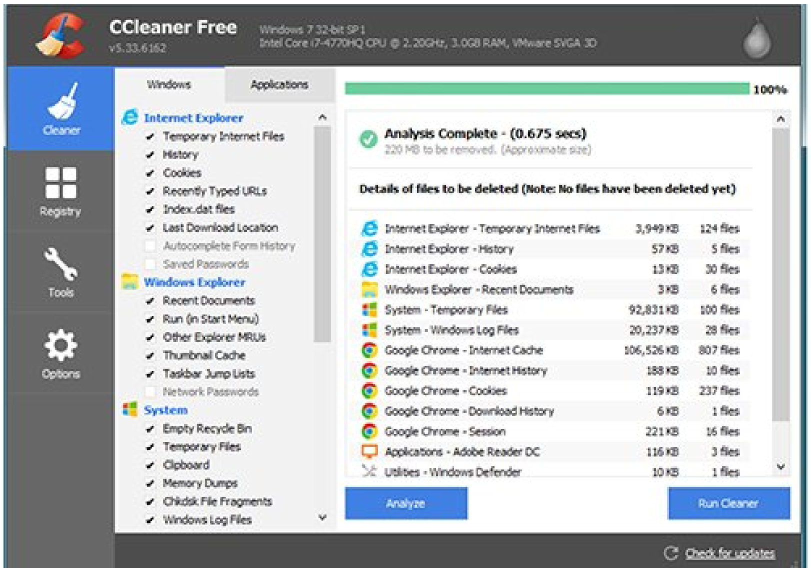
CCleaner Stage 2: In-Depth Analysis of the Payload

crowdstrike.com/blog/in-depth-analysis-of-the-ccleaner-backdoor-stage-2-dropper-and-its-payload/

Karan Sood November 10, 2017



Overview

Recently, CrowdStrike® analyzed the backdoor embedded in the legitimate PC cleaning utility CCleaner version 5.33, as reported in the blog post Protecting the Software Supply Chain: Deep Insights into the CCleaner Backdoor. This was an example of using an organization's supply chain infrastructure as an infection vector, a trend that has been on the rise in 2017 as discussed in another recent post, Software Supply Chain Attacks on the Rise, Undermining Customer Trust. In addition, CrowdStrike Falcon® Intelligence™ reported on the backdoor previously and discussed the possibility of the infrastructure being tied to a Chinese nexus.

Additionally, CrowdStrike Falcon Intelligence also discussed the technical details of the Stage 1 and Stage 2 backdoors with analysis showing that the original backdoor was the first stage in a multi-stage infection chain, meant to download a dropper (Stage 2) that was only deployed to specific targets. Stage 2 drops either a 32-bit or 64-bit binary, depending on the system architecture and is responsible for decrypting the actual payload embedded in a registry key. This payload attains the C2 address via a variety of steps, and downloads an unknown binary which is Stage 3.

This post provides an in-depth analysis of the Stage 2 dropper; the subsequent payload and the steps that are taken to calculate the C2 IP address in order to download the next stage binary.

Technical Analysis

Stage 2 Dropper

The following information describes the Stage 2 dropper that pertains to the CCleaner embedded malware:

Size: 175616
SHA256: DC9B5E8AA6EC86DB8AF0A7AA897CA61DB3E5F3D2E0942E319074DB1AACCFDC83
Compiled: Tue, Sep 12 2017, 8:44:58 — 32 Bit DLL

Once executed, the dropper calls **IsWow64Process** to determine if it's being run in a 64-bit environment. Depending on the result, it will drop a 32-bit or 64-bit binary on the system. The binary is embedded within the malware itself, and it is zlib compressed. The dropper will zlib inflate itself and drop onto the victim computer. The dropper also performs system checks by accessing the USER_SHARED_DATA of its own process and querying the **NtMajorVersion** value to determine if the system is running Windows XP. The output determines the location of the dropped binary.

If XP x86:
location is C:\Windows\System32\spool\prtprocs\\w32x86\\localspl.dll

If XP x64:
location is C:\Windows\System32\spool\x64\localspl.dll

If Windows 7 or higher:
location is C:\Windows\System32\TSMSISrv.dll

Dropped Binary Information

32-bit

Full path on victim machine (Windows 7 or higher): C:\Windows\System32\TSMSISrv.dll
Full path on victim machine (Windows XP): C:\Windows\system32\spool\prtprocs\w32x86\localspl.dll
Size: 173568
SHA256: 07FB252D2E853A9B1B32F30EDE411F2EFBB9F01E4A7782DB5EACF3F55CF34902
Compiled: Wed, Apr 22 2015, 18:20:39 — 32 Bit DLL
Version: 2, 0, 4, 23
File Description: VirtCDRDrv Module
Internal Name: VirtCDRDrv
Original Filename: VirtCDRDrv.dll
Product Name: VirtCDRDrv Module

64-bit

Full path on victim machine (Windows 7 or higher): C:\Windows\System32\TSMSISrv.dll
Full path on victim machine (Windows XP): C:\Windows\system32\spool\prtprocs\x64\localspl.dll
Size: 81408
SHA256: 128ACA58BE325174F0220BD7CA6030E4E206B4378796E82DA460055733BB6F4F
Compiled: Tue, Apr 19 2011, 0:09:20 — 64 Bit DLL
Version: 2.2.0.65
File Description: Symantec Extended File Attributes
Internal Name: SymEFA
Original Filename: EFACli64.dll
Product Name: EFA

It is important to note that both TSMSiSrv.dll and localspl.dll are actually the names of legitimate Microsoft Windows libraries. TSMSiSrv.dll's official description is "Windows Installer Coordinator for Remote Desktop Session Host Server" and it is loaded by the service "SessionEnv" that is the Remote Desktop Configuration service. According to MSDN, localspl.dll is a "Local Print Provider" and handles all print jobs directed to printers that are managed from the local server. This file is loaded by the service "Spooler" that is used for printing services.

After dropping the file, the dropper modifies its date/time stamp so that it matches that of C:\Windows\System32\msvcrt.dll.

Next, the dropper adds the following registry keys. (Note: This is specific to a 32-bit environment. Certain value such as file size will change if the malware is running in a 64-bit environment.)

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\001 → 2b 31 00 00. This is a hardcoded value. This is the size in bytes of the next registry key, which contains an obfuscated PE.
- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\002 → The dropper inserts a data blob in this key. The following explains the structure of the blob:

Position	Byte Size	Content
0	4	Result of GetTickCount() * rand()
4	4	Result of GetTickCount() * rand()
8	0x3123	Data blob

- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\003 → 21 00 00 00. Hardcoded value. Size in bytes of the next registry key
- HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\004 → Contains the following structure

Position	Byte Size	Content
0	4	Result of 0x5908EC83 ^ 0xF3289317 = 0xAA207F94
4	4	Result of 0x40518AB1 ^ 0xF3289317 = 0xB37919A6
8	4	Result of GetTickCount() * rand()
12	4	Result of GetTickCount() * rand()
16	4	Result of GetTickCount() * rand()
20	1	0x90

Leveraging Legitimate Services

The dropper leverages an existing Microsoft Windows service to load the malware. Once the registries have been added, the dropper calls a function to modify and restart an existing service. If executing in Windows 7 or higher, it calls **OpenServiceA** on the existing service, "SessionEnv" — a service for Remote Desktop Configuration — and changes its configuration by calling **ChangeServiceConfigA** with the following parameters:

- hService = Service Handle
- ServiceType = SERVICE_KERNEL_DRIVER|SERVICE_FILE_SYSTEM_DRIVER|SERVICE_ADAPTER|SERVICE_RECOGNIZER_DRIVER|SERVICE_WIN32_OWN_PROCESS|SERVICE_WIN32_SHARE_PROCESS|SERVICE_INTERACTIVE_PROCESS|FFFFFEC0
- StartType = SERVICE_AUTO_START
- ErrorControl = SERVICE_NO_CHANGE
- BinaryPathName = NULL
- LoadOrderGroup = NULL
- pTagId = NULL
- pDependencies = NULL
- ServiceStartName = NULL
- Password = NULL
- DisplayName = NULL

This ensures that the service will auto-start upon system reboot (i.e., a persistence mechanism). The dropper then restarts the service, which invokes the legitimate windows library "SessEnv.dll" located in C:\Windows\system32. It is important to note that SessEnv.dll is loaded in the process svchost.exe. Analysis shows that it attempts to load the legitimate library %SystemRoot%\system32\TSMSISrv.dll by calling **LoadLibrary** on it to call the functions *StartComponent*, *StopComponent*, *OnSessionChange*, and *Refresh* as shown in the image below:

```
  v2 = this;
  v3 = ExpandEnvironmentStringsW(lpSrc, &TSMSISrv_Library, 0x105u);
  if ( !v3 )
    goto LABEL_21;
  if ( v3 > 0x105 )
  {
    v4 = 1359;
    goto LABEL_5;
  }
  v4 = 0;
  v5 = LoadLibraryW(&TSMSISrv_Library);
  *v2 = v5;
  if ( !v5 )
  {
LABEL_21:
    v4 = GetLastError();
LABEL_5:
    if ( v4 )
    {
      sub_408062ED(v2);
      if ( v4 > 0 )
        v4 = v4 | 0x80070000;
    }
    return v4;
  }
  v7 = GetProcAddress(v5, "StartComponent");
  v2[1] = v7;
  if ( !v7 )
  {
    v4 = GetLastError();
    sub_40801251(4, "APPCMP_STARTCOMPONENTFN failed 0x%x", v4);
    goto LABEL_5;
  }
  v8 = GetProcAddress(*v2, "StopComponent");
  v2[2] = v8;
  if ( !v8 )
  {
    v4 = GetLastError();
    sub_40801251(4, "APPCMP_STOPCOMPONENTFN failed 0x%x", v4);
    goto LABEL_5;
  }
  v9 = GetProcAddress(*v2, "OnSessionChange");
  v2[4] = v9;
  if ( !v9 )
  {
    v4 = GetLastError();
    sub_40801251(4, "APPSRVCMP_ONSESSIONCHANGE failed 0x%x", v4);
    goto LABEL_5;
  }
  v10 = GetProcAddress(*v2, "Refresh");
  v2[3] = v10;
  if ( !v10 )
  {
    v11 = GetLastError();
    sub_40801251(4, "PFN_APPCMPREFRESH failed 0x%x", v11);
  }
  return v4;
```

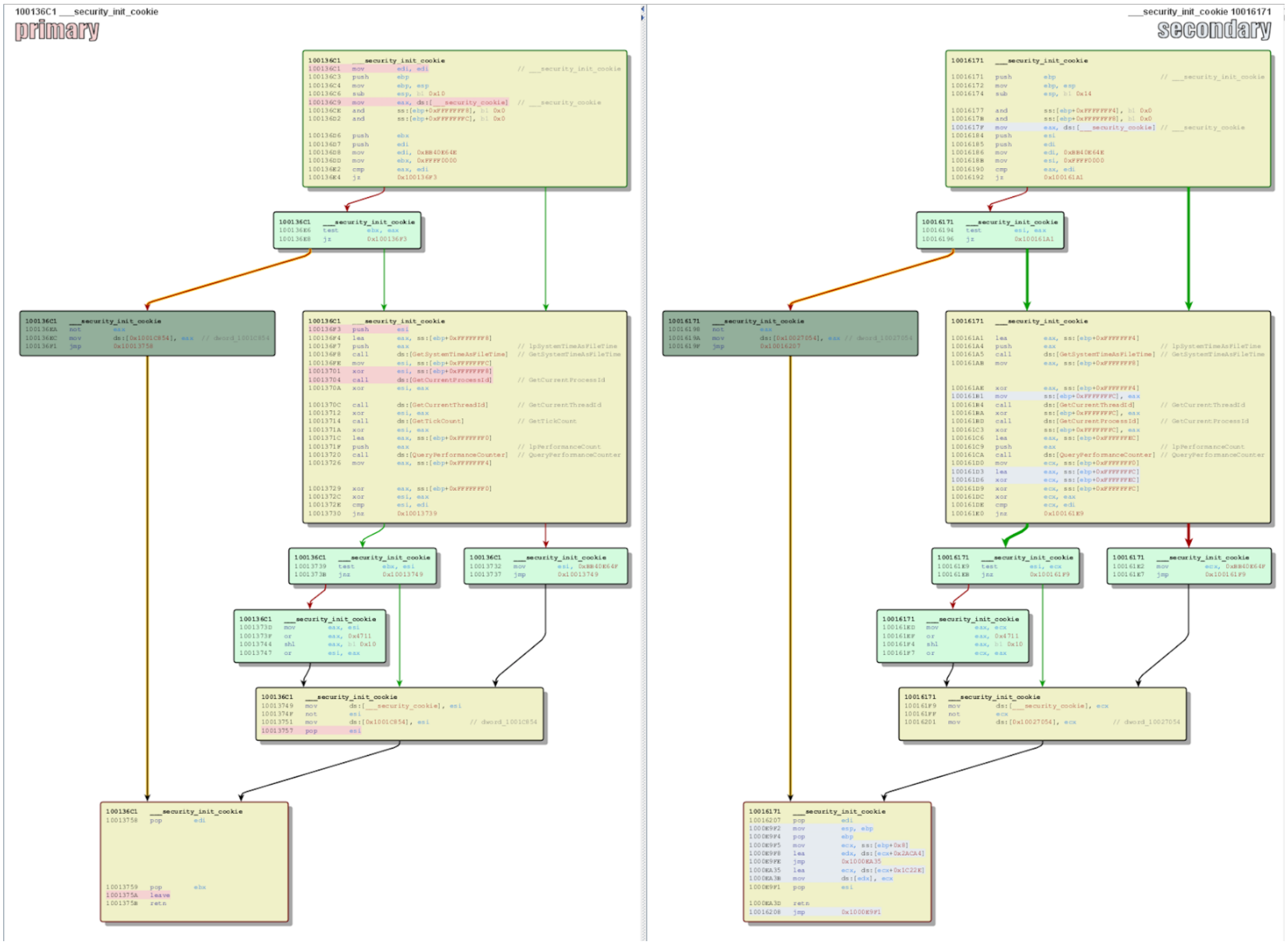
However, at this point in the execution, TSMSiSrv.dll is the name of the malicious binary created by the dropper; therefore, restarting the SessionEnv service loads the malware instead. Similarly, if the Windows version is XP, the malware takes the same steps on the service “Spooler.” Upon restart, the service invokes C:\Windows\system32\spoolsv.exe, which then attempts to load the Windows library localspl.dll that is now the actual malware.

File Modifications

As mentioned earlier, Stage 2 drops either a 32-bit or 64-bit binary on the victim system. Similar to the Stage 1 dropper, which was a modified version of the legitimate utility CCleaner, the 32-bit and 64-bit binaries are modified versions of VirtCDRDrv.dll (a module developed by Corel, Inc.) and EFACLi64.dll (a module developed by Symantec), respectively. However, it should be noted that unlike the trojanized version of CCleaner, these files are NOT signed.

VirtCDRDrv.dll 32-bit

Analysis shows that the **__security_init_cookie** function of the file has been modified. Normally this function is used as mitigation against buffer overflows; however, in this case, a few extra instructions have been added to the end of the function to initialize a global variable. The image below displays the difference between the clean VirtCDRDrv.dll (labeled primary on the left), and the trojanized one on the right.



The primary difference is the set of instructions at the end of the **__security_init_cookie** function. Specifically, these instructions insert a memory address in a global pointer; the memory address is the image base address + 0x1C22E, inserted in the global variable located at image base address + 0x2ACA4 as seen below.

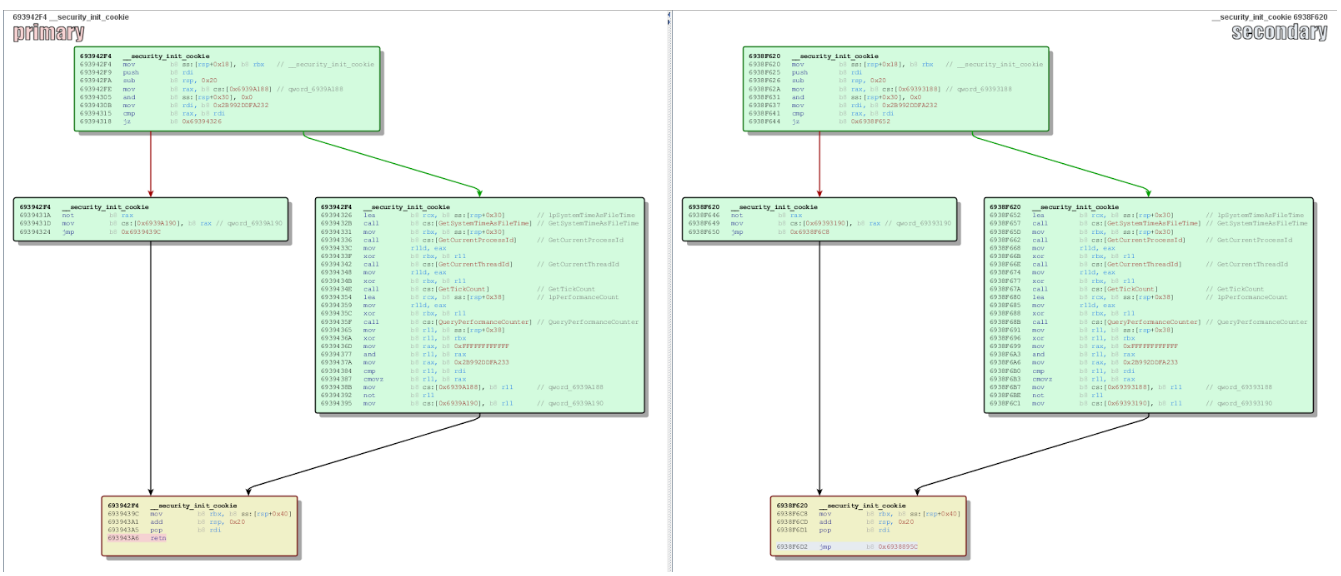
```
6545E9F1 ; START OF FUNCTION CHUNK FOR ___security_init_cookie
6545E9F1
6545E9F1 loc_6545E9F1:
6545E9F1 pop     esi
6545E9F2 mov     esp, ebp
6545E9F4 pop     ebp
6545E9F5 mov     ecx, [ebp+arg_0]
6545E9F8 lea     edx, [ecx+2ACA4h]
6545E9FE jmp     short loc_6545EA35
6545E9FE ; END OF FUNCTION CHUNK FOR ___security_init_cookie
```

```
6545EA35 ; START OF FUNCTION CHUNK FOR ___security_init_cookie
6545EA35
6545EA35 loc_6545EA35:
6545EA35 lea     ecx, [ecx+1C22Eh]
6545EA3B mov     [edx], ecx
6545EA3D retn
6545EA3D ; END OF FUNCTION CHUNK FOR ___security_init_cookie
```

Once the `__security_init_cookie` function is done, the `__DLLMainCRTStartup` function is called, which then makes the call to the function located at the memory address that was inserted into the global variable. This function is responsible for the core functionality of the dropped file. It should be noted that the malicious function is called prior to the entry point of the binary being reached.

EFACli64.dll 64-bit

The 32-bit binary and the 64-bit dropped file have been modified in the same manner. As seen in the image below, the only difference in the `__security_init_cookie` function between the legitimate utility (on the left) and the trojanized version is a jmp instruction at the end of the function.



This jmp instruction leads to the following instructions:

```
loc_6938895C:                           ; CODE XREF: .text:000000006938F6D2↓j
                                        ; DATA XREF: .pdata:0000000069394648↓o
                lea     rdx, [rsi+12891h]
                push    rdx
                pop     qword ptr [rsi+13CC0h]
                retn
```

This inserts the memory address located at image base address + 0x12891 in the global variable located at image base address +0x13CC0. Similar to the 32-bit binary, the malicious function at image base address + 0x12891 is called before the entry point is reached and is responsible for the core functionality of the malware.

Dropped Binary

Once loaded by the service, the binary reads the registries created earlier by the dropper. It allocates a block of memory and reads the data blob from HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\002. In addition, it also reads the first 2 DWORDS from HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WbemPerf\002, XORs them with the value 0x0xF3289317 and prepends them to the data blob. Together, this structure forms a shellcode appended by obfuscated data.

Shellcode

The shellcode utilizes the following scheme, reproduced in Python, to deobfuscate the embedded data:

```
indata = [0xb4, 0x28, 0x00, 0x00, 0xd8, 0x41, 0x00, 0x00, 0x5f, 0xe1, 0x60, 0x8b, 0x7d, 0x2a]  #snippet of obfuscated data
outdata = []
key = 0x5d4fc941

for i in range(0, len(indata)):
  keymod = ((key * 0x343FD) & 0xffffffff) + 0x269EC3
  key = keymod
  nkey = (keymod >> 0x10) & 0xff
  outdata.append(indata[i+8] ^ nkey)
```

It should be noted that the above is a modified version of the Windows function rand(). The decoded data is a set of instructions to unpack yet another shellcode and a DLL in memory. The resultant DLL is the main payload of Stage 2 and, similar to Stage 1, is missing the IMAGE_DOS_HEADER as a possible means to circumvent AV solutions that search for the MZ header in memory. The shellcode that is decoded alongside the DLL is responsible for resolving the needed APIs and calling the OEP (Original Entry Point) of the DLL in memory.

```
0000000: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000010: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000020: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000030: 0000 0000 0000 0000 0000 0000 d000 0000 ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000d0: 5045 0000 4c01 0400 f09b b759 0000 0000 PE..L......Y....
00000e0: 0000 0000 e000 0e21 0b01 0600 0026 0000 .......!.....&..
00000f0: 0016 0000 0000 0000 0010 0000 0010 0000 ................
0000100: 0040 0000 0000 0010 0010 0000 0002 0000 .@..............
0000110: 0400 0000 0000 0000 0400 0000 0000 0000 ................
0000120: 0070 0000 0004 0000 0000 0000 0200 0000 .p..............
0000130: 0000 1000 0010 0000 0000 1000 0010 0000 ................
0000140: 0000 0000 1000 0000 0000 0000 0000 0000 ................
0000150: 5c41 0000 b400 0000 0000 0000 0000 0000 \A..............
0000160: 0000 0000 0000 0000 0000 0000 0000 0000 ................
```

```
0000170: 0060 0000 2002 0000 0000 0000 0000 0000 .`.. ..........
0000180: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000190: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00001a0: 0000 0000 0000 0000 0040 0000 5c01 0000 .........@..\...
00001b0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00001c0: 0000 0000 0000 0000 2e74 6578 7400 0000 .........text…
00001d0: 9025 0000 0010 0000 0026 0000 0004 0000 .%.......&......
00001e0: 0000 0000 0000 0000 0000 0000 2000 0060 ............ ..`
00001f0: 2e72 6461 7461 0000 0608 0000 0040 0000 .rdata.......@..
0000200: 000a 0000 002a 0000 0000 0000 0000 0000 .....*..........
0000210: 0000 0000 4000 0040 2e64 6174 6100 0000 ....@..@.data…
0000220: 4406 0000 0050 0000 0006 0000 0034 0000 D....P.......4..
0000230: 0000 0000 0000 0000 0000 0000 4000 00c0 ............@...
0000240: 2e72 656c 6f63 0000 c202 0000 0060 0000 .reloc.......`..
0000250: 0004 0000 003a 0000 0000 0000 0000 0000 .....:..........
```

Payload

Upon being loaded in memory, the payload creates a thread that performs the core functionality of Stage 2. It creates an event named Global\KsecDDE and only commences execution if the event creation is successful. Analysis shows that there are multiple encoded URLs embedded within the payload, and they are deobfuscated using the scheme reproduced in Python below:

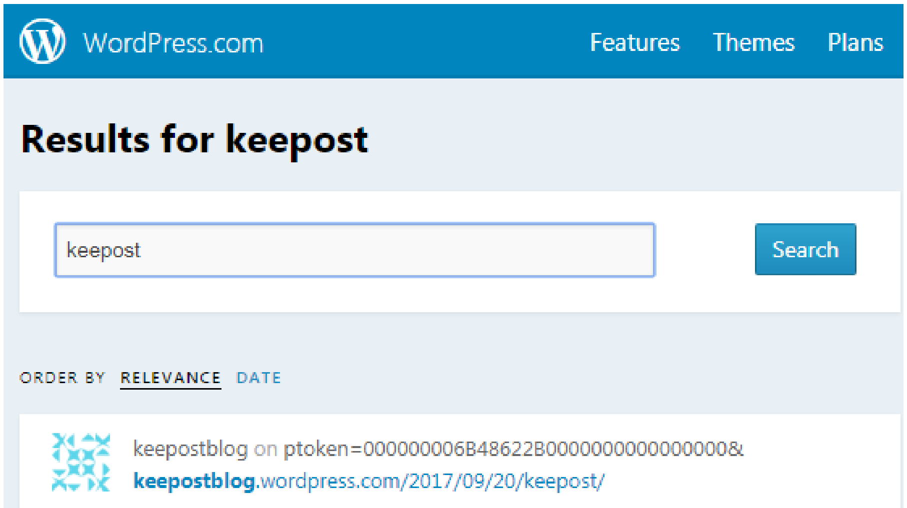
```
indata = [0xec, 0x87, 0x10, 0x23, 0xf5, 0x6d, 0xf7, 0x9a, 0x35, 0x1e, 0x82, 0xd6, 0xbc, 0x5f, 0x94] #indata = [0xe3, 0x96, 0x10, 0x7d, 0xe7,0x33, 0xb7, 0xd7, 0x3e, 0x12, 0xd8, 0xd0, 0xac, 0x49, 0xba, 0x13, 0xd0, 0x40, 0xc5, 0xd2, 0x68, 0xf6, 0x37, 0x3a, 0x1d, 0xbb, 0xd6, 0xad, 0x97, 0xcf, 0x88, 0xdc, 0xa3, 0x3a, 0x4d, 0x2e, 0xdb, 0x8d, 0xe3, 0xf8, 0xf4, 0x20, 0x38, 0x7c, 0xc3, 0xe5, 0x69, 0xfb, 0x40, 0x40, 0xb5, 0x5e, 0x7a, 0xa5, 0x40, 0x7d, 0x4a, 0x6e, 0x85, 0x76, 0x9a, 0xf0]

#indata = [0xe3, 0x96, 0x10, 0x7d, 0xe7, 0x33, 0xb7, 0xd7, 0x3c, 0x15, 0x82, 0xcb, 0xbc, 0x4a, 0xe6, 0x13, 0xd7, 0x3, 0x9d, 0xce, 0x7f, 0xf3, 0x35, 0x2b, 0x10, 0xf7, 0xd4, 0xbe, 0x9e, 0xcf, 0x8c, 0x9d, 0xf0, 0x3c, 0x4d, 0x6b, 0x92, 0x9b, 0xe1, 0xfa, 0xa8, 0x1b, 0x22, 0x7a, 0x9a, 0xe7, 0x72, 0xE5, 0x51, 0x43, 0xfd, 0x0c, 0x2c, 0x94, 0x72]

keyinit = 0xd35125
outdata = []

for i in range(0, len(indata)-1):
   keymod = (0x17879ef * keyinit) & 0xffffffff
   keybyte = keymod & 0xff
   keyinit = keymod >> 8
   outdata.append(indata[i] ^ keybyte)
print ''.join(map(chr, outdata))
```

Following are the decoded URLs:

- get.adoble[.]net
- https://en.search.wordpress[.]com/?src=organic&q=keepost
- https://github[.]com/search?q=joinlur&type=Uses&utf8=%E2%9C%93

Before connecting to any of the above, the payload first attempts to connect to https://www.microsoft.com. If that fails, the payload then attempts to connect to http://update.microsoft.com. This is to perform a connectivity test to ensure that the victim computer is connected to the internet. The payload also ensures that the received data contains the string “Microsoft” or “Internet Explorer”; apart from a connectivity test, this could also be seen as an antisandbox technique. If the test passes, a global variable Connectivity_Flag is set to 1, after which the malware attempts to connect to either the WordPress or the Github URL. At the time of analysis, the Github URL was not available.

The following is the data returned by the WordPress URL:



If the connection is a success, the malware parses the retrieved data for the string “ptoken=”. As the above image shows, the ptoken value is “000000006B48622B0000000000000000&”. The malware converts the string value to a long integer value in base 16 by calling **strtoul**. The result is the DWORD 0x6B48622B, which is then XORd with the value 0x31415926 (value of Pi) to get the value 0x5A093B0D, which translates to the IP address 13.59.9.90.

If the payload fails to connect to both the Github and WordPress URLs, it will attempt to connect to get.adoble[.]com to calculate an IP address. It gets the *hostent* structure by calling **gethostbyname** on the domain, which then gives it a NULL terminated list of IP addresses associated with the domain. The first 2 IP addresses will then be used to calculate the IP address using the algorithm reproduced in Python below:

```
import struct
import socket a1 = 0x659C2A88  # Addresses are returned in network byte order
a2 = 0x6B442ABF  # These are just for example purposes def mod_record(rr):
   rr1 = (((rr & 0xff000000) / 0x1000000) ^ (rr & 0xff)) * 0x1000000
   rr2 = (((rr & 0xff0000) / 0x10000) ^ ((rr & 0xff00) / 0x100)) * 0x10000
   rr3 = rr & 0xff00
   rr4 = rr & 0xff
   return (rr1 | rr2 | rr3 | rr4) newa1 = mod_record(a1)
newa2 = mod_record(a2)
newIP =  (newa2 & 0xffff0000) | (newa1 >> 0x10)  # newIP = 0xD46EEDB6
print socket.inet_ntoa(struct.pack("<L", newIP))  # Output is 182.237.110.212
```

Next, the malware calculates a checksum of the victim computer name using the following algorithm:

```
import struct compname = "WIN-CHB5K9B5QOM"  #example of computer name
checksum = 0
hss = compname.encode('hex')
indata = []
i = 0
def swap(d):
   return struct.unpack("<I", struct.pack(">I", d))[0]
while 1:
   idata = hss[i:i+8]
   if len(idata) < 8:
```

```
        numz = 8 - len(idata)
        strz = '0' * numz
        idata = idata + strz
    indata.append(idata)
    i += 8
    if i >= len(hss):
        break
for i in indata:
    i_ = int(i, 16)
    i_ = swap(i_)
    i_ = (i_ * 0x5E1F1AE) & 0xffffffff
    checksum = (checksum +  i_) & 0xffffffff print hex(checksum)
```

This checksum value is then added to the volume serial number of the victim computer. The LOWORD of the resultant DWORD is then added to the value 0x2DC6C0 to get a unique value. Next, the malware creates a socket and sets up the following packet to send to the newly calculated IP via a DNS query:

Type	Value
Transaction ID	LOWORD of the unique value calculated earlier using the checksum and volume serial number.
Flags	0x100. Denotes that the message is a query.
Questions	0x1. Number of queries.
Query	ds.download.windowsupdate.com
Type	0x1. Type A (Host Address)
Class	0x0001. IN (Internet)

Following is the actual UDP stream seen during analysis:

```
0000000: d47a 0100 0001 0000 0000 0000 0264 7308 .z...........ds.
0000010: 646f 776e 6c6f 6164 0d77 696e 646f 7773 download.windows
0000020: 7570 6461 7465 0363 6f6d 0000 0100 0100 update.com.....
```

At the time of analysis, the IP address was not available; however, analysis shows that the malware performs the following checks on the received response from the IP to ensure its authenticity.

- Transaction ID is 0xD47A (same as the query)
- Total Answer RRs field is 4. Number of entries in the resource record list.
- The 38th word is the value 0x06A4
- The 48th word is the value 0x0A8C

Stage 3

The malware takes values from the response stream at various positions, and calculates the Stage 3 C2 in the following manner:

- First octet → 59th byte ^ 62nd byte
- Second octet → 76th byte ^ 78th byte
- Third octet → 93rd byte ^ 94th byte
- Fourth octet → 110th byte

Once the 3rd stage C2 has been calculated, the malware calls out to it expecting to receive an obfuscated blob. The first DWORD of the blob is the CRC32 hash of the decoded blob. The blob is decoded using the same scheme that is used to decode the URLs, and the CRC32 hash of the decoded data is compared with the first DWORD of the received data to ensure its integrity. Analysis shows that the data is supposed to be yet another DLL, which is then loaded in memory and executed.

Recommendations

CrowdStrike will notify you of any additional activity through the Falcon Intelligence™ detections. CrowdStrike recommends blocking the IP and URLs mentioned in this blog post and the previous one to prevent any communication to the server. In addition, CrowdStrike recommends only using the latest version of the Avast CCleaner software to ensure that the infection does not occur.

Learn more about the CrowdStrike Falcon Intelligence offerings, and read the white paper, "Threat Intelligence, Cybersecurity's Best Kept Secret."