



---

**FACULDADE DE TECNOLOGIA DE AMERICANA**  
**Curso Superior de Tecnologia em Segurança da Informação**

Douglas Mariano Valero

**Negação de Serviço em Aplicações Node.js**

**Americana, SP**  
**2019**



---

**FACULDADE DE TECNOLOGIA DE AMERICANA**  
**Curso Superior de Tecnologia em Segurança da Informação**

Douglas Mariano Valero

**Negação de Serviço em Aplicações Node.js**

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação, sob a orientação do Prof. (o) Especialista Marcus Vinícius Lahr Giraldi.

Área de concentração: Segurança da Informação

**Americana, SP.**

**2019**

**FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS**  
**Dados Internacionais de Catalogação-na-fonte**

V256n VALERO, Douglas Mariano

Negação de serviço em aplicações Node.js. / Douglas Mariano Valero. – Americana, 2019.

62f.

Monografia (Curso Superior de Tecnologia em Segurança da Informação)  
- - Faculdade de Tecnologia de Americana – Centro Estadual de Educação  
Tecnológica Paula Souza

Orientador: Prof. Esp. Marcus Vinícius Lahr Giraldi

1 Segurança em sistemas de informação, I. GIRALDI, Marcus Vinícius  
Lahr II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de  
Tecnologia de Americana

CDU: 681.518.5

---

**Faculdade de Tecnologia de Americana**

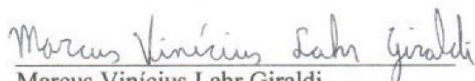
Douglas Mariano Valero

**Negação de Serviço em Aplicações Node.js**

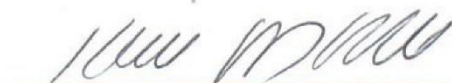
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Segurança da Informação pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana.  
Área de concentração: Segurança da Informação

Americana, 15 de junho de 2019.

**Banca Examinadora:**



Marcus Vinícius Lahr Giraldi  
Especialista  
CEETEPS/Faculdade de Tecnologia - FATEC/Americana



Juliane Borsato Beckedorff Pinto  
Especialista  
CEETEPS/Faculdade de Tecnologia - FATEC/Americana



Edson Roberto Gaseta  
Mestre  
CEETEPS/Faculdade de Tecnologia - FATEC/Americana

## **AGRADECIMENTOS**

Primeiramente a Deus por todas graças que me concedeu.

A todos meus familiares e amigos pelo apoio em cada etapa da minha vida.

Ao meu orientador Prof. Esp. Marcus Vinícius Lahr Giraldi pela paciência, dedicação e incentivo que muito ajudaram na realização deste trabalho.

A professora da disciplina de trabalho de graduação, Dra. Maria Cristina Aranda pelo acompanhamento e ajuda no trabalho.

A todos professores e colegas da FATEC Americana pela convivência, amizade e contribuição na minha formação.

## RESUMO

Este trabalho apresenta algumas das principais vulnerabilidades de negação de serviço em aplicações Node.js. São apresentados conceitos fundamentais sobre segurança da informação, aplicações web, Node.js e negação de serviço. Estuda-se o funcionamento do Node.js, principalmente do *loop* de eventos, programação síncrona e assíncrona, e como utilizar esses elementos de forma adequada, evitando-se a negação de serviço. Apresentam-se o conceito de bibliotecas ou pacotes em Node.js, como utilizá-los de forma segura, e alguns exemplos de pacotes dedicados a segurança. Discute-se um tipo de negação de serviço específico, causado por expressões regulares mal formuladas. Destacam-se boas práticas em Node.js, principalmente focadas em evitar negação de serviço. E conclui-se o trabalho destacando-se a importância de conhecer e combater vulnerabilidades de negação de serviço em aplicações Node.js.

**Palavras Chave:** Negação de Serviço. Node.js. Segurança da Informação.

## **ABSTRACT**

This paper presents some of the major denial of service vulnerabilities in Node.js applications. Key concepts on information security, web applications, Node.js and denial of service are presented. The operation of Node.js is studied, mainly of the event loop, synchronous and asynchronous programming, and how to use these elements properly, avoiding the denial of service. The concept of libraries or packages in Node.js, how to use them in a safe way, and some examples of packages dedicated to security are presented. We discuss a specific type of denial of service caused by poorly worded regular expressions. Good practices in Node.js are emphasized, mainly focused on avoiding denial of service. The paper concludes by highlighting the importance of knowing and combating denial of service vulnerabilities in Node.js applications.

**Keywords:** Denial of Service. Node.js. Information Security.

## LISTA DE FIGURAS

Figura 1 - Diagrama de um servidor <i>multi-threaded</i> bloqueante.....	19
Figura 2 - Diagrama de um servidor <i>single-threaded</i> não bloqueante.....	20
Figura 3 - Diagrama do <i>loop</i> de eventos.....	21
Figura 4 - Lendo um arquivo usando <i>buffer</i> .....	27
Figura 5 - Lendo um arquivo usando <i>stream</i> .....	28
Figura 6 - Arquivo <i>zipSync.js</i> .....	29
Figura 7 - Arquivo <i>zipAsync.js</i> .....	29
Figura 8 - Arquivo <i>zipStream.js</i> .....	30
Figura 9 - Execução das três implementações da aplicação.....	31
Figura 10 - Comando <i>npm audit</i> em uma aplicação com vulnerabilidades conhecidas.....	36
Figura 11 - Comando <i>npm audit</i> em uma aplicação sem vulnerabilidades conhecidas.....	36
Figura 12 - Comando <i>snyc test</i> em uma aplicação com vulnerabilidades conhecidas.....	37
Figura 13 - Comandos <i>snyc test</i> e <i>snyc monitor</i> em aplicação sem vulnerabilidades conhecidas.....	38
Figura 14 - Exemplo de uso da biblioteca <i>express-rate-limit</i> .....	40
Figura 15 - Lista de processos iniciados com o comando <i>pm2 start</i> .....	41
Figura 16 - Exemplo de uso do pacote <i>Helmet</i> .....	43
Figura 17 - Configuração padrão do <i>Helmet</i> .....	44
Figura 18 - Exemplo de uso de expressão regular (arquivo <i>regex.js</i> ) .....	47
Figura 19 - Resultado da execução do arquivo <i>regex.js</i> .....	48
Figura 20 - Exemplo de aplicação com expressão regular vulnerável.....	50
Figura 21 - Usando a aplicação com uma entrada válida.....	51
Figura 22 - Usando a aplicação com uma entrada maliciosa.....	51
Figura 23 – Arquivo <i>crono.js</i> .....	62



## LISTA DE ABREVIATURAS E SIGLAS

APM	<i>Application Performance Management</i>
CEP	Código de Endereçamento Postal
CPS	<i>Content Security Policy</i>
CPU	<i>Central Processing Unit</i>
DNS	<i>Domain Name System</i>
DDoS	<i>Distributed Denial of Service</i>
DoS	<i>Denial of Service</i>
ECMA	<i>European Computer Manufacturers Association</i>
FIFO	<i>First In First Out</i>
HTTP	<i>Hyper Text Transfer Protocol</i>
HTTPS	<i>Hyper Text Transfer Protocol Secure</i>
I/O	<i>Input/Output</i>
JS	<i>JavaScript</i>
JSON	<i>JavaScript Object Notation</i>
ms	milissegundos
REDOS	<i>Regular Expression Denial of Service</i>
regex	<i>Regular Expressions</i>
SQL	<i>Structured Query Language</i>
URL	<i>Uniform Resource Locator</i>
XSS	<i>Cross-Site Scripting</i>

## SUMÁRIO

INTRODUÇÃO .....	10
1 CONCEITOS BÁSICOS .....	13
1.1 SEGURANÇA DA INFORMAÇÃO .....	13
1.2 APLICAÇÕES WEB .....	14
1.3 NODE.JS .....	15
1.4 NEGAÇÃO DE SERVIÇO .....	16
2 O LOOP DE EVENTOS .....	18
2.1 BLOQUEANTE X NÃO BLOQUEANTE .....	18
2.2 O QUE É O LOOP DE EVENTOS .....	20
2.3 SÍNCRONO X ASSÍNCRONO .....	24
2.4 STREAMS .....	26
2.5 EXEMPLO PRÁTICO .....	29
3 BIBLIOTECAS .....	33
3.1 EXPRESS-RATE-LIMIT .....	39
3.2 PM2 .....	41
3.3 HELMET .....	42
3.4 VALIDAÇÃO .....	45
4 EXPRESSÕES REGULARES .....	47
5 BOAS PRÁTICAS .....	54
CONSIDERAÇÕES FINAIS .....	57
REFERÊNCIAS BIBLIOGRÁFICAS .....	59
APÊNDICE A .....	62

## INTRODUÇÃO

Node.js é uma plataforma de desenvolvimento construída em cima do motor de *JavaScript* do Google Chrome. *JavaScript* é a linguagem de programação padrão que os navegadores atuais utilizam principalmente para criar interações entre o usuário e a página web. A plataforma Node.js permite que programas escritos em *JavaScript* sejam executados fora de um navegador, assim é possível criar aplicações completas escritas em *JavaScript*. O Node.js é usado principalmente para o desenvolvimento de aplicações web escaláveis. Foi lançado em 2009 e sua popularidade só aumentou desde então. Porém o crescimento da popularidade também atraiu *hackers* mal-intencionados.

Quando um programador cria uma aplicação, existe a possibilidade de a mesma conter vulnerabilidades provenientes da própria linguagem ou da plataforma utilizada. Por exemplo algumas linguagens, como *JavaScript* e SQL, não interpretam caracteres especiais automaticamente, o que pode criar uma brecha para injeção de código malicioso. Por isso é muito importante que o programador conheça todas vulnerabilidades de sua ferramenta de trabalho e como combatê-las no momento de escrita do código.

Este trabalho apresenta as vulnerabilidades de negação de serviço mais comuns em Node.js e o que é possível fazer na hora de escrever o código para minimizar essas vulnerabilidades.

No primeiro capítulo estabelecem-se conceitos essenciais para o trabalho, como por exemplo: *JavaScript*, Node.js, Vulnerabilidades, Ameaças, Segurança da Informação, Disponibilidade, Negação de Serviço, entre outros.

Em seguida no segundo capítulo fala-se sobre a importância de escrever códigos que não bloqueiam o *loop* de eventos em aplicações Node.js, e como a falha em fazer o mesmo pode causar negação de serviço em uma aplicação.

No terceiro capítulo discute-se os pontos positivos e negativos das bibliotecas disponíveis para Node.js, os cuidados necessários com elas e como identificar quais bibliotecas apresentam vulnerabilidades conhecidas.

Em seguida no capítulo quatro fala-se sobre vulnerabilidades de negação de serviço causada por expressões regulares mal formuladas.

Por fim no quinto capítulo apresenta-se outras boas práticas para mitigar vulnerabilidades no código de aplicações Node.js.

O problema que este trabalho tratou é o fato de desenvolvedores muitas vezes não dedicarem a devida atenção ou não saberem como combater vulnerabilidades, e criam códigos com várias brechas que podem ser exploradas por pessoas mal-intencionadas e algumas vezes até por usuários legítimos sem intenção, causando danos a organização responsável pela aplicação.

A pergunta que este trabalho procurou responder é: como é possível mitigar as vulnerabilidades de negação de serviço em uma aplicação Node.js no momento em que o código é escrito?

A hipótese do trabalho foi conhecer quais são as principais vulnerabilidades de negação de serviço em aplicações Node.js, como combatê-las, e ajudar os desenvolvedores a criarem aplicações mais seguras. Beneficiando usuários, clientes e aos próprios desenvolvedores.

O objetivo geral foi apresentar algumas vulnerabilidades de negação de serviço mais comuns e mostrar como é possível combatê-las na fase de desenvolvimento de aplicações Node.js.

Os objetivos específicos deste trabalho foram: relacionar vulnerabilidades de negação de serviço, explicando como elas podem ser exploradas e os danos que podem causar, a fim de mostrar a importância de combatê-las; Dar soluções para o desenvolvimento de aplicações protegidas contra tais vulnerabilidades, ensinando técnicas, conceitos e apresentando bibliotecas, módulos ou pacotes desenvolvidos especificamente para combater essas vulnerabilidades; Conscientizar principalmente desenvolvedores de aplicações Node.js de que é preciso conhecer e reduzir as vulnerabilidades de suas aplicações, e lembrar que além disso existem outras atividades que devem ser feitas para aumentar a segurança da aplicação.

Node.js é uma tecnologia relativamente nova que ganhou popularidade rapidamente, por esse motivo o número de desenvolvedores que realmente tratam a segurança de suas aplicações com a devida importância ainda é pequeno. Portanto existe uma necessidade de aumentar a conscientização sobre a importância de tentar reduzir as possíveis vulnerabilidades de uma aplicação.

O método utilizado foi a pesquisa bibliográfica, realizada em livros e sites relacionados a segurança de aplicações web, Node.js, e combate a vulnerabilidades.

## 1 CONCEITOS BÁSICOS

Neste capítulo explica-se de forma sucinta alguns conceitos básicos úteis para o entendimento deste trabalho. Esses conceitos são: segurança da informação, disponibilidade, integridade, confidencialidade, vulnerabilidade, ameaça, *frontend*, *backend*, aplicação web, *JavaScript*, Node.js e negação de serviço.

### 1.1 SEGURANÇA DA INFORMAÇÃO

Segundo Peltier (2014) o objetivo da segurança da informação é proteger recursos importantes de uma organização, não somente as informações, mas também recursos físicos, financeiros, legais, funcionários, reputação, entre outros. O autor ainda ressalta que os objetivos da segurança da informação devem se alinhar com os objetivos da empresa, deve ajudar a alcançá-los não atrapalhar. E também todos níveis da organização devem se preocupar e se conscientizar sobre aspectos relacionados à segurança da informação.

Integridade, confidencialidade e disponibilidade são os três principais conceitos de segurança da informação, que são detalhados a seguir.

Whitman e Mattord (2011) dizem que uma informação tem integridade quando ela é inteira, completa e não corrompida. Segundo eles a informação deixa de ser íntegra quando exposta à corrupção, danos, destruição ou outra interrupção de seu estado autêntico. Os autores ainda explicam que uma forma de verificar a integridade de uma informação é o uso de algoritmos *Hash* que geram um valor único para um arquivo e caso ocorra qualquer modificação no mesmo a *Hash* é alterada.

O Tribunal De Contas da União (2012) define confidencialidade como “[...] garantia de que somente pessoas autorizadas tenham acesso às informações armazenadas ou transmitidas por meio de redes de comunicação.” Segundo Whitman e Mattord (2011) algumas das medidas que podem ser tomadas para proteger a confidencialidade da informação são: classificar o nível de confidencialidade da informação, armazenar as informações em locais seguros, aplicar políticas de segurança, educar os usuários e os responsáveis pela informação.

O último dos três principais conceitos de segurança da informação é a disponibilidade:

Consiste na garantia de que as informações estejam acessíveis às pessoas e aos processos autorizados, a qualquer momento requerido, durante o período acordado entre os gestores da informação e a área de informática. Manter a disponibilidade de informações pressupõe garantir a prestação contínua do serviço, sem interrupções no fornecimento de informações para quem é de direito. (Tribunal de Contas da União, 2012)

Disponibilidade é o principal conceito de segurança da informação para este trabalho, pois está diretamente relacionada com negação de serviço.

Vulnerabilidade é um ponto fraco ou falha em um sistema ou mecanismo de proteção que o abre para ataques ou danos. (Whitman; Mattord, 2011). Sendo vulnerabilidade um ponto fraco ou uma falha é possível fortalecer esse ponto ou corrigir tal falha. Neste trabalho estudam-se vulnerabilidades que criam oportunidades para ataques de negação de serviço e buscam-se soluções para reduzir essas falhas ou pontos fracos.

Segundo Whitman e Mattord (2011) uma ameaça é uma categoria de pessoas, objetos ou outras entidades que podem causar danos a um recurso da organização. Eles ainda definem um agente de ameaça como um elemento específico de uma ameaça, por exemplo *hackers* são uma ameaça, enquanto um *hacker* específico é um agente da ameaça. Os autores ainda ressaltam que ameaças sempre estarão presentes, e podem ter o propósito de atingir um alvo determinado ou não serem direcionadas especificamente à um alvo, mas sim a qualquer organização que apresente a vulnerabilidade explorada pelo agente da ameaça.

## 1.2 APLICAÇÕES WEB

*Frontend* é a parte de uma aplicação que é responsável por interagir com o cliente. No caso de uma aplicação web o *frontend* é o código que é interpretado pelo navegador do cliente, que formata e gera uma visualização para o cliente, com elementos que ele pode interagir, por exemplo um formulário que pode ser preenchido.

*Backend* é a parte da aplicação que é executada no servidor, geralmente é responsável por regras de negócio, controles de acesso, manipular informações de um banco de dados, autenticação e segurança. Usando o mesmo exemplo citado no *frontend*, quando um usuário envia o formulário é o *backend* que é responsável por validar, registrar e processar as informações.

Aplicação Web é um sistema de informação ambientado na Web, ou seja, é uma aplicação cujo *backend* reside em servidor web e o *frontend* é interpretado em um programa de acesso à web, que geralmente é um navegador, mas pode ser também, por exemplo, um aplicativo de celular.

*JavaScript* é uma linguagem de programação presente na maioria dos websites atuais. Segundo Düüna (2016) *JavaScript* é uma das linguagens de programação mais incompreendidas do mundo devido a sua história. Inicialmente era chamada de LiveScript, pois foi criada com a intenção de deixar as páginas Web mais ‘vivas’, e era apenas uma linguagem para scripts simples. Recebeu o nome *JavaScript* em uma tentativa (que deu certo) de se aproveitar da fama da linguagem JAVA, porém as duas não tem nenhuma relação além dessa curiosidade histórica. Hoje é uma linguagem robusta para desenvolvimento de aplicações Web, e tem o nome de ECMAScript, pois é mantida pela organização ECMA (*European Computer Manufacturers Association*), porém ainda é referida pela maioria dos desenvolvedores e autores de livros como *JavaScript*. Düüna (2016) ainda destaca que toda ferramenta tem suas peculiaridades, e devido ao crescimento e transformação, por muito tempo não padronizados, *JavaScript* possui algumas características que devem ser evitadas ou usadas com cautela seguindo boas práticas para se evitar vulnerabilidades de segurança e outros problemas em aplicações.

### 1.3 NODE.JS

Segundo Düüna (2016) Node.js é uma plataforma desenvolvida a partir do interpretador de *JavaScript* do Google Chrome: V8, para interpretar códigos escritos em *JavaScript* do lado do servidor (*backend*), o que tornou possível criar aplicações web totalmente escritas na linguagem *JavaScript*. O autor ainda explica que Node.js estende as funcionalidades do *JavaScript* ligando-o a várias bibliotecas escritas nas linguagens de programação C e C++, e também com módulos que permitem acesso a funcionalidades do sistema operacional, manipular dados binários, e outros tipos de requisições. Permitindo assim que o Node.js acesse arquivos, execute comandos no sistema, receba e responda requisições de rede, ou seja, tudo que um servidor necessita, mas não era possível fazer apenas com *JavaScript*.

Düüna (2016) destaca algumas características importantes do Node.js no ponto de vista de segurança. Uma delas é o fato de o Node.js receber e interpretar



requisições em apenas uma *thread* (ou tarefa), ou seja, existe apenas um ponto de entrada e saída de eventos, chamado de *loop* de eventos (*event loop*). Essa característica é muito importante para segurança da informação, especialmente no quesito de disponibilidade, pois se algo bloquear o *loop* de eventos, a aplicação não consegue mais servir os clientes, gerando uma negação de serviço. Estuda-se esse aspecto mais a fundo no próximo capítulo. Outra característica destacada pelo autor é o rico gerenciador de bibliotecas *JavaScript* utilizadas que é instalado por padrão juntamente ao Node.js, o *Node Package Manager* (NPM). A vantagem do NPM é que muitas bibliotecas ajudam desenvolvedores a resolverem tarefas, acelerando assim o processo de desenvolvimento de aplicações, porém essas bibliotecas também estão sujeitas a vulnerabilidades. No terceiro capítulo deste trabalho estuda-se sobre bibliotecas e os cuidados necessários ao usá-las.

#### 1.4 NEGAÇÃO DE SERVIÇO

Um ataque de negação de serviço, *Denial of Service* (DoS), ou ataque de negação de serviço distribuído, *Distributed Denial of Service* (DDoS), é uma tentativa de fazer um recurso computacional ficar indisponível para os usuários legítimos (Rhodes-Ousley, 2013). Portanto pode-se concluir que uma vulnerabilidade de negação de serviço é uma falha ou ponto fraco de uma aplicação que permite esse tipo de ataque.

Segundo Whitman e Mattord (2011) um ataque DoS é realizado de apenas um ponto, enquanto um ataque DDoS é realizado de várias localizações ao mesmo tempo, geralmente cada um desses pontos é um sistema comprometido, infectado por algum programa malicioso que permite o controle remoto de funções como por exemplo realizar requisições a um servidor.

O'Hanley (2014) diz que um ataque de negação de serviço tem como objetivo negar ou degradar a qualidade de acesso de um usuário legítimo a um serviço ou recurso de rede. O autor classifica os ataques DoS em dois tipos: ataques de desativação de serviço e ataques de enfraquecimento de recursos. O primeiro é caracterizado por altos números de requisições, geralmente por vários clientes, um DDoS, com a intenção de atingir o limite da fila de espera para utilizar o recurso e causando a paralisação do mesmo. Já os ataques de enfraquecimento de recurso geralmente exploram uma falha lógica na aplicação fazendo com que um processo

consoma recursos do servidor durante um tempo muito grande, deixando poucos recursos e tempo para o processamento de requisições de usuários legítimos.

Mueller (2016) enfatiza que uma parte importante de um ataque DoS é requisitar uma operação complexa, como por exemplo uma busca, portanto é uma boa ideia exigir autenticação do usuário para realizar esses tipos de operações, criando uma barreira de segurança a mais contra ataques DoS.

Nahari e Krutz categorizam as soluções para combater ataques DoS em dois tipos: preventivas e reativas. Soluções preventivas impedem o ataque tomando medidas de precaução, como por exemplo: filtros, estabelecimento de limites, esconder a localização de recursos e detecção de intrusos. Já as soluções reativas são acionadas durante o ataque e geralmente tem o objetivo de determinar a origem do ataque, alguns exemplos são: marcação de pacotes, testes de conexão e coleta de dados em registros (*logs*).

## 2 O LOOP DE EVENTOS

A regra mais importante ao desenvolver aplicações Node.js é não bloquear o *loop* de eventos, portanto é preciso entender exatamente o que isso significa. Neste capítulo explica-se como o Node.js funciona, o que é o *loop* de eventos, como ele opera em aplicações Node.js, o que significa bloqueá-lo e como evitar esse bloqueio.

### 2.1 BLOQUEANTE X NÃO BLOQUEANTE

Primeiro define-se o que é uma *thread*. Segundo Teixeira (2013) uma *thread* é uma linha de processo que compartilha a memória de um processo com todas outras *threads* que existem dentro do mesmo processo. Na prática o que isso significa é que se o processador do computador é por exemplo *quad-core*, ou seja tem quatro núcleos então ele tem quatro processos executando ao mesmo tempo, e dentro de cada um desses processos podem existir várias *threads* (várias linhas de processo).

De uma maneira simplificada pode-se dizer que um bloqueio ocorre quando operações intensas (em relação ao tempo de execução) são realizadas de forma contínua (de uma vez só), ou seja, enquanto a operação não é finalizada outras tarefas pendentes não tem uma oportunidade de executar. Segundo o site oficial do Node.js existem duas motivações principais para não bloquear uma *thread* de um servidor web: a performance (requisições atendidas por segundo), que é muito maior quando a *thread* executa apenas tarefas rápidas; e a segurança, pois se for possível bloquear a *thread* com alguma requisição ou *input* malicioso, isso cria uma vulnerabilidade que pode ser explorada e causar uma negação de serviço.

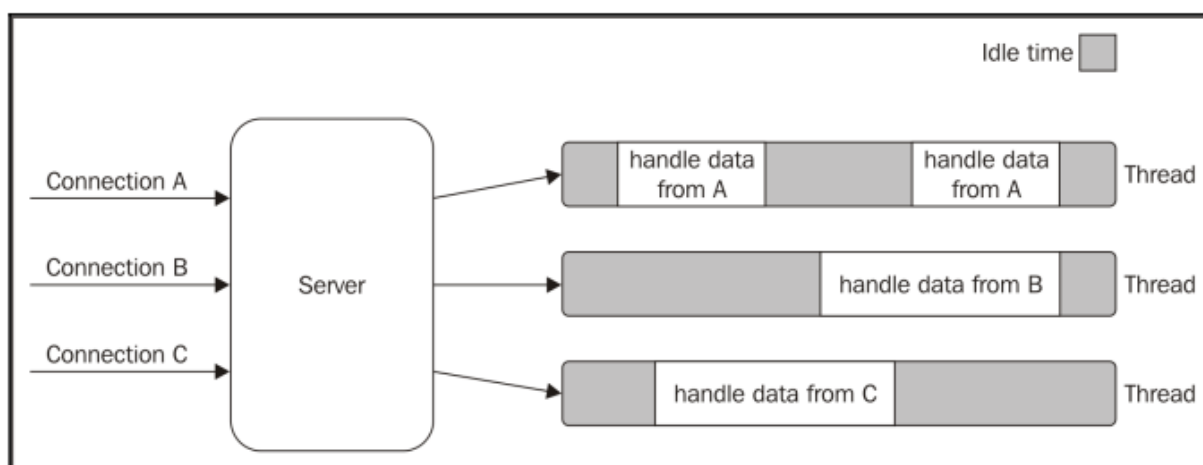
Ryan Dahl(2018) diz que quando ele criou o Node.js, seu objetivo principal era criar uma ferramenta na qual fosse possível desenvolver servidores web não bloqueantes acionados por eventos. Esse tipo de servidor tem melhor performance em aplicações focadas em entrada e saída de dados, *Input* e *Output* (I/O), pois não bloqueiam a *thread*. Até então a maioria dos servidores web eram do tipo baseado em processos, como por exemplo o popular Apache, esse tipo de servidor bloqueia a *thread*, e por isso usa múltiplas *threads* para atender vários clientes.

A princípio pode parecer que criar várias *threads* faz a aplicação ser mais rápida, porém segundo Casciaro e Mammino (2016) existem dois pontos fracos nesse tipo de servidor. O primeiro é o fato de que para cada requisição uma nova *thread* é

criada. E na maior parte do tempo essas *threads* estão em estado de espera (*idle*), aguardando I/O do cliente. Isso é ruim pois a criação de *threads* não é uma operação barata, usa memória e mudanças de contexto. E criar uma *thread* para realizar uma tarefa simples, deixando-a a maior parte do tempo em estado de espera, não proporciona benefícios, em termos de tempo, no final das contas. O segundo ponto fraco é que cada um dessas *threads* é bloqueante, ou seja, ela espera uma operação finalizar para poder iniciar outra. No caso de todas ou a maioria das *threads* do seu servidor estarem ocupados por clientes realizando operações bloqueantes, outros clientes não conseguem usar sua aplicação, gerando uma negação de serviço. Esse tipo de processamento é chamado de paralelo, pois várias operações são realizadas ao mesmo tempo em processos diferentes.

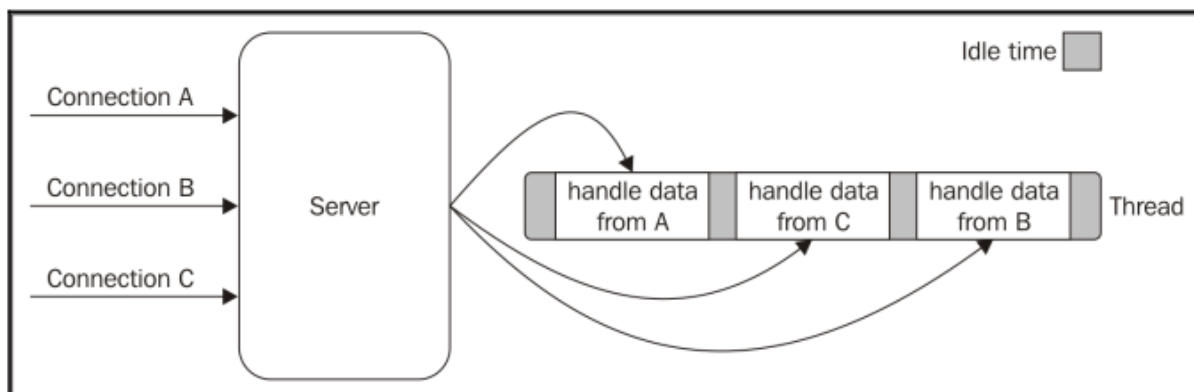
Os autores também explicam que em um servidor com apenas uma *thread* não bloqueante, essa *thread* quase nunca fica em estado de espera, sempre há operações a serem realizadas. O importante é que essas operações não sejam bloqueantes, ou seja, sejam rápidas. Nesse caso o processo é chamado de simultâneo, pois as funções são divididas em tarefas menores e são realizadas intercaladamente, minimizando o tempo de estado de espera, dando ao usuário a sensação de que foram realizadas ao mesmo tempo. As figuras a seguir ilustram servidores com várias *threads* em comparação com servidores de apenas uma *thread*.

**Figura 1: Diagrama de um servidor *multi-threaded* bloqueante**



Fonte: Casciari e Mammino (2016)

**Figura 2: Diagrama de um servidor *single-threaded* não bloqueante**



Fonte: Casciaro e Mammino (2016)

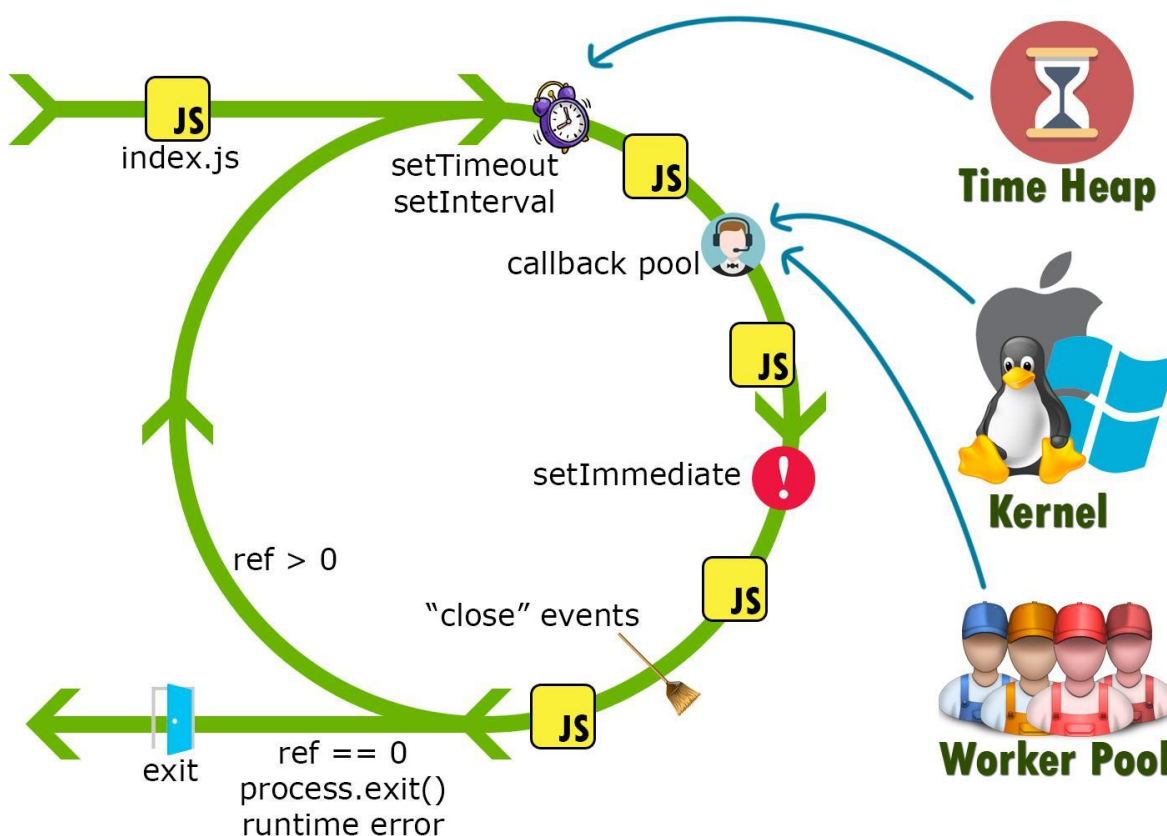
Como observa-se nas Figuras 1 e 2, no servidor *multi-threaded* bloqueante as *threads* ficam muito tempo no estado de espera (*idle*), ou seja, ocupando recursos no processador sem realizar nenhuma tarefa. Enquanto no servidor *single-threaded* não bloqueante, como pode ser observado na Figura 2, o tempo em *idle* é bem menor, ou seja, não desperdiça muitos recursos de processadores, deixando-os livres para realizar outras tarefas, como por exemplo as funções executadas pelo *kernel* e pelo o *Work Pool*, que são discutidas na próxima seção.

## 2.2 O QUE É O LOOP DE EVENTOS

O *loop* de eventos é o que permite aplicações Node.js realizarem operações não bloqueantes, ainda que o *JavaScript* use apenas uma *thread*. De forma simplificada as responsabilidades do *loop* de eventos são: redirecionar operações para o *kernel* do sistema operacional (quando possível) ou para o *work pool* (um local onde algumas tarefas específicas são executadas), agendar *timers*, receber e executar *callbacks* (são funções que executam após uma outra função ter finalizado). Como a maioria dos sistemas operacionais modernos tem um *kernel* capaz de manter várias *threads*, eles podem executar operações no plano de fundo quando necessário. Quando essas operações estão finalizadas elas voltam para o *loop* de eventos para serem executadas na aplicação Node.js.

A *libuv* é uma biblioteca desenvolvida na linguagem C, originalmente foi desenvolvida exclusivamente para possibilitar a natureza *single-threaded* não bloqueante do Node.js, hoje também é utilizada em outras plataformas. Ela é a responsável pela criação do *loop* de eventos, do *worker pool* e de várias funções assíncronas (funções não bloqueantes, discutidas na próxima seção) não disponíveis no *kernel* dos sistemas operacionais. Na figura 3 mostra-se um diagrama que representa o *loop* de eventos, baseado na palestra de Belder (2016), um dos desenvolvedores do Node.js e da biblioteca *libuv*.

**Figura 3: Diagrama do *loop* de eventos**



Fonte: Adaptado de Belder(2016)

Primeiro nota-se as três entidades fora do *loop* de eventos: *Time Heap*; *Kernel*; e *Worker Pool*. Cada uma delas é responsável por realizar determinados tipos de tarefas, segundo Belder (2016), a principal característica em comum é o número de referências, que nada mais é que o número de tarefas que aquela entidade está

realizando ou está aguardando para realizar. Observa-se a seguir quais tipos de tarefa é responsabilidade de cada uma dessas entidades.

Belder (2016) explica que o *Time Heap* tem apenas uma responsabilidade que é controlar as chamadas das funções *setTimeout* (executa uma função, uma única vez, após um período de tempo) e *setInterval* (executa uma função, várias vezes, a cada intervalo de tempo). Essas funções não são nativas da linguagem *JavaScript*, porém a maioria das plataformas que executam códigos *JavaScript* possuem uma implementação delas, como por exemplo os navegadores modernos Chrome e Firefox. O Node.js também tem sua própria implementação dessas funções. O *Time Heap* retorna uma função *callback* para o início do *loop* de eventos quando o tempo determinado for atingido.

Belder (2016) diz que o *Kernel* é responsável por realizar funções como por exemplo: servidores, sockets de conexão tcp/udp, *pipes*, entradas de terminal, resoluções DNS (*Domain Name System*), entre outras. Quando o *kernel* finaliza essas operações, ele retorna-as ao *loop* de evento, mais especificamente na fase de *callback pool*, que se explica mais adiante.

Segundo a documentação oficial do Node.js, o *Worker Pool* ou também chamado de *Thread Pool* possui uma variedade de funções assíncronas criadas pela *libuv*, as quais não existem uma similar no *kernel* que também seja assíncrona. Alguns exemplos de operações realizadas pelo *Worker Pool* são: pesquisa de DNS, operações no sistema de arquivo (acesso, leitura, escrita), alguns tipos excepcionais de *pipes*, *streams*, entre outros. Assim como o *kernel*, o *Worker Pool* retorna as operações finalizada para o *callback pool* no *loop* de eventos, para que sejam executadas na aplicação.

Na Figura 3 nota-se cinco quadrados com “JS” escrito dentro dos mesmos, segundo Belder (2016), eles representam os momentos nos quais o *loop* de evento executa códigos *JavaScript*. São nesses momentos que referências para o *Time Heap*, o *Kernel* e o *Worker Pool* podem ser criadas, através da invocação de funções implementadas pelo Node.js. Agora discute-se como cada etapa do *loop* de eventos funciona.

Belder (2016) explica que quando uma aplicação Node.js é iniciada o primeiro passo que ocorre é a leitura e execução do ponto de entrada da aplicação,

representado por *index.js* na Figura 3. No caso de uma aplicação web é nesse momento que o *kernel* recebe uma referência para começar a “ouvir” requisições HTTPS (*Hyper Text Transfer Protocol Secure*) ou HTTP (*Hyper Text Transfer Protocol*).

O autor ainda diz que após todo código inicial ser executado o *loop* entra na fase dos *timers*, a qual recebe as funções do *Time Heap* que já esperaram o tempo determinado a elas e devem ser executadas. Em seguida o *loop* de eventos executa essas funções recebidas.

A próxima fase, a fase de *pool*, é a mais importante do *loop*. Segundo a documentação oficial do Node.js, essa é a fase que recebe *callbacks* do *kernel* e do *worker pool*, ela é responsável por determinar quanto tempo deve aguardar para receber *callbacks*, gerar uma fila dessas *callbacks* recebidas e executá-las na ordem FIFO (First In First Out). Caso a fila da fase *pool* esteja vazia e não existam *callbacks* aguardando em outras fases (*timers*, *check* e *close*), o *loop* de evento fica parado na fase de *pool* até que ela (ou a fase de *timers*) receba uma *callback*.

Belder (2016) explica que após todas *callbacks* das fases *timers* e *pool* executarem, entra-se na fase de *check*, essa fase é responsável por executar *callbacks* chamadas por uma função de *timer* especial: *setImmediate*. Ela é especial justamente por que possui uma fase só para si, enquanto as outras funções de *timers* são executadas na fase de *timers*. O nome da função refere-se ao fato de ela ser executada imediatamente após a fase de *pool*.

A última fase do *loop* é a fase chamada *close*. Segundo a documentação oficial do Node.js, ela tem esse nome pois é responsável por verificar se alguma *callback* ou *socket* de comunicação foi fechado abruptamente, caso isso ocorra um evento do tipo *close* é emitido e os recursos usados são “limpos”.

Segundo Belder (2016), quando a fase *close* termina o *loop* de eventos verifica o número de referências do *Time Heap*, *Kernel* e *Worker Pool*, caso os três sejam iguais a zero o *loop* de eventos finaliza, pois se continuasse ele acabaria parando na fase de *pool* e nunca mais sairia de lá, já que para algum código ser executado seria necessário receber uma *callback* de alguma referência, e existem zero referências em todos três possíveis lugares. Existem outras duas situações em que o *loop* de eventos



finalizaria: caso haja um erro de *runtime* não lidado; caso a função *process.exit()* seja executada em algum momento.

Belder (2016) ainda explica que caso exista pelo menos uma referência em um dos três lugares possíveis (*Time Heap*, *Kernel*, *Worker Pool*) ao final da fase de *close*, o *loop* volta para a fase de *timers* e fecha o *loop*, ou seja, repete os passos descritos acima a partir da fase de *timers*. O autor enfatiza que no caso de um servidor web, como na inicialização o *kernel* recebe uma referência para começar a “ouvir” requisições HTTPS, essa referência sempre estará presente, por isso o servidor web não para mesmo quando não há nenhuma tarefa a ser executada.

## 2.3 SÍNCRONO X ASSÍNCRONO

Os conceitos de síncrono e assíncrono estão diretamente ligados a regra mais importante do Node.js: não bloquear o *loop* de eventos. Também é importante não bloquear o *Worker Pool*, pois ele é responsável por grande parte das tarefas que usam mais recursos do processador.

De maneira simples uma função síncrona realiza todas suas tarefas em sequência e de uma vez só, o que significa que enquanto ela executa, nenhuma outra função tem sua vez na *thread*. Já uma função assíncrona pode realizar uma tarefa pequena (portanto rápida), pausar, liberando a *thread* para outra função, e continuar suas tarefas mais tarde. Segundo Casciaro e Mammino (2016) a arquitetura assíncrona e o fato do Node.js ser *single-threaded*, mudou a maneira com que os desenvolvedores lidam com paralelismo. Pois ao invés de criar uma nova *thread* para cada nova tarefa, usam funções assíncronas para dar um tempo justo para cada tarefa e reduzir o tempo em estado de espera da *thread*.

Devido ao fato das funções assíncronas pararem sua execução regularmente, para que o *loop* de eventos possa checar se existem outras funções, elas levam mais tempo do que uma função síncrona para finalizar. É por isso que muitas das funções disponíveis nas bibliotecas do Node.js têm uma versão síncrona e uma versão assíncrona. Pois se essa função deve ser executada uma única vez na inicialização da aplicação, é mais vantajoso usar a versão síncrona. Porque é mais rápida, e o fato dela ser bloqueante não importa nesse caso pois na inicialização da aplicação não há requisições de clientes ainda.

Segundo Simpson (2015), funções assíncronas ajudam a criar aplicações que não bloqueiam por dois motivos: primeiro porque como elas podem ser divididas em tarefas menores que não precisam ser executadas de uma vez, é possível executar outras funções entre essas tarefas; o segundo motivo é o fato de que algumas funções precisam de um tempo em *idle*, por exemplo para aguardar o recebimento de dados de outra aplicação, e isso pode ser feito enquanto outras funções executam. Já em uma função síncrona esse tempo de espera deixaria a *thread* em estado *idle*, ou seja, bloqueando a execução de outras tarefas.

Exemplificando, em determinado momento um programa deve executar uma função que faz essencialmente duas tarefas: requisita dados de uma aplicação externa, e em seguida faz um cálculo simples com esses dados. A requisição leva 5 ms (milissegundos), e o cálculo 15 ms, porém o tempo de espera para a aplicação externa responder a requisição é de 780 ms. Se essa função for realizada de forma síncrona (bloqueante) a *thread* será ocupado por essa função por 800 ms e durante esse tempo nenhuma outra função poderá ser executada. Já se a função for assíncrona o programa executa a tarefa de requisição em 5 ms, retorna o controle do *thread* para o *loop* de eventos que pode realizar outras funções que estão na fila de espera e quando a aplicação externa retornar os dados (780 ms depois da requisição) uma *callback* com esses dados entra na fila de execução, e quando chegar sua vez é executada.

Existem várias formas de escrever funções assíncronas em *JavaScript*, apenas quatro delas serão abordadas neste trabalho, discutidas por Simpson (2015) e Casciaro e Mammino (2016): *Callbacks*, *Promises*, *Async/Await*, e *Generators*. Segundo Simpson (2015) *callbacks* são a maneira mais fundamental e mais usada para escrever código assíncrono em *JavaScript*. Define-se, simplificada, *callbacks* como funções que são passadas como parâmetro para uma outra função e executam de após a função que as contém é executada. É importante destacar que nem toda *callback* é assíncrona, isso depende de como a função foi escrita.

Simpson (2015) define *Promise* como um mecanismo facilmente repetível para encapsular e compor valores futuros. De maneira simplificada uma *Promise* retorna um estado e um valor, o estado pode ser pendente, resolvida ou recusada. Geralmente o estado inicial é pendente e não há valor, quando a tarefa realizada pela *Promise* é finalizada o estado muda para resolvida e o valor é o resultado dessa tarefa.

Caso ocorra algum erro na execução da tarefa, o estado passa a ser recusada e o valor é o erro ocorrido. As vantagens de *Promises* é que podem ser encadeadas facilmente; lidar com erros é mais fácil; a forma de escrita é mais compreensível do que *callbacks*.

Simpson (2015) explica que para escrever uma função assíncrona utilizando *Async/Await* basta utilizar a palavra reservada em *JavaScript* *async* antes da declaração de uma função, e dentro dessa função a palavra reservada *await* pode ser utilizada para pausar assincronamente a execução da função e aguardar uma tarefa, que quando finalizada retoma a execução da função.

O autor ainda diz que um *Generator* é declarado usando um asterisco (\*) após a palavra chave *function*, uma função desse tipo pode ser pausada em qualquer lugar dentro de sua declaração utilizando-se a palavra reservada *yield*. E a execução só é retomada quando o método *next()* é chamado na referência desse *Generator*.

## 2.4 STREAMS

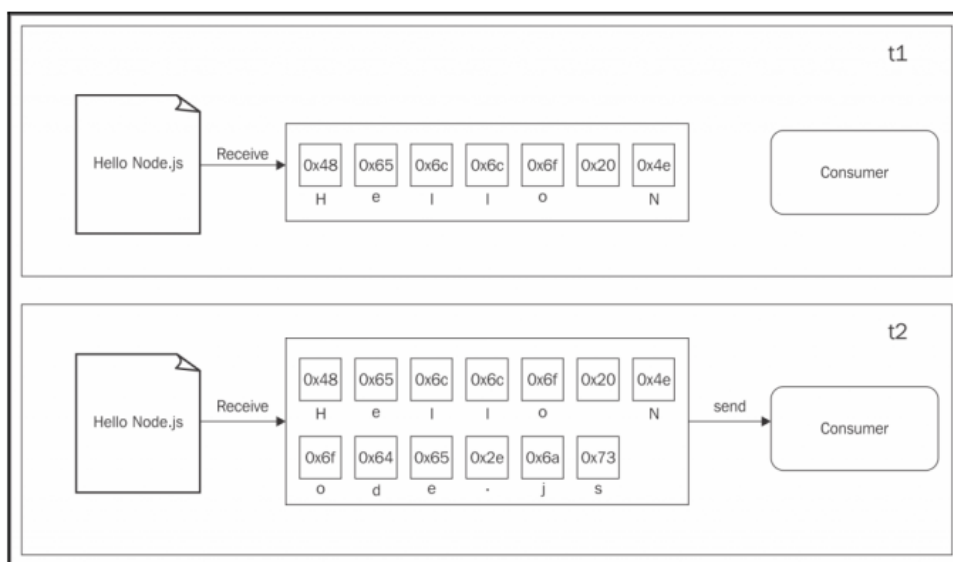
Teixeira (2013) define *stream* como uma construção abstrata que é implementada por vários objetos do Node.js. Segundo Casciaro e Mammino (2016), em uma plataforma baseada em eventos, como Node.js, a maneira mais eficiente de lidar com entrada e saída de dados (I/O) é consumir a entrada assim que estiver disponível e enviar a saída assim que estiver pronta, e é exatamente isso que *streams* fazem.

Segundo Casciaro e Mammino (2016) algumas funções assíncronas, apesar de não bloquearem o *loop* de eventos, usam um *buffer* para armazenar os resultados das tarefas realizadas por ela e só retornam esses resultados após todas as tarefas finalizarem. Existem três problemas em usar um *buffer*: enquanto todas as tarefas dessa função não finalizarem, a próxima função que usa esse resultado não pode iniciar; o *buffer* ocupa espaço na memória, caso os dados sejam muito grandes (um arquivo de vídeo por exemplo) ou vários clientes estejam usando essa função com *buffer*, o servidor pode ocupar toda sua memória, causando uma indisponibilidade; o *buffer* tem um tamanho máximo, que no Node.js é aproximadamente 1 *gigabyte*, se os dados excederem esse valor um erro de *buffer overflow* ocorre.

Em Node.js *streams* podem ser de quatro tipos: *Readable*, *Writable*, *Duplex* e *Transform*. Casciari e Mammino (2016) definem uma *readable stream* como uma representação de uma fonte de dados, por exemplo um arquivo a ser lido. Os autores também definem uma *writable stream* como uma representação de um destino de dados, como por exemplo um arquivo a ser gravado. Uma *duplex stream* é *readable* e *writable* ao mesmo tempo. Já uma *transform stream* é um tipo especial de *duplex stream*, pois nela existe uma relação estabelecida entre os dados de entrada e de saída, enquanto na *duplex stream* essa relação não é estabelecida.

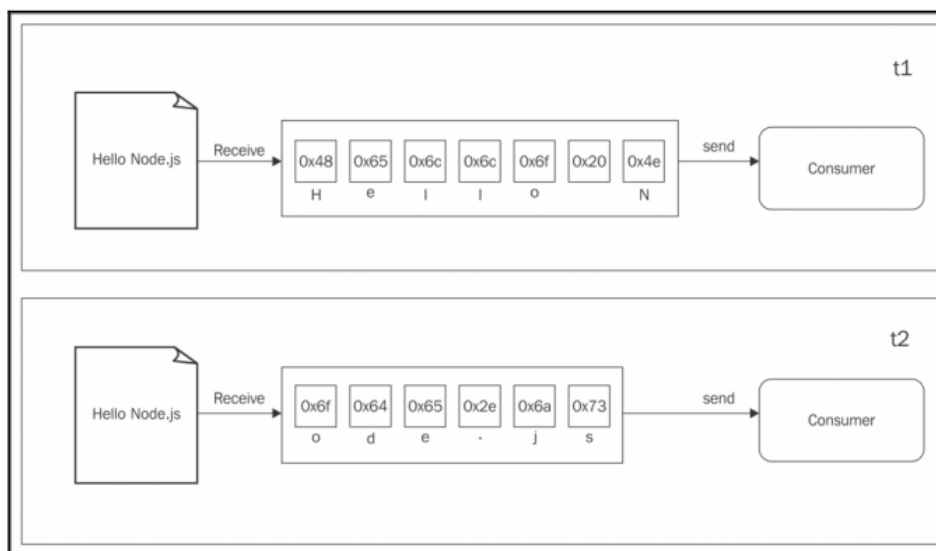
Segundo Casciari e Mammino (2016), *streams* podem ser encadeadas, semelhante às *promises*, a diferença é que no caso das *promises* uma operação da cadeia deve ser finalizada para que a próxima comece, já com *streams* cada pedaço que passa por uma operação já é enviado e pode ser processado pela próxima operação. As Figuras 4 e 5 esclarecem essa diferença.

**Figura 4: Lendo um arquivo usando *buffer***



Fonte: Casciari e Mammino (2016).

**Figura 5: Lendo um arquivo usando *stream***



Fonte: Casciari e Mammino (2016).

Observa-se no primeiro passo da Figura 4 que uma parte do arquivo é lida e armazenada no *buffer* da memória, no segundo passo a leitura do arquivo é finalizada e todo conteúdo contido no *buffer* da memória é enviado à próxima operação. Enquanto na Figura 5 nota-se que no primeiro passo, assim que parte do arquivo está na memória, já é enviada para próxima operação, e no segundo passo a outra parte do arquivo é lida e enviada para próxima operação.

Segundo Casciari e Mammino (2016), em alguns casos a operação seguinte pode ser mais lenta do que a leitura, o que causaria um acúmulo de dados na memória semelhante ao uso de *buffer*, porém *streams* possuem um mecanismo interno para evitar isso. Esse mecanismo cria um *buffer* bem pequeno, e enquanto esse *buffer* estiver cheio a leitura é pausada, assim que houver espaço no *buffer* a leitura é retomada, evitando um erro de *buffer overflow*.

## 2.5 EXEMPLO PRÁTICO

Um exemplo prático para esclarecer as vantagens e desvantagens do uso de código síncrono, assíncrono e *streams* é analisado a seguir. Neste exemplo criou-se uma aplicação de linha de comando simples, que apenas lê um arquivo, o compacta, logo em seguida o descompacta e finalmente o grava. É um bom exemplo para visualizar as diferenças em performance e ordem de execução entre código síncrono, assíncrono e *streams*.

**Figura 6: Arquivo zipSync.js**

```

1  const fs = require('fs');           //biblioteca do sistema de arquivos
2  const zlib = require('zlib');       //biblioteca de compactação
3  const crono = require('./crono.js'); //biblioteca do cronômetro
4  const file = process.argv[2];       //recebe o local do arquivo
5  const timer = process.hrtime();     //inicia o cronômetro
6
7  let buffer = fs.readFileSync(file); //lê o arquivo
8  buffer = zlib.gzipSync(buffer);     //compacta o arquivo
9  buffer = zlib.gunzipSync(buffer);   //descompacta o arquivo
10 fs.writeFileSync(file + '_copy', buffer); //grava o arquivo
11
12 crono.stop(timer);                  //para o cronômetro
13
14 console.log("Outras Tarefas");     //executa outras tarefas

```

Fonte: Autoria própria

**Figura 7: Arquivo zipAsync.js**

```

1  const fs = require('fs');           //biblioteca do sistema de arquivos
2  const zlib = require('zlib');       //biblioteca de compactação
3  const crono = require('./crono.js'); //biblioteca do cronômetro
4  const file = process.argv[2];       //recebe o local do arquivo
5  const timer = process.hrtime();     //inicia o cronômetro
6
7  fs.readFile(file, (err, buffer) => { //lê o arquivo
8      zlib.gzip(buffer, (err, buffer) => { //compacta o arquivo
9          zlib.gunzip(buffer, (err, buffer) => { //descompacta o arquivo
10             fs.writeFile(file + '_copy', buffer, err => { //grava o arquivo
11                 crono.stop(timer); //para o cronômetro
12             });
13         });
14     });
15 });
16
17 console.log("Outras Tarefas");     //executa outras tarefas

```

Fonte: Autoria própria

**Figura 8: Arquivo zipStream.js**

```

1  const fs = require('fs');           //biblioteca do sistema de arquivos
2  const zlib = require('zlib');        //biblioteca de compactação
3  const crono = require('./crono.js'); //biblioteca do cronômetro
4  const file = process.argv[2];        //recebe o local do arquivo
5  const timer = process.hrtime();      //inicia o cronômetro
6
7  fs.createReadStream(file)             //lê o arquivo
8    .pipe(zlib.createGzip())             //compacta o arquivo
9    .pipe(zlib.createGunzip())           //descompacta o arquivo
10   .pipe(fs.createWriteStream(file + '_copy')) //grava o arquivo
11   .on('finish', () => {
12     crono.stop(timer);                 //para o cronômetro
13   });
14
15 console.log("Outras Tarefas");        //executa outras tarefas

```

Fonte: Autoria própria

Nas Figuras 6, 7 e 8 tem-se respectivamente o código fonte da nossa aplicação de três formas diferentes: síncrona, assíncrona com *buffer* e por último também assíncrona, porém utilizando *streams*. Explica-se agora os códigos fonte, linha por linha.

As linhas de 1 a 5 são iguais nas três figuras. Nas linhas 1 e 2 carregam-se duas bibliotecas do Node.js que possuem funções para lidar com o sistema de arquivos e compactar arquivos respectivamente. Na linha 3 define-se uma variável que recebe o caminho do arquivo a ser lido. Na linha 4 carrega-se uma biblioteca, mostrada no apêndice A, que apenas serve para cronometrar o tempo de execução da aplicação. E na linha 5 inicia-se a contagem do tempo. Também em comum nas três figuras é a última linha que imprime no terminal a *string* “Outras Tarefas”. Isso serve para visualizar quando seriam executadas possíveis outras tarefas na aplicação, antes ou depois das operações no arquivo serem finalizadas.

Entre a quinta e a última linha dos códigos fontes, são executadas funções para ler, compactar, descompactar e gravar o arquivo, nesta ordem. Todas essas funções fazem parte das bibliotecas nativas do Node.js: *fs* e *zlib*. A diferença é que na Figura 6 as funções usadas são síncronas, na Figura 7 as funções são assíncronas com *buffer*, e na Figura 8 as funções são assíncronas com *streams*.

Observa-se agora, na Figura 9, o resultado da execução das três implementações de nossa aplicação utilizando o mesmo arquivo (de 346MB) como entrada.

**Figura 9: Execução das três implementações da aplicação**

```
[douglas@DESKTOP-1PMVMSI exemplos]$ node zipSync.js /home/douglas/Downloads/FreeBSD.iso
Tempo: 20 segundos, 145 milisegundos, 653 microsegundos, 359 nanosegundos
Outras Tarefas
[douglas@DESKTOP-1PMVMSI exemplos]$ node zipAsync.js /home/douglas/Downloads/FreeBSD.iso
Outras Tarefas
Tempo: 24 segundos, 229 milisegundos, 594 microsegundos, 168 nanosegundos
[douglas@DESKTOP-1PMVMSI exemplos]$ node zipStream.js /home/douglas/Downloads/FreeBSD.iso
Outras Tarefas
Tempo: 22 segundos, 182 milisegundos, 601 microsegundos, 811 nanosegundos
[douglas@DESKTOP-1PMVMSI exemplos]$ █
```

Fonte: Autoria própria

Nas três primeiras linhas observa-se o resultado da aplicação com funções síncronas, o mais importante a ser notado aqui é o fato da *string* “Outras Tarefas” aparecer após a finalização do cronômetro, isso significa que outras tarefas seriam bloqueadas e só executariam após o término das operações feitas no arquivo. Observa-se também que a aplicação síncrona é a mais rápida de todas como é de se esperar já que as operações no arquivo não pausam.

Da quarta linha até a sexta, tem-se a execução da aplicação assíncrona em *buffer*. Nota-se que ela é a mais lenta de todas, leva cerca de 20% a mais de tempo em relação a aplicação síncrona. Porém o destaque é o fato da *string* “Outras Tarefas” aparecer antes da finalização do cronômetro, ou seja, caso existam outras tarefas estas não precisam aguardar as operações no arquivo finalizarem para executar, o que é de acordo com a filosofia de não bloquear o *loop* de eventos.

Da sétima linha à nona linha, exibe-se o resultado da aplicação assíncrona utilizando *streams*. Assim como a aplicação assíncrona com *buffer*, as “Outras Tarefas” são executadas antes das operações no arquivo finalizarem. E também se observa que o tempo de execução é menor em relação a implementação com *buffer*, isso deve-se ao fato de que o encadeamento de *streams* permite que por exemplo a



operação de compactar inicie assim que uma pequena parte da operação de leitura está pronta.

Conclui-se então que é melhor optar por código síncrono quando o bloqueio não é um problema, por exemplo na inicialização de um servidor pois o código executa apenas uma vez e ainda não entrou no *loop* de eventos. Já para operações dentro do *loop* de eventos é melhor optar pela opção assíncrona com *streams*, quando possível, principalmente para lidar com arquivos grandes, pois além de ser mais rápida, é mais leve em relação a memória como já foi visto anteriormente.

### 3 BIBLIOTECAS

Mueller (2016) define uma biblioteca como qualquer código externo que é adicionada a sua aplicação. O autor ainda afirma que bibliotecas são puramente códigos que são baixados e executados como parte de uma aplicação. É possível usar funções diretamente, algumas vezes o código fonte está disponível para se mudar o comportamento dessas funções.

Casciaro e Mammino (2016) definem um módulo como um meio fundamental para estruturar o código de um programa. É um bloco de construção para criar aplicações e bibliotecas reusáveis chamadas de pacotes. Os autores ainda destacam que um dos princípios do Node.js é criar módulos pequenos, pois são mais fáceis de entender e reusar, simples de testar e perfeitos para compartilhar com o navegador. O termo pacote é usado para se referir a um módulo ou biblioteca de código.

O *Node Package Manager* (NPM) é o gerenciador de pacotes utilizado pelo Node.js. Foi agrupado ao Node.js bem cedo no processo de desenvolvimento, e segundo Düüna (2016) foi uma boa decisão, pois o NPM é um dos motivos do sucesso do Node.js. Com o NPM é fácil instalar, publicar e gerenciar dependências de pacotes. Existem alguns comandos e scripts que facilitam e aceleram o processo de desenvolvimento de aplicações.

Dahl (2018) diz que um de seus maiores arrependimentos quanto a criação do Node.js é a falta de atenção à segurança. Um exemplo que ele cita é o fato de bibliotecas terem acesso ao computador e a rede de seu servidor ou computador que está executando a aplicação. Esse é um dos motivos pelo qual recomenda-se executar aplicações em um *sandbox*, e nunca como *root* ou administrador.

Düüna (2016) alerta que para usar bibliotecas de terceiros é preciso ter confiança. Isso significa ter certeza que as bibliotecas foram escritas por pessoas bem-intencionadas, não pessoas que querem causar danos e prejuízos. Além disso é preciso confiar que as bibliotecas e suas dependências não possuem erros ou vulnerabilidades conhecidas. O autor mostra que um projeto pode conter dezenas e até centenas de bibliotecas e dependências, o que faz a checagem manual das mesmas impossível. Afinal a ideia é usar bibliotecas para ganhar tempo, e não gastar tempo procurando vulnerabilidades.

Segundo Dũuna (2016) existem três opções para lidar com uma escolha de pacotes: escolher pacotes populares; obscurecer pacotes; e escrever seu próprio código. A primeira opção baseia-se no fato de que quanto mais pessoas usam um pacote maior a comunidade por trás do mesmo. O que implica em um maior número de pessoas procurando vulnerabilidades e atualizando o pacote para remover as mesmas. Assim as vulnerabilidades mais óbvias muito provavelmente já foram encontradas e removidas. Além disso é comum pacotes populares terem uma ou mais empresas grandes por trás, seja desenvolvendo ou patrocinando, e geralmente elas também cedem recursos para melhorar a segurança do pacote. Porém o autor alerta que um maior número de usuários também significa uma atração maior para *hackers* mal-intencionados.

A segunda opção discutida por Dũuna (2016) é obscurecer pacotes. Isso significa expor o mínimo possível de sua aplicação, como por exemplo quais pacotes ela usa. A última opção descrita pelo autor é escrever o próprio código. Essa geralmente é a opção mais demorada e cara. Além disso o autor lembra que todos estão suscetíveis ao erro, e em uma biblioteca de terceiros é possível que o próprio desenvolvedor ou alguém que usou a biblioteca tenha encontrado e removido vulnerabilidades.

Um exemplo de pacote muito popular é o Express (<https://expressjs.com>), que fornece uma série de recursos para criação de aplicações web. É um projeto de código aberto criado em junho de 2009, e conta com mais de duzentos contribuidores diretos. Como é um dos pacotes mais antigos e mais populares, a chance de uma vulnerabilidade ter passado despercebida é muito pequena.

Dũuna (2016) diz que após a escolha dos pacotes é preciso verificar, auditar e testar os mesmos em relação a segurança. Primeiramente o autor aconselha a observar as funcionalidades do pacote que são usadas na aplicação. Se a maioria delas não for utilizada, talvez o pacote escolhido não seja o ideal. Pacotes inchados podem complicar o código da aplicação e criar dependências desnecessárias.

O autor também recomenda que se verifique o caminho percorrido pelos dados dentro do pacote. Verificar se não são manipulados de forma insegura ou maliciosa. Se os dados se originam de entradas do usuário é preciso verificar se eles estão de acordo com os padrões: não usam a função *eval* ou semelhantes (vulneráveis a

injeção de códigos); as funções não devem ser invocadas antes da validação dos dados; devem existir limites para os valores.

Düüna (2016) destaca que é comum códigos maliciosos serem inseridos em *scripts* dentro do arquivo *package.json* ou de funções temporais, principalmente *setInterval*. Pois assim esses códigos maliciosos são executados pelos scripts ou de tempo em tempo. O autor também recomenda verificar se os pacotes usam os módulos nativos de Node.js (*http*, *fs*, *net*, *tls*, *child\_process*, *cluster*, *udp*, *vm*, entre outros), pois esses módulos são usados para acessar funções do sistema operacional, sistema de arquivos, rede e outros. Por isso também é importante não executar pacotes de terceiros em contas *root* ou administrador.

Outra boa prática relatada por Düüna (2016) é manter os pacotes atualizados, pois assim vulnerabilidades recém encontradas podem ser removidas. O autor destaca ainda que é preciso testar as atualizações antes de usá-las em produção, pois elas podem criar bugs inesperados na aplicação. Dois sites que mantêm bases de vulnerabilidades de pacotes são: [npmjs.com/advisories](https://npmjs.com/advisories) e [snyk.io/vuln](https://snyk.io/vuln). Um site muito bom para procurar pacotes é o [npmjs.com](https://npmjs.com) que avalia os pacotes em relação à popularidade, manutenção e qualidade.

É claro que entrar nos sites acima citados e checar um por um dos pacotes usados em sua aplicação, e todas dependências deles, consumiria muito tempo. Por isso existem ferramentas para realizar essa checagem de forma rápida. O NPM conta com uma ferramenta padrão que verifica os pacotes vulneráveis, basta digitar o comando *npm audit* em um terminal aberto na pasta que contém seu projeto. As Figuras 10 e 11 mostram respectivamente o resultado desse comando em uma aplicação que possui um pacote com vulnerabilidades conhecidas, e em uma aplicação livre de pacotes com vulnerabilidades conhecidas.

**Figura 10: Comando *npm audit* em uma aplicação com vulnerabilidades conhecidas**

```
[douglas@DESKTOP-1PMVMSI glicon (copy)]$ npm audit
```

```

      === npm audit security report ===


```

Manual Review	
Some vulnerabilities require your attention to resolve	
Visit <a href="https://go.npm.me/audit-guide">https://go.npm.me/audit-guide</a> for additional guidance	

<b>Moderate</b>	<b>Denial of Service</b>
Package	url-relative
Patched in	No patch available
Dependency of	url-relative
Path	url-relative
More info	<a href="https://nodesecurity.io/advisories/783">https://nodesecurity.io/advisories/783</a>

```

found 1 moderate severity vulnerability in 10944 scanned packages
1 vulnerability requires manual review. See the full report for details.

```

Fonte: Autoria própria

**Figura 11: Comando *npm audit* em uma aplicação sem vulnerabilidades conhecidas**

```
[douglas@DESKTOP-1PMVMSI glicon]$ npm audit
```

```

      === npm audit security report ===


```

```

found 0 vulnerabilities
in 10943 scanned packages

```

Fonte: Autoria própria

Na Figura 10 uma vulnerabilidade foi encontrada, o comando *npm audit* informa o nível da vulnerabilidade como moderado. Também mostra qual o tipo da vulnerabilidade, nesse caso *Denial of Service*, negação de serviço. Além disso informa qual pacote possui a vulnerabilidade, de qual outro pacote ele é dependência, o caminho para o pacote vulnerável, e fornece um endereço para maiores informações.

Também se mostra a vulnerabilidade já foi removida em alguma versão do pacote, nesse caso ainda não há um *patch* disponível.

A Snyk, empresa focada em encontrar e solucionar vulnerabilidades em pacotes, também possui uma ferramenta para encontrar pacotes vulneráveis em sua aplicação. Primeiro é preciso instalar a ferramenta usando *npm install -g snyk*. É necessário estar autenticado no site da Snyk para utilizar essa ferramenta. Depois utiliza-se o comando *snyk test* para procurar vulnerabilidades. E também é possível usar o comando *snyk monitor*, que monitora as dependências de sua aplicação e te notifica por e-mail caso novas vulnerabilidades sejam encontradas em algum pacote usado por sua aplicação. As Figuras 12 e 13 mostram, respectivamente, exemplos desses comandos em uma aplicação com vulnerabilidades conhecidas, e uma aplicação sem vulnerabilidades conhecidas.

**Figura 12: Comando *snyk test* em uma aplicação com vulnerabilidades conhecidas**

```
[douglas@DESKTOP-1PMVMSI glicon (copy)]$ snyk test
Testing /home/douglas/Documents/Dev/glicon (copy)...
x Medium severity vulnerability found in url-relative
Description: Denial of Service (DoS)
Info: https://snyk.io/vuln/SNYK-JS-URLRELATIVE-173691
Introduced through: url-relative@1.0.0
From: url-relative@1.0.0

Organisation:    douglasmv
Package manager: npm
Target file:     package-lock.json
Open source:     no
Project path:    /home/douglas/Documents/Dev/glicon (copy)

Tested 705 dependencies for known vulnerabilities, found 1 vulnerability, 1 vulnerable path.
Run `snyk wizard` to address these issues.
```

Fonte: Autoria própria

**Figura 13: Comandos *snyk test* e *snyk monitor* em aplicação sem vulnerabilidades conhecidas**

```
[douglas@DESKTOP-1PMVMSI glicon]$ snyk test

Testing /home/douglas/Documents/Dev/glicon...

Organisation:    douglasmv
Package manager: npm
Target file:     package-lock.json
Open source:     no
Project path:    /home/douglas/Documents/Dev/glicon

✓ Tested 704 dependencies for known vulnerabilities , no vulnerable paths found.

Next steps:
- Run `snyk monitor` to be notified about new related vulnerabilities.
- Run `snyk test` as part of your CI/test.

[douglas@DESKTOP-1PMVMSI glicon]$ snyk monitor

Monitoring /home/douglas/Documents/Dev/glicon...

Explore this snapshot at https://app.snyk.io/org/douglasmv/project/0d53ff11-e006-4231-81ea-e678a7361771/history/30bd12e8-3527-4743-86b7-6578301b5775

Notifications about newly disclosed vulnerabilities related to these dependencies will be emailed to you.
```

Fonte: Autoria própria

A ferramenta *snyk* mostra algumas informações diferentes em relação ao comando *npm audit*. O principal diferencial é a capacidade de monitorar as vulnerabilidades com o comando *snyk monitor* e receber atualizações por e-mail.

Como já foi citado, Düüna (2016) recomenda o uso de pacotes de terceiros pois a chance de vulnerabilidades passarem despercebidas por eles é menor do que se o desenvolvedor escrever o próprio código. Isso é mais relevante ainda quando se fala de pacotes que resolvem problemas mais complicados relacionados a segurança, como por exemplo criptografar senhas. Por isso é importante conhecer alguns pacotes úteis relacionados a segurança. Serão explorados a seguir neste capítulo alguns desses pacotes, principalmente relacionados a negação de serviço.

### 3.1 EXPRESS-RATE-LIMIT

Segundo sua documentação (<https://github.com/nfriedly/express-rate-limit>), o *Express-Rate-Limit* é uma biblioteca para aplicações Node.js/Express usada para limitar requisições repetidas a uma aplicação. Isso é muito útil contra ataques de negação de serviço distribuídos (DDoS), pois limita a quantidade de recursos que cada máquina atacante consome.

É claro que é possível limitar requisições através de outros recursos, um *firewall* por exemplo. Porém existem algumas vantagens em limitar requisições no próprio código de sua aplicação. Uma delas discute-se no próximo parágrafo: capacidade de configurar um limite diferente para cada rota da aplicação. Outra vantagem é que muitas vezes a aplicação é hospedada em um serviço terceirizado, e não há garantia de que esse serviço configure seu firewall de maneira adequada. E ainda que a aplicação seja hospedada na própria organização, pode acontecer de alguma configuração do firewall falhar ou ser corrompida. Portanto é melhor ter limitação redundante do que contar apenas com um recurso.

A documentação do *Express-Rate-Limit* mostra também que é possível configurar a quantidade máxima de requisições em um determinado tempo, também configurável. Ele também permite configurar um limite para aplicação toda, ou separadamente para cada rota da aplicação. Assim é possível limitar as requisições de acordo com o recurso. Por exemplo para uma página de *login* talvez seja interessante limitar 5 requisições a cada 15 minutos, para evitar ataques de força bruta contra as senhas dos usuários. Já para uma página de pesquisa por exemplo, pode ser viável configurar 20 requisições a cada 5 minutos. Esse recurso fornece uma flexibilidade para o desenvolvedor que não é possível com um *firewall* por exemplo.

Na Figura 14 observa-se um exemplo de uso da biblioteca *express-rate-limit* aplicando regras diferentes para rotas diferentes. Para a rota */api* tem-se um limite de 100 requisições a cada 15 minutos. Já para rota */create-account* tem-se um limite de cinco requisições por hora. Como é possível perceber, existem algumas opções passadas à função *rateLimit*. A seguir destacam-se algumas opções e suas descrições segundo a documentação oficial da própria biblioteca:

*windowMs*: quanto tempo em milisegundos deve-se manter o registro das requisições.



*max*: número máximo de requisições antes de enviar uma resposta de erro 429.

*message*: mensagem de erro enviada ao usuário após *max* ser atingido. O padrão é “*Too many requests, please try again later*”.

*statusCode*: código de estado HTTP retornado após *max* ser excedido, por padrão é o código 429.

**Figura 14: Exemplo de uso da biblioteca *express-rate-limit***

```
const rateLimit = require("express-rate-limit");

app.enable("trust proxy"); // only if you're behind a reverse proxy
                             (Heroku, Bluemix, AWS ELB, Nginx, etc)
const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100
});
app.use("/api/", apiLimiter);

const createAccountLimiter = rateLimit({
  windowMs: 60 * 60 * 1000, // 1 hour window
  max: 5, // start blocking after 5 requests
  message:
    "Too many accounts created from this IP, please try again after an hour"
});
app.post("/create-account", createAccountLimiter, function(req, res) {
  //...
});
```

Fonte: Disponível em: <<https://github.com/nfriedly/express-rate-limit>>. Acesso em: 03 mar. 2019.

Existem outras opções não destacadas neste trabalho. É possível encontrá-las na documentação da própria biblioteca. Como foi mostrado o *express-rate-limit* é um pacote muito útil para segurança em relação a ataques de negação de serviço. O seu uso é recomendado, porém para alguns tipos de aplicação podem existir pacotes mais apropriados. Na própria documentação do pacote são discutidos estes casos, por isso recomenda-se checar qual o melhor pacote para sua aplicação.

### 3.2 PM2

Segundo sua própria documentação, PM2 Runtime é um gerenciador de processos em produção para aplicações Node.js, com um balanceador de carga embutido. Permite que as aplicações executem o tempo todo, reiniciá-las sem tempo de inatividade, e facilita algumas operações de desenvolvimento comuns.

O PM2 Runtime é gratuito e de código aberto. Existem outras duas versões do PM2 pagas: PM2 Plus e PM2 Enterprise. Essas versões pagas contam com mais funcionalidades e com suporte online.

Para instalar o PM2 Runtime basta usar o comando `npm install -g pm2`. Para iniciar uma aplicação em modo de produção basta usar o comando: `pm2 start app.js`. Sendo `app.js` o ponto de entrada de sua aplicação. Com esse comando o PM2 já mantém sua aplicação executando o tempo todo, reiniciando automaticamente em caso de *crash*, sem tempo de inatividade. E para que a aplicação reinicie automaticamente no caso da máquina em que a aplicação reside reiniciar, basta usar o comando: `pm2 startup`. Para gerenciar os processos da aplicação o PM2 cria uma lista de processos, que pode ser acessada com o comando `pm2 ls`, como mostra a Figura 15.

Figura 15: Lista de processos iniciados com o comando `pm2 start`

```
unitech@t450: ~
>>> pm2 ls
```

App name	id	mode	pid	status	restart	uptime	cpu	mem	watching
api	0	cluster	27287	online	0	2m	0%	35.5 MB	disabled
api	5	cluster	27308	online	0	2m	0%	32.4 MB	disabled
api	6	cluster	27348	online	0	2m	0%	33.7 MB	disabled
front	3	fork	27303	online	0	2m	0%	26.5 MB	enabled
healthcheck	2	fork	0	stopped	0	0	0%	0 B	disabled
mailer	4	fork	27309	online	0	2m	0%	26.1 MB	disabled
worker	1	fork	27292	online	0	2m	0%	24.9 MB	disabled

Use 'pm2 show <id/name>' to get more details about an app

Fonte: Disponível em: <<https://pm2.io/doc/en/runtime/overview/>>. Acesso em: 03 mar. 2019

Para adicionar processos a essa lista, basta usar o comando `pm2 start`, como já foi escrito anteriormente. Para remover processos usa-se o comando `pm2 delete <nome do app>`. Existem ainda outros comandos para gerenciar os processos, como por exemplo: `pm2 stop`; `pm2 reload`; `pm2 restart`.

Também é possível gerar arquivos de log facilmente com o PM2. Basta usar o comando `pm2 logs <nome do app>` para gerar logs de um processo, ou usa-se o comando `pm2 logs all`, para gerar logs de todos processos. Os arquivos de log são salvos na pasta `~/.pm2/logs`. Existem várias opções para gerenciar logs, por exemplo: criar vários arquivos e não apenas um muito grande; apagar os arquivos de log; escolher o formato do arquivo.

Uma funcionalidade muito útil do PM2, principalmente para reduzir chances de negação de serviço, é o modo *cluster*. Nesse modo o PM2 cria vários processos filhos da sua aplicação e balanceia a carga entre eles. Isso aumenta a performance e reduz tempo de inatividade. Para usar essa função basta iniciar a aplicação com o comando: `pm2 start app.js -i max`. Sendo `app.js` o ponto de entrada da aplicação, e `-i` é a opção que controla o número de instâncias. Neste caso *max* significa que o PM2 detecta automaticamente o número de CPUs (*Central Processing Unit*) disponíveis e executa quantos processos forem possíveis. Também é possível usar um número específico no lugar de *max*. Nesse caso o número de instâncias será o menor entre: o número digitado na opção; e o número de CPUs disponíveis.

Outra função interessante do PM2 é o monitoramento direto no terminal. Basta digitar o comando `pm2 monit` em um terminal. Essa tela mostra consumo de CPU e memória, logs de requisições, número de requisições por minuto, *delay* do *loop* de eventos, quantas vezes o servidor foi reiniciado, o tempo de atividade, entre outros. Além disso o PM2 tem várias outras funcionalidades para gerenciar sua aplicação e reduzir o tempo de indisponibilidade da mesma.

### 3.3 HELMET

*Helmet* é um pacote que ajuda a deixar aplicações Node.js feitas com o *framework* Express mais seguras, configurando vários cabeçalhos HTTP. Ele não é voltado especificamente para proteção contra negação de serviço, e sim para vários tipos de vulnerabilidades que podem ser exploradas caso alguns cabeçalhos HTTP estejam mal configurados. O uso do *Helmet* é recomendado por vários profissionais, o próprio site oficial do Express recomenda. Dũuna (2016) mostra como usando o *Helmet* e com apenas algumas linhas de código é possível se proteger de uma variedade de ataques, como mostra a Figura 16.

Figura 16: Exemplo de uso do pacote *Helmet*

```
var express = require('express');
var app = express();

var helmet = require('helmet');
app.use(helmet());           // Use helmet with default settings
app.use(helmet.csp({        // Use CSP with minimal settings
  defaultSrc: ["'self'"]
}));
```

Fonte: Dũüna (2016)

Observa-se que Dũüna (2016) usa a configuração padrão do *Helmet*, e além disso configura o cabeçalho Content Security Policy (CPS). O CPS define de quais origens podem ser os scripts executados na aplicação. A opção *self* determina que apenas scripts do próprio domínio da aplicação podem executar. Isso previne ataques de *Cross-Site Scripting* (XSS).

Um dos cabeçalhos mais importantes configurado pelo *Helmet* é o cabeçalho X-Powered-By. Esse cabeçalho indica qual tecnologia a aplicação usa. É uma boa prática de segurança remover esse cabeçalho. Pois *hackers* podem usá-lo para encontrar aplicações que usam uma determinada tecnologia a qual eles encontraram uma vulnerabilidade. Removendo esse cabeçalho sua aplicação fica protegida contra esses ataques em massa.

A Figura 17 mostra qual a configuração padrão do *Helmet*, usada por Dũüna (2016) em seu exemplo.

**Figura 17: Configuração padrão do *Helmet***

Module	Default?
<code>contentSecurityPolicy</code> for setting Content Security Policy	
<code>crossdomain</code> for handling Adobe products' crossdomain requests	
<code>dnsPrefetchControl</code> controls browser DNS prefetching	✓
<code>expectCt</code> for handling Certificate Transparency	
<code>featurePolicy</code> to limit your site's features	
<code>frameguard</code> to prevent clickjacking	✓
<code>hidePoweredBy</code> to remove the X-Powered-By header	✓
<code>hsts</code> for HTTP Strict Transport Security	✓
<code>ieNoOpen</code> sets X-Download-Options for IE8+	✓
<code>noCache</code> to disable client-side caching	
<code>noSniff</code> to keep clients from sniffing the MIME type	✓
<code>referrerPolicy</code> to hide the Referer header	
<code>xssFilter</code> adds some small XSS protections	✓

Fonte: Disponível em: <<https://helmetjs.github.io/>>. Acesso em: 03 mar. 2019

Ainda no tópico de cabeçalhos HTTP, mas não relacionado ao *Helmet*, em 27 de novembro de 2018 foi lançado uma atualização para o Node.js que resolveu duas vulnerabilidades de negação de serviço relacionadas a cabeçalhos HTTP. A primeira consiste no fato de que antes dessa atualização os cabeçalhos podiam ter até 80

*kilobytes*, e após a atualização só podem ter até 8 *kilobytes*. Isso era um problema pois usando uma combinação de requisições com cabeçalhos de tamanho máximo era possível fazer o servidor HTTP ser interrompido. A segunda vulnerabilidade era o fato de ser possível enviar cabeçalhos HTTP de maneira bem lenta, mantendo as conexões e os recursos alocados por um tempo muito longo, causando indisponibilidade para outros usuários legítimos da aplicação. Mais detalhes sobre essa atualização, e vulnerabilidades, podem ser encontrados no site oficial do Node.js.

### 3.4 VALIDAÇÃO

Nesta seção discute-se a importância de validar dados fornecidos por usuários a fim de evitar ataques de injeção de código. Mostra-se também algumas bibliotecas com foco em validação de dados.

Düüna (2016) define injeção de código como um ataque no qual um código malicioso é inserido na aplicação e faz o programa executá-lo. Esse tipo de ataque faz o servidor realizar algo que não é seu propósito. Isso inclui obter informações confidenciais, modificar ou danificar o servidor, entre outros. O autor ainda afirma que por existirem diversos tipos de injeção de código, este é o tipo de ataque mais usado contra aplicações web. Segundo o autor para combater esse tipo de ataque é preciso validar os dados fornecidos pelo usuário e sanear esses dados. Isso significa remover caracteres especiais, que são usados para escrever códigos.

Geralmente ataques de injeção de código são usados para obter ou excluir informações. Porém De Turckheim (2018) mostra que é possível realizar um ataque de negação de serviço utilizando injeção de código. Esse ataque consiste em utilizar uma falha no MongoDB, um banco de dados muito popular em aplicações Node.js. Essa falha é uma função específica do MongoDB que faz o servidor pausar por um determinado tempo. Ele mostra que injetando essa função em uma busca no banco de dados pode causar indisponibilidade na aplicação.

Uma biblioteca para validação de dados é a Validator.js. Ela conta com mais de cinquenta funções de validação e mais de dez funções de saneamento de *strings*. Outras duas bibliotecas de validação são Joi (<https://github.com/hapijs/joi>) e Celebrate (<https://github.com/arb/celebrate#readme>). Joi usa esquemas no formato de objetos *JavaScript* para validar dados. É muito útil para validar cabeçalhos de requisições

HTTP. Celebrate apenas facilita o uso da biblioteca Joi em aplicações construídas sobre o *framework* Express.

Conclui-se que é muito importante validar e sanear dados. Pois ataques de injeção de código podem causar diversos tipos de dano, inclusive negação de serviço. Também se mostrou que existem diversas bibliotecas especialmente desenvolvidas para validação e saneamento de dados. O uso delas pode significar a proteção de sua aplicação contra a maior parte dos ataques realizados na web.

## 4 EXPRESSÕES REGULARES

Na seção 3.4 mostrou-se a importância da validação de dados fornecidos pelo usuário. E uma das melhores ferramentas para se validar dados são as expressões regulares, também referidas como *regex* (*regular expressions*). Porém é preciso saber escrever expressões regulares seguras. Neste capítulo estuda-se o que são expressões regulares, como elas podem causar vulnerabilidades de negação de serviço, e como escrevê-las de maneira a evitar tais vulnerabilidades.

A documentação oficial do Node.js define expressões regulares como expressões que têm a função de comparar uma *string* de entrada com um padrão. Por exemplo comparar se o valor digitado é do formato de um e-mail. É claro que isso é possível usando métodos de *JavaScript* específicos para *strings*. A vantagem de expressões regulares é o fato de ser possível criar padrões muito complexos em apenas uma linha de código. Não é do escopo deste trabalho ensinar todas regras e sintaxes de expressões regulares, pois são muitas. Kantor *et al* (2019) tem um capítulo, em seu tutorial moderno de *JavaScript*, dedicado apenas a expressões regulares. Neste trabalho apenas explica-se de forma sucinta como as expressões regulares funcionam em *JavaScript*. E foca-se em como evitar vulnerabilidades que podem causar negação de serviço.

**Figura 18: Exemplo de uso de expressão regular (arquivo *regex.js*)**

```
1  const minhaRegex = /\d{5}-\d{3}/;    //regex de um CEP
2
3  const cepValido = "12345-678";      //um CEP válido
4  const cepInvalido1 = "-12345678";   //um CEP inválido
5  const cepInvalido2 = "1234-5678";   //um CEP inválido
6  const cepInvalido3 = "12345--678";  //um CEP inválido
7  /* Testando a validade das strings usando a regex */
8  console.log(minhaRegex.test(cepValido));
9  console.log(minhaRegex.test(cepInvalido1));
10 console.log(minhaRegex.test(cepInvalido2));
11 console.log(minhaRegex.test(cepInvalido3));
```

Fonte: Autoria própria



**Figura 19: Resultado da execução do arquivo *regex.js***

```
[douglas@DESKTOP-1PMVMSI exemplos]$ node regex.js
true
false
false
false
[douglas@DESKTOP-1PMVMSI exemplos]$ █
```

Fonte: Autoria própria

Na Figura 18 mostra-se um exemplo de código que usa uma expressão regular para validar o formato de um CEP (Código de Endereçamento Postal). E na Figura 19 tem-se o resultado ao executar o código do arquivo mostrado na Figura 18.

Na primeira linha da Figura 18 definiu-se a expressão regular e salvou-se a mesma em uma variável chamada *minhaRegex*. Na terceira linha salvou-se uma *string* que contém um valor no formato válido para CEP na variável *cepValido*. Nas linhas 3, 4 e 5 salvou-se outras três *strings* em outras três variáveis, dessa vez representam valores em um formato inválido para CEP. Nas linhas 8 a 11 usa-se uma função que imprime na tela *true* caso a *string* seja válida segundo a expressão regular, e *false* caso contrário. Na Figura 19 observam-se os resultados conforme esperado, *true* para a *string* válida e *false* para as três *strings* inválidas.

Agora explica-se a expressão regular. Observa-se primeiramente que se utilizam barras (/) no começo e no final, essas barras são uma maneira de definir uma expressão regular em *JavaScript*. O primeiro caractere dentro da expressão regular é um circunflexo (^) o qual determina que a expressão regular deve-se encontrar no começo da *string*. O cifrão (\$) no final é similar ao circunflexo, porém indica que a expressão deve-se encontrar no final da *string*. Neste o uso do circunflexo e do cifrão garantem que não há nada antes ou depois do CEP, sem o uso deles a terceira *string* inválida retornaria *true*, o que seria um erro. Depois tem-se \d que identifica dígitos de 0 a 9, e {5} ao lado do \d significa que devem ter cinco dígitos agrupados. O próximo caractere é um hífen (-) que simplesmente identifica o próprio hífen. E em seguida tem-se \d{3} que identifica três dígitos agrupados.

Portanto a expressão regular da Figura 18 apenas vai validar *strings* que começam com exatamente cinco dígitos, seguidos de um hífen, seguido de exatamente três dígitos no final da *string*. O que representa perfeitamente um CEP. Esse exemplo deixa claro o poder das expressões regulares. Imagine escrever esta mesma validação verificando caractere por caractere, certamente seria um código bem maior e mais difícil de compreender.

Segundo a documentação oficial do Node.js, uma expressão regular vulnerável é uma expressão regular que leva um tempo exponencial para finalizar, o que pode causar um ataque REDOS (*Regular Expression Denial of Service*). A documentação ainda indica quatro regras para evitar vulnerabilidades de negação de serviço nas expressões regulares.

A primeira regra é evitar quantificadores aninhados. Por exemplo a expressão `/(\d+)+$/` é um exemplo do que deve ser evitado. Nessa expressão os parênteses são chamados de grupo de captura. Dentro deste grupo tem-se `\d+` que representa um ou mais dígitos numéricos. E fora do grupo tem-se o caractere de adição que significa um ou mais. E logo após tem-se o caractere cifrão que representa o final da *string*. O problema com esse tipo de expressão regular ocorre quando se usa essa expressão em uma *string* composta por vários dígitos numéricos seguidos mas termina em um caractere que não é numérico. Suponha que a *string* seja “1234a”, o que acontece é que a expressão encontra o grupo “1234” que é composto por um ou mais dígitos. Em seguida a expressão verifica se encontrou um ou mais grupos de dígitos, o que é verdadeiro. Então a expressão verifica se a *string* termina em um dígito numérico, o que é falso. Assim a expressão volta ao começo e dessa vez captura dois grupos “123” e “4” mas ainda não termina com um dígito. Então no próximo passo captura as expressões “12” e “34”, depois as expressões “12”, “3” e “4”, e assim por diante. Em resumo a expressão tenta todas combinações possíveis dos grupos de dígitos mas nenhuma vai satisfazer a *regex* pois a *string* sempre termina com o caractere “a”. Agora imagine se um usuário entra com uma *string* que contenha muitos dígitos numéricos e um caractere não numérico no final. O tempo que levaria para o computador testar todos grupos seria muito grande, causando uma negação de serviço pois esses testes consomem todo processador e bloqueariam a *thread*.

Outra regra é evitar “ou” com expressões repetidas. Por exemplo `/(a|a)*/`. Similar a primeira regra, esse tipo de expressão pode causar um consumo excessivo de processador e bloquear a *thread* causando indisponibilidade.

A terceira regra é evitar usar referências a grupos de captura. Por exemplo `/(a.*)1/`. Quando se usa grupos de captura referências aos mesmos são automaticamente criadas, e podem ser usadas com `\1`, `\2`, etc. Essas referências são úteis para evitar reescrever um mesmo grupo duas vezes, porém elas exigem muito da performance do processador. Logo é melhor repetir a escrita de uma parte da expressão para não afetar a performance.

A quarta e última regra é usar o método `indexOf` quando se precisa de uma consulta simples em uma *string*. É possível usar expressões regulares para encontrar palavras em uma *string*, porém o método `indexOf` faz o mesmo e garante que sempre levará o menor tempo possível.

**Figura 20: Exemplo de aplicação com expressão regular vulnerável**

```

1  const http = require('http');           //biblioteca http
2  const url = require('url');             //biblioteca url
3
4  http.createServer((req, res) => {        //cria um web server http
5      let timer = process.hrtime();         //inicia o cronômetro
6      const regex = /(d+)$/;               //regex vulnerável
7      const number = url.parse(req.url, true).query.num; //recebe o valor passado na url
8      res.writeHead(200, {"Content-type": "text/html"}); //Tipo da resposta
9      res.write("Input: " + number);        //mostra o valor recebido
10     res.write("<br>Valido: " + regex.test(number)); //mostra se o valor é válido com a regex
11     timer = process.hrtime(timer);         //finaliza o cronômetro
12     res.write("<br>Timer: ${timer[0]} segundos, ${timer[1]} nanosegundos`"); //mostra o tempo de execução
13     res.end();                           //finaliza a resposta http
14 }).listen(3000);                         //o servidor ouve requisições na porta 3000

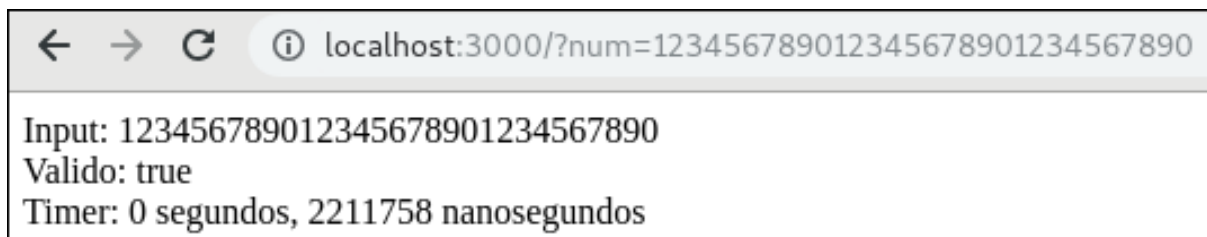
```

Fonte: Autoria própria

A Figura 20 mostra um exemplo de expressão regular vulnerável pois não obedece a primeira regra. De fato é o mesmo exemplo usado para explicar essa regra anteriormente. A expressão regular vulnerável é `/(d+)$/`. Esse código cria um servidor web que lê uma *string* passada pela URL (*Uniform Resource Locator*) e usa a expressão regular para verificar se a *string* termina em dígitos numéricos. É óbvio que não é a melhor maneira de se fazer isso, porém para demonstrar um ataque de negação de serviço explorando expressões regulares vulneráveis é um bom exemplo,

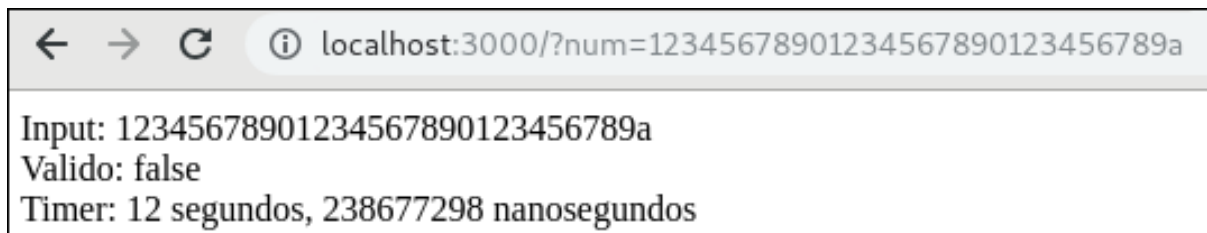
pois o código é sucinto. Após requisitar uma URL utilizando um navegador, é impresso na tela o valor da *string* de entrada, se ela foi validada ou não pela *regex*, e o tempo que o servidor levou para avaliar a *string* usando a *regex* e dar a resposta.

**Figura 21: Usando a aplicação com uma entrada válida**



Fonte: Autoria própria

**Figura 22: Usando a aplicação com uma entrada maliciosa**



Fonte: Autoria própria

As Figuras 21 e 22 mostram o resultado da utilização da aplicação mostrada na Figura 20. Na Figura 21 tem-se um valor de entrada válido de trinta caracteres, com um tempo de resposta de aproximadamente 2 milissegundos. Na Figura 22 tem-se uma entrada maliciosa que explora a *regex* vulnerável, obtendo um tempo de resposta de mais de 12 segundos. Isso é extremamente preocupante pois durante esses 12 segundos a aplicação web não pode enviar respostas para nenhum cliente, gerando uma negação de serviço.

**Tabela 1: Tempos de resposta de acordo com o tamanho da entrada**

Número de Caracteres	Tempo de Resposta (segundos)
26	0,869425535
27	1,616451710
28	3,167267293
29	6,296623415
30	12,238677298
31	24,993667302
32	50,202110160
33	100,528942781
34	199,121742321

Fonte: Autoria própria

A tabela 1 mostra o tempo de resposta da aplicação da Figura 20 para entradas de diferentes tamanhos (número de caracteres). Sendo que todos caracteres são dígitos numéricos, exceto o último, criando uma entrada maliciosa, semelhante a Figura 22, porém de diversos tamanhos.

Observa-se na tabela 1 que o tempo de resposta da aplicação aproximadamente dobra a cada caractere adicionado na *string* de entrada. Isto é, a ordem de crescimento do tempo é exponencial em relação ao tamanho da entrada. Pode-se estimar que, nesse exemplo, com quarenta caracteres o tempo de resposta seria de aproximadamente três horas e trinta minutos. O que significaria uma indisponibilidade da aplicação durante todo esse tempo. É claro que esse tempo depende de muitos fatores como por exemplo o poder computacional do servidor que está executando a aplicação. Mesmo assim, se sua aplicação roda em um servidor cem vezes mais rápido do que o usado neste exemplo, com apenas quarenta caracteres seria gerada uma indisponibilidade de dois minutos, que é uma eternidade na web.

Goldberg *et al* (2019), recomendam usar, quando possível, uma biblioteca de validação, como por exemplo a *validator.js* já citada na seção 4.4. Quando for necessário usar uma expressão regular recomendam o uso da biblioteca *safe-regex*,

que detecta expressões regulares potencialmente vulneráveis a ataques de negação de serviço. Os autores ainda recomendam validar o tamanho máximo da entrada antes de realizar qualquer operação com a mesma (incluindo testes com *regex*). Pois assim evita-se que, caso haja uma vulnerabilidade, o atacante não tenha o poder de usar uma entrada muito grande, minimizando os danos.

## 5 BOAS PRÁTICAS

Este capítulo destaca algumas boas práticas em Node.js relacionadas a segurança da informação, principalmente no aspecto de disponibilidade. O objetivo é mostrar ações que podem evitar uma variedade de vulnerabilidades em aplicações e aumentar consideravelmente a segurança da mesma. A principal base deste capítulo é o maior guia de boas práticas de Node.js atualmente, mantido por Goldberg *et al* (2019). O guia conta com 82 boas práticas, no momento de escrita deste trabalho, e é constantemente atualizado e expandido pelos responsáveis e por contribuições da comunidade. Algumas das boas práticas deste guia, principalmente relacionadas a negação de serviço, são listadas a seguir:

- Uma das principais boas práticas em Node.js é executar o Node.js como usuário que não seja *root* ou administrador do sistema. Isso é importante pois os pacotes de terceiros do Node.js têm acesso à áreas críticas do sistema, como por exemplo o sistema de arquivos. Caso a aplicação possua um pacote contendo código malicioso, e esteja executando em modo *root*, o atacante tem acesso e permissão para ler, escrever e excluir arquivos importantes do sistema.
- Existe um dilema, uma aplicação web precisa de acesso a porta 80 ou 443, e essas portas só podem ser acessadas por um usuário *root*. A solução é uma outra boa prática: usar um *proxy* reverso. Os autores usam o Nginx como exemplo, que redireciona as requisições a aplicação Node.js. Além disso recomendam que tudo o que for possível seja delegado ao *proxy* reverso, como por exemplo servir arquivos estáticos, TLS (*Transport Layer Security*) e gzip. Como já visto, o Node.js é eficiente para requisições de entrada e saída. Porém outras operações como servir arquivos, criptografia e compactação de arquivos, não são compatíveis com a filosofia *single-threaded* não bloqueante do Node.js. Delegar esse tipo de operações para um *proxy* reverso faz a aplicação Node.js se comportar de forma mais eficiente, aumentando a disponibilidade.
- Outra prática é utilizar todos núcleos do processador. Como já visto o Node.js é *single-threaded*, ou seja, executa em apenas um núcleo processador. Na prática isso não é ideal pois todos servidores atuais

possuem processadores com vários núcleos. Não por acaso, o Node.js possui um módulo nativo para criação de *clusters*, que com poucas linhas de código permite o balanceamento de carga entre todos núcleos do processador disponíveis. Ainda mais fácil é o uso do pacote terceirizado PM2, já discutido na seção 3.2 deste trabalho.

- O PM2 também é útil para mais uma boa prática. Saber quando é necessário finalizar e reiniciar o processo em caso de erro. Pois nem sempre que ocorre um erro é necessário reiniciar, as vezes apenas registrar o erro em um *log* é o suficiente. Além disso deve se evitar reiniciar o processo quando o erro é gerado a partir da entrada de um usuário. Os autores dão como exemplo o envio de uma entrada JSON (*JavaScript Object Notation*) vazia para uma aplicação que não valida esse tipo de entrada. Se isso gerar um erro e reiniciar o processo, um atacante pode gerar várias requisições com uma entrada vazia e reiniciar o processo várias vezes consecutivas em pouco espaço de tempo, causando uma negação de serviço.
- É importante medir e monitorar os recursos do servidor, principalmente memória e CPU. Para evitar o vazamento de memória os autores sugerem por exemplo evitar o uso de variáveis globais e funções anônimas. Para o monitoramento os autores destacam soluções de monitoramento de fornecedores de nuvem, como por exemplo AWS CloudWatch e Google StackDriver. Os autores ainda indicam o uso de APM (*Application Performance Management*), que é o monitoramento e gerenciamento de desempenho e disponibilidade de aplicações. Os autores ainda sugerem o uso de *smart logging*, que consiste em registrar, agrupar e visualizar os *logs* de forma inteligente, de preferência utilizando bibliotecas especializadas nisso,
- Outra boa prática em Node.js é definir a variável de ambiente `NODE_ENV` igual a *production*. Definindo esse valor para essa variável de ambiente, remove a aplicação do modo de desenvolvimento (que é o padrão) e a coloca em modo de produção. Segundo os autores, isso faz com que o número de requisições que o Node.js pode manipular aumente cerca de dois terços. Além disso o uso de CPU diminui um pouco. Na prática a aplicação três vezes mais rápida segundo os



autores. Além disso nesse modo os detalhes de erros não são exibidos para o usuário, ao invés disso é exibida uma mensagem de erro genérica quando ocorre um erro. Isso é importante pois quanto menos detalhes um atacante mal-intencionado souber mais difícil dele achar uma vulnerabilidade.

- Também é recomendado limitar requisições simultâneas, utilizando bibliotecas como *express-rate-limit* visto na seção 3.1 deste trabalho. Os autores destacam principalmente a limitação de requisições para rotas de *login*, a fim de evitar ataques de força bruta. Outra biblioteca indicada pelos autores é a *express-brute*, cujo diferencial é a capacidade de impor o limite compartilhado entre os *clusters* da aplicação.

A documentação do Node.js destaca a boa prática de validar o tamanho da entrada do usuário, antes de qualquer outro tipo de verificação. Pois uma entrada muito grande pode causar uma negação de serviço. O exemplo dado na documentação do Node.js é o uso de uma entrada JSON muito grande, que na maioria dos casos precisa passar por processos de conversão de JSON para *string* ou vice-versa. E esse tipo de operação é muito demorada e usa muito processamento, podendo causar uma negação de serviço. E também como já foi visto no capítulo 4, um caractere *a* mais pode dobrar o tempo de execução de uma expressão regular vulnerável.

Düüna (2016) destaca algumas boas práticas para se evitar ataques de negação de serviço através de funções assimétricas. Funções assimétricas são aquelas cujo tempo de execução dependem do tamanho da entrada do usuário. A primeira recomendação do autor é evitar funções assimétricas quando possível. Mas algumas vezes não existe uma maneira de evitá-las, então o autor recomenda limitar o tamanho da entrada, como foi visto no parágrafo anterior. Düüna (2016) ainda recomenda que apenas usuários autenticados possam usar de funções assimétricas, diminuindo a possibilidade de ataques de negação de serviço.

## CONSIDERAÇÕES FINAIS

Este trabalho mostrou conceitos de segurança da informação como vulnerabilidade, ameaça, disponibilidade e negação de serviço. Além disso foram vistas as vulnerabilidades de negação de serviço em aplicações Node.js mais comuns e como evitá-las.

Foi discutido o funcionamento do Node.js, a sua natureza *single-threaded* não bloqueante, o *loop* de eventos e a importância de não o bloquear. Estudou-se também a diferença entre códigos síncronos, assíncronos com *buffer* e assíncronos com *streams*.

Concluiu-se que em situações que o bloqueio não causa danos, ou seja, antes da execução entrar no *loop* de eventos, é preferível o uso de código síncrono, pois em geral é mais rápido. Já em situações em que o bloqueio é danoso, ou seja, a aplicação já está executando no *loop* de eventos e recebendo requisições de vários usuários, é necessário o uso de código assíncrono.

Discutiu-se também sobre bibliotecas, como encontrar vulnerabilidades nelas, e exemplos de bibliotecas específicas para segurança. Concluiu-se que o uso de bibliotecas deve ser cauteloso, considerando-se quesitos como popularidade, manutenção e qualidade. E ainda que algumas bibliotecas focadas em segurança podem reduzir consideravelmente o risco de ameaças em uma aplicação com apenas algumas poucas linhas de código.

Verificou-se também a importância de se utilizar expressões regulares de forma segura. Podendo se aproveitar das vantagens delas sem comprometer a segurança, principalmente a disponibilidade, da aplicação.

Também se observou algumas boas práticas em Node.js que ajudam a mitigar vulnerabilidades e aumentar a disponibilidade de aplicações.

É importante também destacar que as vulnerabilidades vistas neste trabalho não correspondem a totalidade das mesmas em aplicações Node.js. Existem outras vulnerabilidades que se relacionam por exemplo com serviços de terceiros para hospedagem e manutenção da aplicação, recursos de *hardware* disponíveis, recursos humanos, entre outros. Como visto, existem vulnerabilidades de outras naturezas além de negação de serviço, como por exemplo: injeção de código, *cross-site-script* e roubo de sessão. É de extrema importância proteger a aplicação contra esses tipos

de vulnerabilidades também. Isso é possível usando uma variedade de políticas, controles e técnicas. Por exemplo *hardening*, que consiste em implementar medidas de segurança no servidor, na rede ou na organização.

Conclui-se assim que muitas vulnerabilidades de negação de serviço podem ser evitadas tomando-se medidas mostradas neste trabalho. No ambiente de segurança da informação, novas tecnologias e técnicas surgem a todo momento, tanto para ajudar na segurança quanto para tentar quebrá-la. Portanto é importante também manter-se sempre atualizado sobre novos tipos de vulnerabilidades, ameaças e como combatê-las, para que suas aplicações estejam sempre protegidas da melhor maneira possível.

## REFERÊNCIAS BIBLIOGRÁFICAS

BELDER, BERT. *Everything You Need to Know About Node.js Event Loop*. In: Node.js Interactive 2016, Vancouver, 24 set. 2016. Disponível em: <<https://www.youtube.com/watch?v=PNa9OMajw9w>> Acesso em: 20 fev. 2019.

CASCIARO, Mario; MAMMINO, Luciano. *Node.js Design Patterns*. 2. ed. Birmingham: Packt Publishing, 2016.

CELEBRATE. Biblioteca para aplicações Node.js/Express para validação de dados, 2019. Disponível em: <<https://github.com/arb/celebrate#readme>> Acesso em: 04 mar. 2019.

COLLINA, Matteo. *Protecting Node.js from uncontrolled resource consumption headers attacks*. 28 nov. 2018. Disponível em: <<https://www.nearform.com/blog/protecting-node-js-from-uncontrolled-resource-consumption-headers-attacks/>> Acesso em: 20 fev. 2019.

DAHL, Ryan. *10 Things I Regret About Node.js*. In: JSConf EU 2018, Berlin, 6 jun. 2018. Disponível em: <<https://www.youtube.com/watch?v=M3BM9TB-8yA>> Acesso em: 20 fev. 2019.

DAVIS, James C. *A Sense of Time for JavaScript and Node.js*. In: USENIX Security '18, Santa Clara, 18 set. 2018. Disponível em: <<https://www.youtube.com/watch?v=Dm7Xyw3KueY>> Acesso em: 20 fev. 2019.

DE TURCKHEIM, Vladimir. *Node.js Applicative DoS Through MongoDB Injection*. In: Node.js Interactive 2018, Vancouver, 19 out. 2018. Disponível em: <<https://www.youtube.com/watch?v=xJWZsoYmsIE>> Acesso em: 20 fev. 2019.

DÜÜNA, Karl; RASHID, Fahmida Y. (Ed.). *Secure Your Node.js Web Application*. Raleigh: Pragmatic Programmers LLC, 2016.

EXPRESS. *Production Best Practices: Security*. Boas práticas de segurança para aplicações Node.js/Express. Disponível em: <<https://expressjs.com/en/advanced/best-practice-security.html>> Acesso em: 20 fev. 2019.

EXPRESS RATE LIMIT. Biblioteca para aplicações Node.js/Express que limita o número de requisições a um servidor http, 2019. Disponível em: <<https://github.com/nfriedly/express-rate-limit>> Acesso em: 03 mar. 2019.

GOLDBERG, Yoni. *et al. Node.js Best Practices*. 2019. Disponível em: <<https://github.com/i0natan/nodebestpractices>> Acesso em: 17 abr. 2019.

HELMET. Biblioteca para aplicações Node.js/Express que ajuda a proteger aplicações configurando cabeçalhos http, 2019. Disponível em: <<https://helmetjs.github.io/>> Acesso em: 03 mar. 2019.

JOI. Biblioteca para aplicações Node.js que valida dados, 2019. Disponível em: <<https://github.com/hapijs/joi>>. Acesso em: 04 mar. 2019.

KANTOR, Ilya. *et al. The Modern JavaScript Tutorial*. Disponível em: <<https://javascript.info/>> Acesso em: 11 mar. 2019.

MUELLER, John Paul. *Security for Web Developers*. 1. Ed. Sebastopol: O'Reilly Media Inc, 2016.

NAHARI, Hadi; KRUTZ, Ronald L. *Web Commerce Security Design and Development*. Indianapolis: Wiley Publishing Inc, 2011.

NODE.JS. *Don't Block the Event Loop (or the Worker Pool)*. Disponível em: <<https://nodejs.org/en/docs/guides/dont-block-the-event-loop/>> Acesso em: 20 fev. 2019.

NODE.JS. *November 2018 Security Releases*. Disponível em: <<https://nodejs.org/en/blog/vulnerability/november-2018-security-releases/>> Acesso em: 20 fev. 2019.

NODE.JS. *Overview of Blocking vs. Non-Blocking*. Disponível em: <<https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/>> Acesso em: 20 fev. 2019.

NODE.JS. *The Node.js Event Loop, Timers and process.nextTick()*. Disponível em: <<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>> Acesso em: 20 fev. 2019.

NODE SEC ROADMAP. Guia de boas práticas de segurança em Node.js. Disponível em: <<https://nodesecroadmap.fyi/>> Acesso em: 20 fev. 2019.

NPM Security Advisories. Banco de dados oficial de vulnerabilidades em bibliotecas distribuídos pelo NPM, 2019. Disponível em: <<https://www.npmjs.com/advisories>>. Acesso em: 20 fev. 2019.

NPMS. Ferramenta de busca de bibliotecas para Node.js. Disponível em: <<https://npms.io/>> Acesso em: 20 fev. 2019.

O'HANLEY, Richard (Ed.); TILLER, James S. (Ed.). *Information Security Management Handbook*. 6. ed. Boca Raton: CRC Press, 2014.

OPEN WEB APPLICATION SECURITY PROJECT. *Denial of Service*. 2 fev. 2015. Disponível em: <[https://www.owasp.org/index.php/Denial\\_of\\_Service](https://www.owasp.org/index.php/Denial_of_Service)> Acesso em: 20 fev. 2019.

OPEN WEB APPLICATION SECURITY PROJECT. *Regular expression Denial of Service - ReDoS*. 2 fev. 2015. Disponível em: <[https://www.owasp.org/index.php/Regular\\_expression\\_Denial\\_of\\_Service\\_-\\_ReDoS](https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS)> Acesso em: 20 fev. 2019.

PELTIER, Thomas R. *Information Security Fundamentals*. 2. ed. Boca Raton: CRC Press, 2014.

PM2. Biblioteca para aplicações Node.js/Express que gerencia processos, 2019. Disponível em: <<https://pm2.io/doc/en/runtime/overview/>> Acesso em: 03 mar. 2019.  
RHODES-OUSLEY, Mark. *Information Security - The Complete Reference*. 2. ed. New York: McGraw-Hill Education, 2013.

SAFE-REGEX. Biblioteca para aplicações Node.js que detecta expressões regulares vulneráveis, 2019. Disponível em: <<https://github.com/davisjam/safe-regex>> Acesso em: 26 mar. 2019.

SAMUEL, Mike. *A Node.js Security Roadmap*. In: JSConf EU 2018, Berlin, 17 jul. 2018. Disponível em: <<https://www.youtube.com/watch?v=1Gun2lRb5Gw>> Acesso em: 20 fev. 2019.

SHARMA, Tarun. *Secure Node JS Apps*. 24 jun. 2018. Disponível em: <<https://medium.com/@tkssharma/secure-node-js-apps-7613973b6971>> Acesso em: 20 fev. 2019.

SIMPSON, Kyle. *You Don't Know JS: Async & Performance*. Sebastopol : O'Reilly Media, 2015.

SNYK. Empresa que mantém bancos de dados de vulnerabilidades em várias bibliotecas de código aberto, 2019. Disponível em: <<https://snyk.io/vuln/>> Acesso em: 20 fev. 2019.

STAICU, Cristian-Alexandru. *Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers*. In: USENIX Security '18, Santa Clara, 18 set. 2018. Disponível em: <<https://www.youtube.com/watch?v=612mO3leexs>> Acesso em: 20 fev. 2019.

TEIXEIRA, Pedro. *Professional Node.js*. Indianapolis: John Wiley & Sons Inc, 2013.

TRIBUNAL DE CONTAS DA UNIÃO. *Boas Práticas em Segurança da Informação*. 4. ed. Brasília: TCU, Secretaria de Fiscalização de Tecnologia da Informação, 2012.

VALIDATOR.JS. Biblioteca para aplicações Node.js que valida dados, 2019. Disponível em: <<https://github.com/chriso/validator.js>>. Acesso em: 04 mar. 2019.

WHITMAN, Michael E.; MATTORD, Hebert J. *Principles of Information Security*. 4. ed. Boston: Course Technology, 2011.

## APÊNDICE A

Biblioteca crono.js, utilizada para cronometrar o tempo de execução nos exemplos da seção 2.5.

Figura 23: crono.js

```
1 exports.stop = (timer) => {  
2     const end = process.hrtime(timer); /*para o cronômetro*/  
3     /*divide o tempo em mili, micro e nanosegundos*/  
4     end[3] = end[1] % 1000;  
5     end[2] = Math.floor((end[1] - end[3]) / 1000) % 1000;  
6     end[1] = Math.floor(end[1] / 1000000);  
7     /*mostra no console o tempo*/  
8     console.log(`Tempo: ${end[0]} segundos,  
9                 ${end[1]} milisegundos,  
10                ${end[2]} microsegundos,  
11                ${end[3]} nanosegundos`);  
12 }
```

Fonte: Autoria própria