open access
Published online on: https://journal.uii.ac.id/jurnalsnati/
Check for updates
DOI: 10.20885/snati.v5.i1.42248
JURNAL SNATI Sains, Nalar, dan Aplikasi Teknologi Informasi
Jurnal Sains, Nalar, dan Aplikasi Teknologi Informasi
Vol. 5 No. 1 (2026) 62 - 71 ISSN Media Electronic: 2807-5935

Implementasi Mitigasi Berlapis Kerentanan Unrestricted File Upload dan Server-Side JavaScript Injection

Implementation Layered Mitigation Techniques for Unrestricted File Upload and Server-Side JavaScript Injection

Salman Akbar Hasbullah[1], Mohamad Nurkamal Fauzan[2], Roni Andarsyah[3*]
[1,2,3] Program Studi D4 Teknik Informatika, Sekolah Vokasi, Universitas Logistik dan Bisnis Internasional, Bandung, Indonesia
[1]1214073@std.ulbi.ac.id, [2]m.nurkamal.f@ulbi.ac.id, [3*]roniandarsyah@ulbi.ac.id

Abstract

The popularity of Node.js as a server-side application development platform has introduced new security challenges stemming from the dynamic features of JavaScript. Vulnerabilities such as Unrestricted File Upload (UFU) and Server-Side JavaScript Injection (SSJI) often arise from insecure input handling and over-reliance on third-party libraries. This research aims to design, implement, and evaluate a multi-layered security mitigation model for Node.js-based web applications built using the Express.js framework. A constructive research approach was employed, wherein hybrid security middleware was developed to enforce comprehensive validation. This middleware integrates content-based file type validation (magic numbers), file name sanitization to prevent path traversal, and malicious input pattern blocking to mitigate SSJI and prototype pollution. The effectiveness of the model was empirically evaluated within a controlled local testing environment using the Jest testing framework by comparing a vulnerable application against its secured counterpart. Test results demonstrate that the proposed mitigation model successfully blocked 100% of the tested attack scenarios, achieving 100% test code coverage on the core security logic. This research yields a practical solution capable of enhancing the resilience of Node.js applications against common attacks exploiting language-specific features

Keywords: Node.js; Unrestricted File Upload; Server-Side JavaScript Injection; Security Middleware

Abstrak

Popularitas *Node.js* sebagai *platform* pengembangan aplikasi sisi *server* telah membawa tantangan keamanan baru yang bersumber dari fitur-fitur dinamis *JavaScript*. Kerentanan seperti *Unrestricted File Upload* (UFU) *dan Server-Side JavaScript Injection* (SSJI) seringkali muncul akibat penanganan input yang tidak aman dan kepercayaan berlebih pada *library* pihak ketiga. Penelitian ini bertujuan untuk merancang, mengimplementasikan, dan mengevaluasi sebuah model mitigasi keamanan berlapis untuk aplikasi *web* berbasis *Node.js* yang dibangun menggunakan *framework Express.js*. Metode yang digunakan adalah pendekatan penelitian konstruktif, di mana sebuah *middleware* keamanan hibrida dikembangkan untuk menerapkan validasi komprehensif. *Middleware* ini mengintegrasikan validasi tipe *file* berbasis *magic number*, sanitasi nama *file* untuk mencegah *path traversal*, serta pemblokiran pola input berbahaya untuk menanggulangi SSJI dan *prototype pollution.* Efektivitas model dievaluasi secara empiris dalam lingkungan pengujian lokal yang terkontrol menggunakan *framework* pengujian *Jest* dengan membandingkan aplikasi yang rentan dengan aplikasi yang telah diamankan. Hasil pengujian menunjukkan bahwa model mitigasi yang diusulkan berhasil memblokir 100% skenario serangan yang diujikan, dengan cakupan kode pengujian mencapai 100% pada logika inti keamanan. Penelitian ini menghasilkan sebuah solusi praktis yang dapat meningkatkan ketahanan aplikasi *Node.js* terhadap serangan umum yang berbasis pada fitur bahasa.

Kata kunci: Node.js; Unrestricted File Upload; Server-Side JavaScript Injection; Middleware keamanan

1. Pendahuluan

Popularitas *JavaScript* sebagai bahasa pemrograman sisi *server*, yang didorong oleh ekosistem *Node.js,* telah mengubah cara aplikasi web modern dibangun [1-2]. Kemampuannya dalam menangani operasi I/O asinkron membuatnya efisien untuk aplikasi yang membutuhkan skalabilitas tinggi [3-4]. Namun, adopsi yang luas ini diiringi dengan peningkatan vektor ancaman dan kerentanan keamanan yang signifikan [5-6]. Sifat dinamis *JavaScript*, fleksibilitas fitur-fiturnya, dan interaksinya yang kompleks dengan berbagai API [7] dapat secara tidak sengaja memperkenalkan celah keamanan jika tidak dikelola dengan hati-hati [8-9]. Keamanan aplikasi *web* menjadi semakin krusial,

Received: 24-07-2025 | Revised: 25-12-2025 | Accepted: 29-12-2025 | Published: 31-01-2026

mengingat potensi serangan siber yang terus berkembang [10-11].

Penelitian sebelumnya telah menyoroti berbagai risiko spesifik dalam ekosistem *Node.js*. Studi oleh Ntantogian et al. [12] mendemonstrasikan bahaya *Server-Side JavaScript Injection* (*SSJI*) yang timbul dari penyalahgunaan fitur eksekusi kode dinamis [13]. Di sisi lain, Oz et al. [14] mengungkap bahwa *library* unggah file populer seringkali gagal memberikan perlindungan yang memadai terhadap serangan *Unrestricted File Upload* (*UFU*). Kerentanan lain yang berakar pada sifat dinamis *JavaScript* adalah *Prototype Pollution*, di mana manipulasi objek dapat mengubah perilaku aplikasi secara global [15-16]. Tantangan dalam menganalisis kode *JavaScript* yang dinamis juga menjadi perhatian utama [17], yang menunjukkan adanya kesenjangan (gap) antara kebutuhan keamanan dan kemampuan alat yang tersedia. Praktik pengembangan perangkat lunak yang aman (secure SDLC) dan proses code review yang efektif menjadi sangat penting dalam konteks ini, meskipun seringkali sulit diterapkan secara konsisten [18-19].

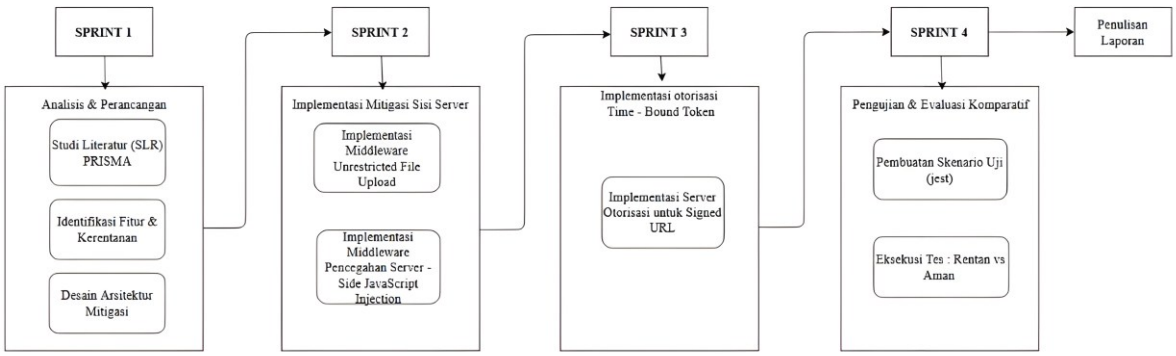
Meskipun SSJI dan UFU merupakan kelas kerentanan yang telah dikenal, mitigasi yang ada seringkali bersifat parsial dan reaktif [13]. Penelitian ini bertujuan untuk merancang, mengimplementasikan, dan mengevaluasi sebuah model keamanan hibrida yang mengintegrasikan validasi sisi *server* yang kuat dengan mekanisme otorisasi modern seperti *time-bound token*. Pendekatan berlapis ini diharapkan dapat menciptakan solusi yang lebih tangguh, efisien, dan komprehensif. Oleh karena itu, tujuan dari penelitian ini adalah untuk menganalisis cara kerja dan sumber kerentanan *Server-Side JavaScript Injection* (SSJI) dan *Unrestricted File Upload* (UFU) dalam lingkungan *Node.js*/Express.js, merancang serta mengimplementasikan model mitigasi berlapis yang mengintegrasikan beberapa lapisan pertahanan seperti validasi konten file, sanitasi nama file, dan pemfilteran input berbahaya, serta mengukur efektivitas dan dampak kinerja dari model gabungan tersebut melalui pengujian fungsional dan pengujian beban secara komparatif terhadap pendekatan yang tidak menggunakan mitigasi.

Untuk menjaga fokus penelitian serta memastikan kedalaman analisis, ruang lingkup, dan batasan masalah dalam penelitian ini ditetapkan sebagai berikut:

1. Fokus penelitian pada kerentanan *Unrestricted File Upload* dan *Server-Side JavaScript Injection* serta dampaknya (*XSS, Prototype Pollution*)
2. Implementasi dilakukan pada lingkungan *runtime Node.js* menggunakan *framework Express.js*.
3. Pengujian performa berfokus pada metrik *Connect Time, Elapsed Time,* dan Latency menggunakan beban 500 hingga 1000 *threads*
4. Mekanisme keamanan diterapkan sebagai *Middleware* sisi server, tidak mencakup keamanan sisi klien atau konfigurasi *firewall*

2. Metodologi Penelitian

Penelitian ini menggunakan pendekatan penelitian konstruktif, di mana fokus utamanya membangun sebuah artefak; dalam hal ini, sebuah *framework* mitigasi perangkat lunak untuk menyelesaikan masalah praktis yang telah teridentifikasi. Proses penelitian ini dilaksanakan melalui empat fase utama yang terstruktur: pertama, tahap analisis dan perancangan yang berfokus pada identifikasi kerentanan serta perancangan arsitektur mitigasi; kedua, implementasi mitigasi sisi *server* dengan membangun *middleware* keamanan untuk menangani UFU dan SSJI; ketiga, implementasi otorisasi *time-bound token* sebagai mekanisme kontrol akses terhadap file yang diunggah; dan keempat, pengujian serta evaluasi komparatif untuk mengukur efektivitas dan dampak kinerja dari sistem yang telah diamankan dibandingkan dengan sistem tanpa mitigasi.



Gambar 1. Alur proses penelitian

Gambar 1 merupakan tahapan alur penelitian yang menggunakan pendekatan konstruktif; tiap fase memiliki tujuan spesifik tertentu untuk menghasilkan komponen yang dapat dievaluasi.

2.1. Analisis dan Perancangan

Tahapan awal ini berfokus pada pengembangan landasan teoritis dan konseptual untuk penelitian. Aktivitas utama pada fase ini adalah sebagai berikut:

1. Studi Literatur Sistematis PRISMA melakukan tinjauan terhadap 41 artikel ilmiah untuk mengidentifikasi fitur-fitur *JavaScript* yang berisiko di lingkungan *server*-side dan jenis-jenis kerentanan yang terkait.
2. Mengidentifikasi Fitur & Kerentanan dari kerentanan *Unrestricted File Upload* dan *Server-Side JavaScript Injection*.
3. Merancang Desain Arsitektur Mitigasi dengan framework keamanan hibrida. Desain ini mencakup pembuatan *middleware* dan pembuatan model untuk otorisasi unggahan file menggunakan *time-bound token.*

2.2. Implementasi Mitigasi Sisi *Server*

Fokus dari Fase ini adalah pengembangan komponen dari inti *framework* keamanan di sisi *server*.

2.2.1. Implementasi *Middleware* UFU

Mengembangkan *middleware* untuk *Node.js* yang menggabungkan tiga lapisan validasi: pemeriksaan tipe file berbasis magic numbers (konten), sanitasi nama file dengan menghasilkan *UUID*, dan (opsional) sanitasi konten dasar.

2.2.2. Implementasi *Middleware* SSJI

Mengembangkan *middleware* untuk mendeteksi dan memblokir payload berbahaya yang menargetkan *SSJI* dan *Prototype Pollution*. Ini melibatkan penggunaan regular expressions untuk mencocokkan pola kode yang berbahaya dan validasi properti objek untuk mencegah polusi prototipe.

2.3. Implementasi Otorisasi *Time-bound Token*

Fase ini didedikasikan untuk membangun komponen kedua dari model keamanan, yaitu arsitektur unggahan file yang lebih modern, dengan mengimplementasikan *server* otorisasi yang mengembalikan endpoint *API* pada *server Node.js* yang bertugas membuat *token* yang aman dan berbatas waktu (*time-bound token*) sebagai respons atas permintaan unggahan dari klien.

2.4. Pengujian dan Evaluasi Komparatif

Fase terakhir ini berfokus pada validasi dari solusi yang telah dibangun, antara lain:

1. Pembuatan Skenario Uji: Merancang dan mengimplementasikan serangkaian *test cases* otomatis menggunakan *framework Jest*. Skenario uji ini mencakup berbagai payload serangan untuk *UFU* dan *SSJI*.
2. Eksekusi Pengujian Komparatif: Menjalankan test suite terhadap kedua aplikasi (rentan dan aman) untuk membandingkan hasilnya. Metrik utama yang diukur adalah tingkat keberhasilan serangan (apakah berhasil diblokir atau tidak).

3. Hasil dan Pembahasan

Bab ini merinci proses eksperimen yang dilakukan untuk menguji dan mengevaluasi efektivitas model mitigasi yang diusulkan. Eksperimen dirancang untuk memvalidasi kemampuan framework dalam mengatasi kerentanan *Unrestricted File Upload* (*UFU*) dan *Server-Side JavaScript Injection* (*SSJI*), serta kerentanan terkait seperti *Cross-Site Scripting* (*XSS*) dan *Prototype Pollution*. Pengujian dilakukan secara komparatif pada dua lingkungan: aplikasi yang sengaja dibuat rentan (sebelum mitigasi) dan aplikasi yang telah diamankan menggunakan m*iddleware* yang dikembangkan (sesudah mitigasi).

3.1. Lingkungan Pengujian

Terdapat dua target aplikasi *server Node.js/Express* yang telah disiapkan

3.1.1. Aplikasi Rentan (vulnerableApp.js)

Sebuah aplikasi dasar yang mengimplementasikan fitur upload file dan pencarian dengan validasi minimal, sengaja dirancang untuk rentan terhadap serangan UFU dan SSJI.

```
// Kerentanan Unrestricted File Upload
const vulnerableStorage = multer.diskStorage({
  destination: "./uploads/safe/",
  Windsurf: Refactor | Explain | Generate JSDoc | X
  filename: (req, file, cb) => {
    // KESALAHAN: Menggunakan nama file asli, rentan terhadap Path Traversal
    cb(null, file.originalname);
  },
});

const vulnerableUpload = multer({ storage: vulnerableStorage });
```

Gambar 1. Kode yang rentan terhadap *UFU*

Gambar 2 merupakan contoh kode multer yang rentan,dikarenakan menggunakan nama *file* asli, yang rentan terhadap *path traversal*

```
app.post("/upload", vulnerableUpload.single("file"), (req, res) => {
  if (!req.file) {
    return res.status(400).send("Upload gagal.");
  }
  res.status(200).send({
    message: "File berhasil diunggah (secara tidak aman).",
    filename: req.file.filename,
  });
});
```

Gambar 2. Kode tanpa validasi

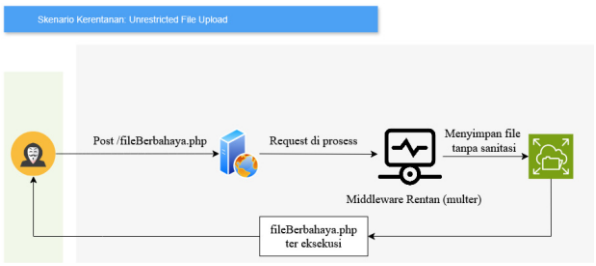
```
// Kerentanan Server-side Javascript Injection
app.get("/search", (req, res) => {
  const query = req.query.q || "";
  try {
    const result = eval(query);
    res.send(`Hasil: ${result}`);
  } catch (e) {
    res.status(500).send(`Error mengeksekusi query: ${e.message}`);
  }
});
```

Gambar 3. Kode rentan terhadap SSJI

Pada gambar 3, salah satu *handler/upload* yang rentan dikarenakan tidak ada validasi atau *middleware* yang diberikan. Sedangkan pada gambar 4, terdapat fungsi *eval()* yang akan mengeksekusi semua kode *JavaScript* yang diberikan, termasuk kode berbahaya. Ini termasuk kerentanan *Server-Side JavaScript Injection*, karena input dari pengguna langsung dieksekusi sebagai kode.

3.1.2. Skenario Serangan dan Mitigasi yang diuji

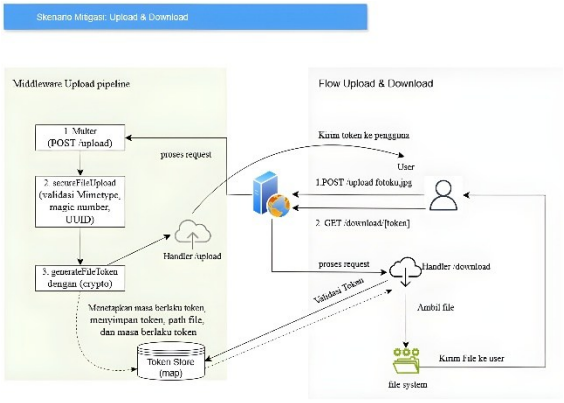
Pada gambar 5 merupakan skenario seseorang penyerang mengunggah file berbahaya ke *server*, seperti fileBerbahaya.*php*, atau file lain yang telah dimanipulasi tipe *MIME*-nya agar lolos validasi permukaan. *Request* tersebut dikirim ke *server* melalui *endpoint* upload (misalnya /upload).



Gambar 4. Skenario serangan UFU

middleware multer, yang dikonfigurasi secara rentan, memproses file tanpa melakukan pemeriksaan yang memadai terhadap nama file, isi file, maupun ekstensi. File langsung disimpan ke dalam sistem file server.

Karena tidak ada sanitasi ataupun validasi pada file tersebut, penyerang dapat mengakses file berbahaya itu secara langsung dan mengeksekusinya di *server*. Hal ini membuka celah *Remote Code Execution* (RCE), yang memungkinkan penyerang mengendalikan *server* secara ilegal.

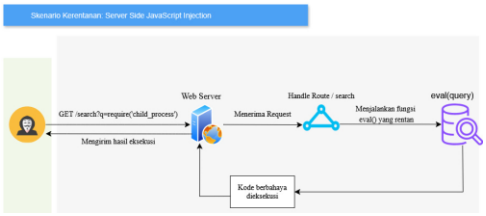


Gambar 5. Alur Mitigasi UFU

Pada gambar 6 terdapat alur untuk mencegah serangan file upload berbahaya; pipeline upload ditingkatkan dengan tiga lapisan m*iddleware*. *Multer* tetap digunakan untuk menangani file dari *request POST /upload*, namun *file* disimpan secara sementara di folder *uploads/temp* dengan nama acak, bukan langsung ke lokasi final.

Middleware secureFileUpload kemudian memvalidasi tipe file menggunakan magic number (bukan hanya ekstensi), kemudian nama file diubah menjadi UUID untuk mencegah eksploitasi path, dan file hanya diproses jika lulus seluruh validasi tersebut.

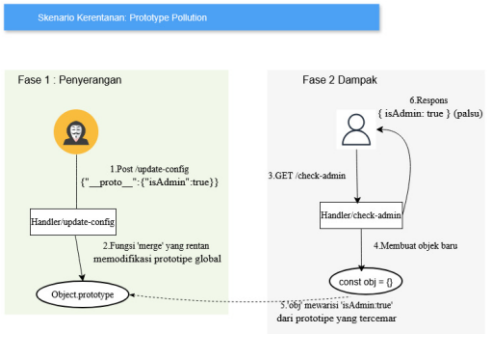
Middleware generateFileToken membuat token yang terkait dengan *file*. *Token* dibuat menggunakan *cypto* dari *Node.js* menggunakan algoritma SHA256. *Token* berlaku dalam batas yang sudah ditentukan, kemudian *token* disimpan di *token store* bersama *path file* dan waktu kadaluarsa. Setelah *upload* berhasil, pengguna mendapatkan *token* sebagai gantinya, bukan *URL* langsung ke *file*. Untuk mengunduh *file*, pengguna harus mengirim *GET /download/:token*, yang kemudian diverifikasi melalui *token store*, jika valid dan belum *expired*, *server* mengirimkan *file* dari sistem *file*. Jika tidak valid, *request* akan ditolak dengan status 403 *forbidden*.



Gambar 6. Skenario serangan SSJI

Serangan *Server-Side JavaScript Injection* (SSJI) terjadi ketika penyerang mengirimkan permintaan *GET* ke *endpoint /search* dengan parameter berbahaya, seperti q=require('child_process'). Web *server* menerima permintaan tersebut dan meneruskannya ke handler route /search, di mana query tersebut diproses secara langsung menggunakan fungsi eval(query).

Penggunaan *eval()* di sisi *server* sangat berisiko karena akan mengeksekusi string sebagai kode *JavaScript*, memungkinkan penyerang untuk menyisipkan dan menjalankan kode arbitrer. Dalam skenario ini, kode berbahaya dijalankan di *server*, dan hasil dari eksekusi tersebut dikirimkan kembali ke penyerang. Hal ini membuka peluang besar untuk eksploitasi sistem, pengambilalihan *server*, atau pencurian data sensitif.

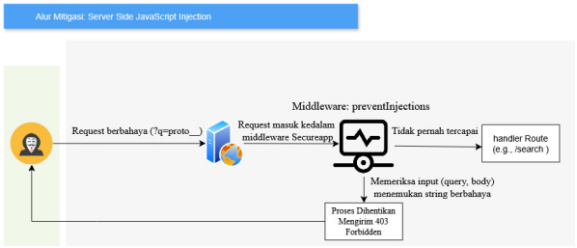


Gambar 7. Skenario serangan *Prototype Pollution*

Pada gambar 8 merupakan skenario serangan Prototype *Pollution* dimulai ketika penyerang mengirimkan permintaan *POST* ke *endpoint* /update-config dengan *payload JSON* berbahaya seperti *{ "__proto__": { "isAdmin": true } }*. Permintaan ini diproses oleh *handler /update-config*, yang menggunakan fungsi *merge()* untuk menggabungkan konfigurasi. Sayangnya, fungsi ini tidak aman dan memungkinkan properti *__proto__* untuk dimodifikasi. Akibatnya, objek global *Object.prototype* tercemar dengan properti baru *isAdmin: true*.

Pada fase kedua, ketika pengguna biasa mengakses *endpoint /check-admin*, *server* membuat objek baru seperti *const obj = {}* di dalam handler. Karena objek baru ini mewarisi dari Object.prototype, maka secara otomatis objek tersebut ikut memiliki properti isAdmin: true, meskipun tidak pernah diset secara eksplisit.

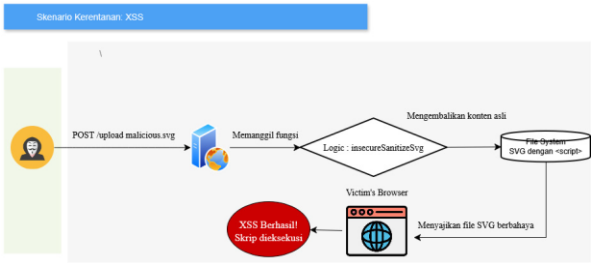
Akhirnya, *server* salah mengenali pengguna biasa sebagai admin dan merespons dengan data palsu *{ isAdmin: true }*, sehingga memungkinkan terjadinya eskalasi hak akses atau pelanggaran otorisasi.



Gambar 8. Skenario mitigasi SSJI

Untuk mencegah serangan *SSJI*, sistem diimplementasikan dengan *Middleware* keamanan seperti *Secureapp*. Ketika *request* masuk ke *server*, *Middleware PreventInjections* secara otomatis memeriksa input dari *query string* dan *body* terhadap pola-pola berbahaya, seperti *?q=__proto__* atau *string* lain yang berpotensi disalahgunakan.

Jika ditemukan pola mencurigakan, proses segera dihentikan sebelum mencapai handler route seperti */search*, dan *server* mengirimkan respons dengan status 403 F*orbidden*. Dengan *pendekatan* ini, kode berbahaya tidak pernah dieksekusi, sehingga melindungi aplikasi dari eksploitasi fungsi-fungsi *JavaScript* seperti *eval()* yang rentan.

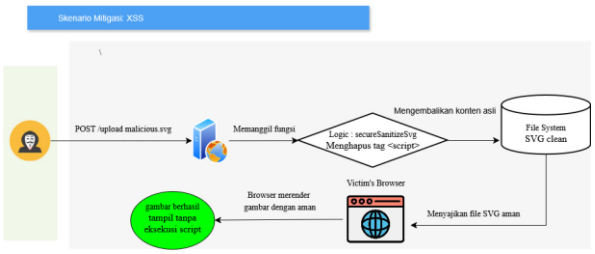


Gambar 9. Skenario serangan *xss*

Penyerang mengunggah *file SVG* berbahaya melalui *endpoint POST /upload malicious. SVG*, yang di dalamnya menyisipkan tag <script> untuk menyuntikkan JavaScript berbahaya.

Web server kemudian memproses file tersebut menggunakan fungsi sanitasi yang tidak aman (insecureSanitizeSVG) yang gagal menghapus elemen-elemen berbahaya seperti <script>. Akibatnya, file SVG dengan konten berbahaya tetap tersimpan di sistem file server tanpa pembersihan.

Ketika file SVG ini disajikan ke browser pengguna (korban), browser merender konten tersebut, termasuk menjalankan skrip yang tertanam. Hal ini menyebabkan serangan *XSS* berhasil, memungkinkan penyerang mencuri data sensitif, menyalahgunakan sesi, atau memanipulasi tampilan halaman.



Gambar 10. Skenario Mitigasi *XSS*

Gambar 11 merupakan skenario mitigasi *cross-site scripting*. Skenario di atas dimulai dari mencegah serangan *XSS* melalui *file SVG*, *server*

mengimplementasikan fungsi pembersih yang aman, seperti *Middleware secureSanitizeSVG*. Saat penyerang mengunggah file *SVG* berbahaya melalui *POST /upload malicious. SVG*, *server* memanggil fungsi ini untuk menyaring elemen berbahaya, termasuk tag <*script*>.

Fungsi sanitasi tersebut secara otomatis menghapus skrip atau atribut yang dapat dieksploitasi, lalu menyimpan versi bersih file SVG ke dalam sistem file.

Ketika file SVG yang telah dibersihkan disajikan ke browser korban, browser hanya akan merender gambar tanpa menjalankan skrip apa pun. Dengan demikian, tampilan tetap aman dan serangan *XSS* berhasil dicegah.

3.2. Analisis Pengujian *Unrestricted File Upload*

Pengujian *UFU* dilakukan dengan mensimulasikan tiga vektor serangan utama, seperti *path traversal*, *mime type spoofing*, dan unggahan file dengan konten berbahaya xss melalui SVG.

Payload yang Digunakan: Sebuah file dikirim dengan nama ../../test.txt. Tujuannya adalah untuk mencoba menulis file di luar direktori unggahan yang telah ditentukan. Dalam pembuatan skenario terdapat dua respon diantaranya:

1. *Response* Aplikasi Rentan
   Aplikasi rentan menerima file tersebut dan, karena menggunakan file.originalname tanpa sanitasi, ia mencoba menulis file ke *path* relatif *uploads/safe/../../test.txt*. Bergantung pada konfigurasi *server*, ini dapat mengakibatkan file ditulis di direktori root aplikasi, yang merupakan celah keamanan serius. Pengujian mengkonfirmasi bahwa permintaan ini diterima dengan status 200 OK dan nama file berbahaya tidak diubah.
2. *Response* Aplikasi Aman
   Aplikasi yang dilindungi oleh *securityMiddleware.js* menerima unggahan file, namun *Middleware* mengabaikan sepenuhnya nama file yang dikirim klien. Sebaliknya, ia menghasilkan nama file baru yang unik menggunakan *UUID* (misalnya, a1b2c3d4-e5f6-....ext). Permintaan berhasil dengan status 200 OK, tetapi serangan *Path Traversal* sepenuhnya digagalkan.

Dengan tidak menggunakan nama file yang disediakan klien, *Middleware* menghilangkan vektor serangan *Path Traversal* pada akarnya. Hal ini sejalan dengan temuan dari studi *(In)Security of File Uploads in Node.js* yang menekankan bahwa validasi nama file adalah salah satu dari tiga pilar penting keamanan unggahan. Berikut ini cuplikan kode pengujian *Jest* yang bisa dilihat pada gambar 12,

```
55
56    test("HARUS MENCEGAH Path Traversal dengan mengganti nama
      file", async () => {
57      const response = await request(secureApp).post("/upload").
        attach("myFile", samplePngPath, "../../../../etc/passwd.
        png");
58
59      expect(response.status).toBe(200);
60      expect(response.body.filename).not.toContain("..");
61    });
```

Gambar 11. Skenario pengujian mitigasi *path traversal*

Pengujian *MIME Type Spoofing* dilakukan dengan sebuah file skrip *JavaScript* (*fake-image.js*) diunggah, namun dengan filename diubah menjadi not-a-script.png dan *header* Content-Type diatur ke image/png.

1. *Response* Aplikasi Rentan
   Aplikasi rentan hanya memeriksa *header* Content-Type. Karena nilainya adalah image/png (yang ada di dalam daftar putih), unggahan diterima. File skrip berbahaya berhasil disimpan di *server* dengan nama .png
2. *Response* Aplikasi Aman
   Middleware secureFileUpload menerima file di lokasi sementara. Kemudian, ia menggunakan *library file-type* untuk membaca beberapa byte pertama dari file (magic numbers). *Library* ini mengidentifikasi konten file sebagai application/*JavaScript*, yang tidak ada dalam daftar allowedMimeTypes. Akibatnya, file sementara dihapus dan permintaan ditolak dengan status 415 Unsupported Media Type.

Ini membuktikan bahwa validasi sisi *server* yang berbasis pada konten file (bukan metadata yang dikirim klien) adalah mekanisme pertahanan yang krusial dan efektif. Mitigasi ini berhasil memblokir serangan yang akan lolos dari filter sederhana.

Pengujian *Xss* dilakukan dengan sebuah *file SVG* yang valid (*malicious.SVG*) yang berisi tag *JavaScript* tersembunyi, seperti <*SVG onload="alert('XSS')"*>.

1. *Response* Aplikasi Rentan
   Aplikasi rentan, yang tidak memiliki mekanisme sanitasi konten, menerima dan menyimpan file *SVG* berbahaya ini apa adanya. Jika file ini kemudian ditampilkan di browser klien, skrip akan dieksekusi, menyebabkan serangan Stored XSS.
2. *Response* Aplikasi Aman
   Aplikasi aman, yang menggunakan *Middleware* secureSanitize*SVG*, mem-parsing konten file *SVG* sebelum menyimpannya. *Middleware* ini secara spesifik mengidentifikasi dan menghapus tag <script> serta atribut event handler seperti onload. File yang disimpan menjadi aman dan tidak lagi dapat mengeksekusi kode *JavaScript*.

Ini menunjukkan pentingnya lapisan pertahanan ketiga, yaitu sanitasi konten. Bahkan jika tipe file diizinkan (seperti *SVG*), kontennya tetap bisa berbahaya. Mitigasi ini secara efektif menetralkan ancaman tanpa harus menolak tipe file yang secara fungsional dibutuhkan.

Pengujian SSJI berfokus pada kemampuan aplikasi untuk menahan upaya injeksi kode dan manipulasi objek. *Payload* yang digunakan (*string 100-50*) dikirim sebagai parameter query ke endpoint/search yang menggunakan *eval()*

1. *Response* Aplikasi Rentan
   Server mengeksekusi string tersebut sebagai kode *JavaScript* dan mengembalikan hasilnya, yaitu 50. Ini mengkonfirmasi bahwa *server* rentan terhadap eksekusi kode arbitrer.
2. *Response* Aplikasi Aman
   Middleware PreventInjections memeriksa semua input yang masuk. *Regex* (\beval\b|Function\s*\(|...) tidak cocok karena payload tidak mengandung kata kunci berbahaya secara eksplisit. Namun, jika payload diubah menjadi eval('100-50'), *Middleware* akan mendeteksinya. Permintaan yang aman (tanpa kode) diteruskan, tetapi dieksekusi oleh route handler yang aman yang hanya memperlakukan input sebagai string, sehingga mengembalikan "Hasil pencarian untuk: 100-50"

Meskipun *Middleware* berbasis pola *Regex* efektif untuk memblokir kata kunci yang jelas-jelas berbahaya, mitigasi terbaik tetaplah dengan tidak menggunakan *eval()* di route handler. *Middleware* berfungsi sebagai jaring pengaman, tetapi praktik pengkodean yang aman adalah pertahanan utamanya.

Pengujian Prototype Pollution dengan menggunakan sebuah objek *JSON* yang berisi kunci *__proto__* seperti *{ "__proto__":{"isAdmin":true}}*, dikirim melalui *request body* ke *endpoint/update-config*

1. *Response* Aplikasi Rentan
   Aplikasi rentan, yang menggunakan fungsi merge objek yang tidak aman, tanpa sadar memodifikasi Object.prototype global. Pengujian ini diverifikasi dengan melakukan *request* kedua ke endpoint lain (/check-admin) yang menunjukkan bahwa semua objek baru sekarang memiliki properti isAdmin: true.
2. *Response* Aplikasi Aman
   Middleware PreventInjections secara rekursif memindai objek input untuk mencari kunci berbahaya (__proto__, constructor, prototype). Ketika __proto__ ditemukan, permintaan segera diblokir dengan status 403 Forbidden sebelum dapat mencapai logika aplikasi.

Ini menunjukkan efektivitas pendekatan proaktif dalam memfilter input. Dengan memblokir pola yang diketahui berbahaya di tingkat *Middleware*, seluruh aplikasi terlindungi dari kerentanan *Prototype Pollution* yang mungkin ada di berbagai fungsi atau *library* yang digunakannya. Berikut ini cuplikan kode pengujian yang bisa dilihat pada Gambar 13.

```
30    test('HARUS MEMBLOKIR request POST yang mengandung
      "prototype" di body', async () => {
31      const pollutionPayload = { config: { prototype: { isAdmin:
        true } } };
32      const response = await request(secureApp).post("/search").
        send(pollutionPayload);
33      expect(response.status).toBe(403);
34    });
35
36    test('HARUS MEMBLOKIR request POST yang mengandung
      "constructor" di body', async () => {
37      const pollutionPayload = { config: { constructor: {
        prototype: {} } } };
38      const response = await request(secureApp).post("/search").
        send(pollutionPayload);
39      expect(response.status).toBe(403);
40    });
```

Gambar 12. Skenario pengujian mitigasi *Prototype Pollution*

3.3. Hasil

Pada tahap ini, dilakukan pengujian untuk mengevaluasi efektivitas dan efisiensi model mitigasi yang diusulkan. Pengujian dibagi menjadi dua bagian utama: uji keamanan fungsional untuk memvalidasi kemampuan mitigasi dalam memblokir serangan, dan uji kinerja untuk mengukur overhead yang ditimbulkan. Pengujian keamanan fungsional yang dilakukan menggunakan Jest menunjukkan bahwa aplikasi yang diamankan dengan *Middleware* yang diusulkan berhasil memblokir 100% dari 33 skenario serangan yang diujikan. Tabel 1 merangkum perbandingan hasil antara aplikasi rentan dan aplikasi aman yang bisa dilihat hasil *code coverage-nya* pada Table 1.

Tabel 1. Hasil code coverage

File	*%statements*	*%Branch*	*% Functions*	*% Lines*
Security *Middleware*.js	100%	93.33%	100%	100%
secureApp.js	88.00%	70.00%	80.00%	88%
vulnerableApp.js	91.17%	71.42%	85.71%	91.17 %
Rata-rata Proyek	93.85%	80.85%	88.88%	93.80 %

Hasil pada tabel 1 menunjukkan bahwa *securityMiddleware.js*, yang merupakan file inti dari implementasi model mitigasi, mencapai cakupan kode yang sangat tinggi. Dengan cakupan *Statements, Functions,* dan *Lines* yang mencapai 100%, ini

menandakan bahwa setiap baris kode dan setiap fungsi di dalam *Middleware* keamanan telah dieksekusi sedikitnya satu kali oleh skenario pengujian. Skenario uji bisa dilihat pada table 2.

Tabel 2. Skenario Uji

Kategori Kerentanan	Jumlah Skenario	Deskripsi singkat	Hasil Mitigasi
Unrestricted File Upload	10	*Path traversal*	100% Blocked
Server-Side Javascript Injection	8	Injeksi *eval*()	100% Blocked
Cross-Site Scripting	5	*File SVG* dengan *tag <script>*	100% Blocked
Prototype Pollution	10	Manipulasi Json	100% Blocked

Hasil pengujian fungsional menunjukkan tingkat keberhasilan 100% pada seluruh 33 skenario uji. Analisis mendalam terhadap hasil ini mengungkap bahwa efektivitas model terletak pada arsitektur validasi hibrida yang diterapkan. Pada kasus *Unrestricted File Upload* (*UFU*), keberhasilan mitigasi didorong oleh pemisahan antara metadata yang dikirim klien (ekstensi/*MIME type*) dengan properti fisik *file* sebenarnya. Penggunaan *magic numbers* terbukti mampu mendeteksi pemalsuan tipe *file* yang lolos dari validasi standar *Multer*. Lebih lanjut, strategi penggantian nama file menjadi *UUID* secara efektif memutus rantai serangan Path Traversal dengan menghilangkan referensi ke direktori sistem.

Hasil pengujian ini sejalan dengan temuan [14], yang menyatakan bahwa ketergantungan pada validasi ekstensi *file* saja tidak memadai untuk mencegah serangan *Unrestricted File Upload* (*UFU*). Penelitian ini membuktikan bahwa validasi konten (*magic numbers*) yang diterapkan pada *middleware* berhasil memitigasi risiko yang terlewatkan oleh *library*. Selain itu, efektivitas pemblokiran fungsi *eval()* dan pola berbahaya lainnya mendukung penelitian [12]. Mengenai bahaya eksekusi kode dinamis pada *Server-Side Javascript Injection*. Pendekatan *hybrid* yang diusulkan dalam penelitian ini mengisi celah keamanan yang disebutkan oleh [13]. Terkait *Prototype Pollution* dengan memblokir akses ke property *__proto__* sebelum mencapai logika aplikasi.

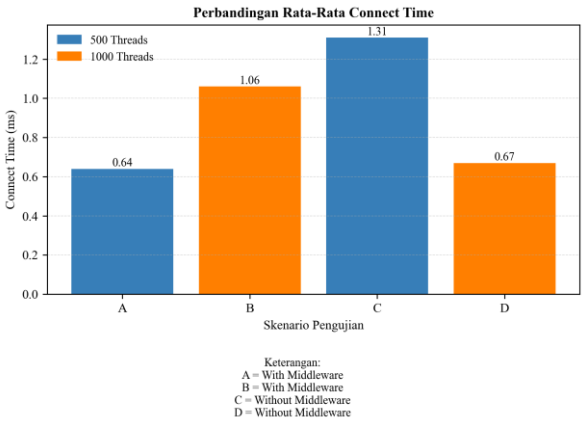
3.4. Hasil Uji *Performance*

Untuk mengevaluasi dampak performa dari penerapan *middleware* mitigasi terhadap fitur file upload, dilakukan pengujian beban menggunakan skenario simultan sebanyak 500 dan 1000 threads [20]. Pengujian ini membandingkan performa sistem dengan dan tanpa *middleware* pada tiga metrik utama: connect time, elapsed time, dan latency, yang seluruhnya diukur dalam satuan milidetik (ms).

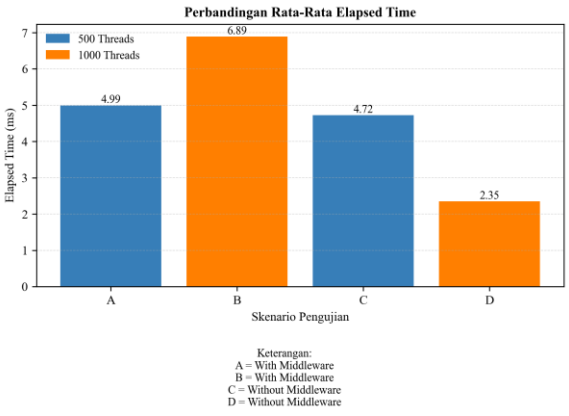
Ringkasan hasil perbandingan performa antara sistem rentan (sebelum mitigasi) dan sistem aman (sesudah mitigasi dapat dilihat pada table 3 berikut :

Tabel 3. Perbandingan Performa Sebelum dan Sesudah Mitigasi

Metrik	Beban	Tanpa Middleware	Dengan Middleware	Overhead
Connect Time	500	1.31ms	0.64ms	-0.67ms
	1000	0.67ms	1.06ms	+0.39ms
Elapsed Time	500	4.72ms	4.99ms	+4.54ms
	1000	2.35ms	6.89ms	+0.29ms
Latency	500	4.69ms	4.98ms	+0.29
	1000	2.34ms	6.87ms	+4.53



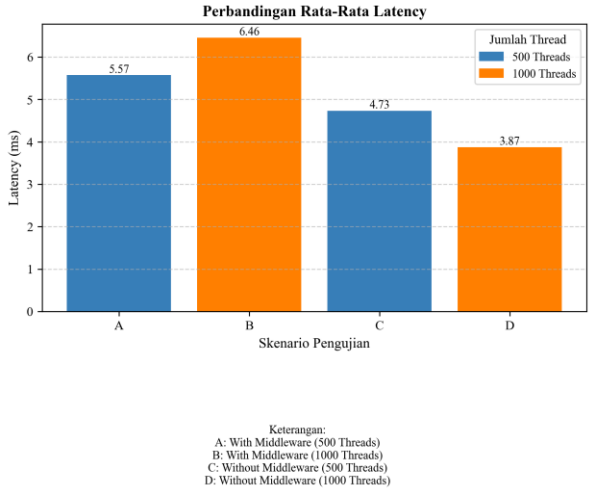
Gambar 13. Perbandingan Connect Time



Gambar 14. Perbandingan elapsed time

Gambar 14 menunjukkan perbandingan waktu koneksi awal antar klien dan *server*. Sistem tanpa *Middleware* menunjukkan hasil waktu koneksi lebih cepat secara konsisten, terutama pada skenario 1000 *threads* dengan waktu rata-rata 0.67 ms dibandingkan dengan 1.06 ms saat menggunakan *Middleware*. Pada skenario 500 *threads*, waktu koneksi sistem tanpa *Middleware* adalah 1.31 ms, sedikit lebih tinggi dibandingkan *Middleware* (0.64 ms), menunjukkan inkonsistensi kecil yang masih dalam batas wajar.

Gambar 15 memperlihatkan hasil waktu total permintaan dari awal hingga akhir (elapsed time). *Middleware* menambah beban proses yang cukup signifikan pada skenario 1000 *threads*, yaitu 6.89 ms, dibandingkan hanya 2.35 ms tanpa *Middleware*. Hal serupa terjadi pada 500 *threads*, dengan *Middleware* memakan waktu 4.99 ms, sedangkan tanpa *Middleware* hanya 4.72 ms. Hasil ini menunjukkan bahwa walaupun *Middleware* menambah sedikit waktu eksekusi, skalanya masih dalam batas yang dapat diterima untuk aplikasi berskala sedang.



Gambar 15. Perbandingan latency

Metrik *latency* yang ditampilkan pada Gambar 16 mengukur waktu tunda antara permintaan dan respons *server*. Pada pengujian 1000 *threads, latency* dengan *Middleware* tercatat sebesar 6.87 ms, lebih tinggi dari tanpa *Middleware* sebesar 2.34 ms. Pada skenario 500 *threads*, *Middleware* menghasilkan latency 4.98 ms dibandingkan 4.69 ms tanpa *Middleware*. Lonjakan latency ini sebanding dengan beban kerja tambahan dari proses validasi dan sanitasi berlapis pada Middleware.

Peningkatan latensi ini disebabkan oleh proses validasi tambahan di sisi *server* yang dilakukan oleh *Middleware*, seperti pembacaan konten file untuk verifikasi *magic number* dan proses sanitasi. Metrik *connect time* tidak menunjukkan perbedaan yang signifikan, yang mengindikasikan bahwa *overhead* terjadi pada level pemrosesan aplikasi, bukan pada level koneksi jaringan. Meskipun terjadi penurunan kinerja, *overhead* ini bersifat konsisten dan tidak menyebabkan kegagalan sistem (*error rate* tetap 0%) di bawah beban. Hal ini menunjukkan bahwa *Middleware* dapat diimplementasikan dalam aplikasi nyata, dengan *trade-off* yang dapat diterima antara sedikit penurunan performa dan peningkatan keamanan yang sangat signifikan.

4. Kesimpulan

Penelitian ini menyimpulkan bahwa kerentanan *Server-Side JavaScript Injection* (SSJI) dan *Unrestricted File Upload* (UFU) merupakan konsekuensi langsung dari karakteristik dasar bahasa *JavaScript* dan ketergantungan tinggi terhadap pustaka eksternal dalam ekosistem *Node.js*. Fitur seperti eksekusi kode dinamis dan manipulasi prototipe menjadi vektor utama serangan, sementara minimnya validasi bawaan dalam modul upload file membuka peluang eksploitasi terhadap jalur file, konten, dan kontrol akses.

Sebagai respons terhadap permasalahan tersebut, telah berhasil dirancang dan diimplementasikan sebuah model mitigasi berlapis dalam bentuk *Middleware* modular untuk Express.js. Model ini menggabungkan pencegahan injeksi berbasis pola, validasi unggahan berbasis konten dan struktur nama file, serta otorisasi akses file berbasis *time-bound token*. Pendekatan ini menyatukan metode reaktif dan proaktif untuk mencapai perlindungan yang komprehensif terhadap berbagai kelas serangan yang relevan dalam konteks aplikasi Node.js.

Evaluasi empiris menunjukkan bahwa model mitigasi ini mampu memblokir seluruh skenario serangan yang diuji, serta mencakup jalur logika secara penuh dalam pengujian dengan cakupan kode 100%. Validasi berlapis terbukti efektif dalam mencegah manipulasi jalur (path traversal), spoofing MIME type, injeksi XSS berbasis *SVG*, serta serangan *SSJI* dan *Prototype Pollution*. Temuan ini mengonfirmasi bahwa arsitektur mitigasi yang dirancang mampu secara signifikan meningkatkan ketahanan aplikasi terhadap kerentanan kritis yang umum ditemukan dalam pengembangan *JavaScript* sisi *server*. Temuan ini mengonfirmasi bahwa arsitektur mitigasi yang dirancang mampu secara signifikan meningkatkan ketahanan aplikasi terhadap kerentanan kritis yang umum ditemukan dalam pengembangan JavaScript sisi server.

Penelitian selanjutnya disarankan untuk:

1. Menguji efektivitas middleware pada arsitektur *microservices* dan lingkungan *cloud serverless* untuk melihat dampak latensi jaringan yang lebih nyata.
2. Mengembangkan mekanisme deteksi berbasis Machine *Learning* untuk mengenali pola injeksi yang lebih kompleks yang mungkin lolos dari filter berbasis *Regex*.

Melakukan analisis keamanan terhadap vektor serangan *Regular Expression Denial of Service* (*ReDoS*) mengingat penggunaan *Regex* yang intensif pada middleware ini.

Reference

[1] H. Hong, S. Woo, and S. Park, "CIRCUIT: A JavaScript Memory Heap-Based Approach for Precisely Detecting Cryptojacking Websites," IEEE Access, vol. 10, no. September, pp. 95356–95368, 2022, doi: 10.1109/ACCESS.2022.3204814.

[2] T. Brito et al., "Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages," IEEE Trans. Reliab., vol. 72, no. 4, pp. 1324–1339, 2023, doi: 10.1109/TR.2023.3286301.

[3] S. An, A. Leung, J. B. Hong, T. Eom, and J. S. Park, "Toward Automated Security Analysis and Enforcement for Cloud Computing Using Graphical Models for Security," IEEE Access, vol. 10, no. June, pp. 75117–75134, 2022, doi: 10.1109/ACCESS.2022.3190545.

[4] S. Fugkeaw and S. Rattagool, "FPRESSO: Fast and Privacy-Preserving SSO Authentication With Dynamic Load Balancing for Multi-Cloud-Based Web Applications," IEEE Access, vol. 12, no. September, pp. 157888–157900, 2024, doi: 10.1109/ACCESS.2024.3485996.

[5] Y. Chen et al., "Understanding the Security Risks of Websites Using Cloud Storage for Direct User File Uploads," IEEE Transactions on Information Forensics and Security, vol. 20, pp. 2677–2692, 2025, doi: 10.1109/TIFS.2025.3544082.

[6] M. Alfadel, N. A. Nagy, D. E. Costa, R. Abdalkareem, and E. Shihab, "Empirical analysis of security-related code reviews in npm packages," Journal of Systems and Software, vol. 203, p. 111752, 2023, doi: 10.1016/j.jss.2023.111752.

[7] S. Calzavara, S. Casarin, and R. Focardi, "Dynamic Security Analysis of JavaScript: Are We There Yet?," WWW 2025 - Proceedings of the ACM Web Conference, pp. 1105–1115, 2025, doi: 10.1145/3696410.3714614.

[8] M. Kang et al., "Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability," Proc. IEEE Symp. Secur. Priv., vol. 2023-May, pp. 1059–1076, 2023, doi: 10.1109/SP46215.2023.10179352.

[9] L. Yan, G. Zhao, X. Li, and P. Sun, "Secure software development: leveraging application call graphs to detect security vulnerabilities," PeerJ Comput. Sci., vol. 11, pp. 1–26, 2025, doi: 10.7717/PEERJ-CS.2641.

[10] M. B. I. N. Muzammil, M. Bilal, S. Ajmal, S. C. Shongwe, and Y. Y. Ghadi, "Unveiling Vulnerabilities of Web Attacks Considering Man in the Middle Attack and Session Hijacking," IEEE Access, vol. 12, no. January, pp. 6365–6375, 2024, doi: 10.1109/ACCESS.2024.3350444.

[11] M. F. Rozi and T. A. O. Ban, "Detecting Malicious JavaScript Using Structure-Based Analysis of Graph Representation," IEEE Access, vol. 11, no. September, pp. 102727–102745, 2023, doi: 10.1109/ACCESS.2023.3317266.

[12] C. Ntantogian, P. Bountakas, D. Antonaropoulos, C. Patsakis, and C. Xenakis, "NodeXP: NOde.js server-side JavaScript injection vulnerability DEtection and eXPloitation," Journal of Information Security and Applications, vol. 58, no. January, p. 102752, 2021, doi: 10.1016/j.jisa.2021.102752.

[13] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 268–279, 2021, doi: 10.1145/3468264.3468542.

[14] H. Oz, A. Acar, A. Aris, A. Kharraz, and S. Uluagac, "(In) Security of File Uploads in Node.js," pp. 1573–1584, doi: 10.1145/3589334.3645342.

[15] A. Sajadi, B. Le, A. Nguyen, K. Damevski, and P. Chatterjee, "Do LLMs consider security? an empirical study on responses to programming questions," vol. 123, pp. 1–29, 2025, doi: 10.1007/s10664-025-10658-6.

[16] K. Iwamura, A. Akmal, and A. Mohd, "Secure User Authentication With Information Theoretic Security Using Secret Sharing-Based Secure Computation," IEEE Access, vol. 13, no. January, pp. 9015–9031, 2025, doi: 10.1109/ACCESS.2025.3526632.

[17] M. Ferreira, I. I. S. Técnico, and U. De Lisboa, "Efficient Static Vulnerability Analysis for JavaScript with Multiversion Dependency Graphs," vol. 8, no. June, 2024, doi: 10.1145/3656394.

[18] S. A. Ebad, "Exploring How to Apply Secure Software Design Principles," IEEE Access, vol. 10, no. September, pp. 128983–128993, 2022, doi: 10.1109/ACCESS.2022.3227434.

[19] R. A. Khan, "Evaluating Performance of Web Application Security Through a Fuzzy Based Hybrid Multi-Criteria Decision-Making Approach: Design Tactics Perspective," vol. 8, 2020.

[20] A. N. Syauqi and N. Q. Nada, "Analisis Kinerja Website Informatika UPGRIS melalui Pengujian Performa Menggunakan JMeter," in Prosiding Seminar Nasional Informatika, 2023, pp. 965–971.