Security

This topic describes Angular's built-in protections against common web-application vulnerabilities and attacks such as cross-site scripting attacks. It doesn't cover application-level security, such as authentication and authorization.

For more information about the attacks and mitigations described below, see OWASP Guide Project.

You can run the in Stackblitz and download the code from there.

{@a report-issues}

Reporting vulnerabilities

To report vulnerabilities in Angular itself, email us at security@angular.io.

For more information about how Google handles security issues, see Google's security philosophy.

{@a best-practices}

Best practices

- **Keep current with the latest Angular library releases.** We regularly update the Angular libraries, and these updates might fix security defects discovered in previous versions. Check the Angular change log for security-related updates.
- **Don't modify your copy of Angular.** Private, customized versions of Angular tend to fall behind the current version and might not include important security fixes and enhancements. Instead, share your Angular improvements with the community and make a pull request.
- **Avoid Angular APIs marked in the documentation as "*Security Risk*."** For more information, see the Trusting safe values section of this page.

Preventing cross-site scripting (XSS)

Cross-site scripting (XSS) enables attackers to inject malicious code into web pages. Such code can then, for example, steal user data (in particular, login data) or perform actions to impersonate the user. This is one of the most common attacks on the web.

To block XSS attacks, you must prevent malicious code from entering the DOM (Document Object Model). For example, if attackers can trick you into inserting a `<script>` tag in the DOM, they can run arbitrary code on your website. The attack isn't limited to `<script>` tags—many elements and properties in the DOM allow code execution, for example, `<img onerror="...">` and `<a href="javascript:...">`. If attacker-controlled data enters the DOM, expect security vulnerabilities.

Angular's cross-site scripting security model

To systematically block XSS bugs, Angular treats all values as untrusted by default. When a value is inserted into the DOM from a template binding, or interpolation, Angular sanitizes and escapes untrusted values. If a value was already sanitized outside of Angular and is considered safe, communicate this to Angular by marking the value as trusted.

Unlike values to be used for rendering, Angular templates are considered trusted by default, and should be treated as executable code. Never generate templates by concatenating user input and template syntax. Doing this would enable attackers to inject arbitrary code into your application. To prevent these vulnerabilities, always use the default AOT template compiler in production deployments.

An additional layer of protection can be provided through the use of Content security policy and Trusted Types. These web platform features operate at the DOM level which is the most effective place to prevent XSS issues because they can't be bypassed using other, lower-level APIs. For this reason, we strongly encourage developers to take advantage of these features by configuring the content security policy for their application and enabling trusted types enforcement.

Sanitization and security contexts

Sanitization is the inspection of an untrusted value, turning it into a value that's safe to insert into the DOM. In many cases, sanitization doesn't change a value at all. Sanitization depends on context: a value that's harmless in CSS is potentially dangerous in a URL.

Angular defines the following security contexts:

- **HTML** is used when interpreting a value as HTML, for example, when binding to `innerHtml` .
- **Style** is used when binding CSS into the `style` property.
- **URL** is used for URL properties, such as `<a href>` .
- **Resource URL** is a URL that is loaded and executed as code, for example, in `<script src>` .

Angular sanitizes untrusted values for HTML, styles, and URLs; sanitizing resource URLs isn't possible because they contain arbitrary code. In development mode, Angular prints a console warning when it has to change a value during sanitization.

Sanitization example

The following template binds the value of `htmlSnippet` , once by interpolating it into an element's content, and once by binding it to the `innerHTML` property of an element:

Interpolated content is always escaped—the HTML isn't interpreted and the browser displays angle brackets in the element's text content.

For the HTML to be interpreted, bind it to an HTML property such as `innerHTML` . But binding a value that an attacker might control into `innerHTML` normally causes an XSS vulnerability. For example, one could execute JavaScript in a following way:

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the `script` element but keeps safe content such as the `<b>` element.

A screenshot showing interpolated and bound HTML values

Direct use of the DOM APIs and explicit sanitization calls

Unless you enforce Trusted Types, the built-in browser DOM APIs don't automatically protect you from security vulnerabilities. For example, `document` , the node available through `ElementRef` , and many third-party APIs contain unsafe methods. In the same way, if you interact with other libraries that manipulate the DOM, you likely won't have the same automatic sanitization as with Angular interpolations. Avoid directly interacting with the DOM and instead use Angular templates where possible.

For cases where this is unavoidable, use the built-in Angular sanitization functions. Sanitize untrusted values with the DomSanitizer.sanitize method and the appropriate `SecurityContext` . That function also accepts values that were marked as trusted using the `bypassSecurityTrust` ... functions, and will not sanitize them, as described below.

{@a bypass-security-apis}

Trusting safe values

Sometimes applications genuinely need to include executable code, display an `<iframe>` from some URL, or construct potentially dangerous URLs. To prevent automatic sanitization in any of these situations, tell Angular that you inspected a value, checked how it was generated, and made sure it will always be secure. But *be careful*. If you trust a value that might be malicious, you are introducing a security vulnerability into your application. If in doubt, find a professional security reviewer.

To mark a value as trusted, inject `DomSanitizer` and call one of the following methods:

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`
- `bypassSecurityTrustStyle`
- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

Remember, whether a value is safe depends on context, so choose the right context for your intended use of the value. Imagine that the following template needs to bind a URL to a `javascript:alert(...)` call:

Normally, Angular automatically sanitizes the URL, disables the dangerous code, and in development mode, logs this action to the console. To prevent this, mark the URL value as a trusted URL using the `bypassSecurityTrustUrl` call:

A screenshot showing an alert box created from a trusted URL

If you need to convert user input into a trusted value, use a component method. The following template lets users enter a YouTube video ID and load the corresponding video in an `<iframe>`. The `<iframe src>` attribute is a resource URL security context, because an untrusted source can, for example, smuggle in file downloads that unsuspecting users could execute. So call a method on the component to construct a trusted video URL, which causes Angular to let binding into `<iframe src>`:

{@a content-security-policy}

Content security policy

Content Security Policy (CSP) is a defense-in-depth technique to prevent XSS. To enable CSP, configure your web server to return an appropriate `Content-Security-Policy` HTTP header. Read more about content security policy at the Web Fundamentals guide on the Google Developers website.

The minimal policy required for brand new Angular is:

```
default-src 'self'; style-src 'self' 'unsafe-inline';
```

- The `default-src 'self';` section allows the page to load all its required resources from the same origin.
- `style-src 'self' 'unsafe-inline';` allows the page to load global styles from the same origin (`'self'`) and enables components to load their styles (`'unsafe-inline'` - see `angular/angular#6361`).

Angular itself requires only these settings to function correctly. As your project grows, however, you may need to expand your CSP settings beyond this minimum to accommodate additional features specific to your application.

{@a trusted-types}

Enforcing Trusted Types

We recommend the use of Trusted Types as a way to help secure your applications from cross-site scripting attacks. Trusted Types is a web platform feature that can help you prevent cross-site scripting attacks by enforcing safer coding practices. Trusted Types can also help simplify the auditing of application code.

Trusted Types might not yet be available in all browsers your application targets. In the case your Trusted-Types-enabled application runs in a browser that doesn't support Trusted Types, the functionality of the application will be preserved, and your application will be guarded against XSS by way of Angular's DomSanitizer. See caniuse.com/trusted-types for the current browser support.

To enforce Trusted Types for your application, you must configure your application's web server to emit HTTP headers with one of the following Angular policies:

- `angular` - This policy is used in security-reviewed code that is internal to Angular, and is required for Angular to function when Trusted Types are enforced. Any inline template values or content sanitized by Angular is treated as safe by this policy.
- `angular#unsafe-bypass` - This policy is used for applications that use any of the methods in Angular's DomSanitizer that bypass security, such as `bypassSecurityTrustHtml` . Any application that uses these methods must enable this policy.
- `angular#unsafe-jit` - This policy is used by the JIT compiler. You must enable this policy if your application interacts directly with the JIT compiler or is running in JIT mode using the platform browser dynamic.

You should configure the HTTP headers for Trusted Types in the following locations:

- Production serving infrastructure
- Angular CLI (`ng serve`), using the `headers` property in the `angular.json` file, for local development and end-to-end testing
- Karma (`ng test`), using the `customHeaders` property in the `karma.config.js` file, for unit testing

The following is an example of a header specifically configured for Trusted Types and Angular:

Content-Security-Policy: trusted-types angular; require-trusted-types-for 'script';

The following is an example of a header specifically configured for Trusted Types and Angular applications that use any of the methods in Angular's DomSanitizer that bypasses security.

Content-Security-Policy: trusted-types angular angular#unsafe-bypass; require-trusted-types-for 'script';

The following is an example of a header specifically configured for Trusted Types and Angular applications using JIT:

Content-Security-Policy: trusted-types angular angular#unsafe-jit; require-trusted-types-for 'script';
Community contributions
To learn more about troubleshooting Trusted Type configurations, the following resource might be helpful:

Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types

{@a offline-template-compiler}

Use the AOT template compiler

The AOT template compiler prevents a whole class of vulnerabilities called template injection, and greatly improves application performance. The AOT template compiler is the default compiler used by Angular CLI applications, and you should use it in all production deployments.

An alternative to the AOT compiler is the JIT compiler which compiles templates to executable template code within the browser at runtime. Angular trusts template code, so dynamically generating templates and compiling them, in particular templates containing user data, circumvents Angular's built-in protections and is a security anti-pattern. For information about dynamically constructing forms in a safe way, see the Dynamic Forms guide.

{@a server-side-xss}

Server-side XSS protection

HTML constructed on the server is vulnerable to injection attacks. Injecting template code into an Angular application is the same as injecting executable code into the application: it gives the attacker full control over the application. To prevent this, use a templating language that automatically escapes values to prevent XSS vulnerabilities on the server. Don't generate Angular templates on the server side using a templating language; doing this carries a high risk of introducing template-injection vulnerabilities.

{@a http}

HTTP-level vulnerabilities

Angular has built-in support to help prevent two common HTTP vulnerabilities, cross-site request forgery (CSRF or XSRF) and cross-site script inclusion (XSSI). Both of these must be mitigated primarily on the server side, but Angular provides helpers to make integration on the client side easier.

{@a xsrf}

Cross-site request forgery

In a cross-site request forgery (CSRF or XSRF), an attacker tricks the user into visiting a different web page (such as `evil.com`) with malignant code that secretly sends a malicious request to the application's web server (such as `example-bank.com`).

Assume the user is logged into the application at `example-bank.com`. The user opens an email and clicks a link to `evil.com`, which opens in a new tab.

The `evil.com` page immediately sends a malicious request to `example-bank.com`. Perhaps it's a request to transfer money from the user's account to the attacker's account. The browser automatically sends the `example-bank.com` cookies (including the authentication cookie) with this request.

If the `example-bank.com` server lacks XSRF protection, it can't tell the difference between a legitimate request from the application and the forged request from `evil.com`.

To prevent this, the application must ensure that a user request originates from the real application, not from a different site. The server and client must cooperate to thwart this attack.

In a common anti-XSRF technique, the application server sends a randomly generated authentication token in a cookie. The client code reads the cookie and adds a custom request header with the token in all subsequent requests. The server compares the received cookie value to the request header value and rejects the request if the values are missing or don't match.

This technique is effective because all browsers implement the *same origin policy*. Only code from the website on which cookies are set can read the cookies from that site and set custom headers on requests to that site. That means only your application can read this cookie token and set the custom header. The malicious code on `evil.com` can't.

Angular's `HttpClient` has built-in support for the client-side half of this technique. Read about it more in the HttpClient guide.

For information about CSRF at the Open Web Application Security Project (OWASP), see Cross-Site Request Forgery (CSRF) and Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. The Stanford University paper Robust Defenses for Cross-Site Request Forgery is a rich source of detail.

See also Dave Smith's talk on XSRF at AngularConnect 2016.

{@a xssi}

Cross-site script inclusion (XSSI)

Cross-site script inclusion, also known as JSON vulnerability, can allow an attacker's website to read data from a JSON API. The attack works on older browsers by overriding built-in JavaScript object constructors, and then including an API URL using a `<script>` tag.

This attack is only successful if the returned JSON is executable as JavaScript. Servers can prevent an attack by prefixing all JSON responses to make them non-executable, by convention, using the well-known string `")]}',\n"`.

Angular's `HttpClient` library recognizes this convention and automatically strips the string `")]}',\n"` from all responses before further parsing.

For more information, see the XSSI section of this Google web security blog post.

{@a code-review}

Auditing Angular applications

Angular applications must follow the same security principles as regular web applications, and must be audited as such. Angular-specific APIs that should be audited in a security review, such as the *bypassSecurityTrust* methods, are marked in the documentation as security sensitive.