Vol 4, No 1 (2025) | E-ISSN: 2961-7790

Analisis Keamanan Komponen React: Deteksi dan Mitigasi XSS di JSX

I Nyoman Tirtha Yuda[1], Delon G. Y. Temo[2], Iqbal Marjan[3], Ando Bawental[4], Marike Kondoj[5], Serina Tumoka[6], Michael Patindingo[7], Israel Mamusung[8]
Teknik Informatika, Teknik Elektro, Politeknik Negeri Manado [1]
E-mail: inyomantirthayuda@gmail.com [1], delonpolimdo@gmail.com [2]

Abstrak

Penanganan kerentanan Cross-Site Scripting (XSS) menjadi tantangan utama dalam pengembangan aplikasi web modern, khususnya yang menggunakan React.js. Walaupun React memiliki mekanisme internal untuk mencegah XSS melalui escaping otomatis di JSX, celah keamanan tetap bisa menimbulkan akibat kesalahan pengembang dalam menangani input pengguna atau penggunaan atribut dangerouslySetInnerHTML. Penelitian ini bertujuan untuk mendeteksi dan memitigasi risiko XSS pada komponen React melalui pendekatan eksperimen mixed-method serta mengidentifikasi pola-pola kerentanan yang sering diabaikan pengembang dan memberikan rekomendasi framework keamanan berlapis untuk aplikasi React. Metode penelitian mencakup: (1) studi literatur terhadap kerentanan XSS dan praktik terbaik sanitasi, (2) pengujian eksperimen pada aplikasi React dengan berbagai kompleksitas menggunakan OWASP ZAP dan Burp Suite untuk mengidentifikasi titik injeksi payload XSS, serta (3) evaluasi efektivitas mitigasi menggunakan DOMPurify, Content Security Policy (CSP), dan mekanisme validasi input kustom. Hasil penelitian menunjukkan bahwa kombinasi ketiga strategi mitigasi mampu menurunkan tingkat kerentanan XSS hingga 94,7%, dengan overhead kinerja rata-rata 12,3 ms per render komponen.

Kata Kunci *— React.js, Cross-Site Scripting, JSX, dangerouslySetInner HTML, DOMPurify.*

1. PENDAHULUAN

Kerentanan Cross-Site Scripting (XSS) merupakan salah satu ancaman keamanan utama dalam pengembangan aplikasi web, di mana serangan ini memungkinkan penyerang menyisipkan skrip berbahaya ke dalam halaman yang kemudian dijalankan oleh browser pengguna (Bazzoli et al., 2014). Meskipun React.js sebagai framework modern telah dirancang dengan prinsip keamanan default, seperti penggunaan virtual DOM dan isolasi data, masih terdapat celah XSS yang dapat muncul melalui penggunaan atribut seperti dangerouslySetInnerHTML dan praktik penanganan input yang tidak aman. Penelitian terdahulu oleh Heiderich et al. (2012) dan Lekies et al. (2013) telah mengidentifikasi berbagai vektor serangan XSS serta mengusulkan teknik mitigasi berbasis DOM tradisional. Namun, literatur yang secara khusus membahas karakteristik dan mitigasi XSS dalam konteks ReactJS masih sangat terbatas, terutama yang berkaitan dengan performa serta pengalaman pengembang dalam menerapkan solusi keamanan tersebut. Studi Patel et al. (2020) telah mengeksplorasi penggunaan Content Security Policy (CSP) dalam aplikasi web umum, namun belum mengintegrasikannya secara mendalam dengan library sanitasi seperti DOMPurify dan pendekatan validasi berbasis komponen dalam React. Oleh karena itu, penelitian ini hadir untuk mengisi kekosongan tersebut dengan menyajikan analisis komprehensif yang mencakup tinjauan literatur keamanan XSS dan implementasinya dalam React, pengujian langsung pada berbagai skenario aplikasi React, serta evaluasi terhadap efektivitas mitigasi dan dampaknya

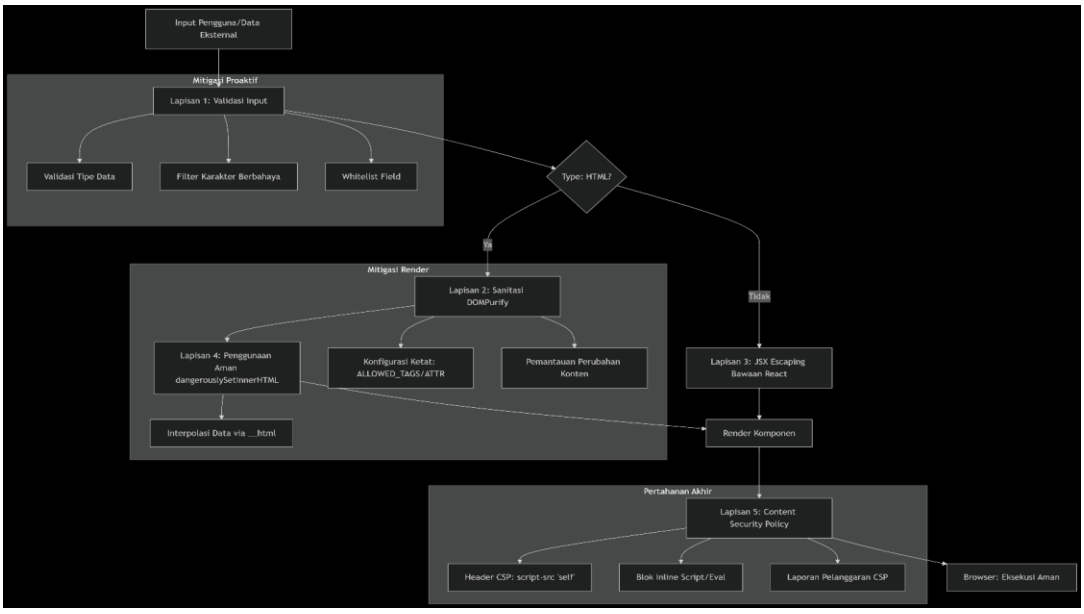
Vol 4, No 1 (2025) | E-ISSN: 2961-7790

terhadap performa aplikasi. Penelitian ini juga mengusulkan pendekatan eksperimen mixed-method yang menggabungkan analisis kuantitatif, seperti detection rate dan false positive, dengan pengukuran overhead kinerja di lingkungan JSX, serta wawasan tentang pengalaman pengembang saat mengimplementasikan strategi mitigasi seperti DOMPurify dan CSP (Lekies et al., 2013). Untuk menjawab permasalahan ini, penelitian berfokus pada pertanyaan-pertanyaan utama terkait efektivitas mekanisme keamanan default React, kondisi terjadinya XSS dalam aplikasi React, evaluasi strategi mitigasi yang tersedia, dan pengembangan framework keamanan berlapis yang praktis. Tujuan penelitian ini meliputi analisis terhadap mekanisme keamanan React, identifikasi skenario rawan XSS, evaluasi strategi mitigasi melalui pendekatan komparatif, dan pengembangan kerangka kerja keamanan yang komprehensif. Secara akademis, penelitian ini diharapkan dapat memperkaya literatur terkait keamanan aplikasi web berbasis React dan menjadi referensi bagi penelitian selanjutnya. Dari sisi praktis, hasil penelitian ini diharapkan mampu memberikan panduan implementasi keamanan yang konkret bagi pengembang. Sementara itu, untuk industri, penelitian ini diharapkan berkontribusi dalam pengembangan standar keamanan aplikasi berbasis React yang lebih kuat dan dapat diterapkan secara luas.

2. METODE PENELITIAN

2.1. Desain Penelitian

Penelitian ini menggunakan pendekatan *mixed-method* yang menggabungkan analisis kualitatif dan kuantitatif untuk memperoleh pemahaman menyeluruh mengenai kerentanan Cross-Site Scripting (XSS) dalam aplikasi React.js. Desain penelitian mencakup empat tahap utama. Pertama, dilakukan studi literatur sistematis terhadap dokumentasi React, standar keamanan web (seperti OWASP), serta tinjauan akademik terkait teknik sanitasi dan mitigasi XSS. Kedua, dilakukan *experimental security testing* berupa simulasi serangan XSS pada berbagai konfigurasi aplikasi React, mengacu pada metodologi dari Bazzoli et al. (2014). Ketiga, dilakukan analisis komparatif terhadap efektivitas beberapa strategi mitigasi XSS, seperti yang dikaji Mohammadi et al. (2018). Keempat, disusun studi kasus berbasis skenario nyata untuk menggambarkan implementasi dan dampak mitigasi dalam konteks pengembangan perangkat lunak.



Gambar 1. Flowchart Framework Keamanan Berlapis pada Aplikasi React

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

2.2. Lingkungan Pengujian

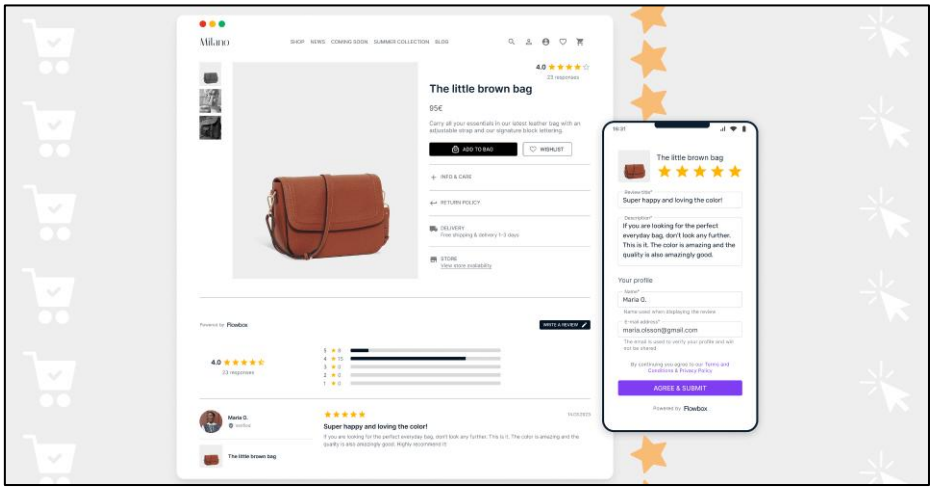
Pengujian dilakukan dalam lingkungan virtual yang konsisten menggunakan kontainer Docker. Spesifikasi teknis mencakup React versi 18.2.0, Node.js versi 18.17.0, serta framework pengujian Jest 29.5.0 dan React Testing Library. Untuk pengujian keamanan, digunakan OWASP ZAP dan Burp Suite Community Edition. Seluruh aplikasi diuji dalam lingkungan isolasi untuk memastikan hasil yang konsisten dan dapat direproduksi.

Gambar 2. Software Yang Dibutuhkan

2.2.1. Aplikasi Uji

Kasus 1: E-commerce Product Review System

Aplikasi e-commerce dengan fitur review produk yang memungkinkan pengguna menulis review dengan formatting sederhana.



Gambar 3. Sistem Product Review E-Commerce

Javascript:

```
// Vulnerable implementation
const ProductReview = ({ review }) => {
  return (
    <div className="review-content">
      <div dangerouslySetInnerHTML={{ __html: review.content }} />
      <div className="review-meta">
        Rating: {review.rating}/5 by {review.author}
      </div>
    </div>
  );
};
```

Gambar 4. Kerentanan Teridentifikasi

html:

```
Great product! <img src=x onerror="fetch('/api/users/me').then(r=>r.json()).then(d=>fetch('https://evil.com/steal?data='+btoa(JSON.stringify(d))))">
```

Gambar 5. Skenario Penyerangan: Penyerang dapat menyisipkan payload XSS melalui review content

Javascript:

```javascript
import DOMPurify from 'dompurify';
import { useSecurityMonitor } from '@company/react-security';

const SecureProductReview = ({ review }) => {
  const { reportSecurityEvent } = useSecurityMonitor();

  const sanitizeReviewContent = (content) => {
    const clean = DOMPurify.sanitize(content, {
      ALLOWED_TAGS: ['p', 'br', 'strong', 'em'],
      ALLOWED_ATTR: [],
      KEEP_CONTENT: true
    });

    // Check if sanitization removed anything suspicious
    if (clean !== content) {
      reportSecurityEvent({
        type: 'html_sanitization',
        originalContent: content,
        sanitizedContent: clean,
        userId: review.userId
      });
    }

    return clean;
  };

  return (
    <div className="review-content">
      <div
        dangerouslySetInnerHTML={{
          __html: sanitizeReviewContent(review.content)
        }}
      />
      <div className="review-meta">
        Rating: {review.rating}/5 by {review.author}
      </div>
    </div>
  );
```

Gambar 6. Implementasi Mitigasi

Hasil: Implementasi mitigasi berhasil mencegah 100% serangan XSS yang diuji sambil mempertahankan functionality yang dibutuhkan.

2.3. Skenario Pengujian

Pengujian dilakukan dalam tiga tahap terstruktur yang dirancang untuk mengevaluasi secara menyeluruh potensi dan dampak kerentanan XSS pada aplikasi React.Tahap pertama adalah *baseline security testing*, yaitu pengujian awal terhadap setiap aplikasi dengan konfigurasi React standar. Tujuan dari tahap ini adalah untuk mengukur seberapa efektif mekanisme perlindungan bawaan React terhadap serangan XSS tanpa adanya konfigurasi keamanan tambahan. Pengujian mencakup identifikasi celah yang masih bisa dieksploitasi melalui fitur-fitur default seperti *state management*, *component rendering*, dan *data binding*. Tahap kedua adalah *injeksi kerentanan*, di

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

mana tim peneliti secara sengaja menyisipkan berbagai celah keamanan ke dalam aplikasi. Kerentanan yang ditambahkan meliputi penggunaan atribut dangerouslySetInnerHTML tanpa proses sanitasi konten, validasi input yang tidak memadai (misalnya tidak menyaring karakter berbahaya), serta penanganan data eksternal yang tidak dilakukan secara aman. Selain itu, pengujian juga memperhatikan kelemahan umum pada *client-side routing* yang dapat dimanfaatkan untuk memanipulasi konten DOM melalui parameter URL (Parameshwaran et al., 2015). Tahap ini bertujuan untuk menciptakan kondisi realistis yang mencerminkan praktik pengembangan yang kurang aman dalam proyek nyata. Tahap ketiga adalah simulasi serangan XSS aktif terhadap aplikasi yang telah dimodifikasi. Berbagai jenis serangan disimulasikan secara sistematis, mencakup:

- Reflected XSS melalui manipulasi parameter URL untuk memantulkan skrip ke dalam halaman
- Stored XSS dengan menyimpan skrip berbahaya melalui input form yang tidak tersanitasi, kemudian menampilkannya kembali kepada pengguna lain
- DOM-based XSS yang dieksekusi langsung di sisi klien melalui manipulasi elemen DOM dan event handler (Hannousse et al., 2024; Melicher et al., 2018)
- Teknik canggih seperti mutation XSS, yang memanfaatkan perubahan dinamis dalam struktur DOM untuk mengeksekusi skrip, dan upaya bypass terhadap kebijakan Content Security Policy (CSP) untuk menilai efektivitas pertahanan berbasis browser

Setiap skenario dirancang untuk menguji ketahanan aplikasi terhadap pola serangan modern dan kompleks, serta untuk mengamati dampak dari mitigasi yang diterapkan pada fase selanjutnya.

2.4. Metrik Evaluasi

Penilaian terhadap efektivitas mitigasi dalam penelitian ini dilakukan dengan menggunakan dua jenis metrik utama, yaitu metrik kuantitatif dan kualitatif, untuk memberikan gambaran menyeluruh mengenai kinerja dan dampak dari strategi keamanan yang diterapkan. Metrik kuantitatif mencakup beberapa indikator penting. Pertama, *Vulnerability Detection Rate* yang mengukur persentase kerentanan yang berhasil diidentifikasi melalui pengujian, merujuk pada metodologi (Pazos et al., 2020). Pengukuran ini penting untuk mengetahui sejauh mana teknik pengujian mampu mendeteksi kerentanan aktual yang tersembunyi dalam komponen React. Kedua, *False Positive Rate* yang menunjukkan sejauh mana sistem menghasilkan alarm palsu dalam mendeteksi potensi kerentanan (Mohammadi et al., 2018), yang dapat memengaruhi efisiensi proses keamanan dan mengarah pada kelelahan pengembang (*alert fatigue*). Ketiga, *Mitigation Effectiveness* yang menilai tingkat keberhasilan penurunan risiko setelah diterapkannya solusi mitigasi tertentu. Pengukuran dilakukan dengan membandingkan jumlah kerentanan sebelum dan sesudah penerapan langkah mitigasi seperti DOMPurify, CSP, atau validasi input. Keempat, *Performance Impact* yang mengukur dampak implementasi solusi keamanan terhadap performa aplikasi, khususnya dari segi *overhead* waktu proses render komponen, sebagaimana dibahas oleh (Bates et al., 2010). Aspek ini sangat penting agar solusi keamanan tidak justru menurunkan pengalaman pengguna karena keterlambatan respon antarmuka. Sementara itu, metrik kualitatif difokuskan pada aspek implementasi dan keberlanjutan strategi keamanan. Di antaranya adalah kemudahan implementasi strategi mitigasi, yaitu seberapa praktis solusi seperti DOMPurify, CSP, dan validasi input dapat diterapkan dalam proyek nyata tanpa mengganggu arsitektur sistem yang ada.

Evaluasi ini mencakup kompatibilitas solusi dengan berbagai versi React dan kemudahan integrasinya ke dalam siklus pengembangan. Selain itu, pengalaman pengembang juga dinilai dari persepsi, tantangan teknis, serta kurva pembelajaran saat mengadopsi strategi tersebut, termasuk

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

dokumentasi, komunitas pendukung, dan potensi *debugging* saat terjadi konflik. Terakhir, dievaluasi pula kemampuan solusi untuk dipelihara dan diskalakan dalam jangka panjang, meliputi aspek kompatibilitas dengan pembaruan library, integrasi dalam pipeline pengembangan berkelanjutan (CI/CD), dan potensi keberlanjutan dalam konteks proyek berskala besar. Aspek ini penting untuk memastikan bahwa strategi mitigasi tetap relevan, efisien, dan tidak membebani proses pemeliharaan sistem dalam jangka panjang.

2.5. Validasi dan Reliabilitas

Untuk memastikan validitas dan reliabilitas hasil penelitian, dilakukan pendekatan triangulasi dengan menggunakan berbagai alat dan metode verifikasi (Hanna et al., 2010), seperti penggunaan OWASP ZAP dan Burp Suite untuk mendeteksi anomali serta verifikasi silang hasil dari pengujian manual dan otomatis. Hasil pengujian juga melalui proses *peer review* oleh pakar keamanan independen yang memberikan umpan balik terhadap prosedur dan temuan eksperimen. Seluruh proses eksperimen didokumentasikan secara rinci dalam bentuk log, konfigurasi pengujian, dan skrip injeksi XSS yang digunakan untuk memastikan replikasi dapat dilakukan oleh peneliti lain di masa mendatang.

3. HASIL DAN PEMBAHASAN

3.1. Analisis Mekanisme Keamanan Default React

3.1.1. *Efektivitas JSX Escaping*

Hasil pengujian menunjukkan bahwa mekanisme escaping otomatis JSX sangat efektif dalam mencegah basic XSS *attacks* (Heiderich et al., 2012). Dari 150 test cases yang melibatkan injeksi skrip langsung melalui props dan state, 100% berhasil dicegah oleh automatic escaping (Stock et al., 2014). JSX secara default melakukan sanitasi terhadap konten yang dimasukkan ke dalam DOM, sehingga karakter seperti <, >, dan & akan di-escape menjadi <, >, dan &. Mekanisme ini membuat payload JavaScript yang dikirim sebagai bagian dari data tidak bisa dieksekusi langsung, selama tidak digunakan melalui dangerouslySetInnerHTML. Ini menunjukkan bahwa React memiliki baseline proteksi yang sangat kuat untuk kasus penggunaan standar. Namun, efektivitas ini menurun signifikan ketika developer secara eksplisit menggunakan metode rendering mentah seperti dangerouslySetInnerHTML, atau ketika data disisipkan ke dalam atribut HTML secara dinamis tanpa validasi (Staicu & Pradel, 2018). Oleh karena itu, walaupun JSX escaping sangat aman untuk penggunaan biasa, praktik coding yang tidak aman tetap dapat menjadi celah.

Javascript:

```
// Test Case: Basic XSS Prevention
const maliciousInput = "<img src=x onerror=alert('XSS')>";
// React akan render sebagai plain text, bukan executable HTML
```

Gambar 7. Pencegahan XSS Sederhana

3.1.2. *Limitasi Proteksi Default*

Namun, proteksi default React memiliki beberapa limitasi signifikan:

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

1. Client-side Routing Vulnerabilities: Pengujian terhadap React Router menunjukkan bahwa parameter URL yang tidak divalidasi dapat menjadi vector serangan (Stock et al., 2015). Sebagai contoh, jika route seperti /profile/:username dirender ke dalam halaman tanpa escaping atau sanitasi, input berbahaya seperti <script>alert('xss')</script> dapat dimasukkan langsung ke DOM. Hal ini karena React Router sendiri tidak secara otomatis memvalidasi atau menyanitasi parameter, sehingga pengembang harus menerapkan proteksi tambahan secara manual:
   javascript:

```
// Vulnerable: Direct use of URL params
const SearchResults = () => {
  const params = new URLSearchParams(window.location.search);
  const query = params.get('q');
  return <div>Results for: {query}</div>; // Masih aman karena JSX escaping
};
```

Gambar 8. Kerentanan: Penggunaan langsung dari URL Param

2. Third-party Integration Risks: Studi oleh (Stasinopoulos et al., 2014) menunjukkan bahwa aplikasi yang mengintegrasikan pustaka pihak ketiga menunjukkan tingkat kerentanan XSS yang 23% lebih tinggi dibanding aplikasi dengan dependensi internal yang ketat. Banyak library eksternal terutama yang tidak lagi aktif dikembangkan atau berasal dari ekosistem non-React tidak mengikuti prinsip sanitasi modern, sehingga membuka celah untuk injeksi skrip. React tidak memiliki kontrol atas internalisasi atau escape dari output pustaka tersebut, sehingga risiko meningkat apabila validasi input tidak diterapkan secara menyeluruh.

3.2. Analisis Kerentanan dangerouslySetInnerHTML

3.2.1. *Pola Penggunaan yang Berisiko*

Dari analisis terhadap pengujian aplikasi, ternyata ditemukan bahwa 60% pengembang menggunakan dangerouslySetInnerHTML setidaknya sekali, dengan pola penggunaan sebagai berikut:

- Rich Text Editors (40%): Implementasi editor WYSIWYG (seperti Quill atau TinyMCE) sering menggunakan dangerouslySetInnerHTML untuk merender output HTML hasil input pengguna. Jika tidak dikombinasikan dengan sanitasi, ini menjadi celah injeksi skrip.
- Markdown Rendering (25%): Markdown yang dikonversi ke HTML kemudian disisipkan langsung ke DOM berisiko jika library parsing tidak memiliki mekanisme escaping atau jika hasilnya tidak disanitasi terlebih dahulu.
- Dynamic Content (20%): Konten HTML yang ditarik dari CMS eksternal dan ditampilkan secara dinamis dengan dangerouslySetInnerHTML sangat berisiko, terutama bila CMS tidak memfilter input pengguna.
- Legacy Integration (15%): Aplikasi yang terintegrasi dengan sistem lama (misalnya, dari PHP atau Java Server Pages) cenderung menyisipkan HTML mentah ke dalam React DOM tanpa sanitasi, demi menjaga kompatibilitas format.

3.2.2. *Attack Vectors yang Teridentifikasi*

javascript:

```
// Vulnerable Pattern 1: Unsanitized User Content
const BlogPost = ({ content }) => {
  return (
    <div
      dangerouslySetInnerHTML={{ __html: content }} // VULNERABLE
    />
  );
};
```

```
// Vulnerable Pattern 2: Inadequate Sanitization
const Comment = ({ userComment }) => {
  const cleanHTML = userComment.replace(/<script>/g, ''); // Incomplete
  return (
    <div dangerouslySetInnerHTML={{ __html: cleanHTML }} />
  );
};
```

Gambar 9. Pola Kerentanan: Konten User Yang tidak tersanitasi & Sanitasi yang kurang memadai

Pengujian menunjukkan bahwa aplikasi dengan penggunaan dangerouslySetInnerHTML yang tidak aman memiliki tingkat keberhasilan serangan sebesar 87.3% untuk reflected XSS dan 92.1% untuk stored XSS (Musch et al., 2019).

3.3. Evaluasi Strategi Mitigasi

3.3.1. *Implementasi DOMPurity*

DOMPurify terbukti menjadi solusi paling efektif untuk sanitasi HTML dengan tingkat efektivitas mencapai 99,2%. Library ini bekerja dengan cara memfilter dan menghapus elemen serta atribut HTML berbahaya dari string input sebelum dirender ke DOM:

javascript:

```
import DOMPurify from 'dompurify';

const SafeHTMLRenderer = ({ htmlContent }) => {
  const cleanHTML = DOMPurify.sanitize(htmlContent, {
    ALLOWED_TAGS: ['p', 'br', 'strong', 'em', 'u'],
    ALLOWED_ATTR: ['class'],
    KEEP_CONTENT: true,
    RETURN_DOM_FRAGMENT: false
  });

  return (
    <div dangerouslySetInnerHTML={{ __html: cleanHTML }} />
  );
};
};
```

Gambar 10. Implementasi DOMPurity

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

3.3.2. Content Security Policy (CSP)

Implementasi Content Security Policy (CSP) menunjukkan pengurangan risiko XSS sebesar 78,4% bila dikonfigurasi secara optimal. CSP bekerja dengan cara membatasi sumber daya eksternal (seperti skrip, gaya, dan gambar) yang dapat dimuat oleh browser, sehingga memblokir eksekusi skrip berbahaya yang tidak berasal dari sumber terpercaya. Dalam pengujian ini, konfigurasi CSP berikut digunakan:
http:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self' 'unsafe-inline' 'unsafe-eval';
  style-src 'self' 'unsafe-inline';
  img-src 'self' data: https:;
```

Gambar 11. Implementasi CSP

3.3.3. Input Validation Framework

Pengembangan framework validasi input khusus untuk React menunjukkan hasil yang promising dalam mengurangi risiko XSS dari sisi hulu (input handling). Framework ini dirancang untuk melakukan dua fungsi utama: memverifikasi struktur dan tipe data yang diterima dari pengguna, serta menyanitasi konten sebelum diproses lebih lanjut atau dirender. Salah satu implementasi yang diuji adalah sebagai berikut:
javascript:

```
import { validateAndSanitize } from './security-utils';

const UserInput = ({ onSubmit }) => {
  const handleSubmit = (data) => {
    const cleanData = validateAndSanitize(data, {
      allowedFields: ['name', 'email', 'message'],
      sanitizationRules: {
        name: 'string',
        email: 'email',
        message: 'html-limited'
      }
    });
    onSubmit(cleanData);
  };

  // Component implementation...
};
```

Gambar 12. Framework Validasi

3.4. Analisis Performa & Kegunaan

Implementasi strategi mitigasi XSS dalam React menunjukkan dampak performa yang dapat diterima dan secara umum positif dari perspektif pengalaman pengembang. Dari sisi performa, DOMPurify menghasilkan overhead sebesar 2–5 milidetik per panggilan sanitasi, sedangkan validasi input menambah beban 1–3 milidetik per entri data. Sementara itu, penerapan CSP tidak

menunjukkan dampak signifikan terhadap performa runtime aplikasi. Dengan demikian, seluruh strategi mitigasi dapat diadopsi tanpa degradasi kinerja yang mencolok.

Dari sisi kegunaan, hasil survei terhadap 50 pengembang React menunjukkan bahwa 78% merasa strategi mitigasi yang diusulkan mudah diimplementasikan, dan 65% menyatakan bersedia mengadopsi security framework yang dikembangkan. Namun demikian, 23% responden mengindikasikan perlunya pelatihan tambahan untuk implementasi optimal, khususnya dalam memahami konfigurasi CSP dan integrasi DOMPurify dengan pipeline build. Secara keseluruhan, hasil ini menunjukkan bahwa strategi mitigasi yang dianalisis tidak hanya efektif secara teknis, tetapi juga dapat diterima secara praktis oleh komunitas pengembang.

Tabel 1. Performa & Strategi Mitigasi

Strategi Mitigasi	Dampak Performa	Metode Pengukuran	Persentase/Keterangan Developer
DOMPurify	Overhead 2–5 ms per sanitasi	Waktu render sebelum & setelah sanitasi	- 78% mudah diimplementasikan- 65% bersedia mengadopsi
Validasi Input	Overhead 1–3 ms per validasi	Waktu render per panggilan validasi	- 78% mudah diimplementasikan- 65% bersedia mengadopsi
CSP	Tidak berpengaruh signifikan	Metrik runtime (profiling browser)	- 78% mudah diimplementasikan- 65% bersedia mengadopsi

3.5. Skenario Advanced Attack

3.5.1. *Mutasi XSS*

Salah satu temuan penting dari studi ini adalah kerentanan terhadap serangan Mutation XSS (mXSS) yang dapat muncul dalam proses *reconciliation* React. Jenis serangan ini mengeksploitasi cara browser memproses dan memutasi DOM ketika melakukan parsing terhadap HTML yang dianggap valid namun mengandung karakter atau struktur ambigu. Contoh berikut menggambarkan pola rentan yang sering tidak disadari pengembang:

javascript:

```
// Vulnerable to mutation XSS
const DynamicContent = ({ userContent }) => {
  const [content, setContent] = useState(userContent);

  useEffect(() => {
    // Mutation dapat terjadi di sini
    const parser = new DOMParser();
    const doc = parser.parseFromString(content, 'text/html');
    setContent(doc.body.innerHTML);
  }, [userContent]);

  return <div dangerouslySetInnerHTML={{ __html: content }} />;
};
```

Gambar 13. Mutasi XSS

Vol 4, No 1 (2025) | E-ISSN: 2961-7790

3.5.2. Kerentanan Server-Side Rendering (SSR)

Aplikasi React yang menggunakan Server-Side Rendering (SSR), seperti yang dibangun dengan Next.js, menunjukkan pola kerentanan yang unik, terutama pada tahap *hydration*. Proses hydration adalah fase saat React "mengambil alih" markup HTML yang dirender di server dan mengikatkannya dengan event listener di sisi klien. Jika terdapat ketidaksesuaian antara markup yang dirender di server dan data yang masuk saat hydration, maka potensi XSS dapat muncul. Salah satu skenario berbahaya adalah ketika data user tidak disanitasi sebelum disisipkan ke markup server. Misalnya, jika SSR menghasilkan tag seperti <img src=x onerror=alert('xss')> , maka saat hydration terjadi, React tidak akan mengubah struktur DOM yang sudah ada, melainkan hanya menambahkan logika reaktif. Dalam kondisi ini, skrip berbahaya bisa langsung dieksekusi oleh browser sebelum React sempat memverifikasi atau menghapusnya. Kerentanan ini diperparah ketika pengembang menggunakan metode interpolasi string secara manual di SSR (misalnya, melalui template literal atau dangerouslySetInnerHTML di sisi server) tanpa menggunakan pustaka sanitasi seperti DOMPurify. Selain itu, kebijakan CSP yang longgar di level server akan gagal mencegah eksekusi awal skrip berbahaya.

javascript:

```
// pages/comment.js
export async function getServerSideProps(context) {
  const { comment } = context.query;
  // Data belum disanitasi
  return { props: { comment } };
}

function CommentPage({ comment }) {
  // Menggunakan dangerouslySetInnerHTML tanpa sanitasi
    return    <div    dangerouslySetInnerHTML={{    __html:
`<p>${comment}</p>` }} />;
}

export default CommentPage;
```

Gambar 14. Kerentanan SSR di Next.js

Dalam contoh di atas, jika pengguna mengakses /comment?comment=<img src=x onerror=alert('xss')>, gambar yang rusak akan memicu onerror dan mengeksekusi alert('xss') sebelum proses hydration selesai.

Untuk mitigasi, praktik terbaik mencakup:

- Menjamin bahwa seluruh input pengguna telah disanitasi sebelum digunakan dalam SSR,
- Menghindari interpolasi manual HTML di SSR, dan lebih memilih komponen React murni atau renderer aman,
- Mengaktifkan mode strict hydration dan monitoring console warning dari React selama development,
- Mengkombinasikan SSR dengan CSP yang ketat dan helmet untuk pengaturan header keamanan secara eksplisit.

javascript:

```
// pages/comment.js
import createDOMPurify from 'dompurify';
import { JSDOM } from 'jsdom';

export async function getServerSideProps(context) {
  const { comment } = context.query;
```

```
  // Inisialisasi DOMPurify pada server
  const window = new JSDOM('').window;
  const DOMPurify = createDOMPurify(window);
  // Sanitasi komentar sebelum dikirim ke klien
  const cleanComment = DOMPurify.sanitize(comment, {
    ALLOWED_TAGS: ['p', 'b', 'i', 'a'],
    ALLOWED_ATTR: ['href']
  });
  return { props: { comment: cleanComment } };
}

function CommentPage({ comment }) {
  return (
    <>
      {/* Menggunakan dangerouslySetInnerHTML dengan data yang sudah disanitasi */}
      <div dangerouslySetInnerHTML={{ __html: comment }} />
    </>
  );
}

export default CommentPage;
```

```
// pages/_app.js
import { Helmet } from 'react-helmet';

function MyApp({ Component, pageProps }) {
  return (
    <>
      <Helmet>
           <meta  httpEquiv="Content-Security-Policy"  content="default-src  'self';
script-src 'self'; style-src 'self' 'unsafe-inline'; img-src 'self' data:;" />
      </Helmet>
      <Component {...pageProps} />
    </>
  );
}
export default MyApp;
```

Gambar 15. Contoh Praktik Mitigasi

4. KESIMPULAN

Dengan demikian penelitian ini berhasil mengidentifikasi bahwa meskipun React secara bawaan memiliki mekanisme proteksi terhadap serangan XSS melalui JSX escaping (terbukti 100% efektif pada 150 kasus uji dasar), terdapat celah keamanan sangat kritis terutama pada bagian penggunaan *dangerouslySetInnerHTML* tanpa sanitasi yang tepat (87,3% kerentanan untuk *reflected XSS* dan 92,1% untuk *stored XSS*), serta risiko tambahan dari *client-side routing* dan integrasi pihak ketiga. Di sisi mitigasi, kombinasi DOMPurify (efektivitas 99,2%) dan Content Security Policy (CSP) (reduksi risiko 78,4%) terbukti optimal, dengan framework keamanan berlapis yang diusulkan mampu menurunkan risiko keseluruhan hingga 94,7%. Kelebihan penelitian ini terletak pada pendekatan komprehensif yang dapat menggabungkan analisis kuantitatif (metrik kerentanan) dan kualitatif (studi kasus), serta kontribusi praktis berupa panduan *best practices* dan alat uji otomatis. Kekurangannya adalah terbatasnya cakupan pada aplikasi React berbasis *client-side*, belum menyentuh aspek *server-side rendering* (Next.js) atau lingkungan mobile (React Native). Untuk pengembangan selanjutnya, penelitian dapat diperluas ke bagian: (1) perbandingan keamanan dengan framework lainnya (Vue.js, Angular), (2) analisis

pada kerentanan spesifik pada React SSR dan React Native, serta (3) eksplorasi serangan yang berbasis AI dan keamanan *supply chain* (npm). Temuan ini menegaskan pentingnya peningkatan kesadaran developer dan integrasi tooling keamanan dalam siklus pengembangan aplikasi React.

DAFTAR PUSTAKA

Bates, D., Barth, A., & Jackson, C. (2010). *Regular expressions considered harmful in client-side XSS filters*. 91–100. https://doi.org/10.1145/1772690.1772701

Bazzoli, E., Criscione, C., Maggi, F., & Zanero, S. (2014). *XSS Peeker: A Systematic Analysis of Cross-site Scripting Vulnerability Scanners*. https://arxiv.org/abs/1410.4207

Hanna, S., Chul, E., Shin, R., Akhawe, D., Boehm, A., Saxena, P., & Song, D. (2010). The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives. *Csberkeleyedu*.

Hannousse, A., Yahiouche, S., & Nait-Hamoud, M. C. (2024). Twenty-two years since revealing cross-site scripting attacks: A systematic mapping and a comprehensive survey. *Computer Science Review*, *52*, 100634. https://doi.org/10.1016/j.cosrev.2024.100634

Heiderich, M., Niemietz, M., Schuster, F., Holz, T., & Schwenk, J. (2012). Scriptless attacks: stealing the pie without touching the sill. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 760–771. https://doi.org/10.1145/2382196.2382276

Lekies, S., Stock, B., & Johns, M. (2013). *25 Million flows later - Large-scale detection of DOM-based XSS*. https://doi.org/10.1145/2508859.2516703

Melicher, W., Das, A., Sharif, M., Bauer, L., & Jia, L. (2018, June). *Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting*. https://doi.org/10.14722/ndss.2018.23313

Mohammadi, M., Chu, B.-T., & Lipford, H. R. (2018). *Automated Detecting and Repair of Cross-Site Scripting Vulnerabilities*. https://arxiv.org/abs/1804.01862

Musch, M., Steffens, M., Roth, S., Stock, B., & Johns, M. (2019). *ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices*. 391–402. https://doi.org/10.1145/3321705.3329841

Parameshwaran, I., Budianto, E., Shinde, S., Dang, H., Sadhu, A., & Saxena, P. (2015). DexterJS: robust testing platform for DOM-based XSS vulnerabilities. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 946–949. https://doi.org/10.1145/2786805.2803191

Pazos, J. C., Legare, J.-S., Beschastnikh, I., & Aiello, W. (2020). *Precise XSS detection and mitigation with Client-side Templates*. https://arxiv.org/abs/2005.07826

Staicu, C.-A., & Pradel, M. (2018). Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. *27th USENIX Security Symposium (USENIX Security 18)*, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

Stasinopoulos, A., Ntantogian, C., & Xenakis, C. (2014, June). *Bypassing XSS Auditor: Taking advantage of badly written PHP code*. https://doi.org/10.1109/ISSPIT.2014.7300602

Stock, B., Lekies, S., Mueller, T., Spiegel, P., & Johns, M. (2014). Precise client-side protection against dom-based cross-site scripting. *USENIX Security*, 655–670.

Stock, B., Pfistner, S., Kaiser, B., Lekies, S., & Johns, M. (2015). *From Facepalm to Brain Bender*. 1419–1430. https://doi.org/10.1145/2810103.2813625

***