Kerentanan Keamanan *Dependency* pada *Node Package Manager* (NPM)

Alfian Maulana Ibrahim

Sistem dan Teknologi Informasi, Institut Teknologi Bandung

Email: 18217038@std.stei.itb.ac.id

***Abstract*—Sebagai salah satu bahasa pemrograman terpopuler di dunia, Node.js memiliki *package manager* terbesar yaitu Node Package Manager (NPM) dengan lebih dari satu juta *dependency* yang tersimpan di dalamnya. *Dependency* tersebut memiliki beberapa kerentanan keamanan yang dapat menyebabkan kerusakan beruntun di aplikasi yang dependen terhadapnya. Kelemahan yang ada di antaranya disebabkan oleh *syntax* dan *design choices* JavaScript dan Node.js, kelemahan oleh *trivial packages*, dan kelemahan yang disebabkan oleh *technical lag*. Pada makalah ini, akan dibahas mengenai kelemahan tersebut beserta serangan yang mungkin dilakukan, dan juga mitigasi dan pencegahan yang dapat dilakukan seperti dengan melakukan *code review*, mitigasi serangan, minimalisasi *trivial packages*, dan pengadopsian *semantic versioning*.**

***Index Terms*—*Dependency*, NPM, Node.js, *Technical lag*, *Trivial packages*, *Packages*, *Modules*, *Code review*, *Semantic versioning*.**

I. PENDAHULUAN

Dewasa ini, dunia sudah memasuki zaman modern di mana manusia hidup melekat dan berdampingan dengan teknologi. Salah satu contoh teknologi yang sedang terus berkembang dan menjadi inti dalam kehidupan kita adalah aplikasi perangkat lunak (*software application*), baik aplikasi *desktop*, *mobile*, *website*, dan lain sebagainya. Aplikasi-aplikasi yang ada tersebut dikembangkan menggunakan suatu bahasa pemrograman yang dipilih oleh *developer*, dan salah satunya adalah JavaScript.

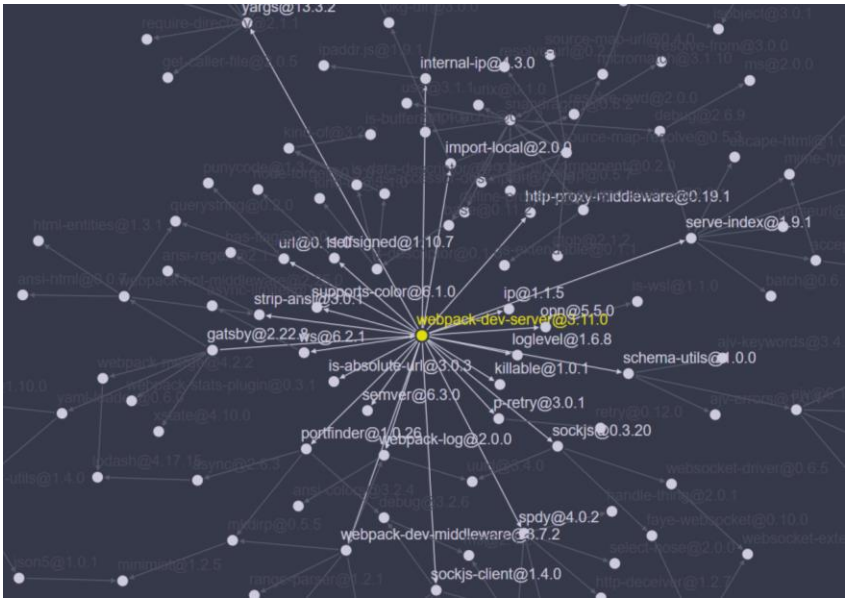
JavaScript (JS) adalah bahasa pemrograman yang ringan dan *interpreted*, umumnya digunakan dalam pengembangan *website* [15]. JavaScript sendiri juga menempati urutan

pertama dalam jajak pendapat mengenai Bahasa Pemrograman Terpopuler 2019 oleh Stack Overflow dengan 67,8% pengguna dari 87.354 responden, bahkan mendominasi di urutan pertama tersebut selama 7 tahun berturut-turut [10].

Node.js merupakan *runtime environment* dari JavaScript yang dibangun berdasarkan mesin JavaScript V8 milik Chrome sehingga bisa dijalankan tanpa harus menggunakan *browser* seperti Chrome, Firefox, dan semacamnya [9]. Node.js juga memiliki komunitas yang besar dibuktikan dengan menempati urutan pertama dalam jajak pendapat mengenai Frameworks, Libraries, dan Tools Terpopuler 2019 oleh Stack Overflow dengan 49.9% pengguna dari 58.543 responden [10].

Dengan komunitas sebesar itu, tidak heran jika Node.js memiliki *package manager* sendiri untuk mengelola *packages* atau *libraries* yang dibuat oleh *developer*, yaitu Node Package Manager (NPM). NPM ini terdiri dari sebuah *command line client*, yang juga disebut npm, dan sebuah basis data daring yang menyimpan *packages* publik dan privat, yang disebut dengan npm *registry*. *Registry* tersebut dapat diakses melalui *client* dan *packages* atau *libraries* yang ada dapat dicari melalui situs web npm [11]. Terdapat sebanyak lebih dari 610.000 *packages* yang terdaftar dan tersimpan sebagai *registry* pada NPM pada tahun 2018 [2], dan bahkan terjadi kenaikan drastis yaitu dua kali lipat sejak tahun 2018-2020 sehingga jumlah *modules* yang ada sekarang mencapai lebih dari 1.200.000 [12].

Pada Node.js, *packages* atau *modules* juga sering disebut sebagai *dependencies*, karena umumnya aplikasi atau *script* yang kita buat sering bergantung (*dependant*) pada kode orang lain atau pihak ketiga. Modul-modul tersebut akan dibuat secara lokal pada *folder* `node_modules` dan dikelola dalam *file* `package.json` dan `package-lock.json`. *Dependencies* pada JavaScript *modules* dapat digambarkan dengan diagram di bawah.



Gambar 1. Dependencies graph dari webpack-dev-server@3.11.0

Graf di atas merupakan contoh gambaran dari *dependencies* yang digunakan oleh salah satu *module* Node.js yaitu webpack-dev-server@3.11.0 [13]. Graf tersebut dihasilkan menggunakan npm Dependency Graph Anvaka. Modul webpack-dev-server tersebut menggunakan lebih dari 20 modul lainnya, dan tiap modul menggunakan banyak modul lainnya pula, sehingga terus bercabang dan menyebabkan ketergantungan yang amat kompleks yang disebut dengan JavaScript Dependency Hell [12]. Karena ketergantungan ini, tiap aplikasi berbasis Node.js memiliki kerentanan keamanan apabila menggunakan modul-modul yang berbahaya baik tersengaja maupun tidak.

Pada makalah ini, akan dibahas mengenai hal-hal berikut:

1. Identifikasi celah keamanan pada *dependency* NPM dan dampaknya kepada aplikasi *dependant* (dibahas pada bagian II).
2. Mitigasi dan pencegahan yang dapat dilakukan untuk mengantisipasi celah keamanan yang mungkin terjadi (dibahas pada bagian III).
3. Kesimpulan (dibahas pada bagian IV).

II. IDENTIFIKASI CELAH KEAMANAN DAN DAMPAKNYA

Celah keamanan pada *dependency* NPM dapat terjadi karena kesengajaan maupun ketidaksengajaan. Pada bagian ini akan dibahas mengenai kelemahan-kelemahan yang ada dan juga serangan yang dapat dilakukan dengan memanfaatkan celah tersebut.

1. Kelemahan berbasis *dependency* yang disebabkan oleh *syntax* JavaScript

Pada poin ini, akan dibahas mengenai kelemahan yang disebabkan oleh desain sintaksis pada JavaScript dan Node.js. Ada tiga kelemahan yang teridentifikasi pada bagian ini.

1.1. Global variable

Bahasa JavaScript terdiri dari fungsi dan variabel (atau *object*) yang disimpan di *namespace* atau *scope* global. *Global variable* sendiri adalah sebuah variabel yang dideklarasikan di luar suatu fungsi atau dengan *window object*, sehingga bisa diakses oleh seluruh fungsi yang ada. Berbeda dengan *local variable* yang hanya bisa diakses pada suatu fungsi yang menaungi variabel tersebut [14].

Program yang menggunakan *global variable* tidak dapat mengatur akses terhadap variabel ini, dan juga tidak dapat melindunginya dari modifikasi eksternal. Hal ini menyebabkan *global variable* tersebut dapat diakses dan dimanipulasi dari segala modul pada aplikasi, termasuk melalui *dependencies*. Dengan melakukan perubahan pada variabel tersebut, otomatis akan memengaruhi seluruh keberjalanan aplikasi [6].

1.2. Monkey-patching

JavaScript mendukung *dynamic modification* terhadap kelas dan fungsi pada *runtime*. Itulah yang disebut dengan fungsi *monkey-patching* [6]. Berikut adalah contoh dari *basic monkey-patching*.

```
1   function MyClass() {};
2
3   MyClass.prototype.suatuFungsi = function() {
4       // Implementasi awal
5   };
6
7   function fungsiMonkeyPatch() {
8       var fungsiAsli = MyClass.prototype.suatuFungsi;
9       MyClass.prototype.suatuFungsi = function() {
10          // Implementasi baru (monkey-patched)
11          // fungsiAsli dapat dipanggil di sini
12      };
13  };
```

Gambar 2. Contoh fungsiMonkeyPatch

Pada kode di atas, kita mendefinisikan suatu kelas bernama `MyClass` dan fungsi dari kelas tersebut yang diberi nama `suatuFungsi`. Di dalam fungsi `fungsiMonkeyPatch`, kita dapat menyalin fungsi asli ke dalam suatu variabel, lalu kita timpa (*overwrite*) `suatuFungsi` dengan fungsi baru sesuai kehendak kita. Pada JavaScript, tidak ada cara untuk mengetahui suatu fungsi itu ter-*monkey-patch* atau

tidak. Sehingga aplikasi Node.js harus memercayai bahwa tidak ada *dependencies* yang melakukan hal tersebut [6].

1.3. Loaded modules cache

Sebuah modul atau *dependency* yang digunakan pada aplikasi Node.js harus secara eksplisit di-*load* menggunakan fungsi `require`. Fungsi `require` tersebut memanfaatkan *cache* untuk mengurangi beban *load*. *Cache* inilah yang dapat diakses oleh semua modul pada aplikasi karena *cache* ini bersifat *shared*, sehingga mirip dengan *global variable*, semua modul bisa memanipulasi isi *cache* tersebut [6].

2. Serangan berbasis *dependency* yang mungkin terjadi dikarenakan kelemahan pada *syntax* JavaScript

Pada poin ini, akan dibahas mengenai empat jenis serangan berbasis *dependency* yang dapat dilakukan dengan mengeksploitasi kelemahan pada poin (1).

2.1. Global leakage

Serangan ini ditujukan untuk membocorkan informasi sensitif yang berada pada suatu program Node.js. Berikut merupakan contoh kode dari serangan ini.

```
1   // deklarasi leak function
2   function bocor(data) {...}
3
4   // pemanggilan leak function
5   bocor(process.env)
```

Gambar 3. Contoh leak function

Pada kode tersebut, dideklarasikan fungsi `bocor` yang mana memiliki parameter `data`. Parameter `data` tersebut dapat kita isi dengan informasi penting seperti *environment variables* (`process.env`) yang seharusnya disembunyikan dari *end-user*. Cara ini memanfaatkan kelemahan *global variable* pada JavaScript. Dengan cara ini, sebuah *dependency* bisa mengakses variabel yang mereka inginkan dan membocorkan kredensial dari korban [5][6].

2.2. Global manipulation

Berikut adalah contoh dari *global manipulation attack.*

```
1   // Pemanggilan modul dan fungsi
2   var dependency = require('dependency');
3   var suatuFungsi = dependency.suatuFungsi;
4
5   // Overwriting suatu fungsi pada dependency
6   dependency.suatuFungsi = function(value) {
7       // Implementasi baru dari suatu fungsi pada suatu dependency
8   }
```

Gambar 4. Contoh global manipulation

Pada serangan ini, yang menjadi tujuan utama adalah memanipulasi atau mengubah variabel atau fungsi yang bisa diakses secara global untuk mengubah perilaku aplikasi tersebut. Serangan di atas memanfaatkan kelemahan *monkey-patching*.

Diberikan suatu *dependency* yang disimpan dalam variabel `dependency`, yang mana modul tersebut memiliki suatu fungsi tertentu yang disebut `suatuFungsi`. Kita dapat mengubah perilaku dari fungsi tersebut seperti untuk mengelabui validasi email, kata sandi, dan lain sebagainya. Serangan ini dapat memberi jalan bagi serangan yang lain seperti serangan XSS atau SQL Injections [5][6].

2.3. Local manipulation

Variabel lokal pada Node.js hanya dapat diakses dalam konteks si pemanggil, seperti dalam fungsi atau kelas yang menaunginya. Akan tetapi, dengan memanfaatkan kelemahan *loaded module cache*, suatu *dependency* dapat memanipulasi alur *dependency* lainnya.

Misalkan terdapat *dependency* A dan *dependency* B. Terdapat suatu aplikasi yang menggunakan *dependency* A dan keduanya (aplikasi dan *dependency* A) sama-sama menggunakan *dependency* B. Di sini, *dependency* A dapat memodifikasi nilai dari suatu variabel dari *dependency* B untuk memengaruhi perilaku aplikasi. Dapat juga dilakukan *monkey-patching* terhadap fungsi yang diekspor dari B [6].

2.4. Dependency-tree manipulation

Serangan ini melakukan manipulasi terhadap *object* yang dikembalikan oleh fungsi `require` dengan menggunakan *cache object* secara langsung pada fungsi `require.resolve`. Terdapat dua variasi pada serangan ini.

```
require('library-berbahaya');
require.cache[require.resolve('library-korban')]
    = require.cache[require.resolve('library-berbahaya')];
```

Gambar 5. Contoh basic dependency tree manipulation attack

```
var original = require('library-korban');
require.cache[require.resolve('library-korban')].exports
    = function() {
        // Melakukan monkey-patching
        return original.apply(this, arguments);
    }
```

Gambar 6. Contoh dependency tree manipulation attack terhadap exported function

Pada contoh pertama, *dependency* asli akan ditimpa dengan *dependency* yang berbahaya. Hal ini menyebabkan semua modul pada aplikasi yang melakukan *load* terhadap *dependency* asli malah akan menggunakan *dependency* berbahaya tersebut.

Untuk variasi kedua, mirip dengan kasus yang pertama, akan tetapi kita memodifikasi fungsi tertentu pada *dependency* asli. Serangan ini dapat membuka celah untuk me-*bypass* kontrol keamanan yang dimiliki oleh aplikasi tersebut [6].

3. Kelemahan yang disebabkan oleh *trivial packages*

Trivial packages adalah modul-modul yang mengimplementasikan hal-hal simpel dan sepele yang sebenarnya tidak perlu dibuat dalam bentuk *dependency packages*. Hal ini disebabkan oleh *code reusability* yang terlalu berlebihan. Pada tahun 2017, terdapat 38.845 *trivial packages* (16,8%) dari total 231.092 *packages* yang terdapat pada NPM [8].

Berdasarkan riset yang dilakukan Rabe Abdalkareem, terdapat tiga alasan utama mengapa *developer* menggunakan *trivial packages*.

a. Terimplementasi dan teruji dengan baik (54,6%)
b. Meningkatkan produktivitas (47,7%)
c. Kode yang terpelihara dengan baik (9,1%)

Akan tetapi, ada harga yang harus dibayar karena menggunakan *trivial packages*, di antaranya adalah tiga alasan berikut [8].

a. *Dependency overhead* (55,7%)

   Terjadi kerepotan dalam memastikan *dependency* yang ada agar tetap terbarui dan sulitnya berurusan dengan rantai *dependencies* yang kompleks.

b. Kerusakan aplikasi (18,2%)

   Para *developer* khawatir akan potensi terjadinya kerusakan aplikasi karena *package* spesifik yang menjadi tidak dapat digunakan. Contoh kasus ini pernah terjadi pada Netflix dan Facebook.

c. Penurunan kinerja (15,9%)

   Penambahan *dependency* tertentu dapat menyebabkan penurunan kecepatan ketika *build* aplikasi maupun meningkatkan waktu instalasi aplikasi.

4. Kelemahan yang disebabkan oleh *technical lag*

Technical lag adalah istilah yang digunakan untuk merefleksikan seberapa usang suatu sistem perangkat lunak terhadap *dependencies*-nya. *Technical lag* menjadi penting dalam *software package distribution* seperti NPM, di mana setiap *packages* bergantung satu terhadap yang lain untuk menggunakan fungsionalitas pihak ketiga [3].

Menurut para peneliti laboratorium perangkat lunak di Universitas Mons, mereka menemukan bahwa satu dari empat *dependencies* dan dua dari lima *releases* mengalami *technical lag*. Sepertiga dari semua *releases* punya setidaknya satu *dependency* yang *target*

package-nya terbarui selama masa rilisnya, dan setengahnya tertinggal dari versi baru, yang menimbulkan atau meningkatkan *technical lag* [3].

Jumlah *packages* yang mengalami kerentanan keamanan (*security vulnerabilities*) terus bertambah setiap waktunya. Sebagian besar memiliki tingkat keparahan (*severity*) sedang atau tinggi. Terdapat 235 *packages* yang mengalami tingkat keparahan sedang dan 139 *packages* yang mengalami tingkat keparahan tinggi dari setiap 399 *packages* [2]. Tidak terkecuali pada Docker Images berbasis Node.js, semua *images* resmi yang ada mengalami kerentanan dengan rata-rata 16 kerentanan per Images [7].

Kerentanan keamanan ini ditemukan dengan waktu lumayan lama, terutama untuk kerentanan dengan tingkat keparahan rendah. Sebagian besar kerentanan ditemukan pada *packages* yang berumur 28 bulan ke atas [2]. Ini menandakan bahwa *technical lag* masih sering terjadi pada *packages* di NPM dengan rata-rata keparahan kerentanan keamanan pada tingkat sedang atau tinggi.

III. MITIGASI DAN PENCEGAHAN

Setelah dibahas mengenai celah keamanan pada *dependency* NPM dan dampaknya, pada bagian ini akan dibahas mengenai mitigasi dan cara pencegahan dari kelemahan dan serangan yang bisa dilakukan. Ada lima poin hal yang bisa dilakukan pada bagian ini.

1. Peninjauan ulang kode *dependency* (*code review*)

   Strategi pertama yang dapat dilakukan adalah dengan cara melakukan peninjauan kode (*code review*) terhadap seluruh *dependency* yang digunakan, yang mana sangat menghabiskan waktu. Peninjauan kode secara manual tidaklah praktis ketika versi baru dari *dependency* ini harus sering diadopsi [6].
2. Perubahan desain Node.js

   Strategi kedua adalah dengan cara mengubah desain dari Node.js itu sendiri. Dapat dilakukan perubahan terutama pada mekanisme *cache* dan penggunaan *global variable* pada Node.js. Akan tetapi, hal ini tidak mungkin dilakukan karena dengan mengubahnya maka akan merusak seluruh aplikasi Node.js yang ada saat ini [6].
3. Mitigasi serangan

   Pada poin ini, mengacu kepada bagian II.2 di mana terdapat variasi serangan yang mengeksploitasi kelemahan JavaScript dan Node.js. Untuk mendeteksi serangan-serangan ini, digunakan cara analisis kode statis (*static code analysis*). Alat yang digunakan adalah

alat analisis kode T.J. Watson Libraries for Analysis (WALA), yang merupakan *libraries packages* untuk menganalisis kode JavaScript dan kode biner Java [6][16].

3.1. *Global leakage*

Kriteria identifikasi pada serangan ini adalah serangan ini memanggil suatu fungsi yang "bocor" (*leaking function*) yang mana salah satu parameter dari fungsi tersebut adalah *global variable*. Semisal pada implementasi protokol HTTP, di mana aplikasi butuh untuk melakukan pemanggilan API kepada penyedia layanan. Analisis yang dapat dilakukan adalah dengan mengidentifikasi pemanggilan HTTP *request* yang melakukan `write` *method*. Dari situ, digunakan analisis alur data (*data flow analysis*), dan dilacak apakah parameter dari *method* tersebut menggunakan *global variable* [6].

3.2. *Global manipulation*

Ada dua variasi dalam serangan ini, yaitu *global object* mengalami penimpaan (*overwriting*) atau modifikasi terhadap *properties* dari *object* tersebut. Untuk variasi yang pertama, dapat dengan mudah dideteksi karena dalam WALA, *overwriting* suatu *global variable* akan direpresentasikan oleh satu instruksi IR (Intermediate Representation, semacam bentuk bahasa sederhana yang digunakan dalam analisis) yaitu `AstGlobalWrite`. Apabila ada instruksi yang merupakan tipe dari `AstGlobalWrite` dan variabel tersebut merupakan *global variable* bawaan dari JavaScript, maka dapat dipastikan bahwa ditemukan sebuah serangan [6][16].

Ada dua kriteria yang harus dipenuhi untuk serangan yang kedua, yaitu minimal satu *property* dari sebuah variabel dimodifikasi dan variabel tersebut terhubung dengan satu dari 32 *global variables* JavaScript. Analisis yang dilakukan adalah dengan cara mengidentifikasi seluruh instruksi IR `AstGlobalRead` pada 32 *global variables* tersebut dan mempropagasikannya pada *module* yang digunakan untuk mengidentifikasi variabel yang terhubung. Lalu, kita dapat identifikasi instruksi IR `JavaScriptPropertyWrite` dari variabel yang terhubung tersebut [6][16].

3.3. *Local manipulation*

Kriteria identifikasi dari serangan ini adalah suatu *dependency* dimuat menggunakan fungsi `require` di dalam *scope* suatu modul dan dimanipulasi dalam *scope* modul yang lain. Untuk mengidentifikasi serangan ini, pertama kita identifikasi seluruh pemanggilan fungsi `require`. Lalu, kita lakukan analisis alur data terhadap *objects* yang sudah dimuat dan dilacak instruksi IR `JavaScriptPropertyWrite`

dan `SSAPutInstruction`. Dari situlah dapat diidentifikasi manipulasi suatu obyek dan *properties*-nya [6][16].

3.4. *Dependency-tree manipulation*

Kriteria identifikasi dari serangan ini adalah suatu *interface* dari sebuah *dependency* dimanipulasi dalam *scope* modul yang lain tanpa memuatnya atau modul *dependency* tersebut digantikan dengan *file* lain. Untuk mendeteksi serangan ini, dilakukan identifikasi terhadap seluruh pemanggilan *statement* JavaScript yaitu `require.cache`. Semua obyek yang dihasilkan oleh pemanggilan ini akan dilacak menggunakan analisis alur data untuk dicek manipulasinya [6].

4. Meminimalisir penggunaan *trivial packages*

Terdapat kelebihan dan kekurangan dalam menggunakan *trivial packages* (bagian II.3). Namun, penggunaan *trivial packages* untuk mendukung *code reusability* secara berlebihan tidaklah baik. Menurut Shailesh Kulkarni, pengembangan kode yang harus dimodifikasi secara reguler haruslah dilakukan oleh pengembang aplikasi itu sendiri, tidak terlalu bergantung pada *dependency* luar yang bersifat sepele (*trivial*). *Code reusability* harus dilakukan hanya apabila dibutuhkan dan dilakukan secara hati-hati. Penggunaan *trivial packages* yang tidak benar akan menimbulkan masalah pada aplikasi di kemudian hari dan berdampak pada kredibilitas aplikasi. Beberapa contoh yang pernah mengalami masalah karena *trivial packages* adalah pada aplikasi Netflix dan Facebook [17][18].

5. Semantic versioning

Kelemahan pada *technical lag* (bagian II.4) dapat diminimalisir dengan cara pengadopsian *version numbering* atau *semantic versioning*. *Semantic versioning* adalah cara penomoran versi dengan mengikuti aturan MAJOR.MINOR.PATCH sesuai dengan pada spesifikasi *semantic versioning*. Hal ini dilakukan untuk menghindari *version lock* dalam sistem versi *dependency* yang juga disebut sebagai *dependency hell* [4][19].

Aturan penomoran versi ini inkremen sebagai berikut:

a. Versi MAJOR ketika terdapat perubahan API yang tidak kompatibel.
b. Versi MINOR ketika terdapat penambahan fungsionalitas yang tidak kompatibel ke versi sebelumnya.
c. Versi PATCH ketika terdapat perbaikan *bug* yang kompatibel ke versi sebelumnya.

Sebanyak setengah dari *packages* dependen terpengaruh oleh kerentanan keamanan pada *packages* di atasnya. Sebagian besar dari *packages* dependen yang terpengaruh

tidaklah diperbarui, walaupun terdapat perbaikan pada *packages* di atasnya. Konstrain *dependency* yang tidak sesuai atau terlalu ketat dan *packages* yang tidak dirawat adalah penyebab utama dari kerentanan keamanan yang terjadi pada *packages* yang ada walaupun terdapat perbaikan. Hal ini tetap perlu diperhatikan walaupun sebenarnya 73,3% *packages* yang usang tidak menggunakan kode yang rentan secara keamanan [1][2].

Apabila konstrain *dependency* mengikuti aturan *semantic versioning* yang memungkinkan pembaruan secara otomatis terhadap perubahan kompatibel dari veri sebelumnya, maka proporsi dari *dependency* rilis yang mengalami *technical lag* akan berkurang sebanyak 17,7% dari total *packages* di NPM [3].

IV. KESIMPULAN

Kerentanan keamanan pada *dependency* JavaScript yaitu pada Node Package Manager (NPM) merupakan hal yang penting yang harus diwaspadai oleh seluruh pengguna *packages* pada aplikasi Node.js. Ada kelemahan yang ditimbulkan akibat *syntax* dan pemilihan desain pengembang JavaScript dan Node.js, kelemahan dari penggunaan *trivial packages*, kelemahan yang disebabkan oleh *technical lag*, serta jenis-jenis serangan yang dapat ditimbulkan berdasarkan kelemahan tersebut.

Akan tetapi, ada beberapa cara yang dapat dilakukan pengembang untuk memitigasi dan mencegah kerentanan keamanan tersebut dieksploitasi. Dengan melakukan *code review*, mitigasi serangan dengan alat analisis WALA, meminimalisir penggunaan *trivial packages*, dan menerapkan *semantic versioning*, akan membuat aplikasi Node.js dan JavaScript lebih aman terhadap masalah pada keamanan *dependency*.

REFERENSI

[1] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto and A. Ihara, "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, 2018, pp. 559-563.

[2] A. Decan, T. Mens and E. Constantinou, "On the Impact of Security Vulnerabilities in the npm Package Dependency Network," 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, 2018, pp. 181-191.

[3] A. Decan, T. Mens and E. Constantinou, "On the Evolution of Technical Lag in the npm Package Dependency Network," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, 2018, pp. 404-414.

[4] E. Wittern, P. Suter and S. Rajagopalan, "A Look at the Dynamics of the JavaScript Package Ecosystem," 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, 2016, pp. 351-361.

[5] B. Pfretzschner and L. b. Othmane, "Dependency-Based Attacks on Node.js," 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, 2016, pp. 66-66.

[6] Brian Pfretzschner and Lotfi ben Othmane, "Identification of Dependency-based Attacks on Node.js," In Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES '17), Association for Computing Machinery, New York, NY, USA, Article 68, 2017, pp. 1–6.

[7] A. Zerouali, V. Cosentino, T. Mens, G. Robles and J. M. Gonzalez-Barahona, "On the Impact of Outdated and Vulnerable Javascript Packages in Docker Images," 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 619-623.

[8] Rabe Abdalkareem. 2017. Reasons and drawbacks of using trivial npm packages: the developers' perspective. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017). Association for Computing Machinery, New York, NY, USA, 1062–1064.

[9] "About Node.js," Node.js, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://nodejs.org/en/about/.

[10] "Developer Survey Results 2019," Stack Overflow, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://insights.stackoverflow.com/survey/2019.

[11] "npm (software)," Wikipedia, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://en.wikipedia.org/wiki/Npm_(software).

[12] "Ride Down into JavaScript Dependency Hell," AppSignal, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://blog.appsignal.com/2020/04/09/ride-down-the-javascript-dependency-hell.html.

[13] “npm Dependency Graph,” Anvaka, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://npm.anvaka.com.

[14] “JavaScript Global Variable,” JavaTpoint, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://www.javatpoint.com/javascript-global-variable.

[15] “JavaScript,” MDN web docs, Diakses: 22 Mei 2020. [Daring]. Tersedia pada: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[16] “WALA Tools in JavaScript,” T. J. Watson Libraries for Analysis, Diakses: 26 Mei 2020. [Daring]. Tersedia pada: http://wala.sourceforge.net/wiki/index.php/Main_Page.

[17] “Code Reuse – Why too much is too bad?,” e-Zest, Diakses: 26 Mei 2020. [Daring]. Tersedia pada: https://blog.e-zest.com/code-reuse-why-too-much-is-too-bad.

[18] “How one programmer broke the internet by deleting a tiny piece of code,” Quartz, Diakses: 26 Mei 2020. [Daring]. Tersedia pada: https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code/.

[19] “Semantic Versioning 2.0.0,” SemVer, Diakses: 26 Mei 2020. [Daring]. Tersedia pada: https://semver.org/.