|  |
| --- |
| **Penetration Testing**  **Issues & Solutions** |
| **Document Version / Details: Ver 1.0/1-Mar-19** |
|  |

****

**Record of Release**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version No. | Modified By | Reviewed By | Authorized By | Release Date | Modifications Done |
| Ver.1.0 | *Omkar Ghaisas* | *Ismail Shaikh/Jeetendra Shenoy* |  | 1-Mar-19 | First Version |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Table of Contents**

[1.0 Scope 3](#_Toc8121764)

[2.0 System Wide Fixes 3](#_Toc8121765)

[2.1 Issue: Fixed secret in JWT shall be replaced with RSA key. (Key Rotation) 3](#_Toc8121766)

[2.2 Issue: Currently JWT is token is used with HS256. This shall be replaced with RS256. 3](#_Toc8121767)

[2.3 Issue: jti in JWT to act like Session Token. 3](#_Toc8121768)

[2.4 Issue: Jwt.verify function should have algorithm as parameter. 3](#_Toc8121769)

[2.5 Issue : Nonce to be used in every call. 3](#_Toc8121770)

[2.6 Boundary Check 3](#_Toc8121771)

[2.7 SQL input Sanitization 6](#_Toc8121772)

[2.8 Dynamic Query in Stored Procedure 6](#_Toc8121773)

[2.9 Password 7](#_Toc8121774)

[2.10 Web API input Parameter Sanitization 9](#_Toc8121775)

[2.11 JWT to be sent as Cookie 11](#_Toc8121776)

[2.12 Output Encoding / Sanitization 11](#_Toc8121777)

[2.13 Remove Extra OPTIONS calls 11](#_Toc8121778)

[2.14 Log all input validation failures 12](#_Toc8121779)

[2.15 Log attempts to connect with invalid or expired session tokens 13](#_Toc8121780)

[2.16 Disable AutoComplete/Autofill on portal 13](#_Toc8121781)

[2.17 Converting GET to POST calls 13](#_Toc8121782)

[2.18 Encrypting the Connection Strings 15](#_Toc8121783)

[2.19 Splunk Log – Do not log full user viewed url. Only log page viewed 16](#_Toc8121784)

[2.20 Unnecessary information to be removed from Http response 16](#_Toc8121785)

[3.0 Overall Request Response Flow 17](#_Toc8121786)

[4.0 Low and informational Issues 18](#_Toc8121787)

[Issue 1: Strict transport security not enforced: 18](#_Toc8121788)

[Issue 2: Cross-site scripting (reflected): 19](#_Toc8121789)

[Issue 3 – Invalid input in JSON is returned as response in Exception Message 19](#_Toc8121790)

[Issue 4: Cross-origin resource sharing: 19](#_Toc8121791)

[Issue 5: Input returned in response(reflected): 20](#_Toc8121792)

[Issue 6 – Invalid Input in JSON is returned as output message in Exception message 20](#_Toc8121793)

[Issue 7: Suspicious input transformation: 20](#_Toc8121794)

[Issue 8: SSL Cookie without a secure flag set: 21](#_Toc8121795)

[Issue 9: Cross-domain script include: 21](#_Toc8121796)

[Issue 10: Cookie without HttpOnly flag set: 21](#_Toc8121797)

[Issue 11: Frameable response (potential Clickjacking): 22](#_Toc8121798)

[Issue 12: Email Address disclosed: 22](#_Toc8121799)

[Issue 13: Cacheable HTTPS response: 22](#_Toc8121800)

[Issue 14: HTML does not specify charset : 22](#_Toc8121801)

# Scope

Discuss overall System Wide Security fixes which will also take care of Penetration Testing issue.

# System Wide Fixes

## Issue: Fixed secret in JWT shall be replaced with RSA key. (Key Rotation)

**Fix:** RSA key has two parts, Public and Private Key. Private Key will reside on the server and never be shared. Private will be used to sign and public key will be used to verify signature. Private Key will never leave the server. On Refresh, JWT Token and Refresh token shall be regenerated.

## Issue: Currently JWT is token is used with HS256. This shall be replaced with RS256.

**Fix:** Header and Payload shall be hashed using SHA-256. This can be done fast, and it is unique representation of the input data that is much smaller than the actual data itself. We then take the hash output and sign that instead of the whole data (header plus payload) using RSA private key, which gives us the RS256 signature. We then append it to the JWT as the last of the 3 parts and send it. The receiver of the JWT will take the header and payload and hash everything with SHA-256. Then verify the signature using the public key and obtain the signature hash. The receiver compares the signature hash with the hash that he calculated himself based on the Header and the Payload. If the two hashes match, then JWT was indeed created by the Authentication server.

The core mechanism for hashing, signing and verification of token is executed using the BouncyCastle library - http://www.bouncycastle.org/csharp/)

## Issue: jti in JWT to act like Session Token.

**Fix:** On Login unique guid will be created and put in the jti of JWT. It will act like Session id and will remain same throughout from login till logout. This entry will be present in the boundary database and will removed when User logs out.

## Issue: Jwt.verify function should have algorithm as parameter.

**Fix:** Verify that latest jwt.verify function is used which takes in algorithm (RS256) as parameter.

## Issue : Nonce to be used in every call.

**Fix:** Taken care with 2.3, Nonce is a pseudo-random number issued in an authentication protocol to ensure that old communications cannot be reused in replay attacks. Server will provide nonce to the client in Response which the client has provide to the server in the request. This Nonce with passed as CSRF token in the Header. As async and out of order call is supported, nonce provide by the server, multiple call can come to client with the same nonce. On the server, nonce will have expiry time. All request after the expiry will have 419 HTTP error status (Page Expired) and new nonce will we provide.

## Boundary Check

**Sequence / Flow / Pseudo Logic for Boundary Check – Below we have provided an example how boundary check can be implemented.**

#### **Populating Boundary data -**

Case 1. Login - Flow

1. User logs in with his normal credentials (email id-password)
2. Web API method LoginController - Authenticate gets called with the UserName and Password input.
3. This web API calls LoginUser Stored Procedure (with username and password as input)
4. If invalid credentials, Stored Procedure returns error and Web API returns 401 Http status (Unauthorized)
5. If credentials are valid Stored Procedure returns success.
6. On receiving success Authenticate method calls another method PopulateBoundaryData (written in CommonRepository) with the current logged in user.
7. PopulateBoundaryData will populate the boundary data for the current logged in user by calling new SP – dbo.PopulateUserBoundaryData
8. SP PopulateUserBoundaryData will populate the xml into the table IA\_UserDataBoundaryInformation.
9. Web API will return 200 Http status response with JWT cookie set on the response.

New Table - dbo.IA\_UserDataBoundaryInformation will hold the boundary information for every user.

**Pseudo logic to set the existing boundary information** –

Insert into dbo.IA\_UserDataBoundaryInformation

SELECT IA\_Users.UserID 'UserID',

IA\_AccountUsers.AccountID 'AccountID',

              PIBillAccount 'BillingAccountNumbers',

PIPolicyNum 'PolicyNumbers'

FROM IA\_Policies INNER JOIN PIMaster ON PolicyNumber=PIPolicyNum

INNER JOIN IA\_Account  ON IA\_Account.AccountID=IA\_Policies.AccountID

INNER JOIN IA\_AccountUsers  ON IA\_AccountUsers.AccountID=IA\_Account.AccountID

RIGHT JOIN IA\_Users  ON IA\_Users.UserID=IA\_AccountUsers.UserID

WHERE IA\_Users.Email = @Email

ORDER BY IA\_AccountUsers.AccountID

FOR XML AUTO

**Table Schema** –

Nonce (Primary key – Guid)

JWT Token

Nonce Expiry

EmailId

UserData xml

**Sample UserData xml** –

<IU UserID="860">

<IA AccountID="783" agentCode = "">

<PM billingAccountNumber="200639591" PolicyNumber="4095044" />

<PM billingAccountNumber="200639591" PolicyNumber="4095045" />

<PM billingAccountNumber="200639591" PolicyNumber="4095046" />

<MUserID="123"/>

<MUserID="222"/>

<MUserID="333"/>

</IA>

<IA AccountID="784" agentCode = "">

<PM billingAccountNumber="211639591" PolicyNumber="5095044" />

<PM billingAccountNumber="211639591" PolicyNumber="5095045" />

<PM billingAccountNumber="211639591" PolicyNumber="6095046" />

<CM claimNumber="200639" />

</IA>

</IU>

Case 2. Add/Modify/Delete if any (Policy, User, Account etc)

1. The additions where the boundary data gets affected, boundary data in the database will be updated.
2. On successful update, addition / deletion of user / policy etc. which affects boundary data, a new **SP -** **UpdateUserBoundaryData** will be called to update the user’s boundary data in the database (method written in CommonRepository)

**Pseudo logic to update new boundary information for user**

Requery the latest xml from tables using the Select logic as per Case 1

UPDATE dbo.IA\_UserDataBoundaryInformation

SET UserData = Latest XML

WHERE EmailId = @EmailId (for current user)

#### **Checking of Boundary Data -**

In any Web API calls other than Login, boundary checks will be performed on UserId, AccountId, BillingAccountNumber, PolicyNumber, ClaimNumber & AgentCode. JWT cookie details will be used to identify the user and check with the boundary data.

Every Controller method, will call a CommonRepository class method **ValidateInput** that takes in any input data model and goes thru all the properties inside it.

It will find the specific keys - UserId, AccountId, BillingAccountNumber, PolicyNumber, ClaimNumber & AgentCode and check for boundary values against the values present in xml.

**Pseudo logic for Boundary Check on all API Controllers**

public bool ValidateInput<T>(T controlInput)

{

// Here type T can take in any Generic Input model to iterate over

bool validationStatus = true;

T obj = default(T);

obj = Activator.CreateInstance<T>();

foreach (PropertyInfo prop in obj.GetType().GetProperties())

{

if (**prop.Name**.ToUpperInvariant().Equals(**Primary Key Captured in Boundary XML**) &&

!string.IsNullOrEmpty(Convert.ToString(prop.GetValue(controlInput))))

{

if (NOT **Boundary XML Value for matching field**.Equals(Convert.ToString(prop.GetValue(controlInput))))

{

**validationStatus = false;** // Indicates that the Primary key value sent in Input doesn’t match the value

on boundary XML for this user. Rejecting the request

**break;**

}

}

}

}

This common method ValidateInput will be called from every Controller, where input data is to be validated for boundary condition. Actual controller functionality will be called only when ValidateInput method returns a Boolean true. Else a Forbidden (403) will be returned and an appropriate error message will be shown to the user.

## SQL input Sanitization

For various input data types below type of regex will be defined

1. Regex for numeric input: (\"[0-9]\")\*)
2. Regex for date: ^(0[1-9]|[12][0-9]|3[01])[- / .](0[1-9]|1[012])[- / .](19|20)\d\d$
3. Regex for email: ^(?(")(".+?(?<!\\)"@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`{}|~\w])\*)(?<=[0-9a-z])@))(?([)([(\d{1,3}.){3}\d{1,3}])|(([0-9a-z][-0-9a-z]\*[0-9a-z]\*.)+[a-z0-9][-a-z0-9]{0,22}[a-z0-9]))$
4. Regex for SQL keywords (used on strings): ('(''|[^'])\*')|(;)|(\b(ALTER|CREATE|DELETE|DROP|EXEC(UTE){0,1}|INSERT( +INTO){0,1}|MERGE|SELECT|UPDATE|UNION( +ALL){0,1})\b)

With ignorecase on regexoptions

#### **Implementation of Sanity check for Database Input parameters –**

DatabaseHelper class actually executes all the SPs in the database. All the parameters to the SP are already passed in to the method of this class. Here we will check for input parameter DBType and apply appropriate Regex (based on type) to check against the parameter values.

If any parameter value fails in Regex validation, the Web API call will be rejected with Http status 403 (Forbidden) without executing any SP and an appropriate error will be shown to the user.

## Dynamic Query in Stored Procedure

If the Stored Procedure contains dynamic query where column names are passed as parameter, then these Stored Procedures will be replaced. Copy of existing Stored Procedure will be created with a new name and all the column name will hardcode in the Stored Procedure and only required values will be passed as parameter.

**Pseudo logic to fix the Dynamic Query issue –**

SP’sDDQueryPolHistoryPersonal and DDQueryPolHistoryCommercial – Have dynamic query logic inside

Currently they take in Input Params –

@SearchColumn varchar(30),/\* The column being searched \*/

@SearchValue varchar(40), /\* Value being searched for \*/

@SortColumn varchar(100), /\* Column/s to Order By \*/

@AccessLevel varchar(5), /\* (P)arent, (S)ubproducer, or (E)mployee \*/

@AgentCode varchar(10), /\* Agent Code \*/

@MatchType varchar(10), /\* Any special conditions? If this = "Exact", we need to use =, not 'LIKE') \*/

@InForce varchar(1), /\* IF Y, only show in-force policies \*/

@Copy varchar(1), /\* A=Agent, I=Insured, <anything else>=Both \*/

@NonRenewed varchar(1), /\* IF Y, only show non-renewed policies \*/

@FromPage varchar(1) /\* P=Policy, T=Transaction List \*/

Out of these, only SearchValue (that is primarily a Policy Number sent in) is really used inside to query to pull results.

Rest all other input parameters have hard coded values and as such would be removed from the Input Params list and their current input hard coded value will be directly hard coded inside the SP SQL logic itself.

## Password

We will use SHA-256 hash function for Password Encryption on the front-end (Angular application)

Steps to Include and use SHA-256 Library –

1. Install the library using the below command.

npm install js-sha256

1. Use the below function to convert the password into hash.

sha256('password string’);

Hashing mechanism to be included in below modules.

* Login
* Reset password
* Registration
* Forgot password
* Account profile

**Password Encryption using SHA256 Hash Algorithm on backend server (Web API)**

1. Add New Reference for System.Security.Cryptography
2. To Store a Password

* Generate a random salt using a RNGCryptoServiceProvider

Example:

// function for random number generation

public static string GenerateRandomSalt()

{

var rng = new RNGCryptoServiceProvider();

var bytes = new Byte [32];

rng.GetNonZeroBytes(bytes);

return Convert.ToBase64String(bytes);

}

* Prepend the salt to the password and hash it with a standard password hashing function SHA256.

Example:

public static string ComputeSha256Hash(string salt, string pwd)

{

using (SHA256 sha256Hash = SHA256.Create())

{

byte[] bytes = sha256Hash.ComputeHash(Encoding.UTF8.GetBytes(String.Concat(salt, pwd)));

StringBuilder builder = new StringBuilder();

for (int i = 0; i < bytes.Length; i++)

{

builder.Append(bytes[i].ToString("x2"));

}

return builder.ToString();

}

}

* Save both the salt and the hash in the user's database record.

Example:

INSERT INTO IA\_USERS (Hash,Salt) VALUES(@Hash,@Salt);

1. **To Validate a Password (authenticating a user login)**

* Retrieve the user's salt and hash from the database.

Example:

SELECT Hash, Salt WHERE UserId = 888;

* Prepend the salt to the given password and hash it using the same hash function.

Example:

**Note**: Salt value coming from database

DECLARE @Salt VARCHAR (100);

SET @Salt = (SELECT Salt FROM IA\_Users WHERE UserId = 888)

string salt = Convert.ToString(Salt);

public static string ComputeSha256Hash (string salt, string pwd);

* Compare the hash of the given password with the hash from the database.
* If they match, the password is correct. Otherwise, the password is incorrect.

Example:

private static bool IsPasswordValid(string passwordToValidate, string salt, string oldHash)

{

var newHash = ComputeSha256Hash(passwordToValidate, salt);

byte[] newPass = Encoding.UTF8.GetBytes(newHash);

byte[] oldPass = Encoding.UTF8.GetBytes(oldHash);

return newPass.SequenceEqual(oldPass);

}

**Full Sample example**

static void Main(string[] args)

{

/\*

\* Per User we generate random salt value

\* output

\* saltvalue : new salt generate per user

\*/

var saltValue = GenerateRandomSalt();

/\*

Input Parameter ad8beb1483231412e9c0497e535226103c8df1e6a8420cc370d075727699bbae: SHA256 hash - which is coming from Angular

saltValue : Get from database

Output

hashValue : New Hash Value

\*/string hashValue = ComputeSha256Hash("ad8beb1483231412e9c0497e535226103c8df1e6a8420cc370d075727699bbae", saltValue);

/\*

Input Parameter ad8beb1483231412e9c0497e535226103c8df1e6a8420cc370d075727699bbae: SHA256

hash - which is coming from Angular

saltValue : Get from database

hashValue : Database

Output

true : Valid user

or

false : Invalid User

\*/

if (IsPasswordValid("ad8beb1483231412e9c0497e535226103c8df1e6a8420cc370d075727699bbaq",

saltValue, hashValue))

{

Console.WriteLine("Valid User");

}

else

{

Console.WriteLine("Invalid User");

}

Console.ReadKey();

}

## Web API input Parameter Sanitization

Input JSON shall be scanned for Cross-Site script at the controller level. Key word (script, eval, alert, html etc will be blocked along with < >,. Standard CSS prevention will be added at the controller level

Install AntiXSS Library using Nuget package manager in Visual Studio

After the installation, you'll be able to see two new libraries in your projects references folder: AntiXssLibrary and HtmlSantizationLibrary.

All you need to do is to make a little change in the controller, so as to prevent XSS.

[](http://www.dotnet-programming.com/image.axd?picture=image_13.png)

In case any user will tries inserting a malicious script with the text message, it will be automatically removed.

**Additional Input Data Sanitization using Regex**

Add a reference to the System.ComponentModel.DataAnnotations assembly to every model class.

using System.ComponentModel.DataAnnotations;

namespace SampleApplication.Models

{

public class UserModel

{

[RegularExpression(@"^([a-zA-Z0-9\_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$"]

public string Email { get; set; }

[StringLength(10), MinimumLength = 10)]

public string MobileNumber { get; set; }

}

}

To validate a model explicitly, use ModelState Object in controller. We will check ModelState.IsValid before executing controller code. It will return false whenever the data is found to be invalid.

Model class will be shown as below:

namespace SampleApplication.Controllers

{

public class UserController : Controller

{

[HttpPost]

public HttpResponseMessage ServerMeta(RegistrationMetaModel mRegister)

{

if (!ModelState.IsValid)

{

return HttpResponse.BadRequest();

}

else

{

//Call further Controller functionality

}

}

}

}

## JWT to be sent as Cookie

JWT will be sent as Cookie with HttpOnly and Secure header. Cookies, when used with the HttpOnly cookie flag, are not accessible through JavaScript, and are immune to XSS. You can also set the Secure cookie flag to guarantee the cookie is only sent over HTTPS. CSRF can also be partially prevented by checking the HTTP Referrer and Origin header from your API. CSRF attacks will have Referrer and Origin headers that are unrelated to your application.

## Output Encoding / Sanitization

Specify AntiXssEncoder from the AntiXss library to be used as the default encoder for entire application using the encoderType setting in web.config as shown below.

<httpRuntime encoderType="System.Web.Security.AntiXss.AntiXssEncoder" />

Once configured, all the output needs to be Sanitized using Sanitizer.GetSafeHtmlFragment. For doing so, all string return types will be calling a string extension method that takes in current output string and returns a Sanitized output string.

**Pseudo Code –**

Extension method

public static string SanitizeOutput(this string input)

{

return Sanitizer.GetSafeHtmlFragment(input);

}

All Repository methods where string is being returned (either directly / within a model class) will call this SanitizeOutput method like

String returnValue = DBOutput[“Column1”].SanitizeOutput();

## Remove Extra OPTIONS calls

Remove the Extra Options calls which happen from the Angular Application.

1. OPTIONS requests or pre-flight requests are done when using Cross-origin resource sharing (CORS).
2. They are necessary when we are making requests across different origins.
3. This pre-flight request is made by some browsers as a safety measure to ensure that the request being done is trusted by the server. Meaning the server understands that the method, origin and headers being sent on the request are safe to act upon.

To disable OPTIONS methods call, followings are the possible steps we are need to incorporate in Web API,

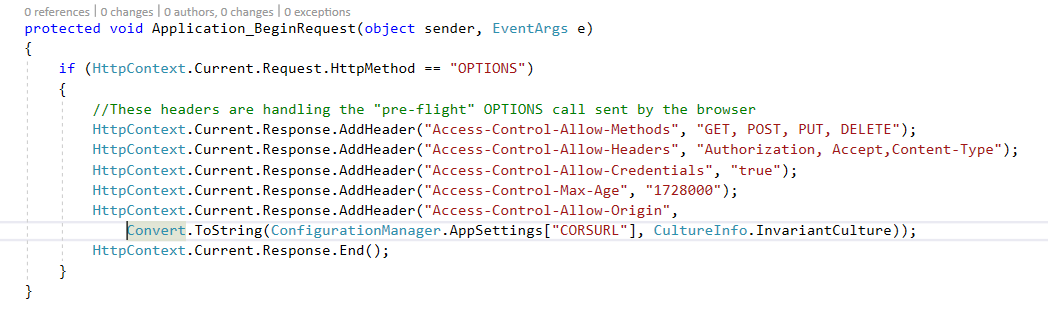
1. In web.config add following code in httpHandlers section,

<httpHandlers>

<add path="\*" verb="OPTIONS" type="System.Web.DefaultHttpHandler" validate="true"/>

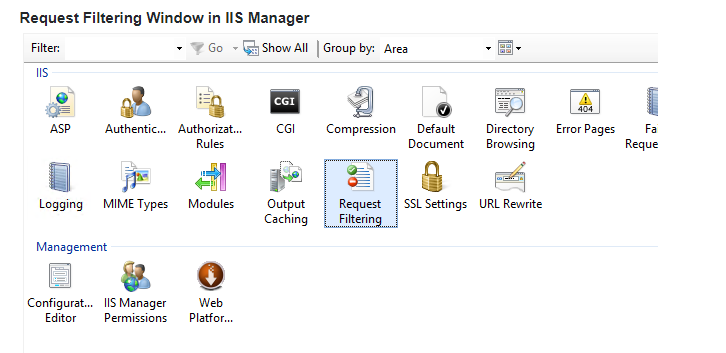
</httpHandlers>

1. In Global.asax.cs, add following code in Application\_BeginRequest event,



The same can be achieved from the IIS server itself on the Azure VM.

* Open the Website in IIS Manager
* Go to **Request Filtering** and open the Request Filtering Window.
* Go to **Verbs** Tab and Add HTTP Verbs to "Allow Verb" or "Deny Verb”. This allow to add the HTTP Verbs in the "Deny Verb" Collection.



## Log all input validation failures

During the Anti-XSS and Web API Input validation checks using the CommonRepository functionality, a try-catch will be wrapping the specific Input sanitization check call.

In case of failure, as the control flow goes to catch block, we will use the logging framework(such As Splunk) e.g. – SplunkLogging.GetInstance.Error(ErrorEventType.InputValidationFailure, exceptionmessage) to send the validation failure message to Splunk.

## Log attempts to connect with invalid or expired session tokens

During the Security layer check using MessageHandler, the http only cookie will be intercepted by the API framework and in case an unauthorized access is detected due to cracked cookie / otherwise, the 403 Forbidden Error will be returned as response.

During that process, when inside the exception catch flow, the existing logging framework will be used(such as Splunk) – e.g. SplunkLogging.GetInstance.Error(ErrorEventType.InputValidationFailure, exceptionmessage) to send the authorization failed message to Splunk.

## Disable AutoComplete/Autofill on portal

We will add the autocomplete=”off” feature to all the form tags to disable the autocomplete feature.

Following modules will be impacted -

* Login
* Forgot Password
* Reset Password
* Registration
* Account Profile
* Report a claim
* Contact Us

Pseudo code -

<form #loginForm="ngForm" [formGroup]="formGroup" novalidate autocomplete="off">

## Converting GET to POST calls

Both the Web API & Angular Application code will be updated to change the HttpGet to HttpPost

1. Portal Web API

In Web API solution, we need to make changes in below 2 projects:

Portal.DataModel Project -

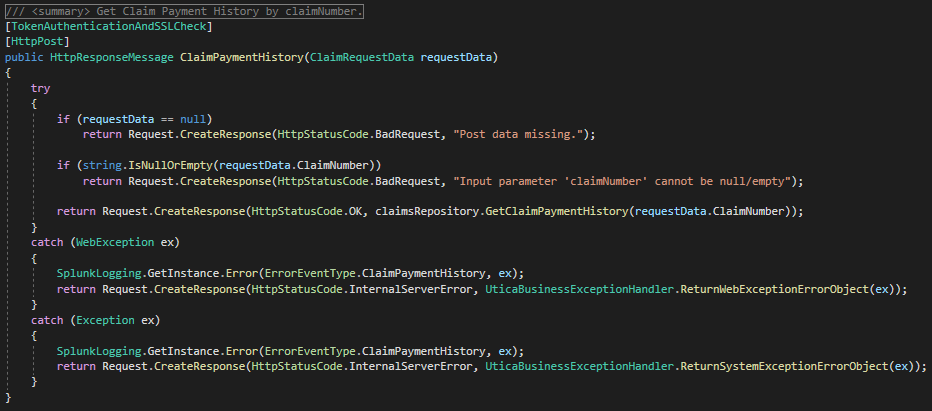
We need to create a RequestData class which will have all the input payload with all required input properties. One common model say ClaimRequestData will be created per module.

Portal.WebAPI Project -

1. In Web API project, we need to change the attribute per controller from [HttpGet] to [HttpPost].
2. Remove existing Input Parameter and replace it with RequestData which was created in Portal.DataModel.

Before processing the request, add a validation to check whether the input payload is not null. If it is null, then respond with 400 – Bad Request.

For E.g. Sample code below after making the above changes:



1. Angular App

In Angular App, we need to make changes in below 2 files:

1. **service.ts**

We need to change the http.get() method to http.post() and remove input parameters from query string and put them into request body.

Changes will look like this:

**Before:**

getYourClaims(accountId): Observable<any> {

const url = Constants.BASE\_URL + "/api/Claims/ViewAllClaims?accountId=" + accountId;

const options = this.createHeader();

return this.http.get(url, { headers: options });

}

**After:**

getYourClaims(requestBody): Observable<any> {

const url = Constants.BASE\_URL + "/api/Claims/ViewAllClaims”;

const options = this.createHeader();

const body = requestBody;

return this.http.post(url, body, { headers: options });

}

1. **component.ts**

In component file we need to remove input parameters from service and send request json body to service.

Changes will look like this:

**Before:**

this.claimService.getYourClaims(AccountId).subscribe(data => {

this.viewAllClaims = data;

}

**After:**

const jsonBody = {

"accountId": 123

}

this.claimService.getYourClaims(JSON.stringifiy(jsonBody)).subscribe(data => {

this.viewAllClaims = data;

}

## Encrypting the Connection Strings

1. Open Command Prompt with Administrator privileges
2. At the Command Prompt, enter:

cd C:\Windows\Microsoft.NET\Framework\v4.0.30319

1. In case your web Config is located in "F:\Portal-WebAPI" directory path, then enter the following to encrypt the ConnectionString:

ASPNET\_REGIIS -pef "connectionStrings" "F:\Portal-WebAPI"

**Note**: The parameter "**connectionStrings**" is case sensitive.

After encryption the ConnectionStrings section, should look something like below

<configuration>

<connectionStrings configProtectionProvider="RsaProtectedConfigurationProvider">

<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"

xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">

<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1\_5" />

<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">

<KeyName>Rsa Key</KeyName>

</KeyInfo>

<CipherData> <CipherValue>ZbDTF00MYzUUW5U3w3PU0rfiAH1UKhvuLSNWPmB/YifBKne6HAWfVc3CnKVimyP8SFyamaR5oAIAxj/xavfpox8EOYXNI+afsksiuA5huSDupCZKNuXq+VCZrdIyn6YOq+W7s3Ojlu7q9VwKcoKurl28l2hcPvWkBk11KYB7hr0=</CipherValue>

</CipherData>

</EncryptedKey>

</KeyInfo>

</EncryptedData>

</connectionStrings>

</configuration>

**Accessing Decrypted Configuration Settings**

ASP.NET automatically decrypts the contents of the *Web.Config* file when it processes the file. No additional steps are needed on the code side.

**Decrypting the Connection String**

Perform the following command to decrypt the connectionStrings element in the Web.config file.

ASPNET\_REGIIS -pdf "connectionStrings" "F:\Portal-WebAPI"

**Note**: - The encrypted connection strings are only readable by the application at runtime, if the app is running under the Admin user privileges on the Azure Server.

## Splunk Log – Do not log full user viewed url. Only log page viewed

Update the Splunk log body being sent from the angular application to only include the page name and not the full url.

Pseudo Code -

Update the "userViewed" property on the Log body to only include window.location.pathname

## Unnecessary information to be removed from Http response

**Current HTTP Response headers**

Example:

Response Header

Access-Control-Allow-Origin: http://localhost:4200

Cache-Control: no-cache

Content-Length: 1053

Content-Type: application/json; charset=utf-8

Date: Fri, 08 Mar 2019 07:15:06 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?B?QzpcVXNlcnNcU1JhZ2l0XFNvdXJjZVxXb3Jrc3BhY2VzXEluZm9yY2UgUG9ydGFsXFVUSUNBLldlYkFQSVxVVElDQS5XZWJcYXBpXExvZ2luXEF1dGhlbnRpY2F0ZQ==?=

1. Removing IIS Server Version

Add below code snippet to the Web.config on the Web API

Example: Server: Microsoft-IIS/10.0

<system.webServer>

<rewrite>

<outboundRules rewriteBeforeCache="true">

<rule name="Remove Server header">

<match serverVariable="RESPONSE\_Server" pattern=".+" />

<action type="Rewrite" value="" />

</rule>

</outboundRules>

</rewrite>

</system.webServer>

1. Remove X-Powered-By

Add below code snippet to the Web.config on the Web API

<system.webServer>

<httpProtocol>

<customHeaders>

<remove name="X-Powered-By" />

</customHeaders>

</httpProtocol>

<system.webServer>

1. Remove X-AspNet-Version

Add below code snippet to the Web.config on the Web API

<httpRuntime enableVersionHeader="false" />

1. RemoveServerHeader requestFiltering in IIS 10.0

Add below code snippet to the Web.config on the Web API

<security>

<requestFiltering removeServerHeader ="true" />

</security>

# Overall Request Response Flow

Only https Request with TLS 1.2 and secure Ciphers will allowed on the WAF/Firewall/Router. Input body and Parameters if any will be checked for CSS (section 2.12).

If Login Request then, (Section 2.9) password algo(SHA256) check and then on login validation JWT cookie (Section 2.1, 2.2, 2.3, 2.11) will be created. Nonce will be stored in DB (Section 2.3) and Boundary Data (Section 2.6) will be applied on every request thereafter.

If Logout Request, then Nonce shall be dropped from database (Section 2.3).

On Refresh Token call, Nonce and existing JWT cookie to be checked (Section 2.1, 2.2, 2.3, 2.4) and new JWT cookie to be created and sent to the client (Section 2.1, 2.2, 2.3).

In all other Web API calls, Nonce and existing JWT cookie to be checked (Section 2.1, 2.2, 2.3, 2.4).

Boundary check validation will done on the inputs (section 2.8).

If any DB Query is to be executed, then (Section 2.9) SQL data sanitization will also be done.

All outputs will have Sanitization (Section 2.12) applied on them before returning from API.

# Low and informational Issues

### Issue 1: Strict transport security not enforced:

The application fails to prevent users from connecting to it over unencrypted connections. An attacker able to modify a legitimate user's network traffic could bypass the application's use of SSL/TLS encryption, and use the application as a platform for attacks against its users. This attack is performed by rewriting HTTPS links as HTTP, so that if a targeted user follows a link to the site from an HTTP page

**FIX** - Strict-Transport-Security: max-age=<expire-time>

Strict-Transport-Security: max-age=<expire-time>; includeSubDomains

Strict-Transport-Security: max-age=<expire-time>; preload

**Reference -** <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>

The element system.applicationHost is not allowed in Web.config and the <hsts> element of the <siteDefaults> element contains attributes that allow you to configure default HTTP Strict Transport Security (HSTS) settings for a site on IIS 10.0 version 1709 and late.

**Reference:**https://docs.microsoft.com/enus/iis/configuration/system.applicationhost/sites/sitedefaults/hsts



### Issue 2: Cross-site scripting (reflected):

Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed into the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request that, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application.

**FIX –** Input should be validated on arrival with Regex and

User input should be HTML-encoded at any point when copied in application response.

This issue will be handled by **section 2.10 Web API input Parameter Sanitization and 2.12 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**

Example - All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (&lt; &gt; etc).

### Issue 3 – Invalid input in JSON is returned as response in Exception Message

The value of the **searchValue** JSON parameter is copied into the HTML document as plain text between tags. The payload **w01fu<a b=c>oeh9h** was submitted in the searchValue JSON parameter. This input was echoed unmodified in the application's response.

For the present example the Content-Type was set to application/json which could not be modified.

For Example:

**Request**- {"searchColumn":"PolicyNum","searchValue":"5003370w01fu<a b=c>oeh9h"}

**Response**- "ExceptionMessage":"Conversion failed when converting the varchar value '5003370 w01fu<a b=c>oeh9h' to data type int."}

**FIX –** Input should be validated on arrival with Regex and

User input should be HTML-encoded at any point when copied in application response.

This issue will be handled by **section 2.12 Web API input Parameter Sanitization and 2.14 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**

Example - All HTML metacharacters, including < > " ' and =, should be replaced with the corresponding HTML entities (&lt; &gt; etc).

### Issue 4: Cross-origin resource sharing:

An HTML5 cross-origin resource sharing (CORS) policy controls whether and how content running on other domains can perform two-way interaction with the domain that publishes the policy. If another domain is allowed by the policy, then that domain can potentially attack users of the application.

**FIX** – Any inappropriate domains should be removed from the CORS policy and the appropriate domains should be defined.

**For Example -**

var http\_origin = SERVER['HTTP\_ORIGIN'];

if (http\_origin == "<http://www.domain1.com>" || http\_origin == "<http://www.domain2.com>" || http\_origin == "<http://www.domain3.com>")

{

header ("Access-Control-Allow-Origin: http\_origin");

}

**Reference** - <https://portswigger.net/blog/exploiting-cors-misconfigurations-for-bitcoins-and-bounties>

### Issue 5: Input returned in response(reflected):

Input being returned in application responses is not a vulnerability. However, it is a prerequisite for many client-side vulnerabilities, including cross-site scripting, open redirection, content spoofing, and response header injection. Additionally, some server-side vulnerabilities such as SQL injection are often easier to identify and exploit when input is returned in responses.

**Request-**/api/AccountProfile/LoadPaperLessAccountInfo4uiikk1yx2?userId=90953&accountId=90765&summary=true HTTP/1.1

**Response -** /api/AccountProfile/LoadPaperLessAccountInfo4uiikk1yx2?userId=90953&accountId=90765&summary=true HTTP/1.1

**FIX** – Input Sanitation or Input Validation at API and Output Encoding at Web Api and changing the Response Message.

This issue will be handled by **section 2.10 Web API input Parameter Sanitization and 2.12 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**

### Issue 6 – Invalid Input in JSON is returned as output message in Exception message

**For Example –**

**Request:** {"policyNumber":"5003370","userId":90953,"lobType":"COM","searchColumn":"PolicyNumod4s9a4eh6","

**Response:**

"ExceptionMessage":"Invalid column name 'PolicyNumod4s9a4eh6'."}

**FIX** – Input Sanitation or Input Validation at API and Output Encoding at Web Api and changing the Response Message.

This issue will be handled by **section 2.10 Web API input Parameter Sanitization and 2.12 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**

### Issue 7: Suspicious input transformation:

Suspicious input transformation arises when an application receives user input, transforms it in some way, and then performs further processing on the result. The types of transformations that can lead to problems include decoding common formats, such as UTF-8 and URL-encoding, or processing of escape sequences, such as backslash escaping.

**For Example** - POST /api/AccountProfile/dqeeuwv2fk%5c%5clhsh2jnsk7? In request **is interpreted** as api/AccountProfile/dqeeuwv2fk%5c%5clhsh2jnsk7? In response.

**FIX –** Change of Error Message and input request would be removed from output Exception Message.

This issue will be handled by **section 2.10 Web API input Parameter Sanitization and 2.12 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**

**Reference -** <https://portswigger.net/blog/backslash-powered-scanning-hunting-unknown-vulnerability-classes>

### Issue 8: SSL Cookie without a secure flag set:

If the secure flag is set on a cookie, then browsers will not submit the cookie in any requests that use an unencrypted HTTP connection, thereby preventing the cookie from being trivially intercepted by an attacker monitoring network traffic.

**FIX** – SET requireSSL="true" flag in <httpCookies requireSSL = “true” [..] />

**Reference -** <https://dotnet-security-guard.github.io/SG0008.htm>

### Issue 9: Cross-domain script include:

If you include a script from an external domain, then you are trusting that domain with the data and functionality of your application, and you are trusting the domain's own security to prevent an attacker from modifying the script to perform malicious actions within your application.

**FIX** – Copy the contents of the script and keep it in a min.js file within the angular app.

### Issue 10: Cookie without HttpOnly flag set:

If the HttpOnly attribute is set on a cookie, then the cookie's value cannot be read or set by client-side JavaScript. This measure makes certain client-side attacks, such as cross-site scripting, slightly harder to exploit by preventing them from trivially capturing the cookie's value via an injected script

**FIX** - HTTP-only cookies aren't accessible via JavaScript through the [**Document.cookie**](https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie) **property, the** [**XMLHttpRequest**](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) **API, or the** [Request](https://developer.mozilla.org/en-US/docs/Web/API/Request) API to mitigate attacks against cross-site scripting ([XSS](https://developer.mozilla.org/en-US/docs/Glossary/XSS)).

**Reference** - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>

### Issue 11: Frameable response (potential Clickjacking):

If a page fails to set an appropriate X-Frame-Options or Content-Security-Policy HTTP header, it might be possible for a page controlled by an attacker to load it within an iframe. This may enable a clickjacking attack

**FIX** - the application should return a response header with the name **X-Frame-Options** and the value **DENY** for preventing framing altogether.

**Reference -** <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>

### Issue 12: Email Address disclosed:

An information exposure is the intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information.

**FIX** – Converting GET to POST calls is handled by section **2.17 of Converting GET to POST calls** in **Penetration Testing Issues & Solutions.**

### Issue 13: Cacheable HTTPS response:

The application uses a cache to maintain a pool of objects, threads, connections, pages, or passwords to minimize the time it takes to access them or the resources to which they connect. If implemented improperly, these caches can allow access to unauthorized information or cause a denial of service vulnerability.

**FIX** - The web server should return the following HTTP headers in all responses containing sensitive content:

* Cache-control: no-store
* Pragma: no-cache

**Reference** - <https://docs.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-2.2>

### Issue 14: [HTML does not specify charset](https://portswigger.net/knowledgebase/issues/details/00800200_htmldoesnotspecifycharset) :

Content type: text/html is set in the headers from response therefore the Browser may not match the standard encoding, which could lead to CSS vulnerability.

**FIX** – Output encoding like **AntiXss**, etc should be done and the Content type header should be set to the same for Example **charset = AntiXss**

This issue will be handled by **section and 2.12 Output Encoding/Sanitization** in **Penetration Testing Issues & Solutions.**