Buenas prácticas para el desarrollo web

Sergio De Luca

Agosto 2022

Práctica Professional Supervisada
Carrera: Analista en Tecnologías de la Información y la Comunicación
Tutor: Matías Murieta

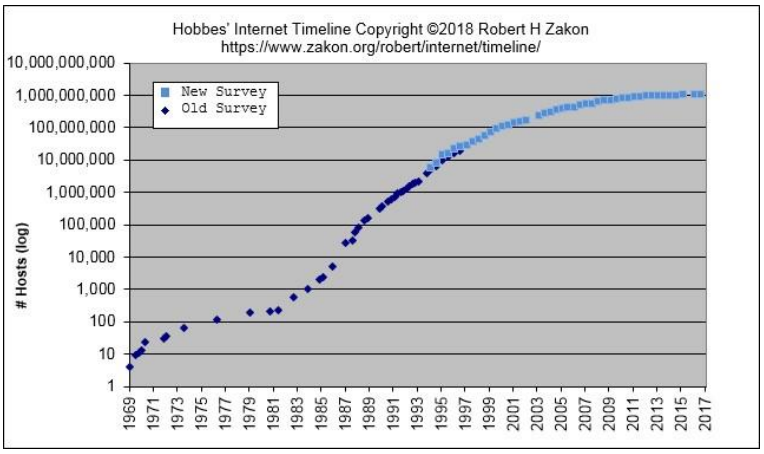
Facultad de Informática | UNLP

1 Introduccion

El modelo cliente-servidor se refiere a un modelo de comunicación que vincula a varios dispositivos informaticos a través de una red. El cliente realiza peticiones de servicios al servidor, que se encarga de satisfacer dichos requerimientos. Con esta arquitectura, las tareas se distribuyen entre los servidores (que proveen los servicios) y los clientes (que demandan dichos servicios), el cliente le pide un recurso al servidor, quien devuelve una respuesta. La aplicación cliente-servidor siempre es instalada en la computadora del cliente, a diferencia de una aplicación web. Las aplicaciones web pueden funcionar en los navegadores directamente y no necesitan instalacion. Tienen como ventaja centralizar el control y acceso a recursos, son escalables porque se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores).

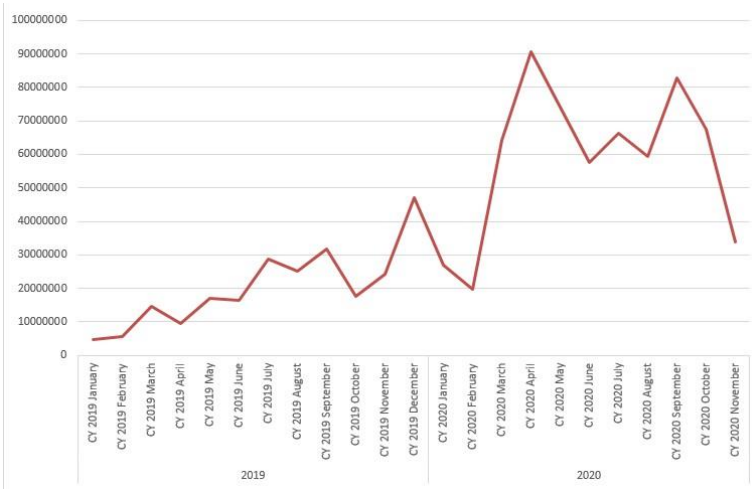
La aplicacion web necesita un servidor web para manejar las solicitudes realizadas por el cliente y guarda la información requerida para realizar la tarea de manera eficiente. El usuario solicita datos o información en particular al servidor web a través de Internet, el servidor web luego reenvía la solicitud al servidor.

El trafico web es el responsable de un buen porcentaje del trafico de Internet. Hoy día el trafico HTTP predomina respecto del resto de los protocolos, y hay una gran cantidad de usuarios que pueden generar una cantidad inmensa de peticiones. En el siguiente gráfico [18] se puede ver el crecimiento logarítmico del trafico web a lo largo del tiempo.



Dado el contexto en el que funciona un sistema web, éstas tienen una alta probabilidad de enfrentarse a amenazas desencadenadas por diversos factores como por ejemplo fallos del sistema debidos a una codificación incorrecta, servidores mal configurados y problemas de diseño de la aplicación. Las vulnerabilidades en el codigo de una aplicación pueden ser aprovechadas por los ciberdelincuentes para acceder a bases de datos, servidores y otros datos sensibles.

La pandemia fue una situación que dejó al descubierto que la gran cantidad de problemas de seguridad que sufrieron las compañías se debía a descuidos menores al momento de desarrollar o desplegar la aplicación y según los datos oficiales de los reportes del equipo de Respuesta ante Emergencias Informáticas nacional [5] se duplicaron los ataques informaticos respecto al año 2020.
En el siguiente grafico [1] se puede ver el crecimiento después del año mencionado.



Para contrarrestar el problema queremos seguir buenas practicas que deriven a un desarrollo seguro.

El objetivo es elaborar una guía de buenas practicas en soluciones SPA utilizando los frameworks mas populares como Angular, React o Vue para el cliente y Nodejs o Nestjs para el servidor.

2 Trabajos relacionados

- Buenas practicas aplicadas a la implementación colaborativo de aplicativos web [4].
- Entorno para experimentación de vulnerabilidades en la enseñanza de buenas practicas de programacion [9].
- Las mejores practicas en codificación de software [22].

3 Metodología de investigación

En base a reportes de los problemas más comunes en los sitemas web se van a plantear escenarios para reproducirlos y soluciones para prevenirlos. Las soluciones van a ser implementadas en aplicaciones SPA utilizando los frameworks mas populares como React, Angular o Vue en el lado del frontend y Nestjs en el lado del backend.

4 Lista de problemas identificados

Common Weakness Enumeration (CWE™) es una organización que publica una lista de las 25 principales vulnerabilidades [6] de software más peligrosas. Esta lista demuestra las debilidades de software más comunes y con mayor impacto de la actualidad, que son faciles de encontrar y explotar y pueden conducir a vulnerabilidades que permitan tomar el control completo de un sistema, robar datos o impedir que las aplicaciones funcionen.
Funciona junto con la base de datos nacional de vulnerabilidades de Estados Unidos para la recopilacion de datos de gestión de vulnerabilidades basados en estandares operados por el gobierno federal. OWASP [21], es una organización sin fines de lucro y de codigo abierto Open Web Application Security Project®, que también trabaja para reforzar la seguridad de la web.

Lor problemas mas comunes reportados son:

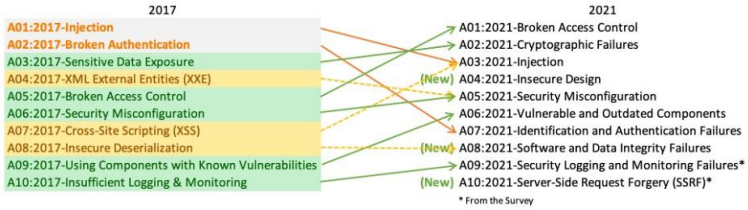
- Fallas en el control de acceso. BROKEN ACCESS CONTROL.

- Validacion de entrada incorrecta.
- Neutralizacion incorrecta de la entrada durante la generación de la pagina web (”Cross-site Scripting”)
- Falsificacion de solicitud entre sitios (CSRF)
- Componentes vulnerables y obsoletos.
- Neutralizacion incorrecta de elementos especiales utilizados en un comando SQL (’inyeccion SQL’)

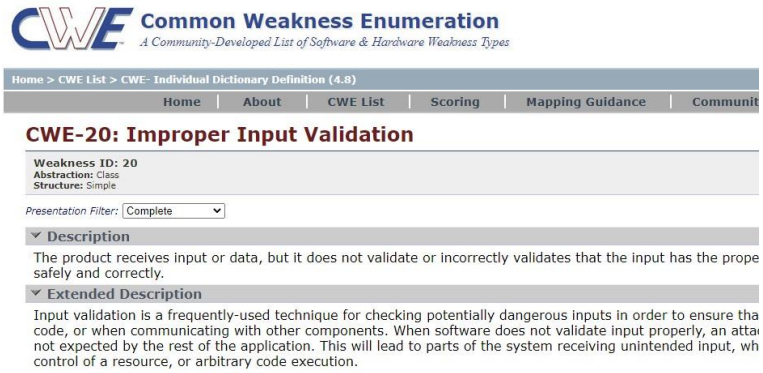
Validacion de los datos de entrada

El problema de no controlar los datos de entrada es la desprotección del software frente a posibles ataques informáticos. Los ataques más graves y comunes son el Broken Access Control, SQL Injection (SQLi), Cross-Site Scripting (XSS), entre otros como se indica en el gráfico publicado en el Top 10 de OWASP [21] y en el ranking de CWE [7].

Como puede verse en el siguiente gráfico, las principales fallas que se enumeran, ocurren por errores en las validaciones.



Un atacante podría no estar usando la aplicación para consumir los servcicios de manera directa. El codigo que se ejecuta del lado del cliente, se ejecuta en una zona de desconfianza ya que el usuario puede modificar el codigo javascript con las herramientas de desarrollador.



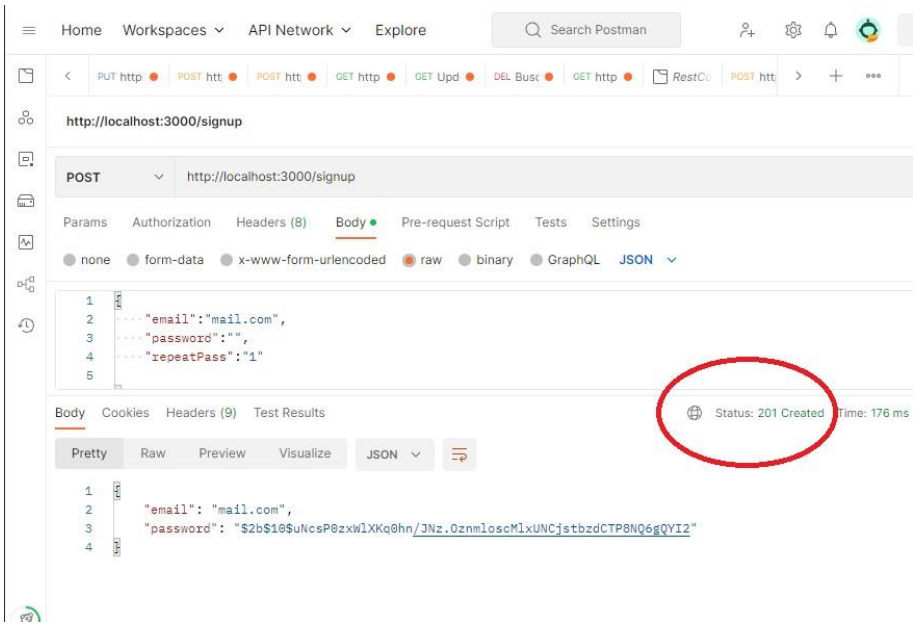
Ejemplo

Tenemos una aplicacion desarrollada en Angular del lado del cliente [15], la cual debe enviar los datos de un nuevo usuario a un servidor [14] que corre en ”http://localhost:3000” que se encarga de registrarlo en la base de datos.
El usuario debe ingresar un email, el cual debe ser valido, una contraseña mayor de 6 caracteres y una confirmación de contraseña. Todos los campos son obligatorios.

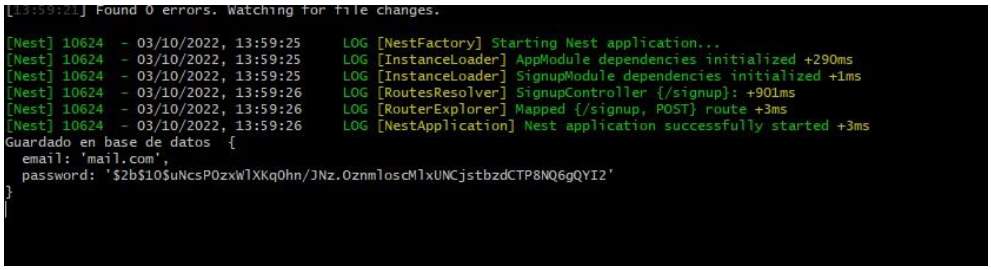
Si el usuario no ingresa los datos correctamente, el botón registrarse no se activa e informa al usuario que los datos son incorrectos.

La aplicacion valida los datos de manera correcta. No hay manera que el usuario se registre con datos que no estarían permitidos.

Pero, ¿Qué pasa si en el servidor no validamos y el usuario en vez de registrarse a través del sitio "http://localhost:4200", intenta usar una aplicación diferente, como por ejemplo Postman [8]?



Como puede verse, el servidor retornó un código 201, y se guardaron los datos en la base de datos, cosa que no debió suceder.



Solucion

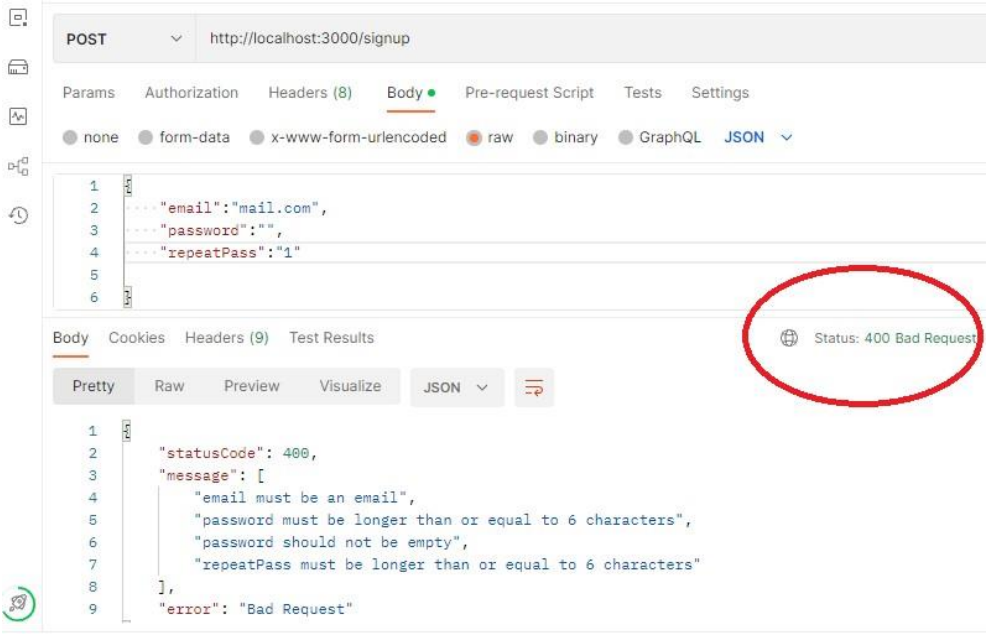
No alcanza con validar solamente en la aplicación cliente, las mismas validaciones deben hacerse en la aplicación del servidor. Usando los decoradores de class-validator en Nestjs [19], se hacen las mismas validaciones que en la aplicacion cliente.

create-signup.dto.ts

src > signup > dto > create-signup.dto.ts > CreateSignupDto

```ts
import { IsNotEmpty, IsString, MinLength, IsEmail } from "class-validator";

export class CreateSignupDto {
    @IsNotEmpty()
    @IsEmail()
    email: string;
    @IsNotEmpty()
    @IsString()
    @MinLength(6)
    password: string;
    @IsNotEmpty()
    @IsString()
    @MinLength(6)
    repeatPass: string;
}
```

De ésta manera, al intentar registrarse con otra aplicación cliente, lo que retorna el servidor es un error 400.



El control de validacion de datos de entrada como la longitud de una contraseña debe hacerse tanto en el cliente como en el servidor, pero principalmente

en el servidor.

Cross Site Scripting (XSS)

Cuando se desarrolla una SPA es muy probable que se esté utilizando un framework. Los frameworks más populares como Vue, Angular, o la librería de javascript React, tienen protecciones integradas para sanitizar las entradas que podrían ser maliciosas como inyección SQL.

Si bien los frameworks mencionados tienen protección integrada, tanto Vue [25] como Angular [2] tienen las mismas recomendaciones en común para minimizar peligros potenciales.

- Usar motores de plantillas de confianza.
- Usar componentes del framework que estén actualizados.
- No acceder al DOM directamente.

Entonces podemos decir que React, Angular y Vue hacen un buen trabajo al ayudar a los desarrolladores a escribir código seguro. Cualquiera que cree aplicaciones con éstos frameworks de la "manera normal" puede estar seguro de que no causar vulnerabilidades XSS.

Falsificacion de solicitud entre sitios

La vulnerabilidad CSRF ocurre cuando aplicaciones web permiten a un atacante inducir a los usuarios a realizar acciones que no pretenden realizar, como por ejemplo, cambiar su dirección de correo electrónico, su contraseña o realizar una transferencia de fondos.
Angular y Vue recomiendan comunicarse con el equipo de backend para coordinar el manejo de tokens con envíos de formularios.

El manejo de las sesiones de los usuarios en la aplicación es importante para poder controlar ésta debilidad que puede causar acciones no deseadas para los usuarios de la aplicacion. En un ataque CSRF, el atacante utiliza métodos como el phishing que haran que el usuario realice solicitudes al servidor. El servidor no puede distinguir la solicitud si usa cookies de sesión que se envían automaticamente con la solicitud.

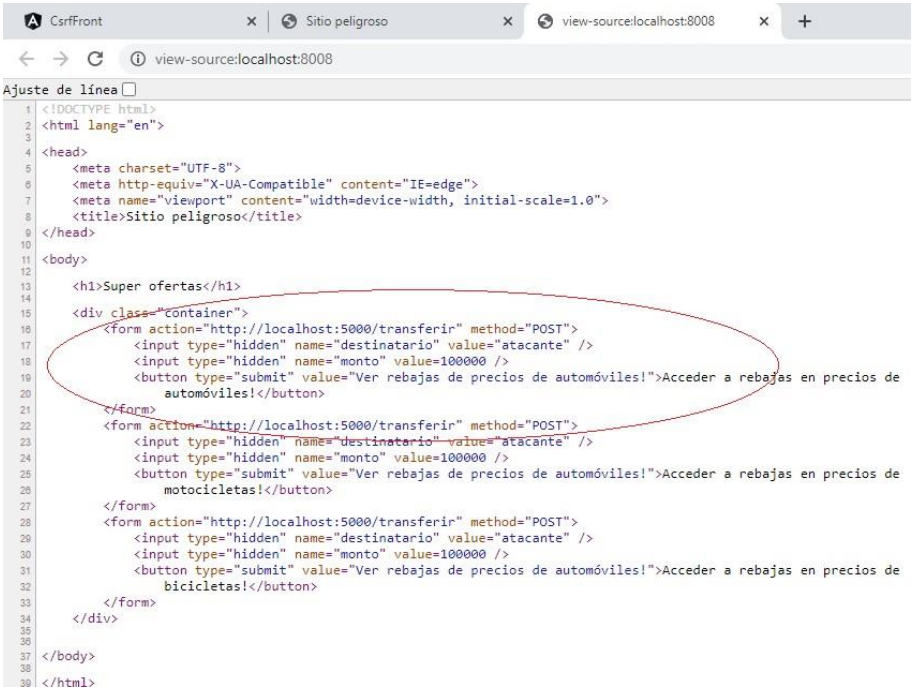
Ejemplo

En el ejemplo tenemos una aplicacion que simula ser la cuenta de un usuario de homebanking [13] en la dirección http://localhost:4200, la cual se comunica con su backend [12] en la dirección http://localhodst:5000, a su vez suponemos que el usuario logueado en la aplicación de homebanking abrió un link de ofertas que le enviaron por correo, la pagina de ofertas [11] corre en la dirección http://localhost:8008.

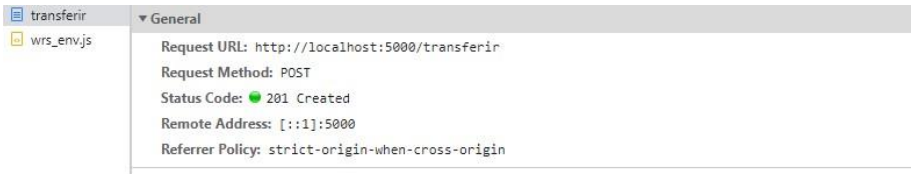
El usuario inició sesion con su cuenta.

Y abrió el link de ofertas que recibió en su correo.

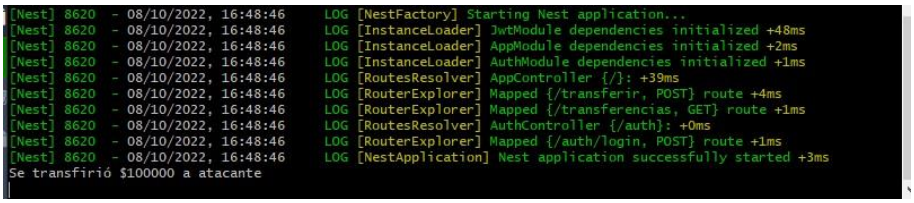
Si vemos el codigo fuente de la pagina de ofertas, podemos ver los formularios ocultos que van a intentar realizar una transferencia a la cuenta del atacante.



Cuando el usuario haga click en una oferta, al tener una sesión iniciada con una cookie valida de su homebanking, el sitio de ofertas hará una petición POST al servidor, y se transferir el monto indicado en el formulario oculto.



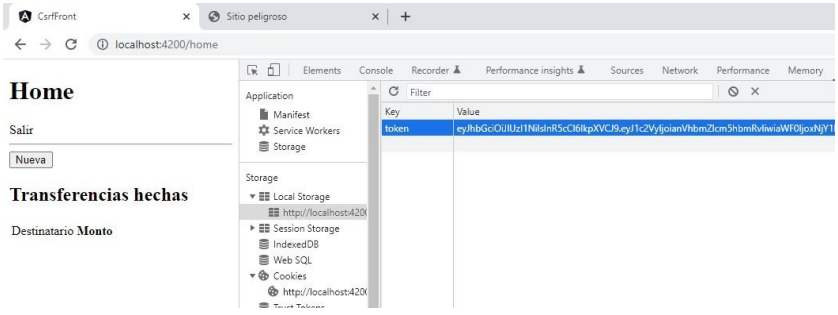
En el servidor se registra la transferencia.



Solucion

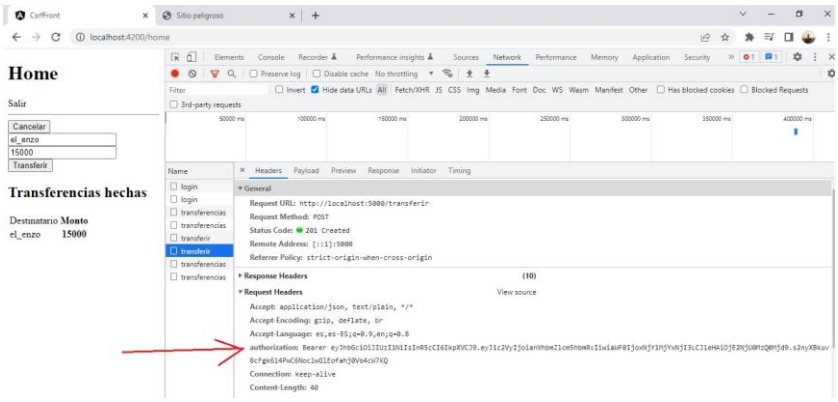
Cuando desarrollamos aplicaciones SPA, es conveniente otra manera de administrar las sesiones. Los ataques de Cross Site Request Forgery no son un problema si se est utilizando JWT [24] almacenado en el local storage.

Cuando el usuario inicia sesión, recibe un token firmado por el servidor y lo guarda en local storage del navegador, luego por cada solicitud al servidor, lo que se hace es enviar el token en el encabezado de la solicitud http, no se envía una cookie de sesion. El servidor antes de procesar cualquier solicitud verifica si el token es valido y decide si acepta o nó la solicitud.



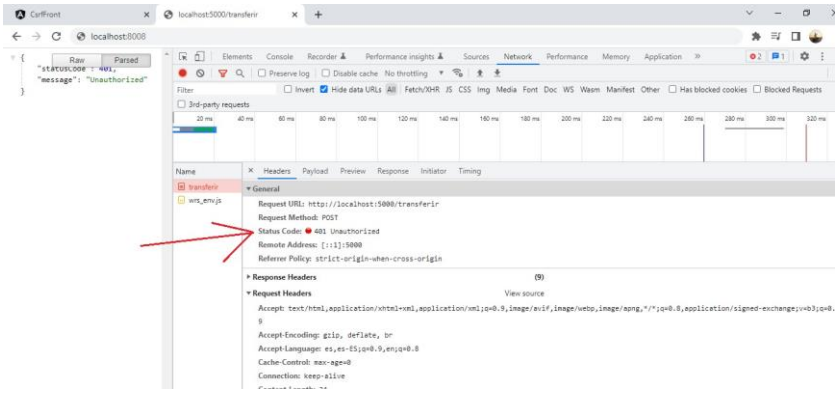
Cuando el usuario quiere hacer una transferencia, el sistema agrega el token

firmado al encabezado, en un atributo authorization: bearer token.

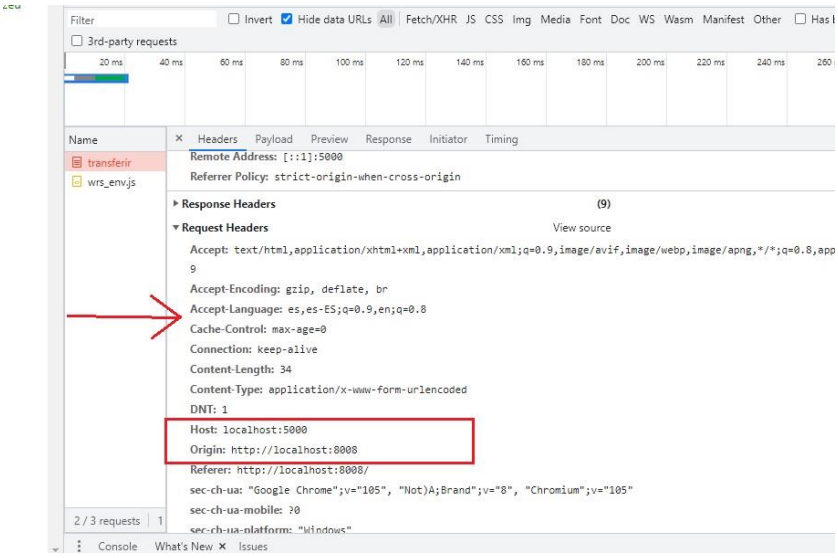



```
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [NestFactory] Starting Nest application...
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [InstanceLoader] JwtModule dependencies initialized +42ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [InstanceLoader] AppModule dependencies initialized +1ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [InstanceLoader] AuthModule dependencies initialized +1ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [RoutesResolver] AppController {/}: +37ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [RouterExplorer] Mapped {/transferir, POST} route +4ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [RouterExplorer] Mapped {/transferencias, GET} route +1ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [RoutesResolver] AuthController {/auth}: +0ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [RouterExplorer] Mapped {/auth/login, POST} route +1ms
[Nest] 9084  - 08/10/2022, 17:33:31     LOG [NestApplication] Nest application successfully started +3ms
Se logueo el usuario juanfernando y se le asigno el token eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoianVhbm
Zlcm5hbmRvIiwiaWFOIjoxNjY1MjYxNjI3LCJleHAiOjE2NjU0MzQ0Mjd9.s2nyXBkuv8cFgk614PwC6NoclwGlEofahj0Vo4cW7KQ
Se transfirió $15000 a el_enzo
```

De ésta manera, cuando el usuario presiona uno de los links que intenta hacer una accion de manera oculta, habrá un error de autorización ya que el sitio atacante no est autorizado para realizar la operación.



Esto ocurre porque el sitio maligno no envía en su header el token de autorizacion:



¿Que pasa si el usuario logueado deja su computadora por un momento y otra persona decide copiarse el token y enviar solicitudes desde otro dispositivo?

Un token, una vez firmado, es valido en todo momento, por lo que es necesario una fecha de expiración. En el caso de los ”Access tokens”, si alguien captura uno, podrá tener acceso a las operaciones permitidas para siempre.

Para facilitar la gestion por parte de los consumidores de los tokens es necesario identificar ip emisor y todos los posibles destinatarios, de esta manera podran identificar la clave de validación y comprobar que está dirigido a ellos. Una buena practica es que los consumidores del token validen estos datos.

Por lo tanto podemos concluir que los JWT se pueden utilizar como un mecanismo de autenticacion que no requiere una base de datos. El servidor puede evitar el uso de una base de datos porque el almacén de datos en el JWT enviado al cliente es seguro.

Para agregar un nivel extra de seguridad, podría ser ofrecer tener la ip como segurad, en una opcion en la pantalla de inicio de sesión. Puede usarse el paquete de Node [10]. Si un usuario elige utilizar la validación de la dirección IP, opta por una mayor seguridad pero acepta el hecho de que, en ocasiones, es posible que tenga que volver a iniciar sesión. O puede elegir una seguridad más baja con su sesion mas estable.

Broken Access Control

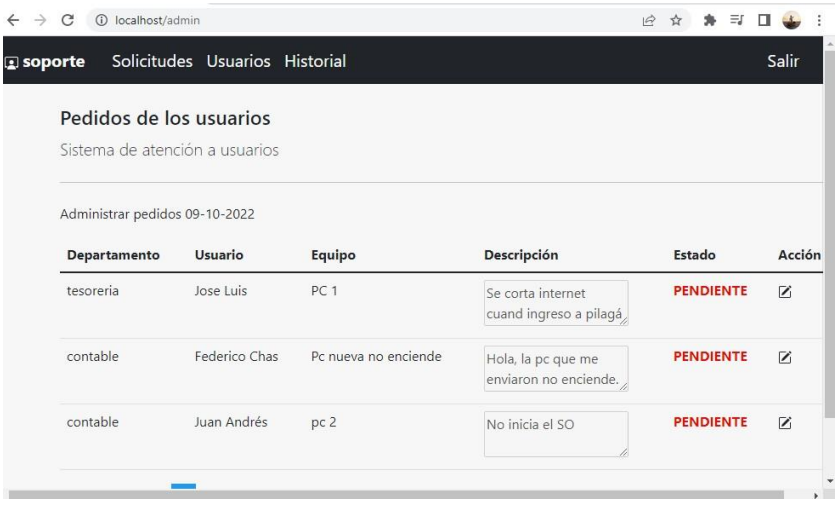
Es la vulnerabilidad mas destacada por OWASP.
Ocurre cuando un usuario se autentica usando usuario y contraseña, luego se aplican permisos para dar acceso a los recursos autorizados. Por ejemplo, el control de acceso limita las paginas que podemos ver.

Esta vulnerabilidad ocurre cuando un usuario puede acceder a un area a la cual no debería poder. Es decir, vulnerabilidades en el control de acceso permiten crear fallos que podrían ser utilizados por un atacante para ver, modificar o borrar contenido al cual no debería poder tener acceso.

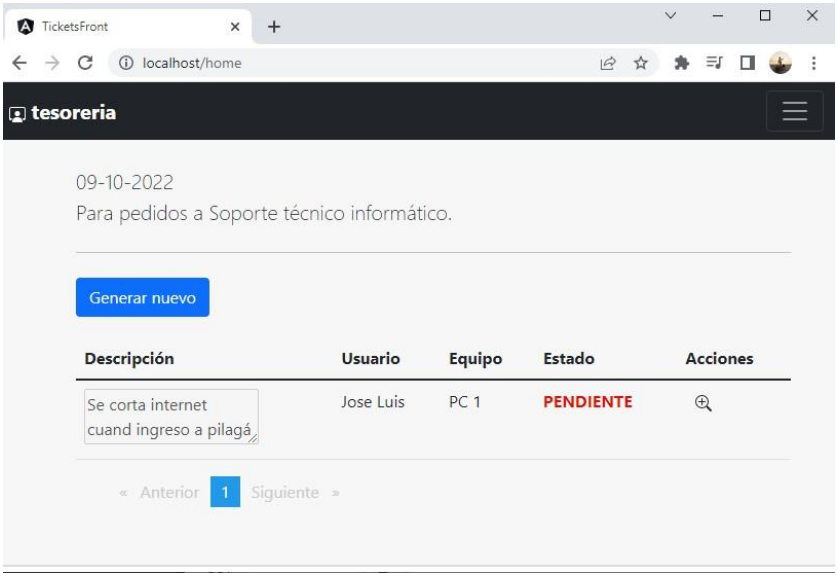
Ejemplo

Tenemos una aplicacion de tickets usada para notificar al departamento de soporte diferentes problemas. Existen dos clases de usuarios, administrador y usuario común. El administrador es quien ve las solicitudes de los demas usuarios. Cada usuario envía solicitudes al usuario administrador y puede ver su historial de solicitudes y no debería poder ver información relacionada a otros usuarios. La aplicacion se compone de un frontend [17] y de un backend [16].

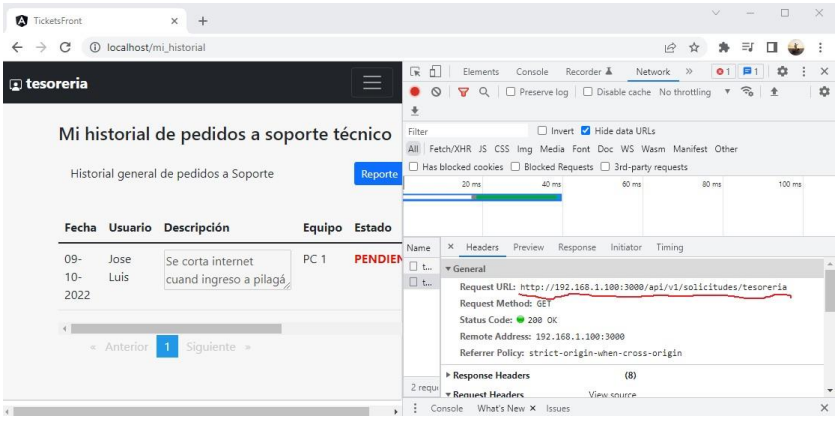
Cuando el usuario soporte que es administrador ingresa, puede ver las solicitudes.



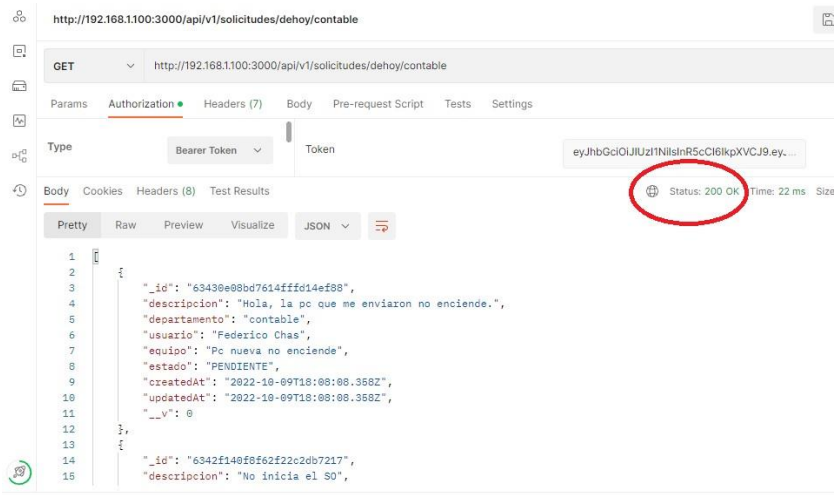
En el caso del usuario tesorería solo puede ver las solicitudes que él mismo hizo.



Pero ¿Qué pasa si el usuario tesorería accede a recursos del sistema que no estan protegidos?



Con el formato de la solicitud, el usuario de tesorería sabe que hay otros departamentos en la dependencia y podría usar la herramienta Postman [8] para acceder...



En éste caso, el usuario tesorería correctamente logueado no debería poder acceder a esos recursos y está pudiendo obtener el listado de solicitudes de los otros usuarios. En éste caso hay una vulnerabilidad en el servidor.

Solucion

La solucion es proteger el endpoint del servidor con una guarda que verifique que el usuario logueado tenga permiso de acceder a la ruta.

solicitudes.controller.ts routes.guard.ts

src > solicitudes > routes.guard.ts > RoutesGuard > canActivate

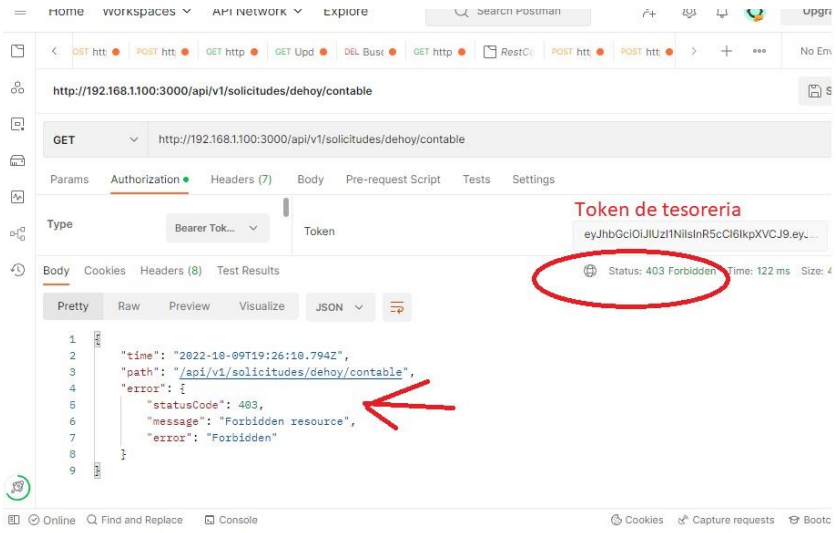
```ts
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class RoutesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {


    let { url } = context.switchToHttp().getRequest();
    let { user } = context.switchToHttp().getRequest();
    url = url.split('/');
    let usrurl = url[url.length-1]
    let usrlog = user.user
    if (usrurl == usrlog){
      return true;
    }else{
      return false;
    }
  }
}

```

También es necesario proteger el path en el lado del cliente. En angular se pueden usar varias guardas para un path, por lo que en éste caso, además de verificar que los usuarios tengan un token valido, agregamos una guarda que verifique si tiene los roles para acceder.

```ts
  },
  { path: 'historial',
    component: HistorialComponent,
    canActivate : [LoginGuard,RolesGuard]
  },
```

```ts
export class RolesGuard implements CanActivate {
  user!: User;
  constructor(private loginService: LoginService, private router: Router) {
    this.loginService.user.subscribe(x => this.user = x);
  }
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    let { user } = this.user;
    let rolesRequired = ['admin'];
    const usr = JSON.parse(JSON.stringify(user)); //convierto string a objeto

    usr.roles.map((rol:Role)=> console.log(rol));

    return rolesRequired.some((role)=> usr.roles?.includes('admin'));


  }
}
```

La guarda RolesGuard, chequea de igual manera que lo hace el servidor. Después de hacer los cambios mencionados volvamos a ingresar con el usuario tesorería e intentar acceder a contable.



Ahora el servidor reconoce que tesoreria no está autorizado para ver las solicitudes de contable.
Como conclusion decimos que los endpoints tengan toda restricción posible, y a medida que se necesite acceso liberarlos para el grupo de usuario concreto.

Exposición pública de contraseñas

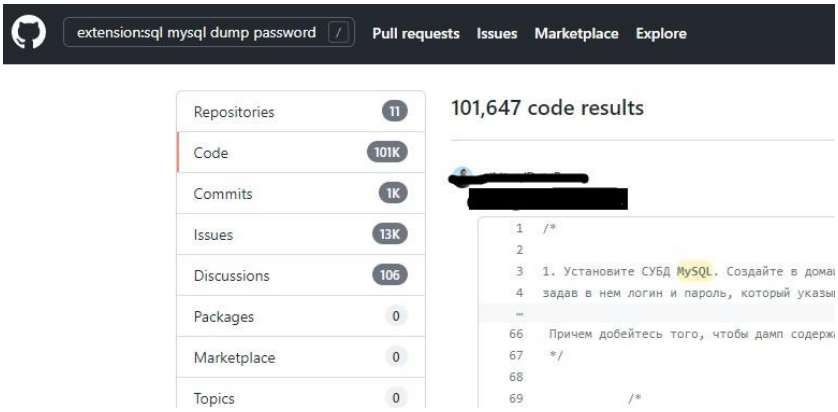
El uso inadecuado de repositorios puede exponer datos sencibles que usa nuestra aplicacion, como por ejemplo contraseñas de acceso a bases de datos, o contraseñas semilla de un jwt. Si el repositorio expone contraseñas, estamos expuestos a perder gran cantidad de datos [23] que pueden ser de suma importancia para la organizacion.

Ejemplo

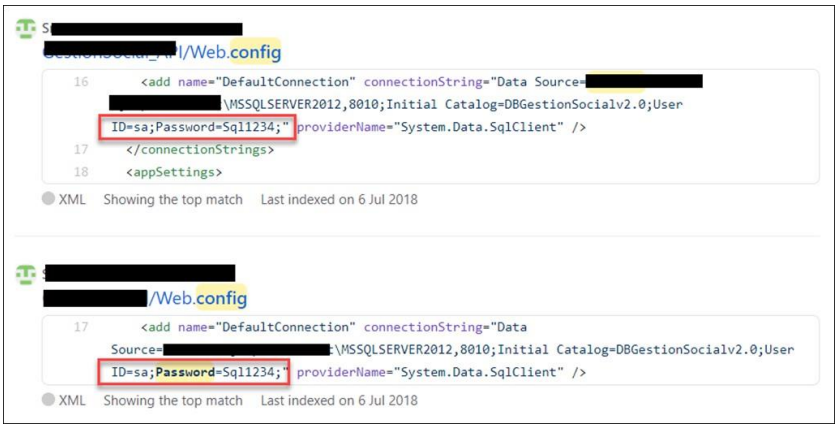
Una actividad de reconocimiento de una organización a la cual se está evaluando puede ser buscar identificar información expuesta en repositorios públicos que puedan ser de utilidad para realizar ataques de mayor complejidad o lograr accesos no autorizados.

Usando la búsqueda avanzada de github como por ejemplo:

- extension:sql mysql dump password
- extension:sql mysql dump
- filename:wp-config.php



Es posible acceder a datos harcodeados.



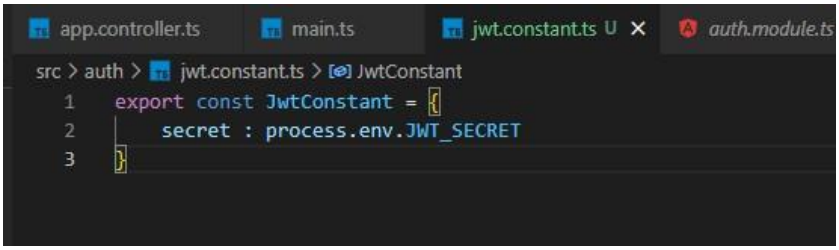
Solucion

Para el caso en el que estamos trabajando con un token JWT [24], es necesario una clave semilla conocida sólo por el servidor y que luego usará el mismo para validar los tokens que reciba en cada solicitud. Si la clave semilla es conocida, entonces cualquiera podría generar su token y firmarlo. Si estamos desarrollando un login con jwt. NO debería hacerse como se muestra en la siguiente imagen.

Lo que debe hacerse es definir en un archivo .env en la raiz del proyecto, las variables de entorno que querramos. En éste caso la clave semilla de nuestro JWT. El archivo .env es ignorado por git al especificarlo en el .gitignore del

proyecto, por lo que nuestra contraseña semilla no queda expuesta. Si otros desarrolladores estan trabajando en el proyecto, se puede defenir un archivo .env-template donde indicamos las variables que necesitamos, pero sin un valor concreto.

Luego declaramos una constante.



Y en el modulo llamamos a la constante, quien toma el valor desde el entorno del sistema.

Auditoría del sistema

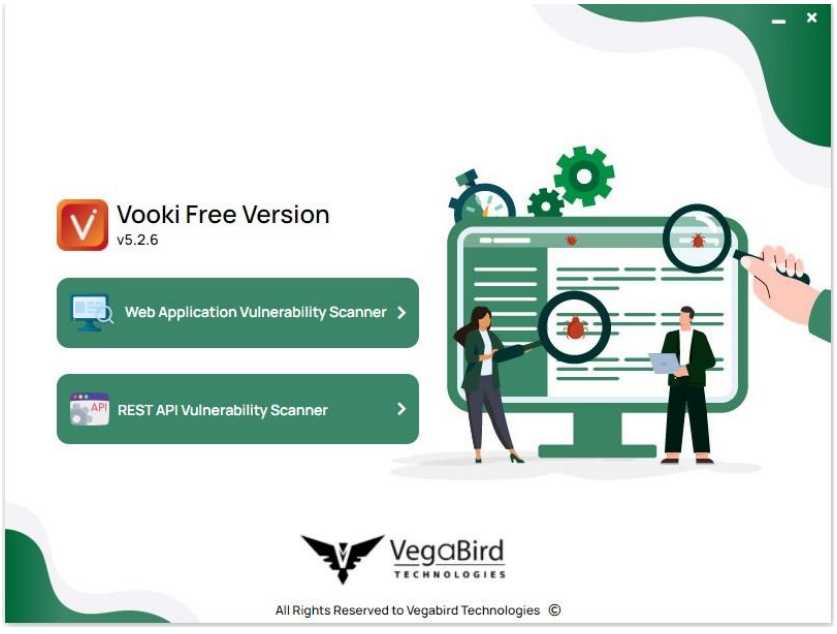
Auditar el sistema, tiene por objeto descubrir las vulnerabilidades de las aplicaciones o servicios web, detectando fallos de diseño e implementación, falta de validacion de entrada datos, etc. Para ésto, presentamos algunas herramientas útiles que pueden ayudar en la auditoría de las aplicaciones..

Ejemplos

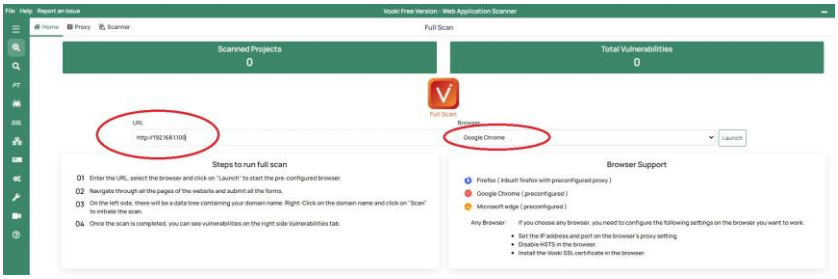
Algunas herramientas que nos permiten trabajar en las auditorias de nuestros sistemas.

- Vooki - Escaner de vulnerabilidades de API y aplicaciones web [27].
- NPM Audit [20].
- Pruebas integrales en las API REST, SOAP y GraphQL, JMS, JDBC y otros servicios web [26].
- API Security Articles [3].

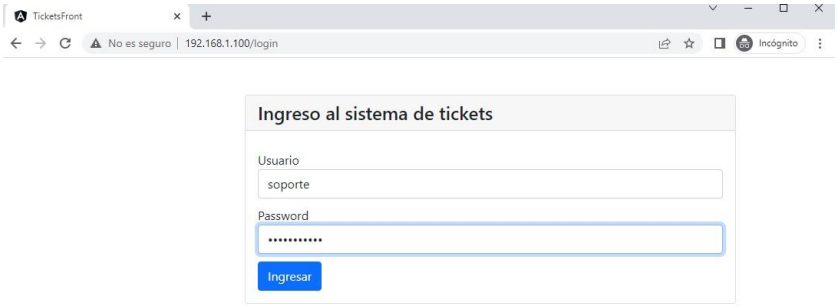
En éste ejemplo, usamos Vooki para analizar la aplicación de tickets mostrada en el ejemplo anterior.



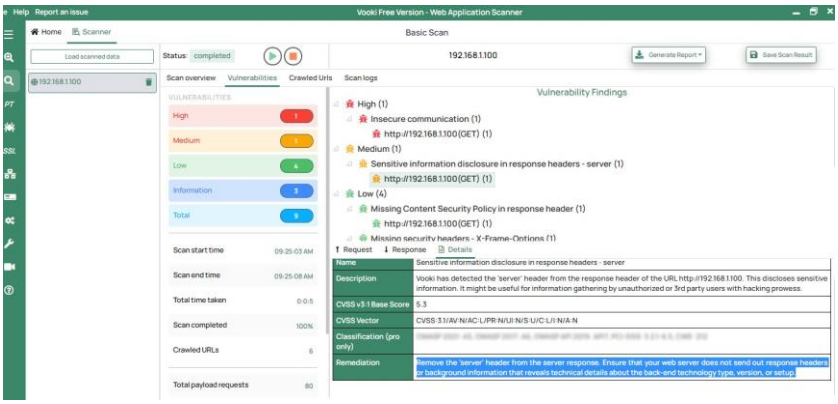
Iniciamos la aplicacion e indicamos el dominio de la aplicación, en éste caso es la ip 192.168.1.100 sobre http.



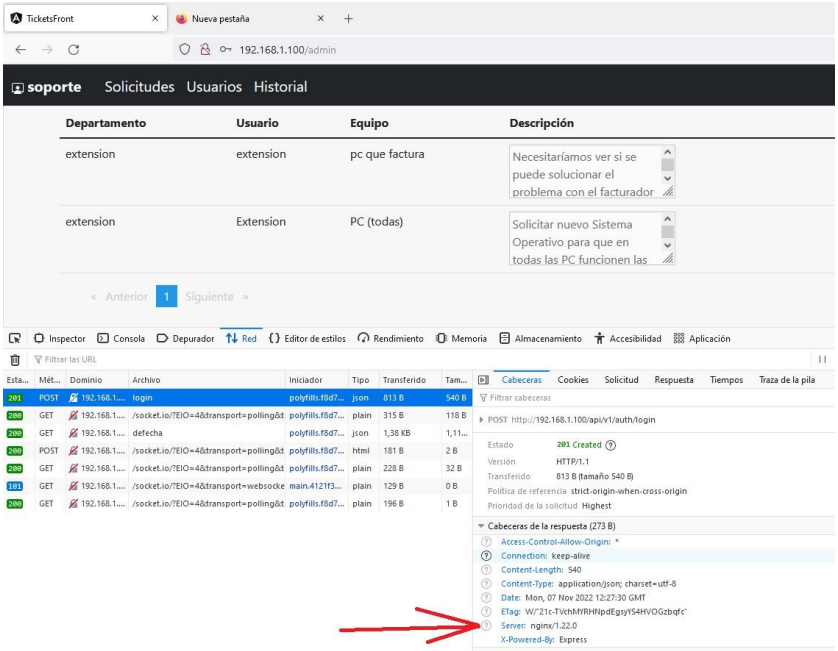
Iniciamos sesion con las credenciales de administrador.



Al realizar un escaneo, la aplicación muestra un informe con los resultados.



El problema mas grave que se encontro, es la comunicación insegura, ésto sucede porque el sistema funciona en una red local donde no se está utilizando ningún certificado ssl. El siguiente problema encontrado es de nivel medio, y el sistema nos recomienda una solución, y para éste caso concreto es remover el encabezado 'servidor' donde se revela tipo y versión del servidor web, dato útil para la recopilacion de información por parte de usuarios no autorizado o con destreza de piratería.



5 Conclusiones

En resumen, podemos enumerar los siguientes ítems para concluir en un desarrollo seguro.

- Validar los datos de entrada en la aplicacion cliente y principalmente en el servidor.
- Usar componentes del framework actualizados.
- No usar componentes de terceros.
- Usar jwt en localstorage.
- No exponer datos sensibles en los repositorios públicos.
- Hacer auditorías de nuestras aplicaciones.

References

[1] América Latina en 2020: ataques cibernéticos, sus consecuencias y lo que se avecina — securelist.lat. https://securelist.lat/america-latina-en-2020-ataques-ciberneticos-sus-consecuencias-y-lo-que-se-avecina/91919/. [Accessed 21-Sep-2022].

[2] Angular — angular.io. https://angular.io/guide/security. [Accessed 21-Sep-2022].

[3] API Security Articles, News, Vulnerabilities Best Practices — apisecurity.io. https://apisecurity.io/. [Accessed 26-Oct-2022].

[4] BUENAS PRACTICAS APLICADAS A LA IMPLEMENTACION COLABORATIVO DE APLICATIVOS WEB — Mundo FESC — fesc.edu.co. https://www.fesc.edu.co/Revistas/OJS/index.php/mundofesc/article/view/67. [Accessed 29-Sep-2022].

[5] CERT.ar — argentina.gob.ar. https://www.argentina.gob.ar/jefatura/innovacion-publica/ssetic/direccion-nacional-ciberseguridad/cert-ar. [Accessed 21-Sep-2022].

[6] CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses — cwe.mitre.org. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. [Accessed 28-Sep-2022].

[7] CWE - Common Weakness Enumeration — cwe.mitre.org. https://cwe.mitre.org/. [Accessed 28-Sep-2022].

[8] Download Postman — Get Started for Free — postman.com. https://www.postman.com/downloads/. [Accessed 02-Oct-2022].

[9] Entorno para experimentacion de vulnerabilidades en la enseñanza de buenas practicas de programacion — sedici.unlp.edu.ar. http://sedici.unlp.edu.ar/handle/10915/50626. [Accessed 29-Sep-2022].

[10] express-jwt-ip — npmjs.com. https://www.npmjs.com/package/express-jwt-ip?activeTab=readme. [Accessed 31-Oct-2022].

[11] GitHub - serdel1979/csrf-app-maligna — github.com. https://github.com/serdel1979/csrf-app-maligna. [Accessed 08-Oct-2022].

[12] GitHub - serdel1979/pps-csrf-back — github.com. https://github.com/serdel1979/pps-csrf-back. [Accessed 08-Oct-2022].

[13] GitHub - serdel1979/pps-csrf-front — github.com. https://github.com/serdel1979/pps-csrf-front. [Accessed 08-Oct-2022].

[14] GitHub - serdel1979/pps-formulario-back — github.com. https://github.com/serdel1979/pps-formulario-back. [Accessed 02-Oct-2022].

[15] GitHub - serdel1979/pps-formulario-front — github.com. https://github.com/serdel1979/pps-formulario-front. [Accessed 02-Oct-2022].

[16] GitHub - serdel1979/tickets-soporte-fba — github.com. https://github.com/serdel1979/tickets-soporte-fba. [Accessed 09-Oct-2022].

[17] GitHub - serdel1979/tickets-soporte-fba-front — github.com. https://github.com/serdel1979/tickets-soporte-fba-front. [Accessed 09- Oct-2022].

[18] Hobbeapos; Internet Timeline - the definitive ARPAnet amp; Internet history — zakon.org. https://www.zakon.org/robert/internet/timeline/. [Accessed 29-Sep-2022].

[19] NestJS - A progressive Node.js framework — nestjs.com. https://nestjs.com/. [Accessed 02-Oct-2022].

[20] npm-audit — npm Docs — docs.npmjs.com. https://docs.npmjs.com/cli/v8/commands/npm-audit. [Accessed 07-Nov-2022].

[21] OWASP Top Ten — OWASP Foundation — owasp.org. https://owasp.org/www-project-top-ten/. [Accessed 21-Sep-2022].

[22] Publicacion de Documentacion Digital - UMSS: LAS MEJORES PRACTICAS EN CODIFICACION DE SOFTWARE — ddigital.umss.edu.bo. http://ddigital.umss.edu.bo:8080/jspui/handle/123456789/14367. [Accessed 29-Sep-2022].

[23] Publicaron más de 1.5 millones de contraseñas asociadas a correos de organismos gubernamentales — WeLiveSecurity — welivesecurity.com. https://www.welivesecurity.com/la-es/2021/04/29/publicaron-mas-de-1-5-millones-de-contrasenas-asociadas-a-correos-de-organismos-gubernamentales/. [Accessed 19-Oct-2022].

[24] RFC 7519: JSON Web Token (JWT) — rfc-editor.org. https://www.rfc-editor.org/rfc/rfc7519. [Accessed 21-Sep-2022].

[25] Security — Vue.js — vuejs.org. https://vuejs.org/guide/best-practices/security.html. [Accessed 21-Sep-2022].

[26] The Worlds Most Popular API Testing Tool — SoapUI — soapui.org. https://www.soapui.org/. [Accessed 26-Oct-2022].

[27] VOOKI - Web Application and API Vulnerability Scanner — Vegabird — vegabird.com. https://www.vegabird.com/vooki/. [Accessed 26-Oct-2022].