

Security in Angular

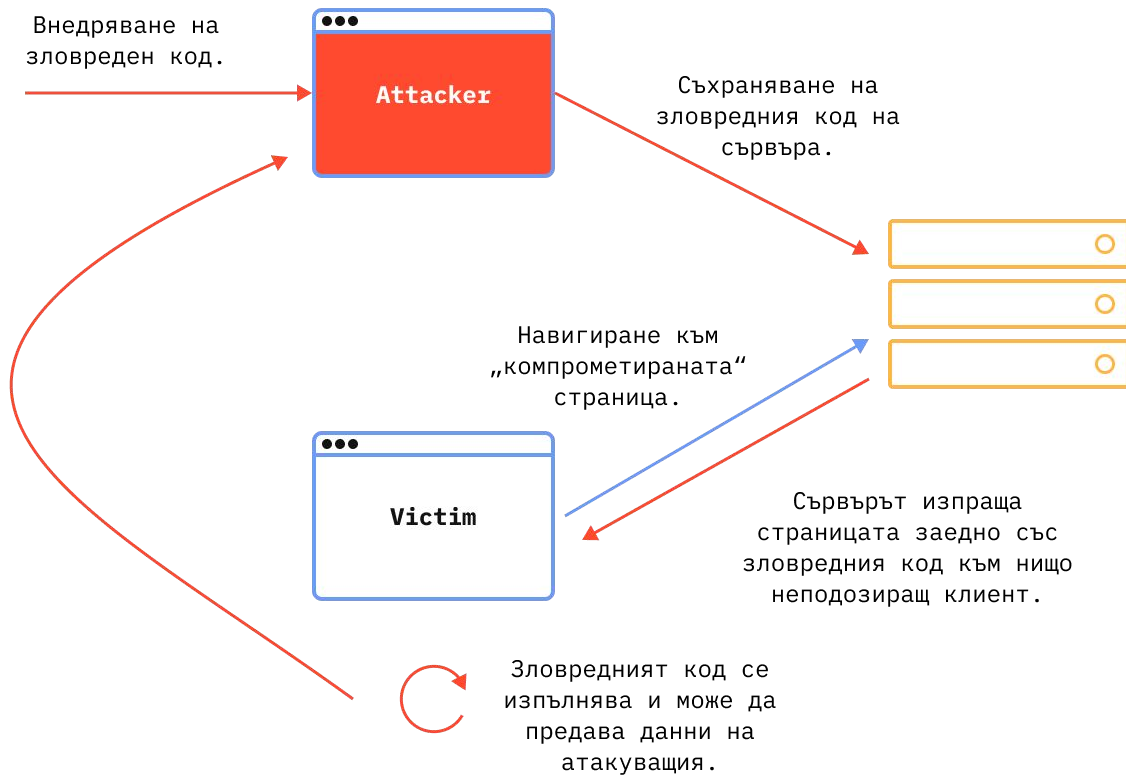
Лектор: Петър Маламов

Cross Site Scripting (XSS)

Какво е XSS?

Cross-Site Scripting (XSS) е тип уязвимост в уеб приложенията, при който атакуващият вкарва зловреден код (обикновено JavaScript) в уеб страници, които се изпълняват от браузъра на други потребители.

Тази уязвимост позволява на нападателя да изпълни скриптове в контекста на жертвата, което може да доведе до кражба на данни, поемане на контрол върху сесии или други вредни действия.



Как работи XSS?

1. **Инжектиране на зловреден код** - Атакуващият добавя JavaScript код чрез форми, URL параметри или други входни точки.
2. **Съхраняване на кода** - Зловредният код се обработва от сървъра и се включва в уеб страницата.
3. **Изпълнение на кода** - Когато жертвата отвори страницата, кодът се изпълнява в нейния браузър.
4. **Резултат** - Зловредният код извършва действия, като кражба на данни или пренасочване към зловреден сайт.

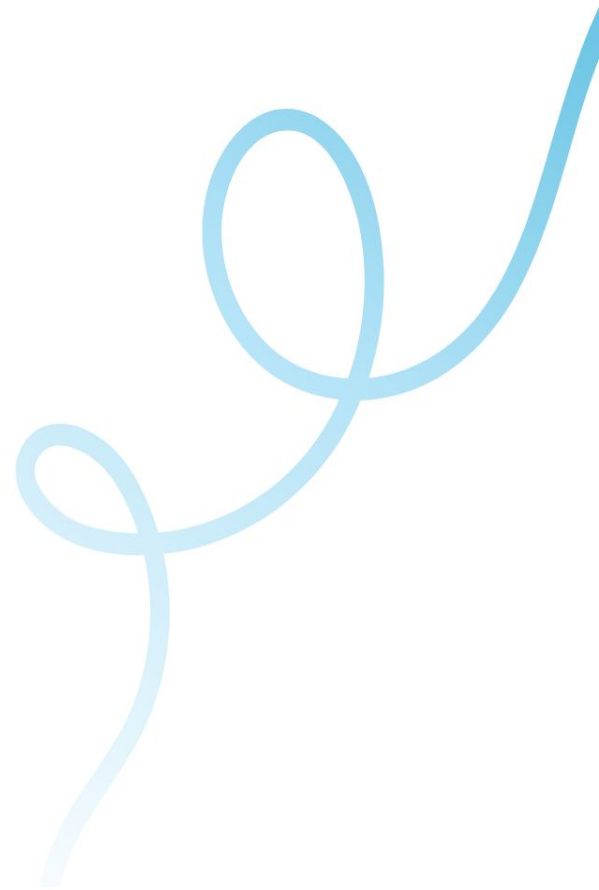
Цели на атаките

- Кражба на бисквитки (cookies).
- Подмяна на съдържание.
- Изпращане на потребителски данни към отдалечен сървър.
- Пренасочване към фишинг страници.

Типове XSS

- **Stored**- Зловредният код се записва на сървъра (напр. в база данни) и се изпълнява всеки път, когато потребителят отвори заразената страница.
- **Reflected**- Зловредният код се предава чрез URL параметри.
- **DOM-based**- Зловредният код се изпълнява директно в браузъра чрез манипулиране на DOM структурата, без да преминава през сървъра.

Пример



XSS в Angular

Angular блокира XSS уязвимости, като третира всички стойности като “ненадеждни” по подразбиране.

Когато стойност се вмъква в DOM чрез **template binding** или **interpolation**, Angular автоматично я почиства и ескейпва.

Angular прави това почистване чрез процес наречен **sanitization**(Санитизация).

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `<div>{{ userInput }}</div> `,
})
export class AppComponent {
  userInput = `<a href="javascript:alert('Hacking!')">Normal Link</a>`;
}
```

Стойността е
почистена и
ескейпната

`Normal Link`

Sanitization

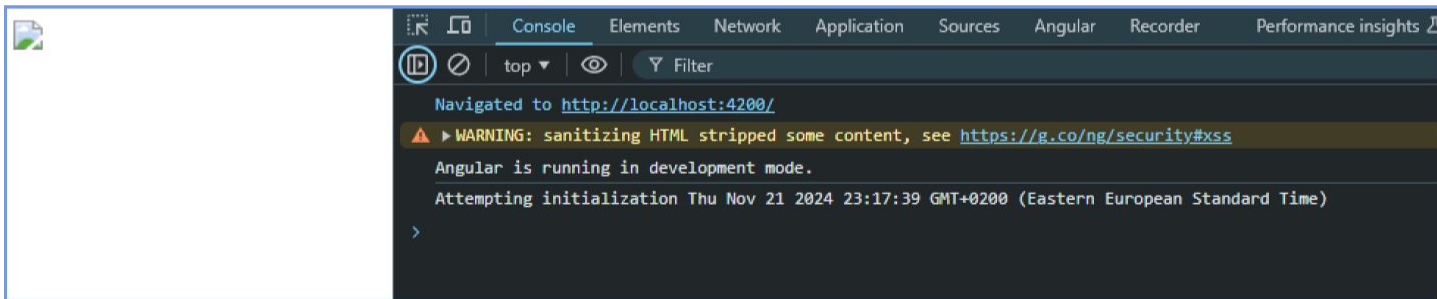
Процес на проверка на ненадеждна стойност и преобразуването ѝ в стойност, която е безопасна за вмъкване в DOM.

Angular автоматично почиства:

- Методи за вмъкване в DOM-а - **innerHTML**
- Начини за зареждане на допълнителни ресурси - **src, href, url**
- Event handlers

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `<div [innerHTML]="downloadedImg"></div> `,
})
export class AppComponent {
  downloadedImg = ``;
}
```



TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  template: `
    <div [innerHTML]="downloadedImg"></div>
    <div>{{ downloadedImg }}</div>
  `,
})
export class AppComponent {
  downloadedImg = ``;
}
```

Sanitization content



Escaped content

Важно!

Angular поддържа списък с безопасни елементи и атрибути, които могат да бъдат санитизирани.

```
// Safe Void Elements - HTML5  
// https://html.spec.whatwg.org/#void-elements  
const VOID_ELEMENTS = tagSet('area,br,col,hr,img,wbr');
```

```
// Attributes that have href and hence need to be sanitized  
export const URI_ATTRS = tagSet('background,cite,href,itemtype,longdesc,poster,src,xlink:href');
```

[packages/core/src/sanitization/html_sanitizer.ts](https://angular.io/packages/core/src/sanitization/html_sanitizer.ts)

TS app.component.ts

```
export class AppComponent {  
  downloadedImg = `}
```

Атрибутът `onerror`
не е безопасен и
бива премахнат.

```
// Safe Void Elements - HTML5  
// https://html.spec.whatwg.org/#void-elements  
const VOID_ELEMENTS = tagSet('area,br,col,hr,img,wbr');
```

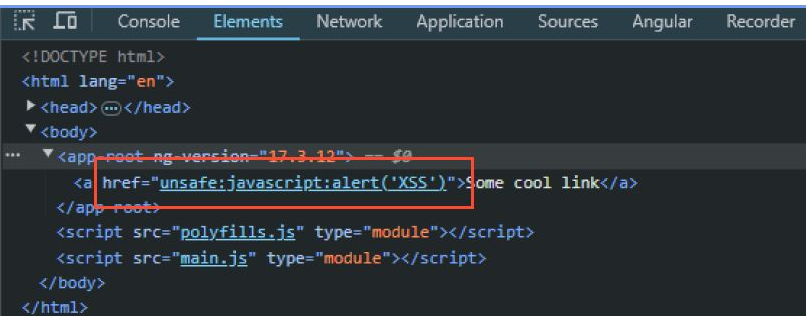
```
// Attributes that have href and hence need to be sanitized  
export const URI_ATTRS = tagSet('background,cite,href,itemtype,longdesc,poster,src,xlink:href');
```

Security Contexts

Начинът, по който Angular третира различни видове данни в зависимост от мястото, в което тези данни ще бъдат използвани в приложението.

Security contexts	Details
HTML	Used when interpreting a value as HTML, for example, when binding to <code>innerHTML</code> .
Style	Used when binding CSS into the <code>style</code> property.
URL	Used for URL properties, such as <code><a href></code> .
Resource URL	A URL that is loaded and executed as code, for example, in <code><script src></code> .

[Some cool link](#)



```
<!DOCTYPE html>
<html lang="en">
  <head>
  </head>
  <body>
    <app-root ng-version="17.3.12">
      <a href="unsafe:javascript:alert('XSS')">Some cool link</a>
    </app-root>
    <script src="polyfills.js" type="module"></script>
    <script src="main.js" type="module"></script>
  </body>
</html>
```

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `
    <a [href]="link">Some cool link</a>
  `,
})
export class AppComponent {
  link = `javascript:alert('XSS')`;
}
```

```
const SAFE_URL_PATTERN = /^(?!javascript:)(?:[a-z0-9+.-]+:|[^&:\/?#]*(?:[\/?#]|$))/i;
export function _sanitizeUrl(url: string): string {
  url = String(url);
  if (url.match(SAFE_URL_PATTERN)) return url;

  if (typeof ngDevMode === 'undefined' || ngDevMode) {
    console.warn(`WARNING: sanitizing unsafe URL value ${url} (see ${XSS_SECURITY_URL})`);
  }

  return 'unsafe:' + url;
}
```

[packages/core/src/sanitization/url_sanitizer.ts](#)

Важно!

Вградените DOM API в браузъра **не** осигуряват автоматична защита срещу сигурностни уязвимости.

Например, методите на document, елементите, достъпни чрез ElementRef, и много външни библиотеки съдържат несигурни методи.

За по-голяма сигурност е препоръчително да се избягва директно взаимодействие с DOM и да се използват Angular шаблони, когато е възможно.

TS app.component.ts

```
export class AppComponent implements AfterViewChecked {  
  @ViewChild('myDiv') div!: ElementRef<HTMLDivElement>;  
  
  ngAfterViewChecked(): void {  
    const element: HTMLElement = document.createElement('div');  
    element.innerHTML = `Angular`;  
    this.div.nativeElement.appendChild(element)  
  }  
}
```

Внедреният код **НЯМА** да
бъде проверен за
сигурност.

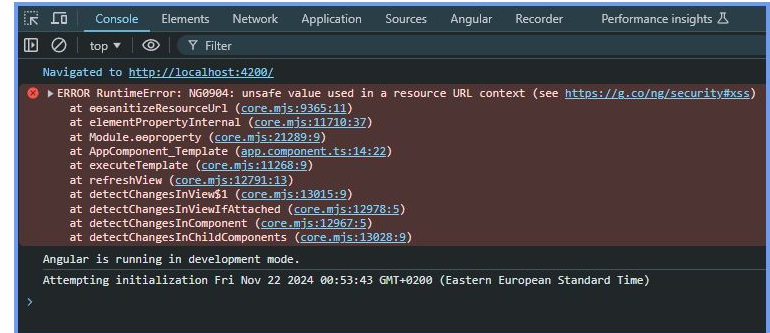
Избягване на автоматична санитизация

Понякога приложенията изискват включване на изпълним код или създаване на потенциално опасни URL адреси.

В тези случаи, за да предотвратите автоматична санитизация, Angular трябва да бъде информиран, че стойността е прегледана, проверена и е безопасна.

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `<iframe [src]="link" frameborder="0"></iframe>`,
})
export class AppComponent {
  link = 'https://www.youtube.com/embed/some-video-id';
}
```



Проверка на стойност

За да маркираме стойност като безопасна, трябва да използваме DomSanitizer и съответната функция за проверка, в зависимост от security context-а.

```
abstract class DomSanitizer implements Sanitizer {  
    abstract sanitize(context: SecurityContext, value: string | SafeValue | 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;  
}
```

TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: `<iframe [src]="safeLink" frameborder="0"></iframe>`,
})
export class AppComponent implements OnInit {
  unsafeLink = 'https://www.youtube.com/embed/JvkX2_46gUY?si=i5pksC4ynmUisByT';
  safeLink!: SafeResourceUrl;

  constructor(private sanitizer: DomSanitizer) {}

  ngOnInit(): void {
    this.safeLink = this.sanitizer.bypassSecurityTrustResourceUrl(
      this.unsafeLink
    );
  }
}
```


Content security policy(CSP)

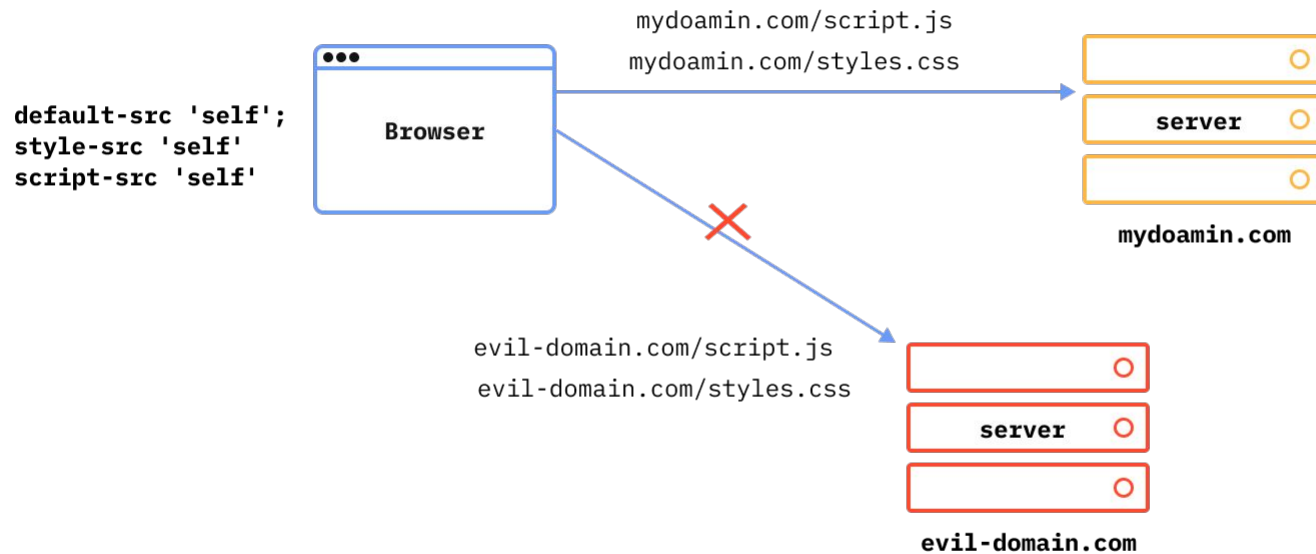
Механизъм за уеб сигурност, който помага за предотвратяване на атаки като Cross-Site Scripting (XSS), data injection и други уязвимости, свързани с изпълнението на неоторизиран код в уеб страници.

Как работи CSP ?

CSP работи чрез задаване на правила, които определят източниците на съдържание, които браузърът има право да зарежда и изпълнява. Тези правила се конфигурират от сървъра и се изпращат към браузъра като HTTP хедър или <meta> таг.

Content-Security-Policy: default-src 'self'; script-src 'self' trusted-scripts.com;

Браузърът няма да зареди или изпълни ресурси от източници, които не са изрично разрешени от CSP.



Контрол на източниците

CSP предоставя набор от директиви за прецизен контрол върху ресурсите, които дадена страница може да зарежда.

Директивите посочва допустимите източници за:

- font-src - шрифтове.
- img-src - стилове.
- script-src - скриптове.
- worker-src - workers и service workers.
- [И други.](#)

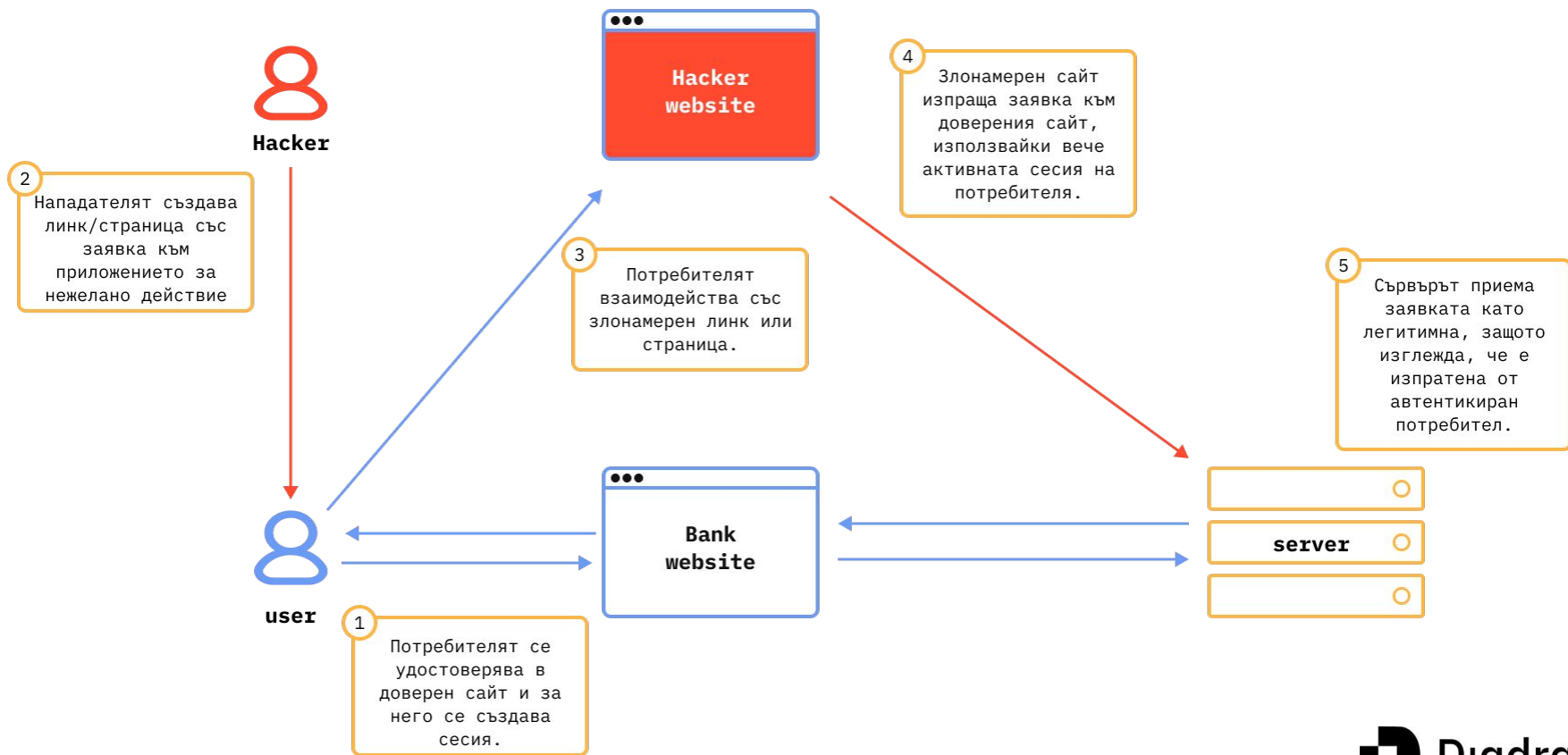
Препоръки

- Винаги започвайте с правилото **default-src 'self'**; и постепенно добавяйте източници според нуждите.
- Използвайте инструмента за отчети на CSP (**report-uri** или **report-to**), за да следите какво съдържание се блокира и да настройвате правилата.
- Тествайте конфигурацията на CSP в различни среди, за да избегнете прекъсвания в работата на приложението.

Cross-site request forgery (CSRF/XSRF)

Какво е CSRF?

Cross-Site Request Forgery (CSRF) е вид уязвимост в уеб приложенията, при която злонамерената страна подвежда потребителя да изпрати нежелана или неоторизирана заявка към уеб приложение, в което вече е автентикиран.



Как работи CSRF

1. Потребителят е влязъл в доверен сайт (**example-bank.com**).
2. Нападателят изпраща линк или съобщение, което съдържа линк към злонамерен сайт **evil.com**.
3. Потребителят случайно кликва върху линка и отваря този сайт в нов таб.
4. Страницата на evil.com автоматично изпраща зловредна заявка към example-bank.com
5. Браузърът автоматично добавя всички cookies за example-bank.com, включително и идентификационния cookie за автентикация, в зловредната заявка.
6. Ако сървърът на example-bank.com не разполага с защита срещу CSRF, той няма начин да различи истинската заявка от нормалния потребител от зловредната заявка от evil.com.

Цели на атаките

- Извършване на неоторизирани действия от името на жертвата.
- Измама с лични данни.
- Достъп до чувствителна информация.

Как да се защитим от CSRF?

За да се предотврати **CSRF**, приложението трябва да гарантира, че заявката на потребителя идва от реалното приложение, а не от различен сайт. Сървърът и клиентът трябва да работят заедно, за да осуетят атаката.

Използване на sameSite

Задаването на **SameSite: Strict** предотвратява изпращането на сесийни cookies от външни сайтове. Тези cookies се изпращат само ако домейнът на сайта съвпада с този, за който е зададен cookie-ът.

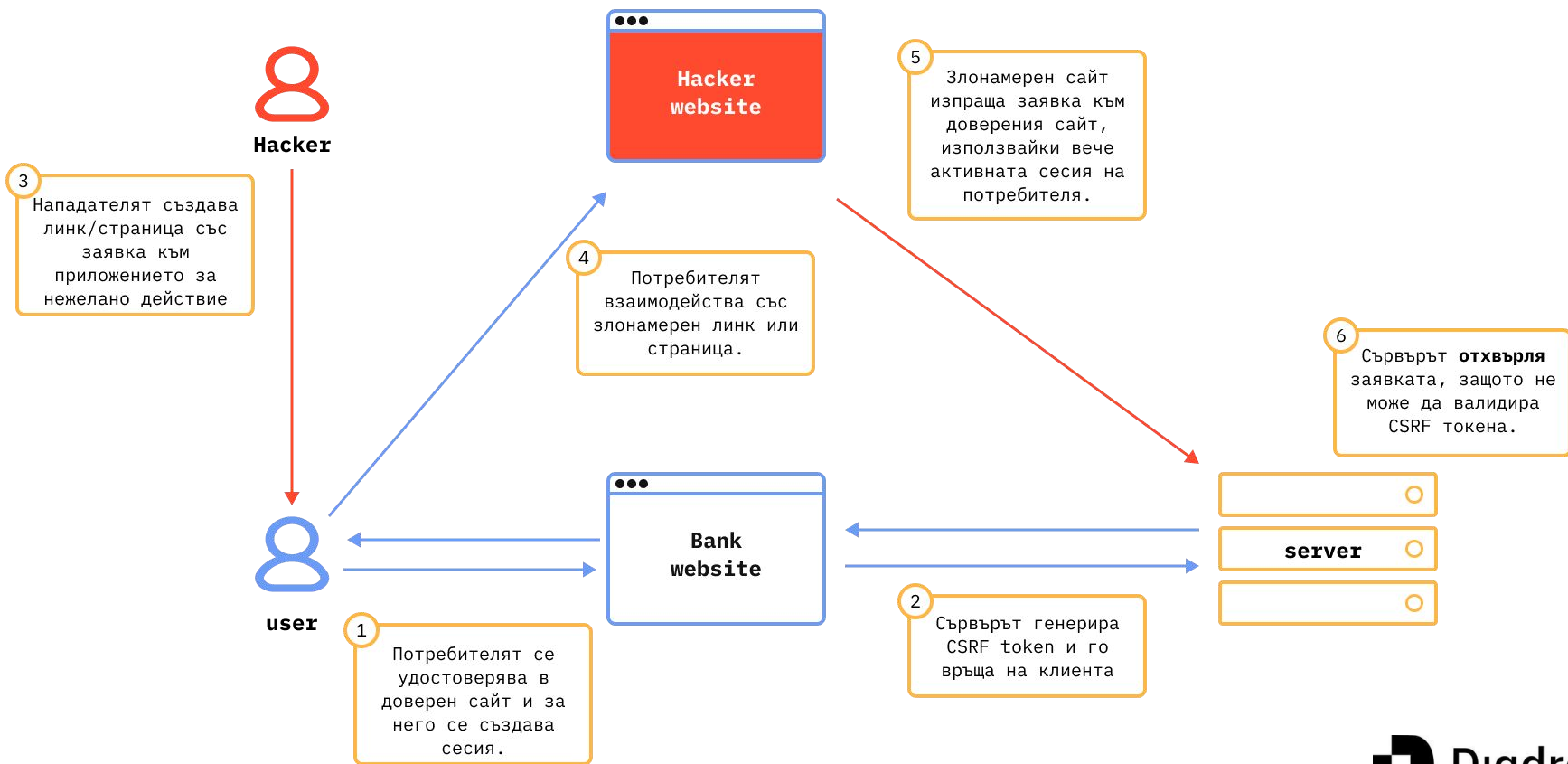
TS server.ts

```
server.get('/', (req,res)=>{  
  res.cookie('cookieName', 'cookieValue', { sameSite: 'strict', secure: true})  
  //.....Other Code  
})
```

Имплементиране на CSRF токен

Сървърът на приложението изпраща случайно генериран токен за автентикация в **cookie**. Клиентският код прочита този cookie и добавя персонализиран хедър с токена във всички последващи заявки. Сървърът сравнява стойността на cookie с хедъра на заявката и отхвърля заявката, ако стойностите липсват или не съвпадат.

Техниката е ефективна, защото всички браузъри следват **same-origin policy**. Само кодът от сайта, на който са зададени cookies, може да чете тези cookies и да задава персонализирани хедъри за заявки към същия сайт.



Имплементиране на CSRF токен в Angular

HttpClient поддържа механизъм за предотвратяване на CSRF атаки. При извършване на HTTP заявки, интерсептор чете токена от cookie (по подразбиране **XSRF-TOKEN**) и го поставя като HTTP хедър **X-XSRF-TOKEN**.

Тъй като само кодът от същия домейн може да чете това cookie, сървърът може да бъде сигурен, че заявката е изпратена от клиентското приложение, а не от нападател.

Важно!

По подразбиране, интерсепторът изпраща този хедър при всички заявки, които променят данни (като POST), но не и при GET заявки.

Поради естеството на CSRF атаките, които преминават през домейни, **same-origin policy** ще предотврати атакуващата страница да извлече резултатите от автентикирани GET заявки.

Конфигуриране на CSRF токена

По подразбиране Angular търси **XSRF-TOKEN** и го поставя като **X-XSRF-TOKEN** в HTTP хедърите.

Има възможност да конфигурираме името на токена и хедъра, който се задава в HTTP заявките.

TS app.config.ts

```
export const appConfig: ApplicationConfig = {  
  providers: [  
    provideHttpClient(  
      withXsrfConfiguration({  
        cookieName: 'CUSTOM_XSRF_TOKEN',  
        headerName: 'X-Custom-Xsrf-Header',  
      })  
    ),  
  ],  
};
```

TS app.config.ts

```
export const appConfig: ApplicationConfig = {  
  providers: [  
    provideHttpClient(  
      withNoXsrfProtection(),  
    ),  
  ],  
};
```

Премахване на CSRF
токена

Authorization

Accidental evaluation of privilege

Ситуация, в която потребител, който няма необходимото ниво на достъп, може да вижда данни, които не би трябвало да може или да изпълнява действия, които не са му разрешени.

Това може да възникне в резултат на неправилно управление на правата на достъп или недостатъчна проверка на привилегиите на потребителите, което води до неумишлено разкриване на чувствителни данни или позволяване на неоторизирани действия.

Спарвяне с **evaluation of privilege**

Angular предоставя механизми, с които можем да ограничим нивото на достъп за определени потребители.

Можем да ограничим нивото на достъп чрез:

- Route guards
- Lazy loading
- Interceptors

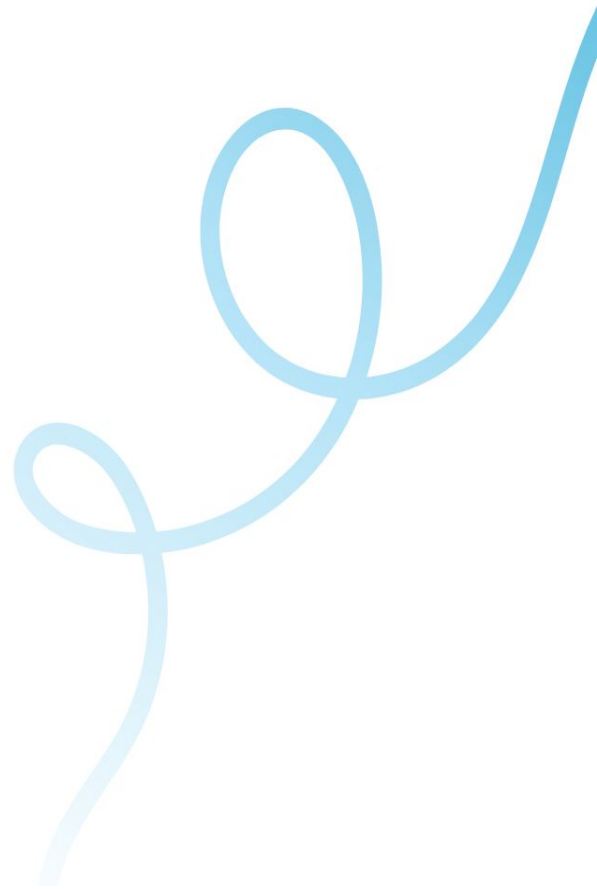
Route guards

Механизми, които се използват за контрол на достъпа до различни маршрути.

Те позволяват да се проверяват условия преди да се извърши навигация към даден маршрут, като така осигуряват защита срещу нежелан достъп или действия.

Типове Route guards

- CanActivate
- CanActivateChild
- CanDeactivate
- Resolve
- CanLoad



CanActivate

TS app.routes.ts

```
export const routes: Routes = [  
  { path: 'protected', component: ProtectedComponent, canActivate: [AuthGuard] },  
];
```

Проверява дали даден маршрут може да бъде активиран

CanActivateChild

TS app.routes.ts

```
export const routes: Routes = [  
  { path: 'protected', component: ProtectedComponent, canActivateChild: [AuthGuard] },  
];
```

Проверява дали под-маршрут
може да бъде активиран

CanDeactivate

TS component.ts

```
export type CanDeactivateType = Observable<boolean>;  
...  
export interface CanComponentDeactivate {  
  canDeactivate: () => CanDeactivateType;  
}  
  
export const canDeactivateGuard: CanDeactivateFn<CanComponentDeactivate> = (  
  component: CanComponentDeactivate  
) => {  
  return component.canDeactivate ? component.canDeactivate() : true;  
};
```

TS app.routes.ts

```
export const routes: Routes = [  
  {  
    path: 'my-route',  
    component: MyRouteComponent,  
    canDeactivate: [canDeactivateGuard],  
  },  
];
```

Проверява дали потребителят
може да напусне текущия
маршрут

Resolve

TS app.routes.ts

```
export const routes: Routes = [
  {
    path: 'products',
    component: ProductsComponent,
    resolve: { products: ProductsResolverService },
  },
];
```

Извършва асинхронни операции
преди зареждането на даден
маршрут.

CanLoad

TS app.routes.ts

```
export const routes: Routes = [
  {
    path: 'feature',
    loadChildren: () =>
      import('./feature/feature.module').then((m) => m.FeatureModule),
    canLoad: [AuthGuard],
  },
];
```

Проверява дали даден модул
трябва да бъде lazy-
loaded.

OWASP

Open Web Application Security Project (OWASP)

OWASP е глобална организация, която се фокусира върху подобряване на сигурността на софтуерните приложения.

Целта на OWASP е да предоставя безплатни ресурси, инструменти и знания за разработчиците и организациите, които да им помогнат да изградят сигурни приложения и да идентифицират и минимизират рисковете и уязвимостите в тях.

OWASP Web Security Checklist

Ръководство, което съдържа набор от препоръки и най-добри практики за сигурността на уеб приложения.

OWASP Web Security Checklist обхваща:

- Защита на данни
- Аутентикация и управление на сесии
- Управление на грешки и логове
- Контрол на достъпа
- Използване на сигурни библиотеки и компоненти
- Защита от уязвимости като инжекции и скриптове

[OWASP Web Security Checklist](#)

Благодаря за вниманието!