

2017 年 10 月 12 日，星期四

# 反汇编程序和运行时分析

作者: *Paul Rascagneres*。

## 引言

在我们之前介绍 CCleaner 64 位第 2 阶段的博文中，我们对攻击者修改作为“Symantec Endpoint”一部分的合法可执行文件进行了说明。相关文件的名称是 EFACli64.dll。这项修改在编译程序添加的运行时代码（更准确地说是在 `__security_init_cookie()` 函数）中执行。攻击者修改了最后一条指令以跳转到恶意代码。常用的 IDA Pro 反汇编程序无法显示这项修改，这一点我们稍后会在本文中说明。最后，我们将提供一种用于识别此类修改的方法，并说明这种方法的局限性。

## IDA PRO 与修改的运行时代码

在对经过修改的第 2 个可执行文件进行分析时，我们发现 IDA Pro 在使用图形视图时难以正确显示修补过的运行时代码程序集：

```
; Attributes: library function
; void __cdecl _security_init_cookie()
_security_init_cookie proc near

SystemTimeAsFileTime= _FILETIME ptr 8
PerformanceCount= LARGE_INTEGER ptr 10h
arg_10= qword ptr 18h

mov     [rsp+arg_10], rbx
push   rdi
sub     rsp, 20h
mov     rax, cs:qword_69393188
and     qword ptr [rsp+28h+SystemTimeAsFileTime.dwLowDateTime], 0
mov     rdi, 28992DDFA232h
cmp     rax, rdi
jz      short loc_6938F652
```

```
not     rax
mov     cs:qword_69393190, rax
jmp     short loc_6938F6C8
```

```
loc_6938F652:                ; lpSystemTimeAsFileTime
lea     rcx, [rsp+28h+SystemTimeAsFileTime]
call   cs:GetSystemTimeAsFileTime
mov     rbx, qword ptr [rsp+28h+SystemTimeAsFileTime.dwLowDateTime]
call   cs:GetCurrentProcessId
mov     r11d, eax
xor     rbx, r11
call   cs:GetCurrentThreadId
mov     r11d, eax
xor     rbx, r11
call   cs:GetTickCount
lea     rcx, [rsp+28h+PerformanceCount] ; lpPerformanceCount
mov     r11d, eax
xor     rbx, r11
call   cs:QueryPerformanceCounter
mov     r11, qword ptr [rsp+28h+PerformanceCount]
xor     r11, rbx
mov     rax, 0FFFFFFFFFh
and     r11, rax
mov     rax, 28992DDFA233h
cmp     r11, rdi
cmovz  r11, rax
mov     cs:qword_69393188, r11
not     r11
mov     cs:qword_69393190, r11
```

```
loc_6938F6C8:
mov     rbx, [rsp+28h+arg_10]
add     rsp, 20h
pop     rdi
_security_init_cookie endp
```

正如我们所看到的那样，最后一条指令为“pop rdi”。如果我们切换为文本视图，可以立即看到最后一条指令实际上是 JMP（跳转至恶意代码）：

```
.text:000000006938F652 loc_6938F652:                ; CODE XREF: __security_init_cookie+24fj
.text:000000006938F652          lea   rcx, [rsp+28h+SystemTimeAsFileTime] ; lpSystemTimeAsFileTime
.text:000000006938F657          call  cs:GetSystemTimeAsFileTime
.text:000000006938F65D          mov   rbx, qword ptr [rsp+28h+SystemTimeAsFileTime.dwLowDateTime]
.text:000000006938F662          call  cs:GetCurrentProcessId
.text:000000006938F668          mov   r11d, eax
.text:000000006938F66B          xor   rbx, r11
.text:000000006938F66E          call  cs:GetCurrentThreadId
.text:000000006938F674          mov   r11d, eax
.text:000000006938F677          xor   rbx, r11
.text:000000006938F67A          call  cs:GetTickCount
.text:000000006938F680          lea   rcx, [rsp+28h+PerformanceCount] ; lpPerformanceCount
.text:000000006938F685          mov   r11d, eax
.text:000000006938F688          xor   rbx, r11
.text:000000006938F68B          call  cs:QueryPerformanceCounter
.text:000000006938F691          mov   r11, qword ptr [rsp+28h+PerformanceCount]
.text:000000006938F696          xor   r11, rbx
.text:000000006938F699          mov   rax, 0FFFFFFFFFh
.text:000000006938FA3          and   r11, rax
.text:000000006938FA6          mov   rax, 2B992DDFA233h
.text:000000006938FB0          cmp   r11, rdi
.text:000000006938FB3          cmovz r11, rax
.text:000000006938FB7          mov   cs:qword_69393188, r11
.text:000000006938FBE          not   r11
.text:000000006938FC1          mov   cs:qword_69393190, r11
.text:000000006938FC8          loc_6938FC8:                ; CODE XREF: __security_init_cookie+30fj
.text:000000006938FC8          mov   rbx, [rsp+28h+arg_10]
.text:000000006938FCD          add   rsp, 20h
.text:000000006938FD1          pop   rdi
.text:000000006938FD1          __security_init_cookie endp
.text:000000006938FD2          loc_6938FD2:                ; DATA XREF: .pdata:0000000069394E70lo
.text:000000006938FD2          jmp   TrojanJump
.text:000000006938FD2          ; -----
.text:000000006938FD7          db   0CCh ; i
.text:000000006938FD8          db   0CCh ; i
```

如果我们在开源反汇编程序 Radare2 中查看，可以确认相关函数其实是以 JMP 指令结束的：

```
[0x6938efb8]> pdf @ 0x6938f620
/ (fcn) sub.KERNEL32.dll_GetSystemTimeAsFileTime_620 198
sub.KERNEL32.dll_GetSystemTimeAsFileTime_620 ();
; CALL XREF from 0x6938efd4 (entry0)
0x6938f620 48895c2418 mov qword [rsp + 0x18], rbx
0x6938f625 57 push rdi
0x6938f626 4883ec20 sub rsp, 0x20
0x6938f62a 488b05573b00. mov rax, qword [0x69393188]; [0x69393188:8]=0x2b992ddfa232
0x6938f631 488364243000. and qword [rsp + 0x30], 0
0x6938f637 48bf32a2df2d. movabs rdi, 0x2b992ddfa232
0x6938f641 483bc7 cmp rax, rdi
;==< 0x6938f644 740c je 0x6938f652
0x6938f646 48f7d0 not rax
0x6938f649 488905403b00. mov qword [0x69393190], rax; [0x69393190:8]=0xffffd466d2205dcd
;===< 0x6938f650 eb76 jmp 0x6938f6c8
; JMP XREF from 0x6938f644 (sub.KERNEL32.dll_GetSystemTimeAsFileTime_620)
--> 0x6938f652 488d4c2430 lea rcx, [rsp + 0x30]; '0'; 48
0x6938f657 ff15cb19ffff call qword sym.imp.KERNEL32.dll_GetSystemTimeAsFileTime; [0x69381028:8]=0x126f0
reloc.KERNEL32.dll_GetSystemTimeAsFileTime_240
0x6938f65d 488b5c2430 mov rbx, qword [rsp + 0x30]; [0x30:8]=-1; '0'; 48
0x6938f662 ff15c819ffff call qword sym.imp.KERNEL32.dll_GetCurrentProcessId; [0x69381030:8]=0x126da reloc
c.KERNEL32.dll_GetCurrentProcessId_218
0x6938f668 448bd8 mov r11d, eax
0x6938f66b 4933db xor rbx, r11
0x6938f66e ff15c419ffff call qword sym.imp.KERNEL32.dll_GetCurrentThreadId; [0x69381038:8]=0x126c4 reloc
.KERNEL32.dll_GetCurrentThreadId_196
0x6938f674 448bd8 mov r11d, eax
0x6938f677 4933db xor rbx, r11
0x6938f67a ff15c019ffff call qword [sym.imp.KERNEL32.dll_GetTickCount]; [0x69381040:8]=0x126b4 reloc.KER
NEL32.dll_GetTickCount_180
0x6938f680 488d4c2438 lea rcx, [rsp + 0x38]; '8'; 56
0x6938f685 448bd8 mov r11d, eax
0x6938f688 4933db xor rbx, r11
0x6938f68b ff15b719ffff call qword sym.imp.KERNEL32.dll_QueryPerformanceCounter; [0x69381048:8]=0x1269a
reloc.KERNEL32.dll_QueryPerformanceCounter_154
0x6938f691 4c8b5c2438 mov r11, qword [rsp + 0x38]; [0x38:8]=-1; '8'; 56
0x6938f696 4c33db xor r11, rbx
0x6938f699 48b8ffffff. movabs rax, 0xffffffff; 281474976710655
0x6938f6a3 4c23d8 and r11, rax
0x6938f6a6 48b833a2df2d. movabs rax, 0x2b992ddfa233
0x6938f6b0 4c3bdf cmp r11, rdi
0x6938f6b3 4c0f44d8 cmov r11, rax
0x6938f6b7 4c891dca3a00. mov qword [0x69393188], r11; [0x69393188:8]=0x2b992ddfa232
0x6938f6be 49f7d3 not r11
0x6938f6c1 4c891dc83a00. mov qword [0x69393190], r11; [0x69393190:8]=0xffffd466d2205dcd
; JMP XREF from 0x6938f650 (sub.KERNEL32.dll_GetSystemTimeAsFileTime_620)
--> 0x6938f6c8 488b5c2440 mov rbx, qword [rsp + 0x40]; [0x40:8]=-1; '0'; 64
0x6938f6cd 4883c420 add rsp, 0x20
0x6938f6d1 5f pop rdi
;=< 0x6938f6d2 e98592ffff jmp 0x6938805c
[0x6938efb8]>
```

这促使我们开始思考：为什么 IDA Pro 不显示最后一条（也是最重要的）指令？

因为此软件不是开源软件，我们无法简单地检查代码。我们假设 IDA Pro 使用 pdata 部分来检索运行时函数的开始和结束部分。有关此假设的说明，请参见下一部分。

第二个问题是：攻击者是否故意使用这种伎俩来干扰分析？我们无法百分之百确定攻击者是特意使用这种伎俩在 IDA Pro 中隐藏跳转，还是侥幸实现了这一目的。

## PDATA 部分

有关 Microsoft 对 pdata 部分的介绍，可[点击此处](#)。本部分包含用于处理异常情况的一系列函数表项。在本文情境中，pdata 部分包含以下结构：

```
+0x000: 开始地址: 相应函数的 RVA。
+0x004: 结束地址: 函数结尾的 RVA。
+0x008: 展开信息: 展开信息的 RVA。
```

以下是有关 `__security_init_cookie()` 函数的数据：

```
+0x000: 0000F620 -> __security_init_cookie() 开头的 RVA。
+0x004: 0000F6D3 -> __security_init_cookie() 结尾的 RVA。
+0x008: 00010464
```

函数 (0xF6D3) 的结束地址位于跳转指令中间。通过修补函数的结束地址（使用 0xF6D7 替换 0xF6D3），IDA Pro 可以完美显示最后一条指令 (JMP)。正因如此，我们才可以假设 IDA Pro 真的使用 pdata 部分检索运行时函数。

## 用于检测陌生运行时代码的 PYTHON 脚本

基于以上解释，我们发布了一个[简单脚本](#)，用于根据 pdata 部分检测异常的运行时代码。所遵循的思路是根据 pdata 部分提供的地址扫描运行时代码并查找最后一条指令。如果指令不是预期指令 (`validInstructions = ["ret", "retn", "jmp", "int3"]` in our POC)，脚本会通知用户运行时函数可疑。以下是 CCleaner 第 2 阶段的输出：

```
user@lab:~$ ./pdata_check.py sample.exe
{ 'ASM': [ u'mov qword ptr [rsp + 0x18], rbx',
u'push rdi',
u'sub rsp, 0x20',
[...redacted...]
u'mov qword ptr [rip + 0x3ac8], r11',
u'mov rbx, qword ptr [rsp + 0x40]',
u'add rsp, 0x20',
u'pop rdi'],
'StartRaw': '0xea20',
'StartVA': '0x0000f620',
'StopRaw': '0xead3',
'StopVA': '0x0000f6d3',
'end': 'KO',
'lastASM': u'pop rdi'}
```

该脚本基于 pefile 和 capstone。输出显示 0x0000f620 (RVA) 处的运行时代码结束并带有一个“pop”指令，这存在异常。

## 限制

这种用于检测此特殊反汇编技术的方法并非灵丹妙药。我们使用大量 64 位二进制文件对其进行测试，而许多合法二进制文件的 pdata 部分并不完全相同。因此，这产生了许多误报。此外，攻击者可以修补 pdata 部分，添加额外的字节。在这种情况下，脚本不会发现任何异常，但 IDA Pro 会在图形视图中正确显示额外的操作码。这种方法仅作为恶意软件研究人员分析二进制文件的补充工具。

## 结论

对恶意软件研究人员而言，对受到攻击的合法二进制文件进行分析是一项大挑战。随着供应链攻击出现新的趋势，请求分析看似合法的二进制代码将变得越来越频繁。当合法应用受到攻击后，恶意负载可以隐藏在大量的合法代码中。在此特定案例中，分析师还面临着其他挑战，即无法完全信任 IDA Pro 的输出。我们不知道攻击者使用的伎俩属于蓄意还是侥幸，但结果都是一样的：分析师会很容易漏掉恶意代码。我们提供脚本来帮助分析师识别可疑的运行时函数，但像往常一样，这不是万灵药，只是添加到我们的工具包中的新工具。

发布者: [PAUL RASCAGNERES](#); 发布时间: [7:26 AM](#)

标签: [CCLEANER](#)、[逆向工程](#)

分享此文

