.get(url, headers=host)  
x\_host = {'X-Forwarded-Host': 'google.com'}  
x\_host\_injection = ScanConfig.session.get(url, headers=x\_host)

RIA validation was performed by checking the access permissions of the ‘'clientaccesspolicy.xml' and 'crossdomain.xml'files. The ‘\*’ access setting is considered overly permissive.

The ‘robots.txt’ Disallow attribute contents were also checked using a list of the most common sensitive file paths string [70].

req\_robots = ScanConfig.session.get(url\_robots)  
robots\_urls = re.findall('Disallow: (.\*)', req\_robots.text)  
if robots\_urls:  
 for sensitive in DataStorage.payloads('ROBO'):  
 sensitive\_list.update(robot for robot in robots\_urls if sensitive.strip() in robot)

If a sensitive path is found, the information gets added to the report as a potential sensitive data disclosure source.

The TRACE and PUT method tests were performed by creating a custom request with dummy data. Each request was sent with the TRACE and PUT method. Generally, web applications should not accept these methods unless they have a specific reason.

trace\_request = requests.Request('TRACE', url)  
prepared\_trace = trace\_request.prepare()

response = ScanConfig.session.put(str(main\_url) + '/test.html', data={"test": 'test'})

Both validations are performed by assuming the 200 status code response is a true positive.

### Other detections and features

The scanner was also implemented to detect possible misconfiguration that are not defined by hard standards, these tests are to be considered useful for providing more informational data that can be reviewed by the tester. This data can be used for further manual testing.

#### Comment Extraction

The first informational feature is the extraction of all the comments from the all the accessed URL as adding them to the report file. The method is simply looking for the HTML comment tags ('(?<=<!--)(.\*)(?=-->)') and adding all the findings under a dictionary file, later to be populated in the report.

comments = re.findall('(?<=<!--)(.\*)(?=-->)', str(ScanConfig.session.get(url).text))  
comm\_dict.update({url: comments})

#### Directory Transversal

The second test is done by performing a directory transversal attempt using a popular list for URL subdirectories [71]. The URL is concatenated with the values extracted from the list, if the received response has status code 200, the new URL is considered valid and the result is added to the report.

word\_list = DataStorage.payloads('WORDS')  
for word in word\_list:  
 if url[-1] != '/':  
 new\_url = url + "/" + word  
 else:  
 new\_url = url + word

#### URL Graph Map

The third informational functionality is provided by the method used for generating a URL hierarchy graph map used for mapping the web applications endpoints.

This method is only used when the scanner is set to run in the full scan mode (more on this in the next chapter – Application Usage). During the ‘spider’ procedure, each new URL and its parent are added to a list of link pairs.

*# Add URLs to main list object, ignore the user - ignored ones.  
# [[a.html,b.html], [b.html,c.html], [a,d], [a,g], [g,h], [b,j]]*if extracted\_url not in self.DataStorage.urls:  
 self.link\_pairs.append([url, extracted\_url])

After the application’s URLs are fully identified, the method used for generating the tree graph is called with the list provided as an argument.

def generate\_tree(list):  
 try:  
 node\_dict = {}  
 tree = {'name': 'root', 'children': []}  
 for pair in list:  
 if pair[0] not in node\_dict:  
 second\_node = {'name': pair[1], 'children': []}  
 node = {'name': pair[0], 'children': [second\_node]}  
 tree['children'].append(node)  
 node\_dict[pair[0]] = node  
 node\_dict[pair[1]] = second\_node  
 else:  
 node = node\_dict[pair[0]]  
 second\_node = {'name': pair[1], 'children': []} if pair[1] not in node\_dict else node\_dict[pair[1]]  
 if pair[1] not in node\_dict:  
 node\_dict[pair[1]] = second\_node  
 node['children'].append(second\_node)  
 return json.dumps(tree)

This method processes the list received from the spider method and transforms them in a JSON object to be parsed to the Javascript code in the report template: *[[a,b], [b,c], [a,d], [a,g], [g,h], [b,j]]* >>{'name': 'root', 'children': []}. This translates to the ‘name’ variable containing the parent of the URL and the children variable containing all the children of the parent URL.

The report.html file contains the Javascript code required for adding the arrows and processing the hierarchization for the graphical view of the URL site map.

function *printTree*(  
 initialTree,  
 printNode,  
 getChildren,  
) }  
 function printBranch(tree, branch) {  
 const isGraphHead = branch.length === 0;  
 const children = getChildren(tree) || [];  
 let branchHead = '';  
 if (!isGraphHead) {  
 branchHead = children && children.length !== 0 ? '┬ ' : '─ '  
 }  
 const toPrint = printNode(tree, `${branch}${branchHead}`);  
 if (typeof toPrint === 'string') {  
 if (toPrint !== '<a href="-1">-1</a>') {  
 const outputDiv = document.getElementById('output');  
 outputDiv.innerHTML += `${branch}${branchHead}${toPrint}\n`;  
 }  
 }  
 let baseBranch = branch;  
 if (!isGraphHead) {  
 const isChildOfLastBranch = branch.slice(-2) === '└─';  
 baseBranch = branch.slice(0, -2) + (isChildOfLastBranch ? ' ' : '│ ');  
 }  
 const nextBranch = baseBranch + '├─';  
 const lastBranch = baseBranch + '└─';  
 children.forEach((child, index) => {  
 printBranch(child, children.length - 1 === index ? lastBranch : nextBranch);  
 });  
 }  
 printBranch(initialTree, '');  
}  
const tree = JSON.parse('{{tree\_json}}');  
*printTree*(tree, node => node === 'root' ? `root` : `<a href="${node.name}">${node.name}</a>`, node => node.children,)

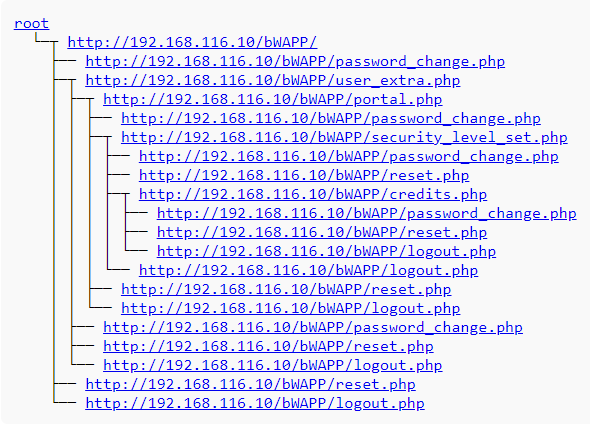
**

Figure 4.4 Example view of the resulted URL Graph Hierarchy

#### CVE Identification

The last information method is the CVE identification procedure. This procedure uses the NIST Vulnerability Database API to perform a version check for the identified software solutions running on the web application.

For this test, the process was split into to phases, the first phase is the software identification steps required for the extraction of the software and version running on the web application. The second step is responsible for checking and reporting the identified CVEs to the report file.

def get\_apps(url):  
 response = ScanConfig.session.get(url)  
 if response.status\_code != 200:  
 return  
 wappalyzer = Wappalyzer.latest()  
 webpage = WebPage.new\_from\_response(response)  
 return wappalyzer.analyze\_with\_versions(webpage)

For software identification, the Python module Wappalyzer was used to return all the available software and versions running on the web server. A dictionary variable is created containing the application software name and the identified version (format: ‘app\_name’: ‘version’).

Before the CVE can be identified, the data model for API interaction needs to match the required format. The software application and its version need to respect the ***C****ommon* ***P****latform* ***E****numeration* (CPE) format. “CPE is a structured naming scheme for information technology systems, software and packages” [72].

The nvdlib Python module was used to interact with the NIST Vulnerability Database.

def get\_cpes(app\_name, version):  
 API\_KEY = ScanConfig.config\_params['API']['API\_KEY']  
 try:  
 cpe\_list = set()  
 cpes = nvdlib.searchCPE(keywordSearch=app\_name + ' ' + version, keywordExactMatch=True, key=API\_KEY ,delay=1)  
 for cpe in cpes:  
 cpe\_array = str(cpe.cpeName).split(':')  
 cpe\_id = ':'.join(cpe\_array[:6])  
 if (':' + str(version) + ':') in str(cpe.cpeName) and (cpe\_array[3] == app\_name or cpe\_array[4] == app\_name) and len(list(filter(lambda x: x.startswith(cpe\_id), cpe\_list))) == 0:  
 cpe\_list.add(cpe.cpeName)  
 return cpe\_list

For each identified software and version, the corresponding CPE is added to a list. After all the identified CPEs matching the parameters are found, the list is returned.

For CVE extraction, the same module is used for getting all the available CVEs associated with the identified CPEs.

def get\_cve(cpe\_name):  
 API\_KEY = ScanConfig.config\_params['API']['API\_KEY']  
 try:  
 cve\_dict = {}  
 cves = nvdlib.searchCVE(cpeName=cpe\_name, limit=5, key=API\_KEY, delay=1)  
 for cve in cves:  
 cve\_dict[cpe\_name] = [str(cve.id), html.unescape(str(cve.descriptions)), str(cve.url), str(cve.score)]  
 return cve\_dict

The identified CVEs and their details are then added to the HTML report.

## Report Generation

In this new version of the web scanner, the report is presented in an easy to read and interactive HTML report. The report contains all the results and their details associated with each finding.

As previously mentioned, the reporting method is based on the Jinja2 templating module. The temple file (report.html) consists of a combination of Python, HTML, CSS and Javascript code required for displaying and dynamically populating all the results and details on the final report file. The reporting template contains Python code required for dynamically interacting with the results returned from the testing procedures. The report consists of six categories of reporting data (vulnerability report based on tests, the URL site map graph, the DOM comments, the external URLs, the hidden URLs and the CVE report). Each category is dynamically populating the data according to the resulted lists from each corresponding testing procedure. Under each HTML tag, the Python Jinja2 code logic is used to interact with the provided lists.

Below is the code required for displaying all the resulted data from the CVE identification method. First, the dictionary variable holding the CVE data is checked if it is populated, if true, the relevant HTML header tag is displayed. Iterations are done to extract and populate the data fields from the dictionary to the relevant HTML tags.

<div class="container">  
 {% if cve\_dict %}  
 <h4>CVE Scan Result:</h4>  
 {% endif %}  
 {% for cve in cve\_dict %}  
 <div class="vulnerability">  
 <div class="severity">ID: {{ cve.cve\_id }}</div>  
 <div class="type">Provider: {{ cve.app\_name\_supp }}</div>  
 <div class="type">Application: {{ cve.app\_name\_prod }}</div>  
 <div class="details">  
 <div class="cve-overview">  
 <strong>Details:</strong>  
 <div class="cve-overview-right">  
 <span style="color:rgba(0, 0, 0, 0.4)">Score <span class="score-color"> {{ cve.cve\_score }} </span></span>  
 &nbsp;  
 &nbsp;  
 &nbsp;  
 <span style="color:rgba(0, 0, 0, 0.4)" >Severity <span class="severity-color">{{ cve.cve\_severity }}</span></span>  
 </div>  
 </div>  
 <p>{{ cve.cve\_description }}</p>  
 <p>CVE URL: <a href={{cve.cve\_url}}>{{ cve.cve\_url }}</a></p>  
 </div>  
 </div>  
 {% endfor %}  
</div>

**HTML Result:**

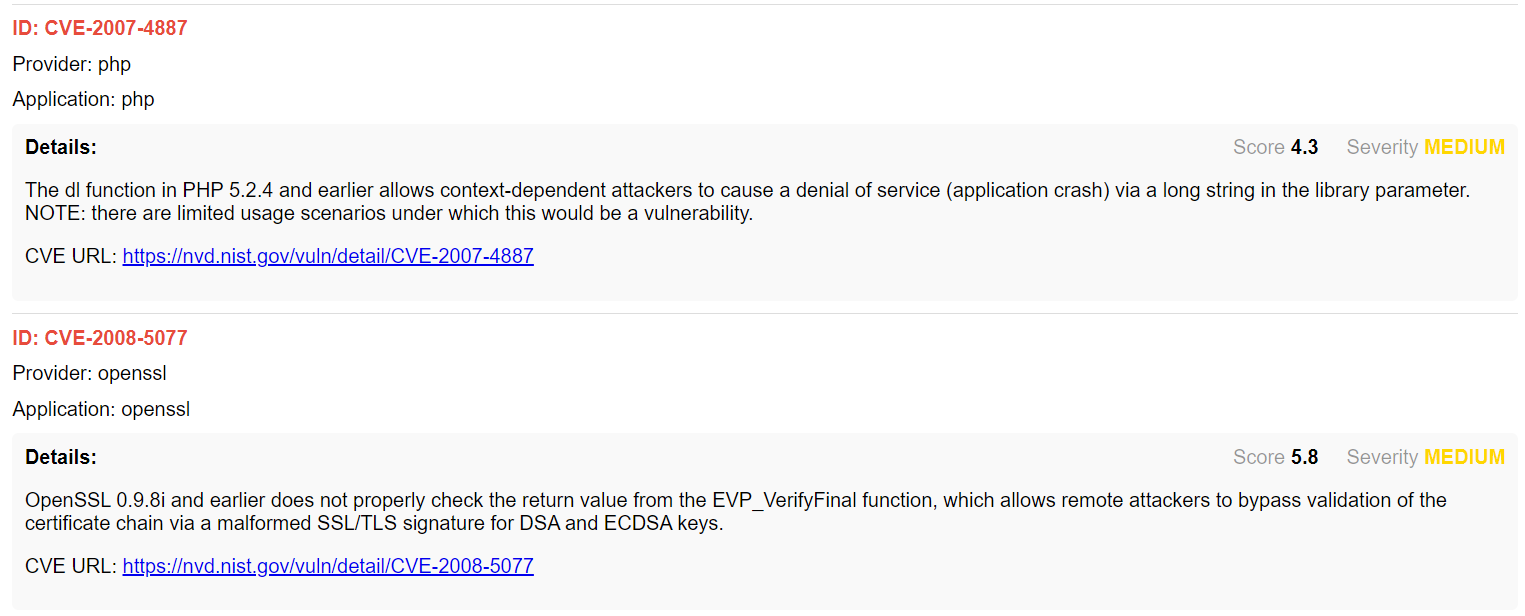


Figure 4.5. CVE category results

The variables available in the HTML template are parsed through Jinja2 module from the ‘html\_report.py’ file. This file contains the relevant code used for populating the list or dictionary variables that are then parsed to the report.

def add\_vulnerability(v\_type, details, confidence='Critical', payload=None, reply=None, comment=None):  
 vulnerabilities.append({'type': v\_type, 'details': details, 'severity': confidence, 'dropdown\_details': {'payload': payload, 'reply': reply, 'comment': comment}})  
 return  
def create\_tree(tree\_list):  
 try:  
 global tree\_json  
 tree\_json = generate\_tree.generate\_tree(tree\_list)  
 return  
 except Exception as e:  
 print(e)  
def add\_external\_link(external):  
 global external\_links  
 external\_links.add(external)  
 return  
def add\_non\_accessible\_link(external):  
 global non\_accessible\_links  
 non\_accessible\_links.add(external)  
 return  
def add\_comments(external):  
 global comments\_dict  
 comments\_dict.update(external)  
 return  
def add\_hidden\_path(path, status\_code):  
 global hidden\_paths  
 hidden\_paths.append({'path': path, 'status\_code': status\_code})  
 return

Parsing the variables based on the report template using the jinja2 module:

env = Environment(loader=FileSystemLoader('Report/templates'))  
template = env.get\_template('report.html')  
*# Render the template with the data*output = template.render(vulnerabilities=vulnerabilities, tree\_json=tree\_json, external\_links=external\_links, non\_accessible\_links=non\_accessible\_links,  
 comments\_dict=comments\_dict, hidden\_paths=hidden\_paths, cve\_dict=cve\_dict, timestamp=datetime.utcnow().strftime("%m/%d/%Y, %H:%M:%S"))

The Javascript code present in the report template is used for creating dynamic scroll-down details, coloring confidentiality results and generating the URL graph map view.

const severityColors = {  
 'LOW': 'grey',  
 'MEDIUM': 'gold',  
 'HIGH': 'red',  
 'CRITICAL': 'purple'  
}  
window.onload = () => {  
 document.querySelectorAll('.severity-color').forEach(d => d.style.color = severityColors[d.innerHTML])  
  
}  
function *toggleDropdown*(event) {  
 event.target.querySelector('.dropdown').classList.add('show');  
 event.target.querySelector('.plus').innerHTML = '-'  
}  
window.onclick = function (event) {  
 if (!event.target.closest('.details')) {  
 document.querySelectorAll('.plus').forEach(d => d.innerHTML = '+');  
 document.querySelectorAll('.dropdown').forEach(d => d.classList.remove('show'));

**HTML Result:**

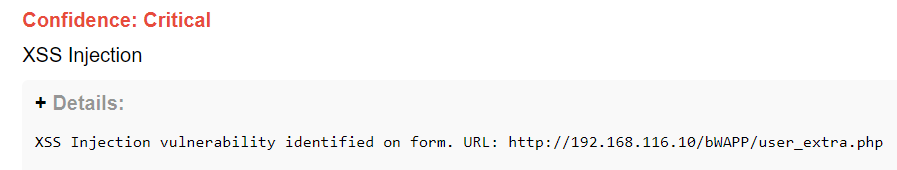


Figure 4.6. XSS Test result compact view

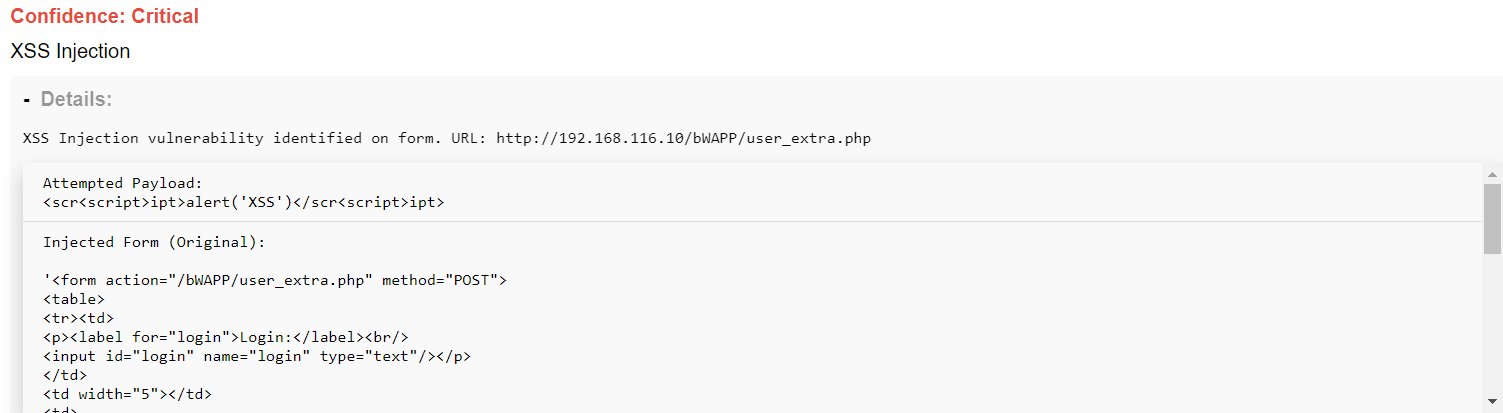


Figure 4.7. XSS Test result drop-down view (clicked)

The final report file is generated in the same folder the script is executed from. The format of the report file is ‘Vulnerability\_Report\_YYYY-MM-DD\_HH-MM.html’. The report file can be opened in any web browser.



Figure 4.8. Report file

In the CLI view, the scanner displays relevant information according to the scanning process.

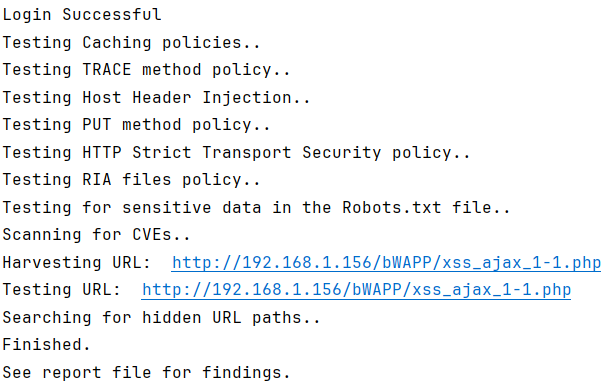


Figure 4.9 CLI view of the scanning progress (single URL scan)

## Application usage

The scanner software can be downloaded from my GitHub Page: (<https://github.com/Ptmlol/WebScanner2>) [73]. The scanner application can only be run from a Linux distribution.

The usage syntax is: *Scanner.py [Target URL].* Additional flags can be used according to the targeted application requirements. -u [Username] and -p [Password] flags need to be provided for bypassing the login page of the targeted website. The special flag -s (static) can be used for single URL scanning. When this flag is specified, the scanner will only test the provided URL. If not provided, the scanner will run in the comprehensive mode and will attempt to crawl and test all the web application endpoints. The -h (help) flag can be used for additional information.

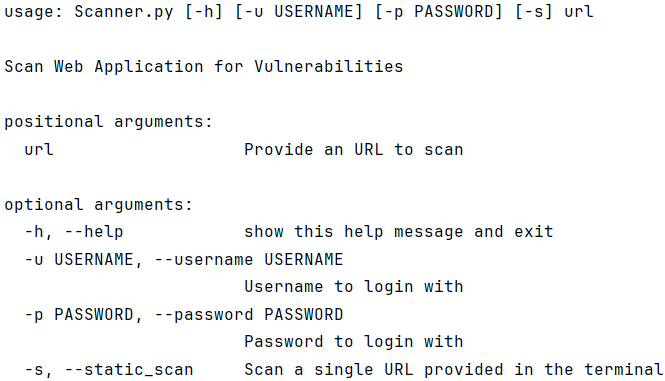


Figure 4.10. Available flags and usage syntax.

# Conclusions, comparisons and future enhancements

Compared to the previous version, this new scanner implementation is now targeted for developers and cybersecurity enthusiasts with and without hands-on experience. The new scanner version provides more detailed vulnerability reporting and, with the additional informational features, it can assist the user in the process of determining the risk exposure of the targeted web application. This project implementation is solely based on my experience and observations from past web application penetration testing during CTF challenges. This new scanner can be visualized as a collection of multiple tools with specific purposes that, depending on the user’s requirements, can be combined or used separately for obtaining the desired results. The scanner is intended for assisting its user and helping him gain the initial starting points that need to be further manually reviewed and validated.

Compared to other web scanning applications, this project provides much more information and customable testing payloads and procedures for the already implemented detections. This scanner is easy to use, requires minimal reconfiguration, is open-source and free. It also provides two methods of scanning based on the needs of the user. At the time this project was built, the most popular web scanners are: WPScan, Probely, OPENVas, Acunetix and Tenable Nessus [74]. WPScan provides best results for WordPress based web application, this solution requires the user to set up an account and register for an API use, the application requires additional configurations and it’s not effective for non-WordPress websites. Probely and OPENVas are very slow and can take more days to complete a single scan, also those require heavy pre-configurations, also the free versions do not provide full test coverage as the current project. Acunetix and Tenable Nessus are also great scanners but very expensive, also they require much more setup requirements and require much more on-prem resources and technical knowledge for usage. Compared to Acunetix, Tenable Nessus and Probely, this scanner project is detecting the same number of vulnerabilities findings (with exception to Brute-Force and File Upload vulnerabilities).

Future enhancements consist of:

* Creating more tests specific for the new frameworks.
* Creating automated updating logic for payload repositories.
* Implementing a Database for Payload and report storage.
* Creating docker container for the project.
* Refining current testing logic to reduce false positive findings.
* Creating chained execution testing.
* Allowing the user to create his own testing conditions.
* Embedding AI into the application for more accurate identifications.

Finally, I want to underline the importance of having multiple automated vulnerability scanning tools available. As the adversaries are continuously changing their tactics, web application scanners need to be as unique and accurate as possible while also be available for usage for efficient identification of security weaknesses in a timely manner [1].

**References**

|  |  |
| --- | --- |
| [1] | T. Naiboiu, Web Application Vulnerability Scanner, Bucharest: University Of Bucharest, 2021. |
| [2] | CISA, 1 February 2021. [Online]. Available: https://www.cisa.gov/news-events/news/what-cybersecurity. [Accessed 1 August 2024]. |
| [3] | BUGTI, "Website Hacking Statistics 2024," BUGTI, 07 05 2024. [Online]. Available: https://www.bugtiwallet.com/website-hacking-statistics-2024/. [Accessed 1 August 2024]. |
| [4] | A. Volle, "Web application," [Online]. Available: https://www.britannica.com/topic/Web-application. [Accessed 1 August 2024]. |
| [5] | W. I. a. W. Browser?. [Online]. Available: https://www.lifewire.com/what-is-a-browser-446234. [Accessed 1 August 2024]. |
| [6] | "OWASP," [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed 1 August 2024]. |
| [7] | K. Crawley, 10 06 2021. [Online]. Available: https://www.hackthebox.com/blog/what-is-ctf. [Accessed 1 August 2024]. |
| [8] | "What is Python?," [Online]. Available: https://www.qamadness.com/knowledge-base/python-for-automation-why-is-it-great-and-how-to-use-it/. [Accessed 1 August 2024]. |
| [9] | "Beautiful Soup Documentation," [Online]. Available: https://beautiful-soup-4.readthedocs.io/en/latest/. [Accessed 1 August 2024]. |
| [10] | "Requests: HTTP for Humans," [Online]. Available: https://docs.python-requests.org/en/latest/index.html. [Accessed 1 August 2024]. |
| [11] | "URL handling modules," [Online]. Available: https://docs.python.org/3/library/urllib.html. [Accessed 1 August 2024]. |
| [12] | "What is Python?," [Online]. Available: https://www.python.org/doc/essays/blurb/. [Accessed 1 August 2024]. |
| [13] | "Why is Python Important for Cybersecurity?," [Online]. Available: https://www.scaler.com/topics/cyber-security/cybersecurity-importance-for-python/. [Accessed 1 August 2024]. |
| [14] | "urllib.parse — Parse URLs into components," [Online]. Available: https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse. [Accessed 1 August 2024]. |
| [15] | "urllib.request — Extensible library for opening URLs," [Online]. Available: https://docs.python.org/3/library/urllib.request.html#module-urllib.request. [Accessed 1 August 2024]. |
| [16] | "What are Threads in Computer Processor or CPU?," [Online]. Available: https://www.geeksforgeeks.org/what-are-threads-in-computer-processor-or-cpu/. [Accessed 1 August 2024]. |
| [17] | "os - Miscellaneous operating system interfaces," [Online]. Available: https://docs.python.org/3/library/os.html. [Accessed 1 August 2024]. |
| [18] | "configparser - Configuration file parser," [Online]. Available: https://docs.python.org/3/library/configparser.html. [Accessed 1 August 2024]. |
| [19] | "A Practical Guide to Regular Expressions," [Online]. Available: https://www.freecodecamp.org/news/practical-regex-guide-with-real-life-examples/. [Accessed 1 August 2024]. |
| [20] | "argparse — Parser for command-line options, arguments and sub-commands," [Online]. Available: https://docs.python.org/3/library/argparse.html. [Accessed 1 August 2024]. |
| [21] | "Escape sequences," [Online]. Available: https://en.cppreference.com/w/cpp/language/escape. [Accessed 1 August 2024]. |
| [22] | "Jinja," [Online]. Available: https://jinja.palletsprojects.com/en/2.10.x/. [Accessed 1 August 2024]. |
| [23] | "NVDLib: NIST National Vulnerability Database API Wrapper," [Online]. Available: https://nvdlib.com/en/latest/. [Accessed 17 August 2024]. |
| [24] | "Wappalyzer," [Online]. Available: https://www.wappalyzer.com/docs/api/v2/basics/. [Accessed 17 August 2024]. |
| [25] | "DVWA," [Online]. Available: https://www.kali.org/tools/dvwa/. [Accessed 1 August 2024]. |
| [26] | "BWapp - An extremely buggy web app!," [Online]. Available: http://itsecgames.com/. [Accessed 1 August 2024]. |
| [27] | "random - Generate pseudo-random numbers," [Online]. Available: https://docs.python.org/3/library/random.html. [Accessed 1 August 2024]. |
| [28] | "User-Agent," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent. [Accessed 1 August 2024]. |
| [29] | "datetime — Basic date and time types¶," [Online]. Available: https://docs.python.org/3/library/datetime.html. [Accessed 1 August 2024]. |
| [30] | "math — Mathematical functions," [Online]. Available: https://docs.python.org/3/library/math.html. [Accessed 1 August 2024]. |
| [31] | "colorama," [Online]. Available: https://pypi.org/project/colorama/. [Accessed 1 August 2024]. |
| [32] | "What Is JSON and How Do You Use It?," [Online]. Available: https://www.howtogeek.com/devops/what-is-json-and-how-do-you-use-it/. [Accessed 1 August 2024]. |
| [33] | TechTarget Contributor, "vulnerability (information technology)," [Online]. Available: https://www.techtarget.com/whatis/definition/vulnerability. [Accessed 03 August 2024]. |
| [34] | A. Ot, "What's the Difference Between Black-Hat and White-Hat Hackers?," [Online]. Available: https://www.makeuseof.com/what-are-black-white-hat-hackers/. [Accessed 03 August 2024]. |
| [35] | B. Lenaerts-Bergmans, "INJECTION ATTACKS," [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/cyberattacks/injection-attack/. [Accessed 03 August 2024]. |
| [36] | S. Gupta, "What Is SQL & How Does It Work? A Guide to Structured Query Language," [Online]. Available: https://www.springboard.com/blog/data-analytics/what-is-sql/. [Accessed 03 August 2024]. |
| [37] | KirstenS, "Cross Site Scripting (XSS)," [Online]. Available: https://owasp.org/www-community/attacks/xss/. [Accessed 03 August 2024]. |
| [38] | A. Dizdar, "Reflected XSS: Examples, Testing, and Prevention," [Online]. Available: https://brightsec.com/blog/reflected-xss/. [Accessed 03 August 2024]. |
| [39] | A. Dizdar, "Stored XSS: Impact, Examples, and Prevention," [Online]. Available: https://brightsec.com/blog/stored-xss/. [Accessed 03 August 2024]. |
| [40] | "HTML Injection," [Online]. Available: https://www.imperva.com/learn/application-security/html-injection/. [Accessed 03 August 2024]. |
| [41] | J. Kent, "What Are iframe," [Online]. Available: https://www.cequence.ai/blog/api-security/what-are-iframe-injection-attacks/. [Accessed 04 August 2024]. |
| [42] | "JavaScript Tutorial," [Online]. Available: https://www.geeksforgeeks.org/javascript/. [Accessed 04 August 2024]. |
| [43] | "Insecure Direct Object Reference Prevention Cheat Sheet," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Insecure\_Direct\_Object\_Reference\_Prevention\_Cheat\_Sheet.html. [Accessed 04 August 2024]. |
| [44] | Aptive, "Local File Inclusion (LFI) — Web Application Penetration Testing," [Online]. Available: https://medium.com/@Aptive/local-file-inclusion-lfi-web-application-penetration-testing-cc9dc8dd3601. [Accessed 04 August 2024]. |
| [45] | "What is PHP?," [Online]. Available: https://www.php.net/manual/en/intro-whatis.php. [Accessed 07 August 2024]. |
| [46] | M. Dahan, "SSI injection attacks and how to avoid them," [Online]. Available: https://www.comparitech.com/blog/information-security/ssi-injection-attacks/. [Accessed 07 August 2024]. |
| [47] | "Server-side request forgery (SSRF) attacks and how to prevent them," [Online]. Available: https://www.comparitech.com/blog/information-security/server-side-request-forgery-attacks/. [Accessed 07 August 2024]. |
| [48] | A. Jalli, "What Is XML?," [Online]. Available: https://builtin.com/data-science/xml. [Accessed 07 August 2024]. |
| [49] | K. Imtiaz, "Security Misconfiguration:," [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/security-misconfiguration/. [Accessed 07 August 2024]. |
| [50] | "Testing for Browser Cache Weaknesses," [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web\_Application\_Security\_Testing/04-Authentication\_Testing/06-Testing\_for\_Browser\_Cache\_Weaknesses. [Accessed 07 August 2024]. |
| [51] | "Cache Poisoning and Cache Deception," [Online]. Available: https://book.hacktricks.xyz/pentesting-web/cache-deception. [Accessed 07 August 2024]. |
| [52] | S. Hobbs, "CORS Tutorial: A Guide to Cross-Origin Resource Sharing," [Online]. Available: https://auth0.com/blog/cors-tutorial-a-guide-to-cross-origin-resource-sharing/. [Accessed 07 August 2024]. |
| [53] | "Cross-Origin Resource Sharing (CORS) Vulnerability: Example and Prevention," [Online]. Available: https://medium.com/@tushar\_rs\_/cross-origin-resource-sharing-cors-vulnerability-example-and-prevention-588d299ff185. [Accessed 07 August 2024]. |
| [54] | "What is HHI (Host Header Injection) & How to Simply Test the Vulnerability," [Online]. Available: https://medium.com/mii-cybersec/what-is-hhi-host-header-injection-how-to-simply-test-the-vulnerability-4f1f74a36f9d. [Accessed 7 August 2024]. |
| [55] | A. Heddings, "What Is HSTS and How Do You Set It Up?," [Online]. Available: https://www.howtogeek.com/devops/what-is-hsts-and-how-do-you-set-it-up/. [Accessed 7 August 2024]. |
| [56] | "Rich Internet Applications," [Online]. Available: https://www.sciencedirect.com/topics/computer-science/rich-internet-applications. [Accessed 7 August 2024]. |
| [57] | "Introduction to robots.txt," [Online]. Available: https://developers.google.com/search/docs/crawling-indexing/robots/intro. [Accessed 7 August 2024]. |
| [58] | "What are cookies on websites?," [Online]. Available: https://www.cloudflare.com/learning/privacy/what-are-cookies/. [Accessed 7 August 2024]. |
| [59] | "Cookie Security," [Online]. Available: https://jscrambler.com/learning-hub/cookie-security. [Accessed 7 August 2024]. |
| [60] | A. Sharif, "What Is CVE?," [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/common-vulnerabilities-and-exposures-cve/. [Accessed 7 August 2024]. |
| [61] | "What is a Framework in Programming and Engineering?," [Online]. Available: https://aws.amazon.com/what-is/framework/. [Accessed 7 August 2024]. |
| [62] | "What is an API (Application Programming Interface)," [Online]. Available: https://www.geeksforgeeks.org/what-is-an-api/. [Accessed 11 August 2024]. |
| [63] | "National Vulnerability Database," [Online]. Available: https://nvd.nist.gov/. [Accessed 11 August 2024]. |
| [64] | "What is a Payload in Cyber Security? A Comprehensive Guide," [Online]. Available: https://networkinterview.com/what-is-a-payload-in-cyber-security/. [Accessed 11 August 2024]. |
| [65] | "SQL Payload List," [Online]. Available: https://github.com/payloadbox/sql-injection-payload-list. [Accessed 23 May 2024]. |
| [66] | "SQL Payload List," [Online]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/SQLite%20Injection.md. [Accessed 25 May 2024]. |
| [67] | "HTML Injection Payloads," [Online]. Available: https://github.com/InfoSecWarrior/Offensive-Payloads/blob/main/Html-Injection-Payloads.txt. [Accessed 26 May 2024]. |
| [68] | "XSS Injection Payloads," [Online]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/XSS%20Injection. [Accessed 26 May 2024]. |
| [69] | "Other Payloads," [Online]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings. [Accessed 26 May 2024]. |
| [70] | "Robots Disallow," [Online]. Available: https://github.com/danielmiessler/RobotsDisallowed/blob/master/top1000.txt. [Accessed 1 August 2024]. |
| [71] | "URL Subdirectories," [Online]. Available: https://github.com/aels/subdirectories-discover/tree/main. [Accessed 1 August 2024]. |
| [72] | "Official Common Platform Enumaration (CPE) Dictionary," [Online]. Available: https://nvd.nist.gov/products/cpe. [Accessed 15 August 2024]. |
| [73] | T. Naiboiu, "WebScanner2," [Online]. Available: https://github.com/Ptmlol/WebScanner2. [Accessed 15 August 2024]. |
| [74] | "The 10 Best Vulnerability Scanners for Effective Web Security," [Online]. Available: https://wpscan.com/blog/vulnerability-scanners/. [Accessed 19 August 2024]. |