# BLU-MAN-006-R01
# Allegato 02-R00
# Aggiornamento del 08/09/2026

# JavaScript



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 Open redirect ..... 5
3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT ..... 6
3.1 Validazione ..... 6
3.2 Sanitizzazione ..... 7
3.3 File validation ..... 9
4 INJECTION ..... 10
4.1 Cross-Site Scripting ..... 10
4.2 JQuery Selector Injection ..... 11
4.3 Prototype Pollution ..... 13
4.4 Cross-Site Script Inclusion (XSSI) ..... 16
4.5 Code Injection ..... 18
5 MISCONFIGURATIONS ..... 20
5.1 Mancato controllo di integrità sulle risorse ..... 20
6 INFORMAZIONI CONFIDENZIALI ..... 21
6.1 Credenziali Hardcoded ..... 21
7 CRITTOGRAFIA ..... 22
7.1 Algoritmi e Chiavi di Cifratura Robusti ..... 22
8 DENIAL OF SERVICE ..... 24
8.1 Regular Expression Denial of Service – ReDos ..... 24
9 Link utili ..... 25

# 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 JavaScript.

## 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 JavaScript. 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 JavaScript, 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.

Inizialmente viene 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.

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, come il mancato controllo sull'utilizzo di componenti/configurazioni deprecate, e quali best practices seguire per poterle evitare.

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.

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.

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. Sono varie le casistiche che possono portare un 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 delle risorse allocate e mai chiuse 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.



Italgas S.p.A - Informazioni Interne

# 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 Open redirect

Gli open redirect, letteralmente reindirizzamenti aperti, occorrono quando un'applicazione utilizza direttamente dati controllabili dall'utente per costruire dinamicamente un url di reindirizzamento. Questa vulnerabilità trova utilizzo soprattutto nelle campagne di phishing, ed il seguente esempio ne mostra la motivazione.

```
https://www.mydomain.com/redirect?url=malicious.net
```

L'url appare a prima vista fidato, dato che comincia con www.mydomain.com. Tuttavia, l'applicativo registrato su questo dominio offre una funzionalità di reindirizzamento, consultabile alla risorsa /redirect. Questa accetta come parametro GET la chiave url, attraverso la quale si può specificare un indirizzo arbitrario al quale redirigere l'utente ignaro. Con buone probabilità, il sito appena raggiunto sarà:

- una copia del sito controllata dall'attaccante, che riuscirà così a far loggare l'utente rubandogli le credenziali;
- una piattaforma di distribuzione di malware;
- Un attacco XSS sul Document Object Model, con esecuzione di codice JS malevolo all'interno del dominio fidato.

Il seguente codice mostra un'implementazione vulnerabile all'attacco appena descritto, implementando senza alcuna forma di protezione, come possono essere validazione, sanitizzazione o livello di indirezione, un redirect con input utente.

```
const queryParams = new URLSearchParams(document.location.search);
const redirectUrl = queryParams.get("url");
document.location = redirectUrl; // vulnerabile
```

Nel seguente snippet invece è proposto un redirect sicuro, in quanto gli host consentiti sono appartenenti al dominio hardcoded https://www.example.com/. I target raggiunti con un url così strutturato sono considerati trusted. Pertanto non pongono un problema alla sicurezza.

```
const queryParams = new URLSearchParams(document.location.search);
const redirectUrl = queryParams.get("url");

if (redirectUrl.startsWith("https://www.example.com/")) {
    document.location = redirectUrl;
}
```

**Nota:** quando si deve reindirizzare ad un link esterno, assicurarsi che la validazione tramite *schema://autorità/* contenga il separatore / come ultimo carattere. Questo per proteggersi da una tecnica nota come Cybersquatting, mostrata di seguito.

```
var text = “https://example.com.malicious.io”;
text.startsWith("https://example.com"); // true, vulnerabile

var text = “https://example.com.malicious.io”;
text.startsWith("https://example.com/"); // false, sicura
```

# 3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

L’input utente è per sua natura da considerare non fidato o untrusted. Pertanto l’applicativo in sviluppo deve essere concepito sempre considerando il caso peggiore quando si tratta di sicurezza. Un utente malintenzionato potrebbe infatti sfruttare una validazione troppo lascia 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 degli input forniti dagli utenti in Javascript è essenziale per garantire che i dati inviati al back-end applicativo siano corretti e nel formato che l’applicativo si aspetta. Tramite Javascript è possibile effettuare i controlli lato client, generalmente applicabili sui form inviati dagli utenti. Per mezzo di opportune funzioni di validazione, da richiamare all’interno del codice HTML, è possibile convalidare nome, password, e-mail, numeri di cellulare e altri campi.

- Nel seguente esempio, l’obiettivo è quello di validare username (non può essere vuoto) e password (deve contenere almeno sei caratteri):



Italgas S.p.A - Informazioni Interne

```
<script>
function validateform(){
    var name=document.myform.name.value;
    var password=document.myform.password.value;

    if (name==null || name==""){
        alert("Name can't be blank");
        return false;
    }else if(password.length<6){
        alert("Password must be at least 6 characters long.");
        return false;
    }
}
</script>
<body>
    <form name="myform" method="post" action="abc.jsp" onsubmit="return validateform()" >
        Name: <input type="text" name="name"><br/>
        Password: <input type="password" name="password"><br/>
        <input type="submit" value="register">
    </form>
</body>
```

- A seconda delle esigenze, è possibile convalidare altri aspetti come input numerici, indirizzi e-mail o, in generale, campi personalizzati. Ad esempio, per convalidare un nome utente senza spazi, si può usare un'espressione regolare come la seguente:

```
function validateUsername(value) {
    const regex = new RegExp("\\s"); // Matches any whitespace character
    return !regex.test(value); // Returns true if no spaces are found
}
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
    Name: <input type="text" name="fname">
    <input type="submit" value="Submit">
</form>
```

## 3.2 Sanitizzazione

La sanitizzazione degli input forniti dagli utenti in Javascript è essenziale per garantire che i dati inviati al back-end applicativo siano ripuliti da caratteri che possono essere interpretati per l'esecuzione di codice. Tramite Javascript è possibile effettuare i controlli lato client, generalmente applicabili sui form inviati dagli utenti. Per mezzo di opportune funzioni di sanitizzazione, da richiamare all'interno del codice HTML, è possibile sanitizzare nome, password, e-mail, numeri di cellulare e altri campi.

- Nel seguente esempio, l'obiettivo è quello di creare una funzione personalizzabile per la sanificare l'input fornito dall'utente, facendo "*escaping*" dei caratteri speciali:

```
function sanitize(string) {
  const map = {
      '&': '&amp;',
      '<': '&lt;',
      '>': '&gt;',
      '"': '&quot;',
      "'": '&#x27;',
      "/": '&#x2F;',
  };
  const reg = /[&<>"'/]/ig;
  return string.replace(reg, (match)=>(map[match]));
}
```

- Esistono funzioni dedicate per effettuare *Input Sanitization* in pagine HTML, come ad esempio quelle messe a disposizione dalla libreria *DOMPurify*, preservando il contenuto:

```
const sanitizedInput = DOMPurify.sanitize(userInput);
```

- Si sconsiglia l'utilizzo della funzione *escape()* al fine di effettuare *Input Sanitization* nelle pagine HTML. Il suo uso è suggerito quando si tratta di controlli sugli URL.

```
function validateURL(surl) {
    const url = parseURL(surl);
    const urlHostname = url.hostname.trim();

    if (urlHostname === '') {
        return true;
    } else {
        if (urlHostname.toUpperCase() === location.hostname.trim().toUpperCase()) {
            return true;
        } else {
            return false;
        }
    }
}
```

### 3.3 File validation

Tramite l'utilizzo di funzioni Javascript è possibile effettuare dei controlli sui file caricati dall'applicativo. La tipologia di controlli più diffusa riguarda l'estensione del file.

- Nel seguente esempio, si suppone che l'unica estensione accettata sia *.gif*. Di seguito si riporta un esempio di funzione di controllo:

```
<input type="file" name="file_uploading" id="filename" onchange="Checkfiles(this)">
<script>
  function Checkfiles(inputElement) {
    var fileName = inputElement.value;
    var ext = fileName.substring(fileName.lastIndexOf('.') + 1).toLowerCase();
    if (ext === "gif") {
      return true;
    } else {
      alert("Upload GIF images only");
      return false;
    }
  }
</script>
```

**Nota:** tutti questi controlli lato client, pur essendo importanti per guidare l'esperienza utente tramite interfaccia grafica dal browser, devono essere supportati da una altrettanto robusta validazione lato server, in quanto le interfacce messe a disposizione sono facilmente bypassabili anche da hacker meno esperti.

# 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 vulnerabilità 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 oltretutto che il payload dell'attacco venga conservato in modo permanente nel browser della vittima, come un database HTML5, con l'arrivo di HTML5 e di altre tecnologie per browser, e non venga mai comunicato al server.
- **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 Javascript vulnerabile.

```
<span id="utente"></span>

<script>
window.onload = (event) => {
    var untrusted = new URLSearchParams(window.location.search).get("user");
    document.getElementById("utente").innerHTML = untrusted;
}
</script>
```

È altamente sconsigliato gestire l'input utente tramite i metodi innerHTML, outerHTML e html in quanto vanno ad interpretare il codice html Runtime. Questi dovrebbero essere infatti rimpiazzati con le rispettive controparti innerText, outerText e text, considerate sicure.

```
<span id="utente"></span>

<script>
window.onload = (event) => {
    var untrusted = new URLSearchParams(window.location.search).get("user");
    document.getElementById("utente").innerText = untrusted;
}
</script>
```

Per prevenire la vulnerabilità in Javascript 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.

Altre contromisure valide sono la sanitizzazione degli header di risposta HTTP e una corretta Content Security Policy (CSP), non trattate in questo documento in quanto non implementabili in JS client-side.

## 4.2 JQuery Selector Injection

JQuery è una libreria Javascript pensata per semplificare la gestione e la manipolazione dell'alberatura HTML contenuta all'interno del Document Object Model (DOM).

Viene messo a disposizione degli sviluppatori un costrutto denominato Selettore, il cui scopo è ritornare i nodi appartenenti al DOM che rispecchiano determinate caratteristiche specificabili in input. Vengono mostrati degli esempi per chiarezza.

```
<script>
    var res = $("#myIdentifier");
</script>

<span id="myIdentifier">Questo è un esempio di selettore per il campo id</span>
```

```
<script>
    var res = $("span");
</script>

<span>Questo è un esempio di selettore per il tagname</span>
<span>Anche questo span viene ritornato</span>
<p> Questo tag 'p' invece non rientra nella selezione </p>
```

Pur svolgendo un ruolo molto utile e semplificando la sintassi classica di Javascript, alcune vecchie versioni di Jquery, per la precisione fino alla 1.7.2 inclusa, presentavano serie criticità legate alla sicurezza di questa loro componente.

Le vulnerabilità risiedono in un comportamento inaspettato del selettore. Nei casi infatti in cui il nodo cercato non esista, questo viene creato sulla base dell'input di ricerca. Oltretutto, se l'input è untrusted, ovvero non codificato in alcun modo e direttamente definito dall'utente, ci si trova davanti ad un algoritmo vulnerabile ad attacchi di Injection, solitamente XSS.

Vengono presentate alcune implementazioni note per non essere sicure, che vanno ad eseguire il codice javascript specificato dall'attributo onerror nel caso la path specificata da src non esista.

```
// JQuery fino a v1.7.2 inclusa
var user_input = '<img src=x onerror=alert(1)>'

$(user_input); // Non compliant
$('.' + user_input); // Non compliant
$('input[id^=listIdServizi][value=]' + user_input) // Non compliant
```

Si raccomanda di aggiornare la componente all'ultima versione disponibile. Se però questa soluzione non è praticabile, l'approccio alternativo suggerito comporta validare o sanitizzare il valore untrusted passato come parametro. Se il valore del selettore non è modificabile direttamente dall'input utente, questo è sicuro.

```
var user_input = '<img src=x onerror=alert(1)>'

$(escape(user_input)); // Compliant
$('.' + escape(user_input)); // Compliant
$('input[id^=example][value=]' + escape(user_input)); // Compliant
```

Le versioni di JQuery fino alla 1.6.2 inclusa erano vulnerabili anche ai selettori specificati con il seguente formato.

```
// JQuery fino a v1.6.2 inclusa
var user_input = '<img src=x onerror=alert(1)>'

$('#' + user_input); // Non compliant
$('#hardcoded' + user_input); // Non compliant
```

Anche in questo caso, la soluzione raccomandata è di aggiornare la libreria all'ultima versione disponibile. L'approccio alternativo in caso non sia possibile aggiornare comporta la validazione o sanitizzazione dell'input utente. Se il valore nel selettore non è modificabile direttamente dall'utente, è sicuro.

```
var user_input = '<img src=x onerror=alert(1)>'

$('#' + escape(user_input)); // Compliant
$('#hardcoded' + escape(user_input)); // Compliant
```

### 4.3 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 Prototype Pollution permette ad un attaccante di definire nuove proprietà ad oggetti prototype globali, le quali vengono poi logicamente ereditate dagli oggetti in uso nell'applicativo JS.

Di per sé questa vulnerabilità non è direttamente sfruttabile, ma permette ad un utente malevolo di controllare l'ambiente d'esecuzione preparandolo per un futuro attacco. In questo caso, essendo Javascript un linguaggio client-side, la vulnerabilità può scaturire in un attacco DOM-XSS.

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 JS a __proto__ , l'operazione di assegnamento delle proprietà innestate avverrà sull'oggetto prototype invece che sull'oggetto per cui era pensato.

Vettori d'attacco comuni per iniettare il payload sono:

**URL Prototype Pollution**
Se l'applicativo accetta parametri passati tramite URL per costruire i campi dell'oggetto, questo ne è un esempio.

```
https://example.com/?__proto__[ShouldNotHappen]=payload
```

Che viene tradotto nell'equivalente

```
targetObject.__proto__.ShouldNotHappen = 'payload';
```

Tutti gli oggetti che implementano targetObject.__proto__ erediteranno anche questa proprietà.

**JSON Prototype Pollution**
JSON (JavaScript Object Notation) è un linguaggio standardizzato utilizzato principalmente per l'archiviazione e lo scambio di dati tramite un paradigma ad oggetti.

Gli oggetti controllabili dall'utente spesso derivano da una stringa JSON che viene tradotta nell'oggetto corrispondente. Viene proposto un esempio per chiarezza.

```
var objectFromJSON = JSON.parse('{ "make": "Ford", "model": "Mustang", "year": "1969" }');
```

Se l'input processato con il metodo JSON.parse() contiene un riferimento a __proto__, le varie proprietà saranno assegnate al prototipo. Questo non vale per gli oggetti dichiarati letteralmente senza alcun parsing.

```
const objectLiteral = {__proto__: {evilProperty: 'payload'}};
const objectFromJson = JSON.parse('{"__proto__": {"evilProperty": "payload"}}');

objectLiteral.hasOwnProperty('__proto__');     // false
objectFromJson.hasOwnProperty('__proto__');    // true
```

Per prevenire questa vulnerabilità esistono alcune alternative

**Sanitizzare le chiavi delle proprietà**
Il processo di validazione o sanitizzazione deve assicurare che non siano presenti riferimenti in alcun modo all'oggetto prototype. Questa implementazione tuttavia è sconsigliata a lungo termine in quanto potrebbe non gestire tutti casi ed esistono alternative più solide che vanno a coprire la problematica.

**Prevenire cambiamenti ai prototipi**
Un approccio più robusto infatti consiste nel prevenire i cambiamenti nello stato dell'oggetto prototype.
Il metodo Object.freeze() assicura che le proprietà di Object ed i relativi valori non cambino nel tempo, congelandolo allo stato attuale senza più possibilità di togliere, modificare o aggiungere contenuti.

Un'alternativa simile ma meno radicale sta nell'utilizzare il metodo Object.seal(), che si differenzia da Object.freeze() in quanto permette di modificare i valori delle proprietà già definite, pur non permettendo di definirne di nuove.

Prevenire cambiamenti ai prototipi è consigliato su tutte, data la sua robustezza ed affidabilità.

**Impostare null il prototipo**
Come suggerisce il titolo, impostare a null il prototipo quando questo non è espressamente richiesto by design può evitare l'exploit della vulnerabilità. Essendo che non sempre un design privo di ereditarietà tra oggetti JS può essere implementato, soprattutto negli algoritmi più complessi, non sempre questa remediation è fattibile.

Viene comunque proposto un esempio in cui viene creato un oggetto con prototype null.

```
let myObject = Object.create(null);
Object.getPrototypeOf(myObject);     // null
```

**Implementare un oggetto Map()**
Un'altra possibilità di protezione dalle Prototype Pollution è l'utilizzo di un oggetto Map() per definire e settare tutte le proprietà interne.
Infatti, pur essendo ancora vulnerabile ad una prototype pollution, un oggetto Map dispone di metodi get e set che ritornano solo le proprietà definite direttamente nella Map dai rispettivi metodi setter.

```
Object.prototype.evil = 'polluted';
let options = new Map();
options.set('transport_url', 'https://normal-website.com');

options.evil;                    // 'polluted'
options.get('evil');             // undefined
options.get('transport_url');    // 'https://normal-website.com'
```

Un'alternativa ad un oggetto Map è l'oggetto Set, che a differenza del primo è una collezione di valori univoci a differenza di Map che può avere valori duplicati.

```
Object.prototype.evil = 'polluted';
let options = new Set();
options.add('safe');

options.evil;          // 'polluted';
option.has('evil');    // false
options.has('safe');   // true
```

**Nota:** se per un attaccante fosse possibile portare a termine un attacco XSS potrebbe richiamare un metodo setter per una proprietà e alterarne il valore. Tuttavia, se un attacco del genere fosse portato a termine, ci sono aspetti più sensibili su cui l'attaccante potrebbe fare leva, come il furto dei cookies o l'esecuzione di codice JS remoto.

### 4.4 Cross-Site Script Inclusion (XSSI)

Il **Cross-Site Script Inclusion (XSSI)** è una vulnerabilità di sicurezza che sfrutta la natura del tag *<script>* in HTML. A differenza della maggior parte delle risorse, che sono soggette alla *Same-Origin Policy* (SOP), gli script possono essere inclusi da domini diversi. Questo comportamento, sebbene utile per l'uso di librerie e risorse esterne, introduce un rischio di sicurezza.

Gli attaccanti possono sfruttare questa vulnerabilità per leggere dati sensibili caricati tramite il tag *<script>*, come cookie di autenticazione o token poiché, per poter accedere a risorse remote autenticate, è necessario che questi vengano inclusi nella richiesta.

Questa vulnerabilità è particolarmente rilevante per codice Javascript dinamico e **JSON with Padding (JSONP)**. JSONP è una tecnica particolarmente utile per caricare dati da API esterne senza incorrere in problemi di sicurezza legati alla Same-Origin Policy.
Generalmente, questi dati sono processati specificando una **callback,** una funziona javascript che deve essere implementata nella pagina che include il JSON**.**

Esistono diversi tipi di vettori attraverso i quali è possibile esfiltrare informazioni sensibili:

- Variabili Globali
- Parametri di Funzioni Globali
- Prototype Chaining utilizzando il *this*
- JSONP tramite callback

Un attaccante può sfruttare attacchi di phishing o ingegneria sociale per portare un utente a anavigare un sito malevolo in cui sono vengono poi sfruttati i precedenti vettori per esfiltrare informazioni sensibili.

#### Variabili Globali

Il seguente codice Javascript è vulnerabile a XSSI in quanto il secret è memorizzato in una variabile globale:

```
(function() {
  window.secret = "supersecretUserAPIkey";
})();
```

Segue uno snippet di codice che un ipotetico attaccante utilizzerebbe per esfiltrare dati sfruttando le variabili globali:

```
<script src="https://vulnerable.com/internal/api.js"></script>
(function() {
  window.secret = "supersecretUserAPIkey";
})();
<script>
```

#### Parametri di Funzioni Globali

Il seguente codice Javascript è vulnerabile a XSSI in quanto il secret è passato come parametro ad una funzione globale:

```
(function() {
  var secret = "supersecretAPIkey";
  window.globalFunction(secret);
})();
```

Segue uno snippet di codice che un ipotetico attaccante utilizzerebbe per esfiltrare dati ridichiarando la funzione globale che viene invocata dal codice insicuro:

```
<script>
    function globalFunction(param) {
        var secret = param;
        // Invia i dati esfiltrati su un listener predisposto da un attaccante
    }
</script>
<script src="https://vulnerable.com/internal/api.js"></script>
```

### Prototype Chaining utilizzando il this

Si prenda in considerazione il seguente codice Javascript:

```
(function() {
  var secret = ["secret1", "secret2"];

  secret.forEach(function(element) {
    // do something here
  });

})();
```

Segue uno snippet di codice che un ipotetico attaccante utilizzerebbe per esfiltrare dati sfruttando il meccanismo di prototype chaining di Javascript, sovrascrivendo l'implementazione della funzione *forEach* nel prototipo della classe *Array*:

```
<script>
   Array.prototype.forEach = function(callback) {
      var secrets = "";
      for (var i = 0, length = this.length; i < length; i++) {
         if (i > 0) {
            secrets += ", ";
         }
         secrets += this[i];
      }
      // Invia i dati esfiltrati su un listener predisposto da un attaccante
   };
</script>
<script src="https://vulnerable.com/internal/api.js"></script>
```

**JSONP tramite callback**

Segue uno snippet di codice che un ipotetico attaccante utilizzerebbe per esfiltrare dati sfruttando una callback per una richiesta JSONP:

```
<script>
     callbackFunction = function (leaked) {
        data = JSON.stringify(leaked);
       // Invia i dati esfiltrati su un listener predisposto da un attaccante
     };
</script>
<script src="https://vulnerable.site/p?jsonp=callbackFunction" type="text/javascript"></script>
```

Questo attacco è possibile solo quando il JSON di risposta ad una richiesta JSONP è eseguibile come Javascript.
Per convenzione la stringa "*)]}',\n*" è utilizzata dai server come prefisso al JSON che forniscono come risposta per indicare che questi non è eseguibile come Javascript.

Per maggiori dettagli fare riferimento a: Testing for Cross-Site Script Inclusion - OWASP Web Security Testing Guide.

## 4.5 Code Injection

Molti linguaggi di programmazione moderni, tra cui JavaScript, consentono l'interpretazione dinamica delle istruzioni sorgente. Questa capacità consente ai programmatori di eseguire istruzioni dinamiche in base agli input ricevuti dall'utente. Le vulnerabilità di code injection si verificano quando il programmatore presume erroneamente che le istruzioni fornite direttamente dall'utente eseguano solo operazioni innocenti, come l'esecuzione di semplici calcoli sugli oggetti attivi dell'utente o la modifica dello stato dell'utente. Tuttavia, senza un'adeguata convalida, un utente potrebbe specificare operazioni che il programmatore non intende eseguire.

Alcune API consentono l'esecuzione di codice dinamico fornendolo come stringhe in fase di esecuzione. Queste API possono essere utili in alcuni casi d'uso molto specifici di meta programmazione. Tuttavia, nella maggior parte dei casi il loro uso è disapprovato, poiché aumentano il rischio di iniezioni di codice malevolo.

Tali attacchi possono essere eseguiti sia sul server che sul client (esempio: attacco XSS) e hanno un enorme impatto sulla sicurezza di un'applicazione.

Esempi sono le chiamate ai costruttori eval() e Function() e all'istruzione 'javascript:'

```
$.getJSON('/tournaments?' + form.tournamentName), {
    location: form.loc,
    etc: form.etc
}, function(data){
    var result = eval(data); // Non compliant
}
```

In questo esempio il fatto che data sia un valore ottenuto come risposta ad un'interrogazione del server non è condizione sufficiente per cui si possa considerare sicuro.

La prima raccomandazione è di non utilizzare questi metodi perché intrinsecamente insicuri, e di implementare una soluzione alternativa in base al contesto e allo scopo.

La seconda è di adottare un sistema di validazione, sanitizzazione (rigorosamente whitelisting) o livello di indirezione in modo da gestire l'input e porre dei limiti al suo contenuto.

Un esempio di implementazione alternativa (pur usando sempre eval, che ricordiamo essere sconsigliato) è il seguente:

```
$.getJSON('/tournaments?' + form.tournamentName), {
    location: form.loc,
    etc: form.etc
}, function(data){
    if (data == "some_data_state")
        var result = eval(data); // Compliant
    else if (data == "some_other_data_state")
        var result = eval(data); // Compliant
        var ok = true;
    else {
      displayError("operazione non consentita");
    }
}
```



Italgas S.p.A - Informazioni Interne

# 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 Mancato controllo di integrità sulle risorse

Quando si caricano risorse esterne, gli script su tutti, oltre ad utilizzare un protocollo cifrato per lo scambio di dati (HTTPS) è necessario utilizzare i controlli di integrità messi a disposizione dal linguaggio.

L'integrità di un messaggio garantisce che il contenuto inviato dal mittente corrisponde al contenuto ricevuto dal destinatario senza che alcuna modifica intermedia non autorizzata sia avvenuta. Questo stato si ottiene grazie agli algoritmi di hashing e ai digest prodotti.

Un algoritmo di hashing, dato in input un valore arbitrario di lunghezza variabile a piacere, restituisce sempre una stringa (chiamata digest) di lunghezza fissa.
Messaggi che differiscono di una sola lettera restituiscono digest completamente diversi, mentre digest troppo corti o algoritmi deprecati riducono la complessità degli attacchi di collisione, che permettono di ottenere lo stesso output con input differenti.

È quindi opportuno utilizzare algoritmi resistenti ad attacchi noti per garantire l'integrità del messaggio, come SHA-2 o SHA-3, che solitamente vengono forniti dal vendor della libreria. Se la libreria è sviluppata internamente, va generato il digest autonomamente. Viene riportato un esempio di codice sicuro.

```
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous">
</script>
```

**Nota: '**sha384' indica l'algoritmo di hashing utilizzato. Ciò che segue, separato dal trattino, è effettivamente il digest ottenuto dall'algoritmo che in automatico verrà comparato con quello ottenuto dalla risorsa scaricata dal link. Se questi due valori combaciano, la risorsa è integra e non è stata alterata.
**Nota:** l'attributo crossorigin="anonymous" permette il controllo dell'integrità della risorsa anche quando questa non rispetta la Same Origin Policy.

# 6 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

## 6.1 Credenziali Hardcoded

Conservare credenziali di accesso in chiaro all'interno del codice sorgente è una pratica errata e pertanto da evitare. Questo perché, com'è facilmente intuibile, espone l'applicazione al rischio di accessi non autorizzati.

Considerando inoltre che Javascript per sua natura è un linguaggio nativamente lato client, la gestione di un protocollo di autenticazione ad un servizio non rientra tra gli scopi per cui è stato pensato, essendo comunemente questo un aspetto gestito dal server. In primo luogo le credenziali di un servizio offerto non devono essere gestite da Javascript.

Al contrario, devono essere conservate lato server all'esterno del codice sorgente, centralizzate all'interno di una soluzione robusta e solamente dopo un processo di one-way hashing. Vengono proposte alcune alternative, le cui implementazioni e raccomandazioni possono differire in base alla tecnologia utilizzata:

- Variabili d'ambiente, salvate e gestite sul server;
- Secret Management System, come Vault o AWS Secrets Manager;
- Database, per l'archiviazione delle utenze con relative credenziali.

# 7 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**.

## 7.1 Algoritmi e Chiavi di Cifratura Robusti

L'utilizzo di algoritmi o chiavi di cifratura poco robusti può pregiudicare la confidenzialità e la privacy delle informazioni cifrate. Tendenzialmente, queste informazioni hanno una natura molto sensibile e possono consistere in credenziali, dati sanitari o bancari, etc.

Segue un esempio di codice Javascript che utilizza l'algoritmo *AES* con modalità CBC, non più considerato sicuro in seguito ad attacchi di critto-analisi:

```
window.crypto.subtle.encrypt(
    { name: " AES-CBC", iv: iv },
    key,
    encoded,
);
```

Una soluzione al precedente problema è quella di utilizzare algoritmi con modalità di funzionamento più robusti come, ad esempio, l'*AES* con modalità di funzionamento Galois/Counter (*GCM)*:

```
window.crypto.subtle.encrypt(
    { name: "AES-GCM", iv: iv },
    key,
    encoded,
);
```

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 versioni 1.0, 2.0 and 3.0
- TLS versioni 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**, nonché 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.

Segue un esempio di generazione di una coppia di chiavi RSA insicure con lunghezza della chiave di 1024 bit:

```
let keyPair = await window.crypto.subtle.generateKey(
  {
    name: "RSA-OAEP",
    modulusLength: 1024,
    publicExponent: new Uint8Array([1, 0, 1]),
    hash: "SHA-256",
  },
  true,
  ["encrypt", "decrypt"],
);
```

Segue un esempio di generazione di una coppia di chiavi RSA sicure con lunghezza della chiave di 4096 bit:

```
let keyPair = await window.crypto.subtle.generateKey(
  {
    name: "RSA-OAEP",
    modulusLength: 4096,
    publicExponent: new Uint8Array([1, 0, 1]),
    hash: "SHA-256",
  },
  true,
  ["encrypt", "decrypt"],
);
```

# 8 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).

## 8.1 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.

Javascript in particolare, essendo eseguito in ambiente single threaded, soffre particolarmente questa problematica. Saturare l'unico thread disponibile dell'ambiente in esecuzione comporta il blocco della logica applicativa.

```
var regex = /([a-z]+)+$/
regex.test('a' + 'a'.repeat(31) + "\x00a")
```

In questo caso, l'input 'a' + 'a'.repeat(31) + "\x00a" corrisponde al pattern dell'espressione regolare, causando però un rallentamento significativo del motore di espressioni regolari, dovuto al fatto che il match viene eseguito con complessità polinomiale partendo dalla fine della stringa.

La maggior parte dei motori di espressioni regolari utilizza il backtracking per provare tutti i possibili percorsi di esecuzione dell'espressione regolare durante la valutazione di un input, ma in alcuni casi può causare problemi di prestazioni, chiamati situazioni di backtracking catastrofico. Nel caso peggiore, la complessità dell'espressione regolare è esponenziale rispetto alla dimensione dell'input, il che significa che un piccolo input accuratamente creato (come 20 caratteri) può innescare un backtracking catastrofico e causare un denial of service dell'applicazione.

Per prevenire gli attacchi ReDoS, è importante seguire delle buone pratiche nella definizione dei pattern delle espressioni regolari. Ad esempio, si dovrebbe evitare l'uso di gruppi con ripetizione all'interno dei quali ci sono altre ripetizioni o alternanze con sovrapposizioni.

Inoltre, è possibile utilizzare strumenti di analisi statica del codice per rilevare automaticamente le vulnerabilità ReDoS nel codice. Tuttavia, la correzione di queste vulnerabilità è ancora un processo manuale e può richiedere di modificare uno o più caratteri nell'espressione regolare.

## 9 Link utili

- **Open Redirect**
  - Unvalidated Redirects and Forwards Cheat Sheet - OWASP Cheat Sheet Series
- **Validazione**
  - Input Validation Cheat Sheet - OWASP Cheat Sheet Series
- **Sanitizzazione**
  - Cross Site Scripting Prevention - OWASP Cheat Sheet Series (Output Encoding)
  - Cross Site Scripting Prevention - OWASP Cheat Sheet Series (HTML Sanitization)
- **File Validation**

File Upload Cheat Sheet - OWASP Cheat Sheet Series

- **Cross Site Scripting**:
  - Cross-site scripting (XSS) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN (mozilla.org)
  - Cross Site Scripting Prevention - OWASP Cheat Sheet Series
- **Prototype Pollution**
  - Prototype Pollution Prevention - OWASP Cheat Sheet Series
  - Map - JavaScript | MDN (mozilla.org)
  - Set - JavaScript | MDN (mozilla.org)
- **Cross-Site Script Inclusion (XSSI)**
  - Testing for Cross-Site Script Inclusion - OWASP Web Security Testing Guide
- **Code Injection**
  - eval() - JavaScript | MDN (mozilla.org)

Function() constructor - JavaScript | MDN (mozilla.org)

- **Mancato controllo di integrità sulle risorse**
  - Subresource Integrity - Security on the web | MDN (mozilla.org)
  - Subresource integrity (SRI) implementation - Security on the web | MDN (mozilla.org)
- **Credenziali Hardcoded**
  - Password Storage - OWASP Cheat Sheet Series

Secrets Management - OWASP Cheat Sheet Series

- **Algoritmi e Chiavi di Cifratura Robusti**
  - Web Crypto API - MDN Docs
  - SubtleCrypto - MDN Docs

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