使用内存签名检测 Cobalt Strike | Elastic Blog

基于签名的检测 — 尤其是内存中扫描 — 是一种极具价值的威胁检测策略。在这篇博文中,了解如何在有效误报率为零的情况下检测 Cobalt Strike, 而不管它的配置或隐蔽功能如何。

在 Elastic 安全中,我们会借助多种方法来应对威胁检测方面的挑战。在传统上,我们专注于研究 Machine Learning 模型和行为。这两种方法非常强大,因为它们可以检测到前所未见的恶意软件。从历史数据看,我们认为签名太容易被规避,但我们也认识到,规避的难易只是需要考虑的众多因素之一。性能和误报率也是衡量检测技术有效性的关键。

签名虽然无法检测到未知的恶意软件,但其误报率接近于零,并且有相关联的标签,这对确定告警优先级会有所帮助。例如,与不受欢迎的潜在广告软件变体相比,针对 TrickBot 或 REvil 勒索软件的告警更需要立即采取行动。即使我们假设通过签名只能捕获一半的已知恶意软件,但考虑到其他方面的益处,再配以其他保护措施,这仍然是一个巨大的胜利。实际上,我们还可以做得更好。

在创建具有长期价值的签名方面,一大障碍就是加壳程序和一次性恶意软件加载程序的广泛使用。这些组件会迅速进化以规避签名检测;但是,最终的恶意软件有效负载终究会在内存中解密并执行。

为了解决加壳程序和加载程序的问题,我们可以将签名检测策略集中在内存中的内容上。这会有效地将签名的有效期限从几天延长到几个月。在这篇博文中,我们将以 Cobalt Strike 为例,介绍如何利用内存中签名。

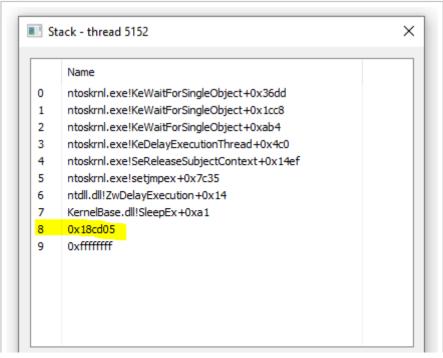
生成 Cobalt Strike 签名

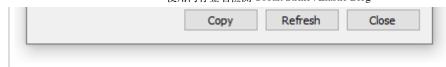
Cobalt Strike 是一种流行的框架,用于进行红队操作和对手模拟。大概是由于它的易用性、稳定性和隐蔽功能,它也成为了有着更多 不轨 意图的 不法者 最爱的工具。现在有多种技术可以检测 Cobalt Strike 的终端有效负载 Beacon。这包括查找 无支持线程 ,以及最近的内置 命名管道 。然而,由于 Beacon 具有很高的 可配置性 ,因此通常有一些方法可以避开公共检测策略。下面,我们将尝试使用内存签名作为替代检测策略。

Beacon 通常是 反射加载 到内存中,并且从不以可直接签名的形式接触磁盘。此外,Beacon 可以配置各种内存中混淆选项以隐藏其有效负载。例如, obfuscate—and—sleep 选项会试图在回调之间屏蔽部分 Beacon 有效负载,以专门避开基于签名的内存扫描。我们在开发签名时需要考虑这个选项,但即使有这些先进的隐蔽功能,对 Beacon 进行签名仍然很容易。

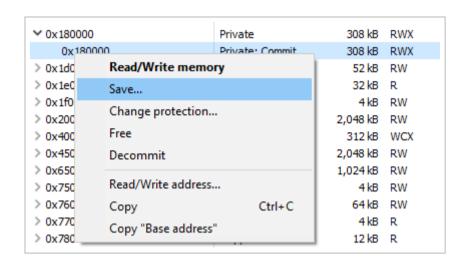
进一步了解

我们将首先获取一些 Beacon 有效负载,并在最新版本中启用和禁用 **sleep_mask** 选项(参考部分中的哈希)。从禁用 **sleep_mask** 的示例开始,在引爆后,我们可以通过 Process Hacker 查找从无支持区域调用 SleepEx 的线程来定位内存中的 Beacon:





然后, 我们可以将关联的内存区域保存到磁盘以供分析:



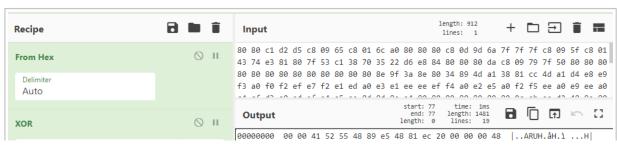
最简单的做法是,从这个区域挑选一些独特的字符串并将它们用作我们的签名。出于演示目的,我们将使用 yara 编写签名,这是用于此目的的行业标准工具:

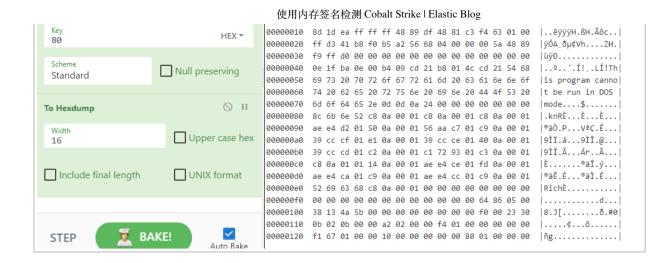
```
rule cobaltstrike_beacon_strings
{
  meta:
    author = "Elastic"
    description = "Identifies strings used in Cobalt Strike Beacon DLL."
  strings:
    $a = "%02d/%02d/%02d %02d:%02d"
    $b = "Started service %s on %s"
    $c = "%s as %s\\%s: %d"
  condition:
    2 of them
}
```

这将为我们提供良好的保障基础,但我们可以通过参照启用了 **sleep_mask** 的示例来做得更好。如果我们看一下内存中通常可以找到 MZ/PE 标头的位置,现在会看到它被模糊化了:

```
00000000 80 80 cl 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 cl 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 9.P......
00000040 8e 9f 3a 8e 80 34 89 4d al 38 8l cc 4d al d4 e8 .....4.M.8..M...
00000050 e9 f3 a0 f0 f2 ef e7 f2 el ed a0 e3 el 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 Oc 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 8l 6l 8a 80 8l b9 4c 4e 8l c0 8a 80 8l .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 ....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 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 .....
```

快速看一下这张图,我们可以看到许多重复的字节(在本例中为 0x80),而我们实际期望看到的是空字节。这可能表明 Beacon 正在使用简单的单字节 XOR 混淆。我们可以使用 CyberChef 来进一步确认一下:





大家可以看到,解码后出现"This program cannot be run in DOS mode"(此程序无法在 DOS 模式下运行)字符串,证实了我们的猜测。因为单字节 XOR 是最老套的技巧之一,yara 实际上支持使用 **xor** 修饰符进行原生检测:

```
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
}
```

到目前为止,我们可以通过在扫描时提供 PID 来确认基于我们 yara 规则的检测:

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

然而,这还没有结束。在使用最新版 Beacon(撰写本文时为 4.2)的示例上测试这个签名后,混淆例程已得到改进。这一例程可以按照前面所示的调用堆栈来找到。它现在使用 13 字节的 XOR 密钥,如以下 IDA Pro 代码段所示:

```
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 );
</pre>
```

幸运的是, Beacon 的 obfuscate-and-sleep 选项只会混淆字符串和数据, 让整个代码部分都可以进行签名。不过, 我们应该为代码部分中的哪个函数开发签名确实也是个问题, 但这需要另写一篇博文详述。现在, 我们可以只在反混淆例程上创建一个签名, 它就应该可以运作得很好:

```
rule cobaltstrike_beacon_4_2_decrypt
{
meta:
    author = "Elastic"
    description = "Identifies deobfuscation routine used in Cobalt Strike Beacon DLL version 4.2."
strings:
### 150 PR 150 PR
```

```
$\( \begin{align*} \b
```

我们可以验证,即使 Beacon 处于隐蔽休眠状态(32 位和 64 位变体),我们也可以检测到它:

```
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 ...
```

为了将其构建为更强大的检测体系,我们可以定期扫描系统上(或整个企业中)的所有进程。这可以通过以下 powershell 单行指令来完成:

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

总结

基于签名的检测虽然经常被人们所忽视,但却是一种极其重要的检测策略 — 尤其是在我们考虑内存中扫描时。只需少量签名,我们就可以在有效误报率为零的情况下检测 Cobalt Strike,而不管它的配置或隐蔽功能如何。

参考哈希

7d2c09a06d731a56bca7af2f5d3badef53624f025d77ababe6a14be28540a17a 277c2a0a18d7dc04993b6dc7ce873a086ab267391a9acbbc4a140e9c4658372a A0788b85266fedd64dab834cb605a31b81fd11a3439dc3a6370bb34e512220e2 2db56e74f43b1a826beff9b577933135791ee44d8e66fa111b9b2af32948235c 3d65d80b1eb8626cf327c046db0c20ba4ed1b588b8c2f1286bc09b8f4da204f2