Securing a REST API with JWT Authentication in C# Using AES-Encrypted Keys

JWT (JSON Web Token) authentication is a popular method for securing REST API access, but by integrating AES encryption, you can add an additional layer of secure key security to protect sensitive information within the token itself.

AES is a symmetric-key encryption algorithm that uses the same key for both encryption and decryption, enhancing security by giving you complete control over the key. Only authorized parties with access to this key can encrypt or decrypt the data, making it virtually impossible for anyone without the key to decrypt the data or token. This secure key, which you manage, ensures that only trusted sources can interact with the encrypted token (you control the encryption key).

In this article, I'll share a C# approach for generating a secure JWT using AES encryption for the secret key, based on techniques that I have developed and implemented in my own REST API projects.

1. Encrypt Sensitive Information with AES

The core of this JWT generation process begins with AES encryption of a sensitive API key. Encrypting the API key ensures that even if a token is exposed, it won't reveal the underlying key data. This method is part of a my "GetToken" API endpoint. Here's a breakdown of the code.

Method: BuildJWTToken

```
private string BuildJWTToken(string api_key)
{
  // define AES encryption key and initialization vector (IV)
  byte[] myAesaKey = Convert.FromBase64String("BBFDAwQFBqwICQuLDA0ODi==");
  byte[] myAesaIV = Convert.FromBase64String("BBFDAwQFBqwICQuLDA0ODi==");

  // instantiate AES
  Aes myAes = Aes.Create();

  // encrypt api_key using AES
  byte[] encrypted = Cryptography.EncryptStringToBytes_Aes(api_key, myAesaKey,
myAesaIV);

  // read appsettings.json for config settings
  var config = new
ConfigurationBuilder().SetBasePath(Directory.GetCurrentDirectory()).AddJsonFile("app
settings.json");
  var configuration = config.Build();
  var secret = api_key;
```

```
  var SecretKey = new
Microsoft.IdentityModel.Tokens.SymmetricSecurityKey(encrypted); // generate secret
key


  // set the issuer and audience for the token
  var issuer = "IssuerName"; // can be any value you want
  var audience = "AudienceName"; // can be any value you want

  // read expiry time from appsettings.json
  // and set the token to the expiry value
  var expiry =
DateTime.UtcNow.AddMinutes(Convert.ToDouble(configuration["CPJwt:JWTTokenExpiry"]));

  var tokenHandler = new JwtSecurityTokenHandler();
  var tokenDescriptor = new SecurityTokenDescriptor
  {
    Subject = new ClaimsIdentity(new Claim[]
  {
      // define a unique identifier claim
      // the claim value can be anything you want
  new Claim(ClaimTypes.NameIdentifier, "16611C29FF595A1B4B99F3DEF224FD56"),
  }),
    Expires = expiry,
    Issuer = issuer,
    Audience = audience,
    SigningCredentials = new SigningCredentials(SecretKey,
SecurityAlgorithms.HmacSha256Signature)
  };

  var token = tokenHandler.CreateToken(tokenDescriptor); // set the token variable
  return tokenHandler.WriteToken(token); // return the token
}
```

2: Explanation of the components of the "BuildJWTToken" method

AES Encryption of the API Key: The method starts by converting the AES key and initialization vector (IV) from base64 strings. This approach uses AES to encrypt the api_key, adding an additional layer of security before using it as the JWT signing key.

Configuration for Expiry: Expiration is configurable via appsettings.json, giving flexibility in managing token validity. You can also store the exiry value in a SQL database. I chose appsettings.json for giving flexibility.

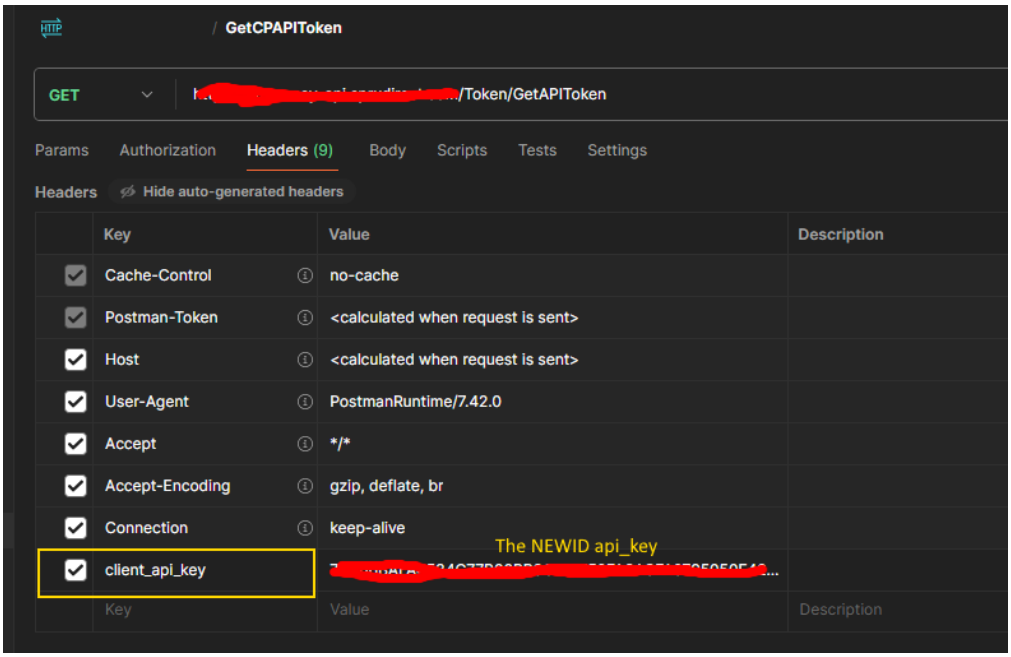
Setting up JWT Claims: The token descriptor defines a claim for the token. In this case, I use the NameIdentifier claim which uniquely identifies the user or API client.

Signing and Creating the Token: The encrypted key, wrapped in a SymmetricSecurityKey object, is used as the signing key for the token. JWT signing credentials use the HmacSha256Signature algorithm, ensuring token integrity and authenticity. The token is then generated and returned as a string. This value is then returned to the audience making the API call.

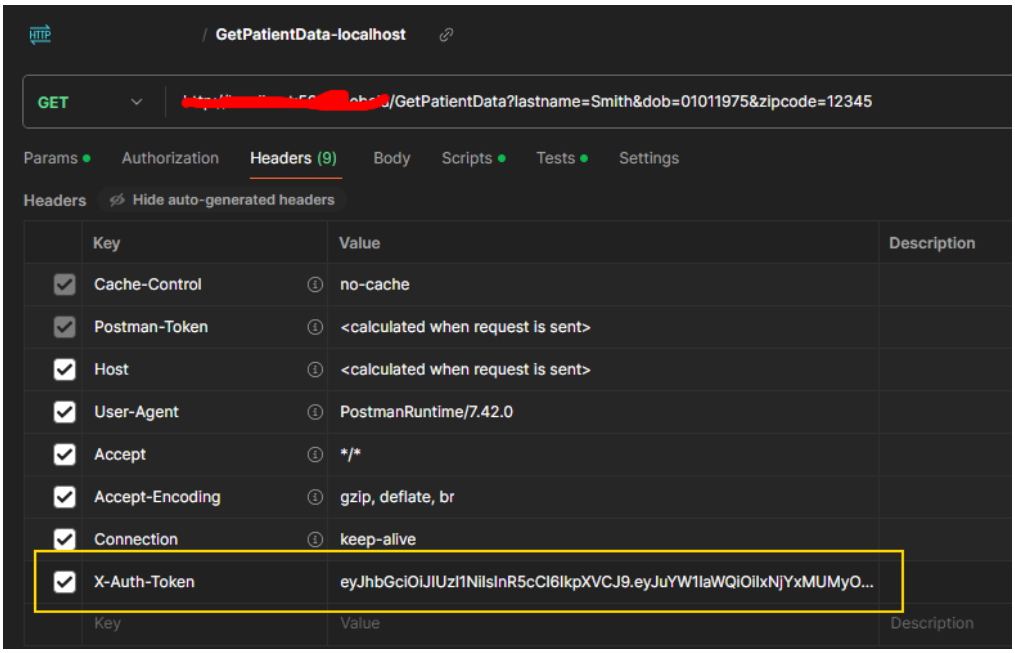
3. The "GetAPIToken" API Endpoint

To interact with the API, the audience first calls the GetAPIToken endpoint, using a unique api_key that I provide. The "api_key" is a unique seed key that I generate using the SQL NEWID() function which I store in an encrypted SQL table, alongside the audience ID, audience name, and api_key value. The audience includes this api_key in API requests as a header key labeled "client_api_key". The GetAPITokenendpoint then returns a secure JWT, which is required as header key labeled "X-Auth-Token" for accessing all API endpoints and ensures secure communication across API calls.

4. Sample of calling the "GetAPIToken" API Endpoint using Postman



4. Sample of calling the "GetPatientData" endpoint using the "X-Auth-Token" JWT value



5. Sample code for the "GetPatientData" endpoint

```
[HttpGet]
[Route("{controller}/{action}/{lastname ?}/{dob ?}/{zipcode ?}")]
public IActionResult GetPatientData(string lastname, string dob, string zipcode)
{
  string? client_api_key = Request.Headers["X-Auth-Token"];

  // check Request.Headers["X-Auth-Token"]
  if (GeneralFunctions.IsNullOrEmpty(client_api_key))
  {
    return new ContentResult() { Content = "Unauthorized", StatusCode = 401 };
  }

  if (!IsCPTokenValid(client_api_key!))// is the token valid?
  {
    return new ContentResult() { Content = "Unauthorized", StatusCode = 401 };
  }


  // this portion is just sample code and not the actual API code
  string ret_list = GetPatientDataByLastDOBZip(lastname, dob, zipcode); // get
search results from sql
  if (ret_list.Length == 0)
  {
    return new ContentResult() { Content = "no data found", StatusCode = 422 };
  }


  return new ContentResult() { Content = ret_list, StatusCode = 200 }; // return
patient data
}
```

The "IsCPTokenValid" method checks if the "X-Auth-Token" provided by the client matches the expected token for the endpoint. If the tokens match, it returns true; otherwise, it returns false. When false is returned, the endpoint responds with a 401 Unauthorized message.

Conclusion

Using AES encryption with JWT authentication is an effective way to secure REST APIs, adding a layer of protection to the signing key before embedding it in the token. This approach ensures sensitive data remains protected, even if a token is intercepted, and provides a robust authentication mechanism for secure API access.