Implementasi Middleware Verifikasi JWT untuk Mencegah Akses Tidak Sah pada API: Studi Kasus di Perusahaan XYZ

Alfonso Hasea Sirait*, Maidel Fani *

* Politeknik Negeri Batam
Rekayasa Keamanan Siber
Jl. Ahmad Yani, Batam Centre, Batam 29461, Indonesia
E-mail: alfonsohaseasirait@gmail.com

* Politeknik Negeri Batam
Rekayasa Keamanan Siber
Jl. Ahmad Yani, Batam Centre, Batam 29461, Indonesia
E-mail: maedal.fani@polibatam.ac.id

Abstrak

Sistem API modern yang menggunakan *JSON Web Token* (JWT) rentan terhadap serangan *token tampering* apabila tidak menerapkan verifikasi tanda tangan digital. Penelitian ini mengidentifikasi celah keamanan pada sistem API Perusahaan XYZ di mana server mempercayai *payload* token tanpa verifikasi *signature*, sehingga memungkinkan manipulasi token oleh pengguna internal. Untuk mengatasinya, dikembangkan *middleware* verifikasi JWT menggunakan Node.js 18, TypeScript, dan *library* jsonwebtoken v9.0.2 dengan algoritma RS256. Penelitian menggunakan metode *Design Science Research* (DSR), dengan pemetaan ancaman melalui kerangka STRIDE dan evaluasi melalui pengujian *black-box* terhadap enam skenario. Hasil pengujian menunjukkan *middleware* menolak 100% token yang dimodifikasi, mencakup manipulasi *claims payload*, serangan alg:none, dan substitusi algoritma RS256 ke HS256, tanpa *false positive* pada token valid. Dua skenario tambahan, token kedaluwarsa dan token pasca-*logout*, memvalidasi bahwa akses melalui token tidak berlaku telah tertangani oleh lapisan pemeriksaan masa berlaku yang keandalannya bergantung pada verifikasi *signature* tersebut.

Kata kunci: JWT, *Middleware*, *Token Tampering*, *API Security*, RS256

Abstract

Modern API systems using JSON Web Token (JWT) are vulnerable to token tampering when digital signature verification is not implemented. This study identifies a security vulnerability in the XYZ Company API system where the server trusts token payloads without verifying the signature, allowing token manipulation by internal users. To address this, a JWT verification middleware was developed using Node.js 18, TypeScript, and the jsonwebtoken v9.0.2 library with RS256. The study employs Design Science Research (DSR), with threat mapping through STRIDE and evaluation through black-box testing across six scenarios. Results show the middleware rejected 100% of modified tokens, covering payload claims manipulation, alg:none attacks, and RS256-to-HS256 algorithm substitution, without false positives on valid tokens. Two additional scenarios, expired and post-logout tokens, validate that access through invalid tokens is handled by an expiry-checking layer whose reliability depends on that signature verification.

Keywords : JWT, Middleware, Token Tampering, API Security, RS256

1. Pendahuluan

Middleware verifikasi JWT dikembangkan sebagai respons terhadap celah keamanan nyata yang ditemukan pada sistem API Perusahaan XYZ sebuah celah yang berakar dari tidak diterapkannya verifikasi tanda tangan digital pada proses pemrosesan token. Temuan tersebut pertama kali masuk melalui sistem *bug ticketing* internal, mengungkap kondisi di mana informasi pada *payload* token langsung diekstraksi dan dipercaya oleh sistem tanpa pemeriksaan keaslian maupun integritas token itu sendiri [1]. Untuk memahami mengapa kondisi tersebut berbahaya, perlu dipahami terlebih dahulu cara kerja JWT sebagai

standar autentikasi yang kini mendominasi implementasi API modern, sebuah konsekuensi dari meluasnya adopsi arsitektur berbasis *Application Programming Interface* (API) sebagai tulang punggung pertukaran data di berbagai perusahaan. JWT beroperasi sebagai mekanisme autentikasi *stateless*: informasi pengguna dibawa langsung dalam token yang ditandatangani secara digital, sehingga integritas data dapat dijamin tanpa server perlu menyimpan *state* sesi [2]. Secara struktural, token JWT tersusun atas tiga segmen yang dipisahkan tanda titik *header*, *payload*, dan *signature*. *Header* mendefinisikan algoritma penandatanganan; *payload* memuat *claims* berupa identitas dan hak akses pengguna; *signature* merupakan hasil komputasi kriptografis atas kombinasi keduanya menggunakan *private key* server. Ketiga segmen diencode dengan Base64URL sehingga isinya dapat dibaca siapa pun, namun keasliannya hanya dapat dikonfirmasi oleh pihak yang memegang *key* yang sesuai [3]. Asumsi mendasar yang menopang seluruh mekanisme ini adalah bahwa server *wajib* memverifikasi *signature* sebelum mempercayai isi *payload*. Apabila tahap tersebut dilewati, token asli dan token yang telah dimanipulasi menjadi tidak dapat dibedakan oleh sistem [2]. Pada implementasi RS256, jaminan kriptografis ini seharusnya sangat kuat: *identity provider* menandatangani token menggunakan *private key* RSA yang tidak pernah dibagikan kepada pihak mana pun, sementara verifikasi dilakukan menggunakan *public key* yang dapat diakses secara publik melalui endpoint JWKS (*JSON Web Key Set*). Penyerang yang berhasil memodifikasi *payload* sekalipun tidak akan mampu menghasilkan *signature* yang valid tanpa menguasai *private key* tersebut, namun seluruh jaminan itu runtuh seketika apabila server tidak menjalankan verifikasi sama sekali, sebab sistem akan menerima token apa pun tanpa pemeriksaan [4].

Ancaman semacam ini bukan sekadar skenario hipotetis. OWASP API Security Top 10 (2023) menempatkan *Broken Authentication* yang salah satu penyebabnya adalah penggunaan token tanpa verifikasi memadai sebagai kerentanan kritis yang paling sering dieksploitasi akibat kontrol autentikasi yang tidak tepat [5]. Pada sistem API Perusahaan XYZ, dampak konkretnya dapat digambarkan sebagai berikut: seorang pengguna internal yang memiliki token valid miliknya sendiri cukup membuka jwt.io atau alat publik sejenis, mengganti nilai *email* pada *payload* dengan milik pengguna lain, lalu mengirimkan token hasil modifikasi ke *endpoint* API, seluruhnya melalui antarmuka grafis tanpa menulis satu baris kode pun. Data seluruh pengguna yang emailnya diketahui dapat diakses dengan cara tersebut, melanggar prinsip *least privilege* secara langsung dan berpotensi menyentuh batas regulasi perlindungan data pengguna [1]. Kerentanan pada sistem JWT tidak berhenti pada manipulasi *payload*. Dua vektor serangan tambahan turut diidentifikasi: pertama, serangan *alg:none*, di mana nilai algoritma pada header dihilangkan dan diganti dengan *none* sehingga token tidak memiliki *signature* yang perlu diverifikasi, memungkinkan modifikasi dilakukan tanpa terdeteksi; kedua, substitusi algoritma RS256 ke HS256, di mana perubahan algoritma pada *header* digunakan untuk mengelabui logika verifikasi server [1][4]. Ketiga vektor tersebut manipulasi *payload*, penghilangan algoritma, dan substitusi algoritma dapat dieksploitasi tanpa jejak selama verifikasi *signature* tidak diberlakukan secara ketat, suatu kondisi yang secara fundamental bertentangan dengan prinsip *Zero Trust Security* yang mengharuskan setiap permintaan divalidasi menyeluruh tanpa asumsi kepercayaan terhadap pihak mana pun [6]. Sebagai solusi atas permasalahan tersebut, *middleware* verifikasi JWT diimplementasikan sebagai lapisan keamanan terpusat pada sistem API Perusahaan XYZ. Dengan menempatkan validasi pada satu titik yang berlaku untuk seluruh sistem, risiko *endpoint* yang tidak sengaja terlewat dalam proses pemeriksaan dapat dieliminasi dan konsistensi penerapan keamanan dapat dijamin sepenuhnya [1]. Setiap permintaan yang masuk diproses oleh middleware terlebih dahulu, algoritma pada *header* diperiksa dan keaslian *signature* dikonfirmasi sebelum diteruskan ke logika bisnis, dengan ukuran keberhasilan yang ditetapkan secara

tegas: penolakan 100% terhadap token yang dimodifikasi, sekaligus penerimaan penuh terhadap seluruh token yang valid.

2. Metode Penelitian

Penelitian ini menggunakan metode Design Science Research (DSR) yang mengacu pada Design Science Research Methodology (DSRM), terdiri dari enam tahap: *problem identification & motivation, define objectives of a solution, design & development, demonstration, evaluation, dan communication* [7]. DSR dipilih karena paradigma ini berorientasi pada pembuatan dan evaluasi artefak teknis untuk memecahkan masalah nyata dalam lingkungan yang relevan [8]. Kesesuaian DSR dengan penelitian ini dapat dijelaskan melalui kerangka *three-cycle view* yang dirumuskan Hevner [8], yang menempatkan aktivitas desain di antara dua lingkungan yang berbeda. *Relevance cycle* menghubungkan penelitian dengan lingkungan aplikasi: permasalahan tidak dirumuskan secara hipotetis, melainkan bersumber dari laporan tim keamanan Perusahaan XYZ melalui sistem *bug ticketing* internal, dan kriteria keberhasilan artefak divalidasi kembali oleh tim QA perusahaan pada lingkungan yang sama. *Rigor cycle* menghubungkan penelitian dengan basis pengetahuan: rancangan *middleware* diturunkan dari standar RFC 7519 [3], klasifikasi ancaman OWASP API Security Top 10 [5], serta prinsip *Zero Trust Security* [6], sekaligus mengembalikan kontribusi berupa bukti empiris efektivitas verifikasi *signature* terpusat pada sistem produksi. *Design cycle* berlangsung di antara keduanya sebagai iterasi bangun-evaluasi terhadap artefak. Posisi penelitian ini pada ketiga siklus tersebut menjelaskan mengapa DSR lebih tepat dibandingkan metode eksperimental murni: artefak yang dihasilkan tidak sekadar diuji dalam kondisi laboratorium, melainkan diterapkan dan divalidasi pada sistem yang telah berjalan di lingkungan industri. Keterkaitan setiap tahap DSRM dengan pengembangan *middleware* verifikasi JWT diuraikan sebagai berikut.

2.1 Problem Identification & Motivation

Tahap ini mengidentifikasi celah keamanan berupa tidak diterapkannya verifikasi tanda tangan digital pada sistem API Perusahaan XYZ. Identifikasi bersumber dari dua aktivitas: penelaahan laporan tim keamanan pada sistem *bug ticketing* internal, dan *code review* terhadap basis kode API yang mengonfirmasi bahwa *payload* token diekstraksi tanpa pemeriksaan integritas. Sebagai bagian dari tahap identifikasi masalah, dilakukan pemetaan ancaman menggunakan kerangka STRIDE untuk mengidentifikasi vektor serangan yang relevan terhadap sistem API [9]. Pemetaan diperluas melampaui vektor *tampering* yang menjadi fokus utama penelitian agar cakupan risiko pada sistem autentikasi berbasis JWT terpetakan secara menyeluruh, mencakup pula risiko *information disclosure* pada *payload* token serta *denial of service* pada komponen pendukung verifikasi. Perluasan ini penting karena mekanisme verifikasi JWT tidak beroperasi secara terisolasi, melainkan bergantung pada *endpoint* JWKS eksternal yang turut membawa permukaan serangannya sendiri. Luaran tahap ini berupa daftar ancaman terstruktur yang menjadi dasar penetapan cakupan fungsional artefak. Tabel 1 menyajikan hasil pemetaan ancaman tersebut.

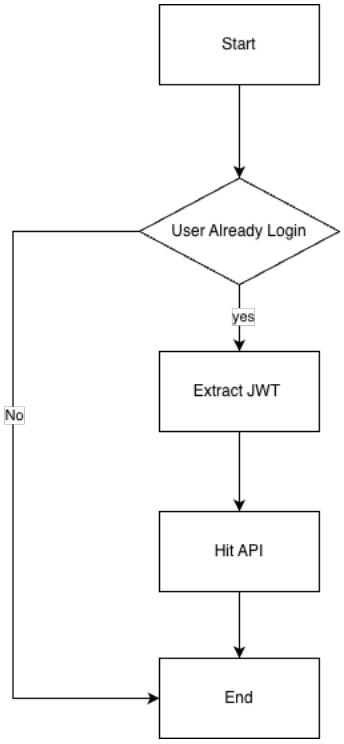
TABEL I

PEMETAAN ANCAMAN DENGAN KERANGKA STRIDE

Ancaman	Kategori STRIDE	Mitigasi
Manipulasi *claims* pada *payload*	Tampering, Elevation of Privilege	Verifikasi *signature* JWT
Serangan alg:none	Tampering, Spoofing	Pengecekan algoritma pada *header*
Substitusi RS256 ke HS256	Tampering	Pengecekan algoritma pada *header*
Keterbacaan *claims* identitas pada *payload*	Information Disclosure	Transport TLS, minimalisasi *claims* sensitif
Denial of service pada *endpoint* JWKS	Denial of Service	*Circuit breaker* pada pengambilan *public key*

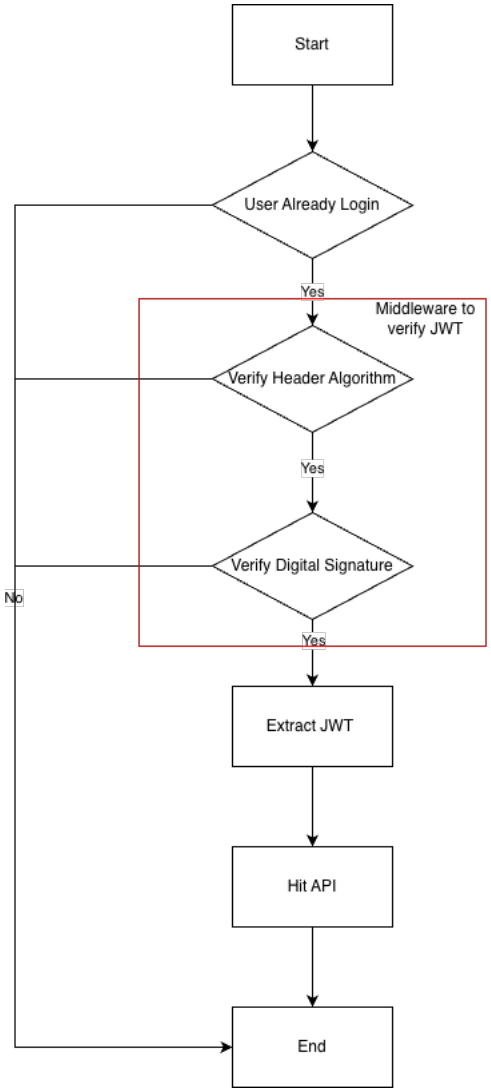
2.2 Define Objectives of a Solution

Ancaman yang telah dipetakan diterjemahkan menjadi tujuan solusi yang terukur, yaitu penolakan 100% terhadap token yang dimodifikasi disertai penerimaan penuh terhadap token yang valid. Kriteria ganda ini ditetapkan karena konteks industri menuntut solusi keamanan yang tidak menimbulkan regresi pada alur autentikasi pengguna yang sah; sebuah *middleware* yang menolak seluruh token akan memenuhi kriteria pertama namun tidak dapat diterapkan pada sistem produksi. Luaran tahap ini berupa spesifikasi fungsional *middleware* beserta metrik evaluasinya. Perbedaan alur pemrosesan *request* sebelum dan sesudah penerapan *middleware* disajikan pada Gambar 1 dan Gambar 2.



Gambar 1: Sebelum Penerapan Middleware

Alur pada Gambar 1 dimulai ketika pengguna mengirimkan *request* ke sistem API. Tahap pemeriksaan pertama menentukan apakah pengguna telah terautentikasi dan membawa token pada *header* Authorization. Apabila tidak, alur berakhir tanpa pemrosesan lebih lanjut. Apabila token tersedia, alur berlanjut ke tahap *extract* JWT, yaitu pembacaan isi *payload* token untuk memperoleh *claims* berupa identitas pengguna dan masa berlaku token. Pada tahap inilah nilai email dan exp diambil dan langsung dipercaya oleh sistem sebagai dasar pengambilan keputusan. *Request* kemudian diteruskan ke *endpoint* API dan logika bisnis dijalankan berdasarkan *claims* tersebut, hingga alur berakhir dengan pengembalian respons. Karakteristik terpenting dari alur ini adalah tidak adanya satu pun tahap pemeriksaan di antara *extract* JWT dan pemanggilan API. Tidak terdapat pengecekan terhadap algoritma yang tercantum pada *header* token, dan tidak terdapat konfirmasi terhadap keaslian *signature*. Konsekuensinya, token yang sah dan token yang telah dimodifikasi menempuh jalur eksekusi yang sepenuhnya identik: keduanya sama-sama didekode, *claims*-nya sama-sama dipercaya, dan keduanya sama-sama menghasilkan respons *200 OK*. Pemeriksaan yang beroperasi terhadap *claims* pada tahap ini pun tidak memberikan jaminan apa pun, sebab nilai yang diperiksa berasal dari *payload* yang integritasnya belum dikonfirmasi dan sepenuhnya berada di bawah kendali klien.



Gambar 2: Setelah Penerapan Middleware

Alur pada Gambar 2 mempertahankan seluruh tahapan pada Gambar 1, namun menyisipkan blok *middleware* verifikasi JWT di antara autentikasi pengguna dan *extract* JWT. Blok tersebut, ditandai dengan garis pembatas pada gambar, terdiri dari dua tahap pemeriksaan yang dilaksanakan secara berurutan. Tahap pemeriksaan pertama, *verify header algorithm*, memeriksa nilai alg yang tercantum pada *header* token. Melalui opsi algorithms: ['RS256'] pada jwt.verify(), hanya token yang menyatakan algoritma RS256 yang diizinkan melanjutkan alur. Token yang mencantumkan none, yang berarti tidak menyertakan *signature* sama sekali, maupun token yang mencantumkan HS256 sebagai upaya *algorithm confusion*, dihentikan pada tahap ini dan alur berakhir dengan respons *401 Unauthorized*. Pemeriksaan ini bersifat pembandingan nilai sederhana dan tidak melibatkan operasi kriptografis apa pun. Tahap pemeriksaan kedua, *verify digital signature*, dilaksanakan hanya terhadap token yang telah lolos pemeriksaan algoritma. *Signature* token dihitung ulang menggunakan *public key* RSA yang diperoleh dari

endpoint JWKS *identity provider*, kemudian dibandingkan dengan *signature* yang dibawa token. Karena *signature* dihasilkan dari kombinasi *header* dan *payload* asli menggunakan *private key* yang tidak pernah dibagikan, perubahan sekecil apa pun pada kedua segmen tersebut menghasilkan ketidakcocokan. Token yang *payload*-nya telah dimodifikasi dihentikan pada tahap ini, dan alur berakhir dengan respons *401 Unauthorized*. Penempatan kedua tahap pemeriksaan secara berurutan bersifat disengaja. Pengecekan algoritma mendahului verifikasi *signature* agar token yang tidak menggunakan algoritma sah dihentikan lebih awal, tanpa membebani sistem dengan komputasi kriptografis terhadap token yang sudah pasti ditolak. Urutan ini sekaligus menutup kemungkinan pustaka verifikasi memproses token dengan algoritma yang tidak diharapkan. Perubahan paling mendasar antara kedua alur terletak pada posisi *extract* JWT. Pada Gambar 1, *claims* dibaca dan dipercaya sebelum keaslian token pernah dikonfirmasi. Pada Gambar 2, tahap yang sama baru dilaksanakan setelah token terbukti menggunakan algoritma yang sah dan membawa *signature* yang valid. Dengan demikian, seluruh *claims* yang menjadi dasar logika bisnis, termasuk email dan exp yang sebelumnya dapat dimanipulasi, kini dijamin merupakan nilai yang diterbitkan oleh *identity provider* dan tidak mengalami perubahan sejak diterbitkan.

2.3 Design & Development

Spesifikasi diterjemahkan menjadi artefak teknis berupa *middleware* verifikasi JWT yang ditempatkan pada lapisan sebelum API *controller*. Keputusan desain pada tahap ini dibatasi oleh kendala lingkungan industri: artefak harus kompatibel dengan ekosistem Node.js dan TypeScript yang telah digunakan perusahaan, memanfaatkan *identity provider* Azure Active Directory B2C yang telah beroperasi, serta dapat diintegrasikan tanpa perubahan struktural pada sistem yang berjalan. Kendala inilah yang mendasari pemilihan pustaka jsonwebtoken v9.0.2 dan strategi *cache-first* berbasis Redis untuk pengambilan *public key* dari *endpoint* JWKS. Ketergantungan pada *endpoint* eksternal tersebut turut mendasari penerapan pola *circuit breaker* sebagai mekanisme ketahanan, agar gangguan pada layanan *identity provider* tidak merambat menjadi kegagalan pada API. Luaran tahap ini berupa implementasi *middleware* yang terdiri dari tujuh komponen utama.

2.4 Demonstration

Artefak diterapkan pada sistem API di *environment development* Perusahaan XYZ untuk membuktikan bahwa solusi berfungsi pada konteks masalah yang sesungguhnya. Demonstrasi dilaksanakan bersama tim QA perusahaan yang mereplikasi serangan *token tampering* berdasarkan dokumen teknis dari laporan awal tim keamanan. Keterlibatan pihak internal perusahaan pada tahap ini memenuhi tuntutan *relevance cycle*, yaitu bahwa artefak diuji pada lingkungan yang menjadi sumber permasalahan.

2.5 Evaluation

Artefak diukur secara sistematis melalui pengujian *black-box* terhadap skenario token yang telah dirancang. Pendekatan *black-box* dipilih karena metrik keberhasilan yang ditetapkan pada tahap sebelumnya bersifat perilaku, yaitu respons sistem terhadap token tertentu, sehingga pengamatan dari sisi eksternal memadai untuk menilai pemenuhan objektif. Pengujian dilakukan pada dua kondisi berurutan, sebelum dan sesudah penerapan *middleware*, agar celah keamanan awal dan mitigasinya dapat dibuktikan secara komparatif. Alat yang digunakan meliputi Google Chrome untuk memperoleh token melalui proses autentikasi normal, jwt.io untuk melakukan *decode* terhadap isi token, token.dev untuk memodifikasi *header* maupun *payload* token serta melakukan penandatanganan ulang, dan Postman untuk mengirimkan *request* ke *endpoint* API. Khusus pada skenario substitusi algoritma, *public key* diperoleh dengan mengakses *endpoint* JWKS milik *identity provider* melalui peramban, kemudian komponen modulus dan eksponen yang tersimpan pada *endpoint* tersebut dikonversi ke format *Privacy Enhanced Mail* (PEM) agar dapat digunakan sebagai *secret* penandatanganan HMAC. Rancangan skenario pengujian disajikan pada Tabel 2.

TABEL 2

RANCANGAN SKENARIO PENGUJIAN

No	Skenario	Ekspektasi Sebelum	Ekspektasi Sesudah
1	Token valid	200 OK	200 OK
2	Manipulasi *claims payload*	200 OK	401 Unauthorized
3	Manipulasi alg:none	200 OK	401 Unauthorized
4	Substitusi RS256 ke HS256	200 OK	401 Unauthorized
5	Token kedaluwarsa	401 Unauthorized	401 Unauthorized
6	Token pasca-*logout*	401 Unauthorized	401 Unauthorized

2.6 Communication

Hasil penelitian dikomunikasikan melalui laporan tugas akhir dan artikel ilmiah ini. Kontribusi yang disampaikan mencakup dua aspek: bagi praktisi,

berupa rancangan *middleware* verifikasi JWT yang dapat direplikasi pada sistem API berbasis Node.js dengan *identity provider* eksternal; bagi basis pengetahuan, berupa bukti empiris bahwa penempatan verifikasi *signature* pada satu titik terpusat mampu memitigasi tiga vektor *token tampering* secara simultan pada sistem yang telah beroperasi.

3. Hasil dan Pembahasan

3.1 Implementasi Middleware

Middleware verifikasi JWT dikembangkan pada lingkungan sistem API Perusahaan XYZ dengan spesifikasi sebagaimana disajikan pada Tabel 3.

TABEL 3

LINGKUNGAN IMPLEMENTASI

Komponen	Keterangan
Runtime	Node.js 18
Bahasa	TypeScript
Framework	Express.js
Library utama	jsonwebtoken v9.0.2
Cache	Redis
Identity Provider	Azure Active Directory B2C
Algoritma JWT	RS256 (RSA Signature with SHA-256)

Middleware ditempatkan pada lapisan sebelum *API controller* sehingga setiap *request* yang masuk wajib melalui proses verifikasi token terlebih dahulu sebelum logika bisnis dijalankan. *Middleware* terdiri dari tujuh komponen utama yang bekerja secara berurutan.

```
const fetchPublicKeys = async (version: "1.0" | "2.0"): Promise<any[]> => {
  const url = version === "1.0" ? B2C_V1_KEYS_URL : B2C_V2_KEYS_URL;
  const circuitBreaker =
    version === "1.0" ? b2cV1CircuitBreaker : b2cV2CircuitBreaker;

  try {
    const keys = await circuitBreaker.execute(async () => {
      const response = await axios.get(url, {
        timeout: 10000, // 10 second timeout
        headers: {
          "Cache-Control": "no-cache",
          "User-Agent": "CUBE-Core/1.0",
        },
      });

      if (!response.data.keys) {
        throw new Error(`No keys found at ${url}`);
      }

      return response.data.keys;
    });

    // Cache the keys for this version
    // cache.set(`publicKeys_${version}`, keys);
    redisCache.set(`publicKeys_${version}`, keys);
    console.log(
      `Successfully fetched and cached ${keys.length} keys for version ${version}`,
    );
    return keys;
  } catch (error) {
    const fetchErrorMessage =
      error instanceof Error ? error.message : "Unknown error";
    console.error(`Error fetching public keys from ${url}:`, fetchErrorMessage);
    console.error(`Circuit breaker state:`, circuitBreaker.getState());

    // Try to return cached keys as fallback
    // const cachedKeys = cache.get<any[]>(`publicKeys_${version}`);
    const cachedKeys: any = await redisCache.get(`publicKeys_${version}`);
    if (cachedKeys) {
      console.log(`Using cached keys as fallback for version ${version}`);
      return cachedKeys;
    }

    throw new Error(
      `Unable to fetch public keys for version ${version}: ${fetchErrorMessage}`,
    );
  }
};
```

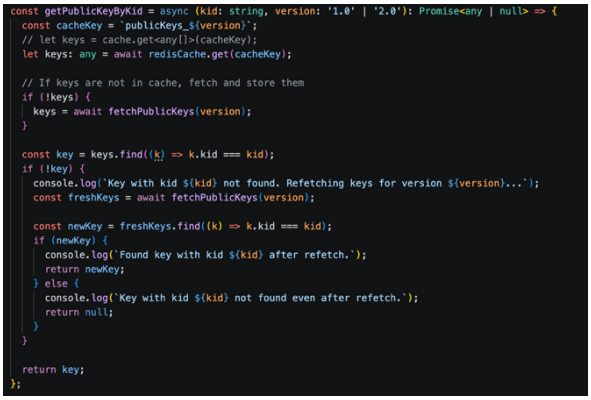
Gambar 3: Kode *Fetch Public Key*

Fungsi fetchPublicKeys() bertanggung jawab mengambil kumpulan *public key* langsung dari *endpoint* JWKS milik Azure Active Directory B2C. Terdapat dua *endpoint* berbeda yang dipanggil bergantung pada versi token yang sedang diproses: *endpoint* Azure B2C *custom policy* untuk token versi 1.0 dan *endpoint* Microsoft Identity Platform untuk token versi 2.0. Pemanggilan dilaksanakan melalui pustaka axios dengan batas waktu 10 detik, dan hasilnya disimpan ke Redis agar tersedia bagi permintaan berikutnya. Ketergantungan pada layanan eksternal menghadirkan persoalan ketahanan yang tidak dapat diabaikan. Apabila *endpoint* JWKS mengalami gangguan atau merespons dengan lambat, setiap permintaan yang memerlukan pengambilan *key* akan tertahan hingga batas waktu terlampaui, sehingga kegagalan pada layanan eksternal merambat menjadi kegagalan pada API. Untuk mencegah kondisi tersebut, pola *circuit breaker* diterapkan sebagai lapisan pelindung pada setiap pemanggilan *endpoint* JWKS.

```
class CircuitBreaker {
  private failures = 0;
  private lastFailTime = 0;
  private state: "CLOSED" | "OPEN" | "HALF_OPEN" = "CLOSED";

  Windsurf: Refactor | Explain | Generate JSDoc | ×
  constructor(
    private threshold = 5,
    private timeout = 30000, // 30 seconds
    private resetTimeout = 60000, // 1 minute
  ) {}

  Windsurf: Refactor | Explain | Generate JSDoc | ×
  async execute<T>(fn: () => Promise<T>): Promise<T> {
    if (this.state === "OPEN") {
      if (Date.now() - this.lastFailTime > this.resetTimeout) {
        this.state = "HALF_OPEN";
      } else {
        throw new Error(
          "Circuit breaker is OPEN - external service unavailable",
        );
      }
    }

    try {
      const result = await Promise.race([
        fn(),
        new Promise<never>((_, reject) =>
          setTimeout(() => reject(new Error("Request timeout")), this.timeout),
        ),
      ]);

      if (this.state === "HALF_OPEN") {
        this.reset();
      }
      return result;
    } catch (error) {
      this.recordFailure();
      throw error;
    }
  }
```

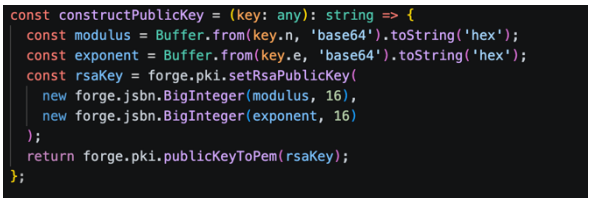
Gambar 4: Kode *Circuit Breaker*

Circuit breaker yang diimplementasikan beroperasi melalui tiga status. Pada status CLOSED, seluruh pemanggilan diteruskan secara normal ke *endpoint* eksternal sembari jumlah kegagalan berturut-turut dicatat. Apabila jumlah kegagalan mencapai ambang batas lima kali, status berubah menjadi OPEN dan seluruh pemanggilan berikutnya ditolak seketika tanpa menyentuh *endpoint* eksternal, sehingga *endpoint* yang sedang bermasalah tidak dibebani permintaan tambahan. Setelah selang waktu 60 detik terlampaui, status berpindah ke HALF_OPEN yang mengizinkan satu pemanggilan percobaan; keberhasilan pemanggilan tersebut mengembalikan status ke CLOSED, sedangkan kegagalannya mengembalikan status ke OPEN. Setiap pemanggilan turut dibatasi oleh batas waktu 30 detik yang diberlakukan secara independen terhadap batas waktu axios, sehingga permintaan yang tergantung tanpa respons tetap dihitung sebagai kegagalan. Dua *circuit breaker* terpisah dipelihara untuk masing-masing *endpoint* JWKS, sehingga gangguan pada *endpoint* token versi 1.0 tidak menghentikan verifikasi token versi 2.0, maupun sebaliknya. Sebagai lapisan pertahanan terakhir, apabila pengambilan *key* gagal sepenuhnya, *middleware* berupaya mengembalikan *key* yang tersimpan di Redis sebagai *fallback*, sehingga verifikasi masih dapat dilanjutkan menggunakan *key* yang terakhir diketahui valid selama entri *cache* belum kedaluwarsa. Selain menjaga ketersediaan layanan, mekanisme ini turut membatasi permukaan serangan *denial of service*. Nilai kid pada *header* token yang tidak ditemukan di *cache* akan memicu pengambilan ulang ke *endpoint* JWKS, sehingga penyerang yang mengirimkan token dengan nilai kid acak secara berulang berpotensi memaksa *middleware* melakukan permintaan keluar dalam volume besar. Dengan *circuit breaker* aktif, rangkaian permintaan semacam itu akan terputus setelah ambang batas kegagalan terlampaui. Analisis lebih lanjut mengenai keterbatasan mitigasi ini disajikan pada Bagian 3.3.



Gambar 5: Kode *Public Key*

Sebelum verifikasi *signature* dapat dilaksanakan, sistem memerlukan *public key* yang tepat dan kebutuhan inilah yang dipenuhi oleh fungsi getPublicKeyByKid(). Berdasarkan nilai *kid* (*Key ID*) yang tercantum di *header* JWT serta versi token yang sedang diproses, fungsi ini bertanggung jawab mengidentifikasi dan mengambil *public key* yang sesuai. Namun demikian, pengambilan *key* langsung dari *endpoint* JWKS eksternal pada setiap permintaan akan membebani waktu respons API dengan latensi yang tidak dapat diabaikan, sebab proses verifikasi baru dapat dilanjutkan setelah respons dari *server* eksternal tersebut diterima. Untuk mengatasi permasalahan tersebut, strategi *cache-first* berbasis Redis diterapkan sebagai lapisan penyangga. *Middleware* terlebih dahulu memeriksa keberadaan *key* di Redis menggunakan cacheKey yang dibentuk dari versi token; apabila ditemukan, proses verifikasi dapat langsung dilanjutkan tanpa koneksi eksternal apa pun. Apabila tidak ditemukan, baik karena sistem baru pertama kali dijalankan maupun karena entri *cache* telah kedaluwarsa, fungsi fetchPublicKeys() dipanggil secara otomatis untuk mengambil *key* langsung dari *endpoint* JWKS dan menyimpannya ke Redis agar tersedia pada permintaan-permintaan berikutnya. Satu kondisi pengecualian ditangani secara eksplisit: apabila nilai *kid* yang dicari tetap tidak ditemukan setelah pengambilan ulang selesai dilaksanakan, fungsi mengembalikan nilai null yang selanjutnya memicu respons *401 Unauthorized*.



Gambar 6: Kode *Construct Public Key*

Public key yang berhasil diperoleh tidak dapat langsung digunakan oleh *library* jsonwebtoken, sebab endpoint JWKS tidak menyimpan *key* dalam format yang siap pakai. Yang disimpan adalah komponen matematis RSA berupa modulus (*n*) dan eksponen (*e*)

yang masing-masing diencode dalam format Base64. Kesenjangan format inilah yang dijembatani oleh fungsi constructPublicKey() melalui tiga langkah konversi yang dilaksanakan secara berurutan. Pertama, modulus dan eksponen didekode dari Base64 ke format heksadesimal. Kedua, kedua komponen hasil dekode digunakan untuk merekonstruksi objek *RSA public key* melalui pemanggilan forge.pki.setRsaPublicKey(). Ketiga, objek yang telah terbentuk dikonversi ke format *Privacy Enhanced Mail* (PEM) menggunakan forge.pki.publicKeyToPem(), format yang secara langsung dapat diterima oleh jsonwebtoken dalam proses verifikasi *signature*. *Public key* dalam format PEM inilah yang selanjutnya diteruskan ke fungsi verifyToken().

```
const verifyToken = async (
  token: string,
  version: "1.0" | "2.0",
): Promise<JwtPayload | null> => {
  const decodedHeader = jwt.decode(token, { complete: true }) as {
    header: { kid: string };
  } | null;

  if (!decodedHeader || !decodedHeader.header.kid) {
    throw new Error("Invalid token header");
  }

  const key = await getPublicKeyByKid(decodedHeader.header.kid, version);
  if (!key) {
    throw new Error(`Key with kid ${decodedHeader.header.kid} not found.`);
  }

  const publicKey = constructPublicKey(key);

  try {
    const verifiedToken = jwt.verify(token, publicKey, {
      clockTolerance: 120,
      algorithms: ["RS256"],
    }) as JwtPayload; // Allow 60 seconds of clock skew
    return verifiedToken;
  } catch (error) {
    if (error instanceof jwt.NotBeforeError) {
      console.error(
        `Token not active. Will be active after: ${(error as jwt.NotBeforeError).date}`,
      );
    } else if (error instanceof jwt.TokenExpiredError) {
      console.error("Token expired:", (error as jwt.TokenExpiredError).message);
    } else if (error instanceof jwt.JsonWebTokenError) {
      console.error("JWT error:", (error as jwt.JsonWebTokenError).message);
    }
    throw error;
  }
};
```

Gambar 7: Kode *Verify Token*

Dengan *public key* yang telah siap, seluruh rangkaian verifikasi dikendalikan oleh fungsi verifyToken() yang bertindak sebagai orkestrator utama memanggil kedua fungsi pendukung sebelumnya dan menjalankan verifikasi melalui tiga tahap berurutan. Pada tahap pertama, token didekode tanpa verifikasi menggunakan jwt.decode() dengan opsi { complete: true } guna mengekstrak nilai *kid* dari header JWT. Proses ini bersifat murni pembacaan metadata; tidak ada validasi *signature* yang dilakukan pada tahap ini. Nilai *kid* diperlukan untuk mengidentifikasi *public key* yang harus digunakan, mengingat *identity provider* dapat menerbitkan beberapa *key* secara bersamaan. Apabila header tidak valid atau nilai *kid* tidak dapat ditemukan, fungsi melempar *error* yang ditangani oleh handleJwtError(). Pada tahap kedua, getPublicKeyByKid() dipanggil untuk mengambil *public key* berdasarkan nilai *kid* dan versi token, yang kemudian dikonstruksi ke format PEM melalui constructPublicKey(). Pada tahap ketiga, verifikasi *signature* dilaksanakan oleh jwt.verify() dengan dua opsi yang ditetapkan secara eksplisit: clockTolerance: 120 untuk memberikan toleransi 120 detik terhadap perbedaan waktu antara server, serta algorithms: ['RS256'] yang membatasi algoritma yang diizinkan hanya pada RS256. Opsi terakhir ini merupakan mekanisme paling kritis dalam keseluruhan implementasi tanpa pembatasan tersebut, jsonwebtoken berpotensi menerima token dengan algoritma apa pun yang tercantum di header, termasuk none yang berarti tidak ada *signature* sama sekali. Dengan pembatasan ini, kesesuaian algoritma diperiksa sebelum verifikasi *signature* bahkan dimulai, sehingga serangan *alg:none* maupun substitusi algoritma RS256 ke HS256 dapat diblokir secara bersamaan.

```
function handleJwtError(error: unknown, res: Response, requestId: any) {
  switch (true) {
    case error instanceof jwt.NotBeforeError:
      Logger.info(
        "passportMiddleware",
        "Token not active",
        `#requestId ${requestId}`,
        "",
        LogLevel.Level2,
        "",
      );
      res.status(401).json({
        message: "Token not active",
        activeAt: (error as jwt.NotBeforeError).date,
      });
      break;

    case error instanceof jwt.TokenExpiredError:
      Logger.info(
        "passportMiddleware",
        "Token expired",
        `#request ${requestId}`,
        "",
        LogLevel.Level2,
        "",
      );
      res.status(401).json({
        message: "Token expired",
        expiredAt: (error as jwt.TokenExpiredError).expiredAt,
      });
      break;

    default:
      Logger.info(
        "passportMiddleware",
        "Invalid token",
        "",
        "",
        LogLevel.Level2,
        "",
      );
      res.status(401).json({ message: "Invalid token" });
  }
}
```

Gambar 8: Kode *Handle JWT Error*

Apabila salah satu tahap dalam verifyToken() menghasilkan kegagalan, penanganannya diserahkan kepada fungsi handleJwtError() yang mengelola seluruh kondisi kegagalan secara terstruktur. Tiga kondisi ditangani secara spesifik: NotBeforeError apabila token belum aktif berdasarkan klaim nbf; TokenExpiredError apabila token telah melampaui waktu kedaluwarsa pada klaim exp; serta seluruh kondisi kegagalan lainnya termasuk kegagalan verifikasi *signature* dan penolakan algoritma yang ditangani oleh blok *default* dengan mengembalikan respons *401 Unauthorized* beserta pesan "Invalid token".

Gambar 9: Kode *Entry Point Middleware*

Keseluruhan rangkaian fungsi tersebut diakses melalui satu titik masuk tunggal: fungsi passportMiddleware() yang didaftarkan langsung ke Express.js. Fungsi ini dirancang untuk menangani dua versi token yang beredar di sistem Perusahaan XYZ, token versi 1.0 dari Azure B2C *custom policy* untuk autentikasi pengguna, serta token versi 2.0 dari *Microsoft Identity Platform* untuk autentikasi antarlayanan. Setiap kondisi kegagalan yang terjadi di sepanjang proses dicatat secara otomatis oleh *logger* dengan menyertakan requestId, sehingga setiap permintaan yang gagal diverifikasi dapat ditelusuri, dipantau, dan diinvestigasi oleh tim keamanan apabila terdapat indikasi serangan. Verifikasi *signature* pada passportMiddleware() mengasumsikan bahwa token yang diterima memiliki struktur JWT yang utuh, yaitu tiga segmen yang dipisahkan tanda titik. Asumsi tersebut tidak dapat dijamin oleh *header Authorization* yang sepenuhnya berada di bawah kendali klien, sehingga token yang cacat secara struktural berpotensi memicu kegagalan pada tahap pembacaan metadata sebelum verifikasi *signature* sempat dilaksanakan. Pada sistem API Perusahaan XYZ, kondisi tersebut telah ditangani oleh *middleware* penentu versi token yang berjalan mendahului passportMiddleware() dalam rantai Express.js. *Middleware* tersebut membaca *payload* token melalui fungsi decodeToken() di dalam blok try-catch, dan mengembalikan respons *401 Unauthorized* apabila proses pembacaan melempar *exception* akibat struktur token yang tidak sesuai spesifikasi RFC 7519 [3]. Dengan demikian, seluruh token yang mencapai *middleware* verifikasi telah dipastikan memenuhi struktur JWT yang sah, sehingga pemeriksaan struktural tidak diduplikasi pada artefak yang dikembangkan dalam penelitian ini. Komponen tersebut berada di luar cakupan artefak dan karenanya tidak dibahas lebih lanjut.

3.2 Hasil Pengujian Black-Box

Pengujian dilakukan pada enam skenario sebelum dan sesudah penerapan *middleware*. Rekapitulasi hasil pengujian disajikan pada Tabel 4.

TABEL 4

REKAPITULASI HASIL PENGUJIAN *BLACK-BOX*

No	Skenario	Hasil Sebelum	Hasil Sesudah	Status
1	Token valid	200 OK	200 OK	Lulus
2	Manipulasi *claims payload*	200 OK	401 Unauthorized	Lulus
3	Manipulasi alg:none	200 OK	401 Unauthorized	Lulus
4	Substitusi RS256 ke HS256	200 OK	401 Unauthorized	Lulus
5	Token kedaluwarsa	401 Unauthorized	401 Unauthorized	Lulus
6	Token pasca-*logout*	401 Unauthorized	401 Unauthorized	Lulus

Enam skenario pengujian dilaksanakan untuk memvalidasi *middleware* secara menyeluruh, mencakup kondisi normal, manipulasi *payload*, dua varian serangan berbasis algoritma, serta dua kondisi token yang tidak lagi berhak memberikan akses. Empat skenario pertama menguji ketahanan sistem terhadap token yang dimodifikasi secara struktural, sedangkan dua skenario terakhir menguji penggunaan token yang secara struktural sah namun telah kehilangan haknya untuk memberikan akses. Setiap skenario diuraikan langkah demi langkah untuk memperlihatkan urutan tindakan yang dilaksanakan serta respons sistem pada setiap kondisi.

Skenario 1 Token valid. Skenario ini memastikan bahwa penerapan *middleware* tidak mengganggu alur autentikasi yang sah. Langkah pengujian dilaksanakan sebagai berikut:

1. Token JWT sah diperoleh melalui proses autentikasi normal pada Google Chrome.
2. Token dikirimkan ke *endpoint* API melalui Postman tanpa modifikasi apa pun.

3. Respons sistem dicatat pada kondisi sebelum *middleware* diterapkan.

4. Langkah kedua diulang pada kondisi sesudah *middleware* diterapkan, dan respons dicatat kembali.

Pada kedua kondisi, sistem mengembalikan respons *200 OK* disertai data profil pengguna yang sah. Hasil identik ini membuktikan bahwa *middleware* tidak menimbulkan *false positive* yang berpotensi berdampak negatif terhadap pengalaman pengguna yang sah.

Skenario 2 Manipulasi claims pada payload. Skenario ini menguji kemampuan *middleware* mendeteksi modifikasi pada *payload*. Langkah pengujian dilaksanakan sebagai berikut:

1. Token sah milik test-login@yopmail.com diperoleh melalui autentikasi normal.

2. Token didekode menggunakan jwt.io untuk menampilkan isi *payload*.

3. Nilai *claim* email pada *payload* diubah menjadi visa_testing@yopmail.com menggunakan token.dev.

4. Token hasil modifikasi dikirimkan ke *endpoint* API melalui Postman pada kedua kondisi pengujian.

Pada kondisi sebelum *middleware* diterapkan, sistem mengembalikan *200 OK* disertai data profil visa_testing@yopmail.com, membuktikan bahwa eksploitasi berhasil tanpa hambatan apa pun. Serangan ini tidak memerlukan keahlian teknis tinggi; seluruh proses dilaksanakan melalui antarmuka grafis tanpa menulis satu baris kode pun. Pada kondisi sesudah *middleware* diterapkan, *request* yang sama ditolak dengan *401 Unauthorized*. Penolakan ini terjadi karena jwt.verify() mendeteksi ketidaksesuaian antara *signature* token dan *payload* yang telah diubah, sebuah konsekuensi langsung dari cara kerja RSA, di mana *signature* dihitung berdasarkan kombinasi *header* dan *payload* asli, sehingga perubahan sekecil apa pun pada *payload* akan menghasilkan *signature* yang tidak valid.

Skenario 3 Manipulasi algoritma menjadi alg:none. Skenario ini menguji ketahanan *middleware* terhadap serangan yang menghilangkan algoritma penandatanganan. Langkah pengujian dilaksanakan sebagai berikut:

1. Token sah diperoleh melalui autentikasi normal.

2. Nilai alg pada *header* token diubah dari RS256 menjadi none menggunakan token.dev, sehingga token tidak lagi memiliki *signature* yang perlu diverifikasi.

3. Token hasil modifikasi dikirimkan ke *endpoint* API pada kedua kondisi pengujian.

Sebelum *middleware* diterapkan, token semacam itu tetap diterima dan mengembalikan data profil pengguna, membuktikan bahwa sistem tidak memeriksa keabsahan algoritma yang tercantum di *header*. Sesudah *middleware* diterapkan, token ditolak dengan *401 Unauthorized* karena opsi algorithms: ['RS256'] pada jwt.verify() mendeteksi bahwa tidak ada algoritma sah yang tercantum di *header*, sehingga pemrosesan dihentikan sebelum verifikasi *signature* bahkan dimulai.

Skenario 4 Substitusi algoritma RS256 ke HS256. Skenario ini menguji ketahanan *middleware* terhadap serangan *algorithm confusion*, yaitu upaya memaksa server memverifikasi token RS256 seolah-olah menggunakan HS256. Serangan ini mengeksploitasi sifat *public key* RSA yang memang dapat diakses secara publik: pada RS256 *public key* hanya digunakan untuk verifikasi, sedangkan HS256 menggunakan satu kunci yang sama untuk penandatanganan dan verifikasi. Apabila server dapat ditipu untuk memproses token sebagai HS256, server akan menggunakan *public key* RSA sebagai *secret* HMAC, sebuah nilai yang juga dikuasai penyerang. Langkah pengujian dilaksanakan sebagai berikut:

1. Token sah diperoleh melalui autentikasi normal.

2. *Public key* diperoleh dengan mengakses *endpoint* JWKS milik *identity provider*, kemudian komponen *key* dikonversi ke format PEM.

3. Pada token.dev, nilai alg pada *header* diubah dari RS256 menjadi HS256.

4. *Public key* dalam format PEM dimasukkan sebagai *secret key* HMAC untuk menandatangani ulang token, sehingga dihasilkan *signature* HS256 yang valid terhadap *public key* tersebut.

5. Token hasil modifikasi dikirimkan ke *endpoint* API pada kedua kondisi pengujian.

Sebelum *middleware* diterapkan, token yang telah ditandatangani ulang dengan skema HS256 diterima oleh sistem, karena server memverifikasi *signature* menggunakan *public key* yang sama dengan yang dipakai penyerang untuk menandatangani. Kondisi ini membuktikan bahwa tanpa pembatasan algoritma, *public key* yang bersifat publik dapat disalahgunakan sebagai *secret* penandatanganan. Sesudah *middleware* diterapkan, token ditolak dengan *401 Unauthorized* karena opsi algorithms: ['RS256'] pada jwt.verify() menolak token pada tahap pemeriksaan algoritma: token yang menyatakan HS256 pada *header*-nya tidak termasuk dalam algoritma yang diizinkan, sehingga

pemrosesan dihentikan sebelum verifikasi *signature* dilaksanakan.

Di antara keempat skenario tersebut, skenario kedua merepresentasikan ancaman yang paling berbahaya dalam konteks sistem Perusahaan XYZ. Tanpa *middleware*, data seluruh pengguna yang emailnya diketahui dapat diakses oleh siapa pun yang memiliki token valid, cukup dengan mendekode token, mengganti nilai *email* pada *payload*, lalu mengirimkan token hasil modifikasi. Kondisi ini secara langsung melanggar prinsip *least privilege* dan berpotensi menimbulkan pelanggaran terhadap regulasi perlindungan data. Setelah *middleware* diterapkan, seluruh vektor serangan tersebut tertutup oleh satu mekanisme terpusat. Pendekatan ini memiliki keunggulan dibandingkan validasi yang dilakukan secara terpisah di setiap *endpoint*, karena konsistensi penerapan keamanan di seluruh sistem dapat dijamin dan risiko *endpoint* yang tidak sengaja terlewat dalam proses validasi dapat dihindari [1]. Skenario kelima dan keenam menguji dimensi ancaman yang berbeda, yaitu penggunaan token yang secara struktural sepenuhnya sah namun tidak lagi berhak memberikan akses. Kedua skenario ini ditambahkan karena tujuan penelitian dirumuskan sebagai pencegahan akses tidak sah secara umum, bukan semata pencegahan *token tampering*. Literatur mencatat bahwa pembatalan token merupakan kelemahan yang melekat pada mekanisme autentikasi *stateless*: karena token membawa sendiri seluruh informasi otorisasinya, *server* tidak secara otomatis mengetahui bahwa sebuah token seharusnya tidak lagi berlaku [10].

Skenario 5 Token kedaluwarsa. Skenario ini menguji perilaku sistem terhadap token yang telah melampaui waktu kedaluwarsa pada klaim exp. Langkah pengujian dilaksanakan sebagai berikut:

1. Token sah diperoleh melalui autentikasi normal, kemudian dibiarkan hingga melampaui masa berlakunya tanpa modifikasi apa pun pada ketiga segmennya.
2. Token yang telah kedaluwarsa dikirimkan ke *endpoint* API pada kondisi sebelum *middleware* diterapkan, dan respons dicatat.
3. Langkah kedua diulang pada kondisi sesudah *middleware* diterapkan.

Pada kedua kondisi, sistem menolak *request* dengan *401 Unauthorized* beserta pesan {"message": "Token expired"}. Hasil identik pada kondisi sebelum dan sesudah menunjukkan bahwa pemeriksaan masa berlaku token telah tersedia pada sistem API Perusahaan XYZ sebelum *middleware* verifikasi diterapkan. Pemeriksaan tersebut dilaksanakan oleh *middleware* penentu versi token yang berjalan mendahului *middleware* verifikasi dalam rantai Express.js, yang membaca klaim exp dari *payload* token dan membandingkannya dengan waktu sistem. Skenario kelima dengan demikian berfungsi sebagai validasi terhadap mekanisme yang telah berjalan, bukan sebagai pengujian terhadap kemampuan baru yang diperkenalkan artefak. Pemeriksaan klaim exp tersebut beroperasi terhadap *payload* yang integritasnya belum dijamin, sehingga nilai yang dibaca sepenuhnya berasal dari data yang dikendalikan klien. Sebelum *middleware* verifikasi diterapkan, nilai exp secara teoretis dapat dimodifikasi ke waktu di masa mendatang sehingga token kedaluwarsa dianggap masih aktif. Setelah verifikasi *signature* diterapkan, modifikasi semacam itu terdeteksi pada tahap jwt.verify(). Verifikasi *signature* dengan demikian tidak menggantikan pemeriksaan masa berlaku, melainkan menjadikannya dapat dipercaya.

Skenario 6 Token pasca-logout. Skenario ini menguji apakah token yang diperoleh sebelum pengguna mengakhiri sesinya masih dapat digunakan setelah proses *logout*. Langkah pengujian dilaksanakan sebagai berikut:

1. Token sah diperoleh melalui autentikasi normal, dan waktu perolehannya dicatat.
2. Pengguna melaksanakan *logout* melalui antarmuka aplikasi.
3. Token yang telah diperoleh pada langkah pertama dikirimkan ke *endpoint* API pada kedua kondisi pengujian.

Pada kedua kondisi, sistem menolak *request* dengan *401 Unauthorized*. Penolakan bersumber dari pemeriksaan masa berlaku token, bukan dari pembatalan sesi: baik *middleware* verifikasi yang dikembangkan maupun *middleware* penentu versi token yang mendahuluinya tidak memelihara daftar token yang dicabut maupun basis data sesi aktif. Manajemen siklus hidup sesi, mencakup penerbitan *refresh token* beserta rotasinya, sepenuhnya didelegasikan kepada Azure Active Directory B2C sebagai *identity provider*. Konfigurasi masa berlaku token yang singkat pada *identity provider* inilah yang membatasi jendela waktu di mana token yang telah ditinggalkan penggunanya masih dapat dimanfaatkan. Hasil kedua skenario tersebut menunjukkan bahwa akses tidak sah melalui token yang tidak lagi berlaku telah tertangani pada sistem API Perusahaan XYZ, meskipun melalui mekanisme yang berbeda dari verifikasi *signature*. Kontribusi *middleware* verifikasi terhadap kedua skenario bersifat menopang: pemeriksaan klaim exp yang menjadi dasar penolakan pada keduanya baru dapat dipercaya sepenuhnya setelah integritas *payload* dijamin melalui verifikasi *signature*. Validasi eksternal dari tim QA Perusahaan XYZ mengonfirmasi bahwa isu *token tampering* yang sebelumnya dilaporkan melalui sistem *bug ticketing* internal dinyatakan selesai, menjadi konfirmasi dari lingkungan perusahaan bahwa *middleware* yang

dikembangkan berfungsi sesuai tujuan yang ditetapkan.

3.3 Analisis Mitigasi Ancaman STRIDE

Tabel 5 menyajikan hasil mitigasi terhadap setiap ancaman yang telah diidentifikasi pada tahap *problem identification*.

TABEL 5

HASIL MITIGASI ANCAMAN STRIDE

Ancaman	Hasil Sebelum Middleware	Hasil Setelah Middleware	Status
Manipulasi *claims payload*	Berhasil, data pengguna lain dapat diakses	Gagal, ditolak 401	Dimitigasi
Serangan alg:none	Berhasil, token tanpa *signature* diterima	Gagal, ditolak 401	Dimitigasi
Substitusi RS256 menjadi HS256	Berhasil, token yang ditandatangani ulang menggunakan *public key* sebagai *secret* HMAC diterima	Gagal, ditolak 401	Dimitigasi
Keterbacaan *claims* identitas pada *payload*	*Claims* dapat dibaca melalui *decode* Base64URL	*Claims* tetap dapat dibaca melalui *decode* Base64URL	Teridentifikasi, mitigasi pada lapisan IdP
Denial of service pada *endpoint* JWKS	Pengambilan *key* berulang tanpa pembatasan	Dibatasi *circuit breaker*	Dimitigasi sebagian

Ketiga ancaman *tampering* berhasil dimitigasi sepenuhnya. Mekanisme inti yang memungkinkan hal ini adalah opsi algorithms: ['RS256'] pada jwt.verify() yang mencegah serangan berbasis manipulasi algoritma, serta verifikasi *signature* RSA yang mendeteksi perubahan sekecil apa pun pada *payload* karena *signature* dihitung berdasarkan kombinasi *header* dan *payload* asli. Validasi eksternal dari tim QA Perusahaan XYZ mengonfirmasi bahwa isu *token tampering* yang sebelumnya dilaporkan melalui sistem *bug ticketing* internal dinyatakan selesai. Dua ancaman selebihnya menuntut penanganan pada lapisan yang berbeda. Ancaman *information disclosure* bersumber dari sifat JWT itu sendiri: ketiga segmennya diencode menggunakan Base64URL dan bukan dienkripsi, sehingga isi *payload* dapat dibaca oleh siapa pun yang memperoleh token tanpa memerlukan *key* apa pun [2]. Pada sistem Perusahaan XYZ, *claims* yang terpapar mencakup nama dan alamat surel pengguna, sementara *claims* yang memuat detail perusahaan turut diencode dalam Base64 yang merupakan skema *encoding* dan bukan enkripsi, sehingga tidak memberikan jaminan kerahasiaan tambahan. Verifikasi *signature* menjamin integritas dan keaslian token, tetapi tidak menjamin kerahasiaan isinya, sehingga ancaman ini berada di luar cakupan artefak yang dikembangkan; mitigasinya berada pada penggunaan TLS di lapisan transport serta pembatasan *claims* atau penerapan JSON Web Encryption (JWE) pada konfigurasi Azure Active Directory B2C sebagai *identity provider*. Ancaman *denial of service* bersumber dari ketergantungan verifikasi *signature* pada *endpoint* JWKS eksternal: apabila nilai kid pada *header* token tidak ditemukan di *cache*, *middleware* memicu pengambilan ulang *key* ke *endpoint* tersebut, sehingga penyerang yang mengirimkan token dengan nilai kid acak secara berulang berpotensi memicu rangkaian permintaan keluar dalam volume besar yang meningkatkan latensi API sekaligus berisiko memicu pembatasan laju dari sisi *identity provider*. Mitigasi terhadap risiko ini telah tersedia dalam bentuk *circuit breaker* yang membungkus setiap pemanggilan *endpoint* JWKS sebagaimana dijelaskan pada Bagian 3.1, yang mencatat kegagalan berturut-turut dan menghentikan pemanggilan keluar setelah ambang batas terlampaui. Perlu dicatat bahwa *circuit breaker* bekerja berdasarkan kegagalan pemanggilan dan bukan berdasarkan laju permintaan, sehingga skenario di mana *endpoint* JWKS tetap merespons secara normal namun dibanjiri permintaan dengan kid tidak dikenal belum tertangani sepenuhnya. Penerapan *rate limiting* pada lapisan *middleware*, atau *caching* negatif terhadap nilai kid yang telah terbukti tidak valid, merupakan penguatan yang direkomendasikan untuk pengembangan lanjutan.

4. Kesimpulan

Tingkat penolakan 100% terhadap seluruh token yang dimodifikasi tanpa satu pun *false positive* pada token valid menjadi ukuran keberhasilan utama yang dicapai oleh *middleware* verifikasi JWT yang dikembangkan dalam penelitian ini. Implementasi dibangun di atas Node.js 18, TypeScript, dan *library* jsonwebtoken v9.0.2 dengan algoritma RS256, di mana validasi tanda tangan digital dilaksanakan menggunakan *public key* yang diambil secara dinamis dari *endpoint* JWKS Azure Active Directory B2C melalui strategi *cache-first* berbasis Redis. Penetapan opsi algorithms: ['RS256'] pada jwt.verify() memastikan bahwa hanya token yang menggunakan algoritma RSA yang sah yang dapat diterima oleh sistem. Pemodelan STRIDE pada tahap identifikasi masalah menghasilkan lima ancaman, yang setelah penerapan *middleware* terbagi ke dalam tiga tingkat penanganan. Tiga ancaman *token tampering*, yaitu manipulasi *claims* pada *payload*, serangan alg:none, dan substitusi algoritma RS256 ke HS256, berhasil dimitigasi sepenuhnya melalui pengecekan algoritma pada *header* yang diikuti verifikasi *signature*. Ancaman *denial of service* pada

endpoint JWKS dimitigasi sebagian melalui *circuit breaker* yang membatasi pemanggilan keluar ketika kegagalan berturut-turut terjadi, namun belum menangani skenario pembanjiran permintaan dengan nilai kid tidak dikenal ketika *endpoint* tetap merespons normal. Ancaman *information disclosure* pada *payload* token berada di luar cakupan artefak, sebab verifikasi *signature* menjamin integritas dan keaslian token tetapi tidak menjamin kerahasiaan isinya; mitigasinya berada pada konfigurasi *identity provider* dan penggunaan TLS di lapisan transport. Dua skenario pengujian tambahan menunjukkan bahwa akses tidak sah melalui token kedaluwarsa maupun token pasca-*logout* telah tertangani pada sistem API Perusahaan XYZ oleh lapisan pemeriksaan masa berlaku yang berjalan mendahului *middleware* verifikasi. Kontribusi artefak terhadap kedua skenario tersebut bersifat menopang: klaim exp yang menjadi dasar penolakan hanya dapat dipercaya setelah integritas *payload* dijamin melalui verifikasi *signature*. Temuan ini menegaskan bahwa verifikasi tanda tangan digital bukan merupakan satu kontrol keamanan di antara banyak kontrol lainnya, melainkan prasyarat yang menopang keandalan seluruh pemeriksaan berbasis *claims* pada sistem autentikasi *stateless*. Konfirmasi dari tim QA Perusahaan XYZ memperkuat hasil tersebut, dengan dinyatakannya isu *token tampering* yang sebelumnya dilaporkan melalui sistem *bug ticketing* internal sebagai selesai. Penerapan *rate limiting* pada lapisan *middleware* serta *caching* negatif terhadap nilai kid yang terbukti tidak valid merupakan penguatan yang direkomendasikan untuk pengembangan lanjutan.

References

[1] A. Venčkauskas, D. Kukta, Š. Grigaliūnas, and R. Brūzgienė, “Enhancing Microservices Security with Token-Based Access Control Method,” *Sensors*, vol. 23, no. 6, Mar. 2023, doi: 10.3390/s23063363.

[2] Seok-Woo Jang and Sang-Hong Lee, “View of Vulnerabilities and Encryption Applications of JWT-Based Authentication Methods”

[3] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” May 2015, doi: 10.17487/RFC7519.

[4] Abed Saif Ahmed Alghawli, “Analysis of Authentication Methods and Secure Web Application Realization With an Integrated Authentication System,” *Applied Mathematics & Information Sciences*, vol. 19, no. 5, pp. 1027–1038, Sep. 2025, doi: 10.18576/amis/190505.

[5] “OWASP Top 10 API Security Risks – 2023 - OWASP API Security Top 10.” Accessed: May 03, 2026. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x11-t10/.

[6] R. Chandramouli, “Guidelines for API Protection for Cloud-Native Systems,” 2025. doi: 10.6028/NIST.SP.800-228.

[7] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, Dec. 2007, doi: 10.2753/MIS0742-1222240302.

[8] A. R. Hevner, S. T. March, J. Park, and S. Ram, “DESIGN SCIENCE IN INFORMATION SYSTEMS RESEARCH 1,” 2004.

[9] A. A. Iwana, R. B. Huwae, and A. H. Jatmika, "Threat Modeling Menggunakan Pendekatan STRIDE dan DREAD untuk Mengetahui Risiko dan Mitigasi Keamanan pada Sistem Layanan Pendidikan," *Jurnal Teknologi Informasi, Komputer dan Aplikasinya (JTIKA)*, vol. 7, no. 1, pp. 142–153, Mar. 2025, ISSN: 2657-0327.

[10] K. F. Lamisa, "Measuring Improper Token Invalidation in Real-World Web Logins," M.A.Sc. thesis, Concordia University, Montreal, QC, Canada, Nov. 2024. [Online]. Available: https://spectrum.library.concordia.ca/id/eprint/994931/