|  |
| --- |
| **JWT Security Implementation** |
| **Document Version / Details: Ver 1.0/11-Mar-19** |
|  |

****

**Record of Release**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version No. | Modified By | Reviewed By | Authorized By | Release Date | Modifications Done |
| Ver.1.0 | *Omkar Ghaisas* | *Jeetendra Shenoy/Sheetal Shinde* |  | 11-Mar-19 | First Version |
| Ver 1.1 | Omkar Ghaisas | Jeetendra Shenoy / Sheetal Shinde |  | 15-Mar-19 | Update with actual JWT code |

**Creating RSA256 Public-Private Key pair**

1. Install OpenSSL for Windows and add the openssl.exe bin folder path to environment variables under the “PATH” section
2. Open command prompt and run the below commands in sequence

//Private Key

openssl genrsa -out private.key 2048

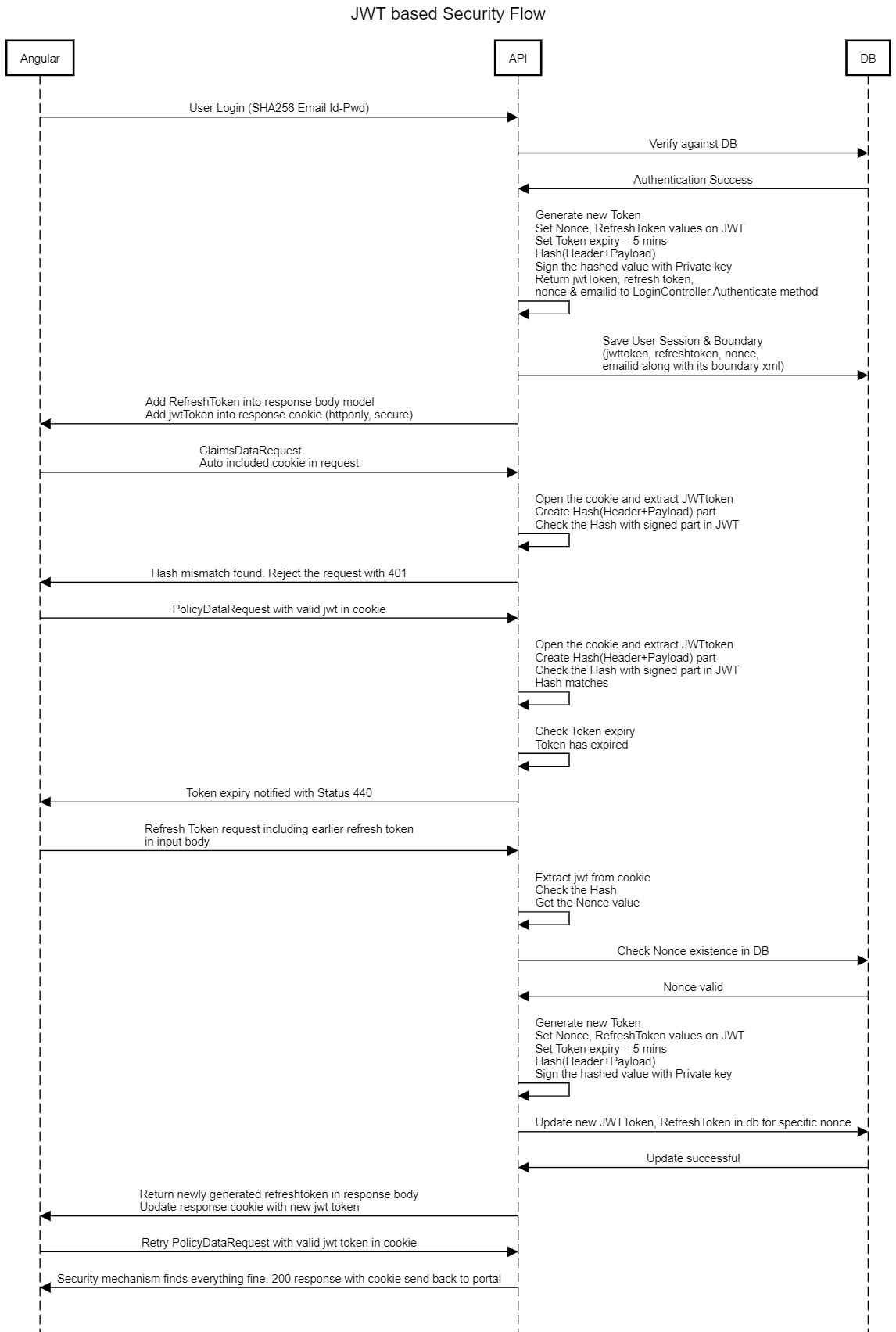
//Public Key

openssl rsa -in private.key -outform PEM -pubout -out public.pem

The above 2 commands will generate both the Public and Private keys in separate files on the default location where the command prompt is opened from.

Move the files to a secure location on the API server itself.

**Key rotation can be achieved by simply creating a new pem file (private-public key pair) and pushing the latest file to the server at the same location as earlier.**



Case 1. **Generating a Token at the time of Login**

Once the user has been authenticated using the database and new password SHA256 mechanism, a new token generation request will be initiated from LoginRepository.Authenticate method.

**The core mechanism of hashing, signing and token verification is done using the BouncyCastle library** (<http://www.bouncycastle.org/csharp/>).

The library is included in the project using NuGet package manager by searching for BouncyCastle.

**Token Generation process** will be as follows –

During the Login process, once user is authenticated, we need to generate a new token. We call the below class’ AuthTokenGenerator.GenerateToken method to get a new token.

public class AuthTokenGenerator : IAuthToken

{

public TokenInformation GenerateToken(string userName, string existingNonce = "")

{

RsJwt jwt = new RsJwt

{

KeySize = Convert.ToInt32(ConfigurationManager.AppSettings["KeySize"])

};

jwt.GenerateToken(userName, existingNonce, out string nonce, out string refreshToken, out string tokenString);

TokenInformation newToken = new TokenInformation

{

JwtToken = tokenString,

RefreshToken = refreshToken,

Nonce = nonce,

EmailId = userName

};

return newToken;

}

}

The **GenerateToken** process is outlined like this -

string privatePath = Convert.ToString(ConfigurationManager.AppSettings["PRPath"], CultureInfo.InvariantCulture);

string publicKeyPath = Convert.ToString(ConfigurationManager.AppSettings["PUPath"], CultureInfo.InvariantCulture);

int expireMinutes = Convert.ToInt32(ConfigurationManager.AppSettings["JWTTokenLifeInMin"], CultureInfo.InvariantCulture);

public override void GenerateToken(string userName, string existingNonce, out string nonce, out string refreshToken, out string tokenString)

{

List<string> segments = new List<string>();

JwtHeader header = **Header**; // Header specific reference code below GenerateToken method logic

byte[] headerBytes = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(header, Formatting.None));

if (string.IsNullOrEmpty(existingNonce))

nonce = Guid.NewGuid().ToString();

else nonce = existingNonce;

refreshToken = Guid.NewGuid().ToString();

List<Claim> claims = new List<Claim>

{

new Claim("emailAddress", userName),

new Claim("iat", DateTimeOffset.Now.ToUnixTimeMilliseconds().ToString()),

new Claim("exp", GetJwtExpiryTime(expireMinutes)), // expireminutes is configurable. Actual code below this method

new Claim("jti", nonce)

};

JwtPayload jwtPayload = new JwtPayload(claims);

var jwtPayloadPart = jwtPayload.Base64UrlEncode();

segments.Add(Base64Helper.UrlEncode(headerBytes));

segments.Add(jwtPayloadPart);

string stringToSign = string.Join(".", segments.ToArray());

byte[] bytesToSign = Encoding.UTF8.GetBytes(stringToSign);

byte[] keyBytes = Convert.FromBase64String(KeyHelper.LoadFromFile(privatePath));

var privKeyObj = Asn1Object.FromByteArray(keyBytes);

var privStruct = RsaPrivateKeyStructure.GetInstance((Asn1Sequence)privKeyObj);

ISigner sig = SignerUtilities.GetSigner(SignerName); // SignerName is configurable and changes based on the SHA key size set

sig.Init(true, new RsaKeyParameters(true, privStruct.Modulus, privStruct.PrivateExponent));

//Here we are using we are using BouncyCastle’s built in implementation to read the private key file using RsaKeyParameters

from Org.BouncyCastle.Crypto.Parameters namespace whereas in case of Public Key (token verify process) the

RSAParameters from System.Security.Cryptography namespace is used.

sig.BlockUpdate(bytesToSign, 0, bytesToSign.Length);

byte[] signature = sig.GenerateSignature(); //The hash is signed using BouncyCastle’s Signing mechanism

segments.Add(Base64Helper.UrlEncode(signature));

tokenString = string.Join(".", segments.ToArray());

}

private JwtHeader Header

{

get

{

return new JwtHeader { alg = "RS" + KeySize.ToString(), typ = "JWT" }; // Here KeySize is configurable based on web.config key. **Currently value is 256 which gives us alg = RS256**

}

}

private string SignerName

{

get

{

return "SHA" + KeySize.ToString() + "withRSA"; // Here KeySize is configurable based on web.config key. **Currently value is 256 which gives us SignerName = SHA256withRSA**

}

}

private string AlgorithmName

{

get

{

return "SHA" + KeySize.ToString(); // Here KeySize is configurable based on web.config key. **Currently value is 256 which gives us AlgorithmName = SHA256**

}

}

private static string GetJwtExpiryTime(int timestamp)

{

return DateTimeOffset.Now.AddMinutes(timestamp).ToUnixTimeMilliseconds().ToString();

}

1. Create a return object of new Model class – TokenInformation from the property values for JwtToken, Nonce, RefreshToken and EmailId into the object.
2. Return TokenInformation to LoginRepository.Authenticate method

**Note -** This entire process of token generation was verified by taking the public and private key values as well as the newly generated jwt token string and making sure that on the jwt.io site the signature shows verified.

**Creating User Boundary and new Session**

LoginRepository.Authenticate will call new SP AddUserBoundaryAndCreateSession to create boundary data and session – passing in - jwtToken, Nonce, RefreshToken and EmailId to the SP.

The SP will pull the current boundary data for currently logged in user’s email id and insert jwttoken, nonce, refreshToken and boundary information into the table.

**Create a Session Cookie and Add to LoginController.Authenticate response**

1. In the Authenticate method response model, update the property value for RefreshToken into the model object.
2. Get the Authenticate method response into an HttpResponseMessage object. Then set the cookie into response using,

private HttpResponseMessage SetResponseCookie(UserInfo userInfo)

{

HttpResponseMessage response = Request.CreateResponse(HttpStatusCode.OK, userInfo);

var cookie = new CookieHeaderValue(Constants.JwtTokenCookie, userInfo.JwtToken)

{

HttpOnly = true,

Secure = true,

Expires = DateTimeOffset.Now.AddDays(1),

Domain = Request.RequestUri.Host,

Path = "/"

};

// Set the cookie in the response message.

response.Headers.AddCookies(new CookieHeaderValue[] { cookie });

return response;

}

Case 2. **Processing Other API calls**

1. Checking validity of input request based on cookie

public override void OnAuthorization(HttpActionContext filterContext)

{

try

{

if (filterContext.Request.RequestUri.Scheme != Uri.UriSchemeHttps &&

Convert.ToBoolean(ConfigurationManager.AppSettings["CheckForSSL"]) &&

!TokenHelper.IsForwardedSsl(filterContext) &&

!TokenHelper.CheckReferrer(filterContext))

{

filterContext.Response = new HttpResponseMessage(HttpStatusCode.Forbidden);

return;

}

Collection<CookieHeaderValue> cookieCollection = new Collection<CookieHeaderValue>();

cookieCollection = filterContext.Request.Headers.GetCookies(Constants.JwtTokenCookie);

//Cookie not found on the Input Request

if (cookieCollection == null || cookieCollection.Count == 0)

{

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

}

else //Cookie found on the Input Request

{

//Read input cookie and extract jwtToken

CookieHeaderValue cookie = cookieCollection.FirstOrDefault();

string jwt = string.Empty;

if (cookie != null)

{

jwt = cookie[Constants.JwtTokenCookie].Value;

string[] jwtParts = jwt.Split('.');

if (jwtParts.Length < 3)

{

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

}

else

{

string payLoadJson = string.Empty;

RsJwt jwtVerifier = new RsJwt

{

KeySize = Convert.ToInt32(ConfigurationManager.AppSettings["KeySize"])

};

if (!jwtVerifier.VerifyToken(jwt, out payLoadJson)) // Actual VerifyToken mechanism given below this method

{

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

SplunkLogging.GetInstance.Error(ErrorEventType.TokenInvalid, new Exception("Token Signature Invalid"));

return;

}

else

{

JavaScriptSerializer jsonSerializer = new JavaScriptSerializer();

JwtPayloadResponse jwtPayloadResponse = jsonSerializer.Deserialize<JwtPayloadResponse>(payLoadJson);

//Extract claims information like nonce and emailid and push into Request.Properties collection

filterContext.Request.Properties.Add("nonce", jwtPayloadResponse.Jti);

filterContext.Request.Properties.Add("emailid", jwtPayloadResponse.EmailAddress);

if (!filterContext.Request.RequestUri.AbsoluteUri.Contains("RefreshToken"))

{

//Check for time expired claim or other claims

if (DateTimeOffset.Now.ToUnixTimeMilliseconds() > Convert.ToInt64(jwtPayloadResponse.Exp))

{

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized);

SplunkLogging.GetInstance.Error(ErrorEventType.TokenExpired, new Exception("Token Expired. Please Refresh Token"));

return;

}

}

}

}

}

}

}

catch (Exception)

{

filterContext.Response = filterContext.Request.CreateResponse(HttpStatusCode.Unauthorized, "JWT is rejected");

return;

}

base.OnAuthorization(filterContext);

}

Inside the current TokenAuthenticationAndSSLCheck class for AuthorizationFilter, add above code so that following actions are taken care of –

1. Make sure Request contains a cookie. If not return Unauthorized
2. If Request contains a cookie, open it and fetch jwtToken.
3. Hash and VerifySignature on the (Header + Payload) JWT to make sure hash matches with the original one.
4. If hash matches, make sure the cookie has not expired.
5. Else return Unauthorized Response.

Actual Token verification happens like this –

public override bool VerifyToken(string token, out string payloadJson)

{

string[] parts = token.Split('.');

string header = parts[0];

string payload = parts[1];

string signature = parts[2];

byte[] crypto = Base64Helper.UrlDecode(parts[2]);

string headerJson = Encoding.UTF8.GetString(Base64Helper.UrlDecode(header));

payloadJson = Encoding.UTF8.GetString(Base64Helper.UrlDecode(payload));

byte[] keyBytes = Convert.FromBase64String(KeyHelper.LoadFromFile(publicKeyPath));

AsymmetricKeyParameter asymmetricKeyParameter = PublicKeyFactory.CreateKey(keyBytes);

RsaKeyParameters rsaKeyParameters = (RsaKeyParameters)asymmetricKeyParameter;

RSAParameters rsaParameters = new RSAParameters

{

Modulus = rsaKeyParameters.Modulus.ToByteArrayUnsigned(),

Exponent = rsaKeyParameters.Exponent.ToByteArrayUnsigned()

};

//Here we are using the RSAParameters from System.Security.Cryptography, whereas in case of Private Key (token

generate process), we are using BouncyCastle’s built in implementation to read the private key file using RsaKeyParameters

from Org.BouncyCastle.Crypto.Parameters namespace.

BouncyCastle doesn’t have RsaPublicKeyStructure mechanism similar to the RsaPrivateKeyStructure (to read in the private key

file) as done in Token Generate process. Hence the slight difference in private vs public.

RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();

rsa.ImportParameters(rsaParameters);

byte[] hash = ComputeHash(header, payload); // Actual code implementation below this VerifyToken method

RSAPKCS1SignatureDeformatter rsaDeformatter = new RSAPKCS1SignatureDeformatter(rsa);

rsaDeformatter.SetHashAlgorithm(AlgorithmName);

byte[] tmp = Base64Helper.UrlDecode(signature);

return rsaDeformatter.VerifySignature(hash, tmp); //This uses BouncyCastle’s signature verification mechanism

}

private byte[] ComputeHash(string header, string payload)

{

HashAlgorithm sha = HashAlgorithm.Create(AlgorithmName);

if (sha == null) throw new Exception("Given key size is not valid.");

return sha.ComputeHash(Encoding.UTF8.GetBytes(header + '.' + payload));

}

1. **Refresh Token** scenario

In the above flow when the input request contains a JWT token which has got an expired timestamp, then the Auth layer returns an Unauthorized Status (401).

On every request call Angular app will capture the status of 401 and if found, will fire a separate RefreshToken request to Web API.

Logic for refresh token on API side –

1. Input cookie on the RefreshToken will be read in and its nonce will be used to check if it is a valid session id containing an entry in the sessionboundary table.
2. If an entry is not found, then the RefreshToken request will be rejected.
3. If an entry is found, then the existing AuthToken.GenerateToken method will be internally called for generating a new JWTToken, RefreshToken and the values for the same will be updated for given nonce and email id (from input cookie) into the backend boundary table.

Inside controller which serves the RefreshToken method,

bool nonceFetch = Request.Properties.TryGetValue("nonce", out object nonceValue);

if (nonceFetch)

{

refreshToken.Nonce = Guid.Parse(Convert.ToString(nonceValue));

}

bool nonceExists = CheckNonceExistence(refreshToken.Nonce, Guid.Parse(refreshToken.CurrentRefreshToken));

if (!nonceExists)

return null;

newToken = authToken.GenerateToken(refreshToken.EmailId, Convert.ToString(refreshToken.Nonce));

if (!UpdateRefreshTokenInDB(Guid.Parse(newToken.Nonce), Guid.Parse(newToken.RefreshToken)))

{

return null;

}

newToken.Nonce = string.Empty;

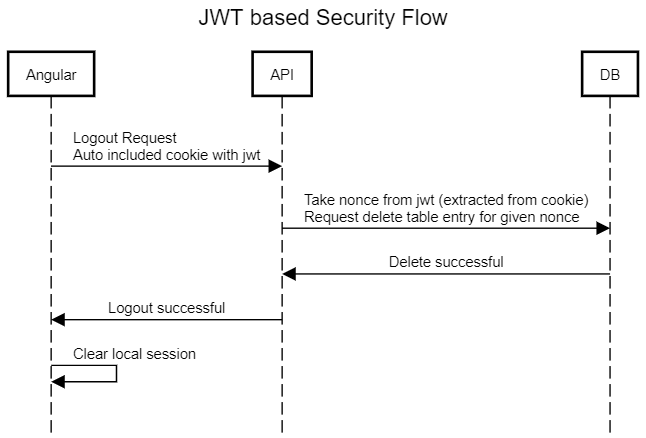
return newToken;

The new object model containing JWTToken & RefreshToken will be read in the RefreshToken controller method and the updated token value for JwtToken will be pushed into cookie along with returning the new RefreshToken in the response body to the angular app.

On receiving a valid response containing a new RefreshToken, the same will be updated on the user session on client side and sent in every further RefreshToken request.

Also, the original failed call will be then reattempted with the new JWT (auto-included in cookie).

1. Logout



Currently Logout updates the datetimestamp on IA\_AccessLog table. Now in addition for the log out request for specific user (given from his nonce on cookie), the table entry on the databoundary table will also be deleted, so that the next time, user tries to use that same cookie somehow, the Unauthorized Error response is automatically returned by the AuthFilter layer itself.

We will update the current SP – RecordSessionEnd to take in nonce as well, so that along with updating the datetimestamp, the nonce entry on databoundary table is also deleted explicitly.

**Securely Launching the App while Impersonation and launching other screens from new portal**

In the current approach, we are launching new portal from @YS by using encrypted query parameters in the request. To make this path secure we will need to make the following changes:

1. Move the query parameters to body over https.
2. Add mutual authentication on @YS and new portal server. To implement this approach, we will have to configure a private and public key pair on both the servers. Using the private key @YS will encrypt the parameters in the body. When the new portal server gets request from @YS server, it will use the public key configured for @YS and decrypt the body. This approach will ensure that the new portal is getting request from the @YS server only.

The above approach will be used with other applications launched from the new portal. Those applications will also have a private and public key configured which they will use for encrypting and decrypting the body.

**Dependencies:** The above-mentioned approach will need changes in @YS and other applications. These applications will need to be configured with the private and public key. They will also need to remove the existing hardcoded encryption key and use the private key for encrypting any data.

**Steps to implement the changes in AYS and other applications –**

1. Create a private key on the server. Share the public key pem file with CCP Application.
2. Remove the hardcoded encryption key used to encrypt the data to be sent to the CCP application.
3. Add logic to read the private key present on the @YS application server and use this key to encrypt the data sent to the CCP application.
4. The data was earlier sent as query string to the CCP application. This will now be moved to body with key same as the one used in the query params.
5. @YS application server will save the public key of CCP application. This key will be used to decrypt the response received from the CCP application.

Similar changes will be done in other applications which are invoked from the CCP application. All the applications will need to share their public key with CCP and save the public key of CCP on their server. So the encryption and decryption algorithm can remain the one used currently, the only difference here will be the key will not be hardcoded.