Detecting Cobalt Strike with memory signatures

elastic.co/blog/detecting-cobalt-strike-with-memory-signatures

March 16, 2021

At Elastic Security, we approach the challenge of threat detection with various methods. Traditionally, we have focused on machine learning models and behaviors. These two methods are powerful because they can detect never-before-seen malware. Historically, we’ve felt that signatures are too easily evaded, but we also recognize that ease of evasion is only one of many factors to consider. Performance and false positive rates are also critical in measuring a detection technique's effectiveness.

Signatures, while unable to detect unknown malware, have false positive rates that approach zero and have associated labels that help prioritize alerts. For example, an alert for TrickBot or REvil Ransomware requires more immediate action than a potentially unwanted adware variant. Even if we could hypothetically catch only half of known malware with signatures, that is still a huge win when layered with other protections, considering the other benefits. Realistically we can do even better.

One roadblock to creating signatures that provide long-term value is the widespread use of packers and throw-away malware loaders. These components rapidly evolve to evade signature detection; however, the final malware payload is eventually decrypted and executed in memory.

To step around the issue of packers and loaders, we can focus signature detection strategies on in-memory content. This effectively extends the shelf life of the signature from days to months. In this post, we will use Cobalt Strike as an example for leveraging in-memory signatures.

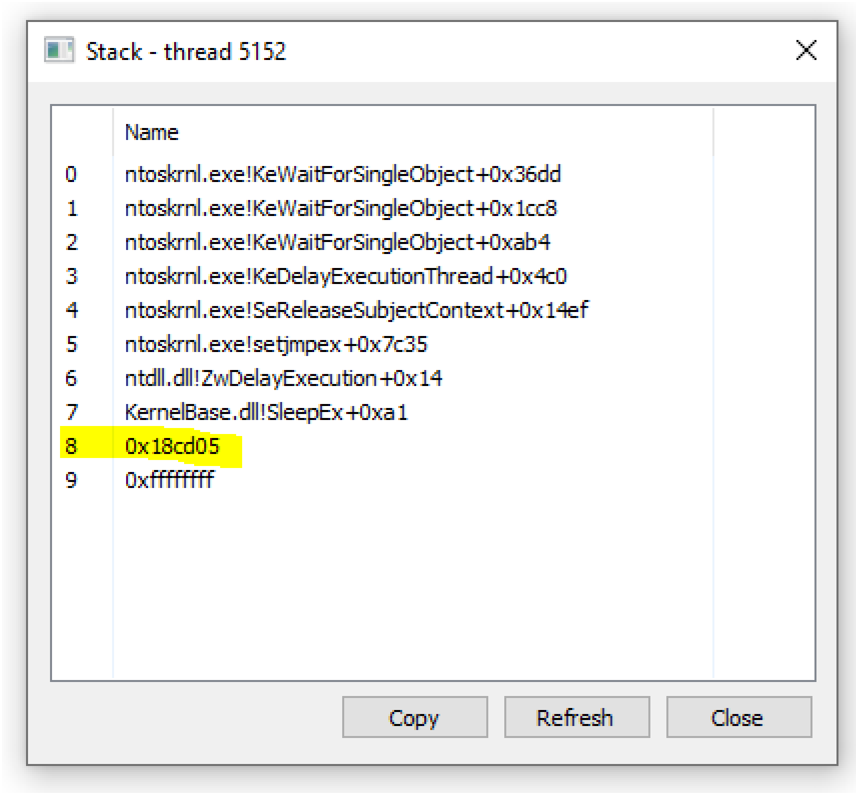
Signaturing Cobalt Strike

Cobalt Strike is a popular framework for conducting red team operations and adversary simulation. Presumably due to its ease of use, stability, and stealth features, it is also a favorite tool for bad actors with even more nefarious intentions. There have been various techniques for detecting Beacon, Cobalt Strike's endpoint payload. This includes looking for unbacked threads, and, more recently, built-in named pipes. However, due to the level of configurability in Beacon, there are usually ways to evade public detection strategies. Here we will attempt to use memory signatures as an alternative detection strategy.

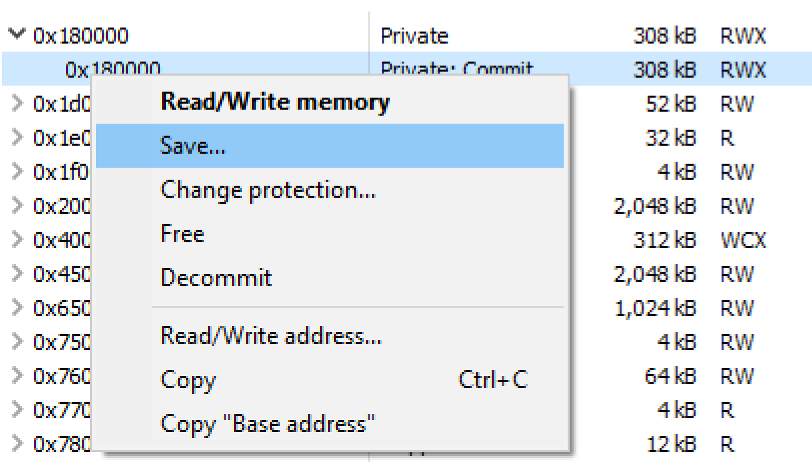
Beacon is typically reflectively loaded into memory and never touches disk in a directly signaturable form. Further, Beacon can be configured with a variety of in-memory obfuscation options to hide its payload. For example, the obfuscate-and-sleep option attempts to mask portions of the Beacon payload between callbacks to specifically evade signature-based memory scans. We will need to consider this option when developing signatures, but it is still easy to signature Beacon even with these advanced stealth features.

Diving in

We will start by obtaining a handful of Beacon payloads with the **sleep_mask** option enabled and disabled with the most recent releases (hashes in reference section). Starting with a sample with **sleep_mask** disabled, after detonation we can locate Beacon in memory with Process Hacker by looking for a thread which calls SleepEx from an unbacked region:



From there, we can save the associated memory region to disk for analysis:



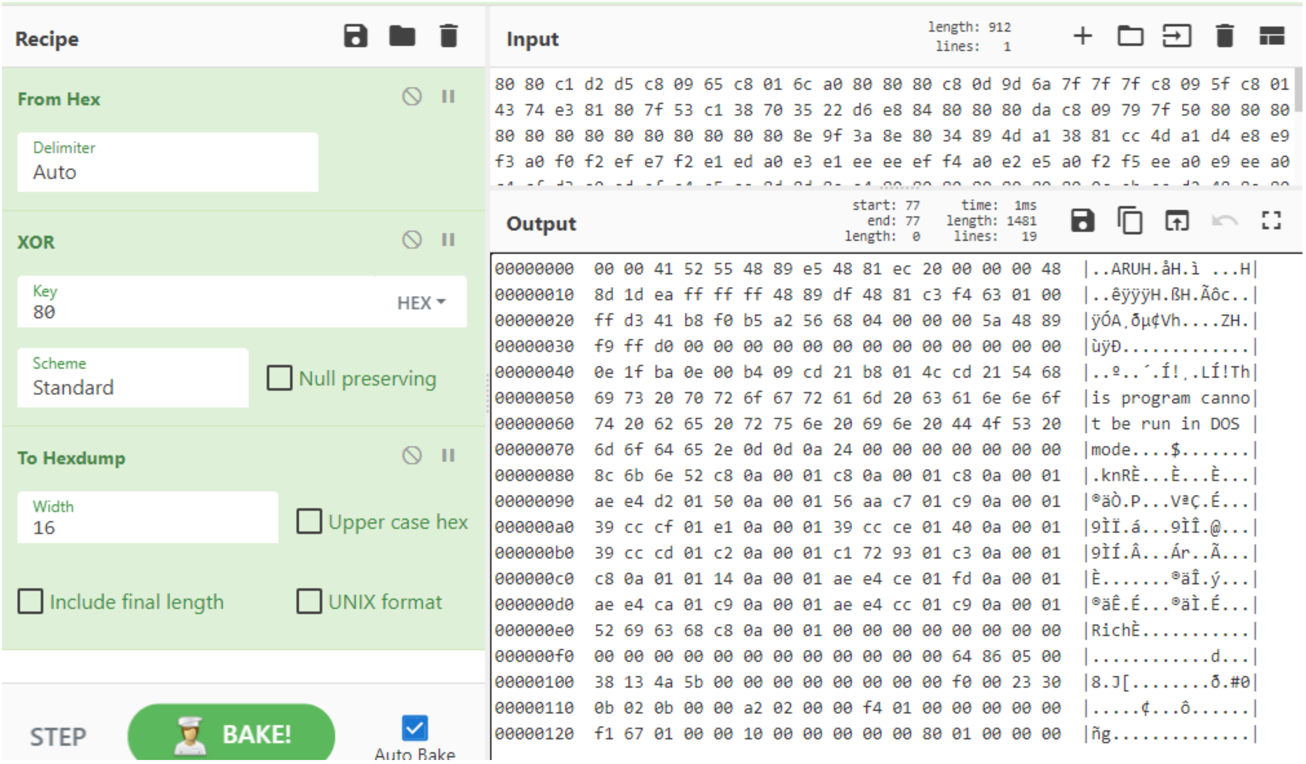
The easiest win would be to pick a few unique strings from this region and use those as our signature. To demonstrate, will will be writing signatures with yara, an industry standard tool for this purpose:

```
rule cobaltstrike_beacon_strings
{
meta:
    author = "Elastic"
    description = "Identifies strings used in Cobalt Strike Beacon DLL."
strings:
    $a = "%02d/%02d/%02d %02d:%02d:%02d"
    $b = "Started service %s on %s"
    $c = "%s as %s\\%s: %d"
condition:
    2 of them
}
```

This would give us a good base of coverage, but we can do better by looking at the samples with **sleep_mask** enabled. If we look in memory where the MZ/PE header would normally be found, we now see it is obfuscated:

```
00000000 80 80 c1 d2 d5 c8 09 65 c8 01 6c a0 80 80 80 c8 ........e..l.....
00000010 0d 9d 6a 7f 7f 7f c8 09 5f c8 01 43 74 e3 81 80 ..j....._..Ct...
00000020 7f 53 c1 38 70 35 22 d6 e8 84 80 80 80 da c8 09 .S.8p5".........
00000030 79 7f 50 80 80 80 80 80 80 80 80 80 80 80 80 80 y.P.............
00000040 8e 9f 3a 8e 80 34 89 4d a1 38 81 cc 4d a1 d4 e8 ..:..4.M.8..M...
00000050 e9 f3 a0 f0 f2 ef e7 f2 e1 ed a0 e3 e1 ee ee ef ................
00000060 f4 a0 e2 e5 a0 f2 f5 ee a0 e9 ee a0 c4 cf d3 a0 ................
00000070 ed ef e4 e5 ae 8d 8d 8a a4 80 80 80 80 80 80 80 ................
00000080 0c eb ee d2 48 8a 80 81 48 8a 80 81 48 8a 80 81 ....H...H...H...
00000090 2e 64 52 81 d0 8a 80 81 d6 2a 47 81 49 8a 80 81 .dR......*G.I...
000000a0 b9 4c 4f 81 61 8a 80 81 b9 4c 4e 81 c0 8a 80 81 .LO.a....LN.....
000000b0 b9 4c 4d 81 42 8a 80 81 41 f2 13 81 43 8a 80 81 .LM.B...A...C...
000000c0 48 8a 81 81 94 8a 80 81 2e 64 4e 81 7d 8a 80 81 H........dN.}...
000000d0 2e 64 4a 81 49 8a 80 81 2e 64 4c 81 49 8a 80 81 .dJ.I....dL.I...
000000e0 d2 e9 e3 e8 48 8a 80 81 80 80 80 80 80 80 80 80 ....H...........
000000f0 80 80 80 80 80 80 80 80 80 80 80 80 e4 06 85 80 ................
00000100 b8 93 ca db 80 80 80 80 80 80 80 80 70 80 a3 b0 ............p...
00000110 8b 82 8b 80 80 22 82 80 80 74 81 80 80 80 80 80 ....."...t......
00000120 71 e7 81 80 80 90 80 80 80 80 80 00 81 80 80 80 q...............
00000130 80 90 80 80 80 82 80 80 85 80 82 80 80 80 80 80 ................
```

Quickly looking at this, we can see a lot of repeated bytes (0x80 in this case) where we would actually expect null bytes. This can be an indication that Beacon is using a simple one-byte XOR obfuscation. To confirm, we can use CyberChef:



As you can see, the "This program cannot be run in DOS mode" string appears after decoding, confirming our theory. Because a single byte XOR is one of the oldest tricks in the book, yara actually supports native detection with the **xor** modifier:

```
rule cobaltstrike_beacon_xor_strings
{
meta:
    author = "Elastic"
    description = "Identifies XOR'd strings used in Cobalt Strike Beacon DLL."
strings:
    $a = "%02d/%02d/%02d %02d:%02d:%02d" xor(0x01-0xff)
    $b = "Started service %s on %s" xor(0x01-0xff)
    $c = "%s as %s\\%s: %d" xor(0x01-0xff)
condition:
    2 of them
}
```

We can confirm detection for our yara rules thus far by providing a PID while scanning:

```
C:\Users\user\Desktop>yara64.exe --print-strings beacon.yar 2308
cobaltstrike_beacon_strings 2308
0x1cadc4:$b: %02d/%02d/%02d %02d:%02d:%02d
0x1cadf0:$b: %02d/%02d/%02d %02d:%02d:%02d
0x94c7c4:$b: %02d/%02d/%02d %02d:%02d:%02d
0x94c7f0:$b: %02d/%02d/%02d %02d:%02d:%02d
0x1cad58:$c: Started service %s on %s
0x94c758:$c: Started service %s on %s

C:\Users\user\Desktop>yara64.exe --print-strings beacon.yar 5180
cobaltstrike_beacon_xor_strings 5180
0x1b004e:$a: Sont'wuh`ufj'dfiihs'eb'uri'ni'CHT'jhcb
0x1dc7c4:$b: "75c("75c("75c'"75c="75c="75c
0x1dc7f0:$b: "75c("75c("75c'"75c="75c="75c
0x1dc758:$c: Tsfusbc'tbuqndb'"t'hi'"t

C:\Users\user\Desktop>
```

However, we are not quite done. After testing this signature on a sample with the latest version of Beacon (4.2 as of this writing), the obfuscation routine has been improved. The routine can be located by following the call stack as shown earlier. It now uses a 13-byte XOR key as shown in the following IDA Pro snippet:

```
do
{
  j = startOffset;
  modulo = (unsigned int)startOffset / 13;
  LODWORD(startOffset) = startOffset + 1;
  *(_BYTE *)(*stuff + i++) ^= *((_BYTE *)stuff + j - 13 * modulo + 16);
}
while ( (unsigned int)startOffset < endOffset );
```

Fortunately, Beacon’s obfuscate-and-sleep option only obfuscates strings and data, leaving the entire code section ripe for signaturing. There is the question of which function in the code section we should develop a signature for, but that is worth its own blog post. For now, we can just create a signature on the deobfuscation routine, which should work well:

```
rule cobaltstrike_beacon_4_2_decrypt
{
meta:
    author = "Elastic"
    description = "Identifies deobfuscation routine used in Cobalt Strike Beacon DLL
version 4.2."
strings:
    $a_x64 = {4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85 C9 75 05 45 85 DB
74 33 45 3B CB 73 E6 49 8B F9 4C 8B 03}
    $a_x86 = {8B 46 04 8B 08 8B 50 04 83 C0 08 89 55 08 89 45 0C 85 C9 75 04 85 D2
74 23 3B CA 73 E6 8B 06 8D 3C 08 33 D2}
condition:
     any of them
}
```

We can validate that we can detect Beacon even while it is in its stealthy sleep state (both 32- and 64-bit variants):

```
C:\Users\user\Desktop>yara64.exe --print-strings beacon.yar 8012
cobaltstrike_beacon_4_2_decrypt 8012
0x6d32ae:$a_x86: 8B 46 04 8B 08 8B 50 04 83 C0 08 89 55 08 89 45 0C 85 C9 75 04 85 D2 74 23 3B CA 73 E6 8B 06 8D ...
0x6f4d56:$a_x86: 8B 46 04 8B 08 8B 50 04 83 C0 08 89 55 08 89 45 0C 85 C9 75 04 85 D2 74 23 3B CA 73 E6 8B 06 8D ...

C:\Users\user\Desktop>yara64.exe --print-strings beacon.yar 8936
cobaltstrike_beacon_4_2_decrypt 8936
0x18fa49:$a_x64: 4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85 C9 75 05 45 85 DB 74 33 45 3B CB 73 E6 49 8B ...
0x1aa73d:$a_x64: 4C 8B 53 08 45 8B 0A 45 8B 5A 04 4D 8D 52 08 45 85 C9 75 05 45 85 DB 74 33 45 3B CB 73 E6 49 8B ...
```

To build this into a more robust detection, we could regularly scan all processes on the system (or entire enterprise). This could be done with the following powershell one-liner:

```
powershell -command "Get-Process | ForEach-Object {c:\yara64.exe my_rules.yar
$_.ID}"
```

Wrapping up

Signature-based detection, while often looked down upon, is a valuable detection strategy — especially when we consider in-memory scanning. With only a handful of signatures, we can detect Cobalt Strike regardless of configuration or stealth features enabled with an effective false positive rate of zero.

Reference hashes

```
7d2c09a06d731a56bca7af2f5d3badef53624f025d77ababe6a14be28540a17a
277c2a0a18d7dc04993b6dc7ce873a086ab267391a9acbbc4a140e9c4658372a
A0788b85266fedd64dab834cb605a31b81fd11a3439dc3a6370bb34e512220e2
2db56e74f43b1a826beff9b577933135791ee44d8e66fa111b9b2af32948235c
3d65d80b1eb8626cf327c046db0c20ba4ed1b588b8c2f1286bc09b8f4da204f2
```

Learn more about Elastic Security

Familiarize yourself (if you haven't already) with the powerful protection, detection, and response capabilities of Elastic Agent. Start your free 14-day trial (no credit card required) or download our products, free, for your on-prem deployment. And take advantage of our Quick Start training to set you up for success.