# BLU-MAN-006-R01
# Allegato 06-R00
# Aggiornamento del 08/09/2026

# Angular

## 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 Route Guards ..... 5
2.2 Cross-Site Request Forgery (CSRF/ XSRF) ..... 6
2.3 Content Security Policy (CSP) ..... 7
2.4 Open Redirect ..... 8
3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT ..... 10
3.1 Built-in Sanitization ..... 10
4 INJECTION ..... 12
4.1 Template Injection ..... 12
4.2 Cross-Site Script Inclusion (XSSI) ..... 13
5 MISCONFIGURATIONS ..... 16
6 LINK UTILI ..... 16

# 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 Angular.

## 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 Angular. 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 Angular, 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.

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 è un elemento fondamentale della sicurezza informatica, che determina chi può accedere a determinati dati, applicazioni e risorse e in quali circostanze. Questo processo si basa su tecniche di autenticazione e autorizzazione, oltre che la corretta configurazione dei sistemi, per verificare l'identità degli utenti e assegnare loro i permessi appropriati.

## 2.1 Route Guards

In Angular, le Route Guard sono strumenti potenti che permettono di controllare l'accesso a determinate parti dell'applicazione. Esistono diversi tipi di guardie, come *CanActivate*, che determina se una route può essere attivata, e *CanDeactivate*, che impedisce agli utenti di lasciare una pagina con modifiche non salvate. Questi meccanismi migliorano l'esperienza utente e la sicurezza, garantendo che solo gli utenti autorizzati possano accedere a determinate funzionalità.

Supponendo che l'applicazione disponga di una route *getUsersList* che ritorna la lista di tutti gli utenti:

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'getUsersList',
        component: UsersListComponent
      }
    ])
  ],
…
})
class AppModule {}
```

questa rotta non è raggiungibile tramite normale navigazione per un utente che non sia amministratore, tuttavia, non essendo in atto alcun controllo degli accessi, un utente malintenzionato potrebbe scoprire la route ed entrare in possesso di informazioni potenzialmente sensibili.

La soluzione è quella di implementare un meccanismo di controllo degli accessi che verifichi che l'utente sia legittimato ad accedere alla rotta richiesta; questo può essere realizzato utilizzando una Route Guard:

```
@NgModule({
  imports: [
    RouterModule.forRoot([
      {
        path: 'getUsersList',
        component: UsersListComponent,
        canActivate: [CanViewUsersList]
      }
    ])
  ],
  providers: [CanViewUsersList]
})
class AppModule {}
```

Dove *CanViewUsersList* è una classe typescript che implementa l'interfaccia *CanActivate:*

```
@Injectable()
class CanViewUsersList implements CanActivate {
…
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): MaybeAsync<GuardResult> {
    … // Implementazione dei controlli d'accesso
   return result;  // ritorna true o false
  }
}
```

## 2.2 Cross-Site Request Forgery (CSRF/ XSRF)

Il Cross-site request forgery (CSRF o XSRF) è un tipo di attacco informatico che sfrutta la fiducia che un'applicazione web ha nei confronti del browser dell'utente. In un attacco CSRF, un malintenzionato induce l'utente finale a eseguire azioni indesiderate su un'applicazione web in cui è autenticato, spesso senza che l'utente ne sia consapevole.

Questo può avvenire attraverso l'ingegneria sociale, come l'invio di un link tramite email o chat. Se la vittima è un utente normale, un attacco CSRF riuscito può costringerlo a eseguire richieste che cambiano lo stato, come trasferire fondi o modificare il proprio indirizzo email. Se la vittima ha un account amministrativo, il CSRF può compromettere l'intera applicazione web.

Per prevenire tali attacchi, è fondamentale implementare misure di sicurezza adeguate, come l'utilizzo di token anti-CSRF e la verifica dell'intestazione 'Origin' o 'Referer' nelle richieste HTTP.

La libreria Angular *HttpClient* supporta il seguente meccanismo di prevenzione per attacchi CSRF: un interceptor, una componente che intercetta tutte le chiamate HTTP in uscita o entrata, legge il token da un cookie *XSRF-TOKEN* proveniente dal server and imposta un header *X-XSRF-TOKEN* contenente il contenuto del cookie sulle richieste provenienti dal client.

Il server quindi validerà il contenuto dell'header *X-XSRF-TOKEN*, verificando che il token ricevuto sia lo stesso di quello presente nel cookie.

Questa tecnica si basa sul presupposto per cui, grazie alla *Same Origin Policy*, non è possibile accedere ai cookie tramite codice Javascript da un dominio diverso. Inoltre, l'utilizzo dell'attributo *SameSite* settato a *strict* sul cookie, rafforza questo presupposto.

Angular permette di customizzare il cookie e l'header andando and impostare la configurazione *withXsrfConfiguration* nel provider *provideHttpClient*:

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withXsrfConfiguration({
        cookieName: MY-XSRF-TOKEN- COOKIE',
        headerName: 'X-MY-XSRF-TOKEN-HEADER,
      }),
    ),
  ]
};
```

Per maggiori informazioni fare riferimento a: [Cross-Site Request Forgery - Angular Dev]

## 2.3 Content Security Policy (CSP)

La Content Security Policy (CSP) è un meccanismo di sicurezza che serve a rafforzare la protezione delle applicazioni web contro attacchi comuni, come il Cross-Site Scripting (XSS). Questi tipi di attacchi possono portare a furto di dati, alterazione di siti web e distribuzione di malware.

Implementando la CSP, gli amministratori di sistema possono specificare quali domini sono considerati fonti affidabili per l'esecuzione di script ed il caricamento di risorse, limitando così i vettori attraverso i quali possono verificarsi attacchi XSS.
Inoltre, la CSP permette di definire quali protocolli possono essere utilizzati per caricare i contenuti, promuovendo l'uso di HTTPS per una trasmissione sicura dei dati.

Per abilitare la CSP, è necessario configurare il server web in modo che restituisca l'header HTTP Content-Security-Policy. Alternativamente, è possibile utilizzare l'elemento `<meta>` all'interno di una pagina HTML.

La configurazione minima di una CSP per una applicazione Angular è la seguente:

```
default-src 'self'; style-src 'self' 'nonce-randomNonce'; script-src 'self' 'nonce-randomNonce';
```

Questa configurazione prevede l'utilizzo dell'approccio basato su **nonce**: il server in ogni risposta HTTP, nella CSP, riporterà una stringa randomica, il nonce; per poter caricare script o stili, è necessario che questi elementi posseggano l'attributo *nonce* avvalorato con la stessa stringa presente nella CSP.

Per poter permettere il corretto caricamento delle risorse, il nonce va settato sull'elemento root di una applicazione Angular:

```
<app ngCspNonce="randomNonce">…</app>
```

È possibile ottenere il valore da iniettare nell'attributo *ngCspNonce* come segue:

```
import {bootstrapApplication, CSP_NONCE} from '@angular/core';
import {AppComponent} from './app/app.component';
bootstrapApplication(AppComponent, {
  providers: [{
    provide: CSP_NONCE,
    useValue: globalThis.myRandomNonceValue
  }]
});
```

Angular supporta e mette a disposizione delle policy **Trusted Types** built-in:

- *angular*
- *angular#unsafe-bypass*
- *angular#unsafe-jit*
- *angular#bundler*

Queste sono utilizzabili nella CSP attraverso la direttiva *trusted-types*.
Per maggiori dettagli fare riferimento a: Content Security Policy - Angular Dev.

## 2.4 Open Redirect

Un open redirect è una vulnerabilità di sicurezza web che si verifica quando un'applicazione web accetta un URL non convalidato come input e reindirizza l'utente a tale URL. Questo può essere sfruttato da attaccanti per indirizzare gli utenti verso siti web malevoli, facendo sembrare che il reindirizzamento provenga da una fonte affidabile. Gli open redirect possono essere utilizzati in attacchi di phishing e per rubare informazioni sensibili. È importante che gli sviluppatori convalidino e limitino i reindirizzamenti per prevenire tali vulnerabilità.

L'utilizzo delle interfacce DOM o della classe *Document* in Angular può rendere vulnerabile l'applicazione a questa tipologia di attacco se un input non sanitizzato derivante da un utente è utilizzato come parte di una redirect.

Segue un esempio di codice insicuro:

```
import { DOCUMENT } from '@angular/common';
export class AppComponent {
    constructor(@Inject(DOCUMENT) private document: Document) { }
    goToUrl(someUserControllableInput): void {

        this.document.location.href = someUserControllableInput;

        //Oppure
         window.location.replace(someUserControllableInput);

        //Oppure
         window.location.href = someUserControllableInput;

        //Oppure
         window.location.assign(someUserControllableInput);

        //Oppure
         window.open(someUserControllableInput, "_blank")
    }
}
```

La soluzione al precedente problema di sicurezza è quella di utilizzare la classe *Location*, che permette di effettuare redirect sicure:

```
import { Component, Inject } from '@angular/core';
import {Location, LocationStrategy, PathLocationStrategy} from '@angular/common';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [Location, {provide: LocationStrategy, useClass: PathLocationStrategy}],
})
export class AppComponent
{
        location: Location;
        constructor(location: Location) {
            this.location = location;
        }

     goToUrl(someUserControllableInput): void {
         this.location.go(someUserControllableInput);
    }
}
```

# 3 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

La validazione e la sanitizzazione degli input sono processi fondamentali nella sicurezza informatica e nello sviluppo software. La validazione consiste nel verificare che i dati inseriti dagli utenti rispettino determinati criteri predefiniti, come formato, lunghezza e tipo di dato. Questo aiuta a prevenire errori e comportamenti imprevisti nel sistema. La sanitizzazione, invece, si occupa di pulire e modificare gli input per rimuovere o neutralizzare contenuti potenzialmente dannosi, come script malevoli o caratteri speciali, proteggendo così il sistema da attacchi come l'injection di codice. Entrambi i processi sono essenziali per garantire l'integrità e la sicurezza delle applicazioni.

## 3.1 Built-in Sanitization

Angular considera di default come untrusted tutti i valori. Di conseguenza, quando un valore è inserito nel DOM tramite interpolazione o template binding, se non indicato diversamente, su questo verrà effettuata automaticamente una sanitizzazione ed un escaping dei caratteri HTML, prevenendo attacchi come il Cross-Site Scripting (XSS).

Segue un esempio di interpolazione in cui il valore passato viene automaticamente sanitizzato:

```
export class SafeHtmlComponent {
  // Input proveniente da un utente.
  userInput = '<script>alert("xss")</script>';
}
…
<div>{{ userInput }}</div>
```

Il risultato della sanitizzazione a seguito della interpolazione precedente è il seguente:

```
&lt;script&gt;alert(&quot;xss&quot;)&lt;/script&gt;
```

Quando è necessario che un valore venga interpretato come HTML, è possibile effettuare il binding con la property *innerHTML*. Quando viene passata una stringa contente HTML questa verrà interpretata; tuttavia, eventuali tag <*script*> verranno automaticamente rimossi da Angular:

```
export class HtmlComponent {
  someHtmlString = '<script>alert("xss")</script> <b>Ciao Mondo!</b>';
}
…
<div [innerHTML]=" someHtmlString"></div>
```

Il risultato della sanitizzazione a seguito del binding sulla property *innerHTML* precedente è il seguente:

```
<b>Ciao Mondo!</b>
```

**Nota**: il precedente codice HTML verrà quindi interpretato dal browser. **Questo implica che per un attaccante potrebbe comunque essere possibile riuscire ad effettuare un attacco XSS!**

Non bisogna mai utilizzare la property *innerHTML* con input provenienti da utenti. Lo stesso discorso è valido quando si utilizzano direttamente le API DOM attraverso l'elemento *document*.

Se questo è strettamente necessario, allora è fondamentale utilizzare una sanitizzazione esplicita attraverso la classe *DomSanitizer* che mette a disposizione il metodo *sanitize*.
Tale metodo prende in input un ***SecurityContext*** ed il valore da sanitizzare.

Angular definisce i seguenti *SecurityContext*, specificati nell'omonima classe enumerativa:

- **NONE**: utilizzando questo context, non verrà eseguita nessuna sanitizzazione. Generalmente l'utilizzo di questo context è sconsigliato a favore della sanitizzazione built-in.
- **HTML**: utilizzato quando è necessario interpretare un valore come HTML.
- **STYLE**: utilizzato quando è necessario associare del CSS nell'attributo *style* di un elemento
- **SCRIPT**: utilizzato quando è necessario utilizzare un valore nel contesto di uno script. Generalmente l'utilizzo di questo context è sconsigliato a favore della sanitizzazione built-in.
- **URL**: utilizzato quando il valore consiste in un URL impiegato, ad esempio, nell'attributo *href* di un anchor.
- **RESOURCE_URL**: utilizzato quando il valore è un URL che punta ad una risorsa che deve essere caricata ed interpretata, come ad esempio uno script remoto (attributo *src* di un tag <*script*>).

Segue un esempio di sanitizzazione esplicita:

```
import { Component, SecurityContext } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';

@Component({
  selector: 'app-example',
  template: \`<div [innerHtml]="safeHtml"></div>\`
})
export class ExampleComponent {
  safeHtml: any;

  constructor(private sanitizer: DomSanitizer) {
    const unsafeHtml = "<img src='x' onerror='alert(\"XSS!!\")'>";
    this.safeHtml = this.sanitizer.sanitize(SecurityContext.HTML, unsafeHtml);
  }
}
```

Quando è necessario includere un valore considerato *trusted* senza applicare una sanitizzazione, la classe *DomSanitizer* mette a disposizione i seguenti metodi, ciascuno dei quali bypassa la sanitizzazione che viene messa in atto per il rispettivo *SecurityContext*:

- *bypassSecurityTrustHtml*
- *bypassSecurityTrustScript*
- *bypassSecurityTrustStyle*
- *bypassSecurityTrustUrl*
- *bypassSecurityTrustResourceUrl*

**L'utilizzo di questi metodi può essere pericoloso quando applicato su input provenienti da un utente**.

Segue un esempio di codice vulnerabile:

```
import { Component, SecurityContext } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';

@Component({
  selector: 'app-example',
  template: \`<a [href]="someUserInput"></div>\`
})
export class ExampleComponent {
  // Valore controllabile da un utente
  someUserInput: any;

  constructor(private sanitizer: DomSanitizer) {
    …
    this.someUserInput = this.sanitizer.bypassSecurityTrustUrl(someUserInput);
  }
}
```

# 4 INJECTION

## 4.1 Template Injection

La Template Injection è una vulnerabilità che si verifica quando un attaccante riesce a iniettare codice malevolo in un template che viene processato lato server, in tal caso **Server-Side Template Injection (STTI)**, o lato client, in tal caso **Client-Side Template Injection (CSTI)**. Questa vulnerabilità si manifesta quando l'input dell'utente viene concatenato direttamente nel template, anziché essere passato come dato.

Gli attaccanti possono sfruttare questa falla per eseguire comandi non autorizzati sul server, potenzialmente ottenendo il controllo completo del sistema. Per prevenire SSTI, è fondamentale validare e sanificare correttamente l'input dell'utente prima di inserirlo nei template.

```
@Component({
template: "<div>Username: " + inputFromUser + "</div>"
})
export class HeaderComponent {
…
}
```

Un attaccante potrebbe iniettare il seguente payload "{{*constructor.constructor('alert(1)')()*}}" e generare una esecuzione di codice javascript nel browser.

La soluzione al precedente problema di sicurezza è quella di passare lo username come dato piuttosto che concatenarlo direttamente nel template:

```
@Component({
template: "<div>Username: {{data.username}} </div>"
})

export class HeaderComponent {
@Input() data: any;
…
}
```

Angular consiglia di utilizzare il compilatore Ahead of Time Compiler (AoT) per la compilazione offline dei templates. Questo permette di prevenire completamente questa classe di vulnerabilità:

```
ng build –aot
ng serve –aot
```

**Nota**: dalla versione 9 in poi, la compilazione AoT **è abilitata di default**.

## 4.2 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>
<script>
    var secret = window.secret;
    // Invia i dati esfiltrati su un listener predisposto da un attaccante
</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 un 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.

La libreria *HttpClient* di Angular supporta la convenzione che utilizza come prefisso la stringa *")]}',\n"* per indicare risposte contenenti JSON non eseguibili come Javascript, eseguendo lo strip del prefisso dalla risposta prima che questa venga processata ulteriormente da altre componenti.

## 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.

In questo contesto, Angular indica alcune linee guida da seguire:

- **Mantenere aggiornate le librerie Angular**: le librerie Angular sono costantemente aggiornate con fix per problemi di sicurezza.
- **Non alterare la copia di Angular**: versioni di Angular customizzate generalmente non includono le fix ed i miglioramenti di sicurezza e performance che vengono periodicamente rilasciati.
- **Evitare le API indicate come "Security Risk" nella documentazione**

## 6 LINK UTILI

- **Route Guards**
  - CanActivate - Angular Dev
- **Cross-Site Request Forgery (CSRF/XSRF)**
  - Cross-Site Request Forgery - Angular Dev
- **Content Security Policy**
  - Content Security Policy - Angular Dev
- **Built-in Sanitization**:
  - Preventing Cross-Site Scripting (XSS) - Angular Dev
  - SecurityContext - Angular Dev
  - DomSanitizer - Angular Dev
- **Template Injection**
  - Angular Security Best Practices - Snyk
  - Ahead of Time Compiler - Angular Dev
- **Cross-Site Script Inclusion (XSSI)**
  - Cross-Site Script Inclusion (XSSI) - Angular Dev
  - Testing for Cross-Site Script Inclusion - OWASP Web Security Testing Guide
- **Misconfigurations**
  - Best Practices - Angular Dev