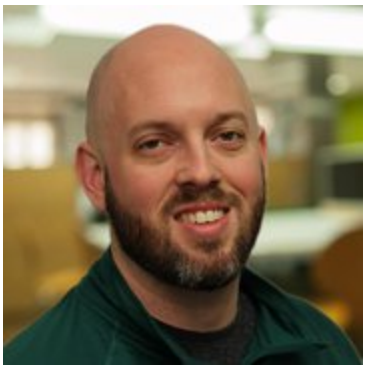
API Hashing Tool, Imagine That

insights.sei.cmu.edu/cert/2019/03/api-hashing-tool-imagine-that.html



Kyle O'Meara and CERT Insider Threat Center

March 25, 2019

In the fall of 2018, the CERT Coordination Center (CERT/CC) Reverse Engineering (RE) Team received a tip from a trusted source about a YARA rule that triggered an alert in VirusTotal. This YARA rule was found in the Department of Homeland Security (DHS) Alert TA17-293A, which describes nation state threat activity associated with Russian activity. I believed this information warranted further analysis.

The YARA rule, shown in Figure 1, is allegedly associated with the Energetic Bear group. The Energetic Bear group, named by security firm CrowdStrike, conducts global intelligence operations, primarily against the energy sector. It has been in operation since 2012. (For more information, see CrowdStrike Global Threat Report: 2013 Year in Review.) This group has also been referred to as Dragonfly (Symantec), Crouching Yeti (Kaspersky), Group 24 (Cisco), and Iron Liberty (SecureWorks), among others. (For more information, see APT Groups and Operations.)

```
rule APT_malware_2
{
meta:
      description = "rule detects malware"
      author = "other"
strings:
      $api_hash = { 8A 08 84 C9 74 0D 80 C9 60 01 CB C1 E3 01 03 45 10 EB ED }
      $http_push = "X-mode: push" nocase
      $http_pop = "X-mode: pop" nocase
condition:
      any of them
}
```

Figure 1: YARA Rule for DHS Alert TA17-293A

Unfortunately, upon reviewing numerous public threat reports from the above vendors, I could not find further information tying this YARA rule or associated exemplars to the Energetic Bear group, but I still believed that the activity warranted further investigation and analysis.

Methodology

I used the following methodology for this analysis:

- analyzed the YARA rule and initial exemplar
  - analyzed exemplar with IDA
  - researched and applied API hashing module routine findings to exemplar
  - mapped research findings and analysis to exemplar with IDA
- created a tightly scoped YARA rule to discover new exemplars
  - created API hash YARA rules to discover more exemplars
  - analyzed new exemplars with refined YARA rule
  - created a tightly scoped YARA rule
- discovered API hashes found in new exemplars
- questioned attribution
- identified future work
- reported results

Analyzed the YARA Rule and Initial Exemplar

I was interested in understanding the string variables found in the YARA rule shown in Figure 1. Specifically, it was not immediately clear what the *$api_hash* variable represented, whereas the variables *$http_post* and *$http_push* appeared to be associated with Hypertext Transfer Protocol (HTTP) header fields. I focused my analysis on the *$api_hash* variable.

Analyzed Exemplar with IDA

After cursory analysis of the initial exemplar (SHA256: 1b17ce735512f3104557afe3becacd05ac802b2af79dab5bb1a7ac8d10dccffd), I determined that the *$api_hash* variable was alerting on the routine (highlighted in green in Figure 2).

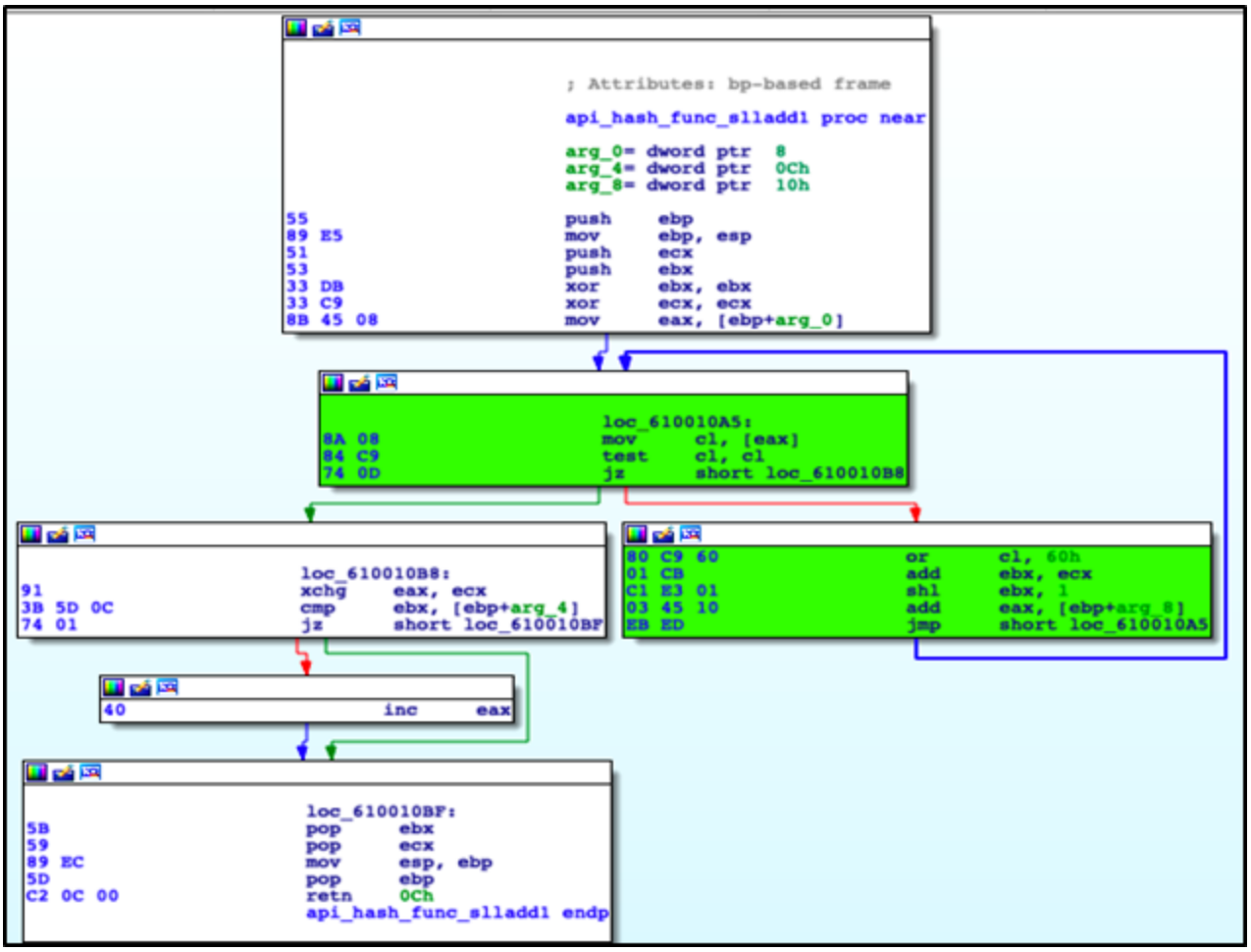


Figure 2: $api_hash Variable Found in Initial Exemplar

Researched and Applied API Hashing Module Routine Findings to Exemplar

The key points to highlight in Figure 2 are the *or* of 0x60, *shift logical left (shl) by 1*, followed by an *add,* and *jump*. Based on this information coupled with the variable name *$api_hash*, I was able to determine that this was a Windows application programming interface (API) hashing routine.

I wanted to find further information on any API hashing techniques. Through open source intelligence (OSINT) gathering, I discovered the FireEye Flare IDA Pro utilities Github page that mentioned a plug-in called *Shellcode Hashes* and an associated blog post from 2012 titled "Using Precalculated String Hashes when Reverse Engineering Shellcode," which further discussed API hashing. (For more information, see FireEye Flare IDA Pro utilities Github and Using Precalculated String Hashes when Reverse Engineering Shellcode.) After I examined the FireEye Flare IDA plug-in script further, I found it contained 23 API hashing modules. I identified an API hashing module, shown in Figure 3, that was very similar to the routine found in the exemplar shown in Figure 2. This API hashing module is a function that contains a *for loop*, which contains an *or* of 0x60 followed by *add* and a *shift left by 1.*

```
def sll1AddHash32(inString,fName):
    if inString is None:
        return 0
    val = 0
    for i in inString:
        b = ord(i)
        b = 0xff & (b | 0x60)
        val = val + b
        val = val << 1
        val = 0xffffffff & val
return val
```

Figure 3: sll1AddHash32 Function from FireEye Flare IDA Plug-In

The CERT/CC has an API hashing tool that creates a set of YARA signatures of API hashes for a given set of dynamic link library (DLL) files. This API hashing tool contained 22 API hashing modules. One of these modules matched the routine from the exemplar shown in Figure 2 and the FireEye API hashing module shown in Figure 3. I called this API hashing module *sll1Add*. I used the CERT/CC API hashing tool and a clean set of DLL files (see Table 1), to create a set of YARA rules for the *sll1Add* routine. After running the entire set of YARA rules against the exemplar, I received an alert for kernel32.dll API hashes shown in Figure 4.

```
Function        Byte Value (big endian)
LoadLibraryA    86 57 0D 00
VirtualAlloc    42 31 0E 00
VirtualProtect  3C D1 38 00
```

Figure 4: API Hashes from kernel32.dll for sll1Add Routine

Mapped Research Findings and Analysis to Exemplar with IDA

I used another CERT/CC tool called UberFLIRT. UberFLIRT calculates and stores position independent code (PIC) hashes of arbitrary functions, easily shares information via a central database, and allows for fewer false positives than IDA's Fast Library Identification and Recognition Technology (FLIRT). I labeled the function shown in Figure 2 in IDA as *api_hash_func_slladd1* and saved it to the Uberflirt database to facilitate future analysis of similar exemplars.

Examining the entry point of the exemplar, I found two values that are pushed onto the stack and passed as parameters to a function. These two values are 0x0038D13C and 0x000D4E88. The value 0x0038D13C is the hash of *VirtualProtect* shown in Figure 4. The other value, 0x000D4E88, is discussed below.

Examining this function, where the API hashes were passed as parameters, I determined that this exemplar uses manual symbol loading techniques, which are very similar to that of shellcode, to interact with the system through APIs.

This process reads the Thread Environment Block (TEB) to find the pointer to Process Environment Block (PEB) structure. The PEB structure is then parsed to find the DllBase of kernel32.dll. This exemplar also checks to ensure that it has the correct kernel32.dll by using 0x000D4E88 hash value to check for the kernel32.dll base name to the kernel32.dll that was found via manual symbol loading. The function then continues to parse the portable executable (PE) export data and passes the virtual protect hash (0x0038D13C) to the hashing algorithm. The same is done for the remaining hashes. This process is shown in Figure 5 with my added comments. I labeled the function from Figure 5 *manual_symbol_resolution* and saved it to the UberFLIRT database to aid in future analysis of similar exemplars.

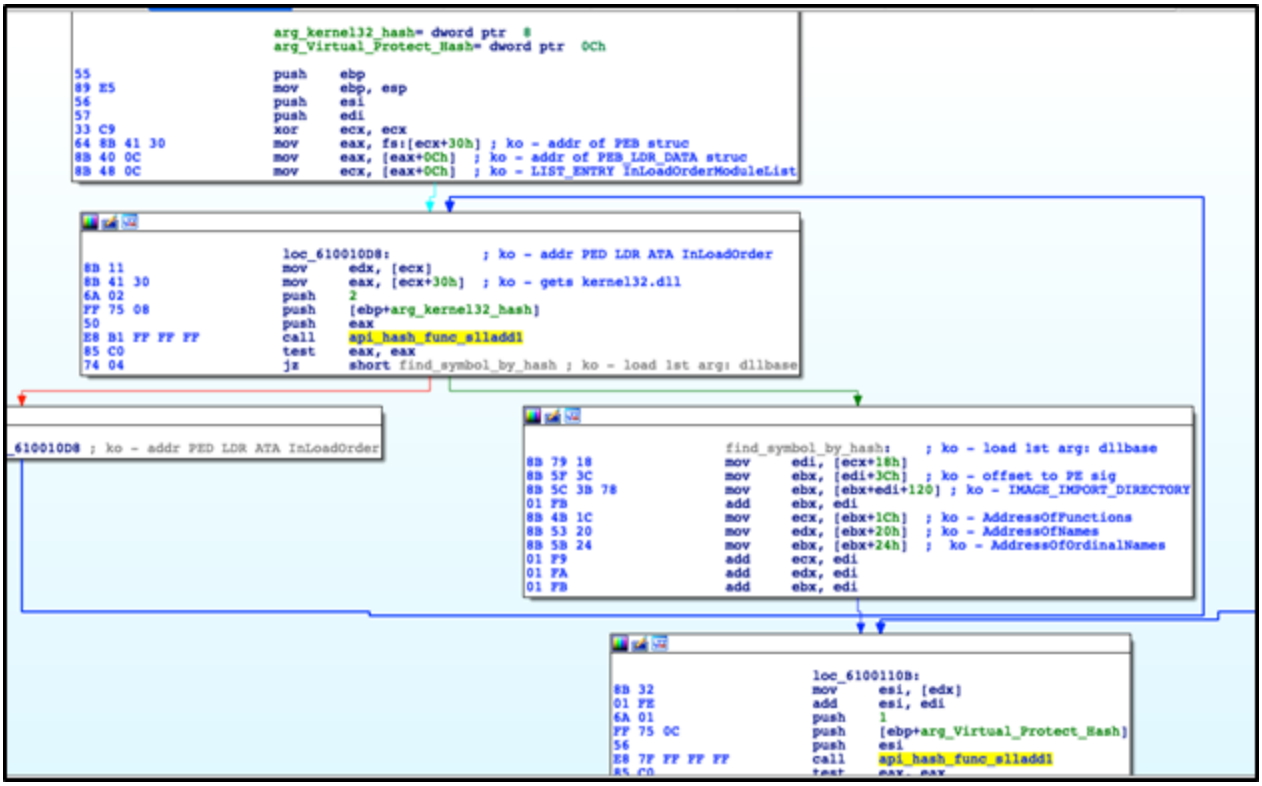


Figure 5: Manual Symbol Loading with Comments of Exemplar

Now that I understood the initial exemplar, I proceeded to find similar exemplars.

Created a Tightly Scoped YARA Rule to Discover New Exemplars

I used the following process to find additional exemplars:

- created API hash YARA rule to discover more exemplars
- analyzed new exemplars with refined YARA rule
- created a tightly scoped YARA rule

Created API Hash YARA Rule to Discover More Exemplars

The YARA rule, shown in Figure 6, represents the *push* of the API hash value (0x0038D13C), the *push* of the DLL base name hash value (0x000D4E88), and the *call* to *manual_symbol_resolution.*

I used the YARA rule, shown in Figure 6, to discover an additional 36 potential exemplars. To discover these files, I used the CERT/CC's large archive of potentially malicious software artifacts called the Massive Analysis and Storage System (MASS). The MASS is a distributed system designed to download, process, analyze, and index terabytes of potentially malicious files on a daily basis.

```
rule api_hashes_2_call
{
   strings:
     (2019-02-22)
     $api_hashes_2_call = { 68 3C D1 38 00 68 88 4E 0D 00 E8 ?? ?? ?? ?? }
   condition:
     uint16(0) == 0x5a4d and $api_hashes_2_call
}
```

Figure 6: API Hashes

Analyzed New Exemplars with Refined YARA Rule

I refined the YARA rule from Figure 1, as shown in Figure 7, to further examine the potential 36 exemplars for the existence of the API hashing routine. I assumed that if additional exemplars contained the string variable from the YARA rule shown in Figure 6, then these exemplars should have the API hashing routine from the YARA rule shown in Figure 7.

```
rule energetic_bear_api_hashing_tool {
meta:

      description = "Energetic Bear - API Hashing"
      assoc_report = "DHS Report TA17-293A"
      author = "CERT RE Team"
      version = "1"

strings:
      $api_hash_func = { 8A 08 84 C9 74 0D 80 C9 60 01 CB C1 E3 01 03 45 10 EB ED }
      $http_push = "X-mode: push" nocase
      $http_pop = "X-mode: pop" nocase

condition:
  $api_hash_func and (uint16(0) == 0x5a4d or $http_push or $http_pop)
```

Figure 7: Refined YARA Rule

Upon further analysis, I realized that some of the new exemplars did not alert with the YARA rule shown in Figure 7. I analyzed this subset of exemplars and discovered two slight variations in the API hashing routine. The first was an addition of one extra byte, while the second dealt with 64-bit files.

Created a Tightly Scoped YARA Rule

I refined the YARA rule further to incorporate these two additional variations shown in Figure 8.

```
rule energetic_bear_api_hashing_tool {
meta:
      description = "Energetic Bear API Hashing Tool"
      assoc_report = "DHS Report TA17-293A"
      author = "CERT RE Team"
      version = "2"

strings:
          $api_hash_func_v1 = { 8A 08 84 C9 74 ?? 80 C9 60 01 CB C1 E3 01 03 45 10 EB
ED }
          $api_hash_func_v2 = { 8A 08 84 C9 74 ?? 80 C9 60 01 CB C1 E3 01 03 44 24 14
EB EC }
      $api_hash_func_x64 = { 8A 08 84 C9 74 ?? 80 C9 60 48 01 CB 48 C1 E3 01 48 03 45
20 EB EA }

      $http_push = "X-mode: push" nocase
      $http_pop = "X-mode: pop" nocase

condition:
      $api_hash_func_v1 or $api_hash_func_v2 or $api_hash_func_x64 and (uint16(0) ==
0x5a4d or $http_push or $http_pop)
}
```

Figure 8: Tightly Scoped YARA Rule with All Variations

This YARA rule, shown in Figure 8, could be refined further by combining the API hash routines into one string variable. However, when identifying new exemplars, I wanted to know which API hashing function was found in the exemplar.

Discovered API Hashes Found in New Exemplars

I turned my attention to identifying the *sll1Add* routine API hash values found in all of the 37 exemplars.

All exemplars had the *sll1Add* routine API hash values for functions from kernel32.dll. These are shown in Figure 9.

```
Function                Byte Value (big endian)
CreateThread            14 F3 0C 00
ExitProcess             6A BC 06 00
GetSystemDirectoryA     E6 B2 9B 06
LoadLibraryA            86 57 0D 00
VirtualAlloc            42 31 0E 00
VirtualFree             8E 18 07 00
VirtualProtect          3C D1 38 00
```

Figure 9: sll1Add Module API Hash Values from kernel32.dll

Most of the exemplars had the *sll1Add* routine API hash values for functions from ws2_32.dll, as shown in Figure 10.

```
Function                Byte Value (big endian)
WSAGetLastError         70 71 71 00
WSAStartup              14 93 03 00
connect                 7C 67 00 00
recv                    C0 0C 00 00
send                    D8 0C 00 00
socket                  A4 36 00 00
```

Figure 10: sll1Add Module API Hash Values from ws2_32.dll

There were a few outliers that had the *sll1Add* routine API hash values for functions from wininet.dll. These are shown in Figure 11.

```
Function                Byte Value (big endian)
HttpAddRequestHeadersA  AE 57 5E 36
HttpEndRequestA         DA 03 6D 00
HttpOpenRequestA        DA BB DA 00
HttpQueryInfoA          EE C3 36 00
HttpSendRequestA        DA B3 DA 00
InternetCloseHandle     1A DE BB 06
InternetConnectA        BA 7B D7 00
InternetOpenA           02 F0 1A 00
InternetOpenUrlA        52 87 D7 00
InternetReadFile        62 81 D7 00
InternetSetOptionA      82 28 5E 03
```

Figure 11: sll1Add Module API Hash Values from wininet.dll

The API hashes shown in Figures 10 and 11 indicate that these exemplars have potential network communications. I analyzed these exemplars to identify the network-based indicators of compromise (IOC). The use of two different DLLs for network communications points to the existence of at least two different versions of the API hashing tool.

I identified 29 unique IP address, including private IP space and port pairings, shown in Table 3, from 33 of 37 exemplars.

The other 4 of 37 exemplars had a structure outbound POST request. For 2 of these 4, I captured the requests in a packet capture (pcap) using FakeNet. I had to infer the outbound POST request structure from strings for the remaining 2 exemplars. These POST requests are shown in Figures 12 and 13. The strings of the POST request are shown in Figures 14 and 15.

```
POST / HTTP/1.1
X-mode: pop
X-id: 0x00000000,0x5547a48a
User-Agent: Mozilla
Host: 187.234.55.76:8080
Content-Length: 0
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 12: Captured Network Communication from Exemplar (SHA256: 2595c306f266d45d2ed7658d3aba9855f2b08682b771ca4dc0e4a47cb9015b64)

```
POST / HTTP/1.1
X-mode: pop
X-id: 0x00000000,0x5bc509c7
User-Agent: Mozilla
Host: 4.34.48.68:18443
Content-Length: 0
Connection: Keep-Alive
Cache-Control: no-cache
```

Figure 13: Captured Network Communication from Exemplar (SHA256: 1b17ce735512f3104557afe3becacd05ac802b2af79dab5bb1a7ac8d10dccffd)

```
X-mode: push\r\nX-type: more\r\nX-id: 0x00000000,0x523fe61c\r\n
X-mode: push\r\nX-type: last\r\nX-id: 0x00000000,0x523fe61c\r\n
X-mode: pop\r\n\r\nX-id: 0x00000000,0x523fe61c\r\n
Mozilla
POST
```

Figure 14: Network Communication Strings from Exemplar (SHA256: 34f567b1661dacacbba0a7b8c9077c50554adb72185c945656accb9c4460119a)

```
X-mode: push\r\nX-type: more\r\nX-id: 0x00000000,0x5bc509c7\r\n
POST
Mozilla
X-mode: pop\r\n\r\nX-id: 0x00000000,0x5bc509c7\r\n
X-mode: push\r\nX-type: last\r\nX-id: 0x00000000,0x5bc509c7\r\n
```

Figure 15: Network Communication Strings from Exemplar (SHA256: 9676bacb77e91d972c31b758f597f7a5e111c7a674bbf14c59ae06dd721d529d)

This information can be turned into signatures for network intrusion detection systems, such as Snort or Suricata.

Questioned Attribution

I attempted to identify other public reporting or research related to this Energetic Bear group API hashing tool. I did not identify any public reporting or research.

Because of the link to the Energetic Bear group, I thought the exemplars could be a remote access Trojan (RAT), such as Havex, which is also attributed to this particular group. I discovered research by Veronica Valeros on A Study of RATs: Third Timeline Iteration. I contacted her directly and ask if she recalled any of the RATs she researched using an API hashing technique. She could not recall, but mentioned that it could have been missed because she was not explicitly looking for this technique. I used her research to attempt to identify RATs that use this API hashing technique. I was unable to identify any publically reported RAT using this technique.

Identified Future Work

This brings me to a couple outstanding questions:

- Why is this API hashing tool linked to the Energetic Bear group?
- Who actually wrote the YARA rule from Figure 1 found in DHS Alert TA17-293A?
- Can the author of the YARA rule provide more insight to this problem?

I hope by publicly discussing this analysis that I can encourage information sharing and allow us, as a community, to engage in more detailed threat reporting.

Lastly, I have reached out to the MITRE ATT&CKTM team to ask for an additional technique, API hashing, to be added to its framework. During my analysis, I could not find this explicit technique listed in the framework.

Reported Results

I expanded the corpus of information from afore mentioned trusted partner regarding DHS Alert TA17-293A. This information includes

- a more concise YARA rule, shown in Figure 8
- additional exemplars shown in Table 2
- network IOCs shown in Table 3 (of which at least 2 different versions exist)

If the attribution and my research are correct, this may be the *first* publicly documented report of an API hashing technique being used by a nation state actor.

Updates

(May 3, 2019)
I've worked with the MITRE Malware Attribute Enumeration and Characterization (MAECâ„¢) team to have API Hashing added to the Malware Behavior Catalog Matrix. You can find the API Hashing listed as a Method under Anti-Static Analysis--Executable Code Obfuscation.

(March 27, 2019)
The power of open source sharing has been positive. It was brought to my attention (thanks to Matt Brooks from Citizen Lab) that this API hashing tool is related to Trojan.Heriplor from Symantec's Dragonfly: Westion energy sector targeted by sophisticated attack group report. The hash in Symantec's report is, in fact, one of the exemplars found shown in the appendix. Symantec provided this hash in the form of a picture, and I must have fat-fingered the hash when transcribing it. However, this specific API hashing technique isn't mentioned in their report.

This corroboration does help to answer my own question from the Identified Future Work section. Symantec's Trojan.Heriplor analysis attributes my analysis of this API hashing tool to Energetic Bear. More importantly, this linkage also shows that this tool is still actively used.

Appendix

List of Clean DLL Files Used to Identify API Hashes

Table 1: List of DLL Files

```
advapi32.dll
advpack.dll
avicap32.dll
comctl32.dll
comdlg32.dll
gdi32.dll
imagehlp.dll
iertutil.dll
IPHLPAPI.DLL
kernel32.dll
mpr.dll
msvcrt.dll
netapi32.dll
ntdll.dll
ntoskrnl.exe
ole32.dll
psapi.dll
oleaut32.dll
secur32.dll
shell32.dll
shlwapi.dll
srvsvc.dll
rlmon.dll
user32.dll
win32k.sys
winhttp.dll
wininet.dll
winmm.dll
wship6.dll
ws2_32.dll
```

Hashes of Exemplars

Table 2: List of Exemplars SHA256 Hash Values

```
2595c306f266d45d2ed7658d3aba9855f2b08682b771ca4dc0e4a47cb9015b64
9676bacb77e91d972c31b758f597f7a5e111c7a674bbf14c59ae06dd721d529d
1b17ce735512f3104557afe3becacd05ac802b2af79dab5bb1a7ac8d10dccffd
b1ef39b2d0e26a23f59554ba4aecca8f266d6a69de1225d6b5c46828e06e9903
759445c7f68b55e90f23111c0e85d0da5456f2437e2360f4e808638d4c9020f7
8893b621b0bbbe8d29bd2cee70b5318b81deaadb42dda3c1a1a970fe0b54e781
1169853e30afd4fd2fbf34ef2c3028da6a81e9b6e1bd3bde077f13ad41e210de
5a9b65f9ed0758d11f74af9195fa3263a93bd1127e389c9ad920d585aee603ce
16e0188364ffcf738130436d08083306a52fe16bc45c2fbb3069d30a0de4995c
8c5bbff5875079cc553a296eae0f8b516eb03410c5a51fa9ffb0b98d13e3e489
aeece7de386715cd187c23d8e6aef165c107183ca15ebec797c9c2c7f9b2782d
0851abdd2b96779a43bd6144b3de4a7274f70c4e72ed96c113237ddcc669d3d0
34f567b1661dacacbba0a7b8c9077c50554adb72185c945656accb9c4460119a
16a3ad20b7c702808d29afacd1bcac626963d7d7b21ba7d0ea4d85403331dab0
b051a5997267a5d7fa8316005124f3506574807ab2b25b037086e2e971564291
12cc855139caed5256901d773c72a618e1cce730f7a47af91aa32541077b96a9
834e4560cec6deae11c378c47b4be806d4048868ee5315ba080fe11650a7c74d
58903fd6f2ecf56d0f90295d32c1ae29fe5250d3cf643ba2982257860b3f01a8
a9982010eeff3d2dd90757f10298fb511aeb538def94236d73b45ae92b416a50
addf1024d36d73bb22d2cfc8db78f118883de9e26092dde3f56605cf2436ef12
41395f0e1efc967fbd3ef2559f6307dd4dc331b1dd39ff9b0e239aeb83906555
689e8995e41a6484b1ff47edf0c0d2e9b660f1965d83836eb94610c6c4110066
064b5ff7890808b9c5ccdc2968fb7401c807a5a53132e6b1359ac46b2bec3c85
c5d75c25ac791ccac327f5f68340a8cfd7f5640dd2614aef7c50af4f6f330d02
c329462d39cdf794af1e4b5f2137a9141d9035932a4d64a99e3ce576219f337b
5d1c2e1be2360d9d58f87bc8131cbb1079813f08f93e7b5d627dd53758372e0d
c51f70707baa65cb88a97f0ffb5a3664d9c62a37e61909bd7710ecd6a2de59e7
1fde10b6ddf54b8740394ead7005126825e1c79617ed771f9f6d20b4aa56782f
c3a5251642fbfcf5a1dc6c91b32e4f37dde5b9bbf50ba3242e780a21c5af3989
600637f424dbcfab99e0aec4397930df9f21f4eff880de8410e68098323d29fd
a9507f96c8730e7dc9b504087c89dece5042e01d931a6f9e0ca72fcdd7d8e57f
16ee4abb23abf28cdce01413fd9bf01ff5e674d8bc97ddf09114b183ac14d2ac
ce8e9241ede7f74ce6c4f21acd5617a96e15be0cae0d543934ab297bfe1f7666
eb16465b4f8f876aa85001a6333f1175c2a20a1642d49f3179b451d26ae7d541
6ca195ee197105a20daf7179d72624a55aff9b4efeff7a1dfc207d8da6135de9
1246d8e86ffd2235bbd9cc9d8c32c3fbd19ede23d8f9f2ad8e58c19ef971c0d2
c3449091b487f77cac165db9c69fbb430bf61b1787846f351cc15b46df83ee69
```

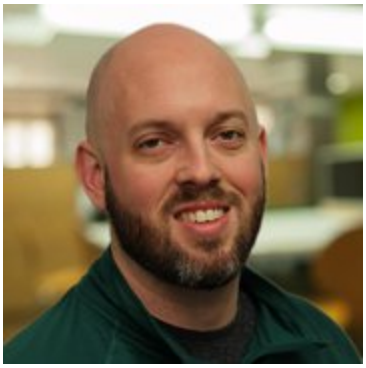
Identified Network Communications (Deduplicated)

Table 3: List of Identified Network Communications

```
IP Address      Port
187.234.55.76   8080
4.34.48.68      18443
78.38.244.10    25
160.211.55.3    5555
160.0.70.36     5555
87.98.212.8     80
143.248.95.119  55555
69.196.157.195  80
8.8.8.8 443
143.248.222.15  55555
121.200.62.194  8443
80.255.10.235   80
78.47.114.3     443
192.9.226.2     5555

172.22.2.16     50001
127.0.0.1       5555
192.168.1.49    25
192.168.50.8    5555
192.168.56.1    5555
192.168.100.153 2222
192.168.100.20  9999
10.201.56.136   25
192.168.1.49    25
192.168.231.1   5555
No IP Address   5555
192.168.19.134  1337
172.16.214.1    5555
172.24.8.41     25
192.168.100.45  No port
```

WRITTEN BY



MORE BY THE AUTHORS

Snake Ransomware Analysis Updates

March 23, 2020 • By Kyle O'Meara

High-Level Technique for Insider Threat Program's Data Source Selection

May 30, 2019 • By Robert M. Ditmore , CERT Insider Threat Center

A New Scientifically Supported Best Practice That Can Enhance Every Insider Threat Program!

April 9, 2019 • By Michael C. Theis , CERT Insider Threat Center

Insider Threat Incident Analysis by Sector (Part 1 of 9)

October 11, 2018 • By Randy Trzeciak , CERT Insider Threat Center

Substance Use and Abuse: Potential Insider Threat Implications for Organizations

April 12, 2018 • By Tracy Cassidy , CERT Insider Threat Center

MORE IN CERT/CC VULNERABILITIES

The Latest Work from the SEI: Coordinated Vulnerability Disclosure, Cybersecurity Research, Cyber Risk and Resilience, and the Importance of Fostering Diversity in Software Engineering

September 6, 2021 • By Douglas C. Schmidt

Vulnerabilities: Everybody's Got One!

June 16, 2021 • By Leigh Metcalf

CERT/CC Comments on Standards and Guidelines to Enhance Software Supply Chain Security

June 1, 2021 • By Jonathan Spring

Cat and Mouse in the Age of .NET

November 19, 2020 • By Brandon Marzik

Adversarial ML Threat Matrix: Adversarial Tactics, Techniques, and Common Knowledge of Machine Learning

October 22, 2020 • By Jonathan Spring