UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA MATEMATICĂ-INFORMATICĂ

LUCRARE DE LICENȚĂ

Web Application Vulnerability Scanner

COORDONATOR ȘTIINȚIFIC:

Conf. Dr. RUXANDRA OLIMID

ABSOLVENT:

NĂIBOIU TEODOR

BUCUREȘTI, IUNIE 2021

Cuprins

1. Introducere 7

1.1 Motivația 7

1.2 Contribuția personală 8

1.3 Structura 9

2. Tehnologii și Instrumente 10

2.1 Python 10

2.2 Requests 10

2.3 Urllib 11

2.3.1 Urllib.parse 11

2.3.2 Urllib.request 11

2.4 BeautifulSoup4 11

2.4.1 BeautifulSoup 11

2.5 SSL 12

2.6 Socket 12

2.7 Subprocess 13

2.8 Threading 13

2.9 Tkinter 13

2.10 OS 14

2.11 Re 14

2.12 ConfigParser 14

2.13 Metasploitable 2 15

2.13.1 DVWA 15

2.14 Alte tehnologii utilizate 16
3. Vulnerabilitățile aplicațiilor web 17
3.1 Injections 17
3.1.1 SQL Injections 18
3.1.2 NOSQL Injections 18
3.1.3 OS Command Injection 19
3.2 Broken Authentication 19
3.2.1 Credential stuffing 19
3.2.2 Session attacks 21
3.3 Sensitive Data Exposure 22
3.4 XML External Entities (XXE) 22
3.5 Broken Access Control 23
3.6 Security Misconfiguration 23
3.7 Cross-Site Scripting (XSS) 24
3.7.1 Reflected XSS 24
3.7.2 Stored XSS 25
3.7.3 DOM XSS 25
4. Arhitectura aplicației și implementarea testelor 26
4.1 Parsarea variabilelor din fișierul de configurare 26
4.2 Clasa LoginTests 27
4.3 Clasa OtherUser 30
4.4 Clasa GUI 32
4.5 Clasa Scanner 35
4.5.1 Extragerea informațiilor 37
4.5.2 Aplicarea testelor asupra informațiilor 41

4.6 Generarea raportului și interacțiunea cu GUI 52
4.6.1 Generarea raportului 52
4.6.2 Interacțiunea cu GUI 55
4.7 Descărcare și instalare 55
4.7.1 Windows 55
4.7.2 Linux 56
4.8 Modul de utilizare 56
4.8.1 Windows 56
4.8.2 Linux 57
5. Testarea și verificarea aplicației 58
5.1 Testarea GUI 58
5.2 Testarea algoritmului de detecție 61
6. Concluzii și îmbunătățiri 63
Bibliografie 64

Rezumat

Proiectul care urmează să fie prezentat în această lucrare este o aplicație software de tipul *scanner* destinată dezvoltatorilor de site-uri web și persoanelor care se ocupă cu testarea și asigurarea securității acestora. *Scannerul* are ca scop indentificarea și raportarea breșelor de securitate găsite în procesul de testare automată numită și *scanare*. Utilizatorii aplicației pot interacționa diferit cu aceasta în funcție de sistemul de operare pe care aceștia aleg să execute scanarea. Pe sistemele Windows aplicația dispune de un *graphical user interface* (GUI) cu care utilizatorul poate interacționa, toate informațiile despre folosirea *scannerului* fiind puse la dispoziție de către interfață. Pe sistemele Linux aplicația trebuie folosită prin intermediul unui terminal, toată interacțiunea fiind realizată exclusiv prin linia de comandă. Odată ce utilizatorul introduce datele aplicației pe care acesta dorește să o supună testării, tot ce rămâne de făcut este să pornească scanarea și să aștepte raportul generat. La final aplicația printează informații cu privire la raport și ghidează utilizatorul către fișierele generate pentru vizualizarea rezultatelor detaliate.

Din cauza diversității metodelor de implementare ale unei aplicații web, acestea sunt predispuse la mici erori de proiectare. Aceste erori cauzează, în majoritatea cazurilor, apariția vulnerabilităților în sistemele care manageriază aceste aplicații. Deci, este nevoie de astfel de unelte automatizate care să ajute la depistarea și rezolvarea acestor erori într-un interval cât mai scurt de timp, astfel acestea ajung în producție mai sigure și scutesc dezvoltatorii din a petrece timp îndelungat în creșterea securității lor manual.

Abstract

The following project is a *scanner-like* software application aimed for developers or security testers that want to evaluate the security of a *web* application. The *scanner's* purpose is to detect and report vulnerabilities found in the automated scanning process. Users can interact with the application in two ways, depending on their operation system. On Windows systems users can interact with the application through the *graphical user interface* (GUI) that provides useful information about the usage of the *scanner*. On Linux systems interaction must be made through *command line interface* (CLI). Once the user provides the *scanner* with all the necessary inputs needed for testing, all that is left to do is to start the process and wait for the raport. When finished, the *scanner* outputs information about the raport and guides the user to the detailed results.

Due to the many ways of structuring a *web* application, there is a high risk for small implementing errors. Those errors are often the main cause of the vulnerability flaws that appear in the host systems. So there is a need for those kinds of automated *scanners* to help with the detection and correction of those errors in a small amount of time. This way, the *web* application can be released in a safer enviroment and the developers can focus on important things, other than spending time manually testing.

1. Introducere

Securitatea este, simplu spus, libertatea față de risc sau pericol [1]. Fiind una dintre nevoile fundamentale umane, oamenii și-au dedicat o lungă perioadă din viață în căutarea a noi metode inovative și eficiente pentru a combate pericolele născute fie din natură, cum ar fi eventimentele meteorologice, animalele, virusurile și bacteriile, fie născocite chiar de aproapele lui. Astfel, cu timpul, am fost nevoiți să ne adăpostim în locuințe confecționate în asa fel încât acestea să asigure protecție față de orice pericol provenit din exteriorul acestora. Pe măsură ce evoluăm ca specie, această nevoie de protecție crește proporțional cu riscurile cauzate de avansul, atât tehnologic cât și evoluționar al omului.

Foarte asemănătoare nouă este și evoluția securității informației. Am observat cum imediat după apariția internetului a fost nevoie de metode nemaiîntâlnite de a ne proteja informațiile și datele personale, păstrate până în acel moment în seifuri sau în locații bine păzite. Apariția internetului ne-a deschis, deci, porțiile către o nouă lume mai conectată și cu o mulțime de noi posibilități dar și noi pericole, noi riscuri asociate acestei invenții.

Astăzi securitatea informației, sau securitatea cibernetică, joacă un rol extrem de important în buna funcționare a societății. Ea se definește ca diverse procese și practici ce au rolul de a proteja rețele, *device-uri*, programe și date în fața atacurilor, distrugerilor sau a accesului neautorizat [2].

1.1 Motivația

Recent am fost martorii unui atac cibernetic de tipul *Ransomware* care a avut ca țintă “Colonial Pipeline”, o rețea de distribuție de combustibil aflată în partea de Est a Statelor Unite. Atacul a cauzat o scădere a aprovizionării stațiilor de alimentare cu combustibil, având astfel un impact ridicat asupra economiei zonei afectate, producând daune de milioane de dolari și instaurând o panică generală în rândul cetățenilor [3]. Observăm astfel impactul dezastruos pe care o breșă de securitate îl poate avea atât pe plan politic și economic cât și psihologic.

Am văzut mai sus un singur tip de atac cibernetic, în realitate există o mulțime de astfel de atacuri de diferite forme și cu diferite intenții. Unele dintre ele sunt cauzate de conflicte politice, economice, iar altele din simplul motiv de a demonstra abilitățiile atacatorilor. O bună parte dintre

aceste atacuri țintește aplicațiile Web, cunoscute și sub numele de *site-uri web*. O aplicație web este un program care utilizează un *browser web* și metode *web* pentru a îndeplini diverse atribuții pe internet [4]. Din cauza frecventelor interacțiuni umane cu aceste aplicații, riscul ca ele să devină ținta atacurilor cibernetice este unul foarte ridicat, astfel specialiștii în securitate cibernetică au investit timp și efort sporit în scopul securizării lor.

Securizarea aplicațiilor *web* este, pe zi ce trece, din ce în ce mai importantă. În fiecare zi zeci de milioane de persoane își încredințează atât datele persoanale cât și confidențialitatea acestor aplicații, sarcina de a asigura această confidențialitate fiind pe umerii dezvoltatoriilor. Totuși pe zi ce trece se descoperă noi vulnerabilități și noi atacuri din care un număr ridicat de oameni are de suferit. Astfel, am început să fiu atras de modalitățile prin care aceste vulnerabilități funcționează. Odată ce am înteles conceptele, am decis că este nevoie să le pun în practică, astfel am creat acest *scanner* pentru a-mi fixa cunoștințele și pentru a interacționa indirect cu dezvoltatorii aflați în procesul de implementare al *aplicațiilor web*.

1.2 Contribuția personală

Interesul pentru această temă este datorat curiozității, cât și pasiunii mele pentru securitatea cibernetică, în special pentru securitatea *aplicațiilor web.* Astfel am decis să creez un program care să ușureze procesul de securizare al aplicației și să minimizeze timpul petrecut practicând testele manual și erorile umane apărute pe parcursul acestora.

Acum că am definit conceptul de *aplicație web* este timpul să ne familiarizăm cu metodele prin care o astfel de aplicație este supusă testelor de siguranță înainte ca aceasta să fie disponibilă publicului. În mod normal, pentru a evita un atac, testerul se asigură că aplicația îndeplinește niște norme de securitate standard definite de OWASP [5].

Pe parcursul acestui proiect, pe langă implementarea aplicației *scanner* și a anumitor teste de detectare a vulnerabilităților, am adăugat contribuții personale și în studiul tipurilor de vulnerabilități. Acolo se găsesc comentarii și exemple provenite din cunoștințele și experiența mea dobândită în procesul de explorare al domeniului securității.

1.3 Structura

În următorul capitol am să prezint instrumentele folosite și conceptele utilizate în decursul implementării aplicației. Apoi, în Capitolul 3, am să vorbesc despre vulnerabilitățile și conceptele testate de *scanner,* urmând ca în Capitolul 4 să vorbesc despre implementare și modul de utilizare. Capitolul 5 este destinat verificării funcționalității *scannerului,* iar ultimul capitol este pentru concluzii și eventualele îmbunătățiri viitoare.

Consider că realizarea acestei lucrări a fost o provocare din care am reușit să-mi aprofundez cunoștințele despre domeniul securității și cred că rezultatul poate fi folosit de oricine dorește să dezvolte sau să testeze o *aplicație web.*

2. Tehnologii și Instrumente

Proiectul este realizat utilizând limbajul de programare Python [6]. În cadrul acestui proiect am folosit diverse module. Un modul este un fișier Pyhon cu diverse definiții și instrucțiuni. Printre cele mai importante module folosite se numără modulul *requests* [7], folosit pentru a comunica cu *aplicația web*, modulul *urllib* [8] folosit pentru manipularea adreselor *web*, modulul *BeautifulSoup4* [9] pentru extragerea informațiilor, și alte module utile despre care urmează să vorbim în subcapitolele următoare.

2.1 Python

Pyhon este un limbaj de programare util pentru aproape orice aplicație. Este un limbaj interpretat, orientat pe obiect cu o colecție vastă de module și o sintaxa atragătoare. Datorită asemănării lui cu limbajul *bash* și capacității sale de a executa o multitudine de operații des întâlnite în securitatea cibernetica, cât și a colecției impresionante de module special create pentru *Penetration Testing*, Pyhon a fost mereu limbajul preferat de inginerii de securitate și testeri [6].

Poate cel mai important avantaj al limbajului Pyhon în acest domeniu îl reprezintă rapiditatea cu care pot fi rezolvate problemele, implementarea codului și rezolvarea erorilor fiind mult mai rapidă în comparație cu alte limbaje de programare [6].

2.2 Requests

Requests este un modul Python care permite utilizatorului să interacționeze cu *request-urile* de tip *Hypertext Transfer Protocol* (HTTP) într-un mod simplu și rapid. Un *request* este o acțiune realizată asupra unei resurse aflate la o anumită adresa *web,* cu alte cuvinte, este o cale intermediară care transportă informații între client (sau *browser*) și server. Tipurile de *request-uri* variază în funcție de scopul acestora. În total există un număr de nouă tipuri de astfel de *request-uri* dar cele mai importante și cele pe care le-am folosit în dezvoltarea acestei lucrări sunt: GET, POST, PUT, DELETE [7].

GET este folosită pentru a obține date de la server.

DELETE șterge o anumită resursă specificată în *request*.

PUT este metoda folosită pentru a modifica reprezentăriile unei stări existente, cu o reprezentare nouă, definită în procesul de *request*.

POST este metoda specifică pentru a trimite date noi către server. Poate fi folosită pentru a actualiza și a crea noi date pe server.

2.3 Urllib

Urllib [8] este un modul care are în componența lui două module importante pe care le-am folosit.

2.3.1 Urllib.parse

Modulul urllib.parse este folosit pentru manipularea și modificarea *Uniform Resource Locator* (URL). Cu ajutorul acestui modul, putem fragmenta, combina și extrage fragmente din URL-uri [10].

2.3.2 Urllib.request

Modulul urllib.request definește instrucțiuni de cod utile pentru accesarea și interacțiunea cu metodele HTTP. Modulul ne furnizează un mod de comunicare puțin mai avansat decât modulul *requests* [11].

2.4 BeautifulSoup4

Din componența modulului BeautifulSoup4 [9] am folosit modulul cu același nume, adică BeautifulSoup.

2.4.1 BeautifulSoup

Modulul BeautifulSoup este cel mai folosit modul în Python pentru preluarea informațiilor din paginile *web*. În principal am folosit acest modul drept metodă de

extragere a diverselor părți componente din codul sursă al paginii *web*. Este eficient în preluarea datelor de tipul HTTP și *Extensible Markup Language* (XML) [9].

2.5 SSL

Modulul SSL (*Secure Sockets Layer)* este un modul care permite accesul la TLS (*Transport Layer Security)*. Pentru ca datele noastre *online* să fie securizate este nevoie de protocoale criptografice care să transforme aceste date în secvențe de litere și cifre aleatoare și indescifrabile computațional. Protocolul SSL folosește un sistem de criptare cu două chei, o cheie publică și o cheie privată. Pentru decriptarea mesajului este nevoie de ambele chei, astfel doar persoanele autorizate pot să aibă acces la datele criptate în mesajul transmis [12].

Acest modul se foloseste pentru interacțiunea cu certificatul de securitate al aplicației *web*, cunoscut și sub numele de *SSL Certificate*. Un astfel de certificat asigură utilizatorii că transferul datelor dintre *browserul* lor și aplicația *web* se realizează printr-un canal securizat, astfel datele fiind protejate [12].

Folosit împreuna cu modulul *socket* acest modul poate obține informații detaliate despre certificatele SSL.

2.6 Socket

Modulul socket definește o gamă largă de procedee care au ca scop conectarea a două noduri de rețea pentru a favoriza comunicarea între ele.
Procesul prin care două noduri de rețea realizează o conexiune este usor explicat daca ne imaginăm că un nod așteaptă un răspuns de la un nod care încearcă să comunice cu acesta. Acest proces simulează o conexiune între un client și un server [13].

Un socket definește un *endpoint* într-o rețea, astfel, socketul crează un canal bidirecțional de comunicare folosindu-se de adresa *Internet Protocol* (IP) și de un port deschis pe aplicație. Un *endpoint* este un *device* conectat la o rețea [13].

2.7 Subprocess

Un proces este definit ca un program care se află în execuție. Astfel de procese sunt controlate de *Central Processing Unit* (CPU) și reprezintă o mulțime de calcule efectuate de procesor (CPU) într-un interval foarte scurt de timp. Aceste procese se află într-o continuă execuție secvențială, astfel, un program execută mereu un singur proces [14].

Modulul Subprocess este folosit pentru a crea, a distruge sau a interacționa cu aceste procese. Așadar utilizăm acest modul pentru a interacționa direct cu diverse programe, prin intermediul unei alte aplicații, procesul nou creat fiind integrat în *firul* principal de execuție [14].

Definim un *fir* de execuție drept un set de astfel de procese care sunt executate pe rând și într-o ordine bine definită și determinată de CPU.

2.8 Threading

Un *thread* este o modalitate prin care putem executa mai multe procese în același timp. Putem să asemănăm *Thread-urile* cu mai multe programe care rulează concomitent și independent unele față de altele. În aplicațiile în care avem nevoie ca mai multe instrucțiuni să fie executate în același timp și în care există riscul ca execuția unei bucăți de cod să blocheze execuția alteia, creăm astfel de *fire de execuție* care fragmenează instrucțiunile programului și comunică cu CPU pentru a aloca memorie separată pentru *firul de execuție* folosit [15].

2.9 Tkinter

Modulul Tkinter este o suită populară de unelte folosite pentru a crea interfețe grafice. Tkinter a fost construit cu scopul de a oferi simplitate și diversitate în alcătuirea unei interfațe grafice cât mai atrăgătoare și intuitive [16].

Structura Tkinter este realizată dintr-o fereastră simplă, dar ușor customizabilă prin metode implicite, astfel adăugarea unui buton, unei liste de conținut sau mesajelor de ajutor poate fi realizată în doar câteva linii de cod. Asupra ferestrei simple, se aplică diverse modificări și funcționlități care, utilizând modului *threading,* rulează în paralel cu execuția din spatele interfeței grafice, oferind astfel o interacțiune placută între utilizator și aplicație.

Pe lângă modului Tkinter, sunt des întâlnite și modulele PyQt5, Kivy, wxPython, PyFrosm, utilitatea lor fiind dată de scopul aplicației.

2.10 OS

Prin intermediul modulului OS putem interacționa eficient cu sistemul de operare pe care rulează aplicația. Acest modul este indispensabil în interacțiunea cu fișiere, căi sau pentru diversele modificări pe care putem să le aducem acestora. Avantajul folosirii acestui modul este dat de transparența față de sistemul de operare. Sintaxa modulului comunică eficient cu sistemul de operare indiferent de sistemul pe care acesta rulează. Spre exemplu sintaxa *os.stat(path)* returnează informații despre calea dată ca argument indiferent dacă aceasta este o cale a sistemului *Windows* sau a sistemului *Unix* [17].

2.11 Re

Numele modulului Re este un acronim pentru *Regular expression*. Re este un set de caractere speciale (*patttern-uri)* folosit pentru căuta și a potrivi diverse șiruri de caractere care îndeplinesc regula definită în expresia specială (*regular expression)* [18].

Un exemplu de o astfel de regulă este " \d.* " care, aplicată pe șirul "Ana are 5 mere", găsește subșirul "5 mere". În expresia *Re* "\d.*" înseamnă orice cifră (\d) urmată de oricâte (*) caractere (.). Acest modul este util în extragerea informațiilor de orice formă din orice șir de caractere.

2.12 ConfigParser

Este o practică standard ca majoritatea aplicațiilor să folosească un fișier de configurare pentru a oferi flexibilitate și libertate utilizatorului. Pentru a exista comunicare între un astfel de fișier și codul aplicației, este nevoie de o metodă prin care limbajul de programare să comunice cu limbajul din fișierul de configurare [19].

Modulul ConfigParser oferă posibilitatea de a accesa astfel de fișiere de forma *nume_fișier.ini* și de a traduce datele astfel încât limbajul aplicației să le înțeleagă. În felul acesta

datele variabile sunt ușor de configurat de oricine, crescând potențialul aplicației pentru o gamă mai largă de utilizări [19].

2.13 Metasploitable 2

Metasploitable 2 este o versiune de Ubuntu Linux, alcătuită special din cod nesecurizat, fiind astfel o colecție de servicii vulnerabile din punctul de vedere al securității. Principalul scop al acestui sistem este de a pune la dispoziție un mediu sigur, închis, care să faciliteze practicile comune de testare, de învățare și de practicare a diverselor metode sau ustensile folosite în procesul învățării conceptelor de securitate cibernetică [20].

Sistemul conține nenumărate vulnerabilități adaugate intenționat sau neintenționat pentru a putea susține aproape orice tip de abordare și pentru a oferi totodată un mediu legal în care profesioniștii dar și persoanele interesate pot să exerseze și să capete mai multe cunoștiințe.

Datorită faptului că aplicația are la baza un sistem Linux, regăsim pe aceasta un server *web,* care susține o mică suită de aplicații web vulnerabile, fiecare cu numele ei reprezentativ. Printre aplicațiile vulnerabile se află aplicația *Damn Vulnerable Web App* (DVWA) care conține o colecție variată de vulnerabilități comune și des întalnite pe *site-urile web* [21].

2.13.1DVWA

DVWA este aplicația preferată pentru testarea conceptelor securității *web* datorită numărului ridicat de vulnerabilități existente pe aceasta și interfeței orientative. Aplicația este folosită nu doar de profesioniști, ci și de studenți, profesori, persoane interesate de securizarea aplicațiilor web sau dezvoltatori care caută un loc în care să-și testeze aplicațiile software fără a le face publice sau a încălca legea [21].

Am ales această aplicație atât datorită numărului ridicat de vulnerabilități cât și tehnologiilor pe care le pune la dispoziție, reușind să mimeze foarte bine aplicațiile *web* des întâlnite zi de zi. Prezența unui număr ridicat de vulnerabilități oferă o arie mai mare de acoperire pentru testele care urmează să fie făcute pe situații reale, atunci când atât tipul cât și numărul de vulnerabilității prezente într-o aplicație o să difere.

2.14 Alte tehnologii utilizate

Alte module care merită menționate și care au ajutat la implementarea acestui proiect sunt module auxiliare, cu scop utilitar, care fie eficientizează programul din punctul de vedere al complexității, fie îndeplinesc mici atribuții.

Modulul *queue* care este folositor pentru implementarea unei cozi de diverse evenimente. În cazul acestui proiect, modul *queue* a eficientizat procesul prin care se testa validitatea unui șir de caractere, astfel, mai multe șiruri de caractere citite dintr-un fișier au fost încărcate și reținute în memorie prin intermediul unei astfel de cozi [22].

Modulul *datetime* ne oferă acces la datele calendaristice și timp. Cu ajutorul acestui modul am manipulat date de forma *ore:minute:secunde* și am reușit să aplic operații de scădere sau adunare a unor perioade de timp asupra datelor de forma specificată anterior [23].

Modulul *nmap* este o implementare a uneltei folosite pentru analizarea rețelelor care poartă același nume. *Network Mapper* (nmap) este folosit în special pentru maparea arhitecturii unei rețele și pentru detalierea serviciilor și *host-urilor* care comunică pe aceasta. Un *host* reprezintă un calculator conectat la o rețea [24]. Am folosit *nmap* pentru analizarea adresei IP a aplicației *web*.

Modulul *ctypes* este un modul care ne oferă tipuri de date compatibile cu limbajul C. Acest modul este folositor atunci când vrem să oprim execuția unui thread [25].

Modulul *shutil* oferă acces la operații de tip *high-level* pe fișiere. Aceste operații sunt ștergerea și copierea. Acest modul a fost implementat pentru a șterge fișierele temporare, create în timpul execuției [26].

Modulul *webbrowser* este un modul care interacționează și controlează *browserul* implicit al sistemului. Cu ajutorul acestui modul am implementat metode prin care utilizatorul poate vedea explicații detaliate disponibile pe internet cu privire la vulnerabilitățile descoperite și testate de *scanner* [27].

3. Vulnerabilitățile aplicațiilor web

Înainte să vorbesc despre implementarea și arhitectura aplicației este nevoie să fixăm câteva noțiuni importante pe care le-am pus în practică în dezvoltarea acesteia. În acest capitol am să explic ce este o vulnerabilitate web, de câte tipuri sunt ele și cum afectează buna funcționare a unui *site web*, urmând ca în capitolul următor să explic detaliat abordarea pentru detecția acestora.

O vulnerabilitate web este o slăbiciune sau o configurare greșită a anumitor funcționalități pe care o *aplicație web* le îndeplinește. Uneori aceste vulnerabilități sunt abuzate de persoane rău intenționate care reușesc să preia controlul asupra aplicației pentru a obține date confidențiale în diverse scopuri personale. Numim aceste persoane *hackeri* sau atacatori. În funcție de modul în atacatorii reușesc să obțina acces la resursele securizate ale *aplicației web*, se disting diferite tipuri de astfel de vulnerabilități.

OWASP Top Ten este documentul oficial în care sunt consemnate tipurile de riscuri la care *aplicațiile web* sunt supuse și tacticile de detecție și neutralizare ale acestora. Atât dezvoltatorii cât și *penetration testerii* urmăresc îndrumările și tehnicile însemnate în document pentru a se asigura că arhitectura *aplicațiilor web* îndeplinește ultimele standarde de securitate. Acest proiect a fost realizat conform abordărilor și metodelor menționate și documentate în lista *Web Security Testing Guide* (WSTG) din cadrul proiectului OWASP [28].

În subcapitolele care urmează am să menționez trăsăturile fiecărui tip de vulnerabilități testate și integrate în acest proiect. Pentru fiecare categorie au fost implementate diferite abordări de testare care urmează să fie detaliate în capitolul urmator.

3.1 Injections

Tipul *Injections* reprezintă vulnerabilitățile care au la bază nesanitizarea *input-urilor* din aplicație. Acestea apar atunci cand un șir de instrucțiuni provenit din exterior este interpretat ca parte din codul aplicație, astfel instrucțiunea în cauză execută acțiuni nedorite cu potențial dăunator și intrusiv. Putem găsi acest tip de vulnerabilitate în orice câmp care primește date de la utilizator [29].

Cele mai populare tipuri de *injection* sunt *SQL Injections, NOSQL Injections, OS Command Injection.*

Figura 3.1. Exemplu SQL Injection

3.1.1 SQL Injections

Structured Query Language (SQL) este un limbaj de programare specific bazelor de date [30]. Astfel un *SQL Injection* este o vulnerabilitate care încearcă să manipuleze înregistrările unei baze de date cu scopul de a accesa înregistrările (*query-urile)* din componența acesteia. O astfel de practică presupune introducerea instrucțiunilor de tip SQL în *input-urile* aplicației care sunt legate la o baza de date SQL, și accesarea înregistrărilor, șirul introdus fiind astfel interpretat ca parte din limbajul SQL, permițând accesul la înregistrări la care utilizatorul nu este autorizat [31].

Observăm în Figura 3.1 un exemplu de un astfel de *injection*. Un atacator introduce șirul de caractere “parola or ‘1’=’1’;--" care, interpretat de limbajul SQL, este o verificare în care prima parte a șirului poate fi ignorată, deoarece valoarea de adevăr a instrucțiunii ‘1’=’1’ este adevărată. La finalul șirului atacatorul introduce ‘;--‘ pentru ca *query-ul* să ignore orice se afla dupa verificarea precedentă. În felul acesta, baza de date returnează toate parolele stocate în tabela accesată atacatorului.

Pentru a preveni această vulnerabilitate soluția este găsirea unei metode care păstrează datele introduse, separat de comenzile și *query-urile* din baza de date.

3.1.2 NOSQL Injections

NOSQL Injections sunt baze de date de tipul SQL cu mai puține restricții și constrângeri. Această tehnologie este mai eficientă computațional față de SQL [32].

Injectionul de tip NOSQL funcționează pe același principiu ca și în cazul *SQL Injection*.

3.1.3 OS Command Injection

Această vulnerabiliate este abuzată prin introducerea de comenzi care pot fi interpretate de sistemul de operare pe *serverul* pe care rulează *aplicația web* prin intermediul interfeței acesteia. Orice *input* din aplicație care nu este sanitizat cum trebuie, poate executa comenzi pe sistemul de operare [33].

Pentru a preveni această vulnerabilitate este nevoie ca *aplicația web* să ruleze sub niște restricții stricte care să nu permită rularea comenzilor pe *serverul* aplicației.

3.2 Broken Authentication

Acest tip definește vulnerabilitățile aparute în urma configurării greșite a funcțiilor de autentificare și gestionare a unei sesiuni. O sesiune este o perioadă de timp definită de dezvoltatorul aplicației în care anumite date și preferințe ale utilizatorului sunt reținune pe parcursul navigării pe *aplicația web.* O funcționalitate este că permite navigarea între paginile unui *site* fără a fi nevoie ca utilizatorul să se autentifice de fiecare dată [34].

Acestă vulnerabilitate este foarte răspândită în lumea *aplicațiilor web* și foarte complexă. Este foarte ușor pentru atacatori să se folosească de liste de parole și nume de utlizatori, folosind unelte automate, pentru a ghici datele de autentificare ale altor utilizatori. Este nevoie doar de un singur cont cu privilegii administrative pentru a compromite o intreagă aplicație [34].

Așa cum am menționat mai sus, există două tipuri importante de astfel de atacuri. Atacuri care se folosesc de vulnerabilități în implementarea funcțiilor care gestionează sesiunile, *session management*, și atacuri care vizeză datele de autentificare, cunoscute sub numele de *credential stuffing*.

3.2.1 Credential stuffing

Un atac de tipul *credential stuffing* constă în încercarea de a introduce date valide în câmpul de autentificare cu speranța că atacatorul nimerește, din întâmplare, un cont valid. Atacul se realizează cu ajutorul uneltelor automatizate care dispun de liste mari ce conțin date de autentificare obținute din scurgerile de date confidențiale din trecut [35].

Aceste liste, numite *dicționare,* conțin și parole comune, de obicei implicite pe baza că dezvoltatorii si administratorii unei aplicații uită să modifice o astfel de parolă.

Această practică este des folosită sub forma unui atac provenit din mai multe zone geografice. Un atacator se află în controlul mai multor calculatoare aflate în diferite zone, acesta lansează un atac simulan de pe toate aceste calculatoare cu scopul de a păcăli sistemele de siguranță. Acest atac este cunoscut sub numele de *Botnet*.

Acest tip de atac poate fi prevenit în multe feluri. Câteva dintre ele presupun ca dezvoltatorul să se asigure că nu a lăsat nici o parola implicită în componența aplicației, că a implementat un sistem care să nu-i permită unui utilizator să încerce să forțeze o autentificare într-un interval scurt de timp și că a implementat o condiție la înregistrarea unui cont ca parola setată să aibă o complexitate ridicată.

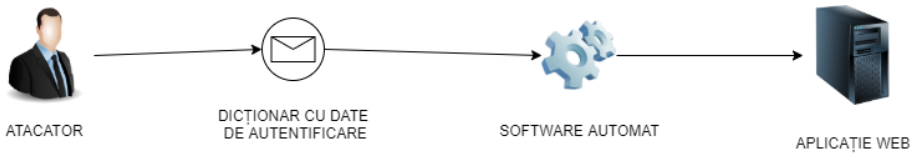


Figura 3.2. Exemplu credential stuffing

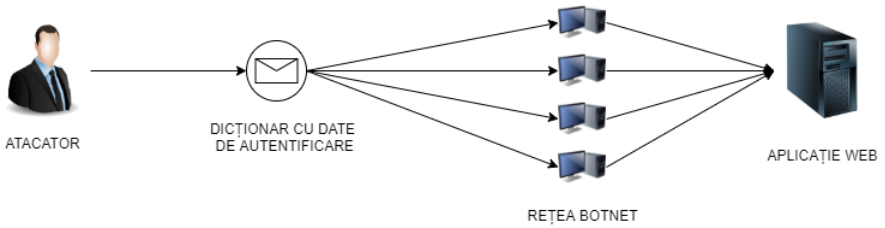


Figura 3.3. Exemplu atac BOTNET

Observăm în Figurile 3.2 si 3.3 diferența uriașă pe care o are impactul unui atac simplu, local, față de un atac care folosește o rețea Botnet. Șansele ca atacul să aibă succes sunt semnificativ mai mari în cazul Figurii 3.3.

3.2.2 Session attacks

Un alt tip de atac care intră în componența *Broken Authentication* este reprezentat de atacurile care vizează impersonarea unui alt utilizator. Acest lucru se întâmplă atunci când un atacator fură *IDul* sesiunii, prin diverse metode, unui utilizator autentificat (vezi Figura 3.4). Acest lucru este realizabil din cauza unei configurari greșite a sistemului care gestionează sesiunile. Este posibil ca *IDul* să nu fie transportat sau stocat într-un mod sigur, astfel un atacator poate să copieze pur și simplu *IDul*. Alte practici care trebuie evitate sunt autentificarea fără un *two factor login,* afișarea *IDului* în URL, păstrarea aceleași sesiuni pe toată durata procesului de autentificare, omiterea invalidării sesiunii atunci când un utilizator se deloghează sau este inactiv o perioadă mai lungă de timp [36].

Toate aceste obiceiuri duc la o aplicație nesigură și cu risc ridicat de compromitere. Pentru a evita astfel de evenimente, dezvoltatorul trebuie să se asigure că folosește un sistem de gestionare al sesiunii care rulează pe server și care este destul de bine configurat pentru a genera și a stoca *IDul* în siguranță.

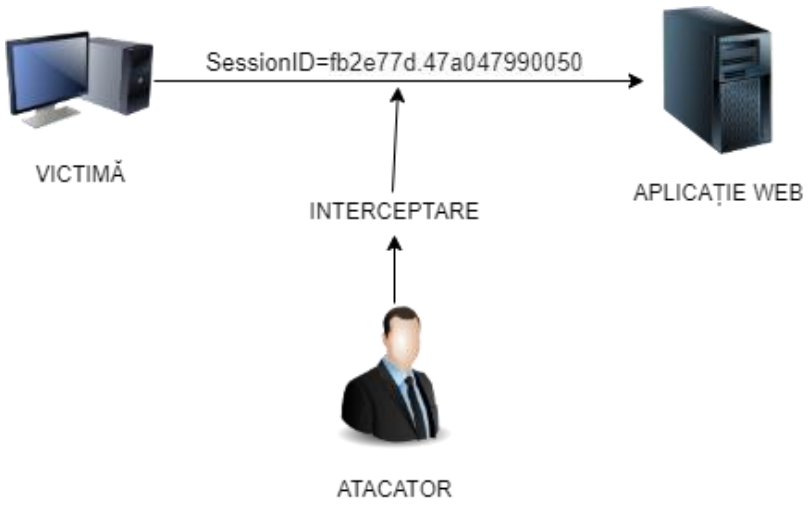


Figura 3.4. Un atacator interceptează o sesiune

3.3 Sensitive Data Exposure

Așa cum reiese și din titlu, acest tip de vulnerabilitate reprezintă expunerea, într-o modalitate care să permită citirea, datelor sensibile, confidențiale, publicului. Atacurile practicate implică interceptarea datelor aflate în tranzit, între victimă și *aplicația web*. Datele personale cum ar fi parolele, secretele de serviciu, mesajele confidențiale, adresele, numerele de telefon, datele bancare, au nevoie de cea mai mare protecție. O metodă practicată în desfășurarea unui astfel de atac este *man-in-the-middle* (MitM), metodă ce implică interpunerea în mod activ între entitățile de comunicare. Dezvoltatorul aplicației trebuie să se asigure că aplicația folosește protocoale criptografice actualizate și că este configurată în asa fel încât datele să fie transmise doar prin protocolul HTTPS. HTTPS este protocolul HTTP la care se adaugă stratul TLS/SSL cu rol de criptare, astfel toate datele aflate în tranzit între *browser* și server sunt transportate printr-un tunel criptat [37].

Alte vulnerabilități care intră în această categorie sunt date de interacțiunea dezvoltatorului cu *aplicația*. Acesta trebuie să se asigure că pe parcursul dezvoltării nu au fost lăsate, din greșeală mesaje sensibile la vedere, în comentarii sau undeva în pagina web. Mai mult, aplicația trebuie să conțină un certificat SSL valid, emis de un *Certificate Authority* (CA) de încredere și autorizat pentru a asigura utilizatorul că datele sunt criptate de o sursa de încredere.

Un atac frecvent întâlnit în astfel de cazuri este realizat de un atacator care se preface a fi un CA de încredere, care emite un certificat fals, care este apoi folosit de o *aplicație web* cu scopul de a păcăli utilizatorii pentru ca aceștia să-și introducă datele personale fără să suspecteze nici o tentativă de fraudă [38].

O astfel de vulnerabilitate poate fi evitată dacă dezvoltatorul evită să stocheze inutil date confidențiale, clasifică datele în funcție de confidențialiatea lor, respectând termenii de confidențialitate legali, se asigură că toate datele sensibile sunt criptate și foloseste parole puternice pentru securizarea punctelor de acces administrativ.

3.4 XML External Entities (XXE)

Foarte asemănător cu tipul *Injections*, vulnerabilitățile de tip XEE au la bază exploatarea modului în care este interpretat un *input* XML. Dacă aplicația acceptă *input* XML în mod direct,

în special de la surse care nu sunt de încredere sau dacă aplicația folosește *document type definitions* (DTD) există riscul ca aplicația să fie vulnerabilă. Vulnerabilitatea poate fi abuzată pentru a extrage date, execută diverse comenzi pe server sau pentru a-l supraîncărca, atac cunoscut sub numele de *Denial-of-Service* (DoS). Asemănător cu BOTNET, un astfel de atac realizat cu ajutorul mai multor calculatoare se numește *Distributed Denial-of-Service* (DDoS) [39].

Fiind asemănătoare cu *Injections*, vulnerabilitatea poate fi evitată dacă *input-urile* sunt sanitizate, tehnologiile de interpretare a XML sunt actualizate și dacă se folosesc structuri de date mai puțin complexe de forma *JavaScript Object Notation* (JSON). JSON este un format de date ușor de interpretat de către aplicații și ușor de citit de către oameni [40].

3.5 Broken Access Control

Funcția de *Access Control* gestionează privilegiile pe care utilizatorii unei aplicații le au. Cu alte cuvinte, funcția controlează ce date sunt accesibile și cui, la un moment dat. În cazul în care funcția nu este bine implementată, există riscul ca un simplu utilizator să aibă acces la resurse care, în mod normal, ar trebui să fie accesabile doar de un administrator. Atacatorii pot abuza de aceste scăpări și pot prelua controlul aplicației cu un minim efort [41].

Obținerea unui astfel de acces poate fi realizată în multe feluri și pentru scopuri diferite. Un atacator poate încerca să obțină acces de citire sau modificare a unor date de pe server, poate obține acces administrativ, sau poate să evite verificările privilegiilor în cazul în care dorește să acceseze o resursă restricționtă [41].

Acestă vulnerabiliate poate fi prevenită dacă dezvoltatorul implementeză metodele de control pe server. Acest lucru rezolvă situația în care un atacator poate să modifice diverse instrucțiuni din metode pentru a-și crea avantaje.

3.6 Security Misconfiguration

Configurările greșite sunt unele dintre cele mai răspândite vulnerabilități de pe o *aplicație web.* Acestea pot apărea în orice strat al aplicației, fie că este vorba de rețeaua pe care rulează serverul, fie de *browserul* de pe care aceasta este accesată. Vulnerabilitatea apare atunci când există prea multe funcționalități inutile pe aplicație, setările implicite nu sunt modificate, când nu se

actualizează sistemele de securitate (Antivirus, Firewall), când există erori care furnizează mesaje sensibile pe care atacatorii le pot abuza pentru a obține informații sensibile sau când se folosesc funcționalități despre care se cunosc deja vulnerabilități [42].

Această vulnerabilitate crește riscul ca o aplicație să fie predispusă unui atac mai mult decât orice altă vulnerabilitate. Din cauza diversității de locuri în care această vulnerabilitate apare, este dificil pentru dezvoltatori și testeri să combată eficient orice greșeală. Deseori acesta este punctul de start în realizarea unui atac, deci este vital ca toate configurările de securitate să fie implementate corect pentru a descuraja orice tentativă la un astfel de atac [42].

3.7 Cross-Site Scripting (XSS)

Cea mai răspândită formă de vulnerabilitate este *Cross-Site Scripting*. Conform OWASP, două din trei aplicații au prezentat o astfel de vulnerabilitate [43]. Această vulnerabilitate folosește *browserul* victimei pentru a executa cod malicios ca parte din HML sau JavaScript. JavaScript este limbajul de programare folosit de *browserul web*. Există trei tipuri de XSS.

3.7.1 Reflected XSS

Acest atac presupune introducerea de cod extern în *browser* folosind doar un singur *request* HTTP. Un astfel de atac are loc atunci când un *browser* interpretează datele introduse de atacator prin *input-uri* nevalidate. Este o practică comună ca un atacator să trimită cod JavaScript pentru a efectua diferite acțiuni asupra paginii web. Atacatorii pot instala *key-loggers*, un timp de software care monitorizează tastele introduse de utilizator, pot modifica *link-uri* sau pot extrage date stocate în *browserul* victimei. Acest tip de atac este realizabil o singură dată, deoarece se bazează pe sistemul *request-response*, adică trimite o cerere serverului, iar acesta răspunde, în funcție de tipul *requestului,* cu datele solicitate. Un *link* este o referință la o secțiune în cadrul documentului din care se accesează sau la alt document disponibil online [44].

3.7.2 Stored XSS

Este același lucru cu tipul precedent, doar că, de data aceasta, aplicația reține codul introdus. Asta cauzează codul să fie executat permanent pe *browser*, astfel fiind posibil ca atacatorul să poată prelua controlul *browserului* victimei, să captureze informații sensibile și să scaneze rețeaua pe care aplicația rulează. Această vulnerabilitate este una dintre cele mai periculoase din cauza modului în care afectează utilizatorii. Un utilizator poate fi victima unui astfel de atac vizitând, pur și simplu o pagină web în care a fost introdus un cod malicios. Acest atac este și mai periculos în zonele des frecventate de administratorii paginii. Datele conturilor acestora pot fi furate prin astfel de metode, atacatorii preluând astfel controlul aplicației [45].

3.7.3 DOM XSS

Este foarte asemănător cu Stored XSS doar că este practicat îndeosebi pe aplicațiile de tip *single-page application* (SPA).

Toate aceste tipuri de vulnerabilități se ramifică în practici meticuloase pe care urmează să le prezint odată cu detalierea testelor implementate de *aplicația scanner.* Deci în capitolul următor voi prezenta arhitectura aplicației, împreună cu implementarea testelor de detecție pentru cele mai importante și răspândite vulnerabilități și voi detalia modul de utilizare și modul de raportare al *aplicației scanner.*

4. Arhitectura aplicației și implementarea testelor

Aplicația este structurată într-un singur fișier Python cu mod de execuție diferit în funcție de sistemul de operare de pe care aceasta este rulată. Raportul generat este afișat în trei fișiere cu nume specific conținutului fiecăruia.

Codul aplicației este structurat în trei clase ajutătoare și una principală care implementează testele propriu-zise, fiecare clasă fiind responsabilă pentru o anumită colecție de metode și tehnici utilizate în procesul de scanare. Utilizatorul dispune de flexibilitate prin intermediul fisișierului de configurare *config.ini* pe care îl poate modifica cu orice editor text (ex. Notepad++).

4.1 Parsarea variabilelor din fișierul de configurare

Acuratețea detecției vulnerabilităților este cea mai importantă calitate a unei *aplicații scanner*. În acest scop am creat un fișier de configurare *config.ini* accesat prin intermediul modulului *ConfigParser,* și am inițializat o variabila cu numele *config_object* cu rol de legatură între fisierul *.ini* și metodele de testare. Acest fișier are rolul de a oferi utilizatorului libertatea de a controla testele efectuate și flexibilitatea de a customiza testele pentru orice *site web*.

```
try:
    config_object = ConfigParser()
    config_object.read("config/config.ini")
except Exception:
    pass
```

Fișierul *config.ini* trebuie să existe în folderul *config* aflat la aceeași cale cu *scannerul*. Acesta conține text explicativ pentru o mai bună interacțiune cu utilizatorul și nume de variabile intuitive (vezi Figura 4.1).

```
[WEBURL]
;put here your target app URL. Ex: www.example.com
target = http://192.168.0.210/dvwa/
;if index exists put here your app index page, if NOT leave blank
index = http://192.168.0.210/dvwa/index.php
;put here your logout page. Ex: www.example.com/logout.php
logout = http://192.168.0.210/dvwa/logout.php
;put here your login page. Ex: www.example.com/login.php
login = http://192.168.0.210/dvwa/login.php
;put here links you want to exclude from analysis. Ex: www.example.com/logout.php
ignored = http://192.168.0.210/dvwa/logout.php
;put here some profile related URL. Ex: www.example.com/shopping_cart
private_info_url = None
[CREDENTIAL]
;put the known username used for login. Ex: admin
username = admin
;put another known username for login. Ex: admin2
username_2 = someuser
;put here the password corresponding to the first account
known_password = password
;put here the password corresponding to the second account
known_password_2 = somepassword
;put a non-existing username here. Ex: noexistuser
wrong_username = certainwrong
;put here a non existing password. Ex: thispasswordddoesnotexist
certain_wrong_passwd = onlyfortest
;put here the username field name
username_field = username
;put here the password field name
password_field = password
;put here the login field name
login_field = Login
;put here the submit button field name
submit_field = submit
```

Figura 4.1. Formatul fișierului de configurare (captură parțială)

4.2 Clasa LoginTests

Această clasă este prima clasă apelată la rularea programului. În momentul apelării aplicația *scanner* testează si încearcă autentificarea la aplicația de test folosind datele introduse în fișierul de configurare.

```
try_brute_force = LoginTests(
    config_object["CREDENTIAL"]["username"],
    config_object["WEBURL"]["login"],
    config_object["FILE"]["password_dict"],
    config_object["CREDENTIAL"]["wrong_username"],
    config_object["CREDENTIAL"]["known_password"],
    config_object["CREDENTIAL"]["certain_wrong_passwd"],
    config_object["WEBURL"]["logout"],
    report_file,
    error_file
)
```

Clasa accesează pagina de login aflată în fișierul de configurare și aplică pe aceasta o suită de teste din categoria *Broken Authentication.* Primul test pe care aplicația îl face este să detecteze daca *siteul web* este vulnerabil la *Brute Force.*

Brute Force este procedeul prin care un atacator ‘forțează’ autentificarea folosind un dicționar de date de autentificare. În cazul *aplicației scanner* dicționarul este *rockyou.txt*. *Rockyou.txt* este un set de parole extras dintr-un *Data Leak* de acum câțiva ani pe care l-am descărcat de pe un repo de Github [46]. Setul conține un număr de 14,341,564 parole unice pe care atacatorii, dar și testerii le folosesc frecvent în practica de *Brute Force*.

```
with open(config_object["FILE"]["password_dict"], "rb") as f:
    pass_list = f.readlines()
    f.close()
pass_q = queue.Queue()
if len(pass_list):
    for passwd in pass_list:
        try:
            passwd = passwd.decode("utf-8").rstrip()
            pass_q.put(passwd)
        except Exception as e:
            print(e)
            passwd = passwd.decode("latin-1").rstrip()
            pass_q.put(passwd)

for pass_word in pass_q.queue:
    http = requests.post(
        self.login_url,
        data={
            config_object["CREDENTIAL"]["username_field"]: self.username,
            config_object["CREDENTIAL"]["password_field"]: pass_word,
            config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
        }
    )
    if http.url == config_object["WEBURL"]["index"]:
        self.password = pass_word
        self.password_found = True
        return 1
```

Metoda de test *Brute Force* deschide fișierul ales de utilizator (în cazul nostru un fișier mai mic, care conține un fragment din *rockyou.txt*) și introduce parolele într-o coadă pentru a nu ține fișierul deschis în timpul testării. Odată ce parolele au fost încărcate și formatate pentru a putea fi introduse în cămpul de *login*, testul trimite un *request* de tipul POST cu datele de autentificare la adresa la care se află pagina de *login* a *aplicației web*. În acest test, numele de utilizator trebuie să fie unul existent pentru a avea rezultate mai rapid, astfel se evită procesul inutil, în cazul de față, de a testa și *username-uri*. *Username-ul* poate fi setat din fișierul de configurare.

Următorul test pe care *scannerul* îl face este testul de verificare a mesajelor de autentificare. Acest test se realizează în același timp cu testul de *Brute Force* și constă în verificarea erorilor apărute în cazul unei autentificări nereușite. Este posibil ca aplicația să furnizeze date utile unui atacator prin mesajele furnizate în cazul în care fie un nume de utilizator este greșit, dar parola este

corectă, fie numele de utilizator este corect și parola greșită. În acest caz, un atacator poate să-și formeze o listă de date valide. O astfel de vulnerabilitate este una din cauzele existenței dicționarelor cu date de autentificare.

```
print("Checking for Account Enumeration and Possible Guessable Users...",
file=self.report_file)
wrong_password_res = requests.post(
    self.login_url,
    data={
        config_object["CREDENTIAL"]["username_field"]: self.username,
        config_object["CREDENTIAL"]["password_field"]: self.wrong_passwords,
        config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
    }
)
wrong_password_res_content = str(wrong_password_res.content)
wrong_username_res = requests.post(
    self.login_url,
    data={
        config_object["CREDENTIAL"]["username_field"]: self.wrong_un,
        config_object["CREDENTIAL"]["password_field"]: self.good_password,
        config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
    }
)
wrong_username_req_content = str(wrong_username_res.content)
if wrong_username_req_content != wrong_password_res_content:
    return 1
return 0
```

Pentru a testa această vulnerabilitate am ales să testez mesajele returnate în ambele situații. Prima oară metoda testează mesajele cu un nume de utilizator corect și o parolă despre care știm sigur că greșită (este introdusă în fișierul de configurare de către utilizator). Se testează apoi prin aceeași metodă procesul invers, în care parola este corectă (din fișierul de configurare) și *usernameul* greșit. Capturăm *responseul* într-o variabilă, în ambele cazuri, și verificăm dacă există diferențe în acesta. Dacă răspunsul este afirmativ, atunci sunt mari sanșe ca această vulnerabilitate să fie prezentă.

O altă vulnerabilitate cunoscută în procesul de autentificare este chiar cauza pentru care metoda *Brute Force* este posibilă. Pentru a evita un atac de tipul *Brute Force* este nevoie ca aplicația să blocheze încercările nereușite de autentificare, succesive și repetate într-un interval scurt de timp.

```
def check_login_attempts(self, n):
    try:
        my_gui.update_list_gui("Checking Lock-Out Mechanism")
        self.session = requests.session()
        for i in range(n):
            self.session.post(
                self.login_url,
                data={
                    config_object["CREDENTIAL"]["username_field"]: self.username,
                    config_object["CREDENTIAL"]["password_field"]:
self.wrong_passwords[0],
                    config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
                }
            )
        correct_login = self.session.post(
            self.login_url,
            data={
                config_object["CREDENTIAL"]["username_field"]: self.username,
                config_object["CREDENTIAL"]["password_field"]: self.good_password,
                config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
            }
        )
        if correct_login.url == config_object["WEBURL"]["index"]:
            self.session.post(self.logout_url)
            self.session.close()
            print("[!!!-!!!] Wrong Password Lock Out Mechanism not triggered after ",
n, " times", file=self.report_file)
            return 1
        return 0
```

Metoda de test presupune încercarea de ”n” ori a unei autentificări nereușite urmate de o autentificare cu date valide. Dacă după ”n” ori *aplicația web* nu blochează încercările nereușite și permite autentificarea validă este posibil să existe această vulnerabilitate. Am implementat acest test prin autentificarea cu date greșite, din fișierul de configurare. După ”n” încercări am verificat autentificarea cu date valide, în cazul în care *scannerul* se autentifică cu succes înseamnă că mecanismul de *lock-out* nu a fost activat, așadar *aplicația* este vulnerabilă.

4.3 Clasa OtherUser

Pentru că orice aplicație de testare automată vrea să simuleze și să fie cel puțin la fel de eficientă, din punct de vedere al acurateței, cu o testare manuală, în unele situații pe care urmează să le detaliez mai târziu este nevoie de utilizarea a două sesiuni diferite, astfel am implementat o clasă care să creeze o sesiune diferită.

Folosim sesiuni separate atunci când vrem să testăm interacțiuni cu alți utilizatori. Aplicația inițializează o sesiune nouă cu ajutorul unor date de autentificare secundare, accesate din fișierul de configurare, și cu ajutorul modulului *requests*.

```
self.session = requests.Session()
```

Pentru a interacționa cu sesiunea am implementat două metode care returnează datele pe care o să le folosim pe parcursul testării.

```
def get_sess_id(self):
    try:
        global session_id
        self.session.get(self.url)
        cookie_dict = self.save_cookies(self.url)
        key_list = list(cookie_dict.keys())
        for key in key_list:
            if key.lower() == "sid" or "sessionid" or "session" or "sessiontoken" or
"sessid":
                session_id = cookie_dict[key]
                self.session.close()
                return session_id
        return
    except Exception as e:
        print("\n[ERROR] Something went wrong when trying to get session id from
another session. Error: ", e, file=self.error_file)
        pass
```

Pentru a accesa *ID-ul* sesiunii este nevoie să accesăm *cookie-urile* salvate atunci când o sesiune se crează. Un *cookie* este o colecție de fișiere de mici dimensiuni create pentru a reține date despre utilizatorul curent cu scopul de a îmbunătății experiența *web*. Deci din *cookie* putem extrage informații despre sesiunea curentă. Am salvat și stocat *cookie-urile* găsite, într-o variabilă pe care o parcurgem apoi pentru extrage și *IDul* sesiunii. În cazul în care este găsit un astfel de *ID* îl salvăm și îl returnăm.

Am salvat și întregul dicționar de *cookies* deoarece o să avem nevoie și de acesta la un moment dat.

```
def save_cookies(self, url):
    try:
        self.session.get(url)
        session_cookies = self.session.cookies
        cookies_dictionary = session_cookies.get_dict()
        return cookies_dictionary
    except Exception as e:
        print("\n[ERROR] Something went wrong when trying to return cookies from
another session. Error: ", e, file=self.error_file)
        pass
```

4.4 Clasa GUI

Un *Graphical User Interface* este o interfață prin care utilizatorul interacționează cu aplicația atât în timpul execuției cât și la finalul acesteia. Utilizatorul poate controla toate funcționalitățiile aplicației prin intermediul GUI.

Clasa are ca scop inițializarea și modificarea modelului GUI pentru cazul când sistemul de operare folosit este Windows. Clasa inițializează o fereastră goală asupra căreia aduc modificări pentru a o customiza.

În componența acesteia am implementat metodele care modifcă interfața și interacționează cu alogritmul și cu clasa de scanare. Metodele folosite au rolul de a "lega" anumite butoane la anumite funcții. La rularea programului, interfața are forma portret, o listă goală de teste efectuate, trei butoane și o bară de progres (vezi Figura 4.2).

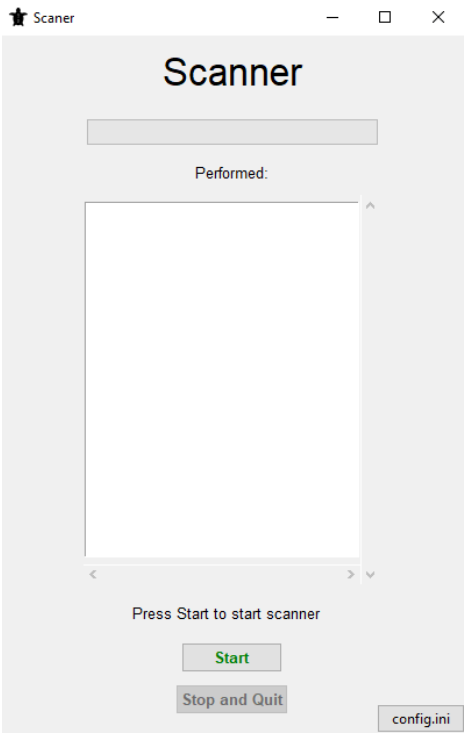


Figura 4.2. Interfața aplicației

Butoanele existente pe interfața sunt ‘legate’ la metodele din clasa GUI.

```
self.start_button = Button(
        self.gui_app, text="Start",
        command=self.start_scan,
        style='B1.TButton'
        )
self.start_button.pack(pady=5)
```

Prin argumentul *command* specificăm butonului ce metodă să apeleze. În cazul butonului *start* acesta apelează metoda *start_scan* care modifică interfața și începe execuția algoritmului de scanare prin crearea unui *fir de execuție* aplicat asupra funcției *start_program*. Acest lucru împiedică aplicația să devină *not responding* în procesul de scanare, fiind complet interacționabilă pe parcursul execuției.

```
def start_scan(self):
    try:
        global t1
        try:
            if self.open_report:
                self.open_report.pack_forget()
                self.close_button.pack_forget()
                self.stop_button = Button(self.gui_app, text="Stop and Quit",
command=self.stop_scan, style='B2.TButton')
                self.stop_button.pack(pady=5)
        except Exception:
            pass
        try:
            if self.list.index("end") != 0:
                self.list.delete(0, 'end')
        except Exception:
            pass
        self.gui_app.geometry('400x600')
        self.label.config(text='Wait..')
        self.state = 'started'
        self.start_button['state'] = DISABLED
        self.start_button.config(text='Scan')
        self.stop_button['state'] = NORMAL
        self.progress_bar['value'] = 0
        t1 = threading.Thread(target=start_program)
        t1.start()
    except Exception:
        pass
```

Mai multe detalii despre aspectul interfeței va fi prezentat în Subcapitolul *Modul de utilizare.*

Fiecare buton este legat la o astfel de funcție. Pentru butonul *Stop and Quit* este apelată funcția *stop_scan* care modifică aspectul și oprește forțat algoritmul.

```
def stop_scan(self):
    try:
        self.stop_button['state'] = DISABLED
        self.start_button['state'] = NORMAL
        self.label.config(text="Scan stopped!")
        self.kill_thread(t1)
        self.gui_app.quit()
        quit()
    except Exception:
        pass
```

La finalul scanării interfața este iarăși modificată pentru a oferi accesibilitate și informații utilizatorului (vezi Figura 4.3).

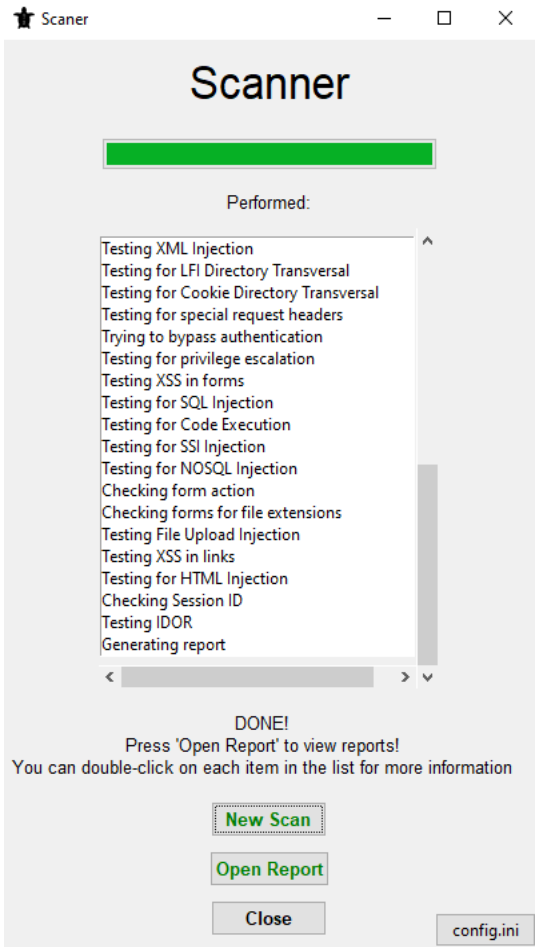


Figura 4.3. Interfața la finalul execuției

Metoda care aduce aceste modificări vizuale este apelată la finalul execuției algoritmului și folosește metode din modulul *Tkinter* pentru a manipula interfața.

```
def done_label(self):
    try:
        self.state = 'stopped'
        self.stop_button.pack_forget()
        try:
            if self.open_report:
                self.open_report.pack_forget()
                self.close_button.pack_forget()
        except Exception:
            pass
        self.gui_app.geometry('400x660')
        self.open_report = Button(self.gui_app, text="Open Report",
command=self.open_output, style='B1.TButton')
        self.open_report.pack(pady=5)
        self.close_button = Button(self.gui_app, text="Close", command=self.stop_scan,
style='B3.TButton')
        self.close_button.pack(pady=5)
        self.start_button['state'] = NORMAL
        self.start_button.config(text="New Scan")
        self.label.config(text="DONE!\nPress 'Open Report' to view reports!\nYou can
double-click on each item in the list for more information")
        self.gui_app.update()
        try:
            shutil.rmtree(temp_dir)
        except Exception:
            pass
    except Exception:
        pass
```

În această clasă se mai găsesc și alte metode utilitare cu rol de actualizare a datelor și aspectului. Spre exemplu metodele *update_list_gui* și *update_progress_bar* sunt apelate la fiecare pas de testare pentru a ține utilizatorul la curent cu progresul aplicației (vezi Subcapitolul 4.6.2 Interacțiunea cu GUI).

```
def update_list_gui(self, text):
    try:
        if not self.contain(text):
            self.list.insert(END, text)
    except Exception:
        pass

def update_progress_bar(self, amount):
    try:
        self.progress_bar['value'] += amount
        self.gui_app.update_idletasks()
    except Exception:
        pass
```

4.5 Clasa Scanner

Această clasă este responsabilă cu scanarea propriu-zisă. În această clasă se află atât metodele care gestionează diverse teste cât și metode care au rol utilitar.

Clasa este apelată în funcția *main* din cadrul funcției *start_program*. Funcția *start_program* are rolul de a deschide fișierele de *output*, în care urmează să fie generat raportul, și de a apela funcția *main* în care se apelează inițial clasa *LoginTests*, pentru stabilirea conexiunii și testarea vulnerabilităților pentru autentificare, urmată de clasa *Scanner*.

```
report_path = os.path.join(report_dir_path, "Report.txt")
open(report_path, 'w').close()
architecture_path = os.path.join(report_dir_path, 'Web Application Map.txt')
open(architecture_path, 'w').close()
error_path = os.path.join(report_dir_path, 'Error Log.txt')
open(error_path, 'w').close()

# Appends to emptied file
rep_file = open(report_path, 'a')
arch_file = open(architecture_path, 'a')
err_file = open(error_path, 'a')

# Runs program
main(rep_file, arch_file, err_file)
```

Funcția *main* este, de fapt, funcția care are ca rol inițializarea algoritmului de testare.

După realizarea testelor pentru autentificare, am apelat clasa *Scanner* cu datele returnate în procesul de testare menționat anterior și am inițializat o sesiune, urmând să apelez funcția care gestionează toate metodele de testare (funcția de rulare a algoritmului).

```
vuln_scanner.session.post(config_object["WEBURL"]["login"], data=data_dict)
vuln_scanner.run_scanner()
```

Variabila *data_dict* este un dicționar în care se află datele de conectare preluate din fișierul de configurare sau, în cazul în care în testarea *Brute Force* am găsit o parolă, o vom folosi pe aceea.

```
data_dict = {
    config_object["CREDENTIAL"]["username_field"]:
config_object["CREDENTIAL"]["username"],
    config_object["CREDENTIAL"]["password_field"]: found_password,
    config_object["CREDENTIAL"]["login_field"]:
config_object["CREDENTIAL"]["submit_field"]
}
```

Sesiunea este inițializată cu ajutorul modulului *requests* cu tipul POST la care adăugăm URLul de autentificare și *payload-ul* cu datele necesare autentificării cu succes. Un *payload* este un set de date, din componența unui mesaj, ce conține mesajul propriu-zis, fără să aibă alte informații utile pentru transport sau pentru alte scopuri, în componența lui.

Odata apelată metoda *run_scanner* aceasta apelează metode implementate în clasă, succesiv, în funcție de opțiunile pe care utilizatorul le-a ales în fișierul de configurare. În această metodă se află metode de culegere a informațiilor, de testare, de generare a raportului și de actualizare a interfeței.

4.5.1 Extragerea informațiilor

Înainte ca aceste teste să fie puse în practică, atât manual cât și automat, este nevoie să culegem informații despre aplicația testată.

```
def run_scanner(self):
    try:
        global final_list
        my_gui.update_progress_bar(10)
        # print('\rPlease wait..', end='', flush=True)
        self.gather_info()
        self.test_broken_auth(test=(True if
config_object['TEST']['test_broken_auth'].lower() == "true" else False))
        self.test_security_misconfiguration(test=(True if
config_object['TEST']['test_security_misconfiguration'].lower() == "true" else
False))
        self.test_sensitive_data_exposure(test=(True if
config_object['TEST']['test_sensitive_data_exposure'].lower() == "true" else
False))
        visible_links, final_list = self.create_return_merge_links()
```

Metodele de mai sus urmează să fie apelate o singură dată. O să vedem mai târziu că în componența acestei metode se află funcționalități care vor fi apelate de mai multe ori pe mai multe *link-uri*.

Metoda *gather_info* este metoda care începe procesul de extragere a *link-urilor* din *aplicația web*. Acest pas este unul foarte important deoarece pentru fiecare *link* extras, în funcție de forma lui, se vor aplica teste de detecție. Această metodă apelează o alta metodă utilitară din clasa *Scanner*, pe care am numit-o *spider*. Procesul de *spidering* este procesul prin care sunt extrase, înr-un mod recursiv, toate *link-urile* prezente în pagina sursă a unei *aplicații web*. În această metodă de spider se verifică numărul maxim de *link-uri* care pot fi extrase. Acest număr este configurabil în fișierul *config.ini*. *Link-urile* extrase sunt introduse apoi, pe rând, într-o listă.

```
for extracted_link in extracted_links:
    extracted_link = urllib.parse.urljoin(url, extracted_link)
    if "#" in extracted_link:
        extracted_link = extracted_link.split("#")[0]
    if self.target_url in extracted_link and extracted_link not in
self.target_links and extracted_link not in self.ignored_links and
len(self.target_links) <=
int(config_object['TEST']['max_number_of_not_hidden_links']):
        self.target_links.append(extracted_link)
        self.spider(extracted_link)
```

Am folosit *regular expression* pentru a localiza și extrage *link-urile* din codul sursă.

```
response = self.session.get(url)
return re.findall(
    'href="(.*?)"',
    str(response.content)
)
```

După acest pas *scannerul* folosește un dicționar pentru a căuta *link-uri* ascunse sau care nu sunt prezente în codul paginii *aplicației* și care pot să ne ofere acces la diverse informații sensibile. Pentru a modifica URL-ul și pentru a găsi astfel de *link-uri* în aplicație, am folosit unelte din modulul *urlparse* și am modificat calea URL cu căi din dicționar disponibil pe un repo Github, sursă de unde am descărcat dicționarul [47].

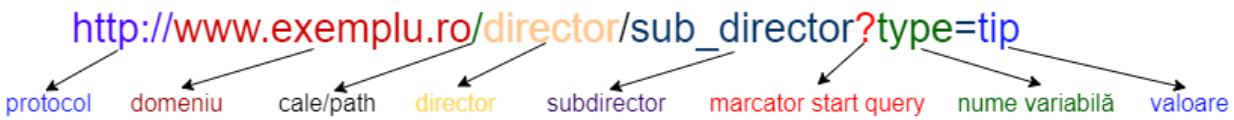


Figura 4.4 Structura unui URL

În procesul de căutare vom adăuga URLului standard, de forma *www.exemplu.ro* o cale către alte fișiere (vezi Figura 4.4). Verificăm dacă acea cale există trimițând un *request* de tip GET la adresa compusă la pasul anterior și verificăm dacă serverul ne răspunde cu codul 200, specific pentru mesajul de conexiune realizată cu succes (OK). După ce am găsit astfel de căi la care am primit mesajul OK, adăugăm într-o listă atât *link-urile* vizibile cât și pe cele ascunse. Aceste *link-uri* urmează să fie iterate și parcurse unul câte unul pentru a testa tot ce se poate pentru fiecare, astfel restrângem aria de căutare a vulnerabilităților în funcție de tipul de URL găsit.

Tot în pasul de extragere al informațiilor încercăm să analizăm rețeaua pe care serverul rulează. În cazul în care în componența URLului există un IP putem afla informații despre acesta cu ajutorul modulului *nmap*. Folosesc o expresie *re* pentru a căuta un IP în adresa *web* a *aplicației* (ex. www.80.5.12.32.com). Apelez modulul *nmap* cu argumentele *-Pn -sT -sV* pentru a obține informații despre porturi, servicii și versiuni disponibile pe *aplicație*. În cazul în care acest test este realizat cu succes, aceste informații urmează să fie introduse în raportul final.

```
web_app_ip_address = re.findall('http://(.*?)/',
config_object['WEBURL']['target'])
if web_app_ip_address:
    web_app_ip_address = [ip_address for ip_address in web_app_ip_address if
ip_address][0]
    nmap_scanner.scan(web_app_ip_address, '0-65535', arguments='-Pn -sT -sV')
    print('\nHost : ', nmap_scanner[web_app_ip_address].hostname(),
file=self.report_file)
    print('State : ', nmap_scanner[web_app_ip_address].state(),
file=self.report_file)
    for protocol in nmap_scanner[web_app_ip_address].all_protocols():
        print('----------', file=self.report_file)
        print('Protocol : ', protocol, file=self.report_file)
        port_list_inner = nmap_scanner[web_app_ip_address][protocol].keys()
        port_list_inner.sort()
        for port in port_list_inner:
            print('Port : ', port, '\tState : ',
nmap_scanner[web_app_ip_address][protocol][port]['state'],
file=self.report_file)
```

Alți pași care sunt realizați imediat după finalizarea funcției *gather_info* sunt testarea anumitor vulnerabilități din categoriile *Broken Authentication, Security Misconfiguration, Sensitive Data Exposure.* Înainte ca aceste funcții să fie apelate, se testează mai întâi dacă utilizatorul a optat pentru un astfel de test. Condiția urmatoare verifică dacă, în fișierul de configurare utilizatorul a selectat opțiunea pe care urmează să o testăm. Această verificare are loc la fiecare apel al acestor suite de funcții categoriale.

```
test=(True if config_object['TEST']['test_sensitive_data_exposure'].lower() ==
"true" else False)
```

În cazul în care utilizatorul a optat pentru astfel de teste, următorul pas al algoritmului este să caute *link-uri* administrative prin exact aceeași modalitate ca în cazul *link-urilor* ascunse, doar că de data asta, se folosește un dicționar care conține numeroase variante de *link-uri* administrative. Aceste *link-uri* au fost extrase dintr-un repo de Github ce cuprinde diverse colecții de astfel de liste utile pentru testare sau învățare [48].

După ce aplicația a terminat de căutat *link-urile* administrative, urmează să apeleze metoda care verifică tipul *Security Misconfigurations*. Aici se analizează lista ”robots.txt”. Această listă este legătura dintre motoarele de căutare și pagina *web*. Aici se introduc căile pe care dezvoltatorul le setează vizibile sau invizibile motoarelor de căutare. În acest pas *scannerul* caută *link-uri* cu informații sensibile pe care dezvoltatorul nu ar trebui să le lase acolo. Apoi cu ajutorul unui alt dicționar ce cuprinde *link-uri* populare din *robots.txt*, *scannerul* încearcă să găsească alte *link-uri* ascunse prin aceeași metodă ca în cazurile anterioare. Acest dicționar a fost extras dintr-un repo de Github ce cuprinde o varietate de astfel de dicționare utile pentru implementarea diverselor teste [49].

Se testează apoi metodele HTTP de transport și securitatea acestora. În mod normal o *aplicație web* nu trebuie să permită utilizatorilor accesul la metode de tip PUT, DELETE. Astfel de metode reprezintă vulnerabilitați în proiectarea aplicației. Pentru testarea metodei PUT sau DELETE, *scannerul* trimite un *request* de tip PUT sau DELETE aplicației web și analizează răspunsul primit. Pentru testarea securității metodei HTTP *scannerul* verifică dacă existența *header-ului Strict Strict Transport Security* este găsită în *response-ul aplicației web*. Un *header* este o porțiune dintr-un *request* care specifică informații despre acesta (tipul, browserul folosit, domeniul, etc).

```
headers = self.get_headers(config_object['WEBURL']['target'])
if 'strict' not in str(headers).lower():
    return True
return False
```

Un ultim test de tipul *Security Misconfiguration* non-recursiv este testul pentru *Rich Internet Applications* (RIA). Acestea sunt fișiere care specifică tipul de permisiune pe care *clientii web* (Java, Adobe Flash, Adobe Reader) le au asupra resurselor de pe *aplicație*. Este vital ca aceste permisiuni să fie restrictive pentru a proteja fișierele sensibile. *Scannerul* verifică dacă în componența codului sursă al paginii există astfel de fișiere în care permisiunile au fost setate ca *All* (*), în caz afirmativ aplicația este vulnerabilă.

```
if 'clientaccesspolicy.xml' in link.lower() or 'crossdomain.xml' in
link.lower():
    content = self.session.get(link)
try:
    if '*' in content:
        return True
except TypeError:
    pass
```

Ultima categorie testată non-recursiv este *Sensitive Data Exposure* unde se testează doar dacă aplicația web are un certificat SSL, și, în caz afirmativ, testează valabilitatea și entitatea emițătoare al acestuia.

```
context = ssl.create_default_context()
with socket.create_connection((self.target_url, self.port)) as sockk:
    with context.wrap_socket(sockk, server_hostname=self.target_url) as
tlssock:
        y = getattr(tlssock, to_check)
        return y()
```

Aplicația folosește un socket pentru a se conecta direct la *aplicație* apoi culege informațiile, despre certificat, de care are nevoie.

```
stripped = self.check_tls("getpeercert")["notAfter"].split(sep, -1)[:-1]
```

În cazul acesta *scannerul* extrage data de expirare a certificatului și o compară cu data actuală, utilizatorului îi este precizat în raport un mesaj informativ.

În același context *scannerul* verifică dacă entitatea emițătoare a certificatului este cunoscută și o returnează utilizatorului în raport.

4.5.2 Aplicarea testelor asupra informațiilor

La finalul execuției verificărilor realizate o singură dată per scanare, aplicația aplică teste asupra *link-urilor* găsite în procesul de *spidering* plus *link-urilor* ascunse. Lista finală este compusă din toate aceste *link-uri* găsite în procesul de *information gathering*, proces care presupune culegerea a cât mai multă informație despre *aplicația web.* Acest proces este comun atât atacatorilor, atunci când vor să infecteze o *aplicație web*, cât și *penetration testeriilor* care vor să testeze aplicația din punctul de vedere al securității.

În acest pas, *scannerul* extrage *tag-uri form* din codul HTML, adică extrage formularele și *input-urile* în care utilizatorul poate să introducă date. Așa cum am observat în Capitolul 3, există o mulțime de vulnerabilități cauzate de lipsa sanitizării *input-urilor*, deci este foarte important ca *scannerului* să aplice teste asupra acestor zone. Formularele sunt extrase cu ajutorul unei funcții utilitare *extract_forms* în care este folosit modulului *BeautifulSoup* pentru a selecta și *parsa* formularele, pentru ca ulterior, acestea să fie introduse înr-o listă. Modul prin care extragem formularele constă în capturarea

răspunsului pe care aplicația ni-l returnează și *parsarea* lui, modulului *BeautifulSoup* care decodează și formatează conținutul pentru a găsi și returna tag-uri de tip *form.*

```
response = self.session.get(url)
parsed_html = BeautifulSoup(response.content, "html.parser",
from_encoding="iso-8859-1")
return parsed_html.findAll("form")
```

Pentru fiecare formular se aplica teste din suitele *Injections, Senstive Data Exposure, Security Misconfiguration*.

Primul tip de vulnerabilitate testată este din tipul *Injections,* anume, Stored XSS. Aplicația pregătește un *payload* XSS de forma ”*<sCriPt>alert('test')</sCRiPt>*” și trimite un *request* cu acesta ca *submit* la formular. Metoda apelează funcția *submit_form* care are ca rol indentificarea câmpului de trimitere a formularului, tipului *request* și adăugarea *payload-ului* în corpul mesajului pe care urmează să îl transmită *aplicației web*.

```
xss_test_script = "<sCriPt>alert('test')</sCRiPt>"
response = self.submit_form(form, xss_test_script, url)
return xss_test_script.lower() in str(response.text) or xss_test_script in
str(response.text)
```

După trimiterea mesajului, verificăm dacă în răspuns se află *payload-ul* XSS. Dacă *aplicația web* nu ar fi vulnerabilă ar trebui să nu fie procesat tagul ‘<script>’. Așadar, dacă acest tag este încă existent în *response,* este clar că *aplicația* este vulnerabilă.

După testul XSS aplicația testează *injection* de tipul SQL. Acest test este asemănător cu mici excepții. Există două tipuri de *SQL Injections*, tipul în care ni se prezintă erori atunci cand încercăm caractere care pot fi interpretate de limbajul SQL, și tipul în care aceste erori sunt absente. Cel de-al doilea tip se numește *Blind SQL Injection*. Prin testul pe care *scannerul* îl face, ambele tipuri de *SQL Injections* sunt verificate cu succes.

```
global sql_injection_dict_injected, sql_injection_dict_normal
sql_no_payload = ""
response_wh_payload = self.submit_form(form, sql_no_payload, url)
normal_response_time = response_wh_payload.elapsed.total_seconds()
sql_detect_payload =
"IF(SUBSTR(@@version,1,1)<5,BENCHMARK(2000000,SHA1(0xDE7EC71F1)),"  \
                    "SLEEP(0.5))/*'XOR(IF(SUBSTR(@@version,1,1)<5," \

"BENCHMARK(2000000,SHA1(0xDE7EC71F1)),SLEEP(0.5)))OR'|'XOR(IF(SUBSTR(@@version,
1,1)<5," \
                    "BENCHMARK(2000000,SHA1(0xDE7EC71F1)),SLEEP(0.5)))OR'*/"
response_w_payload = self.submit_form(form, sql_detect_payload, url)
sql_response_time = response_w_payload.elapsed.total_seconds()

if sql_response_time > normal_response_time and sql_response_time > 1.5:
    sql_injection_dict_normal[url] = normal_response_time
    sql_injection_dict_injected[url] = sql_response_time
    return True
return False
```

Testarea constă în injectarea unui *payload* special destinat detecției ambelor tipuri de vulnerabilități . Acesta cauzează pagina web să ‘doarmă’, aproximativ 5 secunde. Deci aplicația injectează *payload-ul* și compară apoi timpul normal de răspuns (fără *payload*) cu timpul după ce *payload-ul* a fost încarcat. Dacă diferența este mai mare de 1.5 secunde, *scannerul* consideră aplicația vulnerabilă. *Payload-ul* a fost extras de pe siteul *Detectify,* site destinat testării și detectării vulnerabilităților [50]. *Scannerul* testează și tipul *NOSQL* în aceeași modalitate, doar că folosește un *payload* specific limbajului *NOSQL*.

Următoarele două tipuri de *injection* testat sunt *Code Execution* și *Server-Side Includes*. Acestea presupun introducerea unui *payload* care poate fi interpretat și rulat pe sistemul de operare. *Scannerul* introduce comanda de afișare a orei curente, compatibilă cu sistemele Linux, și apoi folosind o expresie *re*, caută în *response* un format de tipul *hh:mm:ss* pentru a detecta ambele forme de *injection*.

```
code_exec_script = "| uptime"
response = self.submit_form(form, code_exec_script, url)
return re.findall('\d\d:\d\d:\d\d', str(response.content))
```

Pentru suita *Sensitive Data Exposure* aplicată pe formulare, am implementat o verficare asupra acțiunii din formular.

```
action = form.get("action")
if not action:
    return False
elif "http" in action and "https" not in action:
    return True
return False
```

Rolul acestei implementări este de a verifica dacă există un formular pe care *aplicația web* îl transmite printr-un protocol HTTP nesecurizat. Am folosit metoda *.get()* din *BeautifulSoup* pentru a accesa acțiunea formularului.

Pentru testarea suitei *Security Misconfigurations* asupra formularelor, am implementat două teste. Primul test constă în verificarea extensiilor din acțiunea formularului. *Scannerul* verifică dacă în componența unui formular există extensii cu atributul *hidden*. Acestea pot fi de folos pentru a extrage informații sensibile din aplicație. De exemplu tehnologiile pe care aceasta le utilizează sau fișierele uitate de dezvoltator pot ajuta atacatorul în pregătirea unui *payload* malicios customizat pentru tipul de tehnologie care rulează pe *site*, astfel crescându-și șansele de reușită în cazul unui atac.

Al doilea test efectuat în această etapă este verificarea punctelor în care *aplicația web* primeste fișiere de la utilizatorii ei. Așa cum am exemplificat înainte, există riscul ca un atacator să creeze un fișier cu conținut malicios și să-i facă *upload*. Astfel acesta poate executa cod de la distanță reușind să preia controlul asupra serverului. *Scannerul* caută în formular cuvinte cheie specifice încărcării datelor. Dacă sunt găsite tipuri de *inputs* de forma *files* sau dacă șirul *'multipart/form-data'*, specific încărcării de fișiere, este prezent în componența formularului, *scannerul* încearcă să facă *upload* la fișierele niște fișiere generate și apoi verifică statusul *requestului*. În cazul în care statusul este OK, *scannerul* consideră că *aplicația web* este vulnerabilă. Fișierele generate sunt trimise la server prin intermediul funcției *submit_form* cu un argument în plus ce reprezintă fișierul de *upload.* Este generat un *request* în care fișierul este introdus în corpul mesajului. Fișierele de *upload* sunt generate de *scanner* în cazul în care este detectată prezența și sunt niște fișiere inofensive cu extensii posibil periculoase pentru un *site web* (.php, .exe, .html, .jps). Aceste fișiere sunt create într-un folder și reținute doar pe parcursul rulării *scannerului*, ele fiind șterse la finalul execuției cu ajutorul modulului *shutil*.

```
filenames = ["filefortest.php", "fIleForTest.Php.JpEG", "fiL3ForTest.hTMl.JPG",
"shell.phPWND", "fIleForTest.eXe.jsp"]
```

```
shutil.rmtree(temp_dir)
```

Acestea sunt toate testele pe care *scannerul* le aplică asupra formularelor. După acest pas *scannerul* testează toate *link-urile* găsite anterior pentru vulnerabilități asemănătoare sau specifice.

După ce testele asupra formularelor au luat sfârșit, *scannerul* testează, din nou, suitele *Injections, Broken Authentication, Sensitive Data Exposure, Broken Acces Control* și *XML External Entities* doar asupra *link-urilor*. În aceste suite se află teste realizate în funcție de parametrii pe care i-am introdus în apelarea funcției, astfel *scannerul* recunoaște tipul de test pe care urmează să-l acceseze din suita respectivă. Spre exemplu diferența dintre apelul simplu al suitei *Sensitive Data Exposure*, apelul pe formulare și apelul pe *link-uri* este dată de argumentele prezente în apel.

```
self.test_sensitive_data_exposure(test=(True if
config_object['TEST']['test_sensitive_data_exposure'].lower() == "true" else
False))
```

Acesta este un exemplu de apel singular al suitei *Sensitive Data Exposure.* Se observă absența oricărui parametru, în afară de parametrul specific verificării dacă utilizatorul a optat pentru test sau nu.

```
self.test_sensitive_data_exposure(link, form, test=(True if
config_object['TEST']['test_sensitive_data_exposure'].lower() == "true" else
False))
```

În cazul apelului pe formular se observă prezența parametrilor ‘form’ și ‘link’în apel.

```
self.test_sensitive_data_exposure(link, test=(True if
config_object['TEST']['test_sensitive_data_exposure'].lower() == "true" else
False))
```

Iar în cazul apelului doar pe *link-uri* se observă absența parametrului ’form’.

```
def test_sensitive_data_exposure(self, url=None, form=None, test=False):
    try:
        if test:
            if not url:
                self.check_tls_version()
                self.check_tls_validity()
                self.check_tls_issuer()
            else:
                if form:
                    if self.check_form_action(form):
                        links_forms__dict_sensitive_info[url] = form
                elif self.check_secure_tag_cookie_sessid(url):
                    links_without_secure_cookie_with_sessid.append(url)
```

Toate suitele de test sunt structurate la fel, dar diferența constă în modalitatea de apelare. În funcție de aceasta, *scannerul* apelează metodele de testare corespunzatoare metodei de apel. Observăm mai sus cum, în cazul în care apelul se realizează fără argumente (cu excepția argumentului de verficare a opțiunii din *config.ini*), *scannerul* aplică doar testele de verficare pentru certificatul SSL, iar în cazul în care apelul este realizat pe un formular *scannerul* aplică doar metodele de testare pe formular. Astfel *scannerul* fiind mai rapid în executarea verificărilor.

Pentru testele efectuate strict pe *link-uri*, din suita *Injections*, *scannerul* încearcă să detecteze vulnerabilitățile care sunt cauzate de modificarile aduse URLului. În acest caz unele testele au fost împărțite în două categorii. În cazul în care există un ”=” în componența URLului se vor aplica anumite teste aplicabile în aceste situații, iar în cazul în care URLul este standard se vor aplica restul testelor aplicabile, indiferent de formatul din URL (vezi Figura 4.4).

```
elif "=" in url:
    if self.test_xss_in_link(url):
        links_xss_link.append(url)
    if self.html_injection(url):
        links_html_injection.append(url)
    if self.ssrf_injection(url):
       links_ssrf_injection.append(url)

elif url:
    if self.javascript_exec(url):
        links_javascript_code.append(url)
    if self.host_header_injection(url):
        links_host_header_injection.append(url)
```

Cinci teste de tipul *Injection* au fost aplicate URL-urilor. Testele în care simbolul ”=” este prezent în url presupun înlocuirea șirului de caractere de dupa ”=” cu un șir malicios.

Am testat Reflected XSS în aceeași modalitate ca Stored XSS, aici excepția este că, în loc să trimit *payload-ul* XSS printr-un formular, am modificat șirul de dupa ‘=’ cu *payload-ul* XSS.

```
xss_test_script = "<sCriPt>alert('test')</sCRiPt>"
url = url.replace("=", "=" + xss_test_script)
```

Ca și în cazul Stored XSS, am trimis un *request* de tipul GET și am verificat existența *payload-ului* în *response-ul aplicației web.*

```
response = self.session.get(url)
return xss_test_script.lower() in str(response.text) or xss_test_script in
str(response.text)
```

Exact aceeași abordare a fost implementată pentru testarea *HTML Injection.* Această vulnerabilitate presupune modificarea URL-ului cu tag-uri HTML. În cazul în care aceste tag-uri nu sunt bine interpretate, există posibilitatea ca *aplicația web* să încorporeze *payload-ul* introdus în structura documentului paginii, fiind astfel posibilă rularea unui cod malicios. Singura diferență în implementarea testului este *payload-ul* HTML. Acesta abuzează de atributul *onerror* care ar trebui sa afișeze o eroare în pagină. În cazul acesta, *onerror*, nu ar trebui să interpreteze bucata codul *alert().*

```
html_payload =
"=<img%20src='aaa'%20onerror=alert(testforhtmlinjectionra872347)>"
url = url.replace('=', html_payload)
```

Server Side Request Forgery (SSRF) este un atac de tip *Injection* în care atacatorul modifică URLul cu instrucțiuni care pot fi rulate pe server. Pentru a testa această vulnerabilitate *scannerul* modifică șirul de după simbolul ”=” cu un șir care ar trebui să afișeze informații despre datele de login din *server.* Aceste date se află la calea */etc/passwd*.

```
ssrf_payload = '=file:///etc/passwd'
url = url.replace('=', ssrf_payload)
```

Testul constă în accesarea unei ”resurse” de pe server, daca aplicația este vulnerabilă resursa accesată trebuie să fie fișierul care stochează astfel de informațiile de autentificare în server. Pentru a verifica dacă *payload-ul* a fost interpretat de sistemul de operare, am verificat prezența șirului ”root:”, șir existent în aproape toate fișierele din calea ”/etc/passwd”, în *response*.

Pentru următorul test *scannerul* testează existența vulnerabilității *Javascript Code Execution.* Aceasta presupune introducerea unei noi căi cu cod JavaScript în URL. Aceeași metodă este abordată, doar că în cazul ăsta, nu se modifică ce este după simbolul ”=” ci doar ce este după simbolul ”/”, simbol care delimitează o cale în URL. Ca și în cazurile precedente se verifică existența bucății de cod în *response*.

```
js_payload = '/?javascript:alert(testedforjavascriptcodeexecutionrn3284)'
if url[-1] != '/':
    new_url = url + js_payload
else:
    new_url = url + js_payload[1:]
response = self.session.get(new_url)
```

Ultima vulnerabilitate de tipul *Injections* testată de *scanner* este *Host Header Injection.* Această vulnerabilitate presupune modificarea cu succes a *headerelor Host* și *X-Forwarded-Host*. Aceste două *Headere* dețin informații cu privire la pagina *web* accesată. În mod normal o *aplicație web* nu trebuie să permită modificarea acestor două *Headere,* deoarece un atacator poate cauza o redirecționare către un domeniu controlat de acesta. *Scannerul* testează existența acestei vulnerabilități introducând domeniul ”www.google.com” în acestea și verificând dacă aplicația răspunde OK sau nu. Ideal ar fi ca aplicația să raspundă cu un mesaj de redirecționare înapoi pe un domeniu controlat de aceasta. În cazul în care mesajul este OK, este clar că aplicația nu redirecționează astfel de domenii străine, deci este vulnerabilă. Testul este realizat printr-un *request* de tip GET cu *Headerele* modificate.

```
host = {'Host': 'www.google.com'}
x_host = {'X-Forwarded-Host': 'www.google.com'}
if self.session.get(url, headers=host).status_code == 200:
    return True
elif self.session.get(url, headers=x_host).status_code == 200:
    return True
return False
```

Următoarea suită testată este *Broken Authentication*. Aici *scannerul* testează dacă sesiunea este gestionată într-un mod securizat, browserul salvează datele local într-un *cache*, posibilitatea modificării rolului pe care un utilizator îl are și dacă vulnerabilitatea *Local File Inclusion* (LFI) este prezentă.

LFI este un proces prin care un atacator include un fișier deja existent pe server. Adică acesta accesează fișiere restricționate. Această vulnerabilitate este testată asemănător cu testul SSRF, diferența este *payload-ul* introdus ”~”. *Scannerul* verifică dacă, odata introdus acest simbol, aplicația răspunde cu OK. În caz afirmativ această vulnerabilitate este prezentă.

Pentru verificarea sesiunii, *scannerul* verifică dacă sesiunea este afișată în *cookies* în text clar și dacă *cookieul* respectiv are setat atributul ‘*secure*’ și atributul ‘*HttpOnly*’. În

cazul în care *cookie-ul* are sesiunea expusă dar are și atributul *'HttpOnly' aplicația web* poate fi vulnerabilă. În lipsa atributului '*secure'* și prezenței sesiunii în *cookie aplicația* este vulnerabilă.

```
if "sid" or "sessionid" or "session" or "sessiontoken" or "sessid" in
str(cookie_dict).lower():
    if not str(cookie_dict.lower()["secure"]) or
str(cookie_dict.lower()["httpOnly"]):
```

Următorul test realizat în această suită este testul în care se verifică dacă rolul utilizatorului este setat prin *cookies.* Dacă rolurile sunt setate prin *cookies*, un atacator poate să modifice aceste roluri pentru obține acces administrativ. *Scannerul* testează prezența cuvintelor care denotă setarea rolului. În cazul în care se găsesc astfel de cuvinte cheie, *scannerul* consideră aplicația vulnerabilă.

```
if "isadmin" in str(cookie_dict).lower():
    if str(cookie_dict.lower()["isAdmin"]).lower() == "true" or\
            str(cookie_dict.lower()["isAdministrator"]).lower() == "true" or\
            str(cookie_dict.lower()["admin"]).lower() == "true" or\
            str(cookie_dict.lower()["administrator"]).lower() == "true":
```

Ultimul test realizat, din suita *Broken Authentication,* este verificarea modalității în care *aplicația web* stochează datele sensibile. Nu este recomandat ca o aplicație să stocheze datele în *cache* pentru ficare pagină. Este posibil ca un atacator să aibă acces la calculatorul victimei, în cazul în care *aplicația* a stocat date de autentificare, date bancare sau orice alte date sensibile local, într-un *cache*, atacatorul poate avea acces la acestea. Memoria *cache* este o memorie în care sunt stocate date temporar, pentru a evita procesul de recalculare a datelor originale, în această memorie sunt stocate doar copii ale datelor deja existente.

Scannerul verifică configurarea atributelor din *Header*. Atributul *Cache-Control* este responsabil pentru accesul la memoria *Cache* pe pagina *web.* În cazul detecției unui astfel de atribuit *scannerul* verifică dacă acesta este setat să rețina datele fără nici o restricție. În caz afirmativ este reținut *linkul* la care această vulnerabilitate este detectată fiind la alegerea utilizatorului *scannerului* dacă *link-ul* reprezintă o zonă cu informații care nu ar trebui să fie reținute.

```
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"):
```

Singurul test realizat de suita *Sensitive Data Exposure* este testul în care se verifică dacă sesiunea este reținută în *cookies* și dacă toate *cookie-urile* au atributul *secure.*

Am ales ca *XML Injection* să nu fie încadrată la categoria *Injections.* Din cauză că proiectul a respectat clasificarea OWASP Top Ten, am decis să păstrez aceeași logică și să încadrez separat acest tip de vulnerabilitate. Pentru testul *XML External Entities* (XEE), *scannerul* caută în document fișiere XML (.xml) și, în caz că există, introduce pe rând mai multe *payload-uri* de detecție. *Payload-urile* au ca scop deschiderea paginii ”www.google.com”. În cazul în care această pagină este deschisă *scannerul* consideră aplicația vulnerabilă.

Ultima suită de teste pe care *scannerul* o verifică este *Broken Access Control.*

Insecure Direct Object References (IDOR) este un tip de vulnerabilitate care are loc atunci când *aplicația web* are legătură directă cu obiectele acesteia. Atacatorii pot exploata această vulnerabilitate prin modificarea referințelor la obiect. În implementarea acestui test, am folosit metode *Regular Expression* pentru a modifica numerele aflate după simbolul ”=” într-un URL. De exemplu pentru șirul ”www.exemplu.ro/factura=1” am modificat ”1” în ”2”. În cazul în care *aplicație web* nu ne redirecționează și ne oferă acces la resursa solicitată există riscul să existe IDOR. Am verificat dacă aplicația ne redirecționează la pagina de unde am încercat un atac IDOR, am reținut *response-ul* inițial și l-am comparat apoi cu *response-ul* după atac. Am considerat diferența în conținut și mesajul OK, în urma IDOR, ca semn al existenței vulnerabilității.

```
sub_string = re.findall('[?](.*)[=]*\d', url)
if sub_string:
    index_from_url = int(str(re.findall('\d', str(sub_string))))
    response = self.session.get(url)
    while attempts < 10:
            url.replace(str(index_from_url), index_from_url + 1)
            response_2 = self.session.get(url)
            if response != response_2 and
str(response_2.status_code).startswith("2"):
```

Urmatoarele teste efecutate au ca scop detecția vulnerabilităților de tip *LFI Directory Transversal* și *Remote File Inclusion* (RFI) . *LFI Directory Transversal* este asemănătoare cu LFI doar că, în cazul acesta am folosit caractere codate pentru o șansă de reușită mai mare. Caracterele sunt codate în formatul ” UTF-8”, un format des întâlnit în protocoalele HTTP. Pentru detecția RFI am ales să modific șirul de după ”=” din URL cu

un domeniul ”www.google.com”, acest domeniu simulează domeniul malicios al unui atacator. Șirul de caractere folosit a fost extras de pe un repo de Github ce cuprinde o mulțime de *payload-uri* utile pentru detecția vulnerabilităților [51].

```
self.lfi_script(url, "%252e%252e%252fetc%252fpasswd%00")
self.rfi_script(url, "https://www.google.com/"):
```

Dacă *aplicația web* nu redirecționează domeniul extern sau dacă în urma testului domeniul este ”www.google.com”, *scannerul* consideră aplicația vulnerabilă.

Pentru urmatorul test *scannerul* modifică *Headerele* ”X-Original-URL” și ”X-Rewrite-URL” pentru a modifica domeniul actual cu unul inexistent. Acest test are ca scop verificarea controlului de acces. Dacă *aplicația* nu este bine configurată ne asteptăm să primim mesajul 404 (Not Found). În acest caz *scannerul* consideră aplicația vulnerabilă.

```
response_w_x_original = self.session.get(url, headers={"X-Original-URL":
"/donotexistrandomstring1238123"})
response_w_x_rewrite = self.session.get(url, headers={"X-Rewrite-URL":
"/donotexistrandomstring1238123"})
if response_wo_headers.status_code == 404 or response_w_x_original.status_code
== 404 or response_w_x_rewrite.status_code == 404:
```

Următorul test verifică dacă este posibil ca un utilizator să-și modifice drepturile.

Am realizat acest test atașând la corpul mesajului un set de instrucțiuni menite să modifice drepturile.

```
privilege_data = ["groupID=grp001&orderID=0001", "grpID=2&item=1",
"grp=group1", "role=5"]
```

```
if "grp" or "group" or "role" in str(response.text).lower() or "grp" or "group"
or "role" in response.url.lower():
    for data in privilege_data:
        response = self.session.post(url, data=data)
        if response.status_code != 401:
            return 1
```

Înițial am testat dacă în *response-ul aplicației* există astfel de cuvinte cheie care ar putea semnala prezența unei astfel de posibilități. Dacă astfel de cuvinte există am introdus pe rând instrucțiuni menite să modifice drepturile curente. Dacă aplicația nu ne returnează mesajul *Unauthorized* (401) *scannerul* consideră *aplicația web* vulnerabilă.

Ultimul test implementat este testul pentru impersonarea altui utilizator. În acest test am folosit clasa OtherUser pentru a extrage *IDul* altei sesiuni. Apoi am încercat să

modific sesiunea curentă cu noua sesiune ”furată” și să accesez o resursă privată, disponibilă doar altui utilizator. În cazul în care interfața *aplicației web* ne permite accesul este clar că aplicația este vulnerabilă.

```
dummy_user = OtherUser(config_object["CREDENTIAL"]["username_2"],
                       config_object["CREDENTIAL"]["known_password_2"], url,
err_file=self.error_file)
dummy_session = dummy_user.session
dummy_session.get(url)
dummy_session.post(config_object["WEBURL"]["private_info_url"], data=data)
```

Este necesar ca utilizatorul *scannerului* să introducă un astfel de URL, disponibil doar utilizatorilor logati, în fișierul de configurare.

4.6 Generarea raportului și interacțiunea cu GUI

4.6.1 Generarea raportului

La finalul tuturor testelor este apelată metoda responabilă cu generarea raportului. În fiecare caz în care *scannerul* a considerat aplicația vulnerabilă, a adăugat datele detecției într-un dicționar sau listă reprezentativă declarată global.

```
if self.test_privilege_escalation(url):
    links_privilege_escal.append(url)
if self.test_xss_in_form(form, url):
    links_forms_dict_xss[url] = form
if self.test_sql(form, url):
    links_forms_dict_sql_injection[url] = form
if self.code_exec(form, url):
    links_forms_dict_code_exec[url] = form
if self.ssi_injection(form, url):
    links_forms_dict_ssi_injection[url] = form
if self.test_nosql(form, url):
    links_forms_dict_nosql_injection[url] = form
```

În pasul de generare al raportului aceste liste sunt afișate în fișierul de raport împreună cu mesaje orientative pentru utilizator (vezi Figura 4.5).

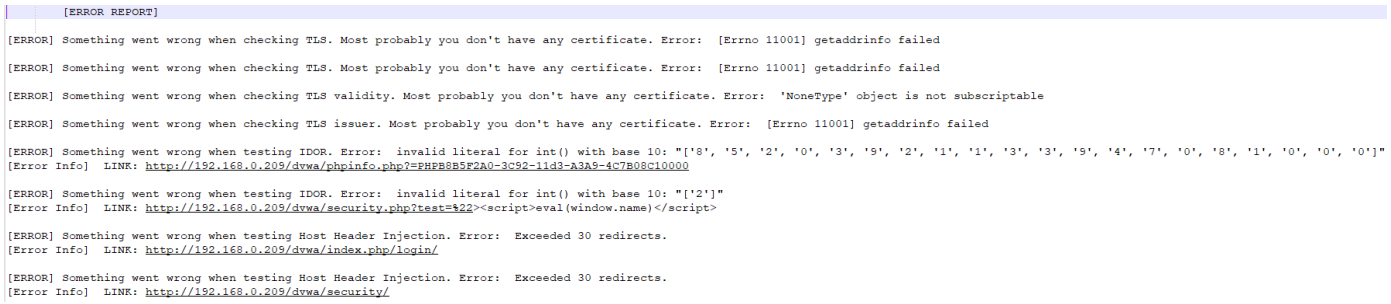
```
if links_bypass_authorization:
    print("\n[!!!---!!!] Bypassing Authorization Vulnerability:\n",
file=self.report_file)
    print("Horizontal Bypassing Authorization on links:\n",
file=self.report_file)
    print(*links_bypass_authorization, sep="\n", file=self.report_file)
else:
    print("\nBypassing Authorization Test Performed", file=self.report_file)
    print("[WARN] No Bypassing Authorization vulnerability found!",
file=self.report_file)

if links_special_header:
    print("[!!!---!!!] Special Request Header Handling on links:\n",
file=self.report_file)
    print(*links_special_header, sep="\n", file=self.report_file)
    print("[END] End of Bypassing Authorization Vulnerability",
file=self.report_file)
else:
    print("\nSpecial Request Header Handling Test Performed",
file=self.report_file)
    print("[WARN] No Special Request Header Handling vulnerability found!",
file=self.report_file)
```

```
                    [@@@]          REPORT          [@@@]


        [LOGIN REPORT]
[!!!-!!!] Wrong Password Lock Out Mechanism not triggered after  5  times
[!!!-!!!] Weak Lockout Mechanism found for a number of invalid password attempts
Checking for Account Enumeration and Possible Guessable Users...
OK! No accounts can be enumerated
[!!!---!!!] Brute Force attack successful!!
Username: admin
Found password: password
        [END LOGIN REPORT]

[NMAP] Nmap Scan Results
[END] End of Nmap Scan Results

[!!!---???] Possible Admin Path discovered for links:
http://192.168.0.209/dvwa/login/admin/
http://192.168.0.209/dvwa/login/admin/admin.asp
http://192.168.0.209/dvwa/login/administrator/
[END] End of admin paths

Analyzing robots.txt for interesting urls..!
No directories found inside robots.txt

[!!!---!!!] HTTP Strict Transport Security NOT found. Application is vulnerable to sniffing and certificate invalidation vulnerability!

Checking TLS extensions version...
[???-???] Cannot find the version of the TLS extension...

Checking Validity of Digital Certificate...
[???-???] Cannot check validity of the TLS extension...

Checking Digital Certificate Issuer...
[???-???] Cannot check issuer of the TLS extension...

Visible Links Length: 26
Hidden Links Length: 34


        [DETAILED REPORT]


Server/s Found:
Apache/2.2.8 (Ubuntu) DAV/2

[!!!-???] Comments inside HTML DOM found.
Found comments inside HTML DOM. Make sure these comments do not contain any sensitive information.
 end align div
```

Figura 4.5. Structura fișierului de raport (captură parțială)

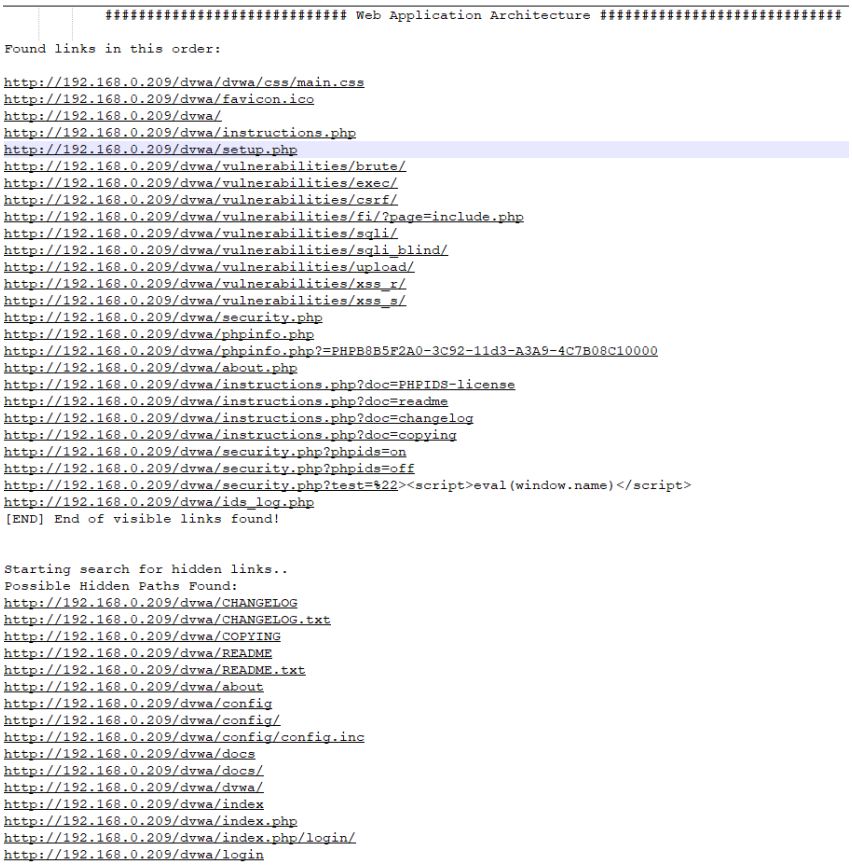
Pe langă fișierul de raport există încă un fișier destinat erorilor (Error Log.txt) (vezi Figura 4.6) și unul destinat arhitecturii *aplicației web* (Web Application Map.txt) (vezi

Figura 4.7). În acestea se găsesc informații suplimentare despre erorile apărute pe parcursul execuție și, respectiv, despre *link-urile* și formularele asupra cărora s-au efectuat testele.



```
        [ERROR REPORT]

[ERROR] Something went wrong when checking TLS. Most probably you don't have any certificate. Error:  [Errno 11001] getaddrinfo failed

[ERROR] Something went wrong when checking TLS. Most probably you don't have any certificate. Error:  [Errno 11001] getaddrinfo failed

[ERROR] Something went wrong when checking TLS validity. Most probably you don't have any certificate. Error:  'NoneType' object is not subscriptable

[ERROR] Something went wrong when checking TLS issuer. Most probably you don't have any certificate. Error:  [Errno 11001] getaddrinfo failed

[ERROR] Something went wrong when testing IDOR. Error:  invalid literal for int() with base 10: "['8', '5', '2', '0', '3', '9', '2', '1', '1', '3', '3', '9', '4', '7', '0', '8', '1', '0', '0', '0']"
[Error Info]  LINK: http://192.168.0.209/dvwa/phpinfo.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000

[ERROR] Something went wrong when testing IDOR. Error:  invalid literal for int() with base 10: "['2']"
[Error Info]  LINK: http://192.168.0.209/dvwa/security.php?test=%22><script>eval(window.name)</script>

[ERROR] Something went wrong when testing Host Header Injection. Error:  Exceeded 30 redirects.
[Error Info]  LINK: http://192.168.0.209/dvwa/index.php/login/

[ERROR] Something went wrong when testing Host Header Injection. Error:  Exceeded 30 redirects.
[Error Info]  LINK: http://192.168.0.209/dvwa/security/
```

Figura 4.6. Fișierul Error Log.txt (captură parțială)



```
############################## Web Application Architecture ##############################

Found links in this order:

http://192.168.0.209/dvwa/dvwa/css/main.css
http://192.168.0.209/dvwa/favicon.ico
http://192.168.0.209/dvwa/
http://192.168.0.209/dvwa/instructions.php
http://192.168.0.209/dvwa/setup.php
http://192.168.0.209/dvwa/vulnerabilities/brute/
http://192.168.0.209/dvwa/vulnerabilities/exec/
http://192.168.0.209/dvwa/vulnerabilities/csrf/
http://192.168.0.209/dvwa/vulnerabilities/fi/?page=include.php
http://192.168.0.209/dvwa/vulnerabilities/sqli/
http://192.168.0.209/dvwa/vulnerabilities/sqli_blind/
http://192.168.0.209/dvwa/vulnerabilities/upload/
http://192.168.0.209/dvwa/vulnerabilities/xss_r/
http://192.168.0.209/dvwa/vulnerabilities/xss_s/
http://192.168.0.209/dvwa/security.php
http://192.168.0.209/dvwa/phpinfo.php
http://192.168.0.209/dvwa/phpinfo.php?=PHPB8B5F2A0-3C92-11d3-A3A9-4C7B08C10000
http://192.168.0.209/dvwa/about.php
http://192.168.0.209/dvwa/instructions.php?doc=PHPIDS-license
http://192.168.0.209/dvwa/instructions.php?doc=readme
http://192.168.0.209/dvwa/instructions.php?doc=changelog
http://192.168.0.209/dvwa/instructions.php?doc=copying
http://192.168.0.209/dvwa/security.php?phpids=on
http://192.168.0.209/dvwa/security.php?phpids=off
http://192.168.0.209/dvwa/security.php?test=%22><script>eval(window.name)</script>
http://192.168.0.209/dvwa/ids_log.php
[END] End of visible links found!


Starting search for hidden links..
Possible Hidden Paths Found:
http://192.168.0.209/dvwa/CHANGELOG
http://192.168.0.209/dvwa/CHANGELOG.txt
http://192.168.0.209/dvwa/COPYING
http://192.168.0.209/dvwa/README
http://192.168.0.209/dvwa/README.txt
http://192.168.0.209/dvwa/about
http://192.168.0.209/dvwa/config
http://192.168.0.209/dvwa/config/
http://192.168.0.209/dvwa/config/config.inc
http://192.168.0.209/dvwa/docs
http://192.168.0.209/dvwa/docs/
http://192.168.0.209/dvwa/dvwa/
http://192.168.0.209/dvwa/index
http://192.168.0.209/dvwa/index.php
http://192.168.0.209/dvwa/index.php/login/
http://192.168.0.209/dvwa/login
```

Figura 4.7. Fișierul Web Application Map.txt (captură parțială)

4.6.2 Interacțiunea cu GUI

Scannerul interacționează cu GUI în permanentă pentru a ține la curent utilizatorul cu testele efectuate și timpul rămas până la finalizarea acestora.

La fiecare test semnificativ efectuat, lista din interfață se actualizează și bara de progres se modifică. Inițial bara de progres este încarcată 10%, urmând ca, pentru fiecare *link* parcurs să se încarce pana la 100%, sau, până când toate *link-urile* au fost parcurse (vezi Figura 4.3).

```
my_gui.update_list_gui("Checking robots.txt")
my_gui.update_list_gui("Checking forms for file extensions")
my_gui.update_list_gui("Testing RIA")
my_gui.update_list_gui("Testing XSS in links")
my_gui.update_list_gui("Searching for admin directories")
my_gui.update_progress_bar(10)

for count, link in enumerate(final_list):
        my_gui.update_progress_bar(90/len(final_list))
```

La finalul execuției algoritmului este apelată funcția *done_label* (vezi Figura 4.3 și Subcapitolul Clasa GUI).

Cele două funcții (*print_report()* și *my_gui.done_label()*) sunt apelate la sfârșitul execuției și reprezintă ultimele funcționalități ale *scannerului.*

4.7 Descărcare și instalare

Aplicația poate fi descărcată de pe pagina mea personală de Github [52]. Acolo se află atât fișierele pentru instalare Windows cât și fișierele pentru instalare Linux.

4.7.1 Windows

În cazul instalării aplicației pe un sistem Windows este îndeajuns descărcarea aplicației impreună cu fișierele de configurare prezente în *repository-ul* de pe Github.

4.7.2 Linux

Pe un sistem Linux utilizatorul poate să descarce *repository-ul* cu ajutorul comenzii ”*$ git clone https://github.com/Ptmlol/LicentaScanner”.* Apoi este nevoie ca utilizatorul să instaleze dependențele cu ajutorul comenzii ”*pip3 install -r requirements.txt*”.

4.8 Modul de utilizare

4.8.1 Windows

Pe sistemul Windows utilizatorul interacționează cu aplicația prin intermediul GUI. Aplicația se execută cu dublu-click pe fișierul executabil (vezi Figura 4.8).

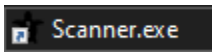


Figura 4.8. Fișierul executabil

Interfața de deschidere este formată dintr-un titlu, o bară de progres, o căsuță în care urmează să apară date despre testele efectuate și o zonă a butoanelor (vezi Figura 4.2).

Înainte de a apăsa start este nevoie ca utilizatorul *scannerului* să configureze aplicația în funcție de preferințe. Așadar poate accesa butonul din dreapta jos (config.ini) pentru a deschide rapid folderul în care fișierul de configurare se află (vezi Figura 4.1). În acest fișier utilizatorul poate modifica diversele funcționalități ale aplicației, cum ar fi testele pe care le efectuează sau numărul maxim de *link-uri* pe care aplicația *scanner* le caută, în plus poate să customizeze anumite câmpuri pentru o mai bună acuratețe și rezultate mai rapide.

După configurarea necesară, utilizatorul poate apăsa butonul ”Start” pentru a începe procesul de scanare. Pe parcursul procesului aplicația oferă un mic feedback afișat în lista ”Progress”. Utilizatorul poate opri oricand execuția cu ajutorul butonului ”Stop and Quit”, acest buton oprește execuția și închide interfața grafică.

La finalul execuției apar trei butoane diferite cu nume reprezentative funcționalității (vezi Figura 4.3). Utilizatorul poate opta acum pentru o nouă scanare (New Scan), poate deschide folderul în care se află raportul (Open Report) sau poate să închidă aplicația (Close).

Dacă utilizatorul nu este familiarizat cu noțiunile testate în procesul de scanare acesta poate să dea dublu-click în lista ”Performed:” pentru a deschide *browserul* implicit și pentru a afla mai multe informații despre testul selectat.

4.8.2 Linux

Programul poate fi rulat în terminal folosind comanda ”python3 scanner.py”. Raportul generat este afișat în același director cu directorul de unde s-a rulat programul.

5. Testarea și verificarea aplicației

Pentru ca orice aplicație trebuie să fie testată riguros înainte de a fi disponibilă publicului, am implementat un mic set de teste efectuate pentru a mă asigura că aplicația îndeplinește anumite funcționalități.

5.1 Testarea GUI

La apăsarea butonului ”Start” aplicația *scanner* trebuie să înceapă procesul de scanare indiferent dacă *aplicația web* pe care urmează să fie aplicate testele există sau nu. Deci există două cazuri:

1. *Aplicația web* există:
    - Începe procesul de testare (vezi Figura 5.1)

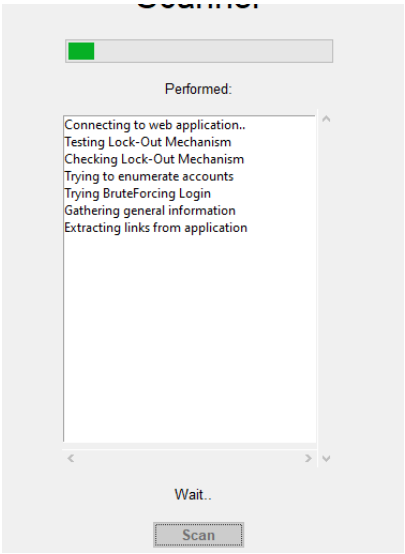


Figura 5.1. Butonul Start (captură parțială)

2. *Aplicația web* nu există
    - Este afișat un mesaj cu rol informativ (vezi Figura 5.2)

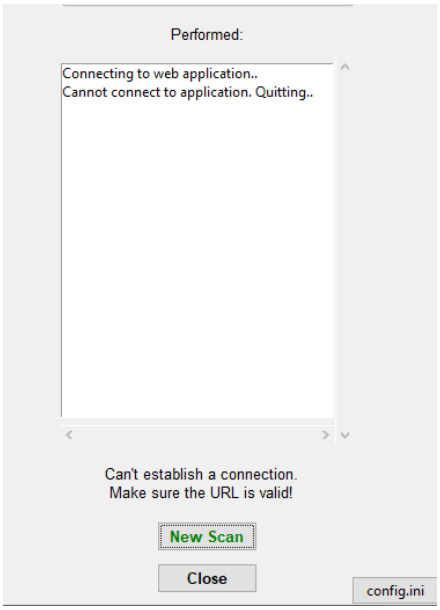


Figura 5.2. Procesul nu poate începe (captură parțială)

Butonul ”Stop and Quit” nu poate fi apăsat inițial fără ca aplicația să se afle în procesul de rulare (vezi Figura 4.2). În timpul rulării, butonul de ”Start” trebuie să fie inactiv și butonul de ”Stop and Quit” trebuie să aibă culoarea roșie și să fie interacționabil (vezi Figura 5.3).

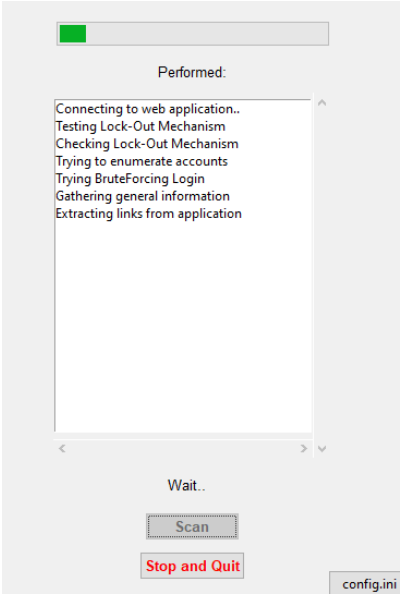


Figura 5.3. Aplicația în rulare (captură parțială)

La apăsarea butonului ”config.ini” aplicația deschide folderul în care se află fișierul de configurare (vezi Figura 5.4).

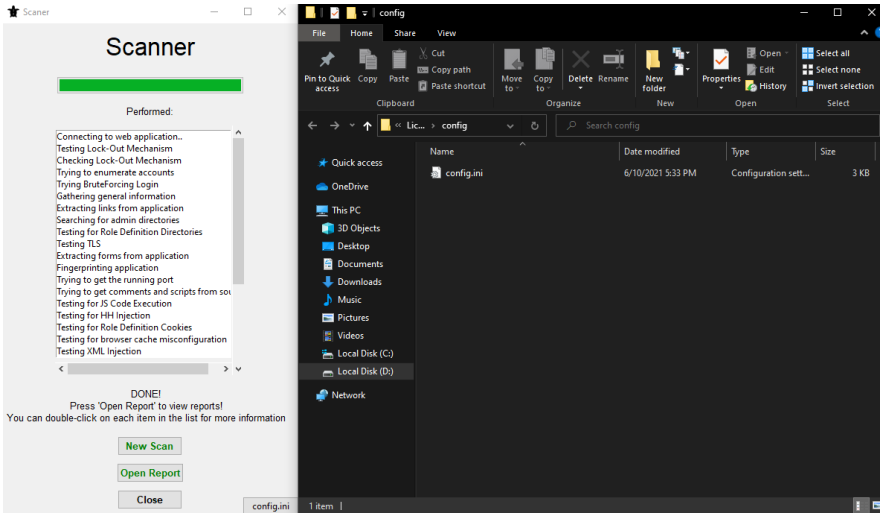


Figura 5.4. Apăsarea butonului ”config.ini”

La finalul execuției apar apar trei butoane cu numele ”New Scan”, ”Open Report” și ”Close” (vezi Figura 4.3). La apăsarea butonului ”New Scan” aplicația începe un nou proces de scanare (vezi Figura 5.4). La apăsarea butonului ”Open Report” aplicația deschide folderul în care se află fișierele de report (vezi Figura 5.5). La apăsarea butonului ”Close” aplicația închide UI-ul și execuția este oprită.

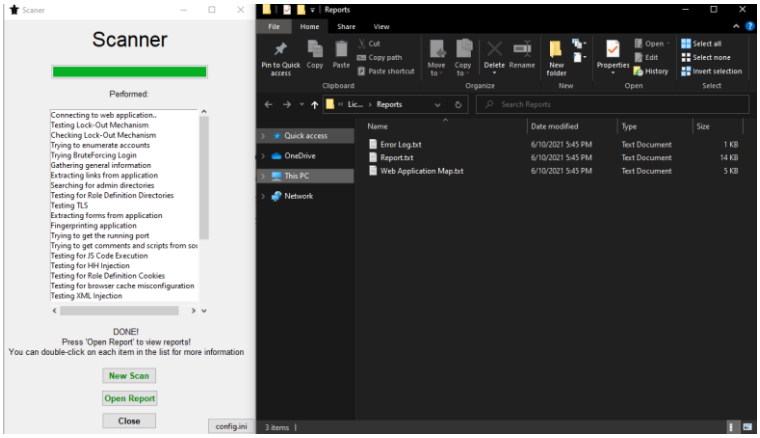


Figura 5.5. Apăsarea butonului ”Open Report”

În timpul execuției bara de progres se încarcă și în lista ”Performed:” apar testele efectuate (vezi Figura 5.6).

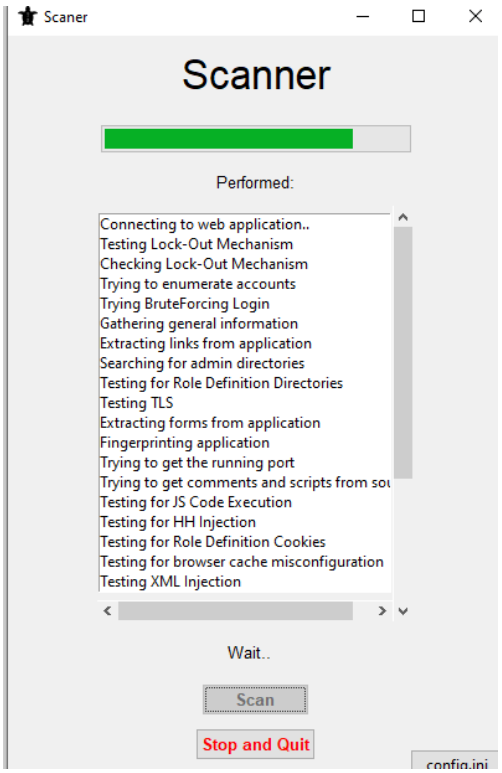


Figura 5.6. Aplicația în rulare

5.2 Testarea algoritmului de detecție

Aplicația DVWA, pe care se realizează testarea aplicației *scanner*, are un număr de vulnerabilități cunoscute. Majoritatea sunt disponibile chiar pe interfața aplicației în partea stângă (vezi Figura 5.7). Alte vulnerabilități detectate de scanner sunt prezente în aplicație și sunt documentate online [53]. Se observă în fișierul de raport (vezi Figura 4.5) că aproape toate vulnerabilitățile detectate de *scanner* sunt prezente în lista de vulnerabilități cunoscute, *scannerul* având o acuratete destul de ridicată.

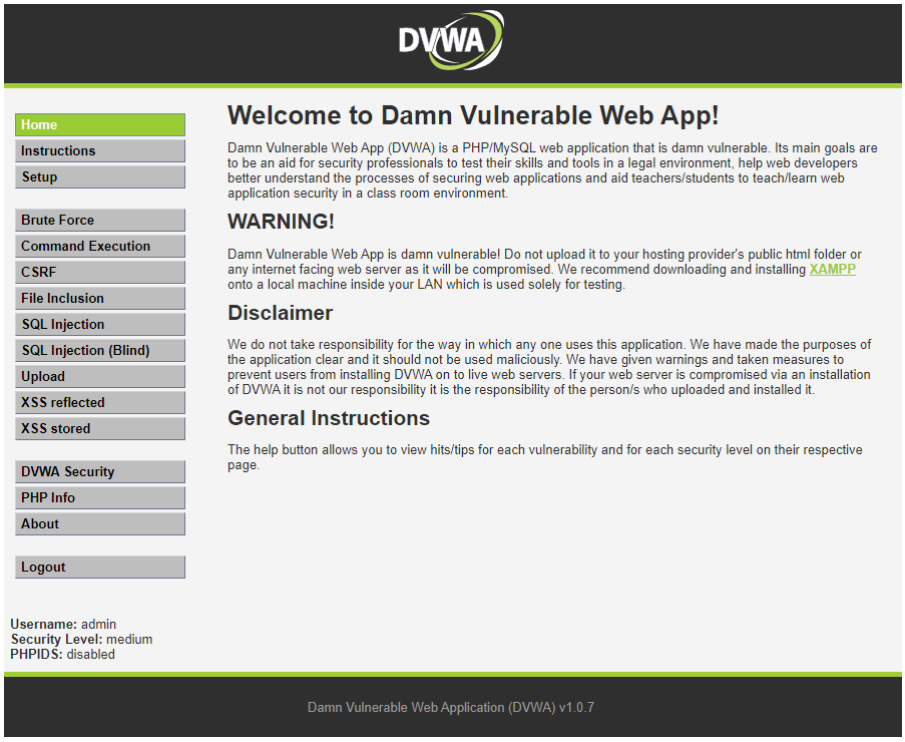


Figura 5.7. Interfața aplicației DVWA [21].

În acest capitol am demonstrat cum aplicația *scanner* își îndeplinește cu succes scopul și prezintă o acuratețe destul de ridicată pentru a putea fi folosită în mediul de dezvoltare.

6. Concluzii și îmbunătățiri

În această aplicație am urmărit să ofer dezvoltatorilor sau persoanelor a căror arie de lucru nu se află în domeniul securității, o unealtă eficientă și usor de folosit, pentru a-i ajuta în procesul de dezvoltare a aplicațiilor web și pentru a se familiariza mai ușor cu noțiunile din securitatea web. S-a încercat păstrarea unui echilibru între flexibilitate și acuratețe, în acest scop am implementat teste pentru vulnerabilități cât mai des întâlnite, și am implementat metode de testare generale care să acopere o arie cât mai largă din *aplicația web* testată.

În comparație cu alte astfel de unelte, *scannerul* este ușor de folosit, nu are o dimensiune mare, poate fi folosit pentru orice tip de aplicație web, fie că este o aplicație în producție fie în dezvoltare, nu consumă foarte multe resurse și oferă un *feedback* ușor de interpretat. Spre exemplu aplicația *OWASP Zed Attack Proxy* (ZAP) poate fi dificil de utilizat pentru un dezvoltator cu puține cunoștințe despre securitate și care nu are nevoie de o scanare aprofundată și greu de implementat.

Ca orice altă aplicație aceasta poate fi îmbunătățită prin metode precum:

- Adăugarea unui proxy care să captureze *request-urile* astfel se elimină orice verificare *client-sided;*
- Adăugarea unor teste mai precise pentru vulnerabilitățile existente;
- Detectarea mai multor vulnerabilități;
- Îmbunătățirea interfeței;
- Dicționare de dimensiuni mai mari;
- Teste customizabile;
- Reprezentarea unui mod de raportare mai detaliat.

În încheiere aș vrea să atrag atenția asupra importanței securității în viitor. Cu cât tehnologia avansează cu atât uneltele folosite de atacatori vor fi mai puternice, așadar este important ca dezvoltatorii și testerii să aibă la dispoziție o gamă cât mai largă de instrumente cât mai eficiente care să combată proactiv încercările atacatorilor.

Bibliografie

[1] T. B. Azad, Securing Citrix Presentation Server in the Enterprise, Syngress, 2008.

[2] J. D. GROOT, What is Cyber Security? Definition, Best Practices & More, *Data Insider*, 2020.

[3] G. S. M. E. Zachary Cohen, What we know about the pipeline ransomware attack: How it happened, who is responsible and more, CNN, 2021.

[4] R. Gibb, What is a Web Application?, *Stackpath*, 2016.

[5] OWASP, Web Application Security Testing, [Interactiv]. Available: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/. [Accesat 01 Iunie 2021].

[6] Python, [Interactiv]. Available: https://docs.python.org/3/tutorial/. [Accesat 02 Iunie 2021].

[7] Requests: HTTP for Humans, [Interactiv]. Available: https://docs.python-requests.org/en/master/. [Accesat 02 Iunie 2021].

[8] Urllib, [Interactiv]. Available: https://docs.python.org/3/library/urllib.html. [Accesat 02 Iunie 2021].

[9] BeautifulSoup, [Interactiv]. Available: https://www.crummy.com/software/BeautifulSoup/bs4/doc/. [Accesat 02 Iunie 2021].

[10] urllib.parse — Parse URLs into components, [Interactiv]. Available: https://docs.python.org/3/library/urllib.parse.html#module-urllib.parse. [Accesat 02 Iunie 2021].

[11] urllib.request — Extensible library for opening URLs, [Interactiv]. Available: https://docs.python.org/3/library/urllib.request.html#module-urllib.request. [Accesat 02 Iunie 2021].

[12] SSL, [Interactiv]. Available: https://docs.python.org/3/library/ssl.html. [Accesat 02 Iunie 2021].

[13] Socket, [Interactiv]. Available: https://docs.python.org/3/library/socket.html. [Accesat 02 Iunie 2021].

[14] Subprocess, [Interactiv]. Available: https://docs.python.org/3/library/subprocess.html. [Accesat 02 Iunie 2021].

[15] Randy. [Interactiv]. Available: https://whatsabyte.com/blog/processor-threads/. [Accesat 02 Iunie 2021].

[16] TKINTER, [Interactiv]. Available: https://docs.python.org/3/library/tkinter.html. [Accesat 02 Iunie 2021].

[17] OS, [Interactiv]. Available: https://docs.python.org/3/library/os.html. [Accesat 02 Iunie 2021].

[18] RE, [Interactiv]. Available: https://docs.python.org/3/library/re.html. [Accesat 02 Iunie 2021].

[19] ConfigParser, [Interactiv]. Available: https://docs.python.org/3/library/configparser.html. [Accesat 02 Iunie 2021].

[20] Metasploitable 2, [Interactiv]. Available: https://docs.rapid7.com/metasploit/metasploitable-2-exploitability-guide/. [Accesat 02 Iunie 2021].

[21] R. Wood, DVWA, [Interactiv]. Available: https://github.com/digininja/DVWA. [Accesat 02 Iunie 2021].

[22] Queue, [Interactiv]. Available: https://docs.python.org/3/library/queue.html. [Accesat 02 Iunie 2021].

[23] DateTime, [Interactiv]. Available: https://docs.python.org/3/library/datetime.html. [Accesat 02 Iunie 2021].

[24] Python-Nmap, [Interactiv]. Available: https://pypi.org/project/python-nmap/. [Accesat 02 Iunie 2021].

[25] ctypes — A foreign function library for Python, [Interactiv]. Available: https://docs.python.org/3/library/ctypes.html. [Accesat 02 Iunie 2021].

[26] Shutil, [Interactiv]. Available: https://docs.python.org/3/library/shutil.html. [Accesat 02 Iunie 2021].

[27] WebBrowser, [Interactiv]. Available: https://docs.python.org/3/library/webbrowser.html. [Accesat 02 Iunie 2021].

[28] OWASP, Web Application Security Testing, [Interactiv]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/. [Accesat 02 Iunie 2021].

[29] OWASP, Injection, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A1_2017-Injection. [Accesat 02 Iunie 2021].

[30] SQL, [Interactiv]. Available: https://docs.oracle.com/cd/E12151_01/doc.150/e12152/toc.htm. [Accesat 02 Iunie 2021].

[31] OWASP, SQL Injection, [Interactiv]. Available: https://owasp.org/www-community/attacks/SQL_Injection. [Accesat 02 Iunie 2021].

[32] OWASP, NOSQL Injection, [Interactiv]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection. [Accesat 02 Iunie 2021].

[33] OWASP, Command Injection, [Interactiv]. Available: https://owasp.org/www-community/attacks/Command_Injection. [Accesat 02 Iunie 2021].

[34] OWASP, Broken Authentication, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A2_2017-Broken_Authentication. [Accesat 02 Iunie 2021].

[35] OWASP, Credential Stuffing, [Interactiv]. Available: https://owasp.org/www-community/attacks/Credential_stuffing. [Accesat 02 Iunie 2021].

[36] OWASP, Session Management, [Interactiv]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html. [Accesat 02 Iunie 2021].

[37] OWASP, Sensitive Data Exposure, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure. [Accesat 02 Iunie 2021].

[38] A. Kumar, Fake SSL Certificates: How Can They Be a Problem?, [Interactiv]. Available: https://medium.com/globant/fake-ssl-certificates-how-can-they-be-a-problem-901cfe0b34f7. [Accesat 02 Iunie 2021].

[39] OWASP, XML External Entities, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A4_2017-XML_External_Entities_(XXE). [Accesat 02 Iunie 2021].

[40] Introducing JSON, [Interactiv]. Available: https://www.json.org/json-en.html. [Accesat 02 Iunie 2021].

[41] OWASP, Broken Access Control, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A5_2017-Broken_Access_Control. [Accesat 02 Iunie 2021].

[42] OWASP, Security Misconfiguration, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A6_2017-Security_Misconfiguration. [Accesat 02 Iunie 2021].

[43] OWASP, Cross-Site-Scripting, [Interactiv]. Available: https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS). [Accesat 02 Iunie 2021].

[44] OWASP, Testing for Reflected XSS, [Interactiv]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting. [Accesat 02 Iunie 2021].

[45] OWASP, Testing for Stored XSS, [Interactiv]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/02-Testing_for_Stored_Cross_Site_Scripting. [Accesat 02 Iunie 2021].

[46] Rockyou, [Interactiv]. Available: https://github.com/praetorian-inc/Hob0Rules/blob/master/wordlists/rockyou.txt.gz. [Accesat 04 Iunie 2021].

[47] R. Janicki. [Interactiv]. Available: https://github.com/bl4de/dictionaries. [Accesat 04 Iunie 2021].

[48] spaddex. [Interactiv]. Available: https://github.com/spaddex/Admin-Finder/blob/master/admin_locations.txt. [Accesat 04 Iunie 2021].

[49] Y. H. Wong. [Interactiv]. Available: https://github.com/yeukhon/robots-txt-scanner/tree/master/tests/robots. [Accesat 04 Iunie 2021].

[50] F. N. A. Mathias Karlsson, Detectify, [Interactiv]. Available: https://labs.detectify.com/2013/05/29/the-ultimate-sql-injection-payload/. [Accesat 04 Iunie 2021].

[51] Swissky. [Interactiv]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/File%20Inclusion/README.md. [Accesat 04 Iunie 2021].

[52] N. Teodor. [Interactiv]. Available: https://github.com/Ptmlol/LicentaScanner. [Accesat 04 Iunie 2021].

[53] M. Whittle, Ethical Hacking (Part 1): OWASP Top 10 and DVWA, [Interactiv]. Available: https://levelup.gitconnected.com/ethical-hacking-part-1-owasp-top-10-and-dvwa-3f2d55580ba8. [Accesat 06 Iunie 2021].