Journal of Applied Informatics and Computing (JAIC)
Vol.x, No.x, Juli 201x, pp. xx~xx
e-ISSN: 2548-6861

Analisis Efektivitas Content Security Policy Dalam Memblokir Serangan Cross-Site Scripting Pada Aplikasi Web Berbasis PHP

Billton Camry [1]*, Antoni Haikal [2]**
* Rekayasa Keamanan Siber, Politeknik Negeri Batam
** Rekayasa Keamanan Siber, Politeknik Negeri Batam
billton.4332211004@students.polibatam.ac.id [1], antoni@polibatam.ac.id [2]

Article Info

Article history:

Received …
Revised …
Accepted …

Keyword:

Content Security Policy, Cross-Site Scripting, Strict CSP

ABSTRACT

Serangan Cross-Site Scripting (XSS) masih menjadi ancaman utama pada aplikasi web berbasis PHP, sementara penerapan Content Security Policy (CSP) sering kali tidak optimal karena pengembang cenderung menggunakan konfigurasi longgar demi menjaga fungsionalitas. Penelitian ini bertujuan menganalisis efektivitas berbagai konfigurasi CSP dalam memblokir serangan XSS, mengukur overhead performa yang ditimbulkan, serta menilai dampak kompatibilitas terhadap fungsionalitas aplikasi. Penelitian menggunakan metode eksperimen komparatif pada aplikasi web DVWA dengan empat skenario konfigurasi: baseline, Non-Strict CSP (Permissive), Non-Strict CSP, dan Strict-CSP dengan nonce dan strict-dynamic.

Setiap konfigurasi diuji pada tiga jenis serangan XSS (Reflected, Stored, DOM) menggunakan 26 payload dari SecLists yang diinjeksikan secara otomatis melalui Selenium WebDriver. Pengukuran overhead performa meliputi latensi server, ukuran paket jaringan, serta penggunaan sumber daya sisi klien. Pengujian kompatibilitas dilakukan dengan mengobservasi fungsionalitas fitur-fitur aplikasi (seperti sistem tema dan dropdown bahasa) untuk mengidentifikasi adanya kerusakan akibat kebijakan keamanan yang ketat.

Hasil penelitian menunjukkan bahwa Non-Strict CSP dan Strict-CSP berhasil memblokir 100% payload XSS, sedangkan Non-Strict CSP (Permissive) gagal total. Performa overhead CSP secara umum rendah dan tidak berdampak signifikan terhadap pengalaman pengguna. Dari segi kompatibilitas, penerapan Strict-CSP tanpa modifikasi kode dapat menyebabkan kerusakan fitur pada skala minor hingga mayor. Namun, kendala ini dapat diatasi sepenuhnya melalui refaktorisasi kode dengan menyisipkan atribut nonce pada tag skrip yang sah. Penelitian ini menyimpulkan bahwa Strict-CSP dengan nonce dan strict-dynamic merupakan konfigurasi paling optimal karena memberikan keamanan maksimal tanpa mengorbankan fungsionalitas aplikasi setelah dilakukan penyesuaian kode.

I. PENDAHULUAN

Aplikasi web berbasis PHP masih banyak digunakan dalam berbagai sektor [1], sehingga aspek keamanan menjadi perhatian penting. Salah satu ancaman utama yang sering menyerang aplikasi web adalah Cross-Site Scripting (XSS) [2], yaitu serangan yang menyisipkan skrip berbahaya ke dalam halaman web. Dampak serangan ini dapat berupa pencurian cookie sesi, pengambilalihan akun, manipulasi tampilan halaman, hingga penyebaran malware [3].

Permasalahan utama dalam mitigasi XSS adalah perlindungan yang hanya mengandalkan sanitasi dan validasi input belum selalu cukup,terutama karena payload XSS terus berkembang. Content Security Policy (CSP) dapat

e-ISSN: 2548-6861

digunakan sebagai mekanisme keamanan tambahan di sisi browser untuk membatasi sumber daya yang boleh dimuat dan dieksekusi [4][5]. Namun, penerapan CSP masih sangat rendah[6], dan sering kali tidak optimal karena pengembang cenderung menggunakan konfigurasi longgar, seperti unsafe-inline, demi menjaga fungsionalitas aplikasi. Kondisi ini dapat menurunkan efektivitas CSP dalam memblokir serangan XSS.

Beberapa penelitian sebelumnya telah membahas pengujian XSS, konsep dasar CSP, dan efektivitas CSP pada aplikasi web. Namun, masih terdapat celah penelitian pada analisis komparatif berbagai tingkat konfigurasi CSP, mulai dari tanpa CSP, Non-Strict CSP, hingga Strict-CSP, terutama pada aplikasi web berbasis PHP. Selain itu, penelitian yang secara bersamaan mengukur efektivitas keamanan, overhead performa, dan dampak kompatibilitas aplikasi masih terbatas.

Penelitian ini bertujuan untuk menganalisis efektivitas beberapa konfigurasi CSP dalam memitigasi serangan Reflected XSS, Stored XSS, dan DOM-based XSS pada aplikasi web berbasis PHP. Selain itu, penelitian ini juga mengukur overhead performa yang ditimbulkan, mengevaluasi dampak CSP terhadap fungsionalitas aplikasi, serta merekomendasikan konfigurasi CSP yang paling optimal dari sisi keamanan, performa, dan kompatibilitas.

Hasil penelitian ini diharapkan dapat memberikan kontribusi teoritis berupa bukti empiris mengenai penerapan CSP dalam mitigasi XSS. Secara praktis, penelitian ini dapat menjadi panduan bagi pengembang dalam memilih dan menerapkan konfigurasi CSP yang aman, efisien, serta tetap kompatibel dengan fungsionalitas aplikasi web.

II. METODE

Tahap pengujian dalam penelitian ini dirancang secara komprehensif untuk mengevaluasi tiga parameter utama terhadap empat skema konfigurasi Content Security Policy (CSP), yakni efektivitas keamanan, overhead performa, dan kompatibilitas. Objek penelitian yang digunakan sebagai target implementasi dan pengujian adalah Damn Vulnerable Web App (DVWA), yaitu aplikasi web berbasis PHP dan MySQL yang dirancang secara khusus untuk kebutuhan pembelajaran, simulasi, dan pengujian keamanan aplikasi web. DVWA dipilih karena menyediakan modul kerentanan yang terstruktur, termasuk Reflected XSS, Stored XSS, dan DOM-based XSS, sehingga sesuai dengan ruang lingkup penelitian yang berfokus pada pengujian efektivitas Content Security Policy (CSP) terhadap serangan Cross-Site Scripting. Pemilihan DVWA juga didasarkan pada karakteristiknya sebagai aplikasi yang memang dirancang rentan. Pada kondisi dasar, DVWA tidak menerapkan mekanisme pertahanan keamanan yang kuat seperti sanitasi input yang ketat, output encoding menyeluruh, Web Application Firewall (WAF), atau mekanisme proteksi tambahan lain yang dapat memengaruhi hasil pengujian. Kondisi ini penting karena tujuan penelitian bukan untuk menilai gabungan berbagai mekanisme pertahanan, melainkan untuk mengukur kemampuan CSP secara lebih spesifik dalam memblokir eksekusi payload XSS. DVWA digunakan sebagai lingkungan eksperimental agar pengaruh CSP dapat dianalisis secara fokus, terukur, dan tidak tercampur dengan mekanisme pertahanan lain yang dapat mengganggu objektivitas penilaian. Untuk memastikan validitas hasil, eksperimen akan dieksekusi secara komparatif pada lingkungan ter virtualisasi menggunakan Kali Linux.

Penelitian ini mendefinisikan empat konfigurasi CSP yang akan dieksperimenkan. Penerapan CSP dilakukan dengan memodifikasi konfigurasi security level pada modul XSS DVWA agar setiap skenario menghasilkan response header CSP yang sesuai dengan skema pengujian. Konfigurasi CSP yang diuji disusun berdasarkan referensi praktik keamanan dari OWASP dan Google web.dev[7][8]. Referensi tersebut digunakan sebagai dasar dalam merancang skema Non-Strict CSP dan Strict-CSP, Rincian dari keempat konfigurasi Content Security Policy yang diterapkan pada aplikasi web pengujian dapat dilihat pada Tabel 1.

TABEL I
KONFIGURASI CSP YANG AKAN DIUJI

Skema Keamanan	Penjelasan	Sintaks
Baseline	Kondisi asli, aplikasi berjalan murni tanpa perlindungan	-
Non-Strict CSP (Permissive)	Konfigurasi longgar yang membatasai sumber eksternal, namun masih mengizinkan eksekusi skrip inline.	Content-Security-Policy: default-src 'none'; script-src 'self' 'unsafe-inline'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-ancestors 'self'; form-action 'self'; report-uri /DVWA/report-csp.php;
Non-Strict CSP	Konfigurasi standar yang hanya mengizinkan eksekusi skrip dari domain yang dipercaya	Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self'; frame-ancestors 'self'; form-action 'self'; report-uri /DVWA/report-csp.php;
Strict-CSP	Konfigurasi berbasis kriptografi yang menggunakan nilai nonce.	Content-Security-Policy: script-src 'nonce-{nonce}' 'strict-dynamic'; object-src 'none'; base-uri 'none'; report-uri /DVWA/report-csp.php;

A. Pengujian Efektivitas Keamanan

Tahap ini bertujuan untuk mengevaluasi tingkat keberhasilan masing masing skema konfigurasi CSP dalam

memitigasi dan memblokir berbagai skenario eksekusi payload XSS yang diinjeksikan pada aplikasi. Pengujian efektivitas ini dibagi ke dalam tiga skenario serangan XSS, yaitu:

1) *Skenario 1 (Reflected XSS)*: Skrip otomasi akan melakukan injeksi payload berbahaya secara langsung ke dalam form input aplikasi web yang telah disediakan.
2) *Skenario 2 (Stored XSS)*: Skrip otomasi menginjeksi payload ke dalam basis data aplikasi untuk mengevaluasi apakah CSP mampu memblokir eksekusi saat payload tersebut dipanggil dan direnderkembali pada halaman web.
3) *Skenario 3 (DOM-based XSS)*: Pengujian difokuskan pada manipulasi lingkungan DOM di sisi client untuk menguji respons CSP terhadap eksekusi skrip yang tidak bersumber dari respon HTTP server

Pengujian akan dilakukan dengan konfigurasi CSP pada Tabel 1. Adapun payload XSS yang digunakan dalam skenario pengujian bersumber dari repositori standar industri, yaitu SecLists. Payload XSS yang digunakan dalam penelitian ini tidak hanya terbatas pada payload dasar, tetapi mencakup variasi teknik eksekusi yang lebih modern dan beragam. Payload dipilih untuk mewakili beberapa pola serangan, seperti event handler injection, interaction-based payload, encoded/obfuscated payload, filter bypass payload, serta DOM-based payload. Dengan variasi tersebut, pengujian tidak hanya mengukur kemampuan CSP dalam memblokir payload sederhana seperti tag <script>, tetapi juga menguji efektivitas CSP terhadap pola eksekusi JavaScript yang lebih kompleks dan umum digunakan dalam pengujian keamanan aplikasi web modern.

Untuk memverifikasi status keberhasilan pengujian secara objektif, skrip selenium dikonfigurasi untuk memonitor dua indikator utama pada antarmuka client: kemunculan pop-up alert (sebagai bukti eksekusi payload) dan aktivitas pada Console Log browser (bukti intervensi keamanan).

TABEL II
KLASIFIKASI HASIL PENGUJIAN KEAMANAN

Status	Deskripsi
Berhasil dieksekusi	Apabila alert muncul pada layar browser
Berhasil diblokir	Apabila alert tidak muncul dan sistem menemukan catatan log pemblokiran eksekusi skrip oleh kebijakan CSP di dalam console log
Invalid	Apabila alert tidak muncul, namun pada console log juga tidak ditemukan catatan pemblokiran oleh CSP

Untuk memastikan hasil tidak bergantung pada satu jenis browser, seluruh skenario pengujian efektivitas keamanan diulang pada dua browser: Mozilla Firefox dan Google Chrome. Setiap browser dijalankan dengan konfigurasi default tanpa ekstensi tambahan. Pengujian pada Firefox juga memanfaatkan endpoint report-uri untuk menangkap CSP violation karena keterbatasan driver.get_log() pada GeckoDriver

B. Pengujian Performa

Tahap pengujian performa ini menggunakan pendekatan analitis end-to-end untuk mengevaluasi dampak komprehensif dari implementasi dan pemrosesan aturan Content Security Policy terhadap kinerja aplikasi web. Untuk mendapatkan kalkulasi overhead yang presisi, pengukuran dibagi menjadi tiga titik evaluasi utama:

1) Overhead Sisi Server: Tahap ini bertujuan untuk mengukur peningkatan beban komputasi server dalam memproses dan menginjeksi aturan CSP secara dinamis. Pengujian dilakuakn dengan menyimulasikan permintaan konkren berskala besar menggunakan instrumen load testing Apache Jmeter. Parameter evaluasi difokuskan pada Time to First Byte sebagai metrik utama untuk mengukur dampak komputasi dari implementasi CSP. Karena TTFB mencakup seluruh fase pemrosesan di sisi server sebelum transmisi data dimulai, metrik ini mampu mengisolasi dan membuktikan secara empiris beban yang dihasilkan oleh konfigurasi CSP terhadap server web.

2) Overhead Jaringan: Analisis ini bertujuan untuk mengevaluasi dampak penambahan aturan CSP terhadap peningkatan ukuran payload HTTP yang ditransmisikan dari server ke klien. Pengukuran dilakukan dengan mengkalkulasi total ukuran response header sebagai indikator overhead didasarkan pada standar RFC 7540, yang menyatakan bahwa pembengkakan ukuran header secara langsung berkontribusi pada peningkatan latensi jaringan dan konsumsi bandwidth. Metrik ini memungkinkan peneliti untuk mengukur efisiensi transmisi data. Pengukuran metrik dilakukan menggunakan Apache JMeter.

3) Overhead Sisi Client: Analisis ini bertujuan untuk mengevaluasi beban pemrosesan pada browser saat melakukan validasi terhadap aturan CSP sebelum melakukan rendering elemen halaman. Pengukuran dilakukan menggunakan instrumen Google Chrome DevTools dengan fokus pada metrik Main Thread Time dan System Time. Penggunaan metrik ini didasarkan pada arsitektur browser dimana seluruh proses parsing dan penegakan aturan keamanan CSP dilakukan pada Main Thread. Sementara itu, metrik System Time memrepresentasikan aktivitas internal mesin browser untuk proses validasi keamanan. Peningkatan pada Main Thread Time dan System Time secara langsung mempresentasikan overhead komputasi yang harus ditanggung oleh perangkat klien akibat kompleksitas aturan keamanan yang diterapkan.

C. Pengujian Kompatibilitas

Pengujian Kompatibilitas bertujuan untuk mengevaluasi dampak implementasi Content Security Policy (CSP) terhadap fungsionalitas normal aplikasi web. Tujuannya adalah memastikan bahwa kebijakan keamanan yang

e-ISSN: 2548-6861

diterapkan tidak memblokir fitur-fitur sah yang seharusnya berjalan, sehingga aplikasi tetap dapat digunakan secara wajar. Pengujian dilakukan dengan menjalankan skenario interaksi normal pengguna pada setiap konfigurasi CSP pada Tabel 1. Status kompatibilitas diklasifikan sebagai berikut:

TABEL III
KLASIFIKASI KOMPATIBILITAS

Status	Deskripsi
Normal	Semua fitur berfungsi sesuai spesifikasi; tidak ditemukan pemblokiran elemen oleh CSP.
Rusak Minor	Terjadi gangguan pada fitur non-essensial atau aspek kosmetik; alur kerja utama pengguna tetap dapat digunakan.
Rusak Mayor	Fitur essensial atau interaksi pengguna utama terganggu; menghambat tujuan penggunaan aplikasi secara signifikan.
Tidak Berfungsi	Terjadi kegagalan sistem secara total; aplikasi tidak dapat diakses atau digunakan sama sekali.

Penentuan status kompatibilitas pada penelitian ini dilakukan dengan menilai tingkat keparahan dampak terhadap aplikasi. Sistem klasifikasi ini diadaptasi dari metodologi pengujian perangkat lunak oleh Software Testing Help (2022) dalam artikel "Defect Severity And Priority In Testing"[9].

III. HASIL DAN PEMBAHASAN

Pengujian dilakukan untuk efektivitas Content Security Policy (CSP) dalam memitigasi serangan Cross-Site Scripting (XSS), mengukur kinerja overhead yang ditimbulkan, serta menilai dampaknya terhadap kompatibilitas fitur aplikasi. Pengujian dilakukan pada tiga skenario serangan, yaitu Reflected XSS, Stored XSS, dan DOM-based XSS. Setiap skenario diuji menggunakan empat konfigurasi, yaitu baseline tanpa CSP, Non-Strict CSP (Permissive), Non-Strict CSP, dan Strict-CSP berbasis nonce serta strict-dynamic.

A. Hasil Pengujian Efektivitas Keamanan

Pengujian efektivitas keamanan dilakukan pada dua browser, yaitu Mozilla Firefox 140.9.0esr dan Chromium 146.0.7680.153. Penggunaan dua browser bertujuan untuk memastikan bahwa hasil pengujian tidak bergantung pada satu mesin browser saja, karena setiap browser dapat memiliki perbedaan dalam proses parsing HTML, rendering, dan penegakan kebijakan CSP [10].

1) *Hasil Pengujian Keamanan Skenario 1*: Pengujian dilakukan dengan menyuntikkan payload berbahaya melalui formulir input. Hasil pengujian dapat dilihat pada Tabel 2 dan Tabel 3 berikut:

TABEL IV
HASIL PENGUJIAN KEAMANAN SKENARIO 1 PADA CHROMIUM

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	26	0	0%
Non-Strict CSP (Permissive)	26	26	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Berdasarkan hasil pengujian pada Chromium, Penerapan CSP Dasar dan Strict-CSP terbukti 100% efektif memblokir payload.

TABEL V
HASIL PENGUJIAN KEAMANAN SKENARIO 1 PADA FIREFOX

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	14	0	0%
Non-Strict CSP (Permissive)	26	14	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Serupa dengan hasil pada Chromium, pengujian pada Firefox juga menunjukan bahwa penerapan Non-Strict CSP dan Strict-CSP memberikan tingkat mitigasi sebesar 100%. Namun, terdapat perbedaan signifikan pada kondisi baseline dan Non-strict CSP (permissive) di mana hanya 14 dari 26 payload yang berhasil dieksekusi. Perbedaan jumlah eksekusi pada kondisi baseline antara Chromium dan Firefox disebabkan oleh perbedaan rendering engine dalam menangani parsing HTML [10]. Namun efektivitas CSP, tetap konsisten pada angka 100% (26/26) karena mekanisme CSP bekerja pada level policy enforcement yang memvalidasi setiap upaya eksekusi skrip sebelum mesin browser memproses konten tersebut lebih jauh.

2) *Hasil Pengujian Keamanan Skenario 2*: Pengujian dilakukan dengan menyuntikkan payload berbahaya melalui formulir input. Payload pada skenario ini disimpan secara permanen di dalam database server. Hal ini membuat serangan menjadi lebih masif , karena payload akan tereksekusi secara otomatis pada setiap pengguna yang memuat halaman tersebut.

TABEL VI
HASIL PENGUJIAN KEAMANAN SKENARIO 2 PADA CHROMIUM

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	26	0	0%
Non-Strict CSP (Permissive)	26	26	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Berdasarkan hasil pengujian pada Chromium, Penerapan CSP Dasar dan Strict-CSP terbukti 100% efektif memblokir payload Stored XSS.

TABEL VII
HASIL PENGUJIAN KEAMANAN SKENARIO 2 PADA FIREFOX

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	26	0	0%
Non-Strict CSP (Permissive)	26	26	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Beralih pada pengujian menggunakan Firefox (Tabel 5), hasil yang diperoleh menunjukan konsistensi yang serupa

dengan pengujian pada Chromium. Berbeda dengan skenario Reflected XSS di mana Firefox memiliki tingkat eksekusi baseline yang lebih rendah, pada skenario Stored XSS, seluruh payload berhasil dieksekusi sepenuhnya pada kondisi baseline dan Non-Strict CSP (Permissive). Meskipun demikian, penerapan kebijakan Non-Strict CSP dan Strict CSP terbukti tetap konsisten dalam memberikan perlindungan maksimal dengan tingkat mitigasi sebesar 100%. Seluruh upaya eksekusi payload berbahaya berhasil dihentikan oleh kebijakan CSP.

3) *Hasil Pengujian Keamanan Skenario 3*: Pengujian pada kerentanan DOM-based XSS memiliki karakteristik yang berbeda secara fundamental dibandingkan Reflected maupun Stored XSS. Pada Serangan ini, payload tidak pernah dieksekusi oleh server PHP, melainkan dieksekusi murni di sisi klien. Skrip javascript bawaan aplikasi secara tidak aman mengambil input dari URL dan menulisnya langsung ke dalam struktur HTML menggunakan fungsi document.write().

TABEL VIII
HASIL PENGUJIAN KEAMANAN SKENARION 3 PADA CHROMIUM

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	26	0	0%
Non-Strict CSP (Permissive)	26	26	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Berdasarkan hasil pengujian pada Chromium, Penerapan Non-Strict CSP dan Strict-CSP terbukti 100% efektif memblokir payload DOM-based XSS.

TABEL IX
HASIL PENGUJIAN KEAMANAN SKENARIO 3 PADA FIREFOX

Skenario Pengujian	Total Payload	Dieksekusi	Diblokir	Tingkat Mitigasi
Baseline	26	13	0	0%
Non-Strict CSP (Permissive)	26	13	0	0%
Non-Strict CSP	26	0	26	100%
Strict-CSP	26	0	26	100%

Serupa dengan hasil pada Chromium, pengujian pada Firefox juga menunjukan bahwa penerapan Non-Strict CSP dan Strict-CSP memberikan tingkat mitigasi sebesar 100%. Namun, terdapat perbedaan signifikan pada kondisi baseline dan Non-strict CSP (permissive) di mana hanya 13 dari 26 payload yang berhasil dieksekusi. Perbedaan jumlah eksekusi pada kondisi baseline antara Chromium dan Firefox disebabkan oleh perbedaan rendering engine dalam menangani parsing HTML.

Setiap browser memiliki perbedaan dalam mekanisme parsing HTML. Perbedaan ini menyebabkan variasi dalam cara browser memproses input HTML yang tidak valid, yang dalam penelitian tersebut dimanfaatkan untuk melakukan fingerprinting berbasis XSS. Oleh karena itu, dapat disimpulkan bahwa payload yang sama berpotensi menghasilkan perilaku yang berbeda pada setiap browser [10]. Namun efektivitas CSP, tetap konsisten pada angka 100% (26/26) karena mekanisme CSP bekerja pada level policy enforcement yang memvalidasi setiap upaya eksekusi skrip sebelum mesin browser memproses konten tersebut lebih jauh.

B. Hasil Pengujian Overhead Performa

Keberhasilan implementasi Content Security Policy tidak hanya diukur dari kemampuannya memitigasi serangan Cross-Site Scripting (XSS), melainkan juga dari seberapa efisien aturan tersebut diproses oleh sistem. Sub-bab ini menguraikan hasil analisis overhead end-to-end yang mengkalkulasi selisih kinerja aplikasi web antara kondisi murni tanpa perlindungan dengan kondisi saat berbagai level konfigurasi CSP diaktifkan. Pengukuran difokuskan pada pencatatan Time to First Byte (TTFB), Response Time, dan ukuran payload HTTP.

1) *Hasil Pengujian Overhead Sisi Server*: Pengujian latensi dan response time dilakukan untuk mengukur beban tambahan yang muncul pada proses pengiriman data dari server ke klien akibat penerapan berbagai konfigurasi CSP. Metrik utama yang dianalisis yaitu rata-rata latensi hasil pengujian menggunakan JMeter yang dapat dilihat sebagai berikut:

TABEL X
HASIL PENGUJIAN OVERHEAD SISI SERVER

Konfigurasi	Rata-Rata Latensi	*Overhead* Latensi
Baseline	3.561 ms	-
Non-Strict CSP (Permissive)	3.701 ms	+3.93%
Non-Strict CSP	4.409 ms	+23.8%
Strict-CSP	4.025 ms	+13.03%

Berdasarkan hasil pengujian, penerapan CSP menimbulkan overhead latensi yang bervariasi namun masih berada dalam skala milidetik. Konfigurasi Non-Strict CSP mencatat overhead tertinggi sebesar 23,8% dibandingkan baseline, sedangkan Non-Strict CSP (Permissive) hanya meningkat sebesar 3,93%. Sementara itu, Strict-CSP menghasilkan overhead sebesar 13,03%, lebih rendah dibandingkan Non-Strict CSP meskipun menerapkan mekanisme keamanan berbasis nonce dan `strict-dynamic`. Hasil ini menunjukkan bahwa penerapan CSP, khususnya Strict-CSP, tetap relatif efisien dari sisi waktu respons dan tidak memberikan dampak performa yang signifikan pada metrik latensi aplikasi.

2) *Hasil Pengujian Overhead Jaringan*: Pengujian ini bertujuan untuk menganalisis peningkatan ukuran paket data HTTP yang ditransmisikan dari server ke client akibat penambahan header CSP yang kompleks. Pengukuran metrik dilakukan dengan mengkalkulasi total ukuran response header menggunakan instrumen load testing Apache JMeter untuk membandingkan besaran data yang dikirimkan pada setiap skenario.

e-ISSN: 2548-6861

TABEL XI
HASIL PENGUJIAN OVERHEAD JARINGAN

Konfigurasi	Rata-Rata Latensi	*Overhead* Latensi
Baseline	3.561 ms	-
Non-Strict CSP (Permissive)	3.701 ms	+3.93%
Non-Strict CSP	4.409 ms	+23.8%
Strict-CSP	4.025 ms	+13.03%

Penambahan header CSP menyebabkan peningkatan ukuran data yang ditransmisikan melalui jaringan. Overhead jaringan tertinggi terjadi pada konfigurasi Strict-CSP sebesar 5,49% dibandingkan baseline. Peningkatan ini disebabkan oleh penggunaan direktif strict-dynamic dan nilai nonce dinamis pada setiap respons HTTP, sehingga ukuran header menjadi lebih besar. Sementara itu, Non-Strict CSP (Permissive) menghasilkan overhead sebesar 4,63%, sedikit lebih tinggi dibandingkan Non-Strict CSP sebesar 4,17%. Perbedaan ini menunjukkan bahwa variasi sintaks pada setiap konfigurasi CSP turut memengaruhi ukuran header yang dikirimkan.

3) *Hasil Pengujian Overhead Sisi Client*: Pengujian overhead sisi client dilakukan untuk mengukur beban pemrosesan tambahan yang terjadi pada browser akibat implementasi kebijakan Content Security Policy (CSP). Pengukuran difokuskan pada dua metrik utama dari Chrome DevTools Performance Tab, yaitu Main Thread Time (total waktu aktivitas thread utama) dan System Time (waktu yang dihabiskan untuk tugas internal browser, termasuk validasi kebijakan CSP). Setiap skenario konfigurasi CSP diulang sebanyak 5 kali dengan metode record dan reload yang konsisten.

TABEL XII
HASIL PENGUJIAN OVERHEAD MAIN THREAD TIME

Konfigurasi	Rata-Rata Main Thread Time	*Overhead*
Baseline	91.08 ms	-
Non-Strict CSP (Permissive)	97.36 ms	+6.89%
Non-Strict CSP	95.36 ms	+4.69%
Strict-CSP	116.06 ms	+27.42%

Main Thread adalah mencerminkan aktivitas browser melakukan tugas tugas berat seperti mengeksekusi javascript. Dari hasil pengujian, Non-Strict CSP (Permissive) dan Non-Strict CSP mengalami kenaikan yang relatif kecil (dibawah 7%). Hal ini mengindikasikan bahwa metode pemeriksaan daftar putih tidak memberikan beban komputasi yang berat pada alur kerja. Terjadi lonjakan beban yang signifikan pada Strict-CSP sebesar 27.42%. Angka ini menunjukkan bahwa adanya aktivitas pemrosesan ekstra yang dilakukan browser.

TABEL XIII
HASIL PENGUJIAN OVERHead System Time

Konfigurasi	**Rata-Rata System Time**	***Overhead***
Baseline	49.6 ms	-
Non-Strict CSP (Permissive)	54 ms	+8.87%
Non-Strict CSP	57 ms	+14.91%
Strict-CSP	72.8 ms	+46.77%

Pada metrik System Time, Strict-CSP mencatatkan overhead tertinggi hingga 46.77%, yang menunjukan bahwa kebijakan keamanan berbasis kriptografi bekerja jauh lebih intensif. Lonjakan drastis pada Strict-CSP merupakan konsekuensi penggunaan nonce dan strict-dynamic. Browser melakukan validasi kriptografis untuk memastikan nilai nonce pada setiap elemen skrip sesuai dengan nilai yang dikirimkan melalui header HTTP. Mekanisme strict-dynamic juga memaksa browser untuk memvalidasi rantai trust propagation setiap kali sebuah skrip terpecaya membuat skrip baru.

C. Hasil Pengujian Kompatibilitas

Pengujian kompatibilitas bertujuan untuk mengevaluasi dampak implementasi Content Security Policy terhadap fungsionalitas normal aplikasi web DVWA. Tujuannya adalah memastikan bahwa kebijakan keamanan yang diterapkan tidak memblokir fitur-fitur sah yang seharusnya berjalan, sehingga aplikasi tetap dapat digunakan secara wajar. Pengujian dilakukan pada setiap skenario serangan XSS (Reflected, Stored, dan DOM) karena masing-masing halaman memiliki fitur inline script yang berbeda.

1) *Hasil Pengujian Kompatibilitas pada Reflected XSS*: Pada pengujian kompatibilitas halaman Reflected XSS, baseline berjalan normal karena tidak terdapat header CSP yang membatasi eksekusi JavaScript. Konfigurasi Non-Strict CSP (Permissive) juga tetap normal karena masih menggunakan `unsafe-inline`, sehingga inline script dan event handler seperti `onclick` tetap dapat dijalankan. Sebaliknya, Non-Strict CSP menyebabkan kerusakan minor pada fitur pengubah tema karena direktif `script-src 'self'` tanpa `unsafe-inline` memblokir JavaScript inline yang digunakan oleh fitur tersebut. Kerusakan ini dikategorikan minor karena tidak mengganggu fungsi utama halaman. Pada Strict-CSP, fungsionalitas kembali normal setelah dilakukan refaktorisasi dengan mengganti `onclick` menjadi `addEventListener` dan menambahkan nonce valid pada skrip yang sah.

2) *Hasil Pengujian Kompatibilitas pada Stored XSS*: Pada pengujian kompatibilitas halaman Stored XSS, baseline berjalan normal karena tidak terdapat header CSP yang membatasi eksekusi JavaScript. Konfigurasi Non-Strict CSP (Permissive) juga tetap normal karena masih menggunakan `unsafe-inline`, sehingga inline script dan event handler seperti `onclick` tetap dapat dijalankan pada fitur pengubah tema, validasi form guestbook, dan konfirmasi hapus guestbook.

Sebaliknya, Non-Strict CSP menyebabkan status rusak minor karena direktif `script-src 'self'` tanpa `unsafe-inline` memblokir JavaScript inline yang digunakan oleh fitur-fitur tersebut. Akibatnya, fitur pengubah tema, validasi input guestbook, dan konfirmasi penghapusan tidak berjalan sebagaimana mestinya. Namun, kerusakan ini dikategorikan minor karena fungsi utama halaman Stored XSS, yaitu

menerima, menyimpan, dan menampilkan input guestbook, masih tetap dapat digunakan.

Pada konfigurasi Strict-CSP, fungsionalitas kembali normal setelah dilakukan refaktorisasi kode. Penyesuaian dilakukan dengan memisahkan logika JavaScript dari struktur HTML, mengganti penggunaan `onclick` dengan `addEventListener`, serta menambahkan nonce valid pada skrip yang sah. Hasil ini menunjukkan bahwa kebijakan CSP yang ketat tetap dapat diterapkan tanpa mengganggu fungsionalitas apabila kode aplikasi telah disesuaikan dengan prinsip CSP modern.

3) *Hasil Pengujian Kompatibilitas pada DOM*-based XSS: Berdasarkan hasil pengujian, baseline berjalan normal karena tidak terdapat header CSP yang membatasi eksekusi JavaScript. Konfigurasi Non-Strict CSP (Permissive) juga tetap normal karena masih menggunakan `unsafe-inline`, sehingga inline script dan event handler seperti `onclick` pada fitur pengubah tema dan dropdown bahasa tetap dapat dijalankan.

Sebaliknya, Non-Strict CSP menyebabkan status rusak mayor karena direktif `script-src 'self'` tanpa `unsafe-inline` memblokir JavaScript inline yang digunakan oleh fitur pengubah tema dan dropdown bahasa. Akibatnya, tombol tema tidak mengubah tampilan, sedangkan dropdown bahasa tidak dapat memproses pilihan pengguna dengan benar. Kerusakan ini dikategorikan mayor karena dropdown bahasa merupakan bagian utama dari interaksi pada halaman DOM-based XSS, sehingga gangguan tersebut memengaruhi alur penggunaan halaman secara signifikan.

Pada konfigurasi Strict-CSP, fungsionalitas kembali normal setelah dilakukan refaktorisasi kode. Penyesuaian dilakukan dengan mengganti `onclick` menjadi `addEventListener`, memisahkan logika JavaScript dari struktur HTML, serta menambahkan nonce valid pada skrip yang digunakan oleh fitur tema dan dropdown bahasa. Hasil ini menunjukkan bahwa Strict-CSP tetap dapat diterapkan tanpa mengganggu fungsionalitas apabila kode aplikasi telah disesuaikan dengan prinsip CSP modern.

D. Rekomendasi Implementasi CSP

Berdasarkan hasil pengujian yang telah dilakukan, berikut adalah rekomendasi praktis bagi pengembang aplikasi web yang ingin menerapkan Content Security Policy secara efektif tanpa mengorbankan fungsionalitas.

1) *Prioritaskan Penggunaan Strict CSP dengan Nonce*: Konfigurasi Strict-CSP dengan menggunakan mekanisme nonce dan direktif strict-dynamic direkomendasikan sebagai standar keamanan utama untuk aplikasi web dinamis. Strict-CSP terbukti secara konsisten memberikan tingkat mitigasi 100% terhadap seluruh skenario serangan XSS. Meskipun memiliki beban validasi kriptografi, overhead yang dihasilkan tetap berada pada rentang yang sangat rendah sehingga tidak berdampak signifikan pada pengalaman pengguna. Direktif strict-dynamic mempermudah pengelolaan skrip dengan memberikan kepercayaan otomatis (trust propagation) kepada skrip turunan yang dimuat oleh skrip tepercaya.

2) Gunakan Hash untuk Skrip Statis: Jika aplikasi web memiliki inline script yang jarang mengalami perubahan, maka penggunaan hash dalam Content Security Policy dapat menjadi solusi yang efektif. Browser modern mendukung mekanisme ini dengan cara mencocokan nilai hash dari skrip yang dijalankan dengan hash yang didefinisikan pada header CSP. Dengan pendekatan ini, pengembang tidak perlu memodifikasi atribut pada tag HTML, sehingga lebih sederhana untuk skrip yang bersifat statis. Namun demikian, penggunaan hash memiliki keterbatasan. Setiap perubahan sekecil apa pun pada isi skrip akan menghasilkan nilai hash yang berbeda, sehingga pengembang harus menghitung ulang dan memperbarui hash pada header CSP. Oleh karena itu, metode ini kurang efisien untuk skrip yang bersifat dinamis atau sering berubah.

3) *Larangan Pengunaan direktif unsafe-inline*: Penggunaan direktif unsafe-inline dalam Content Security Policy (CSP) sebaiknya dihindari karena dapat meniadakan perlindungan utama yang diberikan oleh CSP terhadap serangan Cross-Site Scripting (XSS). Berdasarkan hasil pengujian yang telah dilakukan, konfigurasi Non-Strict CSP (Permissive) yang menerapkan unsafe-inline terbukti gagal dalam memblokir seluruh payload XSS.

Direktif unsafe-inline memungkinkan eksekusi skrip inline tanpa pembatasan, sehingga penyerang dapat dengan mudah menyisipkan dan menjalankan kode berbahaya melalui celah XSS. Hal ini secara langsung bertentangan dengan tujuan penerapan CSP, yaitu membatasi sumber dan eksekusi skrip yang tidak dipercaya.header CSP. Oleh karena itu, metode ini kurang efisien untuk skrip yang bersifat dinamis atau sering berubah

4) *Refaktorisasi Skrip*: Jika implementasi Strict-CSP berbasis nonce tidak memungkinkan untuk diterapkan, pengembang disarankan melakukan refaktorisasi skrip secara menyeluruh untuk mendukung kebijakan Non-Strict CSP. Seluruh kode JavaScript sebaiknya dipindahkan dari dokumen HTML ke berkas eksternal yang sah. Praktik ini sejalan dengan Stamm et al. (2010) yang menjelaskan bahwa Content Security Policy (CSP), melalui pembatasan dasar seperti pelarangan inline script, mendorong pemisahan antara konten (HTML) dan kode (JavaScript) serta mengharuskan penggunaan skrip eksternal sebagai mekanisme mitigasi terhadap serangan XSS.

5) Menerapkan Teknik Output Buffering untuk Aplikasi Lawas: Output Buffering adalah mekanisme di PHP yang memungkinkan menampung seluruh output yang dihasilkan oleh script sebelum dikirim ke browser. Dengan OB, data ditampung sementara di dalam memori server dan dikirim ke browser setelah script selesai.

Dengan menerapkan teknik OB, pengembang tidak perlu mengubah ribuan file template atau menyisipkan nonce secara manual, Penyisipan nonce diatur dalam satu fungsi

e-ISSN: 2548-6861

yang memudahkan pemeliharaan, sehingga sangat berguna untuk aplikasi yang sudah berjalan lama dan sulit diubah struktunya. Tetapi, Teknik OB tidak disarankan untuk lingkungan yang menerapkan caching halaman penuh, dikarenakan nilai nonce akan ikut tersimpan dalam cache dan menjadi tidak valid pada muatan berikutnya.

IV. KESIMPULAN

Berdasarkan hasil penelitian, pengujian dan analisis yang telah dilakukan mengenai efektivitas Content Securitxy Policy dalam memitigasi serangan XSS pada aplikasi web berbasis PHP, maka dapat ditarik kesimpulan sebagai berikut:

1) *Efektivitas Keamanan*: Penerapan Strict-CSP dan Non-Strict CSP terbukti 100% efektif memblokir seluruh payload serangan Reflected, Stored dan DOM-based XSS pada browser Chromium maupun Firefox. Sebaliknya, konfigurasi Non-Strict CSP (Permissive) yang masih mengizinkan unsafe-inline dinyatakan gagal total.

2) *Dampak Performa*: Implementasi CSP menimbulkan overhead performa. Pada konfigurasi Strict-CSP, terjadi peningkatan beban komputasi internal browser (System Time) yang relatif tinggi, yakni sebesar +46.77%. Namun secara absolut, peningkatan hanya berkisar 23.2 ms (dari 46.77 ms menjadi 72.8 ms). Karena selisih waktu dibawah 100ms tidak dapat dipersepsikan oleh mata manusia, maka dampak komputasi ini dinilai tidak memberikan pengaruh degradasai yang signifikan terhadap pengalaman pengguna.

3) *Dampak Fungsionalitas*: Implementasi CSP tanpa melakukan modifikasi kode dapat menyebabkan kerusakan fungsionalitas, maslah kompatibilitas ini dapat diatasi sepenuhnya melalui refaktorisasi kode, seperti mengganti inline event handler dengan addEventListener dan menyisipkan atribut nonce yang valid.

4) *Konfigurasi Paling Optimal*: Strict-CSP dengan mekanisme nonce dan direktif strict-dynamic merupakan konfigurasi terbaik untuk aplikasi web dinamis. Pendekatan ini memberikan keseimbangan ideal antara keamanan tertinggi, kemudahan pengelolaan skrip dan efisiensi performa.

UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada Politeknik Negeri Batam atas dukungan fasilitas dan kesempatan yang diberikan dalam pelaksanaan penelitian ini. Penulis juga menyampaikan terima kasih kepada dosen pembimbing serta seluruh pihak yang telah memberikan arahan, masukan, dan bantuan selama proses penyusunan penelitian.

DAFTAR PUSTAKA

[1] W3Techs, “Usage statistics of server-side programming languages for websites,” W3Techs.com. Accessed: Jun. 28, 2026. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[2] OWASP, “OWASP Top 10:2025,” owasp.org. Accessed: Jun. 28, 2026. [Online]. Available: https://owasp.org/Top10/2025/A05_2025-Injection/

[3] S. Gupta and B. B. Gupta, “Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art,” Jan. 01, 2017, *Springer*. doi: 10.1007/s13198-015-0376-0.

[4] S. Stamm, B. Sterne, and G. Markham, *Reining in the Web with Content Security Policy*. ACM Digital Library, 2010. Accessed: Oct. 12, 2025. [Online]. Available: https://archives.iw3c2.org/www2010/proceedings/www/p921.pdf

[5] S. Calzavara, A. Rabitti, and M. Bugliesi, “Content security problems?: Evaluating the effectiveness of content security policy in the wild,” in *Proceedings of the ACM Conference on Computer and Communications Security*, Association for Computing Machinery, Oct. 2016, pp. 1365–1375. doi: 10.1145/2976749.2978338.

[6] Suphi Cankurt, “Security Headers Adoption Study 2026,” AppSec Santa. Accessed: Jun. 30, 2026. [Online]. Available: https://appsecsanta.com/research/security-headers-study-2026

[7] Lukas Weichselbaum, “Mitigate cross-site scripting (XSS) with a strict Content Security Policy (CSP),” web.dev. Accessed: Jun. 28, 2026. [Online]. Available: https://web.dev/articles/strict-csp

[8] OWASP Foundation, “Content Security Policy Cheat Sheet,” OWASP Cheat Sheet Series. Accessed: Jun. 28, 2026. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html#csp-sample-policies

[9] Vijay, “Defect Severity and Priority in Testing with High Severity and Low Priority Example,” Software Testing Help. Accessed: Jun. 28, 2026. [Online]. Available: https://www.softwaretestinghelp.com/how-to-set-defect-priority-and-severity-with-defect-triage-process/?utm_source=chatgpt.com

[10] E. Abgrall, Y. Le Traon, M. Monperrus, S. Gombault, M. Heiderich, and A. Ribault, “XSS-FP: Browser Fingerprinting using HTML Parser Quirks,” Nov. 2012, [Online]. Available: http://arxiv.org/abs/1211.4812