Analisis Kerentanan dan Pengembangan Kerangka Pencegahan Serangan Cross-Site Scripting (XSS) pada Aplikasi Web Berbasis PHP

Graciella Nathania
Informatika
Universitas Multimedia Nusantara
Tangerang, Indonesia
graciella.nathania@student.umn.ac.id

Christopher Bryan Budi Susanto
Informatika
Universitas Multimedia Nusantara
Tangerang, Indonesia
christopher.bryan1@student.umn.ac.id

Winson Sovian
Informatika
Universitas Multimedia Nusantara
Tangerang, Indonesia
winson.sovian@student.umn.ac.id

Kelvin Sutadi
Informatika
Universitas Multimedia Nusantara
Tangerang, Indonesia
kelvin.sutadi@student.umn.ac.id

Jeremy Joseph Pohan
Informatika
Universitas Multimedia Nusantara
Tangerang, Indonesia
jeremy.joseph@student.umn.ac.id

Abstrak*—Cross-Site Scripting (XSS) merupakan salah satu kerentanan paling kritis dalam aplikasi web, khususnya pada sistem berbasis PHP yang memiliki fleksibilitas tinggi namun rentan terhadap input berbahaya. Penelitian ini bertujuan untuk menganalisis kerentanan terhadap tiga tipe utama XSS: Reflected, Stored, dan DOM-based, serta merancang kerangka pencegahan melalui pendekatan eksperimen. Metode yang digunakan adalah kuantitatif deskriptif dengan pembuatan simulasi aplikasi rentan, injeksi payload XSS, dan analisis terhadap respons sistem dan browser. Temuan menunjukkan bahwa ketiadaan validasi dan penyandian input menjadi akar dari celah XSS. Penerapan fungsi htmlspecialchars(), textContent, dan kebijakan Content-Security-Policy (CSP) terbukti mampu mencegah eksekusi skrip berbahaya pada ketiga skenario pengujian. Pendekatan ini diperkuat oleh studi Mohanty & Acharya (2021) yang mengusulkan pengujian kombinatif black-box dan white-box, serta Zhao et al. (2024) yang menekankan pentingnya analisis aliran data berbasis opcode PHP. Selain itu, pendekatan deteksi berbasis asosiasi input-output seperti yang dikembangkan Li et al. (2020) semakin menegaskan bahwa validasi dan penyandian kontekstual adalah elemen krusial dalam mitigasi XSS. Hasil akhir menunjukkan bahwa penerapan strategi pengamanan berlapis dapat secara signifikan meningkatkan ketahanan aplikasi terhadap serangan XSS.

***Kata Kunci*— XSS, Validasi Input, PHP, Content-Security-Policy, Keamanan Web**

I. PENDAHULUAN

Perkembangan teknologi informasi telah mendorong semakin luasnya pemanfaatan aplikasi *web* dalam berbagai sektor. Namun, seiring dengan meningkatnya penggunaan *web*, risiko terhadap keamanan siber juga semakin kompleks. Salah satu jenis ancaman yang menonjol adalah *Cross-Site Scripting (XSS)*, yaitu serangan yang memanfaatkan kelemahan dalam validasi dan penyajian input pengguna untuk menyisipkan skrip berbahaya ke dalam halaman *web* [1].

XSS termasuk dalam kategori kerentanan injeksi dan telah secara konsisten masuk dalam daftar OWASP *Top* 10 sebagai salah satu ancaman paling kritis bagi aplikasi *web* [2]. Bahasa pemrograman PHP menjadi sangat relevan dalam konteks ini, mengingat banyaknya aplikasi *web* yang dikembangkan menggunakan PHP. Fleksibilitas PHP yang tinggi sering kali menjadi celah jika pengembang tidak menerapkan praktik keamanan yang baik.

Penelitian ini bertujuan untuk melakukan investigasi terhadap mekanisme kerja serangan XSS dalam aplikasi berbasis PHP serta mengevaluasi strategi pencegahan yang dapat diterapkan, termasuk penggunaan fungsi pengaman bawaan PHP dan *framework* modern.

II. METODE PENELITIAN

Penelitian ini dilakukan untuk menganalisis tingkat kerentanan terhadap serangan *Cross-Site Scripting* (XSS) dalam aplikasi *web* berbasis PHP serta mengembangkan kerangka pencegahan yang efektif. Metode yang digunakan bersifat eksperimen, dengan pendekatan kuantitatif deskriptif untuk mengamati dan mengukur dampak dari serangan XSS pada aplikasi uji. Menurut Creswell (2014), pendekatan kuantitatif digunakan untuk menguji teori dan hipotesis melalui pengumpulan data berbasis angka dan analisis statistik, sedangkan pendekatan eksperimen bertujuan untuk menemukan hubungan sebab-akibat dengan memanipulasi variabel dalam lingkungan yang terkontrol [3].

Dalam penelitian ini, proses dilakukan melalui pembuatan simulasi aplikasi yang rentan terhadap XSS, injeksi *payload* berbahaya, serta analisis terhadap respon dari sisi klien. Evaluasi sistem dilakukan untuk mengetahui bagaimana aplikasi menangani *input* berbahaya dan apakah mekanisme pertahanan yang diterapkan dapat mengurangi risiko serangan.

Pendekatan ini didukung oleh studi Mohanty dan Acharya (2021) yang menekankan pentingnya kombinasi pengujian *black-box* dan *white-box* untuk mendeteksi XSS secara lebih menyeluruh. Mereka menyatakan bahwa “kombinasi metode pengujian ini mampu meningkatkan tingkat deteksi dan mengurangi kemungkinan kesalahan dalam penanganan input pengguna” [4]. Sementara itu, penelitian oleh Zhao et al. (2024) memperkenalkan Yama, sebuah pendekatan analisis aliran data berbasis opcode PHP yang sensitif terhadap

konteks dan jalur. Mereka menyatakan bahwa “pemanfaatan semantik *opcode* PHP memungkinkan pendeteksian jalur data berbahaya yang sebelumnya tidak terdeteksi oleh pendekatan tradisional” [5]. Dengan pendekatan ini, pendeteksian kerentanan menjadi lebih presisi hingga pada tingkat instruksi program, sehingga dapat mendukung pencegahan serangan XSS secara efektif.

A. Objek Penelitian

Penelitian ini menggunakan tiga buah skrip PHP sederhana yang dibangun untuk mewakili tiga jenis serangan XSS:

1. *Reflected XSS*
   Input dari pengguna dikirim melalui URL dan langsung ditampilkan tanpa disimpan.

2. *Stored XSS*
   Input berbahaya disimpan di dalam *database*, lalu ditampilkan kembali kepada pengguna lain.

3. *DOM-based XSS*
   Input dimanipulasi melalui *JavaScript* langsung di browser tanpa melalui server.

Setiap objek diimplementasikan pada *file* berbeda (*reflected_xss.php*, *stored_xss.php*, dan *dom_xss.php*) untuk memudahkan pengujian secara terpisah. Pendekatan ini sesuai dengan model eksperimen rekayasa perangkat lunak yang dijelaskan oleh Pressman, di mana skenario uji dibuat untuk mengamati respons sistem terhadap *input* yang disengaja [6].

B. Teknik Pengumpulan Data

Metode pengumpulan data dalam penelitian ini meliputi:

1. Studi Literatur
   Referensi dalam penelitian ini diambil dari jurnal ilmiah terkini, buku teks keamanan web, serta dokumentasi OWASP yang membahas konsep XSS dan teknik mitigasinya. Validasi terhadap proses sanitasi input dan output merupakan fondasi penting dalam pengujian keamanan web. Li et al. (2020) mengembangkan Cross-Site Scripting Guardian, sebuah alat deteksi XSS statis berbasis penambangan asosiasi input-output aliran data. Mereka menyatakan bahwa "pendekatan ini dapat secara efektif mendeteksi kerentanan XSS dengan mengidentifikasi hubungan antara input yang tidak tepercaya dan output yang berpotensi berbahaya" [7].

2. Ekspretimen Langsung
   Peneliti melakukan simulasi serangan XSS dengan menyisipkan berbagai *payload* ke dalam aplikasi uji, kemudian mengamati *output* dan perilaku *browser*. Pendekatan ini sejalan dengan metode yang dikembangkan oleh Mohammadi et al. (2018), yang menggunakan pengujian unit otomatis untuk mendeteksi kerentanan XSS dengan menghasilkan *input* berbasis *grammar* dan mengevaluasi respons aplikasi terhadap skrip berbahaya [8].

3. Analisis Kode Sumber
   Kode PHP dari masing-masing skenario dianalisis untuk menelusuri titik lemah dalam proses pengolahan *input* dan *output*, sesuai dengan pendekatan analisis statis yang umum digunakan dalam deteksi kerentanan [9].

C. Alat dan Lingkungan Uji

Eksperimen dilakukan pada lingkungan pengujian lokal untuk memastikan keamanan dan kontrol terhadap skenario yang dijalankan. Konfigurasi uji meliputi:

- Bahasa Pemrograman : PHP versi 8.3.10
- *Server Web*: Apache (XAMPP)
- *Database*: *MySQL*
- *Browser* : Google Chrome dengan *Developer Tools*
- *Tools* Tambahan: *Visual Studio Code*

Lingkungan lokal memberikan fleksibilitas untuk pengujian destruktif tanpa membahayakan sistem produksi atau jaringan eksternal [10].

D. Teknik Analisis Data

Data yang diperoleh dari eksperimen dianalisis secara deskriptif dengan pendekatan eksploratif. Tiap *payload* XSS diklasifikasikan ke dalam dua hasil utama: berhasil (*vulnerable*) jika skrip dijalankan, dan gagal (*secure*) jika aplikasi mampu mencegah eksekusi skrip. Analisis dilakukan terhadap:

- Efektivitas eksekusi skrip
- Reaksi sistem terhadap *payload*
- Ketepatan penanganan *input*/*output*
- Apakah kerentanan bersifat reflektif, persisten, atau berbasis DOM

Analisis ini bertujuan untuk memberikan pemahaman komprehensif mengenai kelemahan sistem dan titik-titik penting di mana mitigasi seharusnya diterapkan [11].

III. HASIL DAN PEMBAHASAN

Evaluasi terhadap kerentanan *Cross-Site Scripting* (XSS) dilakukan melalui serangkaian eksperimen terstruktur yang merepresentasikan tiga jenis serangan XSS, yaitu *Reflected XSS*, *Stored XSS*, dan *DOM-based XSS*. Setiap eksperimen diimplementasikan dalam skenario aplikasi *web* sederhana berbasis PHP yang secara sengaja tidak dilengkapi dengan mekanisme validasi *input* atau penyandian *output*.

Tujuan dari pengujian ini adalah untuk mengidentifikasi bagaimana aplikasi merespons *input* berbahaya yang dikirimkan oleh pengguna dan sejauh mana skrip dapat dijalankan oleh *browser* di sisi klien. Evaluasi dilakukan secara deskriptif berdasarkan efektivitas eksekusi *payload*, jalur masuk data, dan risiko keamanan terhadap pengguna akhir.

A. *Reflected XSS*

Pada skenario ini, input berbahaya dikirim melalui form HTML dengan metode *GET* dan secara langsung ditampilkan oleh server ke dalam halaman tanpa penyaringan. Berikut adalah cuplikan kode PHP yang digunakan dalam eksperimen:

Gambar 1. Form input pengguna pada web browser

```
<form method="get">
    <label>Masukkan Nama Anda:</label><br>
    <input type="text" name="nama">
    <input type="submit" value="Kirim">
</form>
```

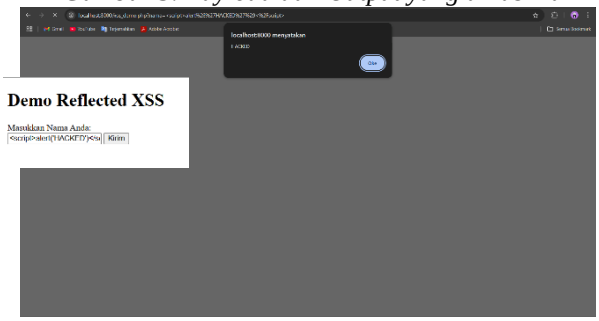
Gambar 1 menunjukkan antarmuka pengguna tempat input dimasukkan ke dalam kolom teks. Input dari pengguna dikirim menggunakan metode *GET* dan akan langsung ditampilkan kembali ke *browser* melalui respon PHP. Absennya mekanisme validasi pada *form* ini memungkinkan penyisipan skrip berbahaya.

Gambar 2. Kode PHP yang rentan

```
<?php
if (isset($_GET['nama'])) {
    $nama = $_GET['nama'];
    echo "<p>Halo, $nama!</p>";
}
?>
```

Gambar 2 menunjukkan kode *backend* pada aplikasi uji. Data dari parameter nama ditampilkan langsung menggunakan *echo* tanpa proses *encoding*. Kondisi ini memungkinkan *browser* mengeksekusi *input* sebagai bagian dari DOM HTML, sehingga membuka celah terhadap *Reflected XSS*.

Gambar 3. Payload dan Output yang dihasilkan



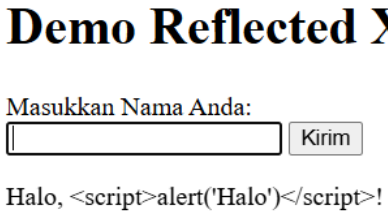
Gambar 3 memperlihatkan payload *"<script>alert('HACKED')</script>"* yang dimasukkan ke dalam *form input*. Setelah form dikirimkan, browser langsung mengeksekusi skrip, menampilkan jendela *alert*. Hal ini membuktikan bahwa aplikasi rentan terhadap serangan karena input tidak melalui proses penyandian seperti *htmlspecialchars()*.

Gambar 4. Kode PHP untuk mencegah Reflected XSS

```
<?php
if (isset($_GET['nama'])) {
    $nama = htmlspecialchars($_GET['nama'], ENT_QUOTES, 'UTF-8');
    echo "<p>Halo, $nama!</p>";
}
?>
```

Fungsi *htmlspecialchars()* berperan dalam melakukan proses *encoding* terhadap karakter-karakter khusus yang memiliki makna sintaksis dalam HTML, seperti <, >, tanda kutip ganda ("), dan tanda kutip tunggal ('). Proses ini mengubah karakter tersebut menjadi entitas HTML yang aman, sehingga *input* yang dimasukkan oleh pengguna akan ditampilkan sebagai teks biasa di halaman *web* tanpa ditafsirkan sebagai elemen atau perintah oleh *browser*. Dengan demikian, potensi eksekusi skrip berbahaya dapat dicegah secara efektif.

Gambar 5. Output dengan pengamanan specialchars()



Gambar 5 menunjukkan bahwa input berbahaya yang disisipkan oleh pengguna, berupa *tag "<script>"*, tidak dieksekusi oleh *browser*, melainkan hanya ditampilkan sebagai teks biasa di halaman. Hal ini menandakan bahwa upaya penyandian *output* berhasil dilakukan, sehingga karakter-karakter yang berpotensi membentuk skrip aktif telah dinonaktifkan sebelum sampai ke *browser*.

Dengan mekanisme ini, *browser* tidak lagi menafsirkan *input* sebagai instruksi atau perintah *JavaScript*, tetapi sebagai bagian dari konten visual yang tidak aktif. Hasilnya, tidak terjadi *pop-up* atau aksi dari skrip apa pun, meskipun input pengguna mengandung kode yang secara sintaksis valid.

Tampilan seperti ini merupakan indikasi bahwa aplikasi telah berhasil menerapkan mekanisme perlindungan terhadap *Reflected XSS*, dan pengguna tidak dapat menyisipkan skrip berbahaya ke dalam alur tampilan halaman. Keberhasilan ini memperlihatkan pentingnya pengamanan *output* sebagai langkah preventif terhadap kerentanan sisi klien.

B. *Stored XSS*

Pada skenario ini, input skrip dikirim melalui *form* komentar menggunakan metode *POST* dan disimpan ke dalam *database*. Data tersebut kemudian ditampilkan kembali di halaman tanpa proses penyandian HTML, sehingga skrip berbahaya dieksekusi ketika halaman diakses ulang.

Gambar 6. Form komentar pada web browser

```
<h2>Form Komentar</h2>
<form method="post">
    <textarea name="komentar" rows="4" cols="50" placeholder="Tulis komentar..."></textarea><br>
    <input type="submit" value="Kirim">
</form>
```

Gambar 6 menunjukkan antarmuka form komentar yang digunakan untuk menyisipkan *input*. Pengguna dapat mengirim teks komentar melalui metode *POST* yang akan diproses oleh server.

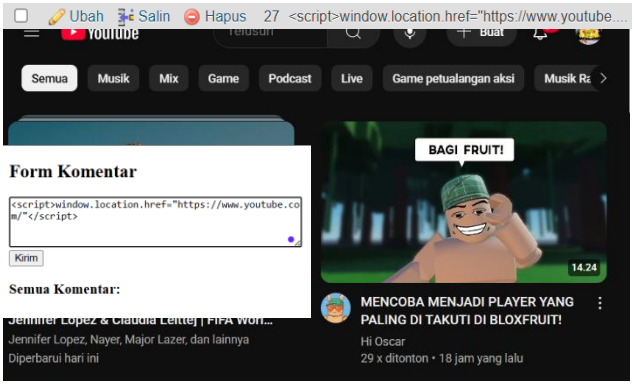
Gambar 7. Kode penyimpanan dan penampilan komentar

```
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    $komen = $conn->real_escape_string($_POST['komentar']);
    $conn->query("INSERT INTO komentar (isi) VALUES ('$komen')");
    header("Location: " . $_SERVER['PHP_SELF']);
    exit();
}

$result = $conn->query("SELECT isi FROM komentar");
?>
```

Potongan kode tersebut memperlihatkan bahwa *input* dari *form* komentar disimpan ke dalam *database* menggunakan *real_escape_string()*, yang hanya mencegah injeksi SQL, namun tidak cukup untuk mencegah XSS. Kemudian, komentar ditampilkan kembali ke halaman menggunakan *echo* tanpa proses penyandian seperti *htmlspecialchars()*. Karena itu, input berbahaya seperti *tag "<script>"* akan dianggap sebagai bagian dari DOM HTML dan langsung dieksekusi oleh *browser* saat halaman dimuat

Gambar 8. Payload dan hasil eksekusi Stored XSS dengan redirect.



Ketika dikirim, skrip akan tersimpan kedalam *database* dan langsung dijalankan secara otomatis karena tidak melalui proses penyandian.

Gambar 9. Kode pencegahan Stored XSS menggunakan htmlspecialchars() dan CSP

```
    <meta charset="UTF-8">
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'none'">
</head>
<body>
    <h2>Form Komentar</h2>
    <form method="post">
        <textarea name="komentar" rows="4" cols="50" placeholder="Tulis komentar..."></textarea><br>
        <input type="submit" value="Kirim">
    </form>
    <h3>Semua Komentar:</h3>
    <?php while ($row = $result->fetch_assoc()): ?>
        <p><?= htmlspecialchars($row['isi'], ENT_QUOTES, 'UTF-8') ?></p>
    <?php endwhile; ?>
```

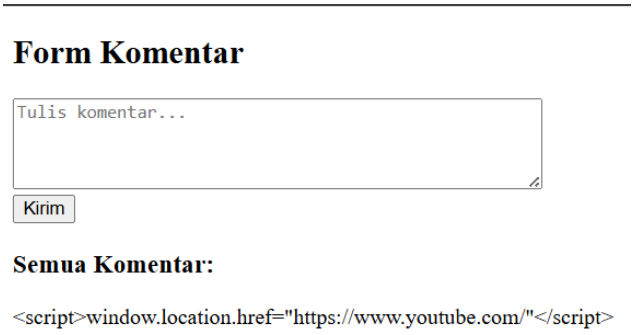
Gambar 9 menampilkan bagian kode yang bertanggung jawab untuk menampilkan komentar pengguna dari *database* ke halaman *web*. Pencegahan terhadap serangan *Stored XSS* dilakukan melalui dua mekanisme utama:

1. *htmlspecialchars():*
   Fungsi ini mengonversi karakter-karakter spesial HTML seperti <, >, ", dan ' menjadi entitas HTML yang aman. Dengan demikian, jika ada skrip berbahaya (misalnya *"<script>"*), maka *browser* tidak akan menafsirkannya sebagai kode, melainkan sebagai teks biasa.

2. *Content-Security-Policy (CSP)*
   CSP ini secara eksplisit melarang eksekusi skrip *JavaScript* di halaman, termasuk yang berasal dari *input* pengguna. Dengan kebijakan ini, bahkan jika ada skrip yang tidak sengaja lolos ke *output*, *browser* tidak akan mengeksekusinya.

Implementasi *htmlspecialchars()* pada sisi *output* dan CSP pada level *header* merupakan praktik terbaik dalam mencegah *Stored XSS*. Kombinasi ini tidak hanya menyaring karakter berbahaya, tetapi juga memperkuat perlindungan melalui aturan *browser*. Pendekatan ini sejalan dengan prinsip keamanan OWASP dan NIST untuk aplikasi *web* yang aman.

Gambar 10. Output dengan pengamanan htmlspecialchars() dan CSP



Gambar 10 memperlihatkan bagaimana *input* berbahaya yang disisipkan melalui kolom komentar, berupa *tag "<script>"* dengan instruksi pengalihan halaman (*redirect*) menggunakan *"window.location.href"*, tidak dieksekusi oleh *browser*, melainkan ditampilkan dalam bentuk teks utuh. Kondisi ini menunjukkan bahwa sistem telah berhasil menerapkan mekanisme pencegahan terhadap serangan *Stored XSS*.

Keberhasilan pencegahan ini disebabkan oleh penerapan fungsi *htmlspecialchars()* pada sisi *output*. Fungsi ini melakukan proses *encoding* terhadap karakter-karakter khusus HTML seperti <, >, dan tanda kutip, sehingga *input* yang sebelumnya berpotensi membentuk struktur skrip akan dikonversi menjadi representasi teks biasa. Akibatnya, *browser* tidak lagi menafsirkan input sebagai elemen atau instruksi aktif, melainkan hanya sebagai bagian dari konten visual halaman.

Selain itu, halaman juga dilengkapi dengan kebijakan *Content-Security-Policy* (CSP) melalui *header* meta, yang membatasi sumber daya skrip hanya dari asal yang dipercaya (misalnya *'self'*). Dengan konfigurasi CSP seperti ini,

meskipun terdapat celah yang belum tertangani secara sempurna, *browser* tetap akan menolak eksekusi skrip yang tidak diizinkan berdasarkan kebijakan yang telah ditentukan.

Hasil pada gambar ini menjadi bukti bahwa kombinasi antara penyandian *output* menggunakan *htmlspecialchars()* dan penerapan kebijakan CSP membentuk lapisan pertahanan yang efektif dalam mencegah eksekusi skrip berbahaya. Pendekatan ini sejalan dengan prinsip-prinsip pengembangan perangkat lunak yang aman dan direkomendasikan dalam pedoman OWASP untuk mitigasi serangan injeksi berbasis klien seperti Cross-Site Scripting.

C. *DOM-Based XSS*

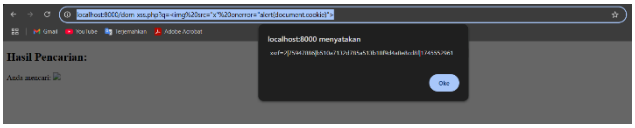
Pada skenario ini, input berbahaya dimasukkan melalui parameter URL dan diproses langsung oleh *JavaScript* di sisi klien menggunakan *innerHTML* tanpa proses penyandian. Karena tidak ada keterlibatan server, skrip berbahaya dapat dijalankan langsung di *browser* korban.

Gambar 11. Kode JavaScript rentang terhadap DOM-based XSS

```
<script>
  const q = new URLSearchParams(window.location.search).get("q");
  if (q) {
    document.getElementById("hasil").innerHTML += q;
  }
</script>
```

Gambar 11 menunjukkan bahwa parameter "q" dari URL dimasukkan ke dalam elemen HTML menggunakan *innerHTML*. Karena *innerHTML* memproses nilai sebagai HTML, input yang berisi *tag* skrip dapat dieksekusi secara langsung oleh *browser*. Ini membuka celah terhadap serangan *DOM-based XSS*.

Gambar 12. Payload DOM XSS dan hasil eksekusi di browser



Gambar 12 menunjukkan hasil keberhasilan eksploitasi *DOM-based XSS*, di mana skrip berbahaya disisipkan melalui parameter URL dan langsung dieksekusi di sisi klien. *Payload* menggunakan elemen gambar dengan atribut *onerror* untuk memicu fungsi *alert()* yang menampilkan isi *cookie* pengguna. Karena *browser* gagal memuat gambar palsu, *onerror* dijalankan dan menghasilkan jendela pop-up yang memperlihatkan nilai *document.cookie*.

Dalam kasus ini, cookie yang ditampilkan mencakup *token* sesi *(_xsrf)*, yang merupakan informasi sensitif yang dapat disalahgunakan oleh pihak yang tidak bertanggung jawab, misalnya untuk pembajakan sesi. Hal ini terjadi karena *input* pengguna diproses menggunakan *innerHTML* tanpa validasi atau *encoding*, sehingga browser memperlakukan *input* sebagai bagian dari struktur HTML aktif.

Peristiwa ini membuktikan bahwa *DOM-based XSS* dapat memungkinkan eksekusi *JavaScript* jahat tanpa keterlibatan server, serta menunjukkan risiko signifikan dari penggunaan manipulasi DOM yang tidak aman. Ini menegaskan pentingnya pengamanan *input* pengguna di sisi klien, termasuk dengan mengganti *innerHTML* menjadi *textContent* agar data hanya ditampilkan sebagai teks biasa, bukan sebagai skrip atau elemen aktif.

Gambar 13. Pencegahan DOM-based XSS menggunakan textContent dan Content-Security-Policy

```
<head>
  <title>DOM XSS Demo</title>
  <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">
</head>
<body>
  <h2>Hasil Pencarian:</h2>
  <div id="hasil">Anda mencari: <span id="search-term"></span></div>
  <script>
    function escapeHTML(unsafeText) {
      const div = document.createElement('div');
      div.textContent = unsafeText;
      return div.innerHTML;
    }
    const q = new URLSearchParams(window.location.search).get("q");
    if (q) {
      document.getElementById("search-term").textContent = q;
    }
  </script>
```

Gambar 13 menampilkan penerapan mekanisme pencegahan *DOM-based XSS* yang efektif dengan menggabungkan dua pendekatan utama: *encoding* sisi klien melalui penggunaan *textContent* dan penguatan kebijakan keamanan melalui *Content-Security-Policy* (CSP).

Penggunaan *textContent* memastikan bahwa *input* dari parameter URL "q" tidak akan ditafsirkan sebagai elemen HTML atau skrip, melainkan hanya akan ditampilkan sebagai teks murni. Hal ini mencegah *browser* dari memproses *tag* berbahaya seperti "*<script>*" atau "*<img onerror=...>*" yang biasa digunakan dalam serangan XSS.

Selain itu, penerapan CSP melalui *tag* meta membatasi pemuatan sumber daya hanya dari *domain* yang sama dan mencegah eksekusi skrip yang berasal dari luar. Dalam konfigurasi ini, bahkan jika *input* berbahaya berhasil disisipkan ke dalam halaman, kebijakan CSP akan mencegah eksekusi skrip yang tidak diizinkan.

Kedua lapisan pertahanan ini membentuk pendekatan berlapis (*defense-in-depth*) dalam mencegah serangan XSS berbasis DOM, menjadikan aplikasi lebih tahan terhadap eksploitasi sisi klien. Implementasi ini mencerminkan praktik terbaik pengamanan *input* dalam pengembangan aplikasi *web* modern.

Gambar 14. Output dengan pengamanan textContent dan Content-Security-Policy

Gambar 14 menunjukkan hasil ketika pengguna mencoba menyisipkan *payload DOM-based XSS* berupa elemen "*<img>*" dengan atribut *onerror* yang berisi perintah

JavaScript untuk menampilkan *alert*. Meskipun *input* berbahaya telah dimasukkan melalui parameter URL, halaman tidak mengeksekusinya. Sebaliknya, *browser* memunculkan pesan peringatan di konsol pengembang (*developer console*).

Pesan konsol tersebut secara eksplisit menyatakan bahwa skrip ditolak untuk dijalankan karena melanggar kebijakan *Content-Security-Policy* yang ditetapkan pada halaman. CSP yang digunakan memiliki direktif *script-src 'self'*, yang artinya hanya skrip yang berasal dari sumber internal (domain yang sama) yang diizinkan untuk dijalankan. Kebijakan ini secara efektif memblokir eksekusi skrip *inline*, termasuk skrip yang dimasukkan oleh pengguna melalui parameter URL.

Dengan CSP seperti ini, *browser* secara otomatis akan menolak segala bentuk injeksi skrip, kecuali jika skrip tersebut disertai dengan *hash*, *nonce*, atau berasal dari sumber yang eksplisit diizinkan. Oleh karena itu, sekalipun input pengguna lolos ke DOM, *browser* tidak akan mengeksekusinya sebagai *JavaScript* aktif.

Kondisi ini menunjukkan bahwa meskipun mungkin masih terdapat titik injeksi dalam struktur HTML, lapisan pertahanan tambahan melalui CSP mampu mencegah dampak eksploitasi. Gambar ini menjadi bukti bahwa kebijakan keamanan sisi klien dapat berfungsi sebagai mekanisme mitigasi efektif terhadap serangan *DOM-based XSS*, terutama dalam skenario di mana validasi *input* tidak sepenuhnya memadai.

Dengan demikian, penerapan CSP bukan hanya melindungi dari serangan skrip aktif, tetapi juga memberikan transparansi dan umpan balik langsung kepada pengembang melalui konsol, yang membantu dalam proses *debugging* dan penguatan keamanan aplikasi.

IV. KESIMPULAN

Berdasarkan hasil analisis yang telah dilakukan terhadap kerentanan XSS pada aplikasi web berbasis PHP menggunakan alat OWASP ZAP dan uji coba script XSS, diperoleh temuan bahwa aplikasi rentan terhadap serangan XSS tipe Reflected. Penelitian ini berhasil mengidentifikasi beberapa celah keamanan yang memungkinkan penyerang menyisipkan script berbahaya ke dalam input form aplikasi. Selain itu, penelitian ini menunjukkan bahwa penerapan metode pencegahan seperti validasi input menggunakan htmlspecialchars() dan penggunaan HTTP header seperti X-*XSS-Protection* secara signifikan mampu mengurangi potensi eksploitasi XSS.

Dari percobaan yang dilakukan, ditemukan bahwa serangan XSS dapat dicegah dengan pendekatan kombinatif antara penyaringan input, enkripsi karakter khusus, serta pemanfaatan *header* keamanan. Dengan demikian, pengembangan kerangka pencegahan yang komprehensif dapat meningkatkan keamanan aplikasi *web* dari serangan XSS secara efektif.

ACKNOWLEDGEMENT

Penulis menyampaikan penghargaan dan terima kasih kepada Bapak Joko Christian selaku dosen pembimbing atas bimbingan, masukan, dan arahan yang sangat berharga selama proses penelitian dan penyusunan karya ilmiah ini. Penulis juga mengucapkan terima kasih kepada Universitas Multimedia Nusantara yang telah menyediakan fasilitas dan lingkungan akademik yang mendukung kelancaran penelitian. Apresiasi juga disampaikan kepada rekan-rekan satu tim atas kolaborasi dan kerja sama yang baik, serta kepada keluarga yang senantiasa memberikan dukungan moral selama proses ini berlangsung.

REFERENCES

[1] S. J. Y. Weamie, “Cross-Site Scripting Attacks and Defensive Techniques: A Comprehensive Survey,” International Journal of Communications, Network and System Sciences, vol. 15, no. 8, pp. 126–148, 2022. doi:10.4236/ijcns.2022.158010

[2] OWASP Foundation, “OWASP Top 10 – 2021: A03 – Injection,” [Online]. Available: https://owasp.org/Top10/I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[3] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 4th ed., SAGE Publications, 2014.

[4] S. Mohanty dan A. A. Acharya, "Detection of XSS Vulnerabilities of Web Application Using Security Testing Approaches," dalam *Intelligent and Cloud Computing*, Springer, 2021, hlm. 267–275. [Online]. Tersedia: https://www.researchgate.net/publication/343969555

[5] J. Zhao, K. Zhu, L. Yu, H. Huang, dan Y. Lu, "Yama: Precise Opcode-based Data Flow Analysis for Detecting PHP Applications Vulnerabilities," *arXiv preprint*, arXiv:2410.12351, 2024. [Online]. Tersedia: https://arxiv.org/abs/2410.12351

[6] R. S. Pressman dan B. R. Maxim, *Software Engineering: A Practitioner’s Approach*, 8th ed., McGraw-Hill, 2014.

[7] Li, C., Wang, Y., Miao, C., & Huang, C. (2020). Cross-Site Scripting Guardian: A Static XSS Detector Based on Data Stream Input-Output Association Mining. *Applied Sciences*, 10(14), 4740. https://doi.org/10.3390/app10144740

[8] M. Mohammadi, B. Chu, and H. R. Lipford, “Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing,” *arXiv preprint arXiv:1804.00755*, 2018.

[9] Y. Jovanovic, N. Kruegel, dan E. Kirda, “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities,” *IEEE Symposium on Security and Privacy*, 2006.

[10] E. Cole, *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*, Syngress, 2012.

[11] A. W. Marashdih, Z. F. Zaaba, and H. K. Omer, “Web Security: Detection of Cross Site Scripting in PHP Web Application using Genetic Algorithm,” *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 8, no. 5, 2017. https://thesai.org/Publications/ViewPaper?Code=IJACSA&Issue=5&SerialNo=9&Volume=8