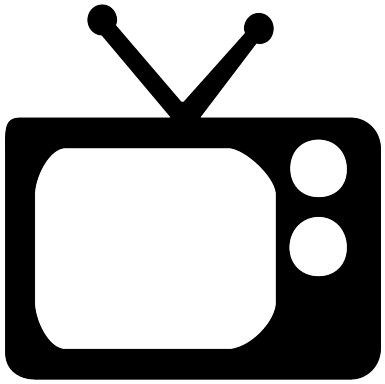

Proyecto Final

Keter Vulnerability

Índice

1. Introducción Teórica

Keter Vulnerability es una página web diseñada para representar algunos errores de configuración a la hora de crear aplicaciones web. El objetivo es disponer a los usuarios de diferentes **retos** donde deberán usar sus conocimientos y su capacidad de búsqueda a través de Internet para explotar dichas vulnerabilidades.

Figura 1: Fuente whiteknightit.com

El objetivo es ser capaces de concienciar a jóvenes **desarrolladores** del peligro que puede llevar realizar páginas web sin conciencia sobre los errores. Al final veremos como el usuario es un factor bastante peligroso para los desarrolladores y como nunca podemos confiar plenamente en sus intenciones.

Es por ello que debemos protegernos con las últimas tecnologías. Mantenernos **actualizados y activos** será la clave para evitar cualquier error.

El objetivo de esta página es su escalabilidad. Una vez terminado el proyecto la página pasará a ser código abierto, con el objetivo de que la comunidad pueda aportar sus propios retos y módulos desde GitHub. Es por ello que gran parte de los contenidos serán en inglés.

Haremos uso de las últimas tecnologías para la realización de este proyecto:

- **LaTex**: haremos uso de esta herramienta de texto mediante código para la creación de esta misma documentación.
- **NodeJS**: utilizaremos este paquete de recurso para la creación de la página web.
- **Mongo Atlas**: nos prooverá una base de datos principal para retos de NoSQL Injection.
- **HerokuApps**: será la página que hosteará nuestra aplicación.
- **GitHub**: allí subiremos el código y utilizaremos la función de GitHub Pages para crear una pequeña página web que muestre un breve resumen.
- **Docker**: crearemos un proyecto adicional con un contenedor en Docker para poder montar y utilizar nuestra aplicación en local.

2. Introducción Técnica

3. API

4. APP

La app **Keter Vulnerability** está creada mediante Typescript, compilada en JavaScript. Utiliza el framework de **Angular**, junto a **Angular Material** y **Bootstrap**.

4.1. Rutas

Para poder tener una mayor optimización he separado las distintas rutas, de esta forma tan solo cargarán unas páginas dependiendo de en que parte de la aplicación se encuentre. Estas rutas son las primeras, se encuentran en el archivo por defecto creado por Angular.

```
const routes: Routes = [
  { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
  { path: 'ctf', redirectTo: 'dashboard', pathMatch: 'full' },
  {
    path: 'dashboard',
    loadChildren: () =>
      import('./components/dashboard/dashboard.module').then(
        (x) => x.DashboardModule
      ),
  },
  {
    path: 'ctf',
    loadChildren: () =>
      import('./components/ctf/ctf.module').then((x) => x.CtfModule),
  },
  { path: '*', redirectTo: 'dashboard', pathMatch: 'full' },
];
```

Figura 2: rutas principales.

La primera declaración redirige al **dashboard** cualquier ruta vacia. La segunda redirige la ruta **ctf** al dashboard igualmente. La siguiente ruta es la del dashboard, en vez de cargar entero el dashboard y todos sus componentes cargamos únicamente la propia página, de igual manera que hacemos con el **ctf**.

Veremos a continuación las rutas del dashboard:

```
const routes: Routes = [
  {
    path: '',
    component: DashboardComponent,
    children: [
      { path: '', component: HomeComponent },
      { path: 'challenges', component: ChallengesComponent },
    ],
  },
];
```

Figura 3: rutas hijas del Dashboard.

Estas rutas son cargadas únicamente cuando se ingresa al Dashboard. De esta forma podemos tener la app dividida en dos partes. Por otro lado el **ctf** tiene su propio archivo de rutas del que parten los demás componentes, esto nos permite ahorrar la carga de todos los retos cuando tan solo hemos entrado al Dashboard. De igual manera, mientras nos encontremos en un reto no tendremos el Dashboard cargando también.

4.2. Módulos

Hemos separado también los módulos para una carga más rápida. Mantenemos el archivo por defecto de los módulos pero importamos un nuevo módulo que hemos creado nosotros: **Shared**.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { SharedModule } from './components/shared/shared.module';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    SharedModule,
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Figura 4: **app.module.ts** por defecto.

En dicho módulo **Shared** importamos todos los módulos que usamos para el proyecto, de esta forma podemos dividir los archivos y tener un mayor control de errores.

4.3. DOMsanitizer

Angular por defecto viene protegido contra ciertas vulnerabilidades, para poder recrearlas ha sido necesario escapar algunos de estas protecciones, como por ejemplo **DOMsanitizer**. Para poder escapar la seguridad lo primero que haremos será usar los formularios de Angular

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { DomSanitizer } from '@angular/platform-browser';

@Component({
  selector: 'app-xss1',
  templateUrl: './xss1.component.html',
  styleUrls: ['./xss1.component.css'],
})
export class Xss1Component implements OnInit {
  form: FormGroup;

  constructor(
    private fb: FormBuilder,
    private sanitizer: DomSanitizer,
  ) {
    this.form = this.fb.group({
      input: ['', Validators.required],
    });
    this.form.value.input = sanitizer.bypassSecurityTrustResourceUrl('');
  }
```

Figura 5: desactivar DOMsanitizer.

para poder tomar el valor en una variable. Importaremos la librería de DOMsanitizer y estableceremos el valor como fiable, de esta forma podremos explotar algunas vulnerabilidades, como XSS.

5. Vulnerabilidades utilizadas

5.1. XSS

Cross-Site Scripting son un tipo de ataque mediante inyección, mediante el cual código malicioso es inyectado en páginas confiables. Los ataques de XSS ocurren cuando el atacante usa una aplicación web para enviar código malicioso, generalmente en forma de script de navegador, para un tercer usuario. Puede ocurrir en multitud de aplicaciones web mediante la entrada de un usuario, sin validar o codificar su valor.

Fuente: OWASP Cross Site Scripting (XSS)

AÑADIR TIPOS

6. Explotaciones

6.1. Send a comment!

La explotación de este reto consiste en utilizar la vulnerabilidad de XSS Reflected.

Con tan solo poner un comando entre las etiquetas **¡script¿...¡/script¿** nos dará por solucionado el reto.

7. Posibles soluciones

7.1. Send a comment!

Para esta vulnerabilidad la solución consiste en depurar la entrada del usuario. En vez de usar DOMsanitizer para desactivarlo, si no lo nombramos por defecto estaría activado. Por asegurarnos podemos especificar que depure esa parte. Para ello debemos usar la siguiente estructura:

```
this.form.value.input = sanitizer.sanitize('');
```

en sustitución del código que teniamos anteriormente:

```
this.form.value.input = sanitizer.bypassSecurityTrustResourceUrl('');
```

Nota: DOMsanitizer tiene distintos formatos y funciones, no se limita únicamente a estas dos, por lo que en algunos casos puede ser mejor utilizar otras.

8. Conclusión

9. Fuentes

Info

- OWASP
- Angular DOMsanitizer
-
- Angular DOMsanitizer by Netanel Basal

Github

- Angular
- PayloadsAllTheThings

Recursos

- Angular Material